

WebSphere Application Server



Programming Guide for Edge Components

Version 7.0

WebSphere Application Server



Programming Guide for Edge Components

Version 7.0

First edition (June 2008)

This edition applies to:

WebSphere Application Server, Version 7.0

and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or through the IBM branch office serving your locality.

© **Copyright International Business Machines Corporation 2008. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures v

About this book vii

Who should read this book	vii
What you should already know	vii
Conventions and terminology used in this book	vii
Accessibility	viii
Related documents and Web sites	viii
How to send your comments	viii

Chapter 1. Overview of Edge components customization 1

Caching Proxy customization	1
Load Balancer customization	1
Locating sample code	2

Chapter 2. The Caching Proxy API 3

Overview of the Caching Proxy API	3
General procedure for writing API programs	3
Server process steps	4
Guidelines	7
Plug-in functions	8
Predefined functions and macros	15
Caching Proxy configuration directives for API steps	21
Compatibility with other APIs	24
Porting CGI programs	24

Caching Proxy API reference information	24
Variables	24
Authentication and authorization	33
Variant caching	36
API examples.	36

Chapter 3. Custom advisors 37

Advisors provide load-balancing information	37
Standard advisor function	37
Creating a custom advisor	38
Normal mode and replace mode	38
Advisor naming conventions	39
Compilation	39
Running a custom advisor	40
Required routines	40
Search order	40
Naming and file path	40
Custom advisor methods and function calls	41
Examples	44
Standard advisor	44
Side stream advisor.	45
Two port advisor	46
WebSphere Application Server advisor	51
Using data returned from advisors	53

Index 57

Figures

- | | | |
|----|---|----|
| 1. | Flowchart of steps in the proxy server process | 5 |
| 2. | HTTP_ and PROXY_ variable prefixes . . . | 25 |
| 3. | Proxy server authentication and authorization process | 34 |

About this book

This section describes the purpose, organization, and conventions of this document, the *WebSphere® Application Server Programming Guide for Edge Components*.

Who should read this book

This book describes the application programming interfaces (APIs) that are available for customizing the Edge components of WebSphere Application Server, Version 7.0. This information is intended for programmers who write plug-in applications and make other customizations. Network designers and system administrators also might be interested in this information as an indication of the types of customization that are possible.

What you should already know

Using the information in this book requires understanding of programming procedures using the Java™ or C programming languages, depending on the API that you plan to use. The methods and structures available in each exposed interface are documented, but you must know how to construct your own application, compile it for your system, and test it. Sample code is provided for some interfaces, but the samples are provided only as examples for constructing your own application.

Conventions and terminology used in this book

This documentation uses the following typographical and keying conventions.

Table 1. Conventions used in this book

Convention	Meaning
Bold	When referring to graphical user interfaces (GUIs), bold face indicates menus, menu items, labels, buttons, icons, and folders. It also can be used to emphasize command names that otherwise might be confused with the surrounding text.
Monospace	Indicates text you must enter at a command prompt. Monospace also indicates screen text, code examples, and file excerpts.
<i>Italics</i>	Indicates variable values that you must provide (for example, you supply the name of a file for <i>fileName</i>). Italics also indicates emphasis and the titles of books.
Ctrl-<i>x</i>	Where <i>x</i> is the name of a key, indicates a control-character sequence. For example, Ctrl-c means hold down the Ctrl key while you press the c key.
Return	Refers to the key labeled with the word Return, the word Enter, or the left arrow.
%	Represents the Linux and UNIX® command-shell prompt for a command that does not require root privileges.
#	Represents the Linux and UNIX command-shell prompt for a command that requires root privileges.
C:\	Represents the Windows command prompt.
Entering commands	When instructed to “enter” or “issue” a command, type the command and then press Return. For example, the instruction “Enter the ls command” means type ls at a command prompt and then press Return.
[]	Enclose optional items in syntax descriptions.

Table 1. Conventions used in this book (continued)

Convention	Meaning
{ }	Enclose lists from which you must choose an item in syntax descriptions.
	Separates items in a list of choices enclosed in { }(braces) in syntax descriptions.
...	Ellipses in syntax descriptions indicate that you can repeat the preceding item one or more times. Ellipses in examples indicate that information was omitted from the example for the sake of brevity.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. These are the major accessibility features in WebSphere Application Server, Version 7.0:

- You can use screen-reader software and a digital speech synthesizer to hear what is displayed on the screen. You can also use voice recognition software, such as IBM[®] ViaVoice[™], to enter data and to navigate the user interface.
- You can operate features by using the keyboard instead of the mouse.
- You can configure and administer Application Server features by using standard text editors or command-line interfaces instead of the graphical interfaces provided. For more information about the accessibility of particular features, refer to the documentation about those features.

Related documents and Web sites

- *Concepts, Planning, and Installation for Edge Components*, GC31-6918-00
- *Caching Proxy Administration Guide*, GC31-6920-00
- *Load Balancer Administration Guide*, GC31-6921-00
- IBM home Web site www.ibm.com/
- IBM WebSphere Application Server www.ibm.com/software/webservers/appserv/
- IBM WebSphere Application Server library Web site www.ibm.com/software/webservers/appserv/library.html
- IBM WebSphere Application Server support Web site www.ibm.com/software/webservers/appserv/support.html
- IBM WebSphere Application Server Information Center www.ibm.com/software/webservers/appserv/infocenter.html
- IBM WebSphere Application Server Edge Components Information Center www.ibm.com/software/webservers/appserv/ecinfocenter.html

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other documentation about the Edge components of WebSphere Application Server:

- Send your comments by e-mail to fsdoc@us.ibm.com. Be sure to include the name of the book, the part number of the book, the version of WebSphere Application Server, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Chapter 1. Overview of Edge components customization

This book discusses the application programming interfaces (APIs) provided for the Edge components of WebSphere Application Server. (The Edge components of WebSphere Application Server include Caching Proxy and Load Balancer.) Several interfaces are provided that enable administrators to customize their installations, to alter how the Edge components interact with each other, or to enable interaction with other software systems.

IMPORTANT: Caching Proxy is available on all Edge component installations, with the following exceptions:

- Caching Proxy is not available for Edge component installations that run on Itanium 2 or AMD Opteron 64-bit processors.
- Caching Proxy is not available for Edge component installations of Load Balancer for IPv4 and IPv6.

The APIs in this document address several categories.

Caching Proxy customization

The Caching Proxy has several interfaces written into its processing sequence where custom processing can be added or substituted for standard processing. Customizations that can be executed include altering or augmenting tasks like the following:

- Client authentication
- Request authorization
- Translating URLs to physical file paths
- Servicing requests
- Logging
- Responding to error conditions

Custom application programs, which are also known as Caching Proxy plug-ins, are called at predetermined points in the proxy server's processing sequence.

The Caching Proxy API has been used to implement certain system features. For example, the proxy server's LDAP support is implemented as a plug-in.

Chapter 2, "The Caching Proxy API," on page 3 describes the interface in detail and includes steps for configuring the proxy server to use plug-in programs.

Load Balancer customization

The Load Balancer can be customized by writing your own advisors. Advisors perform the actual load measurement on the servers. With a custom advisor, you can use a method that you provide and that is relevant to your system to measure the load. This is especially important if you have customized or proprietary Web server systems.

Chapter 3, "Custom advisors," on page 37 provides detailed information about writing and using custom advisors. It includes sample advisor code.

Locating sample code

Sample code for these APIs is included on the Edge Components CD-ROM, in the samples directory. Additional code samples are available from the WebSphere Application Server Web site, www.ibm.com/software/webservers/appserv/

Chapter 2. The Caching Proxy API

This section discusses the Caching Proxy application programming interface (API): what it is, why it is useful, and how it works.

IMPORTANT: Caching Proxy is available on all Edge component installations, with the following exceptions:

- Caching Proxy is not available for Edge component installations that run on Itanium 2 or AMD Opteron 64-bit processors.
- Caching Proxy is not available for Edge component installations of Load Balancer for IPv4 and IPv6.

Overview of the Caching Proxy API

The API is an interface to the Caching Proxy that enables you to extend the proxy server's base functions. You can write extensions, or plug-ins, to do customized processing, including the following examples:

- Enhancing the basic authentication routine, or replacing it with a site-specific process.
- Adding error-handling routines to track problems or alert for serious conditions.
- Detecting and tracking information that comes in from the requesting client, such as server referrals and user agent codes.

The Caching Proxy API provides the following benefits:

- **Efficiency**
 - The API is designed specifically for the threaded processing system used by the Caching Proxy.
- **Flexibility**
 - The API contains rich and versatile functions.
 - The API is platform independent and language neutral. It runs on all Caching Proxy platforms, and plug-in applications can be written in most of the programming languages supported by these platforms.
- **Ease of use**
 - Simple data types are passed by reference instead of by value (for example, `long *`, `char *`).
 - Each function has a fixed number of parameters.
 - C language bindings are included.
 - Plug-ins do not impact allocated memory; plug-in applications allocate and free memory independently of other Caching Proxy processes.

General procedure for writing API programs

Before writing your Caching Proxy plug-in programs, you need to understand how the proxy server works. The behavior of the proxy server can be divided into several distinct processing steps. For each of these steps, you can supply your own customized functions using the API. For example, do you want to do something after a client request is read but before performing any other processing? Or maybe you want to perform special routines during authentication and then again after the requested file is sent.

A library of predefined functions is provided with the API. Your plug-in programs can call the predefined API functions in order to interact with the proxy server process (for example, to manipulate requests, to read or write request headers, or to write to the proxy server's logs). These functions should not be confused with the plug-in functions that you write, which are called by the proxy server. The predefined functions are described in "Predefined functions and macros" on page 15.

You instruct the proxy server to call your plug-in functions at the appropriate steps by using the corresponding Caching Proxy API directives in your server configuration file. These directives are described in "Caching Proxy configuration directives for API steps" on page 21.

This document includes the following:

- A basic explanation of the Caching Proxy steps that can be customized (see "Server process steps")
- Guidelines for writing plug-ins (see "Guidelines" on page 7)
- Prototypes for the customized functions that you can write for each step performed by the server, and their return codes (see "Plug-in function prototypes" on page 8)
- Definitions of predefined functions and macros that you can call from within your plug-ins, and their return codes (see "Predefined functions and macros" on page 15)
- Caching Proxy API configuration directives (see "Caching Proxy configuration directives for API steps" on page 21)

You can use these components and procedures to write your own Caching Proxy plug-in programs.

Server process steps

The basic operation of the proxy server can be broken up into steps based on the type of processing that the server performs during that phase. Each step includes a juncture at which a specified part of your program can run. By adding API directives to your Caching Proxy configuration file (`ibmproxy.conf`), you indicate which of your plug-in functions you want to be called during a particular step. You can call several plug-in functions during a particular process step by including more than one directive for that step.

Some steps are part of the server request process. In other words, the proxy server executes these steps each time it processes a request. Other steps are performed independently of request processing; that is, the server executes these steps regardless of whether a request is being processed.

Your compiled program resides in a shared object, for example, a DLL or `.so` file, depending on your operating system. As the server proceeds through its request process steps, it calls the plug-in functions associated with each step until one of the functions indicates that it has handled the request. If you specify more than one plug-in function for a particular step, the functions are called in the order in which their directives appear in the configuration file.

If the request is not handled by a plug-in function (either you did not include a Caching Proxy API directive for that step, or your plug-in function for that step returned `HTTP_NOACTION`), the server performs its default action for that step.

The following list explains the purpose of each step pictured in Figure 1 on page 5. Note that not all steps are guaranteed to be called for a particular request.

Server Initialization

Performs initialization when the proxy server is started and before any client requests are accepted.

Midnight

Runs a plug-in at midnight, with no request context. This step is shown separately in the diagram because it is not part of the request process; in other words, its execution is independent of any request.

GC Advisor

Influences garbage collection decisions for files in the cache. This step is shown separately in the diagram because it is not part of the request process; in other words, its execution is independent of any request. Garbage collection is done when the cache size reaches the maximum value. (Information about configuring cache garbage collection is included in the *WebSphere Application Server Caching Proxy Administration Guide*.)

PreExit

Performs processing after a request is read but before anything else is done.

If this step returns an indication that the request was processed (HTTP_OK), the server bypasses the other steps in the request process and performs only the Transmogriker, Log, and PostExit steps.

Name Translation

Translates the virtual path (from a URL) to the physical path.

Authorization

Uses stored security tokens to check the physical path for protections, ACLs, and other access controls, and generates the WWW-Authenticate headers required for basic authentication. If you write your own plug-in function to replace this step, you must generate these headers yourself.

See “Authentication and authorization” on page 33 for more information.

Authentication

Decodes, verifies, and stores security tokens.

See “Authentication and authorization” on page 33 for more information.

Object Type

Locates the file system object indicated by the path.

Post Authorization

Performs processing after authorization and object location but before the request is satisfied.

If this step returns an indication that the request was processed (HTTP_OK), the server bypasses the other steps in the request process and performs only the Transmogriker, Log, and PostExit steps.

Service

Satisfies the request (by sending the file, running the CGI, etc.)

Proxy Advisor

Influences proxy and caching decisions.

Transmogriifier

Gives write access to the data portion of the response sent to the client.

Log Enables customized transaction logging.

Error Enables customized responses to error conditions.

PostExit

Cleans up resources allocated for request processing.

Server Termination

Performs clean-up processing when an orderly shutdown occurs.

Guidelines

- Write your program, following the syntax and guidelines provided for the server's plug-in functions. Give each of your plug-in functions a unique function name and call the server's predefined functions as needed.

On AIX® systems, you need an export file (for example, libmyapp.exp) that lists your plug-in functions, and you must link with the Caching Proxy API import file, libhttpdapi.exp.

On Linux, HP-UX, and Solaris systems, you must link with the libhttpdapi and libc libraries.

On Windows® systems, you need a module definition file (.def) that lists your plug-in functions, and you must link with HTTPDAPI.LIB.

Be sure to include HTAPI.h and to use the HTTPD_LINKAGE macro in your function definitions. This macro ensures that all the functions use the same calling conventions.

- The server runs in a multithreaded environment; therefore, your plug-ins must be thread safe. If your application is reentrant, performance does not decrease.
- Keep the actions in your plug-ins to a thread scope. Do not perform any actions at a process scope, for example, exiting, changing the user ID, or registering a signal handler.
- Do not use global variables, or, if you must use them, protect global variables with a mutual exclusion semaphore.
- Remember to set the Content-Type header if you are using the HTTPD_write() function to send data back to the client.
- Always check return codes and provide conditional processing where necessary.
- Compile and link your program, referring to the documentation for your compiler to build a shared object (for example, a DLL or .so file) as required for your operating system.

Use the following compile and link commands as a guideline.

- **AIX**, using IBM CSet++

- **Compile:**

```
cc_r -c -qdbxextra -qcpluscmt foo.c
```

- **Link:**

```
cc_r -bM:SRE -bnoentry -o libfoo.so foo.o -bI:libhttpdapi.exp  
-bE:foo.exp
```

(This command is shown on two lines for readability only.)

- **HP-UX**, using HP C/ANSI C Developer's Bundle and HP aC++ Compiler

- **Compile:**

```
cc -Ae -c +Z +DAportable
```

- **Link:**

- aCC +Z -mt -c +DAportable
- **Linux**, using the Gnu Compiler C (GCC) Version 3.2.X
 - **Compile:**

```
gcc -c foo.c
```
 - **Link:**

```
ld -G -Bsymbolic -o libfoo.so foo.o -lhttpdapi -lc
```
- **Solaris**, using Sun Workshop
 - **Compile:**

```
cc -mt -Bsymbolic -c foo.c
```
 - **Link:**

```
cc -mt -Bsymbolic -G -o libfoo.so foo.o -lhttpdapi -lc
```
- **Windows**, using Microsoft® Visual C++
 - **Compile:**

```
cl /c /MD /DWIN32 foo.c
```
 - **Link:**

```
link httpdapi.lib foo.obj /def:foo.def /out:foo.dll /dll
```

To specify exports, use one of these methods:

- Add `_declspec(dllexport)` definitions in the source.
- Specify `/EXPORT:entryname` on the LIB command line.
- Create a module definition file with an EXPORTS statement.
- Add Caching Proxy API directives to your configuration file to associate your program's plug-in functions with the appropriate steps. There is a separate directive for each step in the server request process. Stop and restart your server to make the new directives take effect.

Note: The Caching Proxy does not unload shared objects (DLL or .so files) even at restart. You must stop and then start the server in order to release shared objects.

- Test your program rigorously before using it in a production environment. Because the Caching Proxy is a threaded server, you must apply more rigorous testing than is necessary for a forking server. Errors in your program can cause the proxy server to fail because the proxy server calls your program directly, and they both run in the same process space.

Plug-in functions

Follow the syntax presented in "Plug-in function prototypes" to write your own program functions for the defined request processing steps.

Each of your functions must fill in the return code parameter with a value that indicates what action was taken:

- The code `HTTP_NOACTION` (value 0) means that no relevant action was taken. If this code is returned, the proxy server takes its default action for this step.
- One of the valid HTTP return codes indicates that the plug-in function handled the step. (See "HTTP return codes and values" on page 14 for a list of valid return codes.) If a valid HTTP return code is given, no other plug-in functions are called to handle that step of this request.

Plug-in function prototypes

The function prototypes for each Caching Proxy step show the format to use and explain the type of processing they can perform. Note that the function names are

not predefined. You must give your functions unique names, and you can choose your own naming conventions. For ease of association, this document uses names that relate to the server's processing steps.

In each of these plug-in functions, certain predefined API functions are valid. Some predefined functions are not valid for all steps. The following predefined API functions are valid when called from all of these plug-in functions:

- HTTPD_set
- HTTPD_extract
- httpd_setvar
- httpd_getvar
- HTTPD_log* functions

Additional valid or invalid API functions are noted in the function prototype descriptions.

The value of the *handle* parameter sent to your functions can be passed as the first argument to the predefined functions. Predefined API functions are described in "Predefined functions and macros" on page 15.

Server Initialization

```
void HTTPD_LINKAGE ServerInitFunction (  
    unsigned char *handle,  
    unsigned long *major_version,  
    unsigned long *minor_version,  
    long *return_code  
)
```

A function defined for this step is called once when your module is loaded during server initialization. It is your opportunity to perform initialization before any requests have been accepted.

Although all server initialization functions are called, a error return code from a function in this step causes the server to ignore all other functions configured in the same module as the function that returned the error code. (That is, any other functions contained in the same shared object as the function that returned the error are not called.)

The version parameters contain the proxy server's version number; these are supplied by the Caching Proxy.

PreExit

```
void HTTPD_LINKAGE PreExitFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request after the request has been read but before any processing has occurred. A plug-in at this step can be used to access the client's request before it is processed by the Caching Proxy.

Valid return codes for the preExit function are the following:

- 0 (HTTP_NOACTION)
- 200 (HTTP_OK)
- HTTP errors in the 4xx or 5xx series (for example, 404, HTTP_NOT_FOUND)

Other return codes must not be used.

If this function returns HTTP_OK, the proxy server assumes that the request has been handled. All subsequent request processing steps are bypassed, and only the response steps (Transmogifier, Log, and PostExit) are performed.

All predefined API functions are valid during this step.

Midnight

```
void HTTPD_LINKAGE MidnightFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step runs daily at midnight and contains no request context. For example, it can be used to invoke a child process to analyze logs. (Note that extensive processing during this step can interfere with logging.)

Authentication

```
void HTTPD_LINKAGE AuthenticationFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request based on the request's authentication scheme. This function can be used to customize verification of the security tokens that are sent with a request.

Name Translation

```
void HTTPD_LINKAGE NameTransFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request. A URL template can be specified in the configuration file directive if you want the plug-in function to be called only for requests that match the template. The Name Translation step occurs before the request is processed and provides a mechanism for mapping URLs to objects such as file names.

Authorization

```
void HTTPD_LINKAGE AuthorizationFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request. A URL template can be specified in the configuration file directive if you want the plug-in function to be called only for requests that match the template. The Authorization step occurs before the request is processed and can be used to verify that the identified object can be returned to the client. If you are doing basic authentication, you must generate the required WWW-Authenticate headers.

Object Type

```
void HTTPD_LINKAGE ObjTypeFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request. A URL template can be specified in the configuration file directive if you want the plug-in function to be called only for requests that match the template. The Object

Type step occurs before the request is processed and can be used to check whether the object exists, and to perform object typing.

PostAuthorization

```
void HTTPD_LINKAGE PostAuthFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called after the request has been authorized but before any processing has occurred. If this function returns HTTP_OK, the proxy server assumes that the request has been handled. All subsequent request steps are bypassed, and only the response steps (Transmogriifier, Log, and PostExit) are performed.

All server predefined functions are valid during this step.

Service

```
void HTTPD_LINKAGE ServiceFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request. A URL template can be specified in the configuration file directive if you want the plug-in function to be called only for requests that match the template. The Service step satisfies the request, if it was not satisfied in the PreExit or PostAuthorization steps.

All server predefined functions are valid during this step.

Refer to the Enable directive in the *WebSphere Application Server Caching Proxy Administration Guide* for information on configuring your Service function to be executed based on the HTTP method rather than on the URL.

Transmogriifier

The functions called in this process step can be used to filter response data as a stream. Four plug-in functions for this step are called in sequence, and each acts as a segment of pipe through which the data flows. That is, the *open*, *write*, *close*, and *error* functions that you provide are called, in that order, for each response. Each function processes the same data stream, in turn.

For this step, you must implement the following four functions. (Your function names do not need to match these names.)

- **Open**

```
void * HTTPD_LINKAGE openFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

The open function performs any initialization (such as buffer allocation) required to process the data for this stream. Any return code other than HTTP_OK causes this filter to abort (the write and close functions are not called). Your function can return a void pointer so that you can allocate space for a structure and have the pointer passed back to you in the *correlator* parameter of the subsequent functions.

- **Write**

```
void HTTPD_LINKAGE writeFunction (  
    unsigned char *handle,  
    unsigned char *data,    /* response data sent by the
```

```

                                origin server */
unsigned long *length, /* length of response data */
void *correlator,      /* pointer returned by the
                        'open' function */
long *return_code
)

```

The write function processes the data and can call the server's predefined HTTPD_write() function with the new or changed data. The plug-in must not attempt to free the buffer passed to it or expect the server to free the buffer it receives.

If you decide not to change the data during the scope of your write function, you still must call the HTTPD_write() function during the scope of either your open, write, or close function in order to pass the data for the response to the client. The *correlator* argument is the pointer to the data buffer that was returned in your *open* routine.

- **Close**

```

void HTTPD_LINKAGE closeFunction (
    unsigned char *handle,
    void *correlator,
    long *return_code
)

```

The close function performs any clean-up actions (such as flushing and freeing the correlator buffer) required to complete processing the data for this stream. The *correlator* argument is the pointer to the data buffer that was returned in your *open* routine.

- **Error**

```

void HTTPD_LINKAGE errorFunction (
    unsigned char *handle,
    void *correlator,
    long *return_code
)

```

The error function enables performance of clean-up actions, such as flushing or freeing the buffered data (or both) before an error page is sent. At this point, your open, write, and close functions are called to process the error page. The *correlator* argument is the pointer to the data buffer that was returned in your *open* routine.

Notes:

- When writing a plug-in for the Transmogriifier step, you must call HTTPD_open(), HTTPD_write(), and HTTPD_close() at some time during the scope of your open, write, and close functions. HTTPD_write() can be called only after the HTTPD_open() function has been called. The purpose of these predefined functions is to give control to the server so that the next function in the sequence can be invoked.
- Calling the HTTPD_* functions is necessary for your Transmogriifier API step and the server to perform correctly. For example, if HTTPD_open() and HTTPD_close() are not called, headers are not returned to the client.
- Be aware that undesirable effects can occur if data filtering applications are not properly selective in their filtering of data streams. It is possible that CGIs will not work if filtered incorrectly, GIF files will not be displayed, and other binary streams will not work as expected.
- It is not necessary for the plug-in to buffer content body. The Caching Proxy automatically determines the content length.

- It is desirable to call `HTTPD_open()` when you are ready to give control of the headers to the server. However, if you need to set a header later in the API program, you can wait until the write or close function to call the `HTTPD_open()` function.

Note: You must set any headers by using `HTTPD_set()` or `httpd_setvar()` before calling the `HTTPD_open()` function.

- The data stream does not include headers. Plug-ins must use set and extract functions to manipulate headers. The plug-in's *open* function is not invoked until all headers have been read.
- You can use multiple transmogriifier plug-ins, which are invoked in the order in which they appear in the configuration file.
- SSL tunneling is not passed through the transmogriifier plug-ins.

GC Advisor

```
void HTTPD_LINKAGE GCAdvisorFunction (
    unsigned char *handle,
    long *return_code
)
```

A function defined for this step is called for each file in the cache during garbage collection. This function enables you to influence which files are kept and which files are discarded. For more information, see the `GC_*` variables.

Proxy Advisor

```
void HTTPD_LINKAGE ProxyAdvisorFunction (
    unsigned char *handle,
    long *return_code
)
```

A function defined for this step is invoked during service of each proxy request. For example, it can be used to set the `USE_PROXY` variable.

Log

```
void HTTPD_LINKAGE LogFunction (
    unsigned char *handle,
    long *return_code
)
```

A function defined for this step is called for each request after the request has been processed and the communication to the client has been closed. A URL template can be specified in the configuration file directive if you want the plug-in function to be called only for requests that match the template. This function is called regardless of the success or failure of the request processing. If you do not want your log plug-in to override the default log mechanism, set your return code to `HTTP_NOACTION` instead of `HTTP_OK`.

Error

```
void HTTPD_LINKAGE ErrorFunction (
    unsigned char *handle,
    long *return_code
)
```

A function defined for this step is called for each request that fails. A URL template can be specified in the configuration file directive if you want the plug-in function to be called only for failed requests that match the template. The Error step provides an opportunity for you to customize the error response.

PostExit

```
void HTTPD_LINKAGE PostExitFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called for each request, regardless of the success or failure of the request. This step enables you to do clean-up tasks for any resources allocated by your plug-in to process the request.

Server Termination

```
void HTTPD_LINKAGE ServerTermFunction (  
    unsigned char *handle,  
    long *return_code  
)
```

A function defined for this step is called when an orderly shutdown of the server occurs. It enables you to clean up resources allocated during the Server Initialization step. Do not call any HTTP_* functions in this step (the results are unpredictable). If you have more than one Caching Proxy API directive in your configuration file for Server Termination, they will all be called.

Note: Because of a current limitation in Solaris code, the Server Termination plug-in step is not executed when the **ibmproxy -stop** command is used to shut down the Caching Proxy on Solaris platforms. Refer to the *WebSphere Application Server Caching Proxy Administration Guide* for information about starting and stopping the Caching Proxy.

HTTP return codes and values

These return codes follow the HTTP 1.1 specification, RFC 2616, published by the World Wide Web Consortium (www.w3.org/pub/WWW/Protocols/). Your plug-in functions must return one of these values.

Table 2. HTTP return codes for Caching Proxy API functions

Value	Return code
0	HTTP_NOACTION
100	HTTP_CONTINUE
101	HTTP_SWITCHING_PROTOCOLS
200	HTTP_OK
201	HTTP_CREATED
202	HTTP_ACCEPTED
203	HTTP_NON_AUTHORITATIVE
204	HTTP_NO_CONTENT
205	HTTP_RESET_CONTENT
206	HTTP_PARTIAL_CONTENT
300	HTTP_MULTIPLE_CHOICES
301	HTTP_MOVED_PERMANENTLY
302	HTTP_MOVED_TEMPORARILY
302	HTTP_FOUND
303	HTTP_SEE_OTHER
304	HTTP_NOT_MODIFIED

Table 2. HTTP return codes for Caching Proxy API functions (continued)

305	HTTP_USE_PROXY
307	HTTP_TEMPORARY_REDIRECT
400	HTTP_BAD_REQUEST
401	HTTP_UNAUTHORIZED
403	HTTP_FORBIDDEN
404	HTTP_NOT_FOUND
405	HTTP_METHOD_NOT_ALLOWED
406	HTTP_NOT_ACCEPTABLE
407	HTTP_PROXY_UNAUTHORIZED
408	HTTP_REQUEST_TIMEOUT
409	HTTP_CONFLICT
410	HTTP_GONE
411	HTTP_LENGTH_REQUIRED
412	HTTP_PRECONDITION_FAILED
413	HTTP_ENTITY_TOO_LARGE
414	HTTP_URI_TOO_LONG
415	HTTP_BAD_MEDIA_TYPE
416	HTTP_BAD_RANGE
417	HTTP_EXPECTATION_FAILED
500	HTTP_SERVER_ERROR
501	HTTP_NOT_IMPLEMENTED
502	HTTP_BAD_GATEWAY
503	HTTP_SERVICE_UNAVAILABLE
504	HTTP_GATEWAY_TIMEOUT
505	HTTP_BAD_VERSION

Predefined functions and macros

You can call the server's predefined functions and macros from your own plug-in functions. You must use their predefined names and follow the format described below. In the parameter descriptions, the letter *i* indicates an input parameter, the letter *o* indicates an output parameter, and *i/o* indicates that a parameter is used for both input and output.

Each of these functions returns one of the HTTPD return codes, depending on the success of the request. These codes are described in "Return codes from predefined functions and macros" on page 21.

Use the handle provided to your plug-in as the first parameter when calling these functions. Otherwise, the function returns an HTTPD_PARAMETER_ERROR error code. NULL is not accepted as a valid handle.

HTTPD_authenticate()

Authenticates a user ID or password, or both. Valid only in PreExit, Authentication, Authorization, and PostAuthorization steps.

```
void HTTPD_LINKAGE HTTPD_authenticate (
    unsigned char *handle,    /* i; handle */
    long *return_code        /* o; return code */
)
```

HTTPD_cacheable_url()

Returns whether the specified URL content is cacheable according to the Caching Proxy's standards.

```
void HTTPD_LINKAGE HTTPD_cacheable_url (
    unsigned char *handle,    /* i; handle */
    unsigned char *url,      /* i; URL to check */
    unsigned char *req_method, /* i; request method for the URL */
    long *retval             /* o; return code */
)
```

The return value HTTPD_SUCCESS indicates that the URL content is cacheable; HTTPD_FAILURE indicates the content is not cacheable. HTTPD_INTERNAL_ERROR also is a possible return code for this function.

HTTPD_close()

(Valid only in the Transmogriifier step.) Transfers control to the next *close* routine in the stream stack. Call this function from the Transmogriifier open, write, or close functions after any desired processing is done. This function notifies the proxy server that the response has been processed and the Transmogriifier step is complete.

```
void HTTPD_LINKAGE HTTPD_close (
    unsigned char *handle,    /* i; handle */
    long *return_code        /* o; return code */
)
```

HTTPD_exec()

Executes a script to satisfy this request. Valid in the PreExit, Service, PostAuthorization, and Error steps.

```
void HTTPD_LINKAGE HTTPD_exec (
    unsigned char *handle,    /* i; handle */
    unsigned char *name,      /* i; name of script to run */
    unsigned long *name_length, /* i; length of the name */
    long *return_code        /* o; return code */
)
```

HTTPD_extract()

Extracts the value of a variable associated with this request. The valid variables for the *name* parameter are the same as those used in the CGI. See "Variables" on page 24 for more information. Note that this function is valid in all steps; however, not all variables are valid in all steps.

```
void HTTPD_LINKAGE HTTPD_extract (
    unsigned char *handle,    /* i; handle */
    unsigned char *name,      /* i; name of variable to extract */
    unsigned long *name_length, /* i; length of the name */
    unsigned char *value,     /* o; buffer in which to put
                               the value */
    unsigned long *value_length, /* i/o; buffer size */
    long *return_code        /* o; return code */
)
```

If this function returns the code HTTPD_BUFFER_TOO_SMALL, the buffer size you requested was not big enough for the extracted value. In this case, the function does not use the buffer but updates the *value_length* parameter with the buffer size that you need in order to successfully extract this value. Retry the extraction with a buffer that is at least as big as the returned *value_length*.

Note: If the variable being extracted is for an HTTP header, the HTTPD_extract() function will extract only the first matching occurrence, even if the request contains multiple headers with the same name. The httpd_getvar() function can be used instead of HTTPD_extract(), and also offers other benefits. Refer to the section on the “httpd_getvar() function” for more information.

HTTPD_file()

Sends a file to satisfy this request. Valid only in the PreExit, Service, Error, PostAuthorization, and Transmogriifier steps.

```
void HTTPD_LINKAGE HTTPD_file (
    unsigned char *handle,      /* i; handle */
    unsigned char *name,        /* i; name of file to send */
    unsigned long *name_length, /* i; length of the name */
    long *return_code           /* o; return code */
)
```

httpd_getvar()

The same as HTTPD_extract(), except that it is easier to use because the user does not have to specify lengths for the arguments.

```
const unsigned char *      /* o; value of variable */
HTTPD_LINKAGE
httpd_getvar(
    unsigned char *handle,    /* i; handle */
    unsigned char *name,      /* i; variable name */
    unsigned long *n          /* i; index number for the array
                               containing the header */
)
```

The index for the array containing the header begins with 0. To obtain the first item in the array, use the value 0 for *n*; to obtain the fifth item, use the value 4 for *n*.

Note: Do not discard or change the contents of the returned value. The returned string is null terminated.

HTTPD_log_access()

Writes a string to the server’s access log.

```
void HTTPD_LINKAGE HTTPD_log_access (
    unsigned char *handle,      /* i; handle */
    unsigned char *value,       /* i; data to write */
    unsigned long *value_length, /* i; length of the data */
    long *return_code           /* o; return code */
)
```

Note that escape symbols are *not* required when writing the percent symbol (%) in server access logs.

HTTPD_log_error()

Writes a string to the server’s error log.

```
void HTTPD_LINKAGE HTTPD_log_error (
    unsigned char *handle,      /* i; handle */
    unsigned char *value,       /* i; data to write */
    unsigned long *value_length, /* i; length of the data */
    long *return_code           /* o; return code */
)
```

Note that escape symbols are *not* required when writing the percent symbol (%) in server error logs.

HTTPD_log_event()

Writes a string to the server's event log.

```
void HTTPD_LINKAGE HTTPD_log_event (
    unsigned char *handle,      /* i; handle */
    unsigned char *value,       /* i; data to write */
    unsigned long *value_length, /* i; length of the data */
    long *return_code           /* o; return code */
)
```

Note that escape symbols are *not* required when writing the percent symbol (%) in server event logs.

HTTPD_log_trace()

Writes a string to the server's trace log.

```
void HTTPD_LINKAGE HTTPD_log_trace (
    unsigned char *handle,      /* i; handle */
    unsigned char *value,       /* i; data to write */
    unsigned long *value_length, /* i; length of the data */
    long *return_code           /* o; return code */
)
```

Note that escape symbols are *not* required when writing the percent symbol (%) in server trace logs.

HTTPD_open()

(Valid only in the Transmogriifier step.) Transfers control to the next routine in the stream stack. Call this from the Transmogriifier open, write, or close functions after any desired headers are set and you are ready to begin the write routine.

```
void HTTPD_LINKAGE HTTPD_open (
    unsigned char *handle,      /* i; handle */
    long *return_code           /* o; return code */
)
```

HTTPD_proxy()

Makes a proxy request. Valid in the PreExit, Service, and PostAuthorization steps.

Note: This is a completion function; the request is complete after this function.

```
void HTTPD_LINKAGE HTTPD_proxy (
    unsigned char *handle,      /* i; handle */
    unsigned char *url_name,    /* i; URL for the
                                proxy request */
    unsigned long *name_length, /* i; length of URL */
    void *request_body,        /* i; body of request */
    unsigned long *body_length, /* i; length of body */
    long *return_code           /* o; return code */
)
```

HTTPD_read()

Reads the body of the client's request. Use HTTPD_extract() for headers. Valid only in the PreExit, Authorization, PostAuthorization, and Service steps and is useful only if a PUT or POST request has been done. Call this function in a loop until HTTPD_EOF is returned. If there is no body for this request, this function fails.

```
void HTTPD_LINKAGE HTTPD_read (
    unsigned char *handle,      /* i; handle */
    unsigned char *value,       /* i; buffer for data */
)
```

```

        unsigned long *value_length,    /* i/o; buffer size
                                         (data length) */
        long *return_code               /* o; return code */
    )

```

HTTPD_restart()

Restarts the server after all active requests have been processed. Valid in all steps except for Server Initialization, Server Termination, and Transmogriifier.

```

void HTTPD_LINKAGE HTTPD_restart (
    long *return_code /* o; return code */
)

```

HTTPD_set()

Sets the value of a variable associated with this request. The variables that are valid for the *name* parameter are the same as those used in the CGI. See “Variables” on page 24 for more information.

Note that you can also create variables with this function. Variables that you create are subject to the conventions for HTTP_ and PROXY_ prefixes, which are described in “Variables” on page 24. If you create a variable that begins with HTTP_, it is sent as a header in the response to the client, without the HTTP_ prefix. For example, to set a Location header, use HTTPD_set() with the variable name HTTP_LOCATION. Variables created with a PROXY_ prefix are sent as headers in the request to the content server. Variables created with a CGI_ prefix are passed to CGI programs.

This function is valid in all steps; however, not all variables are valid in all steps.

```

void HTTPD_LINKAGE HTTPD_set (
    unsigned char *handle,    /* i; handle */
    unsigned char *name,     /* i; name of value to set */
    unsigned long *name_length, /* i; length of the name */
    unsigned char *value,    /* i; buffer with value */
    unsigned long *value_length, /* i; length of value */
    long *return_code        /* o; return code */
)

```

Note: You can use the `httpd_setvar()` function to set a variable value without having to specify a buffer and length. Refer to the section on “`httpd_setvar()` function” for information.

httpd_setvar()

The same as HTTPD_set(), except that it is easier to use because the user does not have to specify lengths for the arguments.

```

long /* o; return code */
HTTPD_LINKAGE httpd_setvar (
    unsigned char *handle,    /* i; handle */
    unsigned char *name,     /* i; variable name */
    unsigned char *value,    /* i; new value */
    unsigned long *addHdr    /* i; add header or replace it */
)

```

The *addHdr* parameter has four possible values:

- HTTPD_SETVAR_REPLACE — Replace all occurrences of the header variable with the new value.
- HTTPD_SETVAR_REPLACE_ADD — If the header variable exists, replace its first occurrence with the new value; if the variable does not exist, append the new value to the headers.
- HTTPD_SETVAR_ADD — Append this value to the headers.

- HTTPD_SETVAR_REMOVE_ALL — Delete all occurrences of this header variable.

These values are defined in HTAPI.h.

httpd_variant_insert()

Inserts a variant into the cache.

```
void HTTPD_LINKAGE httpd_variant_insert (
    unsigned char *handle,      /* i; handle */
    unsigned char *URI,        /* i; URI of this object */
    unsigned char *dimension,  /* i; dimension of variation */
    unsigned char *variant,    /* i; value of the variant */
    unsigned char *filename,   /* i; file containing the object */
    long *return_code          /* o; return code */
)
```

Notes:

1. The dimension argument refers to the header by which this object varies from the URI. For instance, in the example above, a possible dimension value is User-Agent.
2. The variant argument refers to the value of the header for the header given in the dimension argument. This varies from the URI. For instance, in the example above, a possible value for the variant argument is the following:
Mozilla 4.0 (compatible; BatBrowser 94.1.2; Bat OS)
3. The filename argument must point to a null-terminated copy of the file name in which the user has saved the modified content. The user is responsible for removing the file; this action is safe after return from this function. The file contains only the body with no headers.
4. When caching variants, the server updates the content-length header and adds a Warning: 214 header. Strong entity tags are removed.

httpd_variant_lookup()

Determines if a given variant exists in the cache.

```
void HTTPD_LINKAGE httpd_variant_lookup (
    unsigned char *handle,      /* i; handle */
    unsigned char *URI,        /* i; URI of this object */
    unsigned char *dimension,  /* i; dimension of variation */
    unsigned char *variant,    /* i; value of the variant */
    long *return_code);        /* o; return code */
```

HTTPD_write()

Writes the body of the response. Valid in the PreExit, Service, Error, and Transmogriifier steps.

If you do not set the content type before calling this function for the first time, the server assumes that you are sending a CGI data stream.

```
void HTTPD_LINKAGE HTTPD_write (
    unsigned char *handle,      /* i; handle */
    unsigned char *value,       /* i; data to send */
    unsigned char *value_length, /* i; length of the data */
    long *return_code);         /* o; return code */
```

Note: To set response headers, refer to the section on the “HTTPD_set() function” on page 19.

Note: After an HTTPD_* function returns, it is safe for you to free any memory that you passed with it.

Return codes from predefined functions and macros

The server will set the return code parameter to one of these values, depending on the success of the request:

Table 3. Return codes

Value	Status code	Explanation
-1	HTTPD_UNSUPPORTED	The function is not supported.
0	HTTPD_SUCCESS	The function succeeded, and the output fields are valid.
1	HTTPD_FAILURE	The function failed.
2	HTTPD_INTERNAL_ERROR	An internal error was encountered and processing for this request cannot continue.
3	HTTPD_PARAMETER_ERROR	One or more invalid parameters was passed.
4	HTTPD_STATE_CHECK	The function is not valid in this process step.
5	HTTPD_READ_ONLY	(Returned only by HTTPD_set and httpd_setvar.) The variable is read-only and cannot be set by the plug-in.
6	HTTPD_BUFFER_TOO_SMALL	(Returned by HTTPD_set, httpd_setvar, and HTTPD_read.) The buffer provided was too small.
7	HTTPD_AUTHENTICATE_FAILED	(Returned only by HTTPD_authenticate.) The authentication failed. Examine the HTTP_RESPONSE and HTTP_REASON variables for more information.
8	HTTPD_EOF	(Returned only by HTTPD_read.) Indicates the end of the request body.
9	HTTPD_ABORT_REQUEST	The request was aborted because the client provided an entity tag that did not match the condition specified by the request.
10	HTTPD_REQUEST_SERVICED	(Returned by HTTPD_proxy.) The function that was called completed the response for this request.
11	HTTPD_RESPONSE_ALREADY_COMPLETED	The function failed because the response for that request has already been completed.
12	HTTPD_WRITE_ONLY	The variable is write-only and cannot be read by the plug-in.

Caching Proxy configuration directives for API steps

Each step in the request process has a configuration directive that you use to indicate which of your plug-in functions you want to call and execute during that step. You can add these directives to your server's configuration file (ibmproxy.conf) by manually editing and updating it, or by using the API Request Processing form in the Caching Proxy Configuration and Administration forms.

API usage notes

- Except for the Service and NameTrans directives, the API directives for each step do not need to appear in any particular order in the configuration file. Note that the order of multiple entries for one API directive is significant, as described later in this list.
- It is not necessary to include an entry for every API step. If you do not have a plug-in for a particular step, omit the corresponding directive and the standard processing for that step will be used.
- The Service and NameTrans directives work like the other mapping directives (for example, the Pass directive) and are dependent on their occurrence and placement relative to other mapping directives within the configuration file. For example, a rule for /cgi-bin/foo.so must appear before the rule for /cgi-bin/*. This means that the server processes the Service, NameTrans, Exec, Fail, Map, Pass, Proxy, ProxyWAS, and Redirect directives in their sequence within the configuration file. When the server successfully maps a URL to a file, it does not read or process any other of these directives. (The Map directive is an exception. Refer to the *WebSphere Application Server Caching Proxy Administration Guide* for complete information about proxy server mapping rules.)
- You can have more than one configuration directive for a step. For example, you can include two NameTrans directives, each pointing to a different plug-in function. When the server performs the name translation step, it processes your name translation functions in the order in which they appear within the configuration file.

Note: If a plug-in function provided with the Caching Proxy uses the same API directive as a plug-in you have written, place your plug-in's directive after the system plug-in directive.

- Certain plug-in functions do not have to be executed for every request:
 - Several directives include a URL mask. Specifying a URL mask with these directives causes the plug-in application to be called only for requests whose URLs match that pattern. Refer to “API directives and syntax” for information about which steps can use URL masks and to “API directive variables” on page 23 for information about how to use this feature.
 - Specify an authentication scheme with the Authentication directive to indicate that you want the plug-in to be called only for certain types of authentication. Currently, only basic authentication is supported by the HTTP protocol. See “API directive variables” on page 23 for additional information.
- If the server fails to load a specific plug-in function, or if you have a ServerInit directive that does not return an OK return code, no other plug-in functions for that compiled Caching Proxy plug-in are called. Any processing specific to that plug-in that was done up to this point is ignored. Other Caching Proxy plug-ins that you include in these directives, and their functions, are not affected.

API directives and syntax

These configuration file directives must appear in the `ibmproxy.conf` file as one line, with no spaces other than those explicitly specified here. Although line breaks appear for readability in some of the syntax examples, there must be no spaces at those points in the actual directive.

Table 4. Caching Proxy plug-in API directives

ServerInit		/path/file:function_name init_string
PreExit		/path/file:function_name
Authentication	type	/path/file:function_name

Table 4. Caching Proxy plug-in API directives (continued)

NameTrans	/URL	/path/file:function_name
Authorization	/URL	/path/file:function_name
ObjectType	/URL	/path/file:function_name
PostAuth		/path/file:function_name
Service	/URL	/path/file:function_name
Midnight		/path/file:function_name
Transmogriifier		/path/file:open_function_name: write_function_name: close_function_name:error_function
Log	/URL	/path/file:function_name
Error	/URL	/path/file:function_name
PostExit		/path/file:function_name
ServerTerm		/path/file:function_name
ProxyAdvisor		/path/file:function_name
GCAdvisor		/path/file:function_name

API directive variables

The variables in these directives have the following meanings:

type Used only with the Authentication directive to specify whether or not your plug-in function is called. Valid values are the following:

- Basic — The plug-in function is called only for basic authentication requests.
- * — The plug-in function is called for all requests. Currently, only basic authentication is supported by HTTP protocol. For nonbasic authentication requests, you can return an error code indicating that this type of authentication is not supported.

URL Specifies the requests for which your plug-in function is called. Requests with URLs that match this template will cause the plug-in function to be used. URL specifications in these directives are virtual (they do not include the protocol) but are preceded by a slash (/). For example, /www.ics.raleigh.ibm.com is correct, but http://www.ics.raleigh.ibm.com is not. You can specify this value as a specific URL or as a template.

- specific URL — The plug-in function is called only for that exact URL.
- URL template — The plug-in function is called for all URLs that match the template. Templates can include the wildcard character * and can be specified in the forms /URL* or /* or *

Note: A URL template is *required* with the Service directive if you want path translation to occur.

path/file

The fully qualified file name of your compiled program.

function_name

The name that you gave your plug-in function within your program.

The Service directive requires an asterisk (*) after the function name if you want to have access to path information.

init_string

This optional part of the ServerInit directive can contain any text that you want to pass to your plug-in function. Use `httpd_getvar()` to extract the text from the `INIT_STRING` variable.

For additional information, including syntax, for these directives, see the *WebSphere Application Server Caching Proxy Administration Guide*.

Compatibility with other APIs

The Caching Proxy API is backward-compatible with ICAPI and GWAPI, through version 4.6.1.

Porting CGI programs

Use the following guidelines for porting CGI applications written in C to use the Caching Proxy API:

- Remove your `main()` entry point, or rename it so that you can build a DLL.
- Eliminate global variables or protect them with a mutual exclusion semaphore.
- Change the following calls in your programs:
 - Change `printf()` header calls to `HTTPD_set()` or `httpd_setvar()`.
 - Change `printf()` data calls to `HTTPD_write()`.
 - Change `getenv()` calls to `HTTPD_extract()` or `httpd_getvar()`. Note that this returns unallocated memory, so you must free the result.
- Remember that the server runs in a multithreaded environment, and your plug-in functions must be thread safe. If the functions are reentrant, performance does not decrease.
- Remember to set the Content-Type header if you are using `HTTPD_write()` to send data back to the client.
- Check your code meticulously for memory leaks.
- Think about your error paths. If you generate error messages yourself and send them back as HTML, you must return `HTTPD_OK` from your service function or functions.

Caching Proxy API reference information

Variables

When writing API programs, you can use Caching Proxy variables that provide information about the remote client and server system.

Notes:

- User-defined variable names cannot have a prefix of `SERVER_`. The Caching Proxy API function reserves any variable starting with `SERVER_` for the server and, therefore, these variables are read-only. In addition, the prefixes `HTTP_` and `PROXY_` also are reserved for HTTP headers.
- All request headers sent by the client (such as Set-Cookie) are prefixed by `HTTP_` and their values can be extracted. To access variables that are request headers, prefix the variable name with `HTTP_`. You can also create new variables using the `httpd_setvar()` predefined function. For details about these headers, see “Return codes from predefined functions and macros” on page 21.
- Two variable prefixes, `HTTP_` and `PROXY_`, are used to denote whether a variable applies to headers for the request or for the response. The `HTTP_` prefix

refers to variables that flow between the client and the Caching Proxy. The `PROXY_` prefix refers to variables that flow between the Caching Proxy and the origin server (or the next server in a proxy chain). These variables are valid only during the request processing steps.

- Extracting an `HTTP_*` variable gives you the value of a header that was in the client's request to the proxy server.
- Setting an `HTTP_*` variable sets the response header that is sent from the proxy server to the client.
- Extracting a `PROXY_*` variable gives you the value for a header returned from the content server to the proxy server.
- Setting a `PROXY_*` variable sets the request header that is sent from the proxy server to the content server (or to the next server in a proxy chain).

Figure 2 demonstrates the use of these prefixes as the Caching Proxy handles a client request.

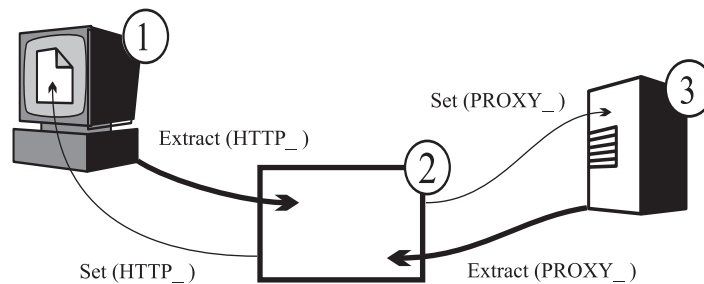


Figure 2. `HTTP_` and `PROXY_` variable prefixes. Legend: 1—Client machine 2—Caching Proxy 3—Origin server

- Some variables are read-only. Read-only variables represent values that you can extract from a request or a response and use in the `httpd_getvar()` predefined function. A return code of `HTTPD_READ_ONLY` results if you try to change read-only variables by using the `httpd_setvar()` function.
- Variables not identified as read-only can be read and set in the `httpd_getvar()` or `httpd_setvar()` predefined functions. These variables represent values that you can extract from a request or response; or values that you can set or create when processing a request or response.

Variable definitions

Note: Header variables that do not begin with `HTTP_` or `PROXY_` prefixes are ambiguous. To avoid ambiguity, always use the `HTTP_` or `PROXY_` prefix with variable names for headers.

ACCEPT_RANGES

Contains the value of the Accept-Ranges response header, which specifies whether the content server can respond to range requests. Use `PROXY_ACCEPT_RANGES` to extract the header value that is sent by the content server to the proxy. Use `HTTP_ACCEPT_RANGES` to set the header value that is sent from the proxy to the client.

Note: `ACCEPT_RANGES` is ambiguous. To eliminate ambiguity, use `HTTP_ACCEPT_RANGES` and `PROXY_ACCEPT_RANGES` instead.

ALL_VARIABLES

Read-only. Contains all of the CGI variables. For example:

ACCEPT_RANGES BYTES
CLIENT_ADDR 9.67.84.3

AUTH_STRING

Read-only. If the server supports client authentication, this string contains the undecoded credentials to be used to authenticate the client.

AUTH_TYPE

Read-only. If the server supports client authentication and the script is protected, this variable contains the method used to authenticate the client. For example, Basic.

CACHE_HIT

Read-only. Identifies whether or not the proxy request was found in the cache. Values returned include the following:

- 0 - The request was not found in the cache.
- 1 - The request was found in the cache.

CACHE_MISS

Write-only. Used to force a cache miss. Valid values are the following:

- 0 - Do not force a cache miss.
- 1 - Force a cache miss.

CACHE_TASK

Read-only. Identifies whether the cache was used. Values returned include the following:

- 0 - The request did not access or update the cache.
- 1 - The request was served from cache.
- 2 - The requested object was in the cache but needed to be revalidated.
- 3 - The requested object was not in the cache and possibly has been added.

This variable can be used in the PostAuthorization, PostExit, ProxyAdvisor, or Log steps.

CACHE_UPDATE

Read-only. Identifies whether or not the proxy request updated the cache. Values returned include the following:

- 0 - The cache was not updated.
- 1 - The cache was updated.

CLIENT_ADDR or CLIENTADDR

Same as REMOTE_ADDR.

CLIENTMETHOD

Same as REQUEST_METHOD.

CLIENT_NAME or CLIENTNAME

Same as REMOTE_HOST.

CLIENT_PROTOCOL or CLIENTPROTOCOL

Contains the name and version of the protocol that the client is using to make the request. For example, HTTP/1.1.

CLIENT_RESPONSE_HEADERS

Read-only. Returns a buffer containing the headers that the server sends to the client.

CONNECTIONS

Read-only. Contains the number of connections being served, or the number of active requests. For example, 15.

CONTENT_CHARSET

Character set of the response for text/*, for example, US ASCII. Extracting this variable applies to the content-charset header from the client. Setting it affects the content-charset header in the request to the content server.

CONTENT_ENCODING

Specifies the encoding used in the document, for example, x-gzip. Extracting this variable applies to the content-encoding header from client. Setting it affects the content-charset header in the request to the content server.

CONTENT_LENGTH

Extracting this variable applies to the header from the client's request. Setting it affects the value of the header in the request to the content server.

Note: CONTENT_LENGTH is ambiguous. To eliminate ambiguity, use HTTP_CONTENT_LENGTH and PROXY_CONTENT_LENGTH.

CONTENT_TYPE

Extracting this variable applies to the header from the client's request. Setting it affects the value of the header in the request to the content server.

Note: CONTENT_TYPE is ambiguous. To eliminate ambiguity, use HTTP_CONTENT_TYPE and PROXY_CONTENT_TYPE.

CONTENT_TYPE_PARAMETERS

Contains other MIME attributes, but not the character set. Extracting this variable applies to the header from the client request. Setting it affects the value of header in the request to the content server.

DOCUMENT_URL

Contains the Uniform Request Locator (URL). For example:
`http://www.anynet.com/~userk/main.htm`

DOCUMENT_URI

Same as DOCUMENT_URL.

DOCUMENT_ROOT

Read-only. Contains the document root path, as defined by pass rules.

ERRORINFO

Specifies the error code to determine the error page. For example, blocked.

EXPIRES

Defines when documents stored in a proxy's cache expire. Extracting this variable applies to the header from client request. Setting it affects the value of header in the request to the content server. For example:

`Mon, 01 Mar 2002 19:41:17 GMT`

GATEWAY_INTERFACE

Read-only. Contains the version of the API that the server is using. For example, ICSAPI/2.0.

GC_BIAS

Write-only. This floating-point value influences the garbage collection decision for the file being considered for garbage collection. The value

entered is multiplied by the Caching Proxy's quality setting for that file type to determine ranking. Quality settings range from 0.0 to 0.1 and are defined by the AddType directives in the proxy configuration file (ibmproxy.conf).

GC_EVALUATION

Write-only. This floating-point value determines whether to remove (0.0) or keep (1.0) the file being considered for garbage collection. Values between 0.0 and 1.0 are ordered by rank, that is, a file with the GC_EVALUATION value 0.1 is more likely to be removed than a file with the GC_EVALUATION value 0.9.

GC_EXPIRES

Read-only. Identifies how many seconds remain until the file under consideration expires in the cache. This variable can be extracted only by a GC Advisor plug-in.

GC_FILENAME

Read-only. Identifies the file being considered for garbage collection. This variable can be extracted only by a GC Advisor plug-in.

GC_FILESIZE

Read-only. Identifies the size of the file being considered for garbage collection. This variable can be extracted only by a GC Advisor plug-in.

GC_LAST_ACCESS

Read-only. Identifies when the file was last accessed. This variable can be extracted only by a GC Advisor plug-in.

GC_LAST_CHECKED

Read-only. Identifies when the files were last checked. This variable can be extracted only by a GC Advisor plug-in.

GC_LOAD_DELAY

Read-only. Identifies how long it took to retrieve the file. This variable can be extracted only by a GC Advisor plug-in.

HTTP_COOKIE

When read, this variable contains the value of the Set-Cookie header set by the client. It can also be used to set a new cookie in the response stream (between the proxy and the client). Setting this variable causes the creation of a new Set-Cookie header in the document request stream, regardless of whether or not a duplicate header exists.

HTTP_HEADERS

Read-only. Used to extract all of the client request headers.

HTTP_REASON

Setting this variable affects the reason string in the HTTP response. Setting it also affects the reason string in the proxy's response to the client. Extracting this variable returns the reason string in the response from the content server to the proxy.

HTTP_RESPONSE

Setting this variable affects the response code in the HTTP response. Setting it also affects the status code in the proxy's response to the client. Extracting this variable returns the status code in the response from the content server to the proxy.

HTTP_STATUS

Contains the HTTP response code and reason string. For example, 200 OK.

HTTP_USER_AGENT

Contains the value of the User-Agent request header, which is the name of the client Web browser, for example, Netscape Navigator / V2.02. Setting this variable affects the header in the proxy's response to the client. Extracting it applies to the header from the client's request.

INIT_STRING

Read-only. The ServerInit directive defines this string. This variable can be read only during the Server Initialization step.

LAST_MODIFIED

Extracting this variable applies to the header from the client request. Setting it affects the value of the header in the request to the content server. For example:

Mon, 01 Mar 1998 19:41:17 GMT

LOCAL_VARIABLES

Read-only. All the user-defined variables.

MAXACTIVETHREADS

Read-only. The maximum number of active threads.

NOTMODIFIED_TO_OK

Forces a full response to the client. Valid in the PreExit and ProxyAdvisor steps.

ORIGINAL_HOST

Read-only. Returns the host name or destination IP address of a request.

ORIGINAL_URL

Read-only. Returns the original URL sent in the client request.

OVERRIDE_HTTP_NOTTRANSFORM

Enables modification of data in the presence of a Cache-Control: no-transform header. Setting this variable affects the response header to the client.

OVERRIDE_PROXY_NOTTRANSFORM

Enables modification of data in the presence of a Cache-Control: no-transform header. Setting this variable affects the request to the content server.

PASSWORD

For basic authentication, contains the decoded password. For example, password.

PATH Contains the fully translated path.

PATH_INFO

Contains the additional path information as sent by the Web browser. For example, /foo.

PATH_TRANSLATED

Contains the decoded or translated version of the path information contained in PATH_INFO. For example:

d:\wwwhome\foo

/wwwhome/foo

PPATH

Contains the partially translated path. Use this in the Name Translation step.

PROXIED_CONTENT_LENGTH

Read-only. Returns the length of the response data actually transferred through the proxy server.

PROXY_ACCESS

Defines whether the request is a proxy request. For example, NO.

PROXY_CONTENT_TYPE

Contains the Content-Type header of the proxy request made through HTTPD_proxy(). When information is sent with the method of POST, this variable contains the type of data included. You can create your own content type in the proxy server configuration file and map it to a viewer. Extracting this variable applies to the header value from the content server response. Setting it affects the header for the request to the content server. For example:

`application/x-www-form-urlencoded`

PROXY_CONTENT_LENGTH

The Content-Length header of the proxy request made through HTTPD_proxy(). When information is sent with the method of POST, this variable contains the number of characters of data. Servers typically do not send an end-of-file flag when they forward the information using standard input. If needed, you can use the CONTENT_LENGTH value to determine the end of the input string. Extracting this variable applies to the header value from the content server response. Setting it affects the header for the request to the content server. For example:

`7034`

PROXY_COOKIE

When read, this variable contains the value of the Set-Cookie header set by the origin server. It also can be used to set a new cookie in the request stream. Setting this variable causes the creation of a new Set-Cookie header in the document request stream, regardless of whether or not a duplicate header exists.

PROXY_HEADERS

Read-only. Used to extract the Proxy headers.

PROXY_METHOD

Method for the request made through HTTPD_proxy(). Extracting this variable applies to the header value from the content server response. Setting it affects the header for the request to the content server.

QUERY_STRING

When information is sent by using a method of GET, this variable contains the information that follows a question mark (?) in a query. This information must be decoded by the CGI program. For example:

`NAME=Eugene+T%2E+Fox&ADDR=etfox%7Cibm.net&INTEREST=xyz`

RCA_OWNER

Read-only. Returns a numeric value, giving the node that owned the requested object. This variable can be used in the PostExit, ProxyAdvisor, or Log steps, and is meaningful only when the server is part of a cache array using remote cache access (RCA).

RCA_TIMEOUTS

Read-only. Returns a numeric value, containing the total (aggregate) number of timeouts on RCA requests to all peers. You can use this variable in any step.

REDIRECT_*

Read-only. Contains a redirection string for the error code that corresponds to the variable name (for example, REDIRECT_URL). A list of possible REDIRECT_ variables can be found in online documentation for the Apache Web server at <http://httpd.apache.org/docs-2.0/custom-error.html>.

REFERRER_URL

Read-only. Contains the last URL location of the browser. It allows the client to specify, for the server's benefit, the address (URL) of the resource from which the Request-URL was obtained. For example:

`http://www.company.com/homepage`

REMOTE_ADDR

Contains the IP address of the Web browser, if available. For example, 45.23.06.8.

REMOTE_HOST

Contains the host name of the Web browser, if available. For example, `www.raleigh.ibm.com`.

REMOTE_USER

If the server supports client authentication and the script is protected, this variable contains the user name passed for authentication. For example, `joeuser`.

REQHDR

Read-only. Contains a list of the headers sent by the client.

REQUEST_CONTENT_TYPE

Read-only. Returns the content type of the request body. For example: `application/x-www-form-urlencoded`

REQUEST_CONTENT_LENGTH

Read-only. When information is sent with the method of POST, this variable contains the number of characters of data. Servers typically do not send an end-of-file flag when they forward the information using standard input. If needed, you can use the CONTENT_LENGTH value to determine the end of the input string. For example, 7034.

REQUEST_METHOD

Read-only. Contains the method (as specified with the METHOD attribute in an HTML form) used to send the request. For example, GET or POST.

REQUEST_PORT

Read-only. Returns the port number specified in the URL, or a default port based on the protocol.

RESPONSE_CONTENT_TYPE

Read-only. When information is sent with the method of POST, this variable contains the type of data included. You can create your own content type in the proxy server configuration file and map it to a viewer. For example, `text/html`.

RESPONSE_CONTENT_LENGTH

Read-only. When information is sent with the method of POST, this variable contains the number of characters of data. Servers typically do not send an end-of-file flag when they forward the information using standard input. If needed, you can use the CONTENT_LENGTH value to determine the end of the input string. For example, 7034.

RULE_FILE_PATH

Read-only. Contains the fully qualified file system path and file name of the configuration file.

SSL_SESSIONID

Read-only. Returns the SSL session ID if the current request is received on an SSL connection. Returns NULL if the current request is not received on an SSL connection.

SCRIPT_NAME

Contains the URL of the request.

SERVER_ADDR

Read-only. Contains the local IP address of the proxy server.

SERVER_NAME

Read-only. Contains the proxy server host name or IP address of the content server for this request. For example, `www.ibm.com`.

SERVER_PORT

Read-only. Contains the port number of the proxy server to which the client request was sent. For example, `80`.

SERVER_PROTOCOL

Read-only. Contains the name and version of the protocol used to make the request. For example, `HTTP/1.1`.

SERVER_ROOT

Read-only. Contains the directory where the proxy server program is installed.

SERVER_SOFTWARE

Read-only. Contains the name and version of the proxy server.

STATUS

Contains the HTTP response code and reason string. For example, `200 OK`.

TRACE

Determines how much information will be traced. Returned values include:

- OFF - No tracing.
- V - Verbose mode.
- VV - Very Verbose mode.
- MTV - Much Too Verbose mode.

URI Read/Write. Same as `DOCUMENT_URL`.

URI_PATH

Read-only. Returns the path portion only for a URL.

URL Read/Write. Same as `DOCUMENT_URL`.

URL_MD4

Read-only. Returns the file name of the potential cache file for the current request.

USE_PROXY

Identifies the proxy to chain to for the current request. Specify the URL. For example, `http://myproxy:8080`.

USERID

Same as `REMOTE_USER`.

USERNAME

Same as REMOTE_USER.

Authentication and authorization

First, a short review of the terminology:

Authentication

The verification of the security tokens associated with this request in order to ascertain the identity of the requester.

Authorization

A process that uses security tokens to determine whether the requester has access to the resource.

Figure 3 on page 34 depicts the proxy server's authentication and authorization process.

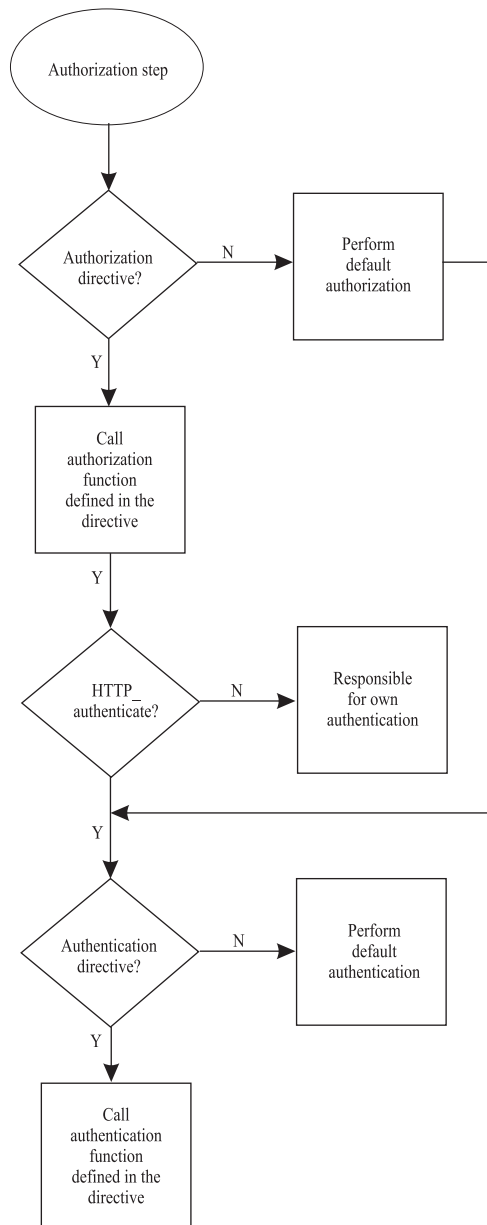


Figure 3. Proxy server authentication and authorization process

As demonstrated in Figure 3, the initiation of the authorization process is the first step in the server's authorization and authentication process.

In the Caching Proxy, authentication is part of the authorization process; it occurs only when authorization is required.

Authentication and authorization process

The proxy server follows these steps when processing a request that requires authorization.

1. First, the proxy server examines its configuration file to determine whether or not there is an authorization directive.
 - If an authorization directive is present in the configuration file, the server calls the authorization function defined in the directive and begins authentication with step 2 on page 35.

- If there is no authorization directive, the server performs a default authorization and then proceeds directly to the authentication procedures in step 3.
- 2. The proxy server begins the authentication process by checking to see if the HTTPD_authenticate header is present in the client request.
 - If the header is present, the server continues the authentication process (see step 3).
 - If the header is not present, authentication must be performed by another method.
- 3. The proxy server checks to see if there is an authentication directive present in the proxy configuration file.
 - If an authentication directive is present in the configuration file, the server calls the authentication function defined in the directive.
 - If there is no directive, the server performs a default authentication.

If your Caching Proxy plug-in provides its own authorization process, *it overrides* the default server authorization and authentication. Therefore, if you have authorization directives in your configuration file, the plug-in functions associated with them must also handle any necessary authentication. The predefined HTTPD_authenticate() function is provided for you to use.

There are three ways to provide for authentication in your authorization plug-ins:

- Write your own separate authorization and authentication plug-ins. In your proxy configuration file, use both the Authorization and the Authentication directives to specify these functions. Be sure to include the HTTPD_authenticate() function call in your authorization plug-in function.
When the Authorization step is executed, it performs your authorization plug-in function, which, in turn, calls your authentication plug-in function.
- Write your own authorization plug-in function, but have it call the default server authentication. In your proxy configuration file, use the Authorization directive to specify your function. In this case, you do not need the Authentication directive. Be sure to call the HTTPD_authenticate() function in your authorization plug-in function.
When the Authorization step is executed, it performs your authorization plug-in function, which, in turn, calls the default server authentication.
- Write your own authorization plug-in function and include all required authentication processing in it. Do not use the HTTPD_authenticate() function in your authorization plug-in. In your proxy configuration file, use the Authorization directive to specify your authorization plug-in. In this case, you do not need the Authentication directive.
When the Authorization step is executed, it performs your authorization plug-in function and any authentication it includes.

If your Caching Proxy plug-in does not provide its own authorization process, you can still provide customized authentication by using the following method:

- Write your own authentication plug-in function. In your proxy configuration file, use the Authentication directives to specify your function. In this case, you do not need the Authorization directive.

When the Authorization step is executed, it performs the default server authorization, which, in turn, calls your authentication plug-in function.

Remember the following points:

- If you do not have any Authorization directives in your configuration file, or if their specified plug-in functions decline to handle the request by returning HTTP_NOACTION, the server's default authorization occurs.
- If you have Authorization directives in your configuration file and their plug-in functions include HTTPD_authenticate(), the server calls any authentication functions specified in the Authentication directives. If you have not defined any Authentication directives, or if their specified plug-in functions decline to handle the request by returning HTTP_NOACTION, the server's default authentication occurs.
- If you have Authorization directives in your configuration file but their plug-in functions do not include HTTPD_authenticate(), no authentication functions are called by the server. You must write your own authentication processing as part of your authorization plug-in functions or make your own calls to other authentication modules.
- The Caching Proxy automatically generates a challenge (prompting the browser to return a user ID and password) if your authorization function returns the codes 401 or 407. However, you must still configure a protection setup in the Caching Proxy so that this action occurs correctly.

Variant caching

Use variant caching to cache data that is a modified form of the original document (the URI). The Caching Proxy handles variants generated by the API. *Variants* are different versions of a base document.

In general, when origin servers send variants, they fail to identify them as such. The Caching Proxy supports only variants created by plug-ins (for example, code page conversion). If a plug-in creates a variant based on criteria that are not in the HTTP header, it must include a PreExit or PostAuthorization step function to create a pseudoheader so that the Caching Proxy can correctly identify the existing variant.

For example, use a Transmogrifier API program to modify the data that users request based on the value of the User-Agent header that the browser sends. In the *close* function, save the modified content to a file or specify a buffer length and pass the buffer as the data argument. Then use the variant caching functions `httpd_variant_insert()` and `httpd_variant_lookup()` to put the content in the cache.

API examples

To help you get started with your own Caching Proxy API functions, look at the sample programs provided in the samples directory of the Edge components installation CD-ROM. Additional information is available on the WebSphere Application Server Web site, www.ibm.com/software/webservers/appserv/.

Chapter 3. Custom advisors

This section discusses writing custom advisors for the Load Balancer.

Advisors provide load-balancing information

Advisors are software agents that work within Load Balancer to provide information about the load on a given server. A different advisor exists for each standard protocol (HTTP, SSL, and others). Periodically, the Load Balancer base code performs an advisor cycle, during which it individually evaluates the status of all servers in its configuration.

By writing your own advisors for the Load Balancer, you can customize how your server machines' load is determined.

Standard advisor function

In general, advisors work to enable load balancing in the following manner.

1. Periodically, the advisor opens a connection with each server and sends it a request message. The content of the message is specific to the protocol running on the server; for instance, the HTTP advisor sends a HEAD request to the server.
2. The advisor listens for a response from the server. After getting the response, the advisor calculates and reports the load value for that server. Different advisors calculate the load value in different ways, but most standard advisors measure the time the server takes to respond, then reports that value in milliseconds as the load.
3. The advisor reports the load to the Load Balancer's manager function. The load appears in the Port column of the manager report. The manager uses the advisor's reported load along with weights set by the administrator to determine how to load balance incoming requests to the servers.
4. If a server does not respond, the advisor returns a negative value (-1) for the load. The manager uses this information to determine when to suspend service for a particular server.

Standard advisors provided with the Load Balancer include advisors for the following functions. Detailed information about these advisors is available in the *WebSphere Application Server Load Balancer Administration Guide*

- Connect
- DB2
- DNS
- FTP
- HTTP
- HTTPS
- IMAP
- LDAP
- NNTP
- Ping
- POP3

- Reach
- Self
- SIP
- SMTP
- SSL
- Telnet
- WebSphere Application Server
- WebSphere Application Server Caching Proxy
- Workload Manager

To support proprietary protocols for which standard advisors are not provided, you must write custom advisors.

Creating a custom advisor

A custom advisor is a small piece of Java code, provided as a class file, that is called by the Load Balancer base code to determine the load on a server. The base code provides all necessary administrative services, including starting and stopping an instance of the custom advisor, providing status and reports, recording history information in a log file, and reporting advisor results to the manager component.

When the Load Balancer base code calls a custom advisor, the following steps happen.

1. The Load Balancer base code opens a connection with the server machine.
2. If the socket opens, the base code calls the specified advisor's `GetLoad` function.
3. The advisor's `GetLoad` function performs the steps that the user has defined for evaluating the server's status, including waiting for a response from the server. The function terminates execution when the response is received.
4. The Load Balancer base code closes the socket with the server and reports the load information to the manager. Depending on whether the custom advisor operates in normal mode or in replace mode, the base code sometimes does additional calculations after the `GetLoad` function terminates.

Normal mode and replace mode

Custom advisors can be designed to interact with the Load Balancer in either *normal* mode or *replace* mode.

The choice for the mode of operation is specified in the custom advisor file as a parameter in the constructor method. (Each advisor operates in only one of these modes, based on its design.)

In normal mode, the custom advisor exchanges data with the server, and the base advisor code times the exchange and calculates the load value. The base code then reports this load value to the manager. The custom advisor returns the value zero to indicate success, or negative one to indicate an error.

To specify normal mode, set the replace flag in the constructor to *false*.

In replace mode, the base code does not perform any timing measurements. The custom advisor code performs whatever operations are specified, based on its unique requirements, and then returns an actual load number. The base code

accepts the load number and reports it, unaltered, to the manager. For best results, normalize your load numbers between 10 and 1000, with 10 representing a fast server and 1000 representing a slow server.

To specify replace mode, set the replace flag in the constructor to *true*.

Advisor naming conventions

Custom advisor file names must follow the form `ADV_name.java`, where *name* is the name that you choose for your advisor. The complete name must start with the prefix `ADV_` in uppercase letters, and all subsequent characters must be lowercase letters. The requirement for lowercase letters ensures that the command for running the advisor is not case sensitive.

According to Java conventions, the name of the class defined within the file must match the name of the file.

Compilation

You must write custom advisors in the Java language and compile them with a Java compiler that is at the same level as the Load Balancer code. To check the version of Java on your system, run the following command from the *install_path/java/bin* directory:

```
java -fullversion
```

If the current directory is not part of your path, you will need to specify that Java should be run from the current directory to ensure you are getting the correct version information. In this case, run the following command from the *install_path/java/bin* directory:

```
./java -fullversion
```

The following files are referenced during compilation:

- The custom advisor file
- The base classes file, *ibmnd.jar*, which is found in the *install_path/servers/lib* directory

Your classpath environment variable must point to both the custom advisor file and the base classes file during the compilation. A compile command might have the following format: For Microsoft Windows systems, a sample compile command is:

```
install_path/java/bin/javac -classpath /opt/ibm/edge/lb/servers/lib/ibmlb.jar ADV_name.java
```

where:

- Your advisor file is named *ADV_name.java*
- Your advisor file is stored in the current directory.

The output of the compilation is a class file, for example, *ADV_name.class*. Before starting the advisor, copy the class file to the *install_path/servers/lib/CustomAdvisors/* directory.

Note: You can compile custom advisors on one operating system and run on another operating system. For example, you can compile your advisor on a Windows system, copy the resulting class file, in binary format, to a Linux machine, and run the custom advisor there. For AIX, HP-UX, Linux, and Solaris operating systems, the syntax is similar.

Running a custom advisor

To run the custom advisor, you must first copy the advisor's class file to the `lib/CustomAdvisors/` subdirectory on the Load Balancer machine. For example, for a custom advisor named `mytyping`, the file path is `install_path/servers/lib/CustomAdvisors/ADV_mytyping.class`

Configure the Load Balancer, start its manager function, and issue the command to start your custom advisor. The custom advisor is specified by its name, excluding the `ADV_` prefix and the file extension:

```
dscontrol advisor start mytyping port_number
```

The port number specified in the command is the port on which the advisor will open a connection with the target server.

Required routines

Like all advisors, a custom advisor extends the functionality of the advisor base class, which is called `ADV_Base`. The advisor base performs most of the advisor's functions, such as reporting loads back to the manager for use in the manager's weight algorithm. The advisor base also performs socket connect and close operations and provides send and receive methods for use by the advisor. The advisor is used only for sending and receiving data on the specified port for the server that is being investigated. The TCP methods provided within the advisor base are timed to calculate load. A flag within the constructor of the advisor base overwrites the existing load with the new load returned from the advisor, if desired.

Note: Based on a value set in the constructor, the advisor base supplies the load to the weight algorithm at specified intervals. If the advisor has not completed processing and cannot return a valid load, the advisor base uses the previously reported load.

Advisors have the following base class methods:

- A constructor routine. The constructor calls the base class constructor.
- An `ADV_AdvisorInitialize` method. This method provides a way to perform additional steps after the base class completes its initialization.
- A `getLoad` routine. The base advisor class performs the socket opening; the `getLoad` function only needs to issue the appropriate send and receive requests to complete the advising cycle.

Details about these required routines appear later in this section.

Search order

Custom advisors are called after native, or standard, advisors have been searched. If the Load Balancer does not find a specified advisor among the list of standard advisors, it consults the list of custom advisors. Additional information about using advisors is available in the *WebSphere Application Server Load Balancer Administration Guide*.

Naming and file path

Remember the following requirements for custom advisor names and paths.

- The custom advisor must be named in lowercase alphabetic characters in order to eliminate case sensitivity when an operator types commands on a command line. The advisor name must be prefixed with `ADV_`

- The custom advisor class must be located within the subdirectory `lib/CustomAdvisors`. The default location for this directory is `/opt/ibm/edge/lb/servers/lib/CustomAdvisors` on Linux and UNIX systems, and `C:\Program Files\IBM\edge\lb\servers\lib\CustomAdvisors\` on Windows systems.

Custom advisor methods and function calls

Constructor (provided by advisor base)

```
public <advisor_name> (
    String sName;
    String sVersion;
    int iDefaultPort;
    int iInterval;
    String sDefaultLogFileName;
    boolean replace
)
```

sName

The name of the custom advisor.

sVersion

The version of the custom advisor.

iDefaultPort

The port number on which to contact the server if no port number is specified in the call.

iInterval

The interval at which the advisor will query the servers.

sDefaultLogFileName

This parameter is required but not used. The only acceptable value is a null string, ""

replace

Whether or not this advisor functions in *replace* mode. Possible values are the following:

- `true` – Replace the load calculated by the advisor base code with the value reported by the custom advisor.
- `false` – Add the load value reported by the custom advisor to the load value calculated by the advisor base code.

ADV_AdvisorInitialize()

```
void ADV_AdvisorInitialize()
```

This method is provided to perform any initialization that might be required for the custom advisor. This method is called after the advisor base module starts.

In many cases, including the standard advisors, this method is not used and its code consists of a *return* statement only. This method can be used to call the `suppressBaseOpeningSocket` method, which is valid only from within this method.

getLoad()

```
int getLoad(
    int iConnectTime;
    ADV_Thread *caller
)
```

iConnectTime

The length of time, in milliseconds, that it took the connection to complete.

This load measurement is performed by the advisor base code and passed to the custom advisor code, which can use or ignore the measurement when returning the load value. If the connection fails, this value is set to -1.

caller

The instance of the advisor base class where advisor base methods are provided.

Function calls available to custom advisors

The methods, or functions, described in the following sections can be called from custom advisors. These methods are supported by the advisor base code.

Some of these function calls can be made directly, for example, *function_name()*, but others require the prefix *caller*. *Caller* represents the base advisor instance that supports the custom advisor that is being executed.

ADVLOG()

The ADVLOG function allows a custom advisor to write a text message to the advisor base log file. The format follows:

```
void ADVLOG (int logLevel, String message)
```

logLevel

The status level at which the message is written to the log file. The advisor log file is organized in stages; the most urgent messages are given status level 0 and less urgent messages receive higher numbers. The most verbose type of message is given status level 5. These levels are used to control the types of messages that the user receives in real time (The **dscontrol** command is used to set verbosity). Catastrophic errors should always be logged at level 0.

message

The message to write to the log file. The value for this parameter is a standard Java string.

getAdvisorName()

The getAdvisorName function returns a Java string with the suffix portion of your custom advisor's name. For example, for an advisor named ADV_cdload.java, this function returns the value cdload.

This function takes no parameters.

Note that it is not possible for this value to change during one instantiation of an advisor.

getAdviseOnPort()

The getAdviseOnPort function returns the port number on which the calling custom advisor is running. The return value is a Java integer (int), and the function takes no parameters.

Note that it is not possible for this value to change during one instantiation of an advisor.

caller.getCurrentServerId()

The getCurrentServerId function returns a Java string which is a unique representation for the current server.

Typically, this value changes each time you call your custom advisor, because the advisor base code queries all server machines in series.

This function takes no parameters.

caller.getCurrentClusterId()

The `getCurrentClusterId` function call returns a Java string which is a unique representation for the current cluster.

Typically, this value changes each time you call your custom advisor, because the advisor base queries all clusters in series.

This function takes no parameters.

caller.getSocket()

The `getSocket` function call returns a Java socket which represents the socket opened to the current server for communication.

This function takes no parameters.

getInterval()

The `getInterval` function returns the advisor interval, that is, the number of seconds between advisor cycles. This value is equal to the default value set in the custom advisor's constructor, unless the value has been modified at run time by using the **dscontrol** command.

The return value is a Java integer (int). The function takes no parameters.

caller.getLatestLoad()

The `getLatestLoad` function allows a custom advisor to obtain the latest load value for a given server object. The load values are maintained in internal tables by the advisor base code and the manager daemon.

```
int caller.getLatestLoad (String clusterId, int port, String serverId)
```

The three arguments together define one server object.

clusterId

The cluster identifier of the server object for which to obtain the current load value. This argument must be a Java string.

port

The port number of the server object for which to obtain the current load value.

serverId

The server identifier of the server object for which to obtain the current load value. This argument must be a Java string.

The return value is an integer.

- A positive return value represents the actual load value assigned for the object that was queried.
- The value -1 indicates that the server asked about is down.
- The value -2 indicates that the status of the server asked about is unknown.

This function call is useful if you want to make the behavior of one protocol or port dependent on the behavior of another. For example, you might use this function call in a custom advisor that disabled a particular application server if the Telnet server on that same machine was disabled.

caller.receive()

The receive function gets information from the socket connection.

```
caller.receive(StringBuffer *response)
```

The parameter *response* is a string buffer into which the retrieved data is placed. Additionally, the function returns an integer value with the following significance:

- 0 indicates data was sent successfully.
- A negative number indicates an error.

caller.send()

The send function uses the established socket connection to send a packet of data to the server, using the specified port.

```
caller.send(String command)
```

The parameter *command* is a string containing the data to send to the server. The function returns an integer value with the following significance:

- 0 indicates data was sent successfully.
- A negative number indicates an error.

suppressBaseOpeningSocket()

The `suppressBaseOpeningSocket` function call allows a custom advisor to specify whether the base advisor code opens a TCP socket to the server on the custom advisor's behalf. If your advisor does not use direct communication with the server to determine its status, it might not be necessary to open this socket.

This function call can be issued only once, and it must be issued from the `ADV_AdvisorInitialize` routine.

The function takes no parameters.

Examples

The following examples show how custom advisors can be implemented.

Standard advisor

This sample source code is similar to the standard Load Balancer HTTP advisor. It functions as follows:

1. A send request, a "HEAD/HTTP" command, is issued.
2. A response is received. The information is not parsed, but the response causes the `getLoad` method to terminate.
3. The `getLoad` method returns 0 to indicate success or -1 to indicate a failure.

This advisor operates in normal mode, so the load measurement is based on the elapsed time in milliseconds required to perform the socket open, send, receive, and close operations.

```
package CustomAdvisors;
import com.ibm.internet.lb.advisors.*;
public class ADV_sample extends ADV_Base implements ADV_MethodInterface {
    static final String ADV_NAME = "Sample";
    static final int ADV_DEF_ADV_ON_PORT = 80;
    static final int ADV_DEF_INTERVAL = 7;
    static final String ADV_SEND_REQUEST =
        "HEAD / HTTP/1.0\r\nAccept: */*\r\nUser-Agent: " +
        "IBM_Load_Balancer_HTTP_Advisor\r\n\r\n";
```

```

//-----
// Constructor

public ADV_sample() {
    super(ADV_NAME, "3.0.0.0-03.31.00",
        ADV_DEF_ADV_ON_PORT, ADV_DEF_INTERVAL, "",
        false);
    super.setAdvisor( this );
}

//-----
// ADV_AdvisorInitialize

public void ADV_AdvisorInitialize() {
    return; // usually an empty routine
}

//-----
// getLoad

public int getLoad(int iConnectTime, ADV_Thread caller) {
    int iRc;
    int iLoad = ADV_HOST_INACCESSIBLE; // initialize to inaccessible

    iRc = caller.send(ADV_SEND_REQUEST); // send the HTTP request to
                                        // the server
    if (0 <= iRc) { // if the send is successful
        StringBuffer sbReceiveData = new StringBuffer(""); // allocate a buffer
                                                            // for the response
        iRc = caller.receive(sbReceiveData); // receive the result

        // parse the result here if you need to

        if (0 <= iRc) { // if the receive is successful
            iLoad = 0; // return 0 for success
        } // (advisor's load value is ignored by
        // base in normal mode)
    }
    return iLoad;
}
}

```

Side stream advisor

This sample illustrates suppressing the standard socket opened by the advisor base. Instead, this advisor opens a side stream Java socket to query a server. This procedure can be useful for servers that use a different port from normal client traffic to listen for an advisor query.

In this example, a server is listening on port 11999 and when queried returns a load value with a hexadecimal int "4". This sample runs in replace mode, that is, the last parameter of the advisor constructor is set to true and the advisor base code uses the returned load value rather than the elapsed time.

Note the call to `supressBaseOpeningSocket()` in the initialization routine. Suppressing the base socket when no data will be sent is not required. For example, you might want to open the socket to ensure that the advisor can contact the server. Examine the needs of your application carefully before making this choice.

```

package CustomAdvisors;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.Date;
import com.ibm.internet.lb.advisors.*;

```

```

import com.ibm.internet.lb.common.*;
import com.ibm.internet.lb.server.SRV_ConfigServer;

public class ADV_sidea extends ADV_Base implements ADV_MethodInterface {
    static final String ADV_NAME = "sidea";
    static final int ADV_DEF_ADV_ON_PORT = 12345;
    static final int ADV_DEF_INTERVAL = 7;

    // create an array of bytes with the load request message
    static final byte[] abHealth = {(byte)0x00, (byte)0x00, (byte)0x00,
                                     (byte)0x04};

    public ADV_sidea() {
        super(ADV_NAME, "3.0.0.0-03.31.00", ADV_DEF_ADV_ON_PORT,
              ADV_DEF_INTERVAL, "",
              true); // replace mode parameter is true
        super.setAdvisor( this );
    }

    //-----
    // ADV_AdvisorInitialize

    public void ADV_AdvisorInitialize()
    {
        suppressBaseOpeningSocket(); // tell base code not to open the
                                     // standard socket
        return;
    }

    //-----
    // getLoad

    public int getLoad(int iConnectTime, ADV_Thread caller) {
        int iRc;
        int iLoad = ADV_HOST_INACCESSIBLE; // -1
        int iControlPort = 11999; // port on which to communicate with the server

        String sServer = caller.getCurrentServerId(); // address of server to query
        try {
            socket soServer = new Socket(sServer, iControlPort); // open socket to
                                                                // server

            DataInputStream disServer = new DataInputStream(
                                     soServer.getInputStream());
            DataOutputStream dosServer = new DataOutputStream(
                                     soServer.getOutputStream());

            int iRecvTimeout = 10000; // set timeout (in milliseconds)
                                     // for receiving data
            soServer.setSoTimeout(iRecvTimeout);

            dosServer.writeInt(4); // send a message to the server
            dosServer.flush();

            iLoad = disServer.readByte(); // receive the response from the server

        } catch (exception e) {
            system.out.println("Caught exception " + e);
        }
        return iLoad; // return the load reported from the server
    }
}

```

Two port advisor

This custom advisor sample demonstrates the capability to detect failure for one port of a server based upon both its own status and on the status of a different server daemon that is running on another port on the same server machine. For

example, if the HTTP daemon on port 80 stops responding, you might also want to stop routing traffic to the SSL daemon on port 443.

This advisor is more aggressive than standard advisors, because it considers any server that does not send a response to have stopped functioning, and marks it as *down*. Standard advisors consider unresponsive servers to be very slow. This advisor marks a server as down for both the HTTP port and the SSL port based on a lack of response from either port.

To use this custom advisor, the administrator starts two instances of the advisor: one on the HTTP port, and one on the SSL port. The advisor instantiates two static global hash tables, one for HTTP and one for SSL. Each advisor tries to communicate with its server daemon and stores the results of this event in its hash table. The value that each advisor returns to the base advisor class depends on both the ability to communicate with its own server daemon and the ability of the partner advisor to communicate with its daemon.

The following custom methods are used.

- `ADV_nte()` is a simple container object to hold information about a server. These objects are stored in the hash table as table elements. Each object has a time stamp that is used to determine whether the element is current.
- `putNte()` and `getNte()` are synchronized methods that ensure that the two advisor instances access the hash table in a controlled fashion.
- `getLoadHTTP` is a method that queries the responsiveness of an HTTP server. It is a low-level routine and does not gather or use information about SSL.
- `getLoadSSL()` is a method that queries the responsiveness of an SSL server. It is a low-level routine and does not gather or use information about HTTP.
- `getLoad()` is the entry point routine for this custom advisor. It can handle both protocols and can store and fetch information from the hash table. This is the routine that links the two ports.

The following error conditions are detected.

- Unresponsive server machine — The base advisor classes periodically send a ping signal to the server address. If the address is not reachable, the base advisor classes marks the server down. Neither of the two instances of the custom advisor is called, and both servers on that machine are marked down.
- One daemon on a server machine becomes unresponsive, but the other is working — When the base code attempts to open a socket with the server, the connection is refused, and the base advisor for this protocol marks the server as down. The custom advisor code for that protocol is not called. Although the custom advisor for the other protocol continues communicating with its server, it learns from the hash table that the other custom advisor cannot communicate with its server daemon. Therefore, the second protocol's advisor also marks its server as down.
- One daemon does not send a response, but the other daemon does — The custom advisor for the unresponding protocol detects the failure to communicate, marks the server as down, and stores the data in the hash table. The custom advisor for the other port learns that information from the hash table and marks its server as down.

This sample is written to link ports 80 for HTTP and 443 for SSL, but it can be tailored to any combination of ports.

```

package CustomAdvisors;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.Date;
import com.ibm.internet.lb.advisors.*;
import com.ibm.internet.lb.common.*;
import com.ibm.internet.lb.manager.*;
import com.ibm.internet.lb.server.SRV_ConfigServer;

//-----
// Define the table element for the hash tables used in this custom advisor

class ADV_nte implements Cloneable {
    private String  sCluster;
    private int     iPort;
    private String  sServer;
    private int     iLoad;
    private Date    dTimestamp;

//-----
// constructor

    public ADV_nte(String sClusterIn, int iPortIn, String sServerIn,
                    int iLoadIn) {
        sCluster = sClusterIn;
        iPort = iPortIn;
        sServer = sServerIn;
        iLoad = iLoadIn;
        dTimestamp = new Date();
    }

//-----
// check whether this element is current or expired
    public boolean isCurrent(ADV_twop oThis) {
        boolean bCurrent;
        int iLifetimeMs = 3 * 1000 * oThis.getInterval(); // set lifetime as
                                                         // 3 advisor cycles

        Date dNow = new Date();
        Date dExpires = new Date(dTimestamp.getTime() + iLifetimeMs);

        if (dNow.after(dExpires)) {
            bCurrent = false;
        } else {
            bCurrent = true;
        }
        return bCurrent;
    }

//-----
// value accessor(s)

    public int getLoadValue() { return iLoad; }

//-----
// clone (avoids corruption between threads)

    public synchronized Object Clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}

```

```

//-----
// define the custom advisor

public class ADV_twop extends ADV_Base
    implements ADV_MethodInterface, ADV_AdvisorVersionInterface {

    static final int ADV_TWOP_PORT_HTTP = 80;
    static final int ADV_TWOP_PORT_SSL = 443;

    //-----
    // define tables to hold port-specific history information

    static Hashtable htTwopHTTP = new Hashtable();
    static Hashtable htTwopSSL = new Hashtable();

    static final String ADV_TWOP_NAME = "twop";
    static final int ADV_TWOP_DEF_ADV_ON_PORT = 80;
    static final int ADV_TWOP_DEF_INTERVAL = 7;
    static final String ADV_HTTP_REQUEST_STRING =
        "HEAD / HTTP/1.0\r\nAccept: */*\r\nUser-Agent: " +
        "IBM_LB_Custom_Advisor\r\n\r\n";

    //-----
    // create byte array with SSL client hello message

    public static final byte[] abClientHello = {
        (byte)0x80, (byte)0x1c,
        (byte)0x01,           // client hello
        (byte)0x03, (byte)0x00, // SSL version
        (byte)0x00, (byte)0x03, // cipher spec len (bytes)
        (byte)0x00, (byte)0x00, // session ID len (bytes)
        (byte)0x00, (byte)0x10, // challenge data len (bytes)
        (byte)0x00, (byte)0x00, (byte)0x03, // cipher spec
        (byte)0x1A, (byte)0xFC, (byte)0xE5, (byte)0x20, // challenge data
        (byte)0xFD, (byte)0x3A, (byte)0x3C, (byte)0x18,
        (byte)0xAB, (byte)0x67, (byte)0xB0, (byte)0x52,
        (byte)0xB1, (byte)0x1D, (byte)0x55, (byte)0x44, (byte)0x0D, (byte)0x0A };

    //-----
    // constructor

    public ADV_twop() {
        super(ADV_TWOP_NAME, VERSION, ADV_TWOP_DEF_ADV_ON_PORT,
            ADV_TWOP_DEF_INTERVAL, "",
            false); // false = load balancer times the response
        setAdvisor ( this );
    }

    //-----
    // ADV_AdvisorInitialize

    public void ADV_AdvisorInitialize() {
        return;
    }

    //-----
    // synchronized PUT and GET access routines for the hash tables

    synchronized ADV_nte getNte(Hashtable ht, String sName, String sHashKey) {
        ADV_nte nte = (ADV_nte)(ht.get(sHashKey));
        if (null != nte) {
            nte = (ADV_nte)nte.clone();
        }
        return nte;
    }

    synchronized void putNte(Hashtable ht, String sName, String sHashKey,
        ADV_nte nte) {

```

```

        ht.put(sHashKey, nte);
        return;
    }

    //-----
    // getLoadHTTP - determine HTTP load based on server response

    int getLoadHTTP(int iConnectTime, ADV_Thread caller) {
        int iLoad = ADV_HOST_INACCESSIBLE;

        int iRc = caller.send(ADV_HTTP_REQUEST_STRING); // send request message
                                                         // to server
        if (0 <= iRc) { // did the request return a failure?
            StringBuffer sbReceiveData = new StringBuffer("") // allocate a buffer
                                                                // for the response
            iRc = caller.receive(sbReceiveData); // get response from server

            if (0 <= iRc) { // did the receive return a failure?
                if (0 < sbReceiveData.length()) { // is data there?
                    iLoad = SUCCESS; // ignore retrieved data and
                                     // return success code
                }
            }
        }
        return iLoad;
    }

    //-----
    // getLoadSSL() - determine SSL load based on server response

    int getLoadSSL(int iConnectTime, ASV_Thread caller) {
        int iLoad = ADV_HOST_INACCESSIBLE;
        int iRc;

        CMNByteArrayWrapper cbawClientHello = new CMNByteArrayWrapper(
            abClientHello);
        Socket socket = caller.getSocket();

        try {
            socket.getOutputStream().write(abClientHello);
            // Perform a receive.
            socket.getInputStream().read();
            // If receive is successful, return load of 0. We are not concerned with
            // data's contents, and the load is calculated by the ADV_Thread thread.
            iLoad = 0;
        } catch (IOException e) {
            // Upon error, iLoad will default to it.
        }
        return iLoad;
    }

    //-----
    // getLoad - merge results from the HTTP and SSL methods

    public int getLoad(int iConnectTime, ADV_Thread caller) {
        int iLoadHTTP;
        int iLoadSSL;
        int iLoad;
        int iRc;

        String sCluster = caller.getCurrentClusterId(); // current cluster address
        int iPort = getAdviseOnPort();
        String sServer = caller.getCurrentServerId();
        String sHashKey = sCluster + ":" + sServer; // hash table key

        if (ADV_TWOP_PORT_HTTP == iPort) { // handle an HTTP server
            iLoadHTTP = getLoadHTTP(iConnectTime, caller); // get the load for HTTP

```

```

ADV_nte nteHTTP = newADV_nte(sCluster, iPort, sServer, iLoadHTTP);
putNte(htTwopHTTP, "HTTP", sHashKey, nteHTTP); // save HTTP load
// information
ADV_nte nteSSL = getNte(htTwopSSL, "SSL", sHashKey); // get SSL
// information
if (null != nteSSL) {
    if (true == nteSSL.isCurrent(this)) { // check the time stamp
        if (ADV_HOST_INACCESSIBLE != nteSSL.getLoadValue()) { // is SSL
                                                                // working?
            iLoad = iLoadHTTP;
        } else { // SSL is not working, so mark the HTTP server down
            iLoad = ADV_HOST_INACCESSIBLE;
        }
    } else { // SSL information is expired, so mark the
              // HTTP server down
        iLoad = ADV_HOST_INACCESSIBLE;
    }
} else { // no load information about SSL, report
        // getLoadHTTP() results
    iLoad = iLoadHTTP;
}
}
else if (ADV_TWOP_PORT_SSL == iPort) { // handle an SSL server
    iLoadSSL = getLoadSSL(iConnectTime, caller); // get load for SSL

    ADV_nte nteSSL = new ADV_nte(sCluster, iPort, sServer, iLoadSSL);
    putNte(htTwopSSL, "SSL", sHashKey, nteSSL); // save SSL load info.

    ADV_nte nteHTTP = getNte(htTwopHTTP, "SSL", sHashKey); // get HTTP
                                                            // information
    if (null != nteHTTP) {
        if (true == nteHTTP.isCurrent(this)) { // check the timestamp
            if (ADV_HOST_INACCESSIBLE != nteHTTP.getLoadValue()) { // is HTTP
                                                                    // working?
                iLoad = iLoadSSL;
            } else { // HTTP server is not working, so mark SSL down
                iLoad = ADV_HOST_INACCESSIBLE;
            }
        } else { // expired information from HTTP, so mark SSL down
            iLoad = ADV_HOST_INACCESSIBLE;
        }
    } else { // no load information about HTTP, report
              // getLoadSSL() results
        iLoad = iLoadSSL;
    }
}
}

//-----
// error handler

else {
    iLoad = ADV_HOST_INACCESSIBLE;
}
return iLoad;
}
}

```

WebSphere Application Server advisor

A sample custom advisor for WebSphere Application Server is included in the *install_path/servers/samples/CustomAdvisors/* directory. The full code is not duplicated in this document.

- ADV_was.java is the advisor source code file that is compiled and run on the Load Balancer machine.

- `LBAdvisor.java.servlet` is the servlet source code that must be renamed to `LBAdvisor.java`, compiled, and run on the WebSphere Application Server machine.

The complete advisor is only slightly more complex than the sample. It adds a specialized parsing routine that is more compact than the `StringTokenizer` example shown above.

The more complex part of the sample code is in the Java servlet. Among other methods, the servlet contains two methods required by the servlet specification: `init()` and `service()`, and one method, `run()`, that is required by the `Java.lang.thread` class.

- `init()` is called once by the servlet engine at initialization time. This method creates a thread named `_checker` that runs independently of calls from the advisor and sleeps for a period of time before resuming its processing loop.
- `service()` is called by the servlet engine each time the servlet is invoked. In this case, the method is called by the advisor. The `service()` method sends a stream of ASCII characters to an output stream.
- `run()` contains the core of the code execution. It is called by the `start()` method that is called from within the `init()` method.

The relevant fragments of the servlet code appear below.

...

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    ...
    _checker = new Thread(this);
    _checker.start();
}

public void run() {
    setStatus(GOOD);

    while (true) {
        if (!getKeepRunning())
            return;
        setStatus(figureLoad());
        setLastUpdate(new java.util.Date());

        try {
            _checker.sleep(_interval * 1000);
        } catch (Exception ignore) { ; }
    }
}

public void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    ServletOutputStream out = null;
    try {
        out = res.getOutputStream();
    } catch (Exception e) { ... }
    ...
    res.setContentType("text/x-application-LBAdvisor");
    out.println(getStatusString());
    out.println(getLastUpdate().toString());
    out.flush();
    return;
}

...
```

Using data returned from advisors

Whether you use a standard call to an existing part of the application server or add a new piece of code to be the server-side counterpart of your custom advisor, you possibly want to examine the load values returned and change server behavior. The Java StringTokenizer class, and its associated methods, make this investigation easy to do.

The content of a typical HTTP command might be GET /index.html HTTP/1.0

A typical response to this command might be the following.

```
HTTP/1.1 200 OK
Date: Mon, 20 November 2000 14:09:57 GMT
Server: Apache/1.3.12 (Linux and UNIX)
Content-Location: index.html.en
Vary: negotiate
TCN: choice
Last-Modified: Fri, 20 Oct 2000 15:58:35 GMT
ETag: "14f3e5-1a8-39f06bab;39f06a02"
Accept-Ranges: bytes
Content-Length: 424
Connection: close
Content-Type: text/html
Content-Language: en

<!DOCTYPE HTML PUBLIC "-//w3c//DTD HTML 3.2 Final//EN">
<HTML><HEAD><TITLE>Test Page</TITLE></HEAD>
<BODY><H1>Apache server</H1>
<HR>
<P><P>This Web server is running Apache 1.3.12.
<P><HR>
<P><IMG SRC="apache_pb.gif" ALT="">
</BODY></HTML>
```

The items of interest are contained in the first line, specifically the HTTP return code.

The HTTP specification classifies return codes that can be summarized as follows:

- 2xx return codes are successes
- 3xx return codes are redirections
- 4xx return codes are client errors
- 5xx return codes are server errors

If you know very precisely what codes the server can possibly return, your code might not need to be as detailed as this example. However, keep in mind that limiting the return codes you detect might limit the future flexibility of your program.

The following example is a stand-alone Java program that contains a minimal HTTP client. The example invokes a simple, general-purpose parser for examining HTTP responses.

```
import java.io.*;
import java.util.*;
import java.net.*;

public class ParseTest {
    static final int iPort = 80;
    static final String sServer = "www.ibm.com";
    static final String sQuery = "GET /index.html HTTP/1.0\r\n\r\n";
    static final String sHTTP10 = "HTTP/1.0";
    static final String sHTTP11 = "HTTP/1.1";
```

```

public static void main(String[] Arg) {
    String sHTTPVersion = null;
    String sHTTPReturnCode = null;
    String sResponse = null;
    int iRc = 0;
    BufferedReader brIn = null;
    PrintWriter psOut = null;
    Socket soServer= null;
    StringBuffer sbText = new StringBuffer(40);

    try {
        soServer = new Socket(sServer, iPort);
        brIn = new BufferedReader(new InputStreamReader(
            soServer.getInputStream()));
        psOut = new PrintWriter(soServer.getOutputStream());
        psOut.println(sQuery);
        psOut.flush();
        sResponse = brIn.readLine();
        try {
            soServer.close();
        } catch (Exception sc) {}
    } catch (Exception swr) {}

    StringTokenizer st = new StringTokenizer(sResponse, " ");
    if (true == st.hasMoreTokens()) {
        sHTTPVersion = st.nextToken();
        if (sHTTPVersion.equals(sHTTP110) || sHTTPVersion.equals(sHTTP11)) {
            System.out.println("HTTP Version: " + sHTTPVersion);
        } else {
            System.out.println("Invalid HTTP Version: " + sHTTPVersion);
        }
    } else {
        System.out.println("Nothing was returned");
        return;
    }

    if (true == st.hasMoreTokens()) {
        sHTTPReturnCode = st.nextToken();
        try {
            iRc = Integer.parseInt(sHTTPReturnCode);
        } catch (NumberFormatException ne) {}

        switch (iRc) {
            case(200):
                System.out.println("HTTP Response code: OK, " + iRc);
                break;
            case(400): case(401): case(402): case(403): case(404):
                System.out.println("HTTP Response code: Client Error, " + iRc);
                break;
            case(500): case(501): case(502): case(503):
                System.out.println("HTTP Response code: Server Error, " + iRc);
                break;
            default:
                System.out.println("HTTP Response code: Unknown, " + iRc);
                break;
        }
    }

    if (true == st.hasMoreTokens()) {
        while (true == st.hasMoreTokens()) {
            sbText.append(st.nextToken());
            sbText.append(" ");
        }
    }
}

```



```
        System.out.println("HTTP Response phrase: " + sbText.toString());  
    }  
}
```

Index

A

- ADV_AdvisorInitialize() 40, 41
- ADV_Base 40
- advisor 1, 36
 - custom 38
 - library functions 40
 - naming conventions 39
 - standard 37
- advisor constructor 41
- advisor cycle 37
- ADVLOG() 42
- API functions
 - Caching Proxy 15
- authentication 33
 - calling plug-ins for Basic type only 22
 - configuration file directive 22
 - function prototype 10
 - proxy server step 6
 - using the Caching Proxy plug-in API 35
- authorization 33
 - configuration file directive 23
 - function prototype 10
 - proxy server step 6
 - using the Caching Proxy plug-in API 35

C

- caching
 - variant 36
- Caching Proxy plug-in API
 - compiling programs 7
 - configuration directives 21
 - configuration file directives 22
 - order for different processing steps 22
 - order for one processing step 22
 - order for Service and Name Translation processing steps 22
 - function prototypes 8
 - guidelines for writing programs 7
 - overview 3
 - procedure for writing programs 3
- Caching Proxy plug-in functions
 - calling for particular requests only 22
- Caching Proxy steps 4
- caller.getCurrentServerId() 42
- caller.getLatestLoad() 43
- caller.receive() 43
- caller.send() 44
- CGI programs
 - porting to the Caching Proxy plug-in API 24
- code samples 2, 36
- compiling
 - Caching Proxy plug-in API programs 7

- compiling (*continued*)
 - custom advisors 39
- configuration file directives (Caching Proxy) 22
- constructor 40
- custom advisor 38
 - constructor 41
 - library functions 40
 - naming conventions 39
- custom advisor modes 38
- custom advisors 1, 36

E

- error
 - configuration file directive 23
 - function prototype 13
 - proxy server step 7
- examples
 - for the Caching Proxy plug-in API 36
- examples (*See also* sample code) 2
 - custom advisors 44

G

- GC advisor
 - configuration file directive 23
 - function prototype 13
 - proxy server step 6
- getAdviseOnPort() 42
- getAdvisorName() 42
- getCurrentServerId() 42
- getInterval() 43
- getLatestLoad() 43
- getLoad() 38, 40, 41
- guidelines for Caching Proxy plug-in API programs 7
- GWAPI 24

H

- HTTP return codes 14
 - for Caching Proxy plug-in API functions 14
- HTTPD_authenticate() 15, 35, 36
- HTTPD_cacheable_url() 16
- HTTPD_close() 16
- HTTPD_exec() 16
- HTTPD_extract() 16
- HTTPD_file() 17
- httpd_getvar() 17
- HTTPD_log_access() 17
- HTTPD_log_error() 17
- HTTPD_log_event() 17
- HTTPD_log_trace() 18
- HTTPD_open() 18
- HTTPD_proxy() 18
- HTTPD_read() 18
- HTTPD_restart() 19

- HTTPD_set() 19
- httpd_setvar() 19
- httpd_variant_insert() 20, 36
- httpd_variant_lookup() 20, 36
- HTTPD_write() 20

I

- ibmnd.jar file 39
- ibmproxy.conf file 21, 22
- ICAPI 24
- iConnectTime 41

L

- library functions
 - Caching Proxy plug-in API (*See also* HTTPD_*) 15
 - Load Balancer custom advisors 40
- Load Balancer advisors 1, 36
- log
 - configuration file directive 23
 - function prototype 13
 - proxy server step 7

M

- method handler 11
- midnight
 - configuration file directive 23
 - function prototype 10
 - proxy server step 6

N

- name translation
 - configuration file directive 23
 - function prototype 10
 - proxy server step 6
- naming conventions for custom advisors 39
- normal mode 38

O

- object type
 - configuration file directive 23
 - function prototype 10
 - proxy server step 6

P

- porting CGI programs for the Caching Proxy plug-in API 24
- post authorization
 - function prototype 11
 - proxy server step 6

- postAuthorization
 - configuration file directive 23
- postExit
 - configuration file directive 23
 - function prototype 13
 - proxy server step 7
- predefined functions
 - Caching Proxy 15
- preExit
 - configuration file directive 22
 - function prototype 9
 - proxy server step 6
- proxy advisor
 - configuration file directive 23
 - function prototype 13
 - proxy server step 6
- proxy configuration file modifications for plug-ins 21

R

- receive() 43
- replace mode 38
- return codes
 - for Caching Proxy plug-in API library functions 21
 - HTTP 14

S

- sample code 2
 - custom advisors 2, 44
 - for the Caching Proxy plug-in API 2, 36
 - processing returned advisor data 53
 - side stream advisor 45
 - standard advisor 44
 - two-port advisor 46
 - WebSphere Application Server advisor 51
- search order
 - for Load Balancer advisors 40
- send() 44
- server initialization
 - configuration file directive 22
 - function prototype 9
 - proxy server step 6
- server process
 - steps 4
- server request process
 - steps 4
- server termination
 - configuration file directive 23
 - function prototype 14
 - proxy server step 7
- service
 - configuration file directive 23
 - function prototype 11
 - proxy server step 6
- side stream advisor
 - code sample 45
- standard advisor 37
 - code sample 44
- steps
 - Caching Proxy 4
- suppressBaseOpeningSocket() 44

- suppressBaseOpeningSocket() *(continued)*
 - example 45
- system plug-ins (Caching Proxy) 22

T

- transmogrifier
 - configuration file directive 23
 - function prototype 11
 - proxy server step 6
- two-port advisor
 - code sample 46

U

- URL template for Caching Proxy plug-in API directives 23

V

- variant caching 36

W

- WebSphere Application Server
 - custom advisor code sample 51



Printed in USA

Spine information:



WebSphere Application Server

Programming Guide for Edge Components

Version 7.0