Web Request Broker™ Programmer's Reference

Release 3.0

ORACLE®

Enabling the Information Age

Web Request Broker™ Programmer's Reference, 3.0

Primary Author: Kennan Rossi

Contributors: Seshu Adunuthula, Mala Anand, Jeffrey Caesar, Elaine Chien, Sam Chou, John Darrow, Jim Stabile

Contents

CHAPTER

1 Overview of the Web Request Broker (WRB)

The Web Request Broker is the central component of the Oracle Web Application Server. It is an asynchronous request handler with an API (Application Program Interface) that enables it to interface dynamically and seamlessly to various back-end technologies called “WRB Cartridges”. It provides an architecture that allows server-side web applications to run under any HTTP server to which the WRB has been ported.

The Web Request Broker offers capabilities similar to CGI scripting. In fact, you can use the WRB API to access CGI data sent by the client. WRB applications, called *cartridges*, however, take advantage of the WRB’s multi-process architecture to get much better performance than ordinary CGI scripts.

The WRB architecture also makes WRB cartridges extremely scalable—that is, they can handle small request loads economically and large request loads efficiently.

Whenever the Web Listener receives a URL that calls for the WRB, it passes execution of the request to the *Dispatcher*. The Dispatcher determines for which cartridge the request is intended and directs the WRB to allocate an execution instance (WRBX) of the cartridge. The Dispatcher then calls the WRBX to handle the request. The result is that the Listener can receive and validate URLs coming in, while each request is handed off to a process that executes it in the background.

WRB cartridges can be of the following types:

- The PL/SQL Cartridge, which executes PL/SQL commands stored in the database. It is better optimized for database access than the Java Cartridge, but it does not have all of Java’s functionality.
- The Java Cartridge, which enables you to execute Java on the server to generate dynamic Web pages. You can also execute PL/SQL from within Java using this cartridge.
- LiveHTML Cartridge, which is Oracle’s implementation and extension of the industry-standard Server Side Includes functionality. *LiveHTML* enables you to

include in your Web pages the output of any program that your Operating System can execute.

- Your Own Cartridges. Since the WRB uses an open API, you can write your own cartridges to use it. Currently, you use the C language to write WRB cartridges. In the future, however, the WRB API will also be available in other languages. Instruction and reference on writing WRB cartridges is available in the application development section of the online documentation.
- Third-Party Cartridges. Since the WRB uses an open API, various independent vendors are writing cartridges for it.

WRB Architecture

The WRB architecture consists of these components:

- The Dispatcher
- The Web Request Broker
- WRB cartridges
- The WRB application engine
- WRB execution instances (WRBXs)

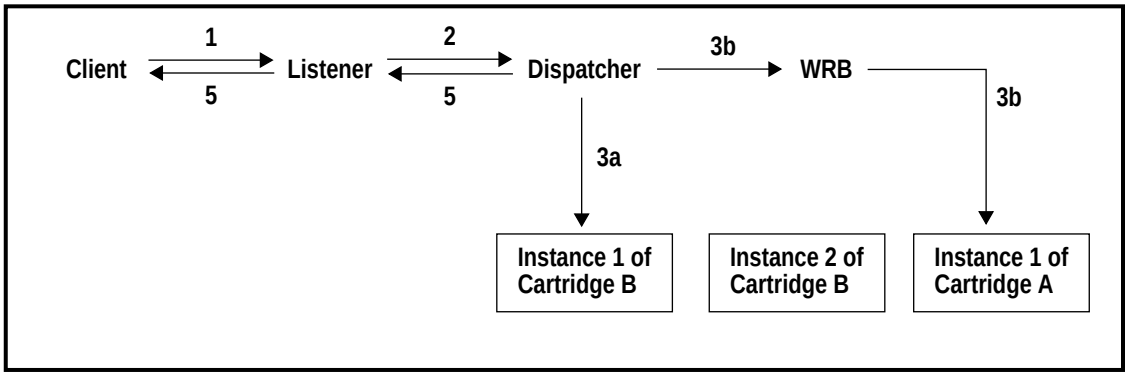
A WRB cartridge is implemented as a shared library that uses the WRB API to handle HTTP requests from web clients.

The *Dispatcher* is a program that provides a CORBA interface between Web Listeners, the Web Request Broker, and WRB cartridges. The Dispatcher manages WRB cartridges and the requests directed to them. When the Web Listener receives an HTTP request directed to a WRB cartridge, it forwards the request to the Dispatcher, which uses its configuration data to map URLs to WRB cartridges.

An execution instance of a WRB cartridge is a process running a program composed of two parts: a copy of the WRB application engine, and the WRB cartridge shared library.

The *WRB application engine* is the executable program that implements the WRB API and runs within each WRBX. It provides the interface between WRB cartridges and the Dispatcher, directs WRB cartridge flow of control, and provides services for WRB cartridges to use.

The Dispatcher then dispatches the request to that WRBX.



The diagram above illustrates how the Web Application Server responds to a request for a cartridge:

1. The client issues a request directed to the WRB cartridge.
2. The Listener receives the request and passes it to the Dispatcher, whose task is to forward the request to a cartridge instance.
3. The Dispatcher does one of two things at this point:

   a. If it knows about an unoccupied cartridge instance, it sends the request to that instance.

   b. If there are no unoccupied cartridge instances, the Dispatcher asks the WRB to create a new cartridge instance. After the instance starts up successfully, it notifies the WRB of its existence, and the WRB notifies the Dispatch of the new instance. The Dispatch then sends the request to the new instance.
4. The cartridge instance handles the request and sends a response directed to the client to the Dispatcher.
5. The Dispatcher passes the response back through the Listener to the client.

The Dispatcher may then forward future requests directly to the cartridge instance and receive responses directly from it. (This behavior is restricted by the Sessions mechanism; see Sessions below.)

The WRB configuration data specifies for each cartridge the maximum number of WRBXs that may run at once, and the minimum number of WRBXs that must always be running. You can tune your WRB performance by adjusting these values using the Web Request Broker administration pages.

The WRB API

The WRB API defines:

- Cartridge functions—the interfaces the WRB application engine uses to call your application.
- WRB application engine functions—the interfaces the WRB application engine provides for your application to use.

To make your cartridge functions available to the WRB application engine, you store pointers to the cartridge functions your application defines in a function table (see The Entry-Point Function in Writing Applications Using the Web Request Broker API in the online documentation).

In addition to the core WRB API functions, the WRB application engine functions also include these groups of functions, called *services*:

- Content service—provides a common interface for storing web content in the local file system or a database.
- Intercartridge exchange (ICX) service—provides a mechanism that cartridges can use to issue requests to other cartridges.
- Transaction service—provides a mechanism for managing database transactions.
- Logger service—provides a mechanism for logging information to database documents or ordinary files.

The WRB application engine functions are declared in the header **wrb.h**. The function names all begin with "WRB". APIs belonging to services are declared in other header files, and these files are included by **wrb.h**.

The Content Service

The content service regulates access to databases, referred to generically as *content repositories*.

The content service allows you to make changes to a local copy of a document in a repository and apply those changes to the original document whenever you choose.

Note: Currently, you cannot use the Transaction Service with the Content Service.

See the *WRB Content Service API Reference* for more information.

The Intercartridge Exchange Service

The intercartridge exchange service allows cartridges to issue HTTP requests to other cartridges on the same machine or on remote machines. The intercartridge exchange service uses a CORBA architecture to make intercartridge requests more efficient than ordinary HTTP requests.

This mechanism allows you to design cartridges to provide common functionality to other cartridges, minimizing redundant code, and when necessary, allowing centralized resource management.

For more information, see *WRB Architecture* in *Writing Applications Using the Web Request Broker API* and the *WRB Intercartridge Exchange Service API Reference* in the online documentation.

The Transaction Service

The transaction service provides an environment for performing database transactions that span several HTTP requests to a cartridge. This allows you to maintain a kind of "connection" between a web client and a database.

To perform actual database access, you must use a database access API such as OCI or Pro*C in conjunction with the transaction service. You must not use the database access API's mechanisms for committing or rolling back transactions, however—you must use the transaction service for these operations.

See the *WRB Transaction Service API Reference* in the Oracle Web Application Server online documentation for more information.

Sessions

The WRB architecture provides a mechanism for associating a web client with a particular WRBX. This association, called a *session*, allows WRB cartridge developers to write cartridges that maintain client state data that remains valid from request to request, because each session-enabled cartridge execution instance receives requests from one and only one client.

See *Maintaining Persistent Client State* in *Writing Applications Using the Web Request Broker API* in the online documentation for more information.

WRB Cartridge Security

WRB cartridges can augment the access control specified by the Listener with cartridge-specific control. The cartridges included with the Web Application Server do not do this, but the WRB API provides a callback routine that enables you to put this functionality in cartridges that you write. This routine enables your cartridge to:

- Prompt for a username and password, using either a Basic or a Digest scheme. The difference between Basic and Digest authentication is that Digest encrypts passwords (for more information on these terms, see *Introduction to Oracle Web Application Server*).
- Specify by name some restriction scheme to be applied to this request. Such restriction schemes are defined in the Listener and limit access by IP address or Domain Name.
- Get the client-side certificate. See `WRB_getClientCert()` for details.

For more information on how WRB cartridges work and how to write your own, refer to the WRB specification that accompanies the Web Application Server online documentation.

CHAPTER

2

Cartridge Development

This section covers how to develop your own custom cartridges. You can find the following information in this section:

- To get architecture and introductory information about writing your own cartridges, see:
  - Overview of the Web Request Broker (WRB)
  - Writing Applications Using the Web Request Broker API
  - Using the WRB Core APIs
- For sample cartridges, see:
  - The MyWRBApp Sample Cartridge
  - Document Server Cartridge Example
- For an example of how to use the Intercartridge Exchange (ICX) Service (which enables one cartridge to communicate with another cartridge), see Intercartridge Exchange Service Sample Code Fragment.
- For an example of how to use the WRB Transaction Service, see The Hello World Sample Cartridge.

The following sections document all the WRB APIs.

- Web Request Broker Core API Reference
- WRB Content Service API Reference
- WRB Intercartridge Exchange Service API Reference

- WRB Transaction Service API Reference
- WRB Logger Service API Reference

CHAPTER

3 Writing Applications Using the Web Request Broker API

This document describes the steps involved in writing a server-side web application, called a *WRB cartridge*, using the Web Request Broker (WRB) API.

Contents

For a general overview of the Web Request Broker architecture under which cartridges run, see Overview of the Web Request Broker (WRB).

Designing a WRB Cartridge

This section discusses concepts you should consider before starting to write your WRB cartridge, such as common WRB cartridge programming models.

Common WRB Cartridge Programming Models

The WRB architecture lends itself to three basic approaches to cartridge design:

- The Request-Response Model
- The Session Model
- The Transaction Model

The Request-Response Model

WRB cartridges in this model respond to client requests individually. After the cartridge responds with the requested content, all data associated with the client and the request is lost. This is consistent with the "statelessness" of ordinary HTTP requests.

To write a cartridge of this type, you can code your Exec function to identify the request, respond with the requested content, and return without saving any state.

The Session Model

WRB cartridges in this model use the WRB sessions mechanism to maintain a persistent association between clients and particular execution instances (WRBXs) of the cartridge. Each association persists until the WRBX has been idle for a configurable time-out period.

In writing a cartridge of this type, you can code your Exec function to save client-related state data in your application context structure, which is local to each cartridge instance. Because the sessions mechanism guarantees an association between client and WRBX, you can code the cartridge as though it is interacting with a single client. For example, if you maintain a client-specific counter in your application context structure, the value of the counter may be different for each cartridge instance, and will be correctly maintained for each client. In contrast, if your cartridge uses the request / reply model, such a counter will be global to all cartridge instances.

Note that in both the request-response and the session models, you can save client data in the application context structure of the cartridge instance. If you are using the request-response mode, however, it does not make sense to do so because the next request from the same client may go to another cartridge instance.

See Maintaining Persistent Client State below for more information.

The Transaction Model

WRB cartridges in this model typically perform database transactions.

The Exec function for a cartridge of this type typically goes through the following states as it handles an incoming request:

1. Receive the request
2. Begin the transaction

3. Perform incremental updates within the transaction context
4. Commit or rollback the transaction (this ends the transaction)
5. Send a response to the client

A transaction can span several calls to the Exec function. To write a cartridge of this type, you must code your Exec function to determine which of the above states currently applies. For example, your Exec function might begin a transaction when the request URI contains the path element “begin” and commit the transaction when the request URI contains the path element “commit”.

If you write this type of cartridge, your Exec function must call `tx_annotate_path()` when handling each request within a transaction. This sends the current transaction ID to the client in the form of a cookie, specifying a virtual path corresponding to the next anticipated request in the transaction.

When your Exec function is executing in a transaction context, the transaction is suspended when Exec returns. When the client subsequently issues a request with a virtual path matching a path specified by the call `tx_annotate_path()`, the client sends the transaction ID in the header of the request. The WRB then resumes the transaction context before calling the cartridge Exec function to handle the request. See `tx_annotate_path()` for more information.

If the client issues a request to your cartridge that does not match a path specified by a call to `tx_annotate_path()`, it does not send the transaction ID in the request header. In this case, the WRB does not resume any transaction context before calling your Exec function. This allows you to support requests that are independent of any transaction.

See the WRB Transaction Service API Reference for more information.

Maintaining Persistent Client State

Under previous versions of the WRB, cartridges had to encode client state information in HTTP cookies and send them to client browsers. The browsers then included the cookies in the headers of future requests to the cartridge, which the cartridge then had to parse.

The WRB 3.0 API greatly simplifies maintaining client state by introducing *sessions*.

When a client sends a request to a *session-enabled* cartridge, the Dispatcher establishes a session between the client and a particular instance of the cartridge. All subsequent requests that the client sends to the cartridge within a certain time-out period are directed to the same cartridge instance.

If your cartridge is session-enabled, you can store client state data locally in your application context structure. Because each cartridge instance is bound to a client, this data will remain valid from request to request. If the client issues no requests to the cartridge for the specified time-out period, the session ends and the cartridge instance terminates.

When you configure your cartridge (see Configuring Your Cartridge), you can enable sessions for your cartridge and set the time-out period.

The MyWRBApp Sample Cartridge illustrates the use of sessions to maintain persistent client state.

Distributing WRB Cartridges Across Machines

To enhance the scalability of WRB cartridges, the WRB 3.0 architecture allows requests received by a Web Listener on one machine to be dispatched to WRB cartridge instances running on other machines. It is important that cartridges that access the local file system do so using path names relative to $ORAWEB_HOME, so that cartridge instances will work correctly regardless of the machine on which they are running.

Note that this feature of running cartridges on other machines is available only in the Advanced version of the Web Application Server; it is not available in the Standard version.

The ways in which WRB cartridges are distributed across machines are determined by the WRB configuration data on all the machines involved. See the Applications, Protocols, and Hosts section of the WRB administration pages for more information.

Distributed Processing Using Intercartridge Exchange

In addition to enhancing scalability, the WRB 3.0 API provides the Intercartridge Exchange service, which allows you to distribute computation among WRB cartridges and across machines. This means you can write a cartridge that implements common functionality that can be shared by other cartridges.

For example, you might want to define a registration cartridge that records personal data about clients before forwarding their requests to other cartridges.

For more information, see the WRB Intercartridge Exchange Service API Reference.

Choosing a WRB Cartridge Authentication Mechanism

As with the WRB 2.x API, cartridges using the WRB 3.0 API can authenticate clients themselves by defining new basic or digest authentication realms and checking client passwords for these realms. See The Authorize Function, `WRB_setAuthBasic()`, `WRB_setAuthDigest()` for more information.

In addition, the WRB 3.0 API also provides an authentication server that can handle client authentication for your cartridge, as well as restrict access according to the client's IP address or DNS domain. The authentication server can also perform basic database user authentication for cartridges that access a database. See The Init Function and `WRB_setAuthServer()` for more information.

Upgrading from the WRB 2.x API to the WRB 3.0 API

This section outlines how to upgrade a WRB cartridge developed using the WRB 2.0 or 2.1 WRB API to take advantage of WRB 3.0 API features. For backward compatibility, the WRB 2.x API is still supported under Web Application Server 3.0, but this support will be removed in a future release.

The following table maps WRB 2.x API calls to WRB 3.0 API calls. Note that this is not necessarily a direct mapping; see the referenced WRB 3.0 API call in each case to learn the new syntax.

Instead of this WRB 2.x API call	Use this WRB 3.0 API call
WRBGetORACLE_HOME()	WRB_getORACLE_HOME()
WRBGetAppConfig()	WRB_getAppConfigSection()
WRBGetConfigVal()	WRB_getAppConfigVal()
WRBGetURI()	WRB_getRequestInfo()
WRBGetURL()	WRB_getRequestInfo()
WRBGetLangauge()	WRB_getRequestInfo()
WRBGetCharacterEncoding()	WRB_getRequestInfo()
WRBGetReqMimeType()	WRB_getRequestInfo()
WRBGetUserID()	WRB_getRequestInfo()
WRBGetPassword()	WRB_getRequestInfo()
WRBGetClientIP()	WRB_getRequestInfo()
WRBGetEnvironment()	WRB_getEnvironment()
WRBGetEnvironmentVariable()	WRB_getEnvironment()
WRBGetDirMap()	WRB_getListenerInfo()
WRBGetMimeType()	WRB_getListenerInfo()
WRBGetParsedContent()	WRB_getParsedContent()
WRBGetNamedEntry()	WRB_getParsedContent()
WRBSetAuthorization()	WRB_setAuthBasic()
WRBSetAuthorization()	WRB_setAuthDigest()
WRBSetAuthorization()	WRB_setAuthServer()
WRBClientWrite()	WRB_write()
WRBClientRead()	WRB_read()
WRBReturnHTTPError()	WRB_sendHeader() and return WRB_ERROR
WRBReturnHTTPRedirect()	WRB_sendHeader()
WRBCloseHTTPHeader()	(obsolete—no WRB 3.0 equivalent)

Writing a WRB Cartridge

To write a WRB cartridge, you implement these cartridge functions:

- Init - performs one-time setup operations for the cartridge, such as allocating data structures or other resources that the cartridge needs to handle requests.
- Exec - handles client requests. You must implement this cartridge function.
- Shutdown - frees any resources the cartridge has allocated and prepares the cartridge to terminate.
- Authorize - determines whether the client issuing a request is authorized to issue the request.
- Reload - reloads the cartridge configuration data; this function is called whenever the Web Listener is signalled to reload its configuration data so your cartridge can reload its configuration data at the same time.

Note: In version 3.0, the Reload callback function is not called.

- Version - returns a version string; this is useful for debugging.
- Version_Free - frees any resources allocated by the Version function; the WRB application engine calls this function after successfully calling the Version function.

Although you must define an Exec function, implementing the other functions is optional. Implementing the Reload function, however, is highly recommended for cartridges that use cartridge configuration data.

You may name your cartridge functions anything you want, but it is a good idea to incorporate the above strings in the names you choose. For example, you might name your Init function something like `MyApp_Init()`. For this reason, this document uses cartridge function names such as *prefix*`_Init()`, where *prefix* indicates the name you choose for your cartridge.

Data Flow of a Typical Request

The following figure illustrates how a typical cartridge might handle an incoming request. The figure assumes that the cartridge handles its own authentication by defining an Authorize function (see The Authorize Function.)

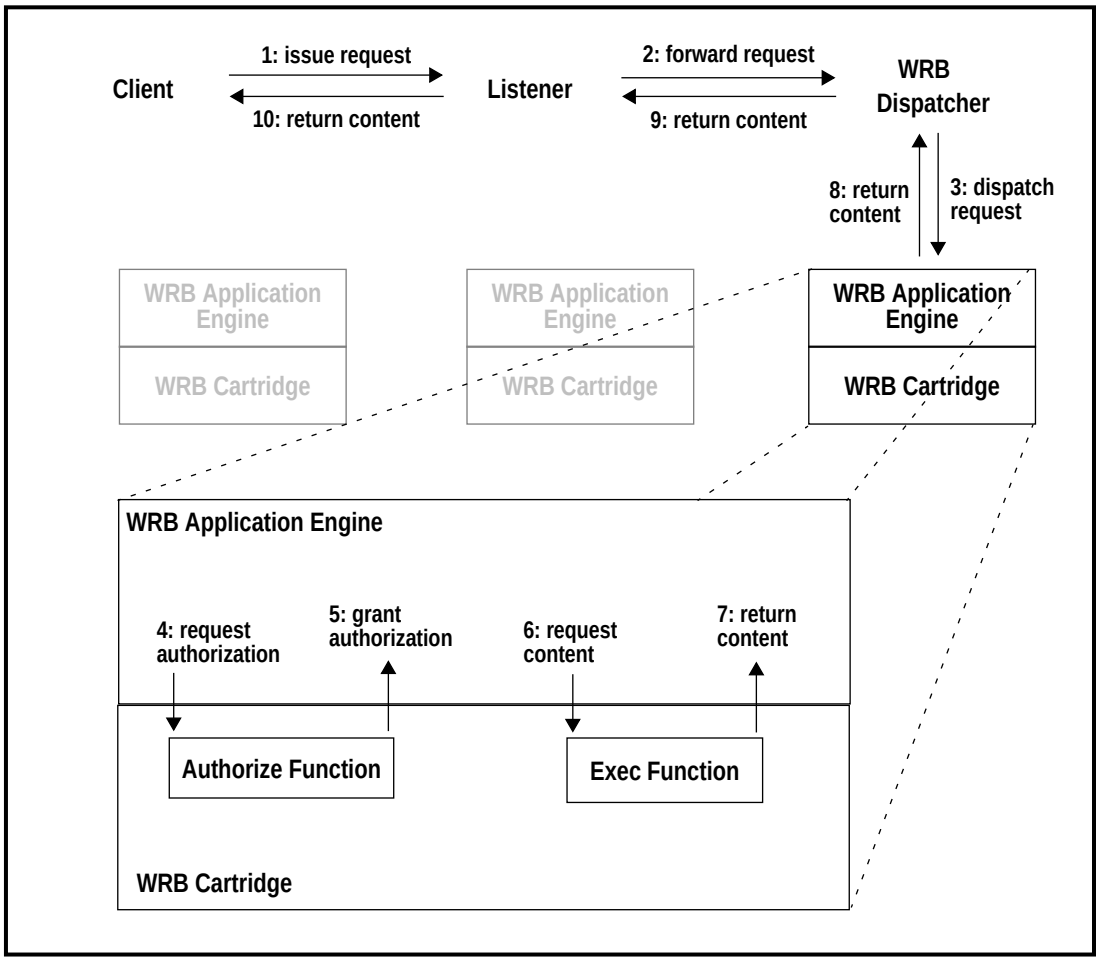


Figure 3-1: A request successfully handled by a WRB cartridge

Compiling, Linking, Configuring, and Running a WRB Cartridge

To run your cartridge, you need to compile and link it as a shared library and configure a Web Listener on your Web Application Server machine so it can access your shared library.

Compiling and Linking Your Cartridge

To compile and link your cartridge, you should start by copying and customizing one of the sample WRB cartridge makefiles:

1. **cd** to one of the subdirectories of **$ORACLE_HOME/ows/3.0/sample**.
2. Copy the file **Makefile** to your source code directory.
3. Customize your copy of the makefile to suit your cartridge. Be sure to set the `DESTDIR` variable to the directory where you want your shared library installed.
4. Type **make install** to compile and link your cartridge shared library and install it in your destination directory.

Configuring Your Cartridge

Once you have linked your shared library and installed it in the destination directory of your choice, you must choose at least one Web Listener on your Web Application Server machine and make your cartridge accessible to that Listener:

- Use the Cartridge Administration page to register your cartridge and define any cartridge-specific configuration parameters that your cartridge needs.
- If your cartridge uses the sessions mechanism to maintain persistent associations between clients and WRBXs, you must enable that mechanism in the WRB administration pages, and also specify the time-out duration, which is the maximum time your cartridge can be idle before the Dispatcher terminates it.
- Use the WRB administration pages to define the following for your cartridge:
  - An entry in the "Applications and Objects" section
  - Virtual directory mappings to allow your cartridge to access any scripts or data files it uses in the local files system (defined in the Applications and Directories section)
- Use the Web Listener administration pages to define virtual directory mappings to allow at least one Listener to access any splash page or other static documents that accompany your cartridge.
- Use the Authentication Server administration pages to define any authentication users, groups, or realms that the Authentication Server needs to perform authentication for your cartridge. If your cartridge handles its own authentication (see `WRB_setAuthBasic()` and `WRB_setAuthDigest()`), you may omit this step.

Debugging Your WRB Cartridge

To debug your cartridge, you can use the WRB Logger service to log debugging messages to log files. See WRB Logger Service API Reference for more information. As you test your cartridge, you can monitor its logging output to see what it is doing.

Each time you edit and recompile your cartridge and before you test it, you must:

1. Stop all running web listener processes.
2. Stop the WRB.
3. Restart the WRB.
4. Restart any web listeners you are using for testing.

You can use the Web Application Server Manager to do this.

The Entry-Point Function

In addition to implementing the cartridge functions, you must define an entry-point function that fills in a function table with pointers to the cartridge functions. The WRB application engine calls this entry-point function to get access to your cartridge functions when your cartridge first starts running.

You can name your entry-point function anything you want. When you configure your cartridge, you will specify the full path to your cartridge shared library and the name of your entry-point function. It is a good idea, though, to incorporate the string "Entry" in your entry-point function name (for example, `MyApp_Entry()`).

Syntax

```
WRBReturnCode prefix_Entry(WRBCallbacks *WRBcalls);
```

Parameters

WRBcalls	A pointer to the cartridge function table to be filled in.

Return Values

Your entry-point function must return one of these values of type `WRBReturnCode`:

Return value	Description
WRB_DONE	The function completed successfully.
WRB_ERROR	An error occurred and the request cannot be completed.
WRB_ABORT	An error occurred from which the application cannot recover; you need to terminate the application.

Example

See the `MyWRBApp_Entry()` function in the MyWRBApp sample cartridge.

The Init Function

Your Init cartridge function should perform any one-time setup and resource allocation that your WRB cartridge needs, such as shared data structures.

You can use your Init function to define a data structure that contains any data you want to share among your cartridge functions, and pass a pointer to it back from Init in the `appCtx` parameter. The WRB application engine then passes this pointer when it calls your other cartridge functions.

Your Init function can also call the `WRB_setAuthServer()` function to set up a client authentication or restriction scheme to be used for the life of the cartridge instance. If your Init function does this, your Authorize function will never be called.

Alternatively, you can use the WRB administration pages to specify authentication or restriction schemes for your cartridge. If you do this, any authentication or restriction schemes that you specify in your Init or Authorize function are ignored.

Syntax

```
WRBReturnCode prefix_Init(void *WRBCtx, void **appCtx);
```

Parameters

WRBCtx	A pointer to an object used by the WRB application engine that is opaque to your application. You must pass this pointer to any WRB API functions you call.
appCtx	To set up an application context structure that will subsequently be passed to your cartridge functions, allocate your context structure and store its address in the location pointed to by this parameter.

Return Values

Your Init function must return a value of type `WRBReturnCode` (see Return Values for the entry-point function).

Example

See the `MyWRBApp_Init()` function in the MyWRBApp sample cartridge.

See Also

The Shutdown Function

The Authorize Function

When the WRB dispatcher directs a request to your cartridge, the WRB application engine calls your Authorize cartridge function if these conditions are true:

- You have implemented an Authorize function and assigned a pointer to it in the `WRBCallbacks` function table.
- Your cartridge has not been protected explicitly using the WRB administration pages.
- Your cartridge Init function did not call `WRB_setAuthServer()`.

Your Authorize function must then pass back a boolean value indicating whether the client is authorized to issue the request. You can use the `WRB_getRequestInfo()` function to determine the client's privileges.

You can also use the `WRB_setAuthBasic or WRB_setAuthDigest` function to set up an authentication realm for the client to use in prompting the user for username and password.

Alternatively, you can call WRB_setAuthServer from your Init function to specify authentication and / or restriction schemes to be used for the life of the cartridge instance.

Another alternative is to use the WRB administration pages to specify authentication and / or restriction schemes. If you do this, any authentication or restriction schemes that you specify in your Init or Authorize function are ignored.

This cartridge function is optional. If you do not implement it, access to your WRB cartridge can be regulated only through the Web Listener and WRB administration pages.

Syntax

```
WRBReturnCode prefix_Authorize(void *WRBCtx,
                               void *appCtx,
                                  boolean *bAuthorized);
```

Parameters

WRBCtx	A pointer to an object used by the WRB application engine that is opaque to your application. You must pass this pointer to any WRB API functions you call.
appCtx	A pointer to the application context structure allocated by your Init function.
bAuthorized	Indicates whether the client is authorized to issue the request.

Return values

Your Authorize function must return a value of type `WRBReturnCode` (see Return Values for the entry-point function).

Example

See the `MyWRBApp_Authorize()` function in the MyWRBApp sample cartridge.

See Also

`WRB_getRequestInfo()`, `WRB_setAuthBasic()`, `WRB_setAuthDigest()`, `WRB_setAuthServer()`

The Exec Function

Your Exec cartridge function must handle requests that the WRB dispatcher directs to your cartridge. Typically, you define a set of requests for your cartridge to support. You must then decide how these requests will be encoded in URLs. Your Exec function can parse the request URL it gets from the Web Listener to determine what action to take. You can use the `WRB_getRequestInfo()` function to get the request URL.

Syntax

```
WRBReturnCode prefix_Exec(void *WRBCtx, void *appCtx);
```

Parameters

WRBCtx	A pointer to an object used by the WRB application engine that is opaque to your cartridge. You must pass this pointer to any WRB API functions you call.
appCtx	A pointer to the application context structure allocated your Init function.

Return values

Your Exec function must return a value of type `WRBReturnCode` (see Return Values for the entry-point function).

Example

See the `MyWRBApp_Exec()` function in the MyWRBApp sample cartridge.

The Shutdown Function

The WRB application engine calls your Shutdown cartridge function to prepare your cartridge to exit. Your Shutdown function must free any resources your cartridge is using. When your Shutdown function returns, your cartridge must be ready to exit at any time.

Syntax

```
WRBReturnCode prefix_Shutdown(void *WRBCtx, void *appCtx);
```

Parameters

WRBCtx	A pointer to an object used by the WRB application engine that is opaque to your cartridge. You must pass this pointer to any WRB API functions you call.
appCtx	A pointer to the application context structure allocated your Init function.

Return values

Your Shutdown function must return a value of type `WRBReturnCode` (see Return Values for the entry-point function). Returning a value of WRB_DONE implies that your cartridge is ready to exit at any time. If not, your cartridge will be shut down too.

Example

See the MyWRBApp sample function `MyWRBApp_Shutdown()`.

The Reload Function

When signaled, the Web Listener reloads it configuration data and then signals the WRB application engine to call each WRB cartridge's Reload cartridge function. If your cartridge uses configuration data, your Reload function should reload that data by calling `WRB_getAppConfigSection()` or `WRB_getAppConfigVal()`.

Note: In version 3.0, the Reload function is not called.

Syntax

```
WRBReturnCode prefix_Reload(void *WRBCtx, void *appCtx);
```

Parameters

WRBCtx	A pointer to an object used by the WRB application engine that is opaque to your cartridge. You must pass this pointer to any WRB API functions you call.
appCtx	A pointer to the application context structure allocated your Init function.

Return values

Your Reload function must return a value of type `WRBReturnCode` (see Return Values for the entry-point function).

Example

See the MyWRBApp sample function `MyWRBApp_Reload()`.

See Also

`WRB_getAppConfigSection()`, `WRB_getAppConfigVal()`

The Version Function

If you implement a Version cartridge function, the WRB application engine calls it on behalf of certain Web Listener utilities. The Version function must allocate and initialize a character string containing your cartridge's version number. Oracle recommends that your version string take the following form:

version.*release*.*update*

version should be a number between 0 and 255—incrementing this number indicates a major change in cartridge functionality.

release should be a number between 0 and 15—incrementing this number indicates relatively minor changes in cartridge functionality, perhaps including the addition of one or two new features.

update should be a number between 0 and 255—incrementing this number indicates bug fixes that correct existing cartridge functionality. This field is optional, and may be omitted when you increment the *release* field.

This cartridge function is optional; if you do implement it, you must also implement a Version_Free function.

Syntax

```
char *prefix_Version();
```

Parameters

none

Return Values

Your Version cartridge function must return a pointer to a character string identifying the version number of your cartridge.

Examples

An alpha version of your cartridge might define the following version string:

```
#define VERSION "0.1"
char * MyApp_Version() {
   char *ver;

   ver = (char *)malloc(sizeof(VERSION));
   strcpy(ver, VERSION);
   return(ver);
}
```

A beta version of your cartridge might define `VERSION` as:

```
#define VERSION "0.5"
```

The first production version of your cartridge might define `VERSION` as:

```
#define VERSION "1.0"
```

The first bug-fixing version of your cartridge might define `VERSION` as:

```
#define VERSION "1.0.1"
```

The next minor functionality update release of your cartridge might define `VERSION` as:

```
#define VERSION "1.1"
```

See Also

The Version_Free Function

The Version_Free Function

The Version_Free function is called after a successful call to the Version function. It must free the version string that the Version function allocated.

This cartridge function is optional; you must implement it only if you also implement a Version function.

Syntax

```
void prefix_Version_Free();
```

Parameters

none

Return values

none

See Also

The Version Function

CHAPTER

4 Using the WRB Core APIs

This chapter describes how you can use the WRB Core APIs to:

- Getting Information about the Request
- Getting Information about the Server
- Using Parameter Blocks
- Authenticating the Client
- Creating a Response
- Using Cookies
- Handling Errors
- Analyzing Performance

Getting Information about the Request

The WRB API provides functions that enable you to get information about the request. The following table lists the functions and the information they return:

This function:	Provides information on:
`WRB_recvHeaders()`	HTTP headers
`WRB_getParsedContent()`	query string

This function:	Provides information on:
`WRB_read()`	query string
`WRB_getRequestInfo()`	• the URL of the request • the IP address of the requestor • the type of listener • the virtual path and the physical path to which the virtual path is mapped • the languages the requestor can accept • the encodings the requestor can accept • the MIME type of the request • the username and password provided by the requestor in response to an authentication request

Several functions return information about the query string; which function you should use depends on the format of the query string:

- If the request uses the standard GET or POST formats, you should use `WRB_getParsedContent()`. This function parses the query string information and returns it in the form of a parameter block. You can then use the parameter block functions to get the value of a specific parameter in the query string, or to walk the whole parameter block to get all the name-value pairs.

  HTML forms submit data in the standard GET or POST formats. The query string is a set of name-value pairs.

- If the request has the "multipart/form" format, use `WRB_getMultipartData()`. This format is defined by RFC 1867.
- If the request is not in a standard format, use `WRB_read()`. This function reads the specified number of bytes into a buffer, which you can then parse yourself.

Getting Information about the Server

The WRB API provides functions that enable you to get information from the configuration file. You might need to do this to avoid hardcoding values in your cartridge's source code. From the configuration file, you can get configuration information about all cartridges, values of environment variables, and listener

information. The following table lists the functions and what information they provide:

This function	Provides information on:
`WRB_getAppConfigSection()`	cartridge configuration data
`WRB_getAppConfigVal()`	value of one configuration parameter
`WRB_getCartridgeName()`	the name of the calling cartridge
`WRB_getEnvironment()`	the cartridge environment variables
`WRB_getListenerInfo()`	information about the Listener that sent the request
`WRB_getMultAppConfigSection()`	configuration data for multiple cartridges

Using Parameter Blocks

A parameter block is a set of name-value pairs, where each name-value pair is encoded in the `WRBpBlockElem` struct type. This struct contains the name, value, and type of a name-value pair. A parameter block itself has the `WRBpBlock` type.

Name-value pairs are commonly used in web applications: for example, environment variables, HTTP header information, query string information, cartridge configuration information, and listener configuration information are all sets of name-value pairs.

Several WRB API functions provide data in the `WRBpBlockElem` type, which you can pass to the parameter block functions. For example, the following code calls `WRB_getParsedContent()` to get a parameter block that contains name-value pairs found in the query string, and then calls `WRB_findPBElemVal()` to get the value of a parameter called “lastname”:

```
WRBpBlock *qstring;
text *lname;

/* gets the query string information and places it in qstring */
WRB_getParsedContent(ctx, qstring);

/* gets the value of the “lastname” parameter in the
   qstring parameter block */
```

```
lname = WRB_findPBElemVal(ctx, qstring, (text *)"lastname",
            -1);
```

WRB API also provides functions that let you walk a parameter block. To do this, you call `WRB_firstPBElem()` to get the first element in the parameter block, and then call `WRB_nextPBElem()` to get each subsequent element. `WRB_nextPBElem()` returns NULL when there are no more elements. The following example calls `WRB_recvHeaders()` to get a parameter block that contains name-value pairs found in the HTTP header, and prints the values out. This example uses fields within the `WRBpBlockElem` struct, which is defined as:

```
typedef struct _WRBPBElem
{
  text  *szParamName; /* the parameter name */
  sb4    nParamName;  /* the length of the parameter name */

  text  *szParamValue; /* the parameter value */
  sb4    nParamValue;  /* the length of the parameter value */

  ub2    nParamType;   /* the type of the parameter value */

  dvoid *pNVdata;
} WRBpBlockElem;
```

The code for the example:

```
WRBpBlock hPBlock;
WAPIReturnCode ret;
WRBpBlockElem *elem;
dvoid *pos;

/* Get the headers and save them in hPBlock */
ret = WRB_recvHeaders(ctx, &hPBlock);
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}

WRB_printf(ctx, "<dl>"); /* start an HTML definition list. */

/* walk the parameter block */
for (elem = WRB_firstPBElem(ctx, hPBlock, &pos);
         elem; elem = WRB_nextPBElem(ctx, hPBlock, pos)) {
   /* return the parameter name and value to the user */
   WRB_printf(ctx, "<dt>%s", elem->szParamName);
   WRB_printf(ctx, "<dd>%s", elem->szParamValue);
}

WRB_printf(ctx, "</dl>"); /* close the HTML definition list */

WRB_destroyPBlock(ctx, hPBlock);
```

The following table lists the functions that let you manipulate parameter blocks:

Function	Description
`WRB_createPBlock()`	Creates a parameter block
`WRB_addPBElem()`	Adds an element to a parameter block
`WRB_copyPBlock()`	Copies a parameter block
`WRB_delPBElem()`	Deletes an element from a parameter block
`WRB_destroyPBlock()`	Destroys a parameter block
`WRB_findPBElem()`	Finds an element by name in a parameter block
`WRB_walkPBlock()`	Finds an element by position in a parameter block
`WRB_numPBElem()`	Returns the number of elements in the parameter block
`WRB_firstPBElem()`	Returns the first element in a parameter block
`WRB_nextPBElem()`	Returns the next element in a parameter block
`WRB_findPBElemVal()`	Returns the value of a parameter in a parameter block

Authenticating the Client

WRB API provides functions that enable you to authenticate the client. You can create a new basic or digest authentication realm and authenticate the user in your Authorize function, or you can specify a list of restriction and/or authentication schemes and let the authentication server authenticate the user.

The following functions enable you to authenticate the client:

- `WRB_setAuthBasic()`—Create a new basic authentication realm
- `WRB_setAuthDigest()`—Create a new digest authentication realm
- `WRB_setAuthServer()`—Use the authentication server for authentication
- `WRB_getClientCert()`—Get an SSL certificate from the client

Creating a Response

WRB API provides functions that enable you to generate HTTP headers and HTML pages dynamically.

To write HTTP headers:

1. Create a parameter block by calling `WRB_createPBlock()`.
2. Add elements to it by calling `WRB_addPBElem()`.
3. Send the header by calling `WRB_sendHeader()`.

To generate HTML pages, use `WRB_printf()` or `WRB_write()`. These functions would contain HTML tags and text that the user sees. You would call these functions after you send the header information.

- `WRB_annotateURL()`—Write a URL with appended query string

Using Cookies

WRB API provides the `WRB_getCookies()` and `WRB_setCookies()` functions, which enable you to set and get cookie data. `WRB_setCookies()` must be called when you are setting HTTP header information for your response, that is, you call it before you send the header information with `WRB_sendHeader()`.

Handling Errors

WRB API provides the `WRB_getPreviousError()` function which lets you get the most recent error that was logged in the log file. You can get the error text for the most recent error and display it to the user.

Analyzing Performance

WRB API provides the `WRB_timestamp()` function, which writes an entry in the log file when this function is executed.

CHAPTER

5

The MyWRBApp Sample Cartridge

MyWRBApp is a simple electronic commerce cartridge that illustrates how to use the WRB API to do these programming tasks:

- Set up and Initialize a WRB Cartridge
- Clean up a WRB Cartridge Before Terminating
- Collect Registration Data From Users
- Authenticate Registered Users
- Parse URLs and Other HTTP Data
- Parse HTML Forms Data
- Maintain Persistent Client State Data (Using Sessions)
- Create HTML Documents Dynamically and Send Them to Clients

Programming Tasks Illustrated by the MyWRBApp Sample Cartridge

Set up and Initialize a WRB Cartridge

All WRB cartridges must implement an entry-point function to initialize a function dispatch table that the WRB applications engine can use to call the cartridge functions.

The `MyWRBApp_Entry()` function sets up this function dispatch table for the MyWRBApp cartridge.

After the entry-point function returns, if the cartridge has defined an Init cartridge function, the WRB application engine calls the Init function to perform any other one-time setup that the cartridge needs. The Init function may allocate an application context structure of any type and pass a pointer to it back to the WRB application engine for that WRBX. The WRB application engine subsequently passes this pointer to every cartridge function that runs in that WRBX.

The `MyWRBApp_Init()` function allocates and initializes an application context structure (myappctx) that the MyWRBApp cartridge uses to keep track of state data between cartridge function calls.

Clean up a WRB Cartridge Before Terminating

If a WRB cartridge defines an Init function, it should also define a Shutdown function to free any resources allocated by Init. Before terminating a WRBX, the WRB application engine for that WRBX calls this Shutdown function. When Shutdown returns, the WRBX may terminate at any time.

In the MyWRBApp cartridge, `MyWRBApp_Shutdown()` frees the application context structure allocated by `MyWRBApp_Init()`.

Access Database Content Using the WRB Content Service

The MyWRBApp cartridge stores user registration data and the catalog of items in a local database and uses the WRB content service to access this data.

The MyWRBApp functions `MyWRBApp_Init()`, `MyWRBApp_Shutdown()`, `newUser()`, `editUserData()`, and `getCatalog()` use WRB content service functions to connect to a local database and access documents.

Collect Registration Data From Users

A WRB cartridge that requires users to authenticate themselves before using the cartridge must maintain a list of valid users in some form. A cartridge that also allows users to register their own usernames with the cartridge and provide other information about themselves, such as address or phone number, must implement a mechanism for collecting and storing this information.

MyWRBApp illustrates one way to do this, using per-user database documents that are shared among WRBXs. The `newUser()`, `getUserData()`, and `editUserData()` functions, and the functions that they call, manipulate these documents.

See also Authenticate Registered Users.

Authenticate Registered Users

WRB cartridges that perform their own authentication must implement an Authorize cartridge function. To authenticate a client, `MyWRBApp_Authorize()` verifies that the username / password pair given by the client is already registered with the cartridge.

Parse URLs and Other HTTP Data

In handling requests, WRB cartridges must have access to request URLs and other HTTP data from the Listener. `MyWRBApp_Authorize()` and `MyWRBApp_Exec()` illustrate how to do this using various WRB API functions

Parse HTML Forms Data

To process HTML forms, WRB cartridges must get access to Listener CGI data. The MyWRBApp functions `newUser()`, `editUserData()`, `placeOrder()`, and `confirmOrder()` illustrate how to do this using the WRB API functions `WRB_getParsedContent()` and `WRBGetParsedContent()`.

Maintain Persistent Client State Data (Using Sessions)

Many WRB cartridge developers want their cartridges to keep track of client data that persists from one HTTP request to the next. A way to do this is to use the WRB configuration data for your cartridge to enable *sessions*. A session is a persistent association, maintained by the dispatcher, between a client and an instance of a WRB cartridge (a WRB execution instance, or WRBX).

When a client issues a request to a session-enabled cartridge, the Dispatcher generates a unique identifier and sends it to the client browser as a cookie in the request response headers. When the client issues subsequent requests to the cartridge, within a configurable time-out period, the Dispatcher uses the unique identifier cookie sent with the request to route the request to the same WRBX.

The MyWRBApp cartridge must run with sessions enabled, because it stored client data in it application context structure (see The myappctx structure). When a user submits the MyWRBApp catalog form to add items to an order, the MyWRBApp cartridge keeps track of the ordered items in the myappctx structure.

The user may then leave and return to the catalog page many times, adding or removing items from the order, and because the Dispatcher maintains the association between client and WRBX, the myappctx structure correctly keeps track of the order. When the user finally confirms the order, the Dispatcher signals the browser to delete the unique identifier cookie from its local storage.

The MyWRBApp functions `placeOrder()` and `confirmOrder()` illustrate use the myappctx structure to keep track of the client's order.

Create HTML Documents Dynamically and Send Them to Clients

The primary reason for writing server-side web applications is to be able to respond to HTTP requests with dynamically generated content. The MyWRBApp functions `getCatalog()`, `placeOrder()`, `formatUserData()`, and `userDataError()` illustrate how to generate HTML dynamically and send it to clients using the WRB API function `WRBClientWrite()`.

mywrbapp.h

The **mywrbapp.h** header file includes system header files that MyWRBApp needs, and defines several data structures and macros. The only features of this header that your cartridge should specifically emulate are the first two `#include` directives, which provide access to the WRB API. The rest of the declarations and definitions in this header are specific to MyWRBApp.

```
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif

#ifndef WRB_ORACLE
#include <wrb.h>
#endif

#ifndef CONTENT_ORACLE
#include <content.h>
#endif

#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define MAXITEMS 170        /* max items allowed in orderitems array */
#define PARTNUMLEN 9

/* info file field names */
#define USERNAME (text *)"Username"
#define PASSWORD (text *)"Password"
#define FULLNAME (text *)"Full Name"
#define ADDRESS (text *)"Street Address"
#define CITY (text *)"City"
#define STATE (text *)"State"
#define ZIP (text *)"Zip Code"
#define COUNTRY (text *)"Country"
#define EMAIL (text *)"Email Address"
```

The orderitem structure

MyWRBApp uses an orderitem structure to store the part number of an item that a user has ordered and the quantity ordered. The cartridge constructs an array of these structures, which it uses to set the cookie data it sends to the client.

```
/* orderitem stores information about an item that a user has ordered */
typedef struct orderitem {
    text partnum[PARTNUMLEN + 1];
    text qty[5];
} orderitem;
```

The myappctx structure

MyWRBApp uses the myappctx structure to store client state data.

```
/*
 * application context--set up by Init function and subsequently passed
 * to each cartridge function by the WRB application engine
 */
typedef struct myappctx {
    dvoid *hRepository; /* handle to our CMS content repository */
    orderitem *items;   /* pointer to ordered items array */
    int numitems;       /* number of orders currently in array */
    text *state;        /* our home state, read from configuration data */
    double taxpct;      /* tax in our area, read from configuration data */
} myappctx;

/*
 * MyWRBApp cartridge functions
 */
WRBReturnCode
MyWRBApp_Entry(WRBCallbacks *WRBCalls);

WRBReturnCode
MyWRBApp_Init(void *WRBCtx, void **appCtx);

WRBReturnCode
MyWRBApp_Shutdown(void *WRBCtx, void *appCtx);

WRBReturnCode
MyWRBApp_Authorize(void *WRBCtx, void *appCtx, boolean *bAuthorized);

WRBReturnCode
MyWRBApp_Exec(void *WRBCtx, void *appCtx);

WRBReturnCode
MyWRBApp_Reload(void *WRBCtx, void *appCtx);
```

mywrbapp.c

```
#include "mywrbapp.h"

#ifdef DEBUG
static char debug[1024];
#endif

/*
 * forward declarations
 */
WRBReturnCode newUser(myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode getUserData(myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode editUserData(myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode getCatalog(myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode placeOrder(myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode confirmOrder(myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode formatUserData(text *user, myappctx *ctx,
  dvoid *WRBCtx);

dvoid userDataError(const text *message, const text *URL, myappctx *ctx,
  dvoid *WRBCtx);

WRBReturnCode storeInfo(text* user, myappctx *ctx, dvoid *WRBCtx);

WRBReturnCode formatOrder(const boolean invoice, myappctx *ctx,
  dvoid *WRBCtx);

boolean taxable(text *user, myappctx *ctx, dvoid *WRBCtx);

int itemcmp(const dvoid* item1, const dvoid* item2);

text *getInfoField(text *user, dvoid *hInfoFile, text *field, myappctx *ctx,
  dvoid *WRBCtx);

text *myfgets(text *buf, int size, dvoid *hFile, dvoid *WRBCtx);

WRBReturnCode uploadCatalog(myappctx *ctx, dvoid *WRBCtx);
```

MyWRBApp_Entry()

MyWRBApp_Entry() sets up the provided WRBCallbacks dispatch table so the WRB application engine can call MyWRBApp's cartridge functions.

```
WRBReturnCode
MyWRBApp_Entry(WRBCallbacks *WRBCalls)
{
    WRBCalls->init_WRBCallback = MyWRBApp_Init;
    WRBCalls->authorize_WRBCallback = MyWRBApp_Authorize;
    WRBCalls->exec_WRBCallback = MyWRBApp_Exec;
    WRBCalls->shut_WRBCallback = MyWRBApp_Shutdown;
    WRBCalls->reload_WRBCallback = MyWRBApp_Reload;
    /* Version and Version_Free currently not implemented */

    return (WRB_DONE);
}
```

MyWRBApp_Init()

MyWBApp_Init() sets up the cartridge to handle client requests. It allocates the myappctx structure and the ordered items array, and opens a connection to the content repository where it will store the catalog and user data files. MyWRBApp_Init() then completes initialization of the context structure and passes it back to the WRB application engine when it returns.

```
WRBReturnCode
MyWRBApp_Init(dvoid *WRBCtx, dvoid **appCtx)
{
    myappctx *ctx;
    WRBReturnCode ret;

    ctx = (myappctx *)malloc(sizeof(myappctx));

    /* open content repository */
    ctx->hRepository = WRB_CNTopenRepository(WRBCtx, (text *)"mywrbapp",
     (text *)"mywrbapp", NULL);

    /* upload catalog file to repository */
    ret = uploadCatalog(ctx, WRBCtx);
    if (ret != WRB_DONE) {
        return (ret);
    }

    /* allocate order items array */
    ctx->items = (orderitem *)malloc((MAXITEMS + 1) * sizeof(orderitem));
    ctx->numitems = 0;
```

```
    /* load configuration parameters */
    MyWRBApp_Reload(WRBCtx, ctx);

    /* pass back context pointer to WRB application engine */
    *appCtx = ctx;

    return (WRB_DONE);
}
```

MyWRBApp_Shutdown()

MyWRBApp_Shutdown() frees resources used by the cartridge in preparation for termination. It saves the data file to the file system, and unmaps and closes the data file. It then frees the ordered items array and the application context structure before returning.

```
WRBReturnCode
MyWRBApp_Shutdown(dvoid *WRBCtx, dvoid *appCtx)
{
    myappctx *ctx = (myappctx *)appCtx;

    WRB_CNTcloseRepository(WRBCtx, ctx->hRepository);

    /* free items array and context structure */
    free(ctx->items);
    free(ctx);

    return (WRB_DONE);
}
```

MyWRBApp_Authorize()

The WRB application engine calls MyWRBApp_Authorize() when it receives a request directed to MyWRBApp. MyWRBApp_Authorize() authenticates the client issuing the request.

MyWRBApp does not require any special authorization to register a new a username, so if the request is "newuser," MyWRBApp_Authorize() grants authorization and returns immediately.

Otherwise, MyWRBApp_Authorize() uses the WRB_setAuthBasic() function to set up a new basic authentication realm named "MyWRBApp" with the client. This tells the client browser to prompt the user for a username and password and cache them. The browser will then provide this username/password pair when the Web Listener asks for authentication data for this realm in the future.

`MyWRBApp_Authorize()` calls WRB_getRequestInfo() to get the username that the user entered in the client browser's authentication dialog. If the function returns NULL, it is probably because the client browser is issuing its first request to MyWRBApp, so the MyWRBApp authentication realm was not established before the client issued the request. This is not an error—MyWRBApp just denies authorization and returns normally. The client browser responds by prompting the user for a username and password using the MyWRBApp realm, and retries the request. Next time, the call to WRB_getRequestInfo() will succeed.

```
RBReturnCode
MyWRBApp_Authorize(dvoid *WRBCtx, dvoid *appCtx, boolean *bAuthorized)
{
    text *URL, *req, *user, *password, *storedPassword;
    myappctx *ctx = (myappctx *)appCtx;
    int cmp;

    /* get request URL */
    URL = WRB_getRequestInfo(WRBCtx, WRBR_URL);
    URL = (text *)strdup((char *)URL);      /* copy to local memory */
#ifdef DEBUG
    sprintf(debug, "MyWRBApp_Authorize: Request URL is: %s", URL);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    /* last path element defines request name */
    req = (text *)strrchr((char *)URL, '/');
    req++;
    if (!strncmp((char *)req, "newuser", 7)) {
        /* client is creating a new account, and is therefore authorized */
        *bAuthorized = TRUE;
        return (WRB_DONE);
    }

    /*
     * set up a new basic authentication realm called "MyWRBApp"
     * and share it with the client
     */
    WRB_setAuthBasic(WRBCtx, (text *)"MyWRBApp");

    /* get the username provided by the client */
    user = WRB_getRequestInfo(WRBCtx, WRBR_USER);
    if (!user) {
        /*
         * This is probably the client's first call to this app--
         * Just return normally with authorization failed
         */
        *bAuthorized = FALSE;
```

```
#ifdef DEBUG
        sprintf(debug,
          "MyWRBApp_Authorize: failed to get User ID");
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        return (WRB_DONE);
    }

    /* copy username to local memory XXX still needed? */
    user = (text *)strdup((char *)user);

    /* get the password provided by the client and copy to local memory */
    password = WRB_getRequestInfo(WRBCtx, WRBR_PASSWORD);
    password = (text *)strdup((char *)password); /* XXX still needed? */

    /* get stored password from user's data file */
    storedPassword = getInfoField(user, NULL, PASSWORD, ctx, WRBCtx);

    /* compare stored password with password given */
    cmp = strcmp((char *)password, (char *)storedPassword);

    if (!cmp) {
        *bAuthorized = TRUE;
    }
    else {
        *bAuthorized = FALSE;
#ifdef DEBUG
        sprintf(debug,
          "MyWRBApp_Authorize: user %s gave an incorrect password: %s",
          user, password);
        WRBLogMessage(WRBCtx, debug, 1);
#endif
    }

    return(WRB_DONE);
}
```

MyWRBApp_Exec()

The WRB application engine calls `MyWRBApp_Exec()` to handle a request when `MyWRBApp_Authorize()` has indicated that the client is authorized. `MyWRBApp_Exec()` reads the URL to determine the request. MyWRBApp supports the following requests:

- newuser—register a new username
- getuserdata—display to the client the currently authenticated user's registration data

- edituserdata—apply the user's changes to his or her registration data
- getcatalog—display the product catalog to the client
- placeorder—display to the client a summary of the items the currently authenticated user has ordered, asking the user to confirm the order
- confirm—display to the client an invoice for a confirmed order, including total price

```
WRBReturnCode
MyWRBApp_Exec(dvoid *WRBCtx, dvoid *appCtx)
{
    text *URL, *req;
    myappctx *ctx = (myappctx *)appCtx;

    /* get request URL */
    URL = WRB_getRequestInfo(WRBCtx, WRBR_URL);
    URL = (text *)strdup((char *)URL);

    /* last path element defines request name */
    req = (text *)strrchr((char *)URL, '/');
    req++;

#ifdef DEBUG
    sprintf(debug, "MyWRBApp_Exec: request is %s", req);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    if (!strncmp((char *)req, "newuser", 7)) {
        return (newUser(ctx, WRBCtx));
    }
    else if (!strncmp((char *)req, "getuserdata", 11)) {
        return (getUserData(ctx, WRBCtx));
    }
    else if (!strncmp((char *)req, "edituserdata", 12)) {
        return (editUserData(ctx, WRBCtx));
    }
    else if (!strncmp((char *)req, "getcatalog", 10)) {
        return(getCatalog(ctx, WRBCtx));
    }
    else if (!strncmp((char *)req, "placeorder", 10)) {
        return(placeOrder(ctx, WRBCtx));
    }
    else if (!strncmp((char *)req, "confirm", 7)) {
        return(confirmOrder(ctx, WRBCtx));
    }

    /* bad URL--send HTTP error */
    /* XXX update to WRB_sendHeader */
    WRBReturnHTTPError(WRBCtx, (WRBErrorCode)400, NULL, TRUE);
```

```
    return(WRB_DONE);
}
```

MyWRBApp_Reload()

MyWRBApp_Reload() calls WRB_getAppConfigVal() to update the application context structure's copies of the cartridge's configurable parameters.

```
WRBReturnCode
MyWRBApp_Reload(dvoid* WRBCtx, dvoid *appCtx)
{
    myappctx *ctx = (myappctx*)appCtx;
    text *tax;
    text *cp;

    /* get configuration parameters */
    ctx->state = WRB_getAppConfigVal(WRBCtx, NULL, (text *)"state");
    tax = WRB_getAppConfigVal(WRBCtx, NULL, (text *)"tax");

    if (tax) {
        /* remove percent sign from tax, if any */
        cp = (text *)strchr((char *)tax, '%');
        if (cp) {
            cp = '\0';
        }
        ctx->taxpct = atof((char *)tax);
    }
    else {
        ctx->taxpct = (double)0;
    }

    return (WRB_DONE);
}
```

newUser()

MyWRBApp_Exec() calls newUser() to handle "newuser" requests. Its primary task is to process the data from the **newuser.html** form that the client has submitted (see the **$ORACLE_HOME/ows/3.0/sample/wrbsdk/mywrbapp** directory on your Web Application Server machine).

newUser() uses WRB API function WRB_getParsedContent() to parse the CGI data from the form. The function passes back an array of WRBEntry structures, each of which encodes the name and value of an input field on the form.

newUser() then uses the WRB API function WRB_findPBElemVal() to extract the values of specific input fields.

```
WRBReturnCode
newUser(myappctx *ctx, dvoid *WRBCtx)
{
    text *user, *password, *confirm, *cp;
    dvoid *hInfoFile;
    WRBReturnCode ret;
    WRBpBlock hPBlock;
    WRBpBlockElem *elem;
    sb4 numelems;
    text wbuf[1024];
    int len;

    /* get username from form data */
    WRB_getParsedContent(WRBCtx, &hPBlock);
    user = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"username", -1);

    /* check whether this name is in use */
    if (hInfoFile = WRB_CNTopenDocument(WRBCtx, ctx->hRepository, user,
      WRBCS_OPEN | WRBCS_READ)) {
        /*
         * The given username is already in use--
         * respond with an error page
         */
        WRB_CNTcloseDocument(WRBCtx, hInfoFile);

        userDataError((text *)"The username you entered is already registered
to another user. Please choose another username.",
          (text *)"/sample/wrbsdk/mywrbapp/newuser.html", ctx, WRBCtx);

        WRB_destroyPBlock(WRBCtx, hPBlock);

        return (WRB_DONE);
    }

    /* get password from form data */
    password = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"password", -1);

    /* get password confirmation and compare with password */
    confirm = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"confirm", -1);
    if (strcmp((char *)password, (char *)confirm)) {
        /*
         * confirmation doesn't match password--generate error page
         */
        userDataError((text *)"The password you entered does not match your co
nfirmation string. Please reenter your password",
          (text *)"/mywrbapp/bin/getuserdata", ctx, WRBCtx);
```

```
            WRB_destroyPBlock(WRBCtx, hPBlock);

            return (WRB_DONE);
        }

        ret = storeInfo(user, ctx, WRBCtx);
        if (ret != WRB_DONE) {
            WRB_destroyPBlock(WRBCtx, hPBlock);

            return (ret);
        }

        /* done--go to catalog */
        WRB_destroyPBlock(WRBCtx, hPBlock);

        return (getCatalog(ctx, WRBCtx));
    }
```

getUserData()

MyWRBApp_Exec() calls getUserData() to handle "getuserdata" requests. Its task is to extract from the data file the username and password of the currently authenticated user, and call formatUserData() to generate and send to the client an HTML form containing the user's registration data. getUserData() uses the WRB API function WRBGetUserID() to identify the currently authenticated user.

```
WRBReturnCode
getUserData(myappctx *ctx, dvoid *WRBCtx)
{
    text *user;
    dvoid *hInfoFile;
    WRBReturnCode ret;

    /* get name of currently authenticated user */
    user = WRB_getRequestInfo(WRBCtx, WRBR_USER);
    user = (text *)strdup((char *)user);
    if (!user) {
        /* this shouldn't happen--Authorize has already been called */
        return (WRB_ERROR);
    }

    /* fill in HTML form template and write it to the client */
    ret = formatUserData(user, ctx, WRBCtx);

    return (ret);
}
```

editUserData()

When the user edits and submits the HTML form generated by `getUserData()`, the form's action specifies the "edituserdata" request. `MyWRBApp_Exec()` then calls `editUserData()` to handle this request.

Like `newUser()`, `editUserData()` uses the WRB API functions `WRBGetParsedContent()` and `WRBGetNamedEntry()` to parse the CGI data from the form.

```
WRBReturnCode
editUserData(myappctx *ctx, dvoid *WRBCtx)
{
    text *user, *curUser, *password, *confirm;
    text wbuf[1024];
    int len;
    dvoid *hInfoFile;
    WRBpBlock hPBlock;
    WRBpBlockElem *elem;
    sb4 numelems;

    /* get data from form */
    WRB_getParsedContent(WRBCtx, &hPBlock);
    user = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"username", -1);
    password = WRB_findPBElemVal(WRBCtx, hPBlock,
      (text *)"password", -1);

    /* get password confirmation and compare with password */
    confirm = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"confirm", -1);
    if (!strcmp((char *)password, (char *)confirm)) {
        /* confirmation doesn't match password--generate error page */
        userDataError((text *)"The password you entered does not match your co
nfirmation string. Please reenter your password",
          (text *)"/mywrbapp/bin/getuserdata", ctx, WRBCtx);

        WRB_destroyPBlock(WRBCtx, hPBlock);

        return (WRB_DONE);
    }

    /* get name of currently authenticated user */
    curUser = WRB_getRequestInfo(WRBCtx, WRBR_USER);

    /*
     * has the user changed his/her username?
     * if so, and the new name is not in use, rehash datablock
     */

    if (strcmp((char *)user, (char *)curUser)) {
```

```
        /* new username */
        hInfoFile = WRB_CNTopenDocument(WRBCtx, ctx->hRepository, user,
          WRBCS_OPEN | WRBCS_READ);
        if (hInfoFile) {
            /* specified username is already in use--generate error page */
            WRB_CNTcloseDocument(WRBCtx, hInfoFile);

            userDataError((text *)"The username you entered is already registe
red to another user. Please choose another username.",
              (text *)"/mywrbapp/bin/getuserdata", ctx, WRBCtx);

            WRB_destroyPBlock(WRBCtx, hPBlock);

            return (WRB_DONE);
        }

        /* destroy old info file before calling storeInfo to create new one */
        WRB_CNTdestroyDocument(WRBCtx, ctx->hRepository, (text *)curUser);

#ifdef DEBUG
        sprintf(debug,
          "editUserData: old username: %s\n\tnew username %s",
          curUser, user);
        WRBLogMessage(WRBCtx, debug, 1);
#endif
    }

    storeInfo(user, ctx, WRBCtx);

    /* done--go to catalog */
    WRB_destroyPBlock(WRBCtx, hPBlock);

    return(getCatalog(ctx, WRBCtx));
}
```

getCatalog()

MyWRBApp_Exec() calls getCatalog() to handle “getcatalog” requests. Its main task is to read the catalog file (catalog.txt), apply HTML formatting to it, and send it to the client as an editable HTML form.

For each catalog item, getCatalog() searches the catalog file for that item’s part number; if it finds the item, it uses its value to set the default value of the input field in the output form. This value represents the quantity of the item ordered.

```
WRBReturnCode
getCatalog(myappctx *ctx, dvoid *WRBCtx)
{
```

```
    dvoid *hCatFile;
    text buf[1024];
    text wbuf[1024];
    int len;
    int item;
    orderitem *items;
    text part[PARTNUMLEN + 1];
    text qty[5];
    text *qp = qty;

    /* open catalog */
    hCatFile = WRB_CNTopenDocument(WRBCtx, ctx->hRepository,
                  (text *)"catalog", WRBCS_OPEN | WRBCS_READ);
    if (!hCatFile) {
#ifdef DEBUG
        sprintf(debug, "getCatalog: failed to open catalog: %s",
          strerror(errno));
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        return (WRB_ABORT);
    }

    /* start generating HTML for the client */
    WRB_printf(WRBCtx, (text *)"Content-type: text/html\n\n");

    WRB_printf(WRBCtx,
      (text *)"<HTML><HEAD><TITLE>Hot Goods Catalog</TITLE></HEAD>\n");
    WRB_printf(WRBCtx, (text *)"<BODY BGCOLOR=\"FFFFFF\">\n");
    WRB_printf(WRBCtx, (text *)"<H1>Hot Goods Catalog</H1>\n");
    WRB_printf(WRBCtx,
      (text *)"<FORM ACTION=\"/mywrbapp/bin/placeorder\" METHOD=\"POST\">\n");

    /*
     * Loop through the catalog generating HTML for each line--
     * for each part number listed in the catalog, check the
     * ordered items to see if the user has already entered an order quantity
     * for the item
     */
    while (myfgets(buf, 1024, hCatFile, WRBCtx)) {
        /* write the description line to the client */
        WRB_printf(WRBCtx, (text *)"<P>");
        WRB_printf(WRBCtx, buf);
        WRB_printf(WRBCtx, (text *)"</P>\n<P>");

        /* get part number line and write it out without the newline */
        myfgets(buf, 1024, hCatFile, WRBCtx);
        WRB_printf(WRBCtx, buf);

        /* the part number is the first PARTNUMLEN chars of buf */
        strncpy((char *)part, (char *)buf, PARTNUMLEN);
```

```
            part[PARTNUMLEN] = '\0';

            items = ctx->items;
            item = 0;
            *qp = '\0';

            /* search through ordered items for part number */
            while (item < ctx->numitems) {
                if (!strncmp((char *)items[item].partnum, (char *)part,
                  PARTNUMLEN)) {
                    /* found part number--now get quantity */
                    qp = (text *)strdup((char *)items[item].qty);
                    break;
                }
                item++;
            }

            /*
             * complete the HTML line with an input box and set its value to
             * the quantity stored in the ordered items array (if any)
             */
            WRB_printf(WRBCtx,
              (text *)" Order quantity: <INPUT TYPE=\"text\" NAME=\"%s\" ", part);
            WRB_printf(WRBCtx, (text *)"SIZE=\"4\" VALUE=\"%s\"></P>\n", qty);
        }

        /* finish HTML page */
        WRB_printf(WRBCtx, (text *)"<INPUT TYPE=\"submit\" NAME=\"submit\" ");
        WRB_printf(WRBCtx, (text *)"VALUE=\"Place Order\">\n");
        WRB_printf(WRBCtx, (text *)"</FORM>\n</BODY>\n</HTML>");

        WRB_CNTcloseDocument(WRBCtx, hCatFile);
        return (WRB_DONE);
    }
```

placeOrder()

When the user edits and submits the HTML form generated by `getCatalog()`, the form's action specifies the "placeorder" request. `MyWRBApp_Exec()` then calls `placeOrder()` to handle this request. Its main task is to generate an HTML form that summarizes the items the client has ordered, and asking the client for confirmation.

To determine the items ordered, `placeOrder()` first parses the cookie data, and then the data from the form generated by `getCatalog()`. In the case of a discrepancy between the cookie data and form data, `placeOrder()` uses the

form data. This allows a user to change the quantity on an order item or delete an item from the order entirely before submitting the “placeorder” request.

```
WRBReturnCode
placeOrder(myappctx *ctx, dvoid *WRBCtx)
{
    WRBReturnCode ret;
    WRBpBlock hPBlock;
    WRBpBlockElem *elem;
    sb4 numelems;
    int item, num;
    text wbuf[1024];
    int len;
    orderitem *items;

    item = 0;
    items = ctx->items;

    /* get form data */
    WRB_getParsedContent(WRBCtx, &hPBlock);
    numelems = WRB_numPBElem(WRBCtx, hPBlock);
    for (num = 0; num < numelems; num++) {
        /*
         * for each name-value pair from the form, search through
         * part numbers found in the ordered items for a duplicate--
         * if found, replace the ordered item data with the form data
         */
        elem = WRB_walkPBlock(WRBCtx, hPBlock, num);
        for (item = 0; item < ctx->numitems; item++) {
            if (!strcmp((char *)elem->szParamName,
              (char *)items[item].partnum)) {
                /* found a duplicate */
                strcpy((char *)items[item].partnum, (char *)elem-
>szParamName);
                strcpy((char *)items[item].qty, (char *)elem->szParamValue);
                elem->szParamName[0] = '\0';
                elem->szParamValue[0] = '\0';
                break;
            }
        }
    }

    /* transfer any remaining form data to items array */
    item = ctx->numitems; /* take up where we left off */
    for (num = 0; (num < numelems) && (item < MAXITEMS); num++) {
        elem = WRB_walkPBlock(WRBCtx, hPBlock, num);
        if (isdigit(elem->szParamName[0]) && elem->szParamValue[0] &&
          strcmp((char *)elem->szParamValue, "0")) {
            /* the name is a part number with a non-zero value (quantity) */
            strcpy((char *)items[item].partnum, (char *)elem->szParamName);
```

```
            strcpy((char *)items[item].qty, (char *)elem->szParamValue);
            item ++;
        }
    }
    ctx->numitems = item;

#ifdef DEBUG
    sprintf(debug, "placeOrder: numitems: %d", ctx->numitems);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    /* sort items array */
    qsort(items, ctx->numitems, sizeof(orderitem), itemcmp);

    /* begin writing response page */
    WRB_printf(WRBCtx, (text *)"Content-type: text/html\n");
    item = 0;

    /* remember the empty line before "<HTML>" */
    WRB_printf(WRBCtx, (text *)"\n");

    if (item >= MAXITEMS) {
        /* XXX orders array maxed out--write warning message */
    }

    WRB_printf(WRBCtx,
      (text *)"<HTML><HEAD><TITLE>Order Form</TITLE></HEAD>\n");
    WRB_printf(WRBCtx, (text *)"<BODY BGCOLOR=\"FFFFFF\">\n");
    WRB_printf(WRBCtx, (text *)"<H1>Placing an Order</H1>\n");
    WRB_printf(WRBCtx,
      (text *)"<P>These are the items and the quantities you have selected. ")
;
    WRB_printf(WRBCtx,
      (text *)"To change the quantity of an item you want to order, ");
    WRB_printf(WRBCtx,
      (text *)"just change the value in the quantity box for that item. ");
    WRB_printf(WRBCtx,
      (text *)"If you decide you don't want to order an item after all, ");
    WRB_printf(WRBCtx,
      (text *)"just delete the contents of the item's quantity box. ");
    WRB_printf(WRBCtx, (text *)"To add more items to your order, you can ");
    WRB_printf(WRBCtx,
      (text *)"<A HREF=\"/mywrbapp/bin/getcatalog\">return ");
    WRB_printf(WRBCtx, (text *)"to the catalog</A>.</P>\n");
    WRB_printf(WRBCtx, (text *)"<FORM ACTION=\"/mywrbapp/bin/confirm\">\n");

    /* write out the ordered items */
    ret = formatOrder(FALSE, ctx, WRBCtx);
    if (ret != WRB_DONE) {
        return (WRB_ERROR);
```

```
    }

    WRB_printf(WRBCtx,
      (text *)"<INPUT TYPE=\"submit\" VALUE=\"Confirm Order\">\n");
    WRB_printf(WRBCtx, (text *)"</FORM>\n</BODY>\n</HTML>\n");

    WRB_destroyPBlock(WRBCtx, hPBlock);
    return (WRB_DONE);
}
```

confirmOrder()

When the user edits and submits the HTML form generated by `placeOrder()`, the form's action specifies the "confirm" request. `MyWRBApp_Exec()` then calls `confirmOrder()` to handle this request. Its main task is to generate an invoice summarizing the client's order. It also signals the client browser to delete the cookies associated with the order.

```
WRBReturnCode
confirmOrder(myappctx *ctx, dvoid *WRBCtx)
{
    WRBReturnCode ret;
    WRBpBlock hPBlock;
    WRBpBlockElem *elem;
    sb4 numelems;
    int item, num;
    text wbuf[1024];
    int len;
    text *cp, *tp;
    text part[PARTNUMLEN + 1];
    orderitem *items;

    /* begin response */
    WRB_printf(WRBCtx, (text *)"Content-type: text/html\n");

    /* remember the blank line */
    WRB_printf(WRBCtx, (text *)"\n");

    WRB_printf(WRBCtx, (text *)"<HTML><HEAD><TITLE>Thank You</TITLE></
HEAD>\n");
    WRB_printf(WRBCtx, (text *)"<BODY BGCOLOR=\"FFFFFF\">\n");
    WRB_printf(WRBCtx, (text *)"<H1>Thank you for your order</H1>\n");
    WRB_printf(WRBCtx, (text *)"<P>These are the items ");
    WRB_printf(WRBCtx, (text *)"and the quantities you have ordered.</P>\n");

    /* get form data and fill in ordered items array */
    WRB_getParsedContent(WRBCtx, &hPBlock);
    numelems = WRB_numPBElem(WRBCtx, hPBlock);
```

```
    items = ctx->items;
    item = 0;
    for (num = 0; (num < numelems) && (item < MAXITEMS); num++) {
        elem = WRB_walkPBlock(WRBCtx, hPBlock, num);
        if (isdigit(elem->szParamName[0]) && elem->szParamValue[0]) {
            strcpy((char *)items[item].partnum, (char *)elem->szParamName);
            strcpy((char *)items[item].qty, (char *)elem->szParamValue);
            item ++;
        }
    }
    ctx->numitems = item;

    /* write out the order summary and total by setting invoice to TRUE */
    ret = formatOrder(TRUE, ctx, WRBCtx);
    if (ret != WRB_DONE) {
        WRB_destroyPBlock(WRBCtx, hPBlock);

        return (WRB_ERROR);
    }

    WRB_printf(WRBCtx,
      (text *)"<P><A HREF=\"/mywrbapp/bin/
getcatalog\">Return to the catalog");
    WRB_printf(WRBCtx, (text *)"</A>.</P>\n</BODY>\n</HTML>\n");

    WRB_destroyPBlock(WRBCtx, hPBlock);

    return (WRB_DONE);
}
```

formatUserData()

`getUserData()` calls `formatUserData()` to generate and send to the client an HTML form containing the registration data for the user specified by the data block `data`.

```
WRBReturnCode
formatUserData(text *user, myappctx *ctx, void *WRBCtx)
{
    dvoid *hInfoFile;
    text *field;

#ifdef DEBUG
    WRBLogMessage(WRBCtx, "formatUserData: opening infofile", 0);
#endif
    hInfoFile = WRB_CNTopenDocument(WRBCtx, ctx->hRepository, user,
      WRBCS_OPEN | WRBCS_READ);
```

```
    if (!hInfoFile) {
        return (WRB_ERROR);
    }

    /*
     * write out data form
     */
#ifdef DEBUG
    WRBLogMessage(WRBCtx, "formatUserData: starting client writes", 0);
#endif

    WRB_printf(WRBCtx, (text *)"Content-type: text/html\n\n");

    WRB_printf(WRBCtx,
      (text *)"<HTML>\n<HEAD><TITLE>Registration Form</TITLE></HEAD>\n");
    WRB_printf(WRBCtx, (text *)"<BODY BGCOLOR=\"FFFFFF\">\n");
    WRB_printf(WRBCtx,
      (text *)"<FORM ACTION=\"/mywrbapp/bin/
edituserdata\" METHOD=\"POST\">\n");

    /* username */
    field = getInfoField(NULL, hInfoFile, USERNAME, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Username: <INPUT TYPE=\"text\" NAME=\"username\" ");
    WRB_printf(WRBCtx,
      (text *)"VALUE=\"%s\" MAXLENGTH=\"10\" SIZE=\"10\"></P>\n", field);

    /* password */
    field = getInfoField(NULL, hInfoFile, PASSWORD, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Password: <INPUT TYPE=\"password\" NAME=\"password\" ");
    WRB_printf(WRBCtx, (text *)"VALUE=\"%s\" MAXLENGTH=\"10\" SIZE=\"10\">\n",
      field);

    /* confirm password */
    WRB_printf(WRBCtx,
      (text *)" Confirm Password: <INPUT TYPE=\"password\" NAME=\"confirm\" ")
;
    WRB_printf(WRBCtx,
      (text *)"VALUE=\"%s\" MAXLENGTH=\"10\" SIZE=\"10\"></P>\n", field);

    /* fullname */
    field = getInfoField(NULL, hInfoFile, FULLNAME, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Full Name: <INPUT TYPE=\"text\" NAME=\"fullname\" ", field);

    /* address */
    field = getInfoField(NULL, hInfoFile, ADDRESS, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Street Address: <INPUT TYPE=\"text\" NAME=\"address\" ");
```

```
    WRB_printf(WRBCtx, (text *)"VALUE=\"%s\"></P>\n", field);

    /* city */
    field = getInfoField(NULL, hInfoFile, CITY, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>City: <INPUT TYPE=\"text\" NAME=\"city\" ");
    WRB_printf(WRBCtx, (text *)"VALUE=\"%s\"></P>\n", field);

    /* state */
    field = getInfoField(NULL, hInfoFile, STATE, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>State: <INPUT TYPE=\"text\" NAME=\"state\" ");
    WRB_printf(WRBCtx,
      (text *)"VALUE=\"%s\" MAXLENGTH=\"2\" SIZE=\"2\"></P>\n", field);

    /* zip */
    field = getInfoField(NULL, hInfoFile, ZIP, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Zip: <INPUT TYPE=\"text\" NAME=\"zip\" ");
    WRB_printf(WRBCtx, (text *)"VALUE=\"%s\"></P>\n", field);

    /* country */
    field = getInfoField(NULL, hInfoFile, COUNTRY, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Country: <INPUT TYPE=\"text\" NAME=\"country\" ");
    WRB_printf(WRBCtx, (text *)"VALUE=\"%s\"></P>\n", field);

    /* email */
    field = getInfoField(NULL, hInfoFile, EMAIL, ctx, WRBCtx);
    WRB_printf(WRBCtx,
      (text *)"<P>Email Address: <INPUT TYPE=\"text\" NAME=\"email\" ");
    WRB_printf(WRBCtx, (text *)" SIZE=\"40\" VALUE=\"%s\"></P>\n", field);
    WRB_printf(WRBCtx, (text *)"<INPUT TYPE=\"submit\" NAME=\"submit\" ");
    WRB_printf(WRBCtx, (text *)"VALUE=\"Save Changes\">\n</FORM>\n");
    WRB_printf(WRBCtx, (text *)"<P><A HREF=\"/mywrbapp/bin/getcatalog\">");
    WRB_printf(WRBCtx, (text *)"Go to the catalog</A></P>\n");
    WRB_printf(WRBCtx, (text *)"</BODY>\n</HTML>\n");

#ifdef DEBUG
    WRBLogMessage(WRBCtx, "formatUserData: finished client writes", 0);
#endif

    WRB_CNTcloseDocument(WRBCtx, hInfoFile);

    return (WRB_DONE);
}
```

userDataError()

`newUser()` and `editUserData()` calls `userDataError()` to generate an HTML error page using the text specified by `message`. The `URL` parameter specifies the registration form to which to provide a link.

```
void
userDataError(const text *message, const text *URL, myappctx* ctx, void *WRBCt
x)
{
    text wbuf[1024];
    int len;

    WRB_printf(WRBCtx, (text *)"Content-type: text/html\n\n");
    WRB_printf(WRBCtx, (text *)"<HTML><HEAD><TITLE>Error</TITLE></HEAD>\n");
    WRB_printf(WRBCtx, (text *)"<BODY BGCOLOR=\"FFFFFF\">\n<H1>Error</H1>\n");
    WRB_printf(WRBCtx, (text *)"<P>%s</P>\n", message);
    WRB_printf(WRBCtx,
      (text *)"<P><A HREF=\"%s\">Return ", URL);
    WRB_printf(WRBCtx, (text *)"to the registration form</A>.</P>\n");
    WRB_printf(WRBCtx, (text *)"</BODY>\n</HTML>\n");

    return;
}
```

storeInfo()

`newUser()` and `editUserData()` call `storeInfo()` to parse all registration form data for a user (except username and password, which are hashed in the data file) and store it in an ASCII file on the host file system.

```
WRBReturnCode
storeInfo(text *user, myappctx *ctx, void *WRBCtx)
{
    dvoid *hInfoFile;
    text buf[256];
    text *cp;
    WRBpBlock hPBlock;
    WRBpBlockElem *elem;
    sb4 numelems;

    /* create infofile for this user */
    hInfoFile = WRB_CNTopenDocument(WRBCtx, ctx->hRepository, user,
      WRBCS_CREATE & WRBCS_WRITE);
    if (!hInfoFile) {
        return (WRB_ABORT);
    }
```

```
    /* get form data */
    WRB_getParsedContent(WRBCtx, &hPBlock);

    cp = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"fullname", -1);
    sprintf((char *)buf, "Full Name: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    cp = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"address", -1);
    sprintf((char *)buf, "Street Address: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    cp = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"city", -1);
    sprintf((char *)buf, "City: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    cp = WRB_findPBElemVal(WRBCtx, (text *)"state", hPBlock, -1);
    sprintf((char *)buf, "State: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    cp = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"zip", -1);
    sprintf((char *)buf, "Zip Code: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    cp = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"country", -1);
    sprintf((char *)buf, "Country: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    cp = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"email", -1);
    sprintf((char *)buf, "Email Address: %s\n", cp);
    WRB_CNTwriteDocument(WRBCtx, hInfoFile, buf, strlen((char *)buf));

    WRB_CNTcloseDocument(WRBCtx, hInfoFile);

    WRB_destroyPBlock(WRBCtx, hPBlock);

    return (WRB_DONE);
}
```

formatOrder()

`placeOrder()` and `confirmOrder()`call `formatOrder()` to format the client order encapsulated in the `ctx` structure in HTML format and send it to the client.

If invoice is `TRUE`, formatOrder() outputs a non-editable HTML page and prints price totals. Otherwise, it outputs an editable HTML form with `<INPUT>` fields. *formatOrder()* does *not* output a Content-type header, an HTML header, or an HTML trailer; the caller is responsible for outputting that information.

```
WRBReturnCode
formatOrder(const boolean invoice, myappctx *ctx, void *WRBCtx)
{
    dvoid *hCatFile;
    text lbuf[1024];
    text *lp;
    text wbuf[1024];
    int len;
    orderitem* items;
    int item = 0;
    double itemprice;
    double quantity;
    double total = 0;

    /* open catalog */
    hCatFile = WRB_CNTopenDocument(WRBCtx, ctx-
>hRepository, (text *)"catalog",
      WRBCS_OPEN | WRBCS_READ);
    if (!hCatFile) {
#ifdef DEBUG
        WRBLogMessage(WRBCtx, "formatOrder: failed to open catalog", 0);
#endif
        return (WRB_ABORT);
    }

    /*
     * because both the items array and the catalog file are sorted by
     * part number, loop through without seeking backward
     */
    items = ctx->items;
    while ((item < ctx->numitems)) {
        if ((items[item].qty[0] == '\0') || (items[item].qty[0] == '0')) {
            /* skip empty quantities */
            item++;
            continue;
        }

        /* skip description line, get part number line */
        myfgets(lbuf, 1024, hCatFile, WRBCtx);
        if (!myfgets(lbuf, 1024, hCatFile, WRBCtx)) {
            break;
        }
        if (!strncmp((char *)lbuf, (char *)items[item].partnum, PARTNUMLEN)) {
            /*
             * found the catalog entry for current item--
             * write it out to the client
             */
            WRB_printf(WRBCtx, (text *)"<P>");
            WRB_printf(WRBCtx, lbuf);
```

```
            if (invoice) {
                /* calculate and print price, add to subtotal */
                lp = (text *)strchr((char *)lbuf, '$');
                lp++;
                itemprice = atof((char *)lp);
                quantity = atof((char *)items[item].qty);
                WRB_printf(WRBCtx,
                  (text *)" Quantity: %s @ $%.2f = $%.2f</P>\n",
                  items[item].qty, itemprice, itemprice * quantity);

                total += itemprice * quantity;
            }
            else {
                /*
                 * not an invoice--write quantity out as an editable
                 * input field
                 */
                WRB_printf(WRBCtx,
                  (text *)" Order quantity: <INPUT TYPE=\"text\" ");
                WRB_printf(WRBCtx, (text *)"NAME=\"%s\" ", items[item].partnum
);
                WRB_printf(WRBCtx,
                  (text *)"SIZE=\"4\" VALUE=\"%s\"></P>\n", items[item].qty);
            }
            item++;
        }
    }

    if (invoice) {
        if (taxable(WRB_getRequestInfo(WRBCtx, WRBR_USER), ctx,
          WRBCtx)) {
            /* print subtotal */
            WRB_printf(WRBCtx, (text *)"<P>Subtotal: $%.2f</P>\n", total);

            /* calc and print tax */
            WRB_printf(WRBCtx, (text *)"<P>Tax @ %.2f%%: $%.2f</P>\n",
              ctx->taxpct, (ctx->taxpct/100)*total);

            /* print total with tax */
            WRB_printf(WRBCtx, (text *)"<P>Your total is: $%.2f</P>\n",
              total + (ctx->taxpct/100)*total);
        }
        else {
            /* print total (no tax) */
            WRB_printf(WRBCtx, (text *)"<P>Your total is: $%.2f</
P>\n", total);
        }
    }

    WRB_CNTcloseDocument(WRBCtx, hCatFile);
```

```
    return (WRB_DONE);
}
```

taxable()

`taxable()` uses the client's registration data and the cartridge configuration parameters to determine whether the client lives in our home state, and hence should be charged sales tax.

```
boolean
taxable(text *user, myappctx *ctx, void *WRBCtx)
{
    text *state;

    /* get user's home state */
    state = getInfoField(user, NULL, (text *)"State", ctx, WRBCtx);

    /* compare it with our home state */
    if (state && ctx->state && !strcmp((char *)state, (char *)ctx->state)) {
        /* they're in our state--tax 'em */
#ifdef DEBUG
        WRBLogMessage(WRBCtx, "taxable: returning TRUE", 0);
#endif
        return (TRUE);
    }

#ifdef DEBUG
        WRBLogMessage(WRBCtx, "taxable: returning FALSE", 0);
#endif
    return (FALSE);
}
```

itemcmp()

`placeOrder()` passes a pointer to `itemcmp()` to the `qsort`(3) library function to use in sorting the `orderitems` array that encapsulates a client's order.

```
int
itemcmp(const void* item1, const void* item2)
{
    return(strcmp((char *)((orderitem *)item1)->partnum,
      (char *)((orderitem *)item2)->partnum));
}
```

getInfoField()

`getInfoField()` returns the value of a specified info file field. The info file is specified either by user name, or by a pointer to an open info file.

```
text *
getInfoField(text *user, dvoid *hInfoFile, text *field, myappctx *ctx,
  dvoid *WRBCtx)
{
    text buf[1024];
    text *cp = NULL;

    if (!hInfoFile) {
        hInfoFile = WRB_CNTopenDocument(WRBCtx, ctx->hRepository, user,
          WRBCS_OPEN & WRBCS_READ);
    }

    if (!hInfoFile) {
        return (NULL);
    }

    /* get the requested field value from the info file */
    while (myfgets(buf, 1024, hInfoFile, WRBCtx)) {
        if (!strncmp((char *)buf, (char *)field, strlen((char *)field))) {
            cp = (text *)strchr((char *)buf, ':');
            cp += 2;
            break;
        }
    }

    WRB_CNTcloseDocument(WRBCtx, hInfoFile);
    return ((text *)strdup((char *)cp));
}
```

myfgets()

`myfgets()` is a utility function that reads a line at a time from the specified open document in a content repository. It behaves similarly to fgets(3), but uses the WRB content service function `WRBGetParsedContent()` to read from the specified content repository document.

```
text *
myfgets(text *buf, int size, dvoid *hFile, dvoid *WRBCtx)
{
    text byte[1];
    int i = 0;

    while (i < (size - 1)) {
```

```
        if (WRB_CNTreadDocument(WRBCtx, hFile, byte, 1)) {
            if (byte[0] == '\n') {
                break;
            }
            buf[i++] = byte[0];
        }
    }
    buf[i] = '\0';

    if (i) {
        return (buf);
    }

    return (NULL);
}
```

uploadCatalog()

uploadCatalog() creates the catalog document in the content repository from data stored in the local file system. This is just a convenience to set up the database content for you the first time you try out MyWRBApp.

```
WRBReturnCode
uploadCatalog(myappctx *ctx, dvoid *WRBCtx)
{
    dvoid *hCatFile;
    FILE *file;
    text *ohome;
    WAPIReturnCode ret;
    text buf[1024];

    ohome = WRB_getORACLE_HOME(WRBCtx);
    if (!ohome) {
        /* ORACLE_HOME is not set */
        return (WRB_ABORT);
    }

    sprintf((char *)buf, "%s/ows/3.0/sample/wrbsdk/mywrbapp/catalog.txt",
      (char *)ohome);
    file = fopen((char *)buf, "r");

    if (!file) {
        /* catalog file is not present */
        return (WRB_ABORT);
    }

    hCatFile = WRB_CNTopenDocument(WRBCtx, ctx-
>hRepository, (text *)"catalog",
```

```
      WRBCS_CREATE & WRBCS_WRITE);

    /* upload catalog file to database one line at a time */
    while (fgets((char *)buf, 1024, file)) {
        WRB_CNTwriteDocument(WRBCtx, hCatFile, buf, strlen((char *)buf));
    }

    WRB_CNTcloseDocument(WRBCtx, hCatFile);
    fclose(file);

    return (WRB_DONE);
}
```

CHAPTER

6 Document Server Cartridge Example

This chapter describes the Document Server cartridge, which is a custom cartridge that uses the Content Service APIs to manage documents in a database repository. It also uses the ICX (Intercartridge Service) APIs to retrieve documents from the Web.

This chapter contains the following sections:

Features of the Document Server Cartridge

The Document Server cartridge allows users to store and manage documents in a database repository. Users can:

- View a list of documents in the repository.
- Select a document to download.
- Upload documents to the repository from their local file system or from a URL.

- View the attributes associated with a document.
- Set attributes on a document.

Users interact with the cartridge through HTML forms. The contents of these forms are generated by the code in the cartridge.

Main Screen

When the user first invokes the cartridge, the main screen appears (Figure 6-1).

Figure 6-1: Main screen of the Document Server cartridge

The main screen is a frameset: the left frame shows the contents of the repository, and the right frame is a splash screen. When the user selects an operation (download a document for viewing, upload a document, set attributes, or get attributes), the right frame changes to display the appropriate page.

Downloading Documents

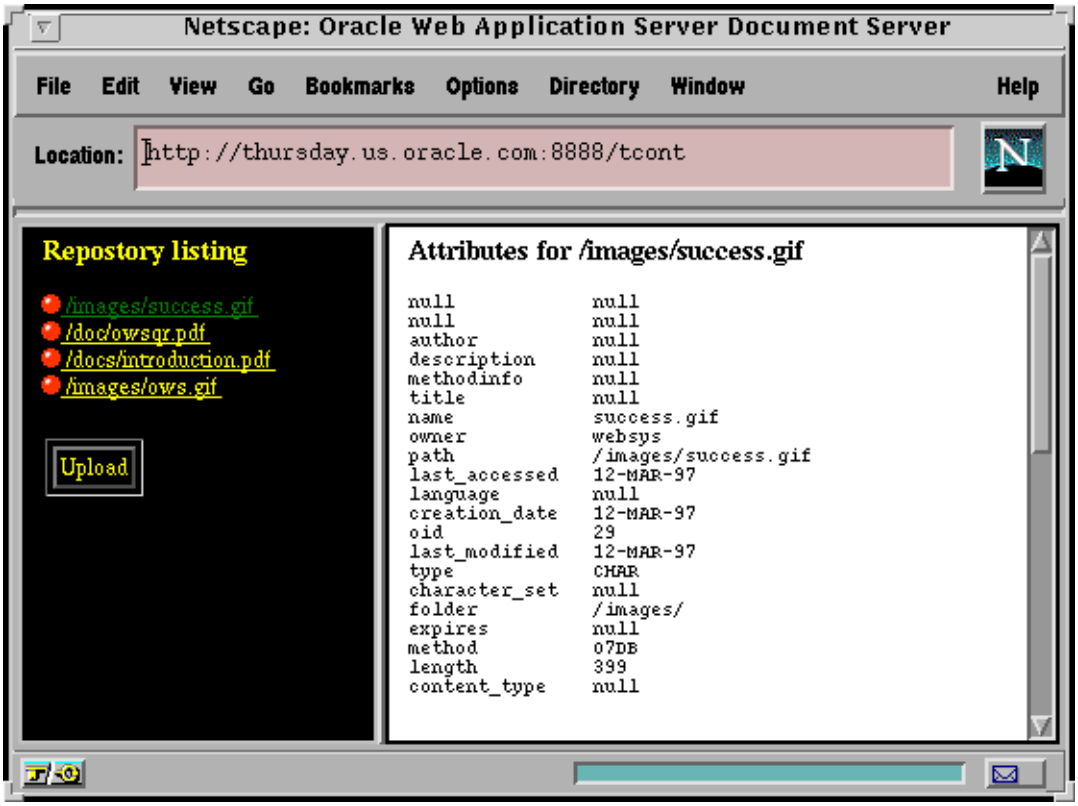
To download a document from the repository, the user clicks the document from the "Repository listing" on the left frame. The selected document is fetched and displayed in the right frame.

Setting and Viewing Attributes

Documents in the repository have attributes, and each attribute has a value associated with it. For example, a document could have an "owner" attribute, the value of which would be the document owner's name.

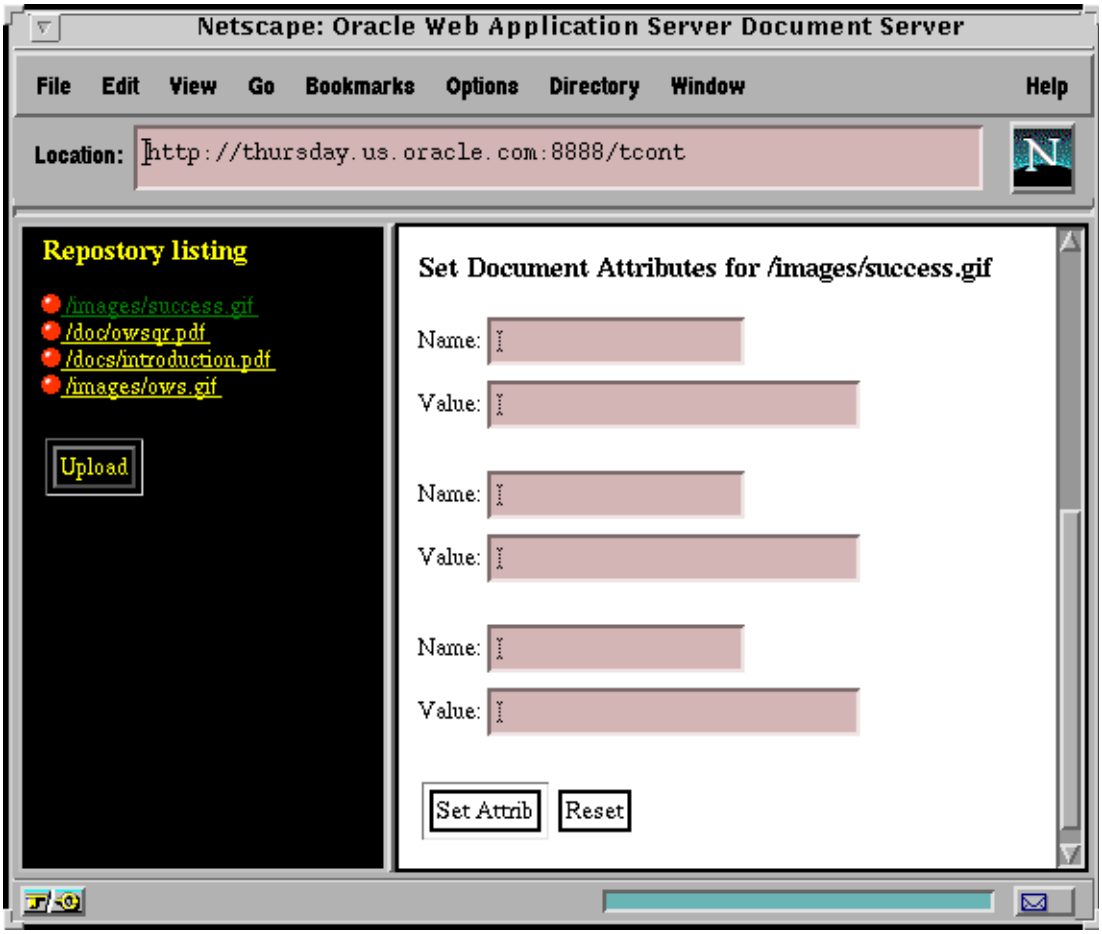
To view the attributes associated with a document, the user clicks the red circle to the left of the document's name. The right frame displays a page that looks like the following:

Figure 6-2: Viewing attributes in the Document Server cartridge



The bottom part of the page in the right frame contains entry fields where the user can define a new attribute or change the value of an existing attribute.

Figure 6-3: Setting attributes in the Document Server cartridge



Uploading Documents

To upload a document to the repository, the user clicks the Upload button to view the Upload page. The top part of the page is for a “local upload”, where the user uploads a document from his file system. The bottom part of the page is for a “remote upload”,

where the user uploads a document on the Web. Remote uploads are done by issuing ICX (intercartridge exchange service) requests.

Figure 6-4: Uploading local documents in the Document Server cartridge

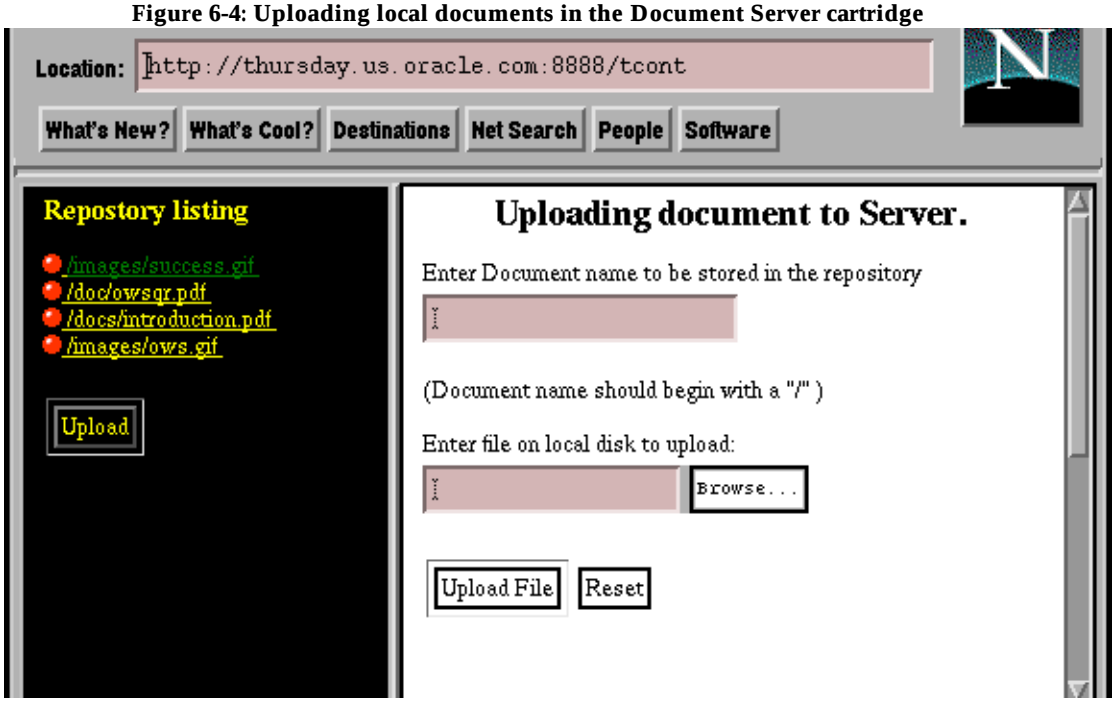
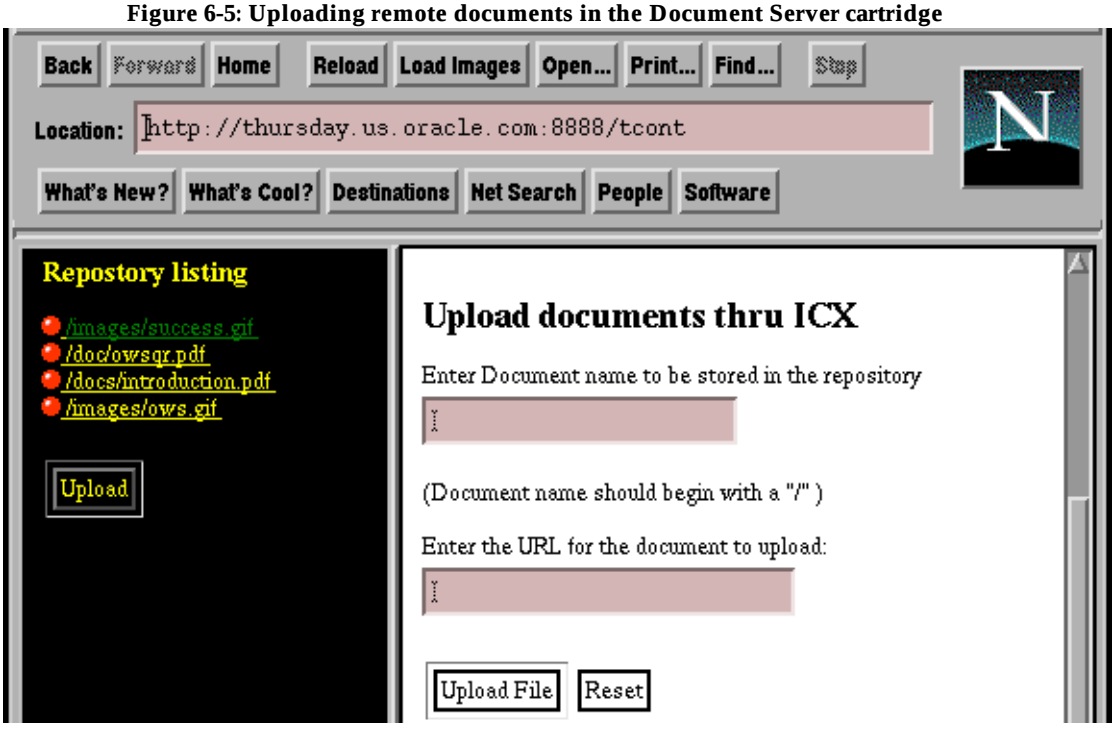


Figure 6-5: Uploading remote documents in the Document Server cartridge



Cartridge Registration

The cartridge is registered with the following information:

Parameter	Value
Virtual Path	`/tcont`
Entry-Point function	`testentry`

The virtual path can be changed in the registration of the cartridge without affecting the cartridge's code, because it is not hardcoded in the code. Instead, the code calls `WRB_getRequestInfo(WRBctx, WRBR_VIRTUALPATH)` to get the virtual path.

Cartridge Invocation and Flow Process

To invoke the cartridge, the user enters the following URL and the main screen appears (Figure 6-1):

```
http://machine:port/tcont
```

When the cartridge starts up for the first time, it executes the Entry-Point function and the Init function. These functions define the callback functions and connect to the repository. See The Callback Functions for details.

The cartridge then handles the request in the Exec function, which checks for cartridge-defined keywords such as "upload" and "download" in the URL. In this case of the first request, the URL does not contain any keywords and the default case in the `switch` statement is executed. The default case calls `content_frameset()`, which creates the two pages in the frameset. The page in the left frame is generated by `content_list()`, and the page in the right frame is generated by `content_main()`.

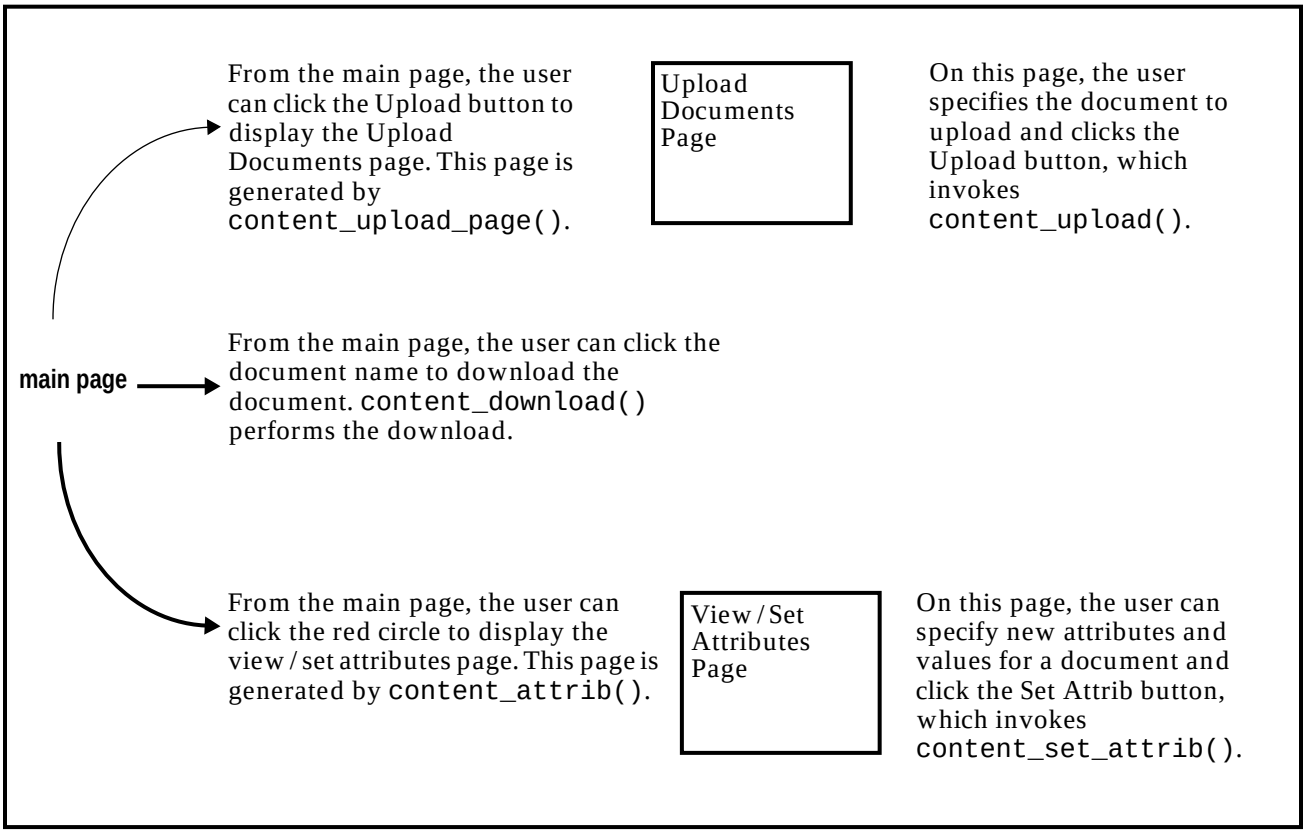
`content_list()` retrieves a list of documents in the repository and generates an HTML page that contains the following:

- For each document in the list, the function generates a red circle that is a link and the name of the document, that is also a link. When a user clicks the red circle, the cartridge displays the attributes of the document. When the user clicks the document name, the document is downloaded from the repository.
- An "Update" button to display the Upload Page.

`content_main()` displays a static splash page.

To submit requests to the cartridge, the user clicks on links in the generated HTML pages; the user does not manually type in the URL.

Here is a schematic of the process flow:



The Callback Functions

The Document Server cartridge implements the following callback functions:

- The Entry-Point Function
- The Init Function
- The Exec Function
- The Shutdown Function

The Entry-Point Function

All WRB cartridges must implement an entry-point function to initialize a function dispatch table that the WRB engine uses to call functions defined in the cartridge. The entry-point function is called only once, when the cartridge first starts up.

For the Document Server cartridge, the entry-point function is `testentry`:

```
WRBReturnCode testentry (WRBCallbacks *WRBCalls)
{
   WRBCalls->init_WRBCallback = content_init;
   WRBCalls->exec_WRBCallback = content_exec;
   WRBCalls->shut_WRBCallback = content_shut;
   return WRB_DONE;
}
```

The Init Function

After the entry-point function returns, the WRB engine calls the cartridge's Init function to perform any other one-time setup that the cartridge needs. The Init function may allocate an application context structure of any type and pass a pointer to it back to the WRB engine for that WRBX. The WRB engine subsequently passes this pointer to every cartridge function that runs in that WRBX.

For the Document Server cartridge, the Init function is `content_init`:

```
typedef struct
{
  dvoid *hRepository;
  dvoid *hDoc;
  text  *szProxy;
  text  *szNoProxyOn;
  text  *szVirtualPath;
} content_struct;

WRBReturnCode content_init (void *WRBCtx, void **clientcxp)
{
   content_struct *cs;
   text *cartridge = WRB_getCartridgeName(WRBCtx);

   cs = (content_struct *)malloc(sizeof(content_struct));
   if (!cs) {
      content_debug("Error in malloc exiting\n");
      return WRB_ABORT;
   }

   if (!(cs->hRepository = WRB_CNTopenRepository(WRBCtx,
               NULL, NULL, (text *)"WEBSYS"))) {
      content_debug("Error in open repository. \
                        Refer to wrb.log for additional info\n");
      return WRB_ABORT;
   }

   if (!(cs->szProxy = WRB_getAppConfigVal(WRBCtx, cartridge,
                                          (text *)"http_proxy")))
      cs->szProxy = (text *)"http://oracle-proxy.us.oracle.com";

   if (!(cs->szNoProxyOn = WRB_getAppConfigVal(WRBCtx, cartridge,
                                              (text *)"no_proxy")))
      cs->szNoProxyOn = "us.oracle.com";

   *clientcxp = cs;
   return (WRB_DONE);
}
```

The Init function allocates an application context structure (`cs`) that the cartridge uses to keep track of the repository and current document.

The cartridge opens the repository in the Init function. By doing this in the Init function, the cartridge avoids the overhead of opening and closing the repository on each request.

The first parameter of `WRB_CNTopenRepository()` is the application context, the second and third parameters specify the username and password to use to log into the repository (these are NULL because these are specified in the fourth parameter), and the fourth parameter specifies connection information. “WEBSYS” is the name of a DAD (database access descriptor), which specifies the database to which to connect, the username and password to use for the login, and NLS information. The return value of `WRB_CNTopenRepository()` is a pointer to the opened repository, and it is stored in the client context.

The Exec Function

After the cartridge is running, the Web Dispatcher can direct requests to it, and the Exec function of the cartridge is invoked to handle the requests. Typically, the Exec function parses the URL to determine what action to take. For this cartridge, the Exec function calls `content_uri_type()` to read the URL and return

For the Document Server, the Exec function is `content_exec`:

```
WRBReturnCode content_exec(void  *WRBCtx, void *clientcxp)
{
  WRBpBlock  pBlock;
  text      *file = NULL;
  text      *uri = WRB_getRequestInfo(WRBCtx, WRBR_URI);
  ub2        uritype = content_uri_type(uri);
  FILE      *fd;
  char       buf[2048], filename[256];
  ub4        bytes_read;
  time_t     tloc = 0;

  content_struct *cs = (content_struct *)clientcxp;

  if (!(cs->szVirtualPath = WRB_getRequestInfo(WRBCtx,
                              WRBR_VIRTUALPATH)))
      cs->szVirtualPath = (text *)"/tcont";

  switch (uritype) {

  case CONTENT_UPLOAD:
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    WRB_printf(WRBCtx, "<HTML>\n");
    WRB_printf(WRBCtx, "<BODY bgcolor=\"black\" text=\"yellow\"\
link=\"yellow\" vlink=\"green\" >");
    if (content_upload(WRBCtx, cs, FALSE))
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, "<BR>Document successfully uploaded\n");
    }
```

```
    else
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, “<BR>Error in uploading\n”);
    }
    WRB_printf(WRBCtx, “</HTML>\n”);
    break;

  case CONTENT_ICX:
    WRB_printf(WRBCtx, “Content-type: text/html\n\n”);
    if (content_upload(WRBCtx, cs, TRUE))
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, “<BR>Document successfully uploaded\n”);
    }
    else
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, “<BR>Error in uploading\n”);
    }
    break;

  case CONTENT_DOWNLOAD:
    if (!content_download(WRBCtx, cs))
      WRB_printf(WRBCtx, “Error in downloadinf\n”);
    break;

  case CONTENT_LIST:
    WRB_printf(WRBCtx, “Content-type: text/html\n\n”);
    content_list(WRBCtx, cs);
    break;

  case CONTENT_MAIN:
    WRB_printf(WRBCtx, “Content-type:text/html\n\n”);
    content_main(WRBCtx, cs);
    break;

  case CONTENT_UPLOAD_PAGE:
    WRB_printf(WRBCtx, “Content-type: text/html\n\n”);
    content_upload_page(WRBCtx, cs);
    break;

  case CONTENT_ATTRIB:
    content_attrib(WRBCtx, cs);
    break;

  case CONTENT_SETATTRIB:
    content_set_attrib(WRBCtx, cs);
```

```
      content_attrib(WRBCtx, cs);
      break;

    case CONTENT_DELETE:
      WRB_printf(WRBCtx, “Content-type: text/html\n\n”);
      WRB_printf(WRBCtx, “<HTML>\n”);
      WRB_printf(WRBCtx, “<BODY bgcolor=\”black\” text=\”yellow\”\
  link=\”yellow\” vlink=\”green\” >”);
      content_delete(WRBCtx, cs);
      content_list(WRBCtx, cs);
      break;

    default:
      WRB_printf(WRBCtx, “Content-type: text/html\n\n”);
      content_frameset(WRBCtx, cs);
      break;
    }

    return (WRB_DONE);
  }
```

`content_uri_type()` is defined as:

```
  ub2 content_uri_type(text *uri)
  {
    if (strstr((char *)uri, “tcont_upload_page”))
      return (CONTENT_UPLOAD_PAGE);
    else if (strstr((char *)uri, “tcont_upload_icx”))
      return (CONTENT_ICX);
    else if (strstr((char *)uri, “tcont_upload”))
      return (CONTENT_UPLOAD);
    else if (strstr((char *)uri, “tcont_download”))
      return (CONTENT_DOWNLOAD);
    else if (strstr((char *)uri, “tcont_list”))
      return (CONTENT_LIST);
    else if (strstr((char *)uri, “tcont_main”))
      return (CONTENT_MAIN);
    else if (strstr((char *)uri, “tcont_set_attrib”))
      return (CONTENT_SETATTRIB);
    else if (strstr((char *)uri, “tcont_attrib”))
      return (CONTENT_ATTRIB);
    else if (strstr((char *)uri, “tcont_delete”))
      return (CONTENT_DELETE);

    return 0;
  }
```

The Exec function:

- gets the URI of the request by calling `WRB_getRequestInfo()`
- calls `content_uri_type()` to look for keywords in the URI
- invokes the appropriate function to perform the operation based on the return value of `content_uri_type()`. For example, if the URI contains the keyword "download", `content_uri_type()` returns `CONTENT_DOWNLOAD`, and the `content_download()` function is executed.

The Shutdown Function

Before terminating a cartridge instance, the WRB engine calls the cartridge's Shutdown function. The Shutdown function performs clean-up operations, such as freeing any resources allocated in the Init function. When the Shutdown function returns, the cartridge instance may terminate at any time.

For the Document Server cartridge, the Shutdown function is `content_shut`:

```
WRBReturnCode content_shut(void *WRBCtx, void *clientcxp)
{
  content_struct *cs = (content_struct *)clientcxp;

  WRB_CNTcloseRepository(WRBCtx, cs->hRepository);
  return (WRB_DONE);
}
```

The Shutdown function closes the repository and frees the client context.

The Upload Page Function

If the URI contains "upload_page", the Exec function calls `content_upload_page()`. This function generates a static page that enables the user to upload documents to the repository. The values of the ACTION attributes are URLs that upload the documents to the repository. "`/tcont/upload`" invokes `content_upload()` and "`/tcont/upload_icx`" invokes `content_upload_icx()`.

Source for content_upload_page()

```
boolean content_upload_page(dvoid *WRBCtx, content_struct *cs)
{
  WRB_printf(WRBCtx,
"<HTML>\n                                    \
<HEAD>\n                                    \
<TITLE>Upload Page</TITLE>\n                \
</HEAD>\n                                   \
<BODY bgcolor =\"white\">\n                 \
<center><h2>Uploading document to Server.</h2></center>\n\
<form ENCTYPE=multipart/form-data method=POST           \
action=%s/tcont_upload target=\"leftnav\">\n              \
```

```
Enter Document name to be stored in the repository       \
<input type=text name=file size=25><p>\n                 \
(Document name should begin with a \"/\" )<p>            \
Enter file on local disk to upload:                      \
<input type=file name=file_data><p>\n                    \
<INPUT TYPE=SUBMIT VALUE=\"Upload File\">\n              \
<INPUT TYPE=RESET VALUE=Reset>\n                         \
</FORM>\n", cs->szVirtualPath );


  WRB_printf(WRBCtx, "<BR><BR>\n                         \
<h2> Upload documents thru ICX</h2>\n                    \
<form method=POST action=%s/tcont_upload_icx target=\"leftnav\">\n\
Enter Document name to be stored in the repository        \
<input type=text name=file size=25><p>\n                  \
(Document name should begin with a \"/\" )<p>            \
Enter the URL for the document to upload:                 \
<input type=text name=url size=30><p>\n                   \
<INPUT TYPE=SUBMIT VALUE=\"Upload File\">\n               \
<INPUT TYPE=RESET VALUE=Reset>\n                          \
</form>\n", cs->szVirtualPath);

  return TRUE;
}
```

The Upload Document (Local and Remote) Function

If the URI contains "`upload`" or "`upload_icx`", the Exec function calls `content_upload()`, which performs the following:

1. Determines if the document to be uploaded is local or remote by checking the *icx* parameter. For local uploads, the parameter is `FALSE`, while for remote uploads, the parameter is `TRUE`.
2. Gets the document's name and data.
   - For local uploads, `content_upload()` calls WRB_getMultipartData() repeatedly to get the data from the `file` and `file_data` fields.
   - For remote uploads, `content_upload()` calls `WRB_getParsedContent()` to convert the request header information into a parameter block. It then calls `WRB_findPBElemVal()` to get the value of the "`file`" and "`url`" fields. The `file` field contains the name under which to store the remote document in the repository.

     `content_upload()` then calls `WRB_ICXcreateRequest()` to create a request with the specified URL. The request is then sent by `WRB_ICXmakeRequest()`. The contents of the remote document is stored in *text_buffer*.
3. Creates a new document in the repository and make it writable. The cartridge does this by calling `WRB_CNTopenDocument()` with the `WRBCS_CREATE | WRBCS_WRITE` flags.

4. Writes the contents of the local or remote file into the document by calling `WRB_CNTwriteDocument()`.
5. Closes the document in the repository by calling `WRB_CNTcloseDocument()`.

After calling `content_upload()`, the Exec function updates the list of documents shown in the browser by calling the `content_list()` function.

Source for content_upload()

```
boolean
content_upload(dvoid *WRBCtx, content_struct *cs, boolean icx)
{
  text *file = NULL, *url = NULL;
  WRBpBlockElem multiPart;
  WRBpBlock     content;
  ub1 *mpData;
  ub4  mpDataLen;
  ub1 *text_buffer;
  ub4  buflen;
  dvoid *hReq;
  text  *errmsg;
  ub4    errmsgl;
  sword  errnum;

  if (icx)
  {
    if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    {
      WRB_printf(WRBCtx, “Error in parsing querystring\n”);
      return FALSE;
    }

    if (!(file = WRB_findPBElemVal(WRBCtx, content, (text *)”file”,
                 -1)))
    {
      WRB_printf(WRBCtx, “File name not specified\n”);
      return FALSE;
    }

    if (!(url = WRB_findPBElemVal(WRBCtx, content, (text *)”url”,
                 -1)))
    {
      WRB_printf(WRBCtx, “File name not specified\n”);
      return FALSE;
    }

    WRB_ICXsetProxy(WRBCtx, cs->szProxy);
    WRB_ICXsetNoProxy(WRBCtx, cs->szNoProxyOn);

    hReq = WRB_ICXcreateRequest(WRBCtx, url);
```

```
  if (WRB_ICXmakeRequest(WRBCtx, hReq, (dvoid **)&text_buffer,
          &buflen, 0, FALSE) != WRB_SUCCESS)
  {
   if (!WRB_getPreviousError(WRBCtx, &errnum, &errmsg, &errmsgl))
 errmsg = “Unknown error”;

    WRB_printf(WRBCtx, “Error in ICX request<BR><PRE>%s</PRE>”,
     errmsg);
    return FALSE;
  }
}
else
{
  while (WRB_getMultipartData(WRBCtx, &multiPart, &mpData,
         &mpDataLen) == WRB_SUCCESS)
  {
    if (!strcmp(multiPart.szParamName, “file”))
    {
 mpData[mpDataLen] = ‘\0’;
 file = (text *)mpData;
    }
    else if (!strcmp(multiPart.szParamName, “file_data”))
    {
 text_buffer = mpData;
 buflen = mpDataLen;
    }
  }
}

if (!file)
{
  WRB_printf(WRBCtx, “Filename not specified\n”);
  return FALSE;
}

if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
                     file, WRBCS_CREATE | WRBCS_WRITE)))
{
  content_debug(“Error creating document %s\n”, file);
  WRB_printf(WRBCtx, “Error creating document %s\n”, file);
  return FALSE;
}

if (WRB_CNTwriteDocument(WRBCtx, cs->hDoc, text_buffer,
         buflen) <= 0)
{
  content_debug(“Error in writing to repostory %s\n”,
      file);
```

```
    WRB_printf(WRBCtx, "Error in writing to repostory %s\n",
          file);
    return FALSE;
  }

  WRB_CNTcloseDocument(WRBCtx, cs->hDoc);
  cs->hDoc = NULL;

  return TRUE;
}
```

The Download Document Function

If the URI contains "`download`", the Exec function calls `content_download()`.

`content_download()` performs the following:

1. Converts the header information into a parameter block using `WRB_getParsedContent()`.
2. Gets the value of the `file` field by calling `WRB_findPBElemVal()`. The `file` field is set by `content_list()`.
3. Gets the MIME type associated with the file extension.
   a. Gets the extension of the file (using `content_extension()`).
   b. Gets a list of extensions supported by the Listener by calling `WRB_getListenerInfo()`.
   c. Checks that the Listener has a MIME type for that extension. If the Listener does not know about the extension, set the MIME type to "`text/plain`".
4. Opens the document in the repository using `WRB_CNTopenDocument()`.
5. Generates a "`Content-type:`" header line with the MIME type.
6. Reads the opened document using `WRB_CNTreadDocument()`. This function places the data into *buf*, which is then written to the requestor with `WRB_write()`.
7. Closes the document with `WRB_CNTcloseDocument()`.
8. Frees the *content* parameter block.

Source for content_download()

```
boolean content_download(dvoid *WRBCtx, content_struct *cs)
{
  text *file, *extn, *mimetype;
  WRBpBlock content;
  sb4  bytes_read;
  text buf[10240];
  WRBpBlock mimetypes;
```

```
  if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    return FALSE;

  if (!(file = WRB_findPBElemVal(WRBCtx, content, PARAM_FILENAME,
                 -1)))
  {
    content_debug("File name not specified for upload op\n");
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    WRB_printf(WRBCtx, "File name not specified\n");
    return FALSE;
  }

  extn = content_extension(file);
  mimetypes = WRB_getListenerInfo(WRBCtx, WRBL_MIMETYPES);
  if (!(mimetype = WRB_findPBElemVal(WRBCtx, mimetypes, extn, -1)))
    mimetype = (text *)"text/plain";

  if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
                   file, WRBCS_OPEN | WRBCS_READ)))
  {
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    content_debug("Error opening document %s\n", file);
    WRB_printf(WRBCtx, "Error opening document %s\n", file);
    return FALSE;
  }

  WRB_printf(WRBCtx, "Content-type: %s\n\n", mimetype);
  while ((bytes_read =
       WRB_CNTreadDocument(WRBCtx, cs->hDoc, (ub1 *)buf,
                     10240)) > 0)
    WRB_write(WRBCtx, buf, bytes_read);

  WRB_CNTcloseDocument(WRBCtx, cs->hDoc);
  WRB_destroyPBlock(WRBCtx, content);

  cs->hDoc = NULL;

  return TRUE;
}
```

The Display Attributes Function

If the URI contains "`attrib`", the Exec function calls `content_attrib()`, which performs the following:

1. Gets the header information by calling `WRB_getParsedContent()`.
2. Gets the document from the `file` field.

3. Opens the document using `WRB_CNTopenDocument()`.
4. Creates a parameter block called *attributes* and calls `WRB_CNTgetAttributes()` to write the attributes to that parameter block.
5. Loops through the contents of the parameter block and writes the attribute name and attribute value to the requestor.
6. Generates the HTML form fields that enable the user to enter new attributes. The names of the fields are used by `content_set_attrib()`.
7. Closes the document with `WRB_CNTcloseDocument()`.
8. Cleans up the *content* and *attributes* parameter blocks.

Source for content_attrib()

```
boolean content_attrib(dvoid *WRBCtx, content_struct *cs)
{
  WRBpBlock      attributes, content;
  WRBpBlockElem *attr_elem;
  dvoid         *pos;
  text          *file;

  WRB_printf(WRBCtx, "Content-type: text/html\n\n");
  WRB_printf(WRBCtx, "<HTML>\n");
  WRB_printf(WRBCtx, "<BODY bgcolor=\"white\">\n");

  if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    goto exit_attrib;

  if (!(file = WRB_findPBElemVal(WRBCtx, content, PARAM_FILENAME,
                -1)))
  {
    content_debug("File name not specified for upload op\n");
    WRB_printf(WRBCtx, "File name not specified\n");
    goto exit_attrib;
  }

  if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
                   file, WRBCS_OPEN | WRBCS_READ)))
  {
    content_debug("Error opening document %s\n", file);
    WRB_printf(WRBCtx, "Error opening document %s\n", file);
    goto exit_attrib;
  }

  attributes = WRB_createPBlock(WRBCtx);
  if (WRB_CNTgetAttributes(WRBCtx, cs->hDoc, attributes) < 0)
    goto exit_attrib;

  WRB_printf(WRBCtx, "<h3>Attributes for %s </h3>\n", file);
  WRB_printf(WRBCtx, "<PRE>\n");
```

```
  for (attr_elem = WRB_firstPBElem(WRBCtx, attributes, &pos);
       attr_elem;
       attr_elem = WRB_nextPBElem(WRBCtx, attributes, pos))
  {
    WRB_printf(WRBCtx, "%-15s\t%s\n",
            null(attr_elem->szParamName),
            null(attr_elem->szParamValue));
  }

  WRB_printf(WRBCtx, "</PRE>\n");
  WRB_printf(WRBCtx, "<BR><BR><P><h3> Set Document Attributes for
                      %s</h3>\n", file);

  WRB_printf(WRBCtx,
"<FORM METHOD=GET action=%s/tcont_set_attrib> \n\
<input type=hidden name=file value=%s> \n\
Name: <input type=text name=name1 size=20> <br>\n\
Value: <input type=text name=value1 size=30> <p> \n\
Name: <input type=text name=name2 size=20> <br>\n\
Value: <input type=text name=value2 size=30> <p> \n\
Name: <input type=text name=name3 size=20> <br>\n\
Value: <input type=text name=value3 size=30> <p>\n\
<INPUT TYPE=SUBMIT VALUE=\"Set Attrib\">       \n\
<INPUT TYPE=RESET VALUE=Reset>                 \n\
</FORM>\n",  cs->szVirtualPath, file);

  WRB_printf(WRBCtx, "</HTML>\n");

  WRB_CNTcloseDocument(WRBCtx, cs->hDoc);
  cs->hDoc = NULL;

  WRB_destroyPBlock(WRBCtx, content);
  WRB_destroyPBlock(WRBCtx, attributes);
  return TRUE;

exit_attrib:
  WRB_printf(WRBCtx, "Error in listing attributes\n");
  WRB_printf(WRBCtx, "</HTML>\n");
  return FALSE;
}
```

The Set Attributes Function

If the URI contains "`set_attrib`", the Exec function calls `content_set_attrib()`, which performs the following:

1. Creates a parameter block (called *attributes*) in which it will write the new attribute names and values.
2. Gets the document name for which the attributes will be set.
   a. Retrieve the header information from the request and save it in a parameter block (called *request*). The header information also contains the names of the new attributes and their values.
   b. Search the parameter block for the `file` field, which contains the document name. The `file` field is set by `content_list()`.
3. Opens the document using `WRB_CNTopenDocument()`.
4. Gets the names of the attributes and their values from the *request* parameter block. The names of the fields are the names of the HTML form elements, which were set in `content_attrib()`.
5. Writes the attributes and their values to the *attributes* parameter block.
6. Sets the attributes in the repository using `WRB_CNTsetAttributes()`.
7. Closes the document using `WRB_CNTcloseDocument()`.
8. Free the *attributes* and *request* parameter blocks.

After calling `content_set_attrib()`, the Exec function calls `content_attrib()` to update the list of attributes for the document.

Source for content_set_attrib()

```
boolean content_set_attrib(dvoid *WRBCtx, content_struct *cs)
{
  WRBpBlock      attributes = NULL;
  WRBpBlock      request;
  text          *name1 = NULL, *name2 = NULL, *name3 = NULL;
  text          *value1= NULL, *value2 = NULL, *value3 = NULL;
  text          *file = NULL;
  boolean        rval = FALSE;
  text           *errmsg;
  ub4            errmsgl;
  sword          errnum;

  if (!(attributes = WRB_createPBlock(WRBCtx)))
    goto exit_set_attrib;

  if (WRB_getParsedContent(WRBCtx, &request) != WRB_SUCCESS)
    goto exit_set_attrib;

  if (!(file = WRB_findPBElemVal(WRBCtx,request, PARAM_FILENAME,
               -1)))
  {
    content_debug("File name not specified for upload op\n");
    WRB_printf(WRBCtx, "File name not specified\n");
    goto exit_set_attrib;
  }
```

```
  if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
                file, WRBCS_OPEN | WRBCS_READ)))
  {
    content_debug("Error opening document %s\n", file);
    WRB_printf(WRBCtx, "Error opening document %s\n", file);
    goto exit_set_attrib;
  }

  name1 = WRB_findPBElemVal(WRBCtx, request, "name1", -1);
  value1 = WRB_findPBElemVal(WRBCtx, request, "value1", -1);
  name2 = WRB_findPBElemVal(WRBCtx, request, "name2", -1);
  value2 = WRB_findPBElemVal(WRBCtx, request, "value2", -1);
  name3 = WRB_findPBElemVal(WRBCtx, request, "name3", -1);
  value3 = WRB_findPBElemVal(WRBCtx, request, "value3", -1);

  if (name1 && value1)
    WRB_addPBElem(WRBCtx, attributes, name1, -1, value1, -1, 0);

  if (name2 && value2)
    WRB_addPBElem(WRBCtx, attributes, name2, -1, value2, -1, 0);

  if (name3 && value3)
    WRB_addPBElem(WRBCtx, attributes, name3, -1, value3, -1, 0);

  if (WRB_CNTsetAttributes(WRBCtx, cs->hDoc, attributes) !=
                  WRB_SUCCESS)
  {
    if (!WRB_getPreviousError(WRBCtx, &errnum, &errmsg, &errmsgl))
      errmsg = "Unknown error";

    WRB_printf(WRBCtx, "Error in setting attributes
<BR><PRE>%s</PRE>\n", errmsg);
    goto exit_set_attrib;
  }

  rval = TRUE;

exit_set_attrib:
  if (cs->hDoc) WRB_CNTcloseDocument(WRBCtx, cs->hDoc);

  if (attributes) WRB_destroyPBlock(WRBCtx, attributes);

  if (request) WRB_destroyPBlock(WRBCtx, request);

  return rval;
}
```

The List Documents Function

The `content_list()` function displays a list of documents in the repository. For each document in the list, it generates a red circle, which is a link that displays the document's attributes, and the document name, which is also a link that downloads the document. The `file` field is used by `content_download()` and `content_attrib()`.

Source for content_list()

```
boolean content_list(dvoid *WRBCtx, content_struct *cs)
{
  WRBpBlock      docList;
  WRBpBlockElem *docListElem;
  dvoid         *pos;
  text          *escaped_str;

  if (!WRBCtx || !cs)
    return FALSE;

  WRB_printf(WRBCtx, "<BODY bgcolor=\"black\" text=\"yellow\"\
 link=\"yellow\" vlink=\"green\" >");

  WRB_printf(WRBCtx, "<H3> Repostory listing </H3>\n");

  if (WRB_CNTlistDocuments(WRBCtx, cs->hRepository, &docList) <= 0)
  {
    WRB_printf(WRBCtx, "Repository contains no documents<BR>");
    goto exit_list;
  }

  for (docListElem = WRB_firstPBElem(WRBCtx, docList, &pos);
       docListElem;
       docListElem =  WRB_nextPBElem(WRBCtx, docList, pos))
  {
    escaped_str = content_escape(docListElem->szParamName);
    WRB_printf(WRBCtx,
            "<A HREF = \"%s/tcont_attrib?file=%s\" \
               target = \"main\"> \
               <img src=\"/ows-img/dot.gif\" width=10 height=10
border=0>\
               </A>\
               <A HREF = \"%s/tcont_delete?file=%s\" \
               target = \"leftnav\"> \
              <img src = \"/ows-img/arrow.gif\" width=10 height=10
border=0>\
               </A>\
               <A HREF=\"%s/tcont_download?file=%s\" \
               target = \"main\"> %s </A> <BR>\n",
```

```
            cs->szVirtualPath, escaped_str,
            cs->szVirtualPath, escaped_str,
            cs->szVirtualPath, escaped_str,
            docListElem->szParamName);
  }

exit_list:
  WRB_printf(WRBCtx, "<FORM METHOD = GET
ACTION=\"%s/tcont_upload_page\" \
target = \"main\">\n\
<INPUT TYPE=SUBMIT VALUE=Upload>\n\
</FORM>", cs->szVirtualPath);
  return TRUE;
}
```

Complete Source for the Document Server Cartridge

```
#include <stdio.h>
#include <stdarg.h>

#ifndef ORATYPES_ORACLE
# include <oratypes.h>
#endif

#ifndef WRB_ORACLE
# include <wrb.h>
#endif

#ifndef CONTENT_ORACLE
# include <content.h>
#endif

#ifndef ICX_ORACLE
# include <icx.h>
#endif

#define ACCEPTABLE(a)   (a>=32 && a<128 && ((isAcceptable[a-32]) &
0x1))


#define null(a) (a)?(a):(text *)"null"

#define PARAM_FILENAME     (text *)"file"
#define CONTENT_UPLOAD      1
#define CONTENT_DOWNLOAD    2
#define CONTENT_LIST        3
#define CONTENT_MAIN        4
```

```
#define CONTENT_UPLOAD_PAGE 5
#define CONTENT_ATTRIB      6
#define CONTENT_ICX         7
#define CONTENT_SETATTRIB   8
#define CONTENT_DELETE      9

WRBReturnCode testentry (WRBCallbacks *WRBCalls);
WRBReturnCode content_init(void *WRBCtx, void **clientcxp);
WRBReturnCode content_exec(void *WRBCtx, void *clientcxp);
WRBReturnCode content_shut(void *WRBCtx, void *clientcxp);


typedef struct
{
  dvoid *hRepository;
  dvoid *hDoc;
  text  *szProxy;
  text  *szNoProxyOn;
  text  *szVirtualPath;
} content_struct;


boolean content_list(dvoid *WRBCtx, content_struct *cs);
boolean content_download(dvoid *WRBCtx, content_struct *cs);
boolean content_upload(dvoid *WRBCtx, content_struct *cs, boolean
icx);
boolean content_delete(dvoid *WRBCtx, content_struct *cs);
boolean content_upload_page(dvoid *WRBCtx, content_struct *cs);
boolean content_frameset(dvoid *WRBCtx, content_struct *cs);

text *content_extension(text *str);
text   *content_escape(text *str);

void content_debug(char *fmt, ...)
{
  va_list vlist;

  va_start(vlist, fmt);
  vfprintf(stderr, fmt, vlist);
  va_end(vlist);
}


WRBReturnCode testentry (WRBCallbacks *WRBCalls)
{
  WRBCalls->init_WRBCallback      = content_init;
  WRBCalls->exec_WRBCallback      = content_exec;
  WRBCalls->shut_WRBCallback      = content_shut;
```

```
  return (WRB_DONE);
}


WRBReturnCode content_init(void *WRBCtx, void **clientcxp)
{
  content_struct *cs;
  text *cartridge = WRB_getCartridgeName(WRBCtx);

  cs = (content_struct *)malloc(sizeof(content_struct));
  if (!cs)
  {
    content_debug("Error in malloc exiting\n");
    return WRB_ABORT;
  }

  if (!(cs->hRepository =
     WRB_CNTopenRepository(WRBCtx, NULL, NULL, (text *)"WEBSYS")))
  {
    content_debug("Error in open repository. \
                  Refer to wrb.log for additional info\n");
    return WRB_ABORT;
  }

  if (!(cs->szProxy = WRB_getAppConfigVal(WRBCtx, cartridge,
                            (text *)"http_proxy")))
    cs->szProxy = (text *)"http://oracle-proxy.us.oracle.com";

  if (!(cs->szNoProxyOn = WRB_getAppConfigVal(WRBCtx, cartridge,
                                (text *)"no_proxy")))
    cs->szNoProxyOn = "us.oracle.com";

  *clientcxp = cs;
  return (WRB_DONE);
}

ub2 content_uri_type(text *uri)
{
  if (strstr((char *)uri, "tcont_upload_page"))
    return (CONTENT_UPLOAD_PAGE);
  else if (strstr((char *)uri, "tcont_upload_icx"))
    return (CONTENT_ICX);
  else if (strstr((char *)uri, "tcont_upload"))
    return (CONTENT_UPLOAD);
  else if (strstr((char *)uri, "tcont_download"))
    return (CONTENT_DOWNLOAD);
  else if (strstr((char *)uri, "tcont_list"))
```

```
    return (CONTENT_LIST);
  else if (strstr((char *)uri, "tcont_main"))
    return (CONTENT_MAIN);
  else if (strstr((char *)uri, "tcont_set_attrib"))
    return (CONTENT_SETATTRIB);
  else if (strstr((char *)uri, "tcont_attrib"))
    return (CONTENT_ATTRIB);
  else if (strstr((char *)uri, "tcont_delete"))
    return (CONTENT_DELETE);

  return 0;
}


WRBReturnCode content_exec(void  *WRBCtx, void *clientcxp)
{
  WRBpBlock  pBlock;
  text      *file = NULL;
  text      *uri = WRB_getRequestInfo(WRBCtx, WRBR_URI);
  ub2        uritype = content_uri_type(uri);
  FILE      *fd;
  char       buf[2048], filename[256];
  ub4        bytes_read;
  time_t     tloc = 0;

  content_struct *cs = (content_struct *)clientcxp;
  if (!(cs->szVirtualPath = WRB_getRequestInfo(WRBCtx,
WRBR_VIRTUALPATH)))
    cs->szVirtualPath = (text *)"/tcont";

  switch (uritype)
  {

  case CONTENT_UPLOAD:
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    WRB_printf(WRBCtx, "<HTML>\n");
    WRB_printf(WRBCtx, "<BODY bgcolor=\"black\" text=\"yellow\"\
link=\"yellow\" vlink=\"green\" >");
    if (content_upload(WRBCtx, cs, FALSE))
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, "<BR>Document successfully uploaded\n");
    }
    else
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, "<BR>Error in uploading\n");
    }
```

```
    WRB_printf(WRBCtx, "</HTML>\n");
    break;

  case CONTENT_ICX:
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    if (content_upload(WRBCtx, cs, TRUE))
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, "<BR>Document successfully uploaded\n");
    }
    else
    {
      content_list(WRBCtx, cs);
      WRB_printf(WRBCtx, "<BR>Error in uploading\n");
    }
    break;

  case CONTENT_DOWNLOAD:
    if (!content_download(WRBCtx, cs))
      WRB_printf(WRBCtx, "Error in downloadinf\n");
    break;

  case CONTENT_LIST:
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    content_list(WRBCtx, cs);
    break;

  case CONTENT_MAIN:
    WRB_printf(WRBCtx, "Content-type:text/html\n\n");
    content_main(WRBCtx, cs);
    break;

  case CONTENT_UPLOAD_PAGE:
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    content_upload_page(WRBCtx, cs);
    break;

  case CONTENT_ATTRIB:
    content_attrib(WRBCtx, cs);
    break;

  case CONTENT_SETATTRIB:
    content_set_attrib(WRBCtx, cs);
    content_attrib(WRBCtx, cs);
    break;

  case CONTENT_DELETE:
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
```

```
    WRB_printf(WRBCtx, “<HTML>\n”);
    WRB_printf(WRBCtx, “<BODY bgcolor=\”black\” text=\”yellow\”\
link=\”yellow\” vlink=\”green\” >”);
    content_delete(WRBCtx, cs);
    content_list(WRBCtx, cs);
    break;

  default:
    WRB_printf(WRBCtx, “Content-type: text/html\n\n”);
    content_frameset(WRBCtx, cs);
    break;
  }

  return (WRB_DONE);
}

WRBReturnCode content_shut(void *WRBCtx, void *clientcxp)
{
  content_struct *cs = (content_struct *)clientcxp;

  WRB_CNTcloseRepository(WRBCtx, cs->hRepository);
  return (WRB_DONE);
}

boolean content_delete(dvoid *WRBCtx, content_struct *cs)
{
  WRBpBlock     content = NULL;
  text          *file;

  if (!WRBCtx || !cs)
    return FALSE;

  if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    goto exit_delete;

  if (!(file = WRB_findPBElemVal(WRBCtx, content, PARAM_FILENAME,
                      -1)))
  {
    content_debug(“File name not specified for upload op\n”);
    WRB_printf(WRBCtx, “File name not specified\n”);
    goto exit_delete;
  }

  if (WRB_CNTdestroyDocument(WRBCtx, cs->hRepository, file)
      != WRB_SUCCESS)
  {
    content_debug(“Error deleting document %s\n”, file);
    WRB_printf(WRBCtx, “Error deleting document %s\n”, file);
```

```
    goto exit_delete;
  }

  WRB_destroyPBlock(WRBCtx, content);
  return TRUE;

exit_delete:
  if (content)
    WRB_destroyPBlock(WRBCtx, content);
  return FALSE;
}

/* This uses multipart/form-data encoding */
boolean
content_upload(dvoid *WRBCtx, content_struct *cs, boolean icx)
{
  text *file = NULL, *url = NULL;
  WRBpBlockElem multiPart;
  WRBpBlock     content;
  ub1 *mpData;
  ub4  mpDataLen;
  ub1 *text_buffer;
  ub4  buflen;
  dvoid *hReq;
  text  *errmsg;
  ub4    errmsgl;
  sword  errnum;

  if (icx)
  {
    if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    {
      WRB_printf(WRBCtx, “Error in parsing querystring\n”);
      return FALSE;
    }

    if (!(file = WRB_findPBElemVal(WRBCtx, content, (text *)”file”,
                    -1)))
    {
      WRB_printf(WRBCtx, “File name not specified\n”);
      return FALSE;
    }

    if (!(url = WRB_findPBElemVal(WRBCtx, content, (text *)”url”,
                 -1)))
    {
      WRB_printf(WRBCtx, “File name not specified\n”);
      return FALSE;
    }
```

```
  WRB_ICXsetProxy(WRBCtx, cs->szProxy);
  WRB_ICXsetNoProxy(WRBCtx, cs->szNoProxyOn);

  hReq = WRB_ICXcreateRequest(WRBCtx, url);

  if (WRB_ICXmakeRequest(WRBCtx, hReq, (dvoid **)&text_buffer,
           &buflen, 0, FALSE) != WRB_SUCCESS)
  {
   if (!WRB_getPreviousError(WRBCtx, &errnum, &errmsg, &errmsgl))
   errmsg = "Unknown error";

    WRB_printf(WRBCtx, "Error in ICX request<BR><PRE>%s</PRE>",
         errmsg);
    return FALSE;
  }
}
else
{
  while (WRB_getMultipartData(WRBCtx, &multiPart, &mpData,
                 &mpDataLen) == WRB_SUCCESS)
  {
    if (!strcmp(multiPart.szParamName, "file"))
    {
      mpData[mpDataLen] = '\0';
      file = (text *)mpData;
    }
    else if (!strcmp(multiPart.szParamName, "file_data"))
    {
      text_buffer = mpData;
      buflen = mpDataLen;
    }
  }
}


if (!file)
{
  WRB_printf(WRBCtx, "Filename not specified\n");
  return FALSE;
}

if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
           file, WRBCS_CREATE | WRBCS_WRITE)))
{
  content_debug("Error creating document %s\n", file);
  WRB_printf(WRBCtx, "Error creating document %s\n", file);
  return FALSE;
}
```

```
  if (WRB_CNTwriteDocument(WRBCtx, cs->hDoc, text_buffer,
                  buflen) <= 0)
  {
    content_debug("Error in writing to repostory %s\n",
            file);
    WRB_printf(WRBCtx, "Error in writing to repostory %s\n",
            file);
    return FALSE;
  }

  WRB_CNTcloseDocument(WRBCtx, cs->hDoc);
  cs->hDoc = NULL;

  return TRUE;
}


boolean content_download(dvoid *WRBCtx, content_struct *cs)
{
  text *file, *extn, *mimetype;
  WRBpBlock content;
  sb4  bytes_read;
  text buf[10240];
  WRBpBlock mimetypes;

  if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    return FALSE;

  if (!(file = WRB_findPBElemVal(WRBCtx, content, PARAM_FILENAME,
                  -1)))
  {
    content_debug("File name not specified for upload op\n");
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    WRB_printf(WRBCtx, "File name not specified\n");
    return FALSE;
  }

  extn = content_extension(file);
  mimetypes = WRB_getListenerInfo(WRBCtx, WRBL_MIMETYPES);
  if (!(mimetype = WRB_findPBElemVal(WRBCtx, mimetypes, extn, -1)))
    mimetype = (text *)"text/plain";

  if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
            file, WRBCS_OPEN | WRBCS_READ)))
  {
    WRB_printf(WRBCtx, "Content-type: text/html\n\n");
    content_debug("Error opening document %s\n", file);
    WRB_printf(WRBCtx, "Error opening document %s\n", file);
```

```
    return FALSE;
  }

  WRB_printf(WRBCtx, "Content-type: %s\n\n", mimetype);
  while ((bytes_read =
       WRB_CNTreadDocument(WRBCtx, cs->hDoc, (ub1 *)buf,
                     10240)) > 0)
    WRB_write(WRBCtx, buf, bytes_read);

  WRB_CNTcloseDocument(WRBCtx, cs->hDoc);
  WRB_destroyPBlock(WRBCtx, content);

  cs->hDoc = NULL;

  return TRUE;
}

boolean content_attrib(dvoid *WRBCtx, content_struct *cs)
{
  WRBpBlock      attributes, content;
  WRBpBlockElem *attr_elem;
  dvoid         *pos;
  text          *file;

  WRB_printf(WRBCtx, "Content-type: text/html\n\n");
  WRB_printf(WRBCtx, "<HTML>\n");
  WRB_printf(WRBCtx, "<BODY bgcolor=\"white\">\n");

  if (WRB_getParsedContent(WRBCtx, &content) != WRB_SUCCESS)
    goto exit_attrib;

  if (!(file = WRB_findPBElemVal(WRBCtx, content, PARAM_FILENAME,
                -1)))
  {
    content_debug("File name not specified for upload op\n");
    WRB_printf(WRBCtx, "File name not specified\n");
    goto exit_attrib;
  }


  if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
              file, WRBCS_OPEN | WRBCS_READ)))
  {
    content_debug("Error opening document %s\n", file);
    WRB_printf(WRBCtx, "Error opening document %s\n", file);
    goto exit_attrib;
  }

  attributes = WRB_createPBlock(WRBCtx);
```

```
  if (WRB_CNTgetAttributes(WRBCtx, cs->hDoc, attributes) < 0)
    goto exit_attrib;

  WRB_printf(WRBCtx, "<h3>Attributes for %s </h3>\n", file);
  WRB_printf(WRBCtx, "<PRE>\n");

  for (attr_elem = WRB_firstPBElem(WRBCtx, attributes, &pos);
      attr_elem;
      attr_elem = WRB_nextPBElem(WRBCtx, attributes, pos))
  {
    WRB_printf(WRBCtx, "%-15s\t%s\n",
            null(attr_elem->szParamName),
            null(attr_elem->szParamValue));
  }

  WRB_printf(WRBCtx, "</PRE>\n");
  WRB_printf(WRBCtx, "<BR><BR><P><h3> Set Document Attributes for
%s</h3>\n", file);

  WRB_printf(WRBCtx,
"<FORM METHOD=GET action=%s/tcont_set_attrib> \n\
<input type=hidden name=file value=%s> \n\
Name: <input type=text name=name1 size=20> <br>\n\
Value: <input type=text name=value1 size=30> <p> \n\
Name: <input type=text name=name2 size=20> <br>\n\
Value: <input type=text name=value2 size=30> <p> \n\
Name: <input type=text name=name3 size=20> <br>\n\
Value: <input type=text name=value3 size=30> <p>\n\
<INPUT TYPE=SUBMIT VALUE=\"Set Attrib\">       \n\
<INPUT TYPE=RESET VALUE=Reset>                 \n\
</FORM>\n",  cs->szVirtualPath, file);

  WRB_printf(WRBCtx, "</HTML>\n");

  WRB_CNTcloseDocument(WRBCtx, cs->hDoc);
  cs->hDoc = NULL;

  WRB_destroyPBlock(WRBCtx, content);
  WRB_destroyPBlock(WRBCtx, attributes);
  return TRUE;

exit_attrib:
  WRB_printf(WRBCtx, "Error in listing attributes\n");
  WRB_printf(WRBCtx, "</HTML>\n");
  return FALSE;
}

boolean content_set_attrib(dvoid *WRBCtx, content_struct *cs)
{
```

```
WRBpBlock      attributes = NULL;
WRBpBlock      request;
text          *name1 = NULL, *name2 = NULL, *name3 = NULL;
text          *value1= NULL, *value2 = NULL, *value3 = NULL;
text          *file = NULL;
boolean        rval = FALSE;
text           *errmsg;
ub4            errmsgl;
sword          errnum;

if (!(attributes = WRB_createPBlock(WRBCtx)))
  goto exit_set_attrib;

if (WRB_getParsedContent(WRBCtx, &request) != WRB_SUCCESS)
  goto exit_set_attrib;

if (!(file = WRB_findPBElemVal(WRBCtx,request, PARAM_FILENAME,
                -1)))
{
  content_debug("File name not specified for upload op\n");
  WRB_printf(WRBCtx, "File name not specified\n");
  goto exit_set_attrib;
}

if (!(cs->hDoc = WRB_CNTopenDocument(WRBCtx, cs->hRepository,
                  file, WRBCS_OPEN | WRBCS_READ)))
{
  content_debug("Error opening document %s\n", file);
  WRB_printf(WRBCtx, "Error opening document %s\n", file);
  goto exit_set_attrib;
}

name1 = WRB_findPBElemVal(WRBCtx, request, "name1", -1);
value1 = WRB_findPBElemVal(WRBCtx, request, "value1", -1);
name2 = WRB_findPBElemVal(WRBCtx, request, "name2", -1);
value2 = WRB_findPBElemVal(WRBCtx, request, "value2", -1);
name3 = WRB_findPBElemVal(WRBCtx, request, "name3", -1);
value3 = WRB_findPBElemVal(WRBCtx, request, "value3", -1);

if (name1 && value1)
  WRB_addPBElem(WRBCtx, attributes, name1, -1, value1, -1, 0);

if (name2 && value2)
  WRB_addPBElem(WRBCtx, attributes, name2, -1, value2, -1, 0);

if (name3 && value3)
  WRB_addPBElem(WRBCtx, attributes, name3, -1, value3, -1, 0);

if (WRB_CNTsetAttributes(WRBCtx, cs->hDoc, attributes) !=
```

```
WRB_SUCCESS)
  {
    if (!WRB_getPreviousError(WRBCtx, &errnum, &errmsg, &errmsgl))
      errmsg = “Unknown error”;

    WRB_printf(WRBCtx, “Error in setting attributes
<BR><PRE>%s</PRE>\n”, errmsg);
    goto exit_set_attrib;
  }

  rval = TRUE;

exit_set_attrib:
  if (cs->hDoc) WRB_CNTcloseDocument(WRBCtx, cs->hDoc);

  if (attributes) WRB_destroyPBlock(WRBCtx, attributes);

  if (request) WRB_destroyPBlock(WRBCtx, request);

  return rval;
}

boolean content_list(dvoid *WRBCtx, content_struct *cs)
{
  WRBpBlock      docList;
  WRBpBlockElem *docListElem;
  dvoid         *pos;
  text          *escaped_str;

  if (!WRBCtx || !cs)
    return FALSE;

  WRB_printf(WRBCtx, “<BODY bgcolor=\”black\” text=\”yellow\”\
 link=\”yellow\” vlink=\”green\” >”);

  WRB_printf(WRBCtx, “<H3> Repostory listing </H3>\n”);

  if (WRB_CNTlistDocuments(WRBCtx, cs->hRepository, &docList) <= 0)
  {
    WRB_printf(WRBCtx, “Repository contains no documents<BR>”);
    goto exit_list;
  }

  for (docListElem = WRB_firstPBElem(WRBCtx, docList, &pos);
       docListElem;
       docListElem =  WRB_nextPBElem(WRBCtx, docList, pos))
  {
    escaped_str = content_escape(docListElem->szParamName);
```

```
    WRB_printf(WRBCtx,
            "<A HREF = \"%s/tcont_attrib?file=%s\" \
               target = \"main\"> \
               <img src=\"/ows-img/dot.gif\" width=10 height=10
border=0>\
               </A>\
               <A HREF = \"%s/tcont_delete?file=%s\" \
               target = \"leftnav\"> \
              <img src = \"/ows-img/arrow.gif\" width=10 height=10
border=0>\
               </A>\
               <A HREF =\"%s/tcont_download?file=%s\" \
               target = \"main\"> %s </A> <BR>\n",
            cs->szVirtualPath, escaped_str,
            cs->szVirtualPath, escaped_str,
            cs->szVirtualPath, escaped_str,
            docListElem->szParamName);
  }

exit_list:
  WRB_printf(WRBCtx, "<FORM METHOD=GET
                     ACTION=\"%s/tcont_upload_page\" \
                     target=\"main\">\n\
                     <INPUT TYPE=SUBMIT VALUE=Upload>\n\
                     </FORM>", cs->szVirtualPath);
  return TRUE;
}


boolean content_frameset(dvoid *WRBCtx, content_struct *cs)
{
  WRB_printf(WRBCtx,
"<HTML>\n\
<HEAD>\n \
<TITLE> Oracle Web Application Server Document Server</TITLE>\n\
</HEAD>\n");

  WRB_printf(WRBCtx,
"<FRAMESET COLS=\"35%,65%\" BORDER=\"3\">\n\
        <FRAME\n\
                NAME=\"leftnav\" \n\
                SRC=\" %s/tcont_list\"\n\
                MARGINHEIGHT=5\n\
                MARGINWIDTH=10\n\
                SCROLLING = \"auto\">\n\
        <FRAME\n\
                NAME=\"main\"\n\
                SRC=\" %s/tcont_main\"\n\
                MARGINHEIGHT=5\n\
```

```
                MARGINWIDTH=10\n\
                SCROLLING = \"auto\">\n\
</FRAMESET>\n\
</HTML>", cs->szVirtualPath, cs->szVirtualPath);

  fprintf(stderr, "---------%s\n", cs->szVirtualPath);
  return TRUE;
}

boolean content_main(dvoid *WRBCtx)
{
  WRB_printf(WRBCtx,
"<HTML>\n\
<BODY bgcolor=\"white\">\n\
<center>\n\
<IMG SRC = \"/ows-img/ows.gif\" width=560 height=116 \
ALT=\"Oracle WebServer\">\n\
<H1> Oracle Web Application Server</H1>\n\
<h2> Document Server </h2>\n\
A Demonstration for the WRB Content Services\n\
</center>\n\
<p><p>\
<LI> <img src = \"/ows-img/dot.gif\" size=10 width=10 border=0>\n\
Lists document attributes<p>\n\
<LI> <img src = \"/ows-img/arrow.gif\" size=10 width=10 border=0>\n\
Deletes the document<p>\n\
</HTML>\n");
  return TRUE;
}

boolean content_upload_page(dvoid *WRBCtx, content_struct *cs)
{
  WRB_printf(WRBCtx,
"<HTML>\n                                    \
<HEAD>\n                                    \
<TITLE>Upload Page</TITLE>\n                \
</HEAD>\n                                   \
<BODY bgcolor =\"white\">\n                 \
<center><h2>Uploading document to Server.</h2></center>\n\
<form ENCTYPE=multipart/form-data method=POST            \
action=%s/tcont_upload target=\"leftnav\">\n              \
Enter Document name to be stored in the repository       \
<input type=text name=file size=25><p>\n                 \
(Document name should begin with a \"/\" )<p>            \
Enter file on local disk to upload:                      \
<input type=file name=file_data><p>\n                    \
<INPUT TYPE=SUBMIT VALUE=\"Upload File\">\n              \
<INPUT TYPE=RESET VALUE=Reset>\n                         \
```

```
</FORM>\n", cs->szVirtualPath );


  WRB_printf(WRBCtx, "<BR><BR>\n                          \
<h2> Upload documents thru ICX</h2>\n                      \
<form method=POST action=%s/tcont_upload_icx target=\"leftnav\">\n\
Enter Document name to be stored in the repository        \
<input type=text name=file size=25><p>\n                   \
(Document name should begin with a \"/\" )<p>             \
Enter the URL for the document to upload:                  \
<input type=text name=url size=30><p>\n                    \
<INPUT TYPE=SUBMIT VALUE=\"Upload File\">\n                \
<INPUT TYPE=RESET VALUE=Reset>\n                           \
</form>\n", cs->szVirtualPath);

  return TRUE;
}

static char *hex = "0123456789ABCDEF";

static ub1 isd[2] = {0,1};
static ub1 isAcceptable[96] = {
  1,1,0,0,1,0,0,1,1,1,1,0,1,1,1,0,    /*  !"#$%&'()*+,-./ */
  1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,    /* 0123456789:;<=>? */
  0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,    /* @ABCDEFGHIJKLMNO */
  1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,    /* PQRSTUVWXYZ[\]^_ */
  0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,    /* `abcdefghijklmno */
  1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0     /* pqrstuvwxyz{\}~ DEL */
};

text *content_escape(text *str)
{
  text *p, *q, *new;
  ub2  unacceptable;

  for (p = str; *p; p++)
  {
    if (!ACCEPTABLE(*p))
      unacceptable++;
  }
  new = (text *)malloc(p-str+unacceptable+unacceptable+1);

  for (q = new, p = str; *p; p++)
  {
    if (*p == ' ')
    {
      *q++ = '+';
    }
```

```
    else if(!ACCEPTABLE(*p))
    {
      *q++ = '%';
      *q++ = hex[*p >> 4];
      *q++ = hex[*p & 15];
    }
    else
    {
      *q++ = *p;
    }
  }
  *q++ = '\0';
  return new;
}


text *content_extension(text *str)
{
  ub2 strl = strlen(str);
  ub2 i;

  for (i = strl-1; i > 0; i--)
    if (str[i] == '.')
      return &(str[i+1]);

  return NULL;
}
```

CHAPTER

7

Web Request Broker Core API Reference

This section provides a reference for the core WRB API.

See Overview of the Web Request Broker (WRB) and Writing Applications Using the Web Request Broker API for details on architecture of the WRB and how to structure your applications.

- `WRB_addPBElem()`—Add an element to a parameter block
- `WRB_annotateURL()`—Write a URL with appended query string
- `WRB_apiVersion()`—Return the current version of the WRB.
- `WRB_copyPBlock()`—Copy a parameter block
- `WRB_createPBlock()`—Create a parameter block
- `WRB_delPBElem()`—Delete an element from a parameter block
- `WRB_destroyPBlock()`—Destroy a parameter block
- `WRB_findPBElem()`—Find an element by name in a parameter block
- `WRB_findPBElemVal()`—Find the value of a parameter block element
- `WRB_firstPBElem()`—Find the first element in a parameter block
- `WRB_getAppConfigSection()`—Get cartridge configuration data
- `WRB_getAppConfigVal()`—Get the value of a configuration parameter

- `WRB_getCartridgeName()`—Get the name of the calling cartridge
- `WRB_getClientCert()`—Get an SSL certificate from the client
- `WRB_getCookies()`—Get cookie data from the current request header
- `WRB_getEnvironment()`—Get the cartridge environment variables
- `WRB_getListenerInfo()`—Get information about a Web Listener
- `WRB_getMultAppConfigSection()`—Get data for multiple cartridges
- `WRB_getMultipartData()`—Get data from a multipart form.
- `WRB_getParsedContent()`—Get request content as name-value pairs
- `WRB_getPreviousError()`—Get the most recent error
- `WRB_getRequestInfo()`—Get information about the current request
- `WRB_nextPBElem()`—Get the next element in a parameter block
- `WRB_numPBElem()`—Get the number of elements in a parameter block
- `WRB_printf()`—Write a formatted text string
- `WRB_read()`—Read POST data from the requestor
- `WRB_recvHeaders()`—Get the request headers
- `WRB_sendHeader()`—Send the response header to the requestor
- `WRB_setCookies()`—Set cookie data in the response header
- `WRB_setAuthBasic()`—Create a new basic authentication realm
- `WRB_setAuthDigest()`—Create a new digest authentication realm
- `WRB_setAuthServer()`—Use the authentication server for authentication
- `WRB_timestamp()`—Write an entry in the log file when this function is executed
- `WRB_walkPBlock()`—Find an element by position in a parameter block
- `WRB_write()`—Write response data to the requestor

WRB Data Types

- The Parameter Block Element Structure
- WRB API Function Return Codes
- WRB Cartridge Function Return Codes
- The WRBCallbacks Structure

WRB_addPBElem()

Adds a single element to the specified parameter block. You must first call `WRB_createPBlock()` to create the parameter block.

Syntax

```
WAPIReturnCode WRB_addPBElem (dvoid *WRBCtx,
                    WRBpBlock hPBlock,
                    text      *nParamName,
                    sb4       szParamName,
                    text      *vParamValue,
                    sb4       nParamValue,
                    ub2       nParamType);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block
→ nParamName	The name of the element to be added
→ szParamName	The size in bytes of the name of the element to be added. If you set this parameter to –1, the function uses the strlen() library function to determine the length of the element name.
→ vParamValue	The value of the element to be added
→ nParamValue	The size in bytes of the value of the element to be added. If you set this parameter to –1, the function uses the strlen() library function to determine the length of the element value
→ nParamType	The parameter type. See The Parameter Block Element Structure for possible values.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_createPBlock()`

WRB_annotateURL()

Appends the specified query string data to the specified URL and writes the result to the requestor.

Syntax

```
sb4 WRB_annotateURL(dvoid *WRBCtx,
                    text *url,
                    WRBpBlock hArguments);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ url	The text of the base URL without the query string, including "http://", the hostname, port (if necessary), and the URI.
→ hArguments	A parameter block containing the name-value pairs to be appended to the URL as the query string.

Return Values

The number of bytes written to the requestor, or –1 when an error occurs.

WRB_apiVersion()

Returns the current version of the WRB API.

You should call this function in your cartridge's Init callback to determine if your cartridge is compatible with the user's WRB version:

```
if ( WRB_apiVersion(WRBCtx) < WRBAPI_CURRENT_VERSION )
      return WRB_ABORT;
```

`WRBAPI_CURRENT_VERSION` defines the version of the WRB that was used to build the cartridge. The code above aborts the cartridge if the current version is earlier than the version that is used by the cartridge.

Syntax

```
ub2 WRB_apiVersion(dvoid *WRBCtx);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Value

A version number.

WRB_copyPBlock()

Makes a copy of the specified parameter block.

Syntax

```
WRBpBlock
WRB_copyPBlock(dvoid *WRBCtx,
               WRBpBlock hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block you want to copy.

Return Values

A value of type WRBpBlock that identifies a copy of the specified parameter block. `WRB_copyPBlock()` returns NULL on failure.

See Also

`WRB_createPBlock()` and `WRB_destroyPBlock()`.

WRB_createPBlock()

Allocates a parameter block that you can fill in and pass it as a parameter to any WRB API function that takes a parameter block as a parameter.

Syntax

```
WRBpBlock
WRB_createPBlock (dvoid *WRBCtx);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A variable of type WRBpBlock that identifies the newly created parameter block. `WRB_createPBlock()` returns NULL on failure.

See Also

`WRB_destroyPBlock()`

WRB_delPBElem()

Deletes an element from a parameter block.

Syntax

```
WAPIReturnCode
WRB_delPBElem (dvoid *WRBCtx,
               WRBpBlock hPBlock,
               text *szName,
               sb4 nNamel);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block.
→ szName	The name of the element to be deleted from the parameter block.
→ nName1	The length in bytes of the element name. If you set this parameter to –1, the function uses the strlen() library function to determine the length of the element name.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_addPBElem()`

WRB_destroyPBlock()

Destroys a specified parameter block and frees the associated resources.

Syntax

```
WAPIReturnCode
WRB_destroyPBlock (dvoid *WRBCtx,
                   WRBpBlock hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block to be destroyed.

Return Values

`WRB_destroyPBlock()` returns a value of type `WAPIReturnCode`.

Usage

You must call `WRB_destroyPBlock()` on a parameter block when you are finished using it.

Examples

See the MyWRBApp function `placeOrder()`.

See Also

`WRB_createPBlock()`.

WRB_findPBElem()

Finds a named element in a parameter block and returns a pointer to it.

Syntax

```
WRBpBlockElem *
WRB_findPBElem(dvoid *WRBCtx,
               WRBpBlock hPBlock,
               text *szName,
               sb4 nNamel);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block to be searched.
→ szName	The element name for which to search.
→ nNamel	The length in bytes of the element name. If you set this parameter to –1, the function uses the strlen() library function to determine the length of the element name.

Return Values

A pointer to the parameter block element identified by szName. `WRB_findPBElem()` returns NULL on failure.

Usage

You can `useWRB_findPBElem()` to retrieve elements from parameter blocks by name.

Note: The returned value is valid only until the retrieved element is deleted.

Do not free the storage referred to by the returned pointer.

See Also

`WRB_createPBlock()`.

WRB_findPBElemVal()

Retrieves the value of a named element in a parameter block, such as the value of field on an HTML form retrieved by `WRB_getParsedContent()`.

Syntax

```
text *
WRB_findPBElemVal(dvoid *WRBCtx,
                  WRBpBlock hPBlock,
                  text *szName,
                  sb4 nNamel);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block to be searched.
→ szName	The element name for which to search.
→ nNamel	The length in bytes of the element name. If you set this parameter to –1, the function uses the strlen() library function to determine the length of the element name.

Return Values

A pointer to text string containing the value of the parameter block element identified by szName. `WRB_findPBElemVal()` returns NULL on failure.

Examples

See the MyWRBApp function `newUser()`

See Also

`WRB_createPBlock()`

WRB_firstPBElem()

Returns a pointer to the first element in a parameter block.

You can useWRB_firstPBElem() with WRB_nextPBElem() to iterate through a parameter block. This is more efficient than using WRB_walkPBlock().

Note: The returned value is valid only until the retrieved element is deleted.

Do not free the storage referred to by the returned pointer.

Syntax

```
WRBpBlockElem *
WRB_firstPBElem(dvoid *WRBCtx,
                WRBpBlock hPBlock,
                dvoid **pos);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block to be searched.
← pos	A pointer to the location in which the function places an opaque pointer representing the next available parameter block element.

Return Values

A pointer to the first element in a parameter block element. WRB_firstPBElem() returns NULL on failure.

Examples

```
WRBpBlock hPBlock;
WRBpBlockElem *elem;
dvoid *pos;

for (elem = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
     elem;
```

```
     elem = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
        /* do something with elem */
}
```

See Also

`WRB_nextPBElem()` and `WRB_createPBlock()`

WRB_getAppConfigSection()

Passes back a parameter block containing the name-value pairs defined in a specified section of the WRB configuration. You can use this function to retrieve configuration data for your own cartridge or other cartridges.

You must call `WRB_destroyPBlock()` on the parameter block when you are finished with it.

Syntax

```
WAPIReturnCode
WRB_getAppConfigSection(dvoid *WRBCtx
                        text *szSectionName
                        WRBpBlock *hSection);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szSectionName	The name of the cartridge configuration section you want to retrieve.
← hSection	A pointer to the location where the function places the parameter block containing the name-value pairs retrieved from the specified section.

Return Values

A value of type `WAPIReturnCode`.

Examples

You can use `WRB_getAppConfigSection()` in conjunction with `WRB_firstPBElem()` and `WRB_nextPBElem()` to retrieve all name-value pairs from a named section of the WRB configuration. This example retrieves and iterates through the name-value pairs defined for the calling cartridge:

```
text *cartname;
WRBpBlock hPBlock;
WAPIReturnCode ret;
```

```
WRBpBlockElem *elem;
dvoid *pos;

cartname = WRB_getCartridgeName(WRBCtx);
ret = WRB_getAppConfigSection(WRBCtx, cartname, &hPBlock);
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}

for (elem = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
     elem;
     elem = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
        /* do something with elem */
}

WRB_destroyPBlock(WRBCtx, hPBlock);
```

See Also

WRB_getAppConfigVal() and WRB_getMultAppConfigSection()

WRB_getAppConfigVal()

Returns the value of a named parameter defined in the the specified section of the WRB configuration data.

Syntax

```
text *
WRB_getAppConfigVal(dvoid *WRBCtx
                    text *szSectionName
                    text *szName);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szSectionName	The name of the WRB configuration section to search.
→ szName	The name of the configuration parameter for which you want the value.

Return Values

`WRB_getAppConfigVal()` returns a pointer to a text string containing the value of the specified parameter defined in the specified section of the WRB configuration data.

Examples

See the MyWRBApp function `MyWRBApp_Reload()`.

See Also

`WRB_getAppConfigSection()` and `WRB_getCartridgeName()`.

WRB_getCartridgeName()

Returns the name by which the calling cartridge is identified in the WRB configuration.

Syntax

```
text *
WRB_getCartridgeName(dvoid *WRBCtx);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to a text string containing the name by which the calling cartridge is identified in the WRB configuration data.

See Also

`WRB_getAppConfigSection()`, `WRB_getMultAppConfigSection()`, and `WRB_getAppConfigVal()`

WRB_getClientCert()

Gets the client's SSL certificate. When this function is executed, the user may see a dialog box asking him or her to select a certificate. This depends on how the user configured the browser.

Before you can use this function, you need to:

- Configure your cartridge to enable it to get the client's certificate.

  To do this, select Enabled in the Client Certificate field in the "New Cartridge Configuration" form in the Web Application Server Manager. This allows the Dispatcher to obtain the client's SSL certificate and pass it to the cartridge.

- Configure the Listener to require the client's certificate. This configuration is necessary only if you are using the Spyglass listener. It is not required if you are using the Netscape listener.

  You do this using the "Web Listener Administration" form in the Web Application Server Manager. In the "Addresses and Ports" section of the form, select "REQ" in the Authentication column for the SSL port.

Syntax

```
WAPIReturnCode WRB_getClientCert(
                    dvoid   *WRBCtx,
                    ub1     **cert,
                    ub4     *certlen);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← cert	A pointer to a pointer to a certificate. The certificate is DER-encoded. The WRB controls the memory for this certificate; you should not free or modify it.
← certlen	The length of the certificate. If the value of this parameter is 0, then there is no certificate associated with this request.

Return Value

A value of type `WAPIReturnCode`.

Example

```
WAPIReturnCode r;
FILE *fp;
ub1 *cert;
ub4 certlen;

if ((r = WRB_getClientCert(WRBCtx, &cert, &certlen))
                                == WRB_SUCCESS) {
    if (certlen > 0) {
      fp = fopen("cert.out", "w");
      fwrite(cert, sizeof(char), certlen, fp);
      fclose(fp);
    }
}
```

WRB_getCookies()

Passes back a parameter block containing the cookies associated with the current request.

When you are finished with the parameter block, you must call `WRB_destroyPBlock()` to free it.

Syntax

```
WAPIReturnCode
WRB_getCookies(dvoid *WRBCtx,
               WRBpBlock *hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← hPBlock	The location in which the function stores a parameter block containing the request cookies.

Return Values

A value of type `WAPIReturnCode`.

Examples

```
WRBpBlock hPBlock;
WAPIReturnCode ret;
WRBpBlockElem *elem;
dvoid *pos;

ret = WRB_getCookies(WRBCtx, &hPBlock);
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}

for (elem = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
     elem;
     elem = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
        /* do something with elem */
}
```

```
WRB_destroyPBlock(WRBCtx, hPBlock);
```

See Also

`WRB_setCookies()` and `WRB_destroyPBlock()`.

WRB_getEnvironment()

Retrieves the environment variable currently defined in the environment inherited by the calling cartridge and passes back this information in the form of a parameter block.

When you are finished with the parameter block containing the data, you must call `WRB_destroyPBlock()` to free the memory used by the parameter block.

Syntax

```
WAPIReturnCode
WRB_getEnvironment(dvoid *WRBCtx, WRBpBlock *hEnvironment);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← hEnvironment	A pointer to the location in which the function places the parameter block containing the cartridge environment variables.

Return Values

A value of type `WAPIReturnCode`.

Examples

You can use `WRB_getEnvironment()` in conjunction with `WRB_findPBElemVal()` to retrieve the value of a specific environment variable:

```
WRBpBlock hPBlock;
WAPIReturnCode ret;
text *owhome;

ret = WRB_getParsedContent(WRBCtx, &hPBlock);
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}
```

```
owhome = WRB_findPBElemVal(WRBCtx, hPBlock, (text *)"ORAWEB_HOME",
-1);

WRB_destroyPBlock(WRBCtx, hPBlock);
```

See Also

WRB_findPBElemVal() and WRB_destroyPBlock()

WRB_getListenerInfo()

Returns a parameter block containing information about the Web Listener that forwarded the current request, such as the listener's virtual directory mappings.

Syntax

```
WRBpBlock
WRB_getListenerInfo(dvoid *WRBCtx,
                    ub2 nInfoType);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ nInfoType	Specifies the type of information being requested. This must be one of the values listed in Listener Information Types.

Listener Information Types

Listener Information Type	Description
WRBL_DIRMAPS	Requests the listener's virtual-to-physical directory mappings.
WRBL_MIMETYPES	Requests a list of the MIME types the listener supports.

Table 7-1: Listener Information Types

Return Values

A parameter block containing the requested information about the Web Listener that forwarded the current request. `WRB_getListenerInfo()` returns NULL on failure.

WRB_getMultAppConfigSection()

Passes back a parameter block containing the name-value pairs defined in all WRB configuration data sections that have names starting with a specified string.

The parameter block that is passed back has an element for each section in the WRB configuration data. Each section's parameter block element in turn uses its `pNBdata` member to point to a parameter block containing the name-value pairs defined by that configuration section. See the example below.

You must call `WRB_destroyPBlock()` on the parameter block when you are finished with it.

Syntax

```
WAPIReturnCode
WRB_getMultAppConfigSection(dvoid *WRBCtx
                        text *szSectionPrefix
                        WRBpBlock *hSection);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szSectionPrefix	The prefix string identifying the cartridge configuration sections you want to retrieve. The function retrieves all sections with names that start with this string. If you pass "" (the empty string) for this parameter, the function retrieves the configuration data for all cartridges.
← hSection	A pointer to the location where the function places the parameter block containing the name-value pairs retrieved from the specified section.

Return Values

A value of type `WAPIReturnCode`.

Examples

You can use `WRB_getMultAppConfigSection()` with `WRB_firstPBElem()` and `WRB_nextPBElem()` to retrieve all name-value pairs from the specified sections of the WRB configuration. This example retrieves and iterates through the name-value pairs defined in all sections with names starting with “jane”:

```
WRBpBlock hPBlock;
WAPIReturnCode ret;
WRBpBlockElem *sect, *elem;
dvoid *sectpos, *pos;

ret = WRB_getMultAppConfigSection(WRBCtx, (text *)"jane",
&hPBlock);
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}

for (sect = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
     sect;
     sect = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
   /* sect points to the parameter block for a section */
   for (elem = WRB_firstPBElem(WRBCtx, sect->pNVdata, &pos);
        elem;
        elem = WRB_nextPBElem(WRBCtx, sect->pNVData, pos)) {
           /* do something with name-value pair */
   }
}

WRB_destroyPBlock(WRBCtx, hPBlock);
```

See Also

`WRB_getAppConfigVal()` and `WRB_getAppConfigSection()`

WRB_getMultipartData()

Retrieves data from a multipart form. The multipart form format is defined in RFC 1867.

For example, if you are uploading files, each part in your request would contain:

- a name-value pair where the name is the name of the header (for instance, “file”), the value is the name of the file you are uploading. This information is returned in the *multiPart* parameter.
- the data, which would be the contents of the file. This information is returned in the *mpData* parameter.

Each call to this function returns data for one part in the form. To get all parts, you call this function repeatedly until it does not return NULL.

Syntax

```
WAPIReturnCode
WRB_getMultipartData(
      dvoid          *WRBCtx,
      WRBpBlockElem  *multiPart,
      ub1            *mpData,
      ub4            mpDataLen);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← multiPart	A parameter block element that contains the name-value pair information.
← mpData	The data for the parameter element, in binary.
← mpDataLen	The length of the data in bytes.

Return Value

A value of type `WAPIReturnCode`.

Example

This example retrieves the value of the `file` and `file_data` headers from a multipart form.

```
while (WRB_getMultipartData(WRBCtx, &multiPart, &mpData,
            &mpDataLen) == WRB_SUCCESS)
{
      if (!strcmp(multiPart.szParamName, "file"))
      {
        mpData[mpDataLen] = '\0';
        file = (text *)mpData;
      }
      else if (!strcmp(multiPart.szParamName, "file_data"))
      {
        text_buffer = mpData;
        buflen = mpDataLen;
      }

}
```

WRB_getORACLE_HOME()

Returns the value of the ORACLE_HOME environment variable defined in the environment of the calling cartridge.

Syntax

```
text *
WRB_getORACLE_HOME(dvoid *WRBCtx);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

The value of the ORACLE_HOME environment variable, or NULL if ORACLE_HOME is not set.

See Also

`WRB_getEnvironment()`

WRB_getParsedContent()

Retrieves the current HTTP request's query string if the request method is GET, or its POST data if the request method is POST.

This function parses this data and passes it back in a parameter block. Each element in the parameter block contains the name and value of a POST data or query string entry.

When you are finished with the parameter block containing the data, you must call `WRB_destroyPBlock()` to free the memory used by the parameter block.

Syntax

```
WAPIReturnCode
WRB_getParsedContent(dvoid *WRBCtx, WRBpBlock *hQueryString);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← hQueryString	A pointer to the location in which the function places the parameter block containing the query string or POST data.

Return Values

A value of type `WAPIReturnCode`.

Examples

You can use `WRB_getParsedContent()` with `WRB_firstPBElem()` and `WRB_nextPBElem()` to retrieve the current request's query string or POST data:

```
WRBpBlock hPBlock;
WAPIReturnCode ret;
WRBpBlockElem *elem;
dvoid *pos;

ret = WRB_getParsedContent(WRBCtx, &hPBlock);
```

```
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}

for (elem = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
     elem;
     elem = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
        /* do something with elem */
}

WRB_destroyPBlock(WRBCtx, hPBlock);
```

See Also

`WRB_firstPBElem()`, `WRB_nextPBElem()` and `WRB_destroyPBlock()`.

WRB_getPreviousError()

Gets the most recent WRB error message. WRB error messages are logged in the **wrb.log** file.

Syntax

```
boolean WRB_getPreviousError(
                    dvoid   *WRBCtx,
                    sword   *nErrorNum,
                    text    **sErrorMesg,
                    ub4     *nErrorMesgl);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← nErrorNum	The Oracle error number from the message file.
← sErrorMesg	The message as it appears in the **wrb.log** file.
← nErrorMesgl	The length of the error message in bytes.

Return Value

TRUE if an error has been logged, FALSE otherwise.

WRB_getRequestInfo()

Returns a specific piece of information about an incoming request, such as the request URI or the fully qualified hostname of the Web Listener that forwarded the request.

Syntax

```
text *
WRB_getRequestInfo(dvoid *WRBCtx,
                   ub2 nInfoType);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ nInfoType	The type of information being requested. See Values for the nInfoType Parameter for a list of valid values.

Values for the nInfoType Parameter

Value	Description
WRBR_URI	Request URI.
WRBR_URL	Request URL.
WRBR_LISTENERTYPE	The type of listener, such as Oracle Web Application Server.
WRBR_VIRTUALPATH	Request Virtual path (a substring of the request URI).
WRBR_PHYSICALPATH	Physical path, if any, to which the virtual path is mapped.
WRBR_QUERYSTRING	Query String of the request.

Table 7-2: Request Information Types

Value	Description
WRBR_LANGUAGE	Comma-separated list of languages the requestor can accept.
WRBR_ENCODING	Comma-separated list of encodings the requestor can accept.
WRBR_REQMIMETYPE	MIME type of the request.
WRBR_USER	The username provided by the requestor in response to an authentication challenge.
WRBR_PASSWORD	The password provided by the requestor in response to an authentication challenge.
WRBR_IP	The requestor's IP address in dotted quad notation.

Table 7-2: Request Information Types

Return Values

Returns a pointer to the requested information in the form of a character string. or NULL on failure.

WRB_nextPBElem()

Returns a pointer to the next available element in a parameter block. You must call `WRB_firstPBElem()` to initialize the pos parameter before calling `WRB_nextPBElem()`.

Using `WRB_nextPBElem()` with `WRB_firstPBElem()` to iterate through a parameter block is more efficient than using `WRB_walkPBlock()`.

Note: The returned value is valid only until the retrieved element is deleted.

Do not free the storage referred to by the returned pointer.

Syntax

```
WRBpBlockElem *
WRB_nextPBElem(dvoid *WRBCtx,
               WRBpBlock hPBlock,
               dvoid **pos);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block to be searched.
→ pos	An opaque pointer representing the next available parameter block element. You must first initialized this value by calling WRB_firstPBElem().

Return Values

A pointer to the next available element in a parameter block element. `WRB_nextPBElem()` returns NULL on failure.

Examples

```
WRBpBlock hPBlock;
WRBpBlockElem *elem;
dvoid *pos;

for (elem = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
```

```
     elem;
     elem = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
        /* do something with elem */
}
```

See Also

WRB_firstPBElem()

WRB_numPBElem()

Returns the number of elements in a parameter block. You can use this function with `WRB_walkPBlock()` to iterate through the elements of a parameter block.

Syntax

```
sb4
WRB_numPBElem (dvoid *WRBCtx,
               WRBpBlock hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block.

Return Values

The number of elements in a specified parameter block.

Examples

See the `WRB_walkPBlock()` example.

See Also

`WRB_walkPBlock()`

WRB_printf()

Constructs a text buffer according to the specified format string and writes it to the requestor. The buffer that is written should be smaller than 10 K. If the buffer exceeds that size, the behavior is undefined.

You can use this function in the same way as the C standard I/O library `printf`(3) function to format and write a text string to the requestor.

Syntax

```
sb4
WRB_printf(dvoid *WRBCtx,
          text *formatStr,
          ...);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ formatStr	A pointer to a text string that specifies the format of the output buffer.

`WRB_printf()` accepts a variable number of arguments in the manner of the C standard I/O library function `printf`(3).

Return Values

The number of bytes written, or –1 if an error occurs.

See Also

`WRB_read()` and `WRB_write()`

WRB_read()

Reads the specified number of bytes from the request POST data.

You can call this function from your Exec function to get the POST data associated with the current request. This function is especially useful for buffering raw POST data.

If the POST data is in the form of name-value pairs, however, it is usually more convenient to call `WRB_getParsedContent()` instead.

Syntax

```
sb4
WRB_read(dvoid *WRBCtx,
         text *sBuffer,
         sb4 nBufferSize);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ sBuffer	A pointer to the buffer into which the function is to read the POST data. You must provide the storage for this buffer.
→ nBufferSize	The size of the buffer in bytes.

Return Values

The number of bytes read, or –1 if an error occurs.

See Also

`WRB_getParsedContent()`, `WRB_write()`

WRB_recvHeaders()

Passes back a parameter block containing the HTTP headers associated with the current request.

When you are finished with the parameter block, you must call `WRB_destroyPBlock()` to free it.

Syntax

```
WAPIReturnCode
WRB_recvHeaders(dvoid *WRBCtx,
                WRBpBlock *hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← hPBlock	The location in which the function stores a parameter block containing the request headers.

Return Values

A value of type `WAPIReturnCode`.

Examples

You can use `WRB_recvHeaders()` in conjunction with `WRB_firstPBElem()` and `WRB_nextPBElem()` to retrieve the request headers:

```
WRBpBlock hPBlock;
WAPIReturnCode ret;
WRBpBlockElem *elem;
dvoid *pos;

ret = WRB_recvHeaders(WRBCtx, &hPBlock);
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}

for (elem = WRB_firstPBElem(WRBCtx, hPBlock, &pos);
     elem;
```

```
     elem = WRB_nextPBElem(WRBCtx, hPBlock, pos)) {
        /* do something with elem */
}

WRB_destroyPBlock(WRBCtx, hPBlock);
```

See Also

`WRB_setCookies()` and `WRB_destroyPBlock()`.

WRB_sendHeader()

Sends to the requestor the HTTP response headers for your response to the current request. You must first allocate a parameter block and add to it an element for each header.

If you want to redirect a request to another URL, you should not use the WRB 2.1 function `WRBReturnHTTPRedirect()`. Instead, you should add a parameter block element with the name "`Location`" and a value containing the URL to which you want to redirect the request. You can then call `WRB_sendHeader()` on this parameter block to redirect the request.

You must not call `WRB_sendHeader()` after calling `WRBClientWrite()`.

Syntax

```
WAPIReturnCode
WRB_sendHeader(dvoid *WRBCtx,
               WRBpBlock hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block containing the header data to be set in the response to the current request.

Return Values

A value of type `WAPIReturnCode`. If you call `WRB_sendHeader()` after calling `WRBClientWrite()`, `WRB_sendHeader()` fails and returns `WRB_TOOLATE`.

See Also

`WRB_createPBlock()`, `WRB_addPBElem()`, and `WRBClientWrite()`

WRB_setAuthBasic()

Creates a new basic authentication realm for authenticating requestors issuing requests to the calling cartridge.

You can call this function in your Authorize function to create a basic authentication realm that your cartridge can use to authenticate requestors. If your cartridge uses this function to create a realm, your Authorize function is responsible for authentication for your cartridge.

The WRB caches the realm specified by `szRealm` whenever this function is called. If your Authorize function returns indicating that the client is not authorized, the WRB used the cached realm name in formulating an error message to be returned to the client.

Syntax

```
WAPIReturnCode
WRB_setAuthBasic(dvoid *WRBCtx,
                 text *szRealm);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szRealm	The name of the realm to be created.

Return Values

A value of type `WAPIReturnCode`.

Examples

See the MyWRBApp function `MyWRBApp_Authorize()`.

See Also

`WRB_setAuthDigest()` and `WRB_setAuthServer()`.

WRB_setAuthDigest()

Creates a new digest authentication realm for authenticating requestors issuing requests to the calling cartridge.

You can call this function in your Authorize function to create a digest authentication realm that your cartridge can use to authenticate requestors. If your cartridge uses this function to create a realm, your Authorize function is responsible for authentication for your cartridge.

The WRB caches the values specified by the `sz*` parameters whenever this function is called. If your Authorize function returns indicating that the client is not authorized, the WRB used the cached realm name in formulating an error message to be returned to the client.

Syntax

```
WAPIReturnCode
WRB_setAuthDigest(dvoid *WRBCtx,
                  text *szRealm,
                  text *szOpaque,
                  text *szNonce,
                  text *szStale);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szRealm	The realm name.
→ szOpaque	The semantics of this parameter are defined in RFC 2069.
→ szNonce	The semantics of this parameter are defined in RFC 2069.
→ szStale	The semantics of this parameter are defined in RFC 2069.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_setAuthBasic()` and `WRB_setAuthServer()`

WRB_setAuthServer()

Specifies authentication and / or restriction schemes for the authentication server to use in authenticating requestors issuing requests to the calling cartridge.

You can call this function in your Init or Authorize function to specify a combination of existing authentication and / or restriction schemes for the authentication server to use in authenticating requestors issuing requests to your cartridge. You must encode the schemes in the szProtectString parameter as follows:

scheme(*realm*) [*op scheme*(*realm*) [*op scheme*(*realm*) ...]]

where *op* is one of '&' (the "and" operator) or '|' (the "or" operator). The square brackets are not part of the syntax, but indicate that additional scheme(realm) pairs are optional. These expressions may not be grouped; the operators are evaluated strictly from left to right.

scheme can be one of the following:

- Basic—basic authentication (passwords are transmitted without encryption between requestor and server)
- Digest—basic authentication (passwords are transmitted in an encrypted form called a "digest")
- Basic_Oracle—basic database authentication (passwords used in connecting to databases are transmitted without encryption)
- IP—Restriction based on the requestor's Internet (IP) address
- Domain—Restriction based on the requestor's DNS domain

For authentication schemes, *realm* must be an authentication realm defined in the WRB configuration data.

For restriction schemes, *realm* must be the name of a group of IP addresses or DNS domain names defined in the WRB configuration.

Note that if you call this function in the cartridge's Init callback, the cartridge's Authorize callback is not called.

Syntax

```
WAPIReturnCode
WRB_setAuthServer(dvoid *WRBCtx,
                  text *szProtectStr);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szProtectString	A text string that specifies the authentication and/or restriction schemes that the authentication server should use in authenticating requestors.

Return Values

A value of type `WAPIReturnCode`.

Examples

In this example, the call to `WRB_setAuthServer()` requires that requestors issuing requests to the calling cartridge authenticate themselves using the Basic authentication realm "Admin Server", defined in the WRB configuration. It also requires that requestors IP addresses be in the "HQ Hosts" group of IP addresses, defined in the WRB configuration:

```
WAPIReturnCode ret;

ret = WRB_setAuthServer(WRBCtx,
       (text *)"Basic(Admin Server) & IP(HQ Hosts)");
if (ret != WRB_SUCCESS) {
   return (WRB_ERROR);
}
```

See Also

`WRB_setAuthBasic()` and `WRB_setAuthDigest()`

WRB_setCookies()

Adds a cookie to the response header for the current request.

Syntax

```
WAPIReturnCode WRB_setCookies(dvoid *WRBCtx,
               text *name,
               text *value,
               text *domain,
               text *path,
               text *expire,
               boolean secure);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ name	The cookie name.
→ value	The cookie value.
→ domain	The DNS domain to which you want the requestor to send the cookie on future request. This is often the domain where your cartridge is running.
→ path	The URI path for which you want the requestor to send the cookie. This is often a URI that refers to your cartridge.
→ expire	The cookie expiration date.
→ secure	If set to TRUE, this parameter specifies that the function should use SSL to send cookies.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_getCookies()`

WRB_timestamp()

Logs an entry in the **wrb.pif** file with the specified location ID and time. You can use this function to analyze the performance of your cartridge by calling this function several times within a request and looking at the time-stamps.

Before you can use this function, you need enable the time-stamping mechanism by adding this line in the [SYSTEM] section of the **wrb.app** file:

```
[SYSTEM]
enable_timing_stats = logger
```

The entry in the **wrb.pif** file has the following format:

```
requestID, locationID, timestamp, type
```

type specifies if the log entry is cumulative or normal. See the description of the *type* parameter below.

Typically, you would call this function at the beginning of an operation and at the end of the operation to measure how long the operation takes.

Syntax

```
void WRB_timestamp(
            dvoid   *WRBCtx,
            ub4     locID,
            ub2     type);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ locID	The identifier for the time-stamp. Note that location IDs 0 through 65536 are reserved by the WRB.

→ type	One of: `WRBTIMESTAMP_CUMULATIVE` - specify this type if you call this function from within a loop in your code. You will get an entry in the **wrb.pif** file each time the function gets executed. `WRBTIMESTAMP_NORMAL` - specify this type if this function is not called from within a loop The reason for specifying the type is that the Log Analyzer processes these types differently.

Return Value

Nothing.

WRB_walkPBlock()

Returns a pointer to the element at a specified position in a parameter block.

You can use this function with `WRB_numPBElem()` to iterate through the elements of a parameter block.

Note: The returned value is valid only until the retrieved element is deleted.

Do not free the storage referred to by the returned pointer.

Syntax

```
WRBpBlockElem *
WRB_walkPBlock (dvoid *WRBCtx,
                WRBpBlock hPBlock,
                sb4 nPosition);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hPBlock	The parameter block.
→ nPosition	The position of the element in the parameter block.

Return Values

A pointer to the element at a specified position in a parameter block. `WRB_walkPBlock()` returns NULL on failure.

Examples

```
WRBpBlockElem *elem;
sb4 numelems;

numelems = WRB_numPBElem(WRBCtx, hPBlock);

for (i = 0; i < numelems; i++) {
    elem = WRB_walkPBlock (WRBCtx, hPBlock, i);
    /* do something with the element */
}
```

See Also

`WRB_findPBElem()`

WRB_write()

Writes the specified number of bytes from the specified buffer to the requestor.

Syntax

```
sb4
WRB_write(dvoid *WRBCtx,
          text *sBuffer,
          sb4 nBufferSize);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ sBuffer	A pointer to the buffer containing the data to be written.
→ nBufferSize	The size of the buffer in bytes.

Return Values

The number of bytes written, or –1 when an error occurs.

See Also

`WRB_read()` and `WRB_printf()`

The Parameter Block Element Structure

The parameter block element structure encodes the name, value, and type of a parameter block element. It also contains an `pNVdata` member, used for various purposes by some WRB API functions.

The nParamName and nParamValue members represent the length in bytes of the szParamName and szParamValue members respectively. If you set nParamName or nParamValue to –1, any WRB API function to which you pass the parameter block uses the strlen() library function to determine the length of the element name or value.

```
typedef struct _WRBPBElem
{
  text  *szParamName;
  sb4    nParamName;

  text  *szParamValue;
  sb4    nParamValue;

  ub2    nParamType;

  dvoid *pNVdata;
} WRBpBlockElem;
```

The nParamType member can take these values:

Value	Meaning
WRBPT_DONTCARE	Leaves the element type unspecified.
WRBPT_NUMBER	The element is a number.
WRBPT_STRING	The element is a character string.
WRBPT_DATE	The element is a calendar date in SQL date type format.
WRBPT_RAW	The element value is raw binary data.

Table 7-3: Parameter block element types

WRB API Function Return Codes

Return Code	Meaning
WRB_NOTLOADED	The function is part of a WRB service that is not loaded.
WRB_LOWMEM	The function could not allocate memory.
WRB_FAIL	The function had an internal error.
WRB_SUCCESS	The function completed successfully.
WRB_TOOLATE	You tried to send header data after beginning to write content data.
WRB_AUTHNEEDED	You must supply the required authentication data and WRBMethod Enumerated Type again.
WRB_MOREDATA	WRBMethod Enumerated Type returned with a partial response: you must call WRBMethod Enumerated Type repeatedly until you have received the entire response.

Table 7-4: API function return codes

WRB Cartridge Function Return Codes

These are the return codes of type WRBReturnCode that cartridge functions must return. Some WRB API functions also return these codes:

Return Code	Description
WRB_DONE	Indicates normal completion.
WRB_ERROR	Indicates that the request could not be completed because of an error.
WRB_ABORT	Indicates a severe error has occurred: the calling WRBX should terminate immediately.

Note: Some WRB API functions return values of type `WAPIReturnCode` instead.

The WRBCallbacks Structure

The `WRBCallbacks` type defines the dispatch table that the WRB application engine uses to call your cartridge functions. See The Entry-Point Function for more information.

```
struct WRBCallbacks
{
   WRBReturnCode (*init_WRBCallback)();
   WRBReturnCode (*exec_WRBCallback)();
   WRBReturnCode (*shut_WRBCallback)();

   WRBReturnCode (*reload_WRBCallback)();

   char         *(*version_WRBCallback)();
   void          (*version_free_WRBCallback)();
   WRBReturnCode (*authorize_WRBCallback)();
};
typedef struct WRBCallbacks WRBCallbacks;
```

CHAPTER

8 WRB Content Service API Reference

Note: The APIs in this section are available only in the Advanced version of the Web Application Server. They are not available in the Standard version.

These APIs allow access to web content repositories from WRB cartridges using a simple set of interfaces reminiscent of standard UNIX file system access library functions.

For a description of WRB cartridges and a general overview of the WRB architecture, see the Overview of the Web Request Broker (WRB).

To learn about the general steps involved in developing WRB cartridges, see Writing Applications Using the Web Request Broker API.

WRB Content Service APIs

- `WRB_CNTopenRepository()` - Opens a content repository
- `WRB_CNTcloseRepository()` - Closes a content repository
- `WRB_CNTopenDocument()` - Opens a document in a content repository
- `WRB_CNTcloseDocument()` - Closes a document
- `WRB_CNTdestroyDocument()` - Deletes a document
- `WRB_CNTgetAttributes()` - Gets document attributes

- `WRB_CNTsetAttributes()` - Sets document attributes
- `WRB_CNTreadDocument()` - Reads from a document
- `WRB_CNTwriteDocument()` - Writes to a local copy of a document
- `WRB_CNTflushDocument()` - Saves changes to an open document
- `WRB_CNTlistDocuments()` - Returns the names of the documents in a repository

WRB_CNTopenRepository()

Establishes a connection to a content repository, such as a database, and returns a pointer to the repository that you can pass to subsequent Content Service functions.

You must open a repository before you can access the documents it contains.

Syntax

```
dvoid *
WRB_CNTopenRepository (dvoid *WRBCtx,
                       text *user,
                       text *passwd,
                       text *connectstr);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ user	The user name, such as a database user ID, to be used in connecting to the repository.
→ passwd	The password of the specified user.
→ connectstr	The connect string, such as a SQL*Net V2 connect string, to be used to connect to a remote repository. For local databases, installed in the same ORACLE_HOME directory as the WRB Dispatcher, you may pass NULL for this parameter.

Return Values

A pointer to the specified repository, or NULL on failure. You can call WRB_getPreviousError() to get more information about the error if there was a failure.

Examples

See the MyWRBApp function `MyWRBApp_Init()`.

See Also

`WRB_CNTcloseRepository()` and `WRB_CNTopenDocument()`.

WRB_CNTcloseRepository()

Closes the connection to the specified repository. You should call this function when you have finished accessing the repository.

Syntax

```
WAPIReturnCode
WRB_CNTcloseRepository (dvoid *WRBCtx,
                        dvoid * hRepository);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRepository	The pointer to the repository to close.

Return Values

A value of type `WAPIReturnCode`.

Examples

See the MyWRBApp function `MyWRBApp_Shutdown()`.

See Also

`WRB_CNTopenRepository()`.

WRB_CNTopenDocument()

Finds a specified document in a content repository and returns a pointer to it that you can use in subsequent calls to Content Service functions.

You must call this function on a document before you can call `WRB_CNTreadDocument()` or `WRB_CNTwriteDocument()` to access the document.

Syntax

```
dvoid *
WRB_CNTopenDocument (dvoid *WRBCtx,
                     dvoid * hRepository,
                     text *szDocName,
                     ub2 mode);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRepository	A pointer to the repository that contains the document.
→ szDocName	A pointer to the document name.
→ mode	Whether to open the document for reading and/or writing, and whether to create the document if it does not exist. You can use the following values: `WRBCS_CREATE` - Create the document if it does not exist. `WRBCS_READ` - Open the document for reading. `WRBCS_WRITE` - Open the document for writing.

Return Values

A pointer to the specified document, or NULL on failure.

Examples

See the MyWRBApp function `getCatalog()`.

See Also

`WRB_CNTcloseDocument()`, `WRB_CNTopenRepository()`, `WRB_CNTreadDocument()`, `WRB_CNTwriteDocument()`.

WRB_CNTcloseDocument()

Closes the specified document in a content repository.

You should call this function to close a document when you have finished accessing it. Any changes you have made to your private copy of the document by calling `WRB_CNTwriteDocument()` are applied to the original document before it is closed.

Syntax

```
WAPIReturnCode
WRB_CNTcloseDocument (dvoid *WRBCtx,
                      dvoid * hDocument);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hDocument	A pointer to the document to be closed.

Return Values

A value of type `WAPIReturnCode`.

Examples

See the MyWRBApp function `getCatalog()`.

See Also

`WRB_CNTopenDocument()`.

WRB_CNTdestroyDocument()

Deletes a specified document from a content repository.

You can use this function to delete a document permanently from a content repository. You must call `WRB_CNTopenRepository()` first to get a pointer to the repository that contains the document you want to destroy.

Syntax

```
WAPIReturnCode
WRB_CNTdestroyDocument (dvoid *WRBCtx,
                        dvoid * hRepository,
                        text *szDocName);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRepository	A pointer to the repository from which the specified document is to be deleted. This should be the pointer returned by WRB_CNTopenRepository().
→ szDocName	A pointer to the name of the document to be deleted.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_CNTopenRepository()`.

WRB_CNTgetAttributes()

Passes back a pointer to document attributes for the specified document in a content repository.

Before calling this function on a document, you must first call `WRB_CNTopenDocument()` to open the document.

Syntax

```
WAPIReturnCode
WRB_CNTgetAttributes (dvoid *WRBCtx,
                      dvoid * hDocument,
                      dvoid * hAttributes);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hDocument	A pointer to a document. This should be the pointer returned by WRB_CNTopenDocument().
← hAttributes	A pointer to a parameter block. You must call WRB_createPBlock() beforehand to allocate the parameter block.

Return Values

A value of type `WAPIReturnCode`.

Usage

You can use this function to get the following information about a document in a content repository. The attribute types referred to below are defined in Parameter block element types.

Attribute name	Attribute type	Description
author	WRBPT_STRING	The name of the document author.

Attribute name	Attribute type	Description
method	WRBPT_STRING	Whether the document is stored in a database or a file system.
content_type	WRBPT_STRING	The MIME type and subtype of the document.
methodinfo	WRBPT_STRING	The file system path of a document stored in a file system. This attribute is NULL for documents stored in databases.
creation_date	WRBPT_DATE	The date the document was created.
name	WRBPT_STRING	The document name.
description	WRBPT_STRING	A text string describing the document. This string must be no longer than 2000 bytes.
encoding	WRBPT_STRING	The encoding applied to the document, if any, such as compress or gzip.
owner	WRBPT_STRING	The user ID of the document owner.
expires	WRBPT_DATE	The date after which you may delete the document.
path	WRBPT_STRING	The full path of the document in the content repository, which is the concatenation of the folder and name attributes.
folder	WRBPT_STRING	The content repository folder that contains the document.
title	WRBPT_STRING	The HTML title of the document, if any.
language	WRBPT_STRING	The language in which the document is written.
type	WRBPT_TYPE	Whether the document is in text or binary format.

Attribute name	Attribute type	Description
last_modified	WRBPT_DATE	The date and time when the document was last modified.

Examples

```
WRBpBlockElem *attrib_elem;
sb4 num_attributes;
WAPIReturnCode ret;
int i;

ret = WRB_CNTgetAttributes(WRBCtx, hDocument, hAttributes);

if (ret == WRB_SUCCESS) {
   num_attributes = WRB_numPBElem(WRBCtx, hAttributes);
   for (i = 0; i < num_attributes; i++) {
      attrib_elem = WRB_walkPBlock(WRBCtx, hAttributes, i);
      /* do something with attribute */
   }
}
```

See Also

`WRB_CNTsetAttributes()` and `WRB_CNTopenDocument()`.

WRB_CNTsetAttributes()

Sets attributes for a specified document in a content repository. See `WRB_CNTgetAttributes()` for a description of the possible attributes.

Before calling this function on a document, you must first call `WRB_CNTopenDocument()` to open the document.

Syntax

```
WAPIReturnCode
WRB_CNTsetAttributes (dvoid *WRBCtx,
                      dvoid * hDocument,
                      dvoid * hAttributes);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hDocument	A pointer to a document. This should be the pointer returned by WRB_CNTopenDocument().
→ hAttributes	A pointer to a parameter block containing the document attributes to be set. You must call WRB_createPBlock() beforehand to allocate the parameter block.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_CNTgetAttributes()` and `WRB_CNTopenDocument()`.

WRB_CNTreadDocument()

Reads a specified number of bytes from the current position in a document in a content repository into a buffer provided by the caller.

Before calling this function on a document, you must first call `WRB_CNTopenDocument()` to open the document.

Syntax

```
sb4
WRB_CNTreadDocument (dvoid *WRBCtx,
                     dvoid * hDocument,
                     ub1 *buffer,
                     sb4 buffersz);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hDocument	A pointer to the document from which to read. This should be the pointer returned by WRB_CNTopenDocument().
← buffer	A pointer to the buffer in which to store the data read. You must provide the storage for this buffer.
→ buffersz	The size of the buffer.

Return Values

The number of bytes successfully read from the specified document.

Examples

See the MyWRBApp function `myfgets()`.

See Also

`WRB_CNTwriteDocument()` and `WRB_CNTopenDocument()`.

WRB_CNTwriteDocument()

Writes the contents of the specified buffer to the current position in a document.

When you call this function to modify a document, you modify your own private copy of the document. To apply these changes to the original document, you must call `WRB_CNTflushDocument()` or close the document by calling `WRB_CNTcloseDocument()`.

Before calling `WRB_CNTwriteDocument()` on a document, you must first call `WRB_CNTopenDocument()` to open the document.

Syntax

```
sb4
WRB_CNTwriteDocument (dvoid *WRBCtx,
                      dvoid * hDocument,
                      ub1 *buffer,
                      sb4 buffersz);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hDocument	A pointer to a document. This should be the pointer returned by WRB_CNTopenDocument().
→ buffer	A pointer to the buffer containing the data to be written.
→ buffersz	The number of bytes to be written.

Return Values

The number of bytes successfully written to the specified document.

Examples

See the MyWRBApp function `storeInfo()`.

See Also

`WRB_CNTreadDocument()` and `WRB_CNTopenDocument()`.

WRB_CNTflushDocument()

Updates the original document stored in a repository to reflect buffered changes to the document.

When you call `WRB_CNTwriteDocument()`, you modify your own private copy of a document in a content repository. To apply those changes to the original document, you must call `WRB_CNTflushDocument()`. If you are finished with the document, you can call `WRB_CNTcloseDocument()`, which flushes the document before closing it.

Syntax

```
WAPIReturnCode
WRB_CNTflushDocument (dvoid *WRBCtx,
                      dvoid * hDocument);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hDocument	A pointer to the document to be synchronized. This should be the pointer returned by WRB_CNTopenDocument().

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_CNTwriteDocument()` and `WRB_CNTopenDocument()`.

WRB_CNTlistDocuments()

Lists the names of the documents in the repository.

Syntax

```
sb4
WRB_CNTlistDocuments (
        dvoid      *WRBCtx,
        dvoid      *hRepository
        WRBpBlock  *docList);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRepository	A pointer to the repository. This should be the pointer returned by WRB_CNTopenRepository().
← docList	A parameter block containing the names of the documents. The document names are in the name field, and the value field is NULL.

Return Values

The number of documents.

CHAPTER

9

WRB Intercartridge Exchange Service API Reference

This section provides a reference for the WRB Intercartridge Exchange Service (ICX) APIs. The ICX Service can issue HTTP requests between WRB cartridges. For a description of WRB cartridges and a general overview of the WRB architecture, see Overview of the Web Request Broker (WRB).

To learn about the general steps involved in developing WRB cartridges, see Writing Applications Using the Web Request Broker API. For an example of how to use the WRB Intercartridge Exchange Service API, see the Intercartridge Exchange Service Sample Code Fragment.

This section describes:

- Getting the Most out of ICX
- WRB Intercartridge Exchange Service API Functions
- Data Types

Getting the Most out of ICX

While the Intercartridge Exchange (ICX) mechanism can be used for WRB load balancing on Web Application Server machines and increasing the scalability of

your Web Application Server environment, the primary benefit of ICX is that it allows you to distribute shared computation modules across many cartridges and many machines.

By designing cartridges that can provide specialize computation services to other cartridges, and distributing these cartridges across multiple machines, you can improve the efficiency of your Web Application Server environment, save storage space, and make maintenance of cartridge code easier by making it more modular.

WRB Intercartridge Exchange Service API Functions

- `WRB_ICXcreateRequest()`—Create a request object
- `WRB_ICXdestroyRequest()`—Destroy a request object
- `WRB_ICXfetchMoreData()`—Get more data when `WRB_ICXmakeRequest()` returns
- `WRB_ICXgetHeaderVal()`—Get the value of a response header
- `WRB_ICXgetInfo()`—Get information about a request
- `WRB_ICXgetParsedHeader()`—Get response headers
- `WRB_ICXmakeRequest()`—Issue a request
- `WRB_ICXsetAuthInfo()`—Set the authorization headers for a request
- `WRB_ICXsetContent()`—Set the content data for a request
- `WRB_ICXsetHeader()`—Set the headers for a request
- `WRB_ICXsetMethod()`—Set the HTTP method to use in a request
- `WRB_ICXsetNoProxy()`—Specify domains for which the proxy server should not be used
- `WRB_ICXsetProxy()`—Specify a proxy server

Data Types

- WRBInfoType Enumerated Type
- WRBMethod Enumerated Type

WRB_ICXcreateRequest()

Allocates and returns a handle to an opaque request object that encodes the request specified by a given URL.

To issue a request, call `WRB_ICXmakeRequest()` after calling `WRB_ICXcreateRequest()`. To abort the request, or when the request is complete, call `WRB_ICXdestroyRequest()` to free resources allocated for the request.

Syntax

```
dvoid *
WRB_ICXcreateRequest(void *WRBCtx,
                     text *url);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ url	A pointer to the request URL

Return Values

A handle to the newly created request object, or NULL on failure.

See Also

`WRB_ICXmakeRequest()` and `WRB_ICXdestroyRequest()`

WRB_ICXdestroyRequest()

Frees resources allocated for a specified request. Call this function:

- when the request is completed
- to cancel a request created by `WRB_ICXcreateRequest()` but not issued by `WRB_ICXmakeRequest()`.

Syntax

```
WAPIReturnCode
WRB_ICXdestroyRequest(void *WRBCtx,
                      dvoid * hRequest);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request to be destroyed. This should be the handle returned by WRB_ICXcreateRequest().

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXcreateRequest()` and `WRB_ICXmakeRequest()`.

WRB_ICXfetchMoreData()

Retrieves the requested number of bytes, or the number of bytes available, when a previous call to `WRB_ICXmakeRequest()` has returned `WRB_MOREDATA`.

Syntax

```
WRBAPIReturnCode
WRB_ICXfetchMoreData(dvoid *WRBCtx,
                     dvoid *hRequest,
                     dvoid **response,
                     ub4 *responseLength,
                     ub4 chunkSize);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request to be issued.
← response	A pointer to the location in which the function is to store a pointer to the response data.
← responseLength	A pointer to the location in which the function is to store the length in bytes of the reponse data.
→ chunkSize	If this parameter is non-zero, the size of the request response in bytes will be limited to this value. In this case, you must call WRB_ICXfetchMoreData() repeatedly until you've received the entire response. If this parameter is zero, no data is passed back.

Return Values

A value of type `WAPIReturnCode`.

Usage

If a call to `WRB_ICXmakeRequest()` returns `WRB_MOREDATA`, you can call `WRB_ICXfetchMoreData()` repeatedly to retrieve *chunkSize* more bytes of the request response until all response data has been received.

Examples

```
WAPIReturnCode ret;
dvoid *hRequest;
ub4 bufsize = 1024;
void buf[bufsize];
ub4 respLength;

/* create and initialize request */

ret = WRB_ICXmakeRequest(WRBCtx, hRequest, &buf, &respLength,
bufsize, 0);

/* do something with data */

while (ret == WRB_MOREDATA) {
   ret = WRB_ICXfetchMoreData(WRBCtx, hRequest, &buf, &respLength,
      bufsize, 0);
   / * do something with data */
}

/* destroy request */
```

See Also

WRB_ICXmakeRequest(), WRB_ICXcreateRequest(), WRB_ICXsetMethod(), WRB_ICXsetAuthInfo(), WRB_ICXsetHeader() and WRB_ICXsetContent()

WRB_ICXgetHeaderVal()

Returns a pointer to the value of a specified HTTP header from the response to a request issued by `WRB_ICXmakeRequest()`.

This is especially useful for retrieving response data that is stored only in the response headers.

Syntax

```
text *
WRB_ICXgetHeaderVal(void *WRBCtx,
                    dvoid * hRequest,
                    text *name);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request for which the header value is to be extracted from the response.
→ name	A pointer to the name of the header for which you want the value.

Return Values

The value of the specified header, or NULL on failure.

See Also

`WRB_ICXgetInfo()`

WRB_ICXgetInfo()

Returns a character string containing information about a specified request. The *infoType* parameter specifies the kind of information to be returned.

When an ICX request completes (when `WRB_ICXmakeRequest()` returns), you can call `WRB_ICXgetInfo()` to get information about the request. This is especially useful for getting the realm name in cases when the request must be re-issued with the appropriate authentication data.

Syntax

```
text *
WRB_ICXgetInfo(void *WRBCtx,
               dvoid * hRequest,
               WRBInfoType infoType);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request about which you want information.
→ infoType	A code that identifies the type of information you want. See WRBInfoType Enumerated Type for more information.

Return Values

A poiner to the requested information as a character string, or NULL on failure.

WRB_ICXgetParsedHeader()

Passes back a parameter block that contains the header values of the response to an ICX request issued by `WRB_ICXmakeRequest()`.

This is especially useful for retrieving response data that is stored only in the response headers.

Syntax

```
WAPIReturnCode
WRB_ICXgetParsedHeader(void *WRBCtx,
                       dvoid * hRequest,
                       WRBpBlock *hPblock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request.
← hPblock	A pointer to the location in which the function is to store the parameter block containing the parsed header data.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXgetHeaderVal()`

WRB_ICXmakeRequest()

Issues the specified request. After you have called `WRB_ICXcreateRequest()` to create a request and other Intercartridge Exchange API functions such as `WRB_ICXsetHeader()` and `WRB_ICXsetContent()` to prepare the request, you can call `WRB_ICXmakeRequest()` to issue the request.

Syntax

```
WAPIReturnCode
WRB_ICXmakeRequest(void *WRBCtx,
                   dvoid * hRequest,
                   void **response,
                   ub4 *responseLength,
                   ub4 chunkSize,
                   ub1 sendToBrowser);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request to be issued.
← response	A pointer to the location in which the function is to store a pointer to the response data. This is set to NULL if the request contains no data.
← responseLength	A pointer to the location in which the function is to store the length in bytes of the reponse data.
→ chunkSize	If this parameter is non-zero, the size of the request response in bytes will be limited to this value. In this case, you must call WRB_ICXfetchMoreData() repeatedly until you have received the entire response.
→ sendToBrowser	If this parameter is non-zero, the response from the request will be sent directly to the originating browser; in this case, the *response* parameter will contain NULL.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXcreateRequest()`, `WRB_ICXsetMethod()`, `WRB_ICXsetAuthInfo()`, `WRB_ICXsetHeader()` and `WRB_ICXsetContent()`

WRB_ICXsetAuthInfo()

Sets the authentication header data to accompany the specified request.

If your cartridge issues requests to another cartridge that requires your cartridge to authenticate itself, you can call this function to set the authentication header data for each request to the other cartridge.

Syntax

```
WAPIReturnCode
WRB_ICXsetAuthInfo(void *WRBCtx,
                   dvoid * hRequest,
                   text *username,
                   text *password);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request for which authentication is to be established. This should be a handle returned by WRB_ICXcreateRequest().
→ username	A pointer to a user name for request authentication.
→ password	A pointer to the password for the username.

Return Values

A value of type `WAPIReturnCode`.

WRB_ICXsetContent()

Sets request content for a specified request.

To set content data for a request, you must first call `WRB_createPBlock()` to allocate a parameter block containing the content data. Then, you can pass the parameter block to `WRB_ICXsetContent()`. You specify the request by passing the request handle returned by `WRB_ICXcreateRequest()`.

Syntax

```
WAPIReturnCode
WRB_ICXsetContent(void *WRBCtx,
                  dvoid * hRequest,
                  WRBpBlock hPBlock);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request for which content is to be specfied. This should be a handle returned by WRB_ICXcreateRequest().
→ hPBlock	The parameter block containing the content.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXcreateRequest()` and `WRB_ICXsetHeader()`

WRB_ICXsetHeader()

Sets HTTP header data for a specified request.

To set header data for a request, you must first call `WRB_createPBlock()` to allocate a parameter block and containing the header data. Then, you can pass the parameter block to `WRB_ICXsetHeader()`. You specify the request by passing the request handle returned by `WRB_ICXcreateRequest()`.

Syntax

```
WAPIReturnCode
WRB_ICXsetHeader(void *WRBCtx,
                 dvoid * hRequest,
                 WRBpBlock hPBlock,
                 boolean useOldHdr);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request for which headers are to be set. This should be the handle returned by WRB_ICXcreateRequest().
→ hPBlock	The parameter block containing the header information.
→ useOldHdr	When set to TRUE, the ICX request incorporates header data from the original request in addition to the data defined by the parameter block. Note that cookie and authorization headers are not included. When set to FALSE, only header data from the parameter block is used.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXcreateRequest()` and `WRB_ICXsetContent()`

WRB_ICXsetMethod()

Sets the request method, such as GET or POST, for a specified request.

After calling `WRB_ICXcreateRequest()` to create a request, you should call `WRB_ICXsetMethod()` to specify a request method for the request. If you do not call `WRB_ICXsetMethod()` on a request, the request's method is GET by default.

Syntax

```
WAPIReturnCode
WRB_ICXsetMethod(void *WRBCtx,
                 dvoid * hRequest,
                 WRBMethod method);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ hRequest	Identifies the request for which the method is to be set. This should be the handle returned by WRB_ICXcreateRequest().
→ method	Specifies the request method. See WRBMethod Enumerated Type for valid values.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXcreateRequest()`

WRB_ICXsetNoProxy()

Specifies a list of DNS domains for which the proxy server specified by `WRB_ICXsetProxy()` should not be used. Any subsequent request URLs directed to the specified domains will not be routed through the proxy server.

If your cartridge calls `WRB_ICXsetProxy()` to set up proxy server request translation, but you don't want requests to all DNS domains to use the proxy server, you can use `WRB_ICXsetNoProxy()` to specify a comma-separated list of domains to which requests should be sent directly, without proxy server intervention.

Instead of calling this function, you can set the `no_proxy` environment variable to a comma-separated list of domains to which requests should be sent directly.

Syntax

```
WAPIReturnCode
WRB_ICXsetNoProxy(void *WRBCtx,
                  text *noProxy);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ noProxy	A pointer to a comma-separated list of DNS domains to which requests should be sent directly, without proxy server intervention.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXsetProxy()`

WRB_ICXsetProxy()

Specifies a proxy server to use in making future ICX requests that need to be routed outside a firewall.

If your cartridge is running on a machine inside a firewall and needs to issue ICX requests to machines outside the firewall, you can use `WRB_ICXsetProxy()` to specify the address of a proxy server that can send requests outside the firewall.

Instead of calling this function, you can set the `http_proxy` environment variable to the name of the proxy server to use.

Syntax

```
WAPIReturnCode
WRB_ICXsetProxy(void *WRBCtx,
                text *proxyAddress);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ proxyAddress	The proxy address in character-string form.

Return Values

A value of type `WAPIReturnCode`.

See Also

`WRB_ICXsetNoProxy()`

WRBInfoType Enumerated Type

`WRB_ICXgetInfo()` takes a parameter of type WRBInfoType that specifies the kind of request information to return. Valid values for this type are:

Value	Meaning
STATUSCODE	Specifies the HTTP response code.
HTTPVERSION	Specifies the version of the HTTP protocol used in the response.
REASONPHRASE	Specifies the reason text string that corresponds to the HTTP response code.
REALM	Specifies the name of the authentication realm specified in the response.

WRBMethod Enumerated Type

`WRB_ICXsetMethod()` takes a parameter of type WRBMethod that specifies the request method to be used for a request. Valid values for this type are:

`OPTIONS`

`GET`

`HEAD`

`POST`

`PUT`

`DELETE`

`TRACE`

CHAPTER

10

Intercartridge Exchange Service Sample Code Fragment

```
{
  HANDLE   reqp;
  ub4      rval;
  ub4      chunkSize = 0;
  ub4      responseLength;
  char    *response;

    /* setup proxy directive */
    WRB_ICXsetProxy(WRBCtx,(text *)”http://www-proxy.us.oracle.com”);
    WRB_ICXsetNoProxy(WRBCtx, (text *)”oracle.com”);

    /* create a request */
    if (!(reqp = WRB_ICXcreateRequest(WRBCtx, (text
*)”http://www.oracle.com”)))
    {
      return WRB_ERROR;
    }

    /* execute the request and send the result directly to the browser*/
    rval = WRB_ICXmakeRequest(WRBCtx, reqp, (void **)&response,
      &responseLength, chunkSize,TRUE);
    if (rval == WRB_FAIL)
    {
      return WRB_ERROR;
    }
```

```
    else
    {
      return WRB_DONE;
    }
}
```

CHAPTER

11 WRB Transaction Service API Reference

Note: The APIs in this section are available only in the Advanced version of the Web Application Server. They are not available in the Standard version.

The WRB Transaction Service enables developers to perform database transactions from a WRB cartridge. The WRB Transaction Service is based on the TX interface defined by the X/Open Company. For more information, see *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*, published by the X/Open Company Ltd.

The Transaction Service provides an environment for performing database transactions that span several HTTP requests to a cartridge. To perform actual database access, you must use a database access API such as OCI or Pro*C in conjunction with the transaction service. However, you must not use the database access API mechanisms for committing or rolling back transactions. Instead, you must use the Transaction Service for these operations. Currently, the Transaction Service supports the OCI and Pro*C database access APIs.

The Transaction Service interface replaces the old WRB_TXN interface. See “Migrating to the New TX Interface” for information on how to migrate your cartridge to use the new interface.

Note: For Transaction Service to work for your cartridge, you need to enable it for your cartridge. You can do this from any of these forms in the Web Application Server Manager:

- From the New Cartridge Configuration form or the Update Cartridge Configuration form, select "TRANSACTIONS" from the Services list.
- From the Web Request Broker Administration form, go to the "Applications and Services" section, and enter "TRANSACTIONS" for the cartridges for which you need the Transaction service.

For an example of using the WRB Transaction Service API, see "The Hello World Sample Cartridge".

For a description of WRB cartridges and a general overview of the WRB architecture, see "Overview of the Web Request Broker (WRB)". To learn about the general steps involved in developing WRB cartridges, see "Writing Applications Using the Web Request Broker API".

WRB Transaction Service API Functions

- `tx_annotate_path()` - Adds transaction context to virtual path
- `tx_annotate_url()` - Adds transaction context to request URL
- `tx_begin()` - Begins a transaction
- `tx_close()` - Closes a connection to resource managers
- `tx_commit()` - Commits the current transaction
- `tx_info()` - Gets information about a transaction
- `tx_open()` - Opens a connection to resource managers
- `tx_reg()` - Registers a resource manager to be opened
- `tx_rollback()` - Rolls back the current transaction
- `tx_set_transaction_timeout()` - Sets time-out for future transactions

Data Type

- TXINFO Structure - A struct that contains information about the current transaction

tx_annotate_path()

Appends the current transaction context to the specified cartridge virtual path and writes the result to the requestor.

You can use this function to send the current transaction ID to the requestor in the form of a cookie. The client can then send this transaction ID with subsequent requests to identify the transaction to which they apply.

Syntax

```
int tx_annotate_path (text *path);
```

Parameters

→ path	The virtual path to which to append the transaction context. This should specify the virtual path or path prefix of an anticipated future request that should be treated as part of the current transaction.

Return Values

- TX_OK
- TX_ERROR

Usage

For example, if you are writing a cartridge called “mycart”, and your Exec function supports “begin”, “update”, “commit”, and “rollback” requests, your Exec function can call tx_annotate_path() in the following ways:

While handling	Call

begin	`tx_annotate_path((text *)"/mycart/update");`
	`tx_annotate_path((text *)"/mycart/commit");`
	`tx_annotate_path((text *)"/mycart/rollback");`

update	`tx_annotate_path((text *)"/mycart/update");`
	`tx_annotate_path((text *)"/mycart/commit");`
	`tx_annotate_path((text *)"/mycart/rollback");`

While handling	Call
commit	(nothing)
rollback	(nothing)

If these are the only requests you define, you can just make a single call while handling “begin” and “update”:

```
tx_annotate_path((text *)"/mycart/");
```

This call specifies that the client should send the transaction ID for all requests using a virtual pathname starting with `"/mycart/"`.

If you want your cartridge to support clients that cannot or will not accept cookies, you can use `tx_annotate_url()` instead of `tx_annotate_url()`.

See Also

`tx_annotate_url()`.

tx_annotate_url()

Appends the current transaction context and the specified query string data to the specified URL and writes the result to the requestor.

Syntax

```
int tx_annotate_url (text *url, text **annotatedUrl);
```

Parameters

→ url	The URL to which to append the transaction context.
← annotatedUrl	The location in which the function passes back a pointer to annotated URL. You must free this buffer when you are finished with it.

Return Values

- TX_OK
- TX_ERROR

Usage

You can use `tx_annotate_url()` instead of `tx_annotate_path()` in cartridges that must support clients that cannot or will not accept cookies. To do this, you must generate a response to the request that contains explicit links to URLs for anticipated future requests that should be handled as part of the current transaction.

You must therefore call `tx_annotate_url()` from the section of code that generates the HTML response to the request.

Examples

```
text *annotatedUrl;
/* output headers... */

tx_annotate_url("/mycart/update", &annotatedUrl);
WRB_printf(WRBCtx,
    (text *)"<FORM ACTION=\"%s\" METHOD=\"POST\">\n",
    annotatedUrl);
```

```
/* output body of form... */
WRB_printf(WRBCtx,
    (text *)"<INPUT TYPE=\"submit\" NAME=\"update\" ");

/* output the rest of the HTML response */
```

See Also

`tx_annotate_path()`.

tx_begin()

Marks the beginning of a global transaction. All subsequent operations performed on open resource managers will be considered part of the transaction until you call `tx_commit()` to commit the transaction or `tx_rollback()` to roll back the transaction.

You must call `tx_open()` to connect to one or more resource managers before calling `tx_begin()`.

See The Transaction Model in *Writing Applications Using the Web Request Broker API* for more information about transaction semantics.

Syntax

```
int tx_begin (void);
```

Parameters

none

Return Values

- TX_OK
- TX_OUTSIDE - The calling cartridge is not transaction enabled.
- TX_PROTOCOL_ERROR - The calling cartridge is already in transactional mode.
- TX_FAIL - Fatal Error. The calling cartridge must return WRB_ABORT.

See Also

`tx_open()`, `tx_commit()`, `tx_rollback()`.

tx_close()

Closes the resource managers opened by a previous call to `tx_open()`. You can call this function to disconnect from resource managers when you are finished performing transactions with them. Before calling `tx_close()`, you must terminate any active transaction by calling `tx_commit()` or `tx_rollback()`.

Syntax

```
int tx_close (void);
```

Parameters

none

Return Values

- TX_OK
- TX_OUTSIDE - The calling cartridge is not transaction enabled.
- TX_ERROR - Transient error in closing one or more resource managers.
- TX_FAIL - Fatal error. The calling cartridge must return WRB_ABORT.

See Also

`tx_open()`.

tx_commit()

Commits the current transaction and terminates the current transaction.

Syntax

```
int tx_commit (void);
```

Parameters

none

Return Values

- TX_OK
- TX_OUTSIDE - The calling cartridge is not transaction enabled.
- TX_ROLLBACK - The commit failed, and a rollback is in progress.
- TX_MIXED - The commit was partially successful, and partial rollback is in progress.
- TX_FAIL - Fatal error. The calling cartridge must return WRB_ABORT.

See Also

`tx_begin()`, `tx_rollback()`.

tx_info()

Fills a supplied TXINFO structure with the following information about the current transaction:

- The transaction ID - A number that uniquely identifies the transaction.
- The transaction timeout - The time in seconds that the transaction may remain idle before it is automatically marked for rollback.
- The transaction state, which can be one of:
  - Active
  - Inactive
  - Marked for rollback because of a timeout

You must allocate the TXINFO structure yourself and pass a pointer to it to the function. See TXINFO Structure for more information.

Syntax

```
int tx_info (TXINFO *info);
```

Parameters

← info	A pointer to a TXINFO structure that you have allocated to contain the requested information. This function does not allocate the TXINFO structure for you.

Return Values

- 0 - The cartridge is out of a transaction mode.
- 1 - The cartridge is in transaction mode.
- TX_PROTOCOL_ERROR - The function was called out of context.
- TX_FAIL - Fatal error. The calling cartridge must return WRB_ABORT.

tx_open()

Opens and connects to all the resource managers registered by calls to `tx_reg()` since the beginning of the calling thread's execution, or the last call to `tx_close()`, whichever is more recent.

You must call `tx_reg()` to register at least one resource manager before calling `tx_open()`. You must also call `tx_open()` before calling `tx_begin()` to begin a transaction.

Syntax

```
int tx_open (void);
```

Parameters

none

Return Values

- TX_OK
- TX_OUTSIDE - The cartridge is not transaction enabled.
- TX_FAIL - Fatal Error. The cartridge must return WRB_ABORT.

See Also

`tx_close()`, `tx_begin()`.

tx_reg()

Specifies a database access descriptor (DAD) to register a resource manager to be opened by a subsequent call to `tx_open()`.

You must call `tx_reg()` to register a resource manager before calling `tx_open()` to open it. To specify the resource manager, you must pass the name of a database access descriptor (DAD) defined by your DAD Administration pages.

You may call `tx_reg()` repeatedly to register several resource managers. When you subsequently call `tx_open()`, all the registered resource managers will be opened.

Syntax

```
int tx_reg() (text *dad);
```

Parameters

→ dad	The name of a database access descriptor (DAD) defined by your DAD Administration pages.

Return Values

- TX_OK
- TX_OUTSIDE - The calling cartridge is not transaction enabled.
- TX_ERROR - The specified DAD is not configured by your DAD Administration pages.
- TX_FAIL - Fatal Error. The calling cartridge must return WRB_ABORT.

See Also

`tx_open()`.

tx_rollback()

Rolls back and terminates the current transaction. You should call `tx_rollback()` whenever you encounter a SQL error during a transaction.

Syntax

```
int tx_rollback (void);
```

Parameters

none

Return Values

- TX_OK
- TX_OUTSIDE - Cartridge not transactions enabled.
- TX_FAIL - Fatal error. The calling cartridge must return WRB_ABORT.

See Also

`tx_begin()`, `tx_commit()`.

tx_set_transaction_timeout()

Specifies the time in seconds that the current transaction may remain idle before it is automatically marked for rollback. You must call `tx_set_transaction_timeout()` before calling `tx_open()`.

Syntax

```
int tx_set_transaction_timeout (TRANSACTION_TIMEOUT timeout);
```

Parameters

→ timeout	The time in seconds that a transaction may remain idle before it is automatically marked for rollback.

Return Values

- TX_OK
- TX_OUTSIDE - The calling cartridge is not transaction enabled.
- TX_ERROR - The function encountered an error in setting the timeout.
- TX_FAIL - Fatal error. The calling cartridge must return WRB_ABORT.

See Also

`tx_open()`.

TXINFO Structure

This structure (defined in **tx.h**) encodes information about a transaction. The `tx_info()` call fills in this structure.

```
struct tx_info_t
{
   XID                  xid;
   COMMIT_RETURN        when_return;
   TRANSACTION_CONTROL  transaction_control;
   TRANSACTION_TIMEOUT  transaction_timeout;
   TRANSACTION_STATE    transaction_state;
};
typedef struct tx_info_t TXINFO;
```

The fields in the struct have the following meaning:

Field	Description
xid	A number that uniquely identifies the transaction.
when_return	Not used.
transaction_control	Not used.
transaction_timeout	The period in seconds that the transaction may remain idle before it is automatically marked for rollback.
transaction_state	The current state of the transaction. It is one of the following: • `TX_ACTIVE` - Indicates the transaction is active and may be committed. • `TX_INACTIVE` - Indicates the transaction has been suspended and may not currently be committed or rolled back. • `TX_TIMEOUT_ROLLBACK_ONLY` - Indicates the transaction has timed out and has been marked for rollback; the transaction may not be committed.

Note: The WRB Transaction Service implementation of the TX interface does not use the `when_return` and `transaction_control` fields.

CHAPTER

12 WRB Logger Service API Reference

The Web Request Broker (WRB) API Logger Service allows you to log information to log files on your local file system or in a database. This information could be related to a specific instance of a cartridge. You can use the Log Analyzer to extract informative statistics from this data.

For a description of WRB cartridges and a general overview of the WRB architecture, see the Overview of the Web Request Broker (WRB).

Logging Guidelines

The default log file location for file system logging is the **wrb.log** file. Although cartridges can set their own log files, having a single log file will simplify debugging and log analysis. Entries in the log file have the following format:

```
date     time     microsec  machine   component   PID    Severity  Mask     Message
-------------------------------------------------------------------------------------
01-14-97 16:33:21 123       pluto     LM          12050  7         0x1000   Log Module has been
                                                                            initialized
```

Severity Levels

Severity levels range from 0 to 15. The lower the number, the more serious the error. The logger logs all messages under a specified severity level; for example, if the severity level is set at 5, messages at severity levels 0 through 4 are logged.

The minimal logging for all cartridges is for all errors (severity <= 3). However, it is recommended that all warnings are logged as well (severity <= 6).

All cartridge developers should follow the severity level guidelines listed in the following table to ensure consistent logging.

Meaning	Severity	Recommended usage
Fatal errors (for example, memory errors)	0	0 indicates core failure occurred.
Soft errors (for example, non-fatal input/output errors)	1	1 indicates that writing to file or resource failed.
	2	(user-defined)
	3	(user-defined)
Warnings (for example, missing file or missing configuration section)	4	4 indicates a configuration error.
	5	(user-defined)
	6	(user-defined)
Tracings (for example, request has been executed)	7	7 indicates the process has entered the init, terminate, or reload stages.
	8	8 indicates the process has entered the authentication and execution stages.
	9	(user-defined)
	10	(user-defined)
Debugging (for example, variable logging)	11	11 is used for printing debugging variables.
	12	(user-defined)
	13	(user-defined)
	14	(user-defined)
	15	(user-defined)

Message Formats

To help administrators read the log file and diagnose errors, it is recommended that you adopt the message formats listed in this section. There are two different formats: one for error and warning messages, and one for tracing and debugging messages.

Format for Error and Warning Messages

Error and warning messages (severity <= 6) should use the following format:

```
productName-msgNumber: message
```

msgNumber is the ORACLE NUMBER in the message file. If a message is generated automatically (for example, the Java Cartridge), then a message number should also be generated.

Example:

```
OWS-05101: Agent execution failed due to Oracle error 6564
```

Format for Tracing and Debugging Messages

Tracing and debugging messages (severity > 6) should use the following format (each # indicates a space character):

```
###########message
```

The product name and message number fields are not included. Instead, the message is preceded by 11 spaces.

Example:

```
###########VPM has been initialized
```

WRB Logger APIs

- `WRB_LOGopen()`—Open a file or database connection
- `WRB_LOGwriteMessage()`—Write a system message to storage
- `WRB_LOGwriteAttribute()`—Write a client-defined attribute to storage
- `WRB_LOGclose()`—Close a file or database connection

`WRBLogMessage()` is a function supported in version 2.0 of the Oracle Web Application Server. In version 3.0, you should use `WRB_LOGwriteMessage()` instead.

WRB_LOGopen()

Opens a file or database connection.

Syntax

```
WAPIReturnCode WRB_LOGOpen( dvoid *WRBCtx,
                            ub4 *logHdl,
                            WRBLogType type,
                            WRBLogDestType dest,
                            text *filename );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← logHdl	This is an output parameter that indicates whether the connection is to a file or a database. You use the passed-in value in other function calls in this section.
→ type	A value identifying the log entry as a message or a client attribute. See WRBLogType for possible values.
→ dest	A value identifying whether to log in a file or database. See WRBLogDestType for a list of possible values.
→ filename	The name of the file, if logging in a file, or a NULL value if logging to database.

Return Values

A value of type `WAPIReturnCode`.

See Also

WRB_LOGclose(), WRB_LOGwriteMessage(), WRB_LOGwriteAttribute()

WRB_LOGwriteMessage()

Writes a system message to the storage specified by logHdl.

Syntax

```
WAPIReturnCode WRB_LOGWriteMessage( dvoid *WRBCtx,
                                    ub4 logHdl,
                                    text *component,
                                    text *msg,
                                       sb4 severity);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ logHdl	This is the logHdl value from the WRB_LOGopen() function call.
→ component	Text description to identify the type of cartridge, such as “java”.
→ msg	Text you wish to log. The maximum length of the message is 2000 bytes. Note that the back single quote character (`) is used as a delimiter. If you need to include that character in your message, you have to escape it with a backslash (that is, \`).
→ severity	The severity of the message. See Log Severity Values for possible values.

Return Values

A value of type `WAPIReturnCode`.

See Also

WRB_LOGopen(), WRB_LOGclose(), WRB_LOGwriteAttribute()

WRB_LOGwriteAttribute()

Writes a client-defined attribute to the storage specified by logHdl. For example, you could use this to monitor the behavior of a cartridge, capturing queue length at given intervals for later analysis.

Syntax

```
WAPIReturnCode WRB_LOGWriteAttribute( dvoid *WRBCtx,
                                      ub4 logHdl,
                                      text *component,
                                      text *name,
                                      text *value);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ logHdl	This is the logHdl value from the WRB_LOGopen() function call.
→ component	Text description to identify the type of cartridge, such as “java”.
→ name	Text identifying a particular attribute you wish to log.
→ value	Additional text to qualify the attribute you have named.

Return Values

A value of type `WAPIReturnCode`.

See Also

WRB_LOGopen(), WRB_LOGclose(), WRB_LOGwriteMessage()

WRB_LOGclose()

Closes the file or database connection specified by logHdl.

Syntax

```
WAPIReturnCode WRB_LOGClose( dvoid *WRBCtx, ub4 logHdl);
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ logHdl	This is the logHdl value from the WRB_LOGopen() function call.

Return Values

A value of type `WAPIReturnCode`.

See Also

WRB_LOGopen(), WRB_LOGwriteMessage(), WRB_LOGwriteAttribute()

WRB Logger API Data Types

These are the data types and possible values for Logger APIs.

WRBLogDestType

Indicates the destination for the logged messages. Possible values are the following:

- `WRBLogDestDefault` - Specifies that the destination is specified in the **wrb.app** file. In the file, the logging destination is specified by the `logger_logsys_desttype` and `logger_logattrib_desttype` attributes. Valid values for these attributes are "`FS`" for filesystem and "`DB`" for database.

  These attributes can be specified in two places in the **wrb.app** file: in the cartridge-specific section or in the generic section. The name of the cartridge-specific section is `[SYSTEM_`*`CARTRIDGENAME`*`]` and the generic section is `[LOGGER]`. The attribute in the cartridge-specific section takes precedence.

- `WRBLogDestDb` - Specifies that the logging destination is a database.
- `WRBLogDestFs` - Specifies that the logging destination is the filesystem.

WRBLogType

Indicates whether it is a message or a client attribute.

- `WRBLogSysMsg` - Specifies that it is a message.
- `WRBLogClientDefAttrib` - Specifies that it is a client attribute.

Log Severity Values

You can use one of these enumerated values to specify the severity level of the message.

Severity Level	Enumerated name
0	`WRBLogSevFatal`
1	`WRBLogSevErr`
4	`WRBLogSevWarn`

Severity Level	Enumerated name
7	`WRBLogSevTraItr`
8	`WRBLogSevAe`
11	`WRBLogSevDebug`

For other severity levels, you can enter the integer values directly.

CHAPTER

13 The Hello World Sample Cartridge

The hello world sample cartridge illustrates how to use the WRB Transaction Service with the OCI database access API to perform database transactions.

helloworld.c

```
/* Copyright (c) Oracle Corporation 1997. All Rights Reserved */

#ifndef ORATYPES_ORACLE
# include <oratypes.h>
#endif

#ifndef WRB_ORACLE
# include <wrb.h>
#endif

#include <tx.h>
#include <ocidfn.h>
#include <ocikpr.h>
#include <stdio.h>
```

```
typedef struct _appctx
{
    ub2    count;
} AppCtx;

WRBReturnCode test_init();
WRBReturnCode test_exec();
WRBReturnCode test_shut();
```

testentry()

```
WRBReturnCode testentry (WRBCalls)
WRBCallbacks *WRBCalls;
{
    WRBCalls->init_WRBCallback      = test_init;
    WRBCalls->exec_WRBCallback      = test_exec;
    WRBCalls->shut_WRBCallback      = test_shut;
    return (WRB_DONE);
}
```

test_init()

```
WRBReturnCode test_init( WRBCtx, clientcxp )
dvoid   *WRBCtx;
dvoid  **clientcxp;
{
    int           *count;
    int            retval;
    AppCtx        *ctx;

    ctx = (AppCtx *)malloc (sizeof(AppCtx));
    if (!ctx)
        return (WRB_ABORT);

    if (tx_reg("SCOTT") == TX_OK)
    {
        fprintf (stderr, "DAD_SCOTT open successful\n");
        retval  = tx_set_transaction_timeout( 5 );
        printf("\n retval %d", retval );
        switch ( retval )
        {
```

```
                case TX_OUTSIDE:
                   fprintf( stderr, " Calling cartridge not in txn context. \n" );
                   free(ctx);
                   return ( WRB_ERROR );
                case TX_ERROR:
                   fprintf( stderr, " Error in setting the timeout. \n" );
                   free(ctx);
                   return ( WRB_ERROR );
                case TX_FAIL:
                   fprintf( stderr, " Fatal error while setting timeout. \n" );
                   free(ctx);
                   return ( WRB_ABORT );
                default:
                   printf ("Timeout for txn set. \n");
                   break;
            }
            if (tx_open() != TX_OK)
            {
                fprintf (stderr, "Error in TX_openRM\n");
                fprintf (stderr, "\nProgram abort!!!\n");
                free(ctx);
                return (WRB_ABORT);
            }
            else
                fprintf (stderr, "TX_openRM successful\n");

            ctx->count = 0;
            *clientcxp = (void *)ctx;
        }
        else
        {
            fprintf (stderr, "Error in open DAD_SCOTT\n");
            free(ctx);
            return (WRB_ABORT);
        }
        return (WRB_DONE);
    }
```

test_exec()

```
    WRBReturnCode test_exec( WRBCtx, clientcxp )
    dvoid   *WRBCtx;
```

```
dvoid   *clientcxp;
{
    int      bufl;
    char    *uri;
    AppCtx  *ctx = (AppCtx *)clientcxp;
    char     buf[1024];
    TXINFO   *info;
    text       db_name[6] = “SCOTT”;

    WAPIReturnCode rc;

    info = ( TXINFO *) malloc( sizeof( TXINFO ) );

    bufl = sprintf (buf, “Content-type:text/html\n\n”);
    WRBClientWrite(WRBCtx, buf, bufl);

    uri = WRBGetURI (WRBCtx);
    if (strstr(uri, “begintxn”))
    {
        /* The following line is for debug */
        printf(“\ninfo return :%d, state : %d\n”, tx_info(info),
                             (int)info->transaction_state );

        if ((!tx_info(info)) && (info->transaction_state == TX_INACTIVE ))
        {
            if (tx_begin() == TX_OK)
             WRB_printf(WRBCtx, (text *)”Started Transaction ID:%d\n”,
                           ctx->count);
            else
            {
                WRB_printf(WRBCtx, (text *)”Error in beginning transaction\n”);
                 free(info);
                 return WRB_ABORT;
            }
        }
        else
            WRB_printf(WRBCtx, (text *)”Already in transactional context\n”);

        ctx->count++;
    }
    else if (strstr(uri, “committxn”))
    {
```

```
        printf("\ninfo return :%d, state : %d\n", tx_info(info),
                    (int)info->transaction_state );

        if ((!tx_info(info)) && (info->transaction_state == TX_ACTIVE ))
        {
            if (tx_commit() == TX_OK)
               WRB_printf(WRBCtx, (text *)"Committed Transaction\n");
            else
            {
               WRB_printf(WRBCtx, (text *)"Error in commiting transaction\n");
               free(info);
               return WRB_ABORT;
            }
        }
        else
               WRB_printf(WRBCtx, (text *)"Not in transactional context\n");

    }
    else if (strstr(uri, "rollbacktxn"))
    {
        printf("\ninfo return :%d, state : %d\n", tx_info(info),
                      (int)info->transaction_state );
        if ((!tx_info(info)) && (info->transaction_state == TX_ACTIVE ))
        {
            if (tx_rollback() == TX_OK)
                WRB_printf(WRBCtx, (text *)"Transaction rollbacked\n");
            else
            {
                WRB_printf(WRBCtx, (text *)"Error in Transaction rollback\n");
                free(info);
                return WRB_ABORT;
            }
        }
        else
            WRB_printf(WRBCtx, (text *)"Not in transactional context\n");
    }
    else if (strstr (uri, "exit"))
    {
        tx_close();
        free(info);
        return WRB_ABORT;
    }
```

```
else if (strstr (uri, “select”) || strstr(uri, “update”))
{
    text     *select
        = (text *)”select ename, empno, sal from emp where empno = 7369”;
    text     *update
        = (text *)”update emp set sal = sal + 10 where empno = 7369”;

    Lda_Def   lda;
    Cda_Def   cda;
    sb4       len = -1;
    text      emp_name[64];
    sword     empno, salary;
    text     *sql;

    do
    {
        if ((!tx_info(info)) && (info->transaction_state == TX_INACTIVE ))
        {
        WRB_printf(WRBCtx, (text *)”Not in transactional context\n”);
        break;
        }

        sqlld2(&lda, db_name, &len);
        if (&lda == NULL)
        {
        WRB_printf (WRBCtx, (text *)”Error in getting Lda\n”);
        break;
        }

        fprintf(stderr, “lda obtained successfully\n”);
        if (oopen (&cda, &lda, (text *)0, -1, -1, (text *)0, -1))
        {
        fprintf(stderr, “error in oopen\n”);
        oerhms(&lda, cda.rc, buf, sizeof(buf));
        break;
        }

        if (strstr(uri, “select”))
        sql = select;
        else
        sql = update;
```

```
            if (oparse (&cda, sql, -1, 1, (ub4)2))
            {
            fprintf(stderr, “error in oparse\n”);
            oerhms(&lda, cda.rc, buf, sizeof(buf));
            break;
            }

            if (strstr(uri, “select”))
            {
            sb2 ind_ename, ind_empno, ind_sal;
            ub2 retc_ename, retc_empno, retc_sal;
            ub2 retl_ename, retl_empno, retl_sal;

                     /* hard code EMPNO. ENAME, SAL */
           odefin (&cda, 1, (ub1 *)emp_name, sizeof(emp_name), (sword)SQLT_STR,
                 -1, &ind_ename, 0, -1, -1, &retl_ename, &retc_ename);

            odefin (&cda, 2, (ub1 *)&empno, sizeof(empno), (sword)SQLT_INT,
                 -1, &ind_empno, 0, -1, -1, &retl_empno, &retc_empno);

            odefin (&cda, 3, (ub1 *)&salary, sizeof(salary), (sword)SQLT_INT,
                 -1, &ind_sal, 0, -1, -1, &retl_sal, &retc_sal);
            }

            if (oexec (&cda))
            {
            fprintf(stderr, “error in oexec\n”);
            oerhms(&lda, cda.rc, buf, sizeof(buf));
            break;
            }

            if (strstr(uri, “select”))
            {
            if (ofetch(&cda))
            {
                fprintf(stderr, “error in ofetch\n”);
                oerhms(&lda, cda.rc, buf, sizeof(buf));
                break;
            }
            WRB_printf(WRBCtx, (text *)”<br>Employee name: %s, ID: %d, Salary
%d<br>”,
                     emp_name, empno, salary);
```

```
                }
                else
                WRB_printf(WRBCtx, (text *)”Employee salary updated<br>”);

            } while (FALSE);
        }
        else if ((!tx_info(info)) && (info->transaction_state == TX_ACTIVE ))
            WRB_printf(WRBCtx, (text *)”TXN Hello World in Transaction\n”);
        else
            WRB_printf(WRBCtx, (text *)”TXN Hello World not in Transaction yet\n”);

    free(info);
    return (WRB_DONE);
    }
```

test_shut()

```
    WRBReturnCode test_shut( WRBCtx, clientcxp )
    dvoid   *WRBCtx;
    dvoid   *clientcxp;
    {
       free(clientcxp);
       return (WRB_DONE);
    }
```

CHAPTER

14

Migrating to the New TX Interface

This chapter describes how to migrate from the obsolete WRB_TXN interface to the new tx interface.

In earlier versions, the Web Application Server supported transactions with the WRB_TXN interface. In version 3.0, the APIs in that interface have been replaced with new calls in the tx interface. The following table shows how the obsolete APIs map to the new APIs.

Obsolete WRB_TXN API	**New tx API**
openRM	tx_reg() followed by tx_open()
closeRM	tx_close()
beginTransaction	tx_begin()
commitTransaction	tx_commit()
rollbackTransaction	tx_rollback()
transactionInfo	tx_info()
setTimeout	tx_set_transaction_timeout()

Obsolete WRB_TXN API	New tx API
inTransaction	tx_info()
annotateURL	tx_annotate_url()
annotatePath	tx_annotate_path()
resourceInfo	Not applicable
registerRM	tx_reg()

Moving from WRB_openRM() to tx_open()

Unlike the old openRM call, tx_open() does not take any parameters. Before you call tx_open(), you need to register all the resource managers using the tx_reg().

For example, this routine uses the old WRB_TXN calls:

```
test_init(dvoid *WRBCtx, dvoid **clientCtx) {
    for (i=0; i<num_rm; i++)
        hRM[i] = WRB_openRM(WRBCtx, user[i], passwd[i],
                    connect_str[i]);
}
```

Using the new tx interface, the routine would look like:

```
test_init(dvoid *WRBCtx, dvoid **clientCtx) {
    for (i=0; i<num_rm; i++)
        tx_reg(dad_name[i]); /*a DAD is used to identify the RMs */
    tx_open();
}
```

Moving from WRB_TXNregisterRM Call

This function is now obsolete; this means that the dynamic registration feature (that is, getting all the RMs involved in a specific transaction) cannot be specified.

For example, this routine uses the old WRB_TXN calls:

```
test_exec (dvoid *WRBCtx, dvoid **clientCtx) {
    uri = WRB_getURI(WRBCtx);
```

```
    uritype = uri_type(uri);

    if (uritype == BEGINTXN) {
        WRB_TXNbeginTransaction(WRBCtx);
        WRB_TXNregisterRM(WRBCtx, hRM[0]);
        WRB_TXNregisterRM(WRBCtx, hRM[2]);
    }
}
```

Using the new tx interface, the routine would look like:

```
test_exec (dvoid *WRBCtx, dvoid **clientCtx) {
    uri = WRB_getURI(WRBCtx);
    uritype = uri_type(uri);

    if (uritype == BEGINTXN)
        tx_begin();
}
```

Migrating from the resourceInfo Call

ResourceInfo is now obsolete. To get the LDA, you need to call `sqlld2` directly. For example:

```
#include <ocidfn.h>
#include <ocikpr.h>
...
Lda_Def    *lda;
sb4        len = -1;
ub1        *hstdef;
...
sqlld2(lda, dad_name, &len);
...
    /* to get hstdef for UPI programs */
    hstdef = (ubl *)lda->rcsp;
```

Note that the LDA retrieved from `sqlld2` is valid only when a transaction is currently in progress.

CHAPTER

15 Web Request Broker (version 2.0) Core API Reference

This section describes the WRB APIs in version 2.0, and is provided for users who need to upgrade their cartridges from 2.0 to 3.0. If you are developing cartridges for Web Application Server 3.0, you should use the new APIs, which are documented in Web Request Broker Core API Reference.

WRB 2.0 APIs

- `WRBClientRead()`—Read POST data from the requestor
- `WRBClientWrite()`—Write response data to the requestor
- `WRBLogMessage()`—Complete HTTP response header
- `WRBGetAppConfig()`—Get cartridge configuration parameters
- `WRBGetCharacterEncoding()`—Get the requestor's preferred character sets
- `WRBGetClientIP()`—Get the requestor's IP address
- `WRBGetConfigVal()`—Get cartridge configuration parameter value
- `WRBGetContent()`—Get the request query string or POST data

- `WRBGetEnvironment()`—Get all Web Listener environment variables
- `WRBGetEnvironmentVariable()`—Get an environment variable value
- `WRBGetLanguage()`—Get the requestor's preferred languages
- `WRBGetMimeType()`—Get the MIME type for a specified file extension
- `WRBGetNamedEntry()`—Get a name-value pair from parsed content
- `WRBGetORACLE_HOME()`—Get the Web Listener's `ORACLE_HOME` value
- `WRBGetParsedContent()`—Get request content as name-value pairs
- `WRBGetPassword()`—Get the currently authenticated user's password
- `WRBGetReqMimeType()`—Get the MIME type of the current request
- `WRBGetURI()`—Get the current request URI
- `WRBGetURL()`—Get the current request URL
- `WRBGetUserID()`—Get the currently authenticated user's username
- `WRBLogMessage()`—Append a text string to the WRBX's log file
- `WRBReturnHTTPError()`—Return a standard HTTP error message
- `WRBReturnHTTPRedirect()`—Redirect the current request to another URI
- `WRBSetAuthorization()`—Set up authentication for a cartridge

WRB 2.0 Data Types

- WRB Cartridge Function Return Codes
- WRB Error Codes
- The WRBCallbacks Structure
- The WRBEntry Structure

WRBClientRead()

Reads a specified number of bytes of the POST data that accompanies the current HTTP request into a specified buffer.

Note: Currently, you must make all `WRBClientRead()` calls for a given HTTP request before your first call to `WRBClientWrite()`.

You can call this function from your Exec function to get the POST data associated with the current request. This function is especially useful for buffering raw POST data.

If the POST data is in the form of name-value pairs, however, it is usually more convenient to call `WRBGetParsedContent()` instead.

The differences between `WRBClientRead()` and `WRBGetContent()` are:

- `WRBGetContent()` can retrieve either query string or POST data, whereas `WRBClientRead()` retrieves only POST data.
- `WRBGetContent()` retrieves all POST data at once, whereas `WRBClientRead()` retrieves only the specified number of bytes.

Syntax

```
ssize_t WRBClientRead( void *WRBCtx, char *szData, int nBytes );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szData	A pointer to the buffer where you want the function to put the data it reads from the requestor.
→ nBytes	The size in bytes of the buffer pointed to by szData.

Return Values

The number of bytes successfully read from the requestor.

See Also

`WRBClientWrite()`, `WRBGetContent()`, and `WRBGetParsedContent()`

WRBClientWrite()

Writes a specified number of bytes from a specified buffer to a requestor in response to the current HTTP request.

Note: Currently, you must make all `WRB_walkPBlock()` calls for a given HTTP request before your first call to `WRBClientWrite()`.

You can call this function from your Exec function to send data to a requestor in response to the current request. You can use this function to send both HTTP data, such as `Content-type:` and `Set-Cookie:` headers, and actual content.

You can also use this function to:

- generate error messages in response to invalid requests or when cartridge errors occur (see `WRBReturnHTTPError()` and `WRBReturnHTTPRedirect()`)
- generate redirection messages

Syntax

```
ssize_t WRBClientWrite( void *WRBCtx, char *szData, int nBytes );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szData	A pointer to the buffer containing the data to be written to the requestor.
→ nBytes	The number of bytes to be written to the requestor.

Return Values

The number of bytes successfully written to the requestor.

See Also

`WRBReturnHTTPError()`, `WRBReturnHTTPRedirect()`, and `WRBCloseHTTPHeader()`

WRBCloseHTTPHeader()

Finishes writing an HTTP header to a requestor after previous calls to `WRBReturnHTTPError()` or `WRBReturnHTTPRedirect()` for which the *close* parameter was set to `FALSE`.

You can call this function from your Authorize or Exec function if you have previously called `WRBReturnHTTPError()` or `WRBReturnHTTPRedirect()` with the *close* parameter set to FALSE.

Syntax

```
ssize_t WRBCloseHTTPHeader( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

The number of bytes successfully written to the requestor.

See Also

`WRBReturnHTTPError()` and `WRBReturnHTTPRedirect()`

WRBGetAppConfig()

Retrieves the cartridge configuration data defined in the cartridge configuration file. It does not read the file directly, but passes back the configuration data that the Web Listener loaded at start-up, or when signalled to reload its own configuration data.

If you implemented a Reload function for your cartridge, it can call `WRBGetAppConfig()` to load the updated configuration data for your cartridge. See The Reload Function.

Syntax

```
char **WRBGetAppConfig( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the first element in an array of pointers to name-value pairs constituting the configuration data for the calling cartridge. Each element of the array points to a character string of the form:

```
parameter=value
```

where *parameter* is the name of a configuration parameter and *value* is its value. The array is terminated by a NULL pointer.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetConfigVal()`

WRBGetCharacterEncoding()

Returns a comma-separated list of character set identifiers indicating the character sets that the requestor can accept in response to the current request.

If your cartridge can serve content using more than one character set, your Exec function can call this function to get a list of the character sets that the requestor prefers. The Exec function should then choose the first character set in this list in which your cartridge can generate a response.

Syntax

```
char *WRBGetCharacterEncoding( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to a comma-separated list of character set identifiers.

See Also

`WRBGetLanguage()`

WRBGetClientIP()

Returns the IP address of the requestor that issued the current HTTP request.

You can call this function from your Authorize or Exec function. Usually, Authorize calls this function to make sure that a requestor issued the current request from a trusted host.

Syntax

```
ub4 WRBGetClientIP( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

An IP address in the form of an unsigned 32-bit integer in host byte order, each byte of which encodes a quad of the IP address.

See Also

`WRBSetAuthorization()`, `WRBGetUserID()`, and `WRBGetPassword()`

WRBGetConfigVal()

Returns a pointer to the value of a specified cartridge configuration parameter. It does not read the configuration file directly, but retrieves the value from the cartridge configuration data that the Web Listener loaded at start-up, or when signalled to reload its own configuration data.

You can call this function whenever you need the current value of a configurable parameter. Your cartridge does not need to maintain local copies of these parameters in static data or in its application context structure.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetConfigVal( void *WRBCtx, char *name );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ name	The name of the configuration parameter of which you want the value.

Return Values

A pointer to the value of the configuration parameter specified by `name`.

See Also

`WRBGetAppConfig()`

WRBGetContent()

Returns a pointer to the current HTTP request's query string if the request method is GET, or the request's POST data if the request method is POST.

You can call this function from your Exec function to get the unmanipulated query string or POST data for the current request.

If your query string or POST data takes the form of name-value pairs, however, such as when you are processing an HTML form, it's usually more convenient to call `WRBGetParsedContent()` instead.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetContent( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the query string or POST data associated with the current HTTP request. It returns NULL if the requestor sent no content.

See Also

`WRBGetParsedContent()`

WRBGetEnvironment()

Retrieves the environment of the Web Listener process. You can call this function from any cartridge function.

To get the value of a specific environment variable, it's usually more convenient to call `WRBGetEnvironmentVariable()` or `WRBGetORACLE_HOME()`.

Syntax

```
char **WRBGetEnvironment( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the first element in an array of pointers to name-value pairs that define the Web Listener environment. Each name-value pair is a character string that has the form:

`variable=value`

where *variable* is the name of an environment variable and *value* is its value. The array is terminated by a NULL pointer.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetORACLE_HOME()` and `WRBGetEnvironmentVariable()`

WRBGetEnvironmentVariable()

Returns a pointer to the value of a specified environment variable that the Web Listener process inherited when it was started, or a CGI environment variable set by the current request. You can call this function from any cartridge function.

If you need the value of `ORACLE_HOME`, it is more convenient to call `WRBGetORACLE_HOME()`.

Syntax

```
char *WRBGetEnvironmentVariable( void *WRBCtx, char *szEnvVar );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szEnvVar	The name of environment variable of which you want the value.

Return Values

A pointer to the value of the Web Listener environment variable specified by *szEnvVar*. It returns NULL if the specified environment variable is not in the Web Listener environment.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetORACLE_HOME()` and `WRBGetEnvironment()`

WRBGetLanguage()

Returns a comma-separated list of language identifiers indicating the natural languages in which the requestor prefers to receive a response to the current request.

If your cartridge can serve content in more than one natural language, your Exec function can call this function to get a list of the languages that the requestor prefers. Exec should then choose the first language in this list in which your cartridge can generate a response.

Syntax

```
char *WRBGetLanguage( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to a comma-separated list of language identifiers.

See Also

`WRBGetCharacterEncoding()`

WRBGetMimeType()

Returns a pointer to the MIME type that the WRB application engine associates with a specified filename extension. You can call this function in your Exec function to get the MIME type that the requestor is requesting. You can then use this value to set the MIME type of your response.

It's usually more convenient, however, to call `WRBGetReqMimeType()` to do this instead.

Syntax

```
char *WRBGetMimeType( void *WRBCtx, char *extension );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ extension	The filename extension for which you want to get the corresponding MIME type.

Return Values

A pointer to the name of the MIME type. If the specified filename extension does not correspond to any MIME type listed in the WRB configuration, `WRBGetMimeType()` returns `"text/html"`.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetReqMimeType()`

WRBGetNamedEntry()

Returns a pointer to the value of an entry in the parsed content array passed back by `WRBGetParsedContent()`. You specify the entry by name.

You can call this function from your Exec function to get the value of a specific query string or POST data entry after first calling `WRBGetParsedContent()`. This is useful for parsing HTML forms when you know in advance the names of specific fields.

Syntax

```
char *WRBGetNamedEntry( char     *entryName,
                        WRBEntry *WRBEntries,
                        int       numEntries );
```

Parameters

→ entryName	The name of the POST data entry of which you want the value.
→ WRBEntries	The pointer to the parsed content array passed back by WRBGetParsedContent().
→ numEntries	The total number of POST data entries passed back by WRBGetParsedContent().

Return Values

The value of the parsed content entry specified by `entryName`. It returns NULL if the specified entry is not found.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetParsedContent()` and The WRBEntry Structure

WRBGetORACLE_HOME()

Returns a pointer to the value of the ORACLE_HOME environment variable that the Web Listener process inherited when it was started.

For convenience, a cartridge can store data files, scripts, or other files it needs in the file system under the ORACLE_HOME directory. This allows the cartridge to store files in a central place without using hard-coded pathnames. To access the files, you can use the value returned by WRBGetORACLE_HOME() as a path prefix.

Syntax

```
char *WRBGetORACLE_HOME( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the value of the ORACLE_HOME environment variable.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

WRBGetEnvironment() and WRBGetEnvironmentVariable()

WRBGetParsedContent()

Retrieves the current HTTP request's query string if the request method is GET, or its POST data if the request method is POST. You can call this function from your Exec function.

This function parses this data and passes back an array of pointers to The WRBEntry Structure structures. Each `WRBEntry` structure contains the name and value of a POST data entry. This function also passes back the number of elements in the array.

The pointers passed back refer to WRB application engine memory that your cartridge should not modify.

If the query string or POST data for a request does not take the form of name-value pairs, you can call `WRBGetContent()` instead to get the data in raw form.

Syntax

```
WRBReturnCode WRBGetParsedContent( void      *WRBCtx,
                                   WRBEntry **WRBEntries,
                                   int       *numEntries );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
← WRBEntries	A pointer to the location where you want the function to put a pointer to the parsed content array. To specify this parameter, you can declare a variable of type `WRBEntry *`, and pass variable's address.
← numEntries	A pointer to the location where you want the function to put the total number of entries.

Return Values

`WRBGetParsedContent()` returns a value of type `WRBReturnCode`.

See Also

`WRBGetContent()`, `WRBGetNamedEntry()`, and The WRBEntry Structure

WRBGetPassword()

Returns a pointer to the password that the requestor entered in response to an authentication challenge from the cartridge (see `WRBSetAuthorization()`).

You can call this function from your Authorize or Exec function. Your Authorize function can call this function plus `WRBGetUserID()` to authenticate the requestor who issued the current request. Your Exec function might call this function if your cartridge maintains user accounts.

Syntax

```
char *WRBGetPassword( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the password. It returns NULL if no password is found.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBSetAuthorization()` and `WRBGetUserID()`

WRBGetReqMimeType()

Returns a pointer to the MIME type that the WRB application engine associates with the filename extension of the current HTTP request URI.

You can call this function in your Exec function to get the MIME type that the requestor is requesting. You can then use this value to set the MIME type of your response.

Syntax

```
char *WRBGetReqMimeType( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the MIME type. The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetMimeType()`

WRBGetURI()

Returns a pointer to the Uniform Resource Identifier (URI) for the HTTP request currently being handled by the calling WRB cartridge.

You can call this function from your Authorize or Exec function. Your Authorize function, for example, can apply different authorization checks for different URIs. Your Exec function can call `WRBGetURI()` to determine how to satisfy the current request.

Although you can extract the query string for a GET request from its URI , it is usually more convenient to call `WRBGetContent()` or `WRBGetParsedContent()` to do this.

Syntax

```
char *WRBGetURI( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the URI of the current HTTP request. The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetURL()`

WRBGetURL()

Returns a pointer to the Uniform Resource Locator (URL) for the HTTP request currently being handled by the calling WRB cartridge.

Your can call this function from your Authorize or Exec function. Your Authorize function, for example, can apply different authorization checks for different URLs. Your Exec function can call this function to determine how to satisfy the current request.

Although you can extract the query string for a GET request from its URL , it is usually more convenient to call `WRBGetContent()` or `WRBGetParsedContent()` to do this.

Syntax

```
char *WRBGetURL( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

A pointer to the URL of the current HTTP request. The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBGetURI()`

WRBGetUserID()

Returns a pointer to the username that the requestor entered in response to an authentication challenge from the cartridge (see `WRBSetAuthorization()`).

You can call this function from your Authorize or Exec function. Your Authorize function can call this function plus `WRBGetPassword()` to authenticate the requestor who issued the current request. Your Exec function might call this function if your cartridge maintains user accounts.

Syntax

```
char *WRBGetUserID( void *WRBCtx );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Value

A pointer to the username. It returns NULL if no username is found.

The returned pointer refers to WRB application engine memory that your cartridge should not modify.

See Also

`WRBSetAuthorization()` and `WRBGetPassword()`

WRBLogMessage()

Appends a specified character string to the log file for the calling WRBX, which resides in the directory **$ORACLE_HOME/ows30/log** and is named **wrb*_cartridge_procID***, where *cartridge* is the cartridge name (defined using the Web Request Broker administration pages), and *procID* is the process ID of the WRBX.

You can call this function from any cartridge function to debug your cartridge, and to keep a log of cartridge transactions.

Syntax

```
void WRBLogMessage( void *WRBCtx, char *message, int nSeverity );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ message	A pointer to the buffer containing the message to be written to the log file.
→ nSeverity	If this parameter is greater than zero, the function logs the specified message. If the parameter is zero, the function returns immediately without logging the message.

Return Value

None.

See Also

Debugging Your WRB Cartridge

WRBReturnHTTPError()

Sends a standard HTTP error to a requestor in response to the current HTTP request. You can call this function from your Authorize or Exec function in response to an invalid request, or to notify the requestor of a cartridge error.

Note: You must make any `WRBReturnHTTPError()` call for a given HTTP request before your first call to `WRBClientWrite()`.

Syntax

```
ssize_t WRBReturnHTTPError( void *WRBCtx,
                            WRBErrorCode nErrorCode,
                            char *szErrorMesg,
                            boolean close );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ nErrorCode	An HTTP status number identifying the nature of the error. This must specify a standard HTTP error code. See `WRBErrorCode`.
→ szErrorMessage	A pointer to the error message string that you want to display to the requestor. You may specify a custom error message to be displayed to the requestor. If you pass a NULL error message pointer, the function uses the standard HTTP error message for the specified error code.
→ close	Specify `TRUE` if you want to end the HTTP header that is sent to the requestor. Specify `FALSE` if you want to write additional HTTP header information, such as `Set-Cookie:` headers, before completing the HTTP header. You must subsequently call `WRBCloseHTTPHeader()` to finish writing the HTTP header to the requestor.

Return Values

The number of bytes successfully written to the requestor.

See Also

`WRBCloseHTTPHeader()` and `WRBReturnHTTPRedirect()`

WRBReturnHTTPRedirect()

Redirects the current HTTP request to a specified URI. You can call this function from your Authorize or Exec function to send a standard HTTP redirection response to a requestor when the current request uses an outdated URI. This capability allows you to revise or reorganize your cartridge while still supporting URIs used by previous versions.

Note: Currently, you must make any `WRBReturnHTTPRedirect()` call for a given HTTP request before your first call to `WRBClientWrite()`.

Syntax

```
ssize_t WRBReturnHTTPRedirect( void *WRBCtx,
                               char *szURI,
                               boolean close );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ szURI	The URI to which to redirect the request.
→ close	Specify `TRUE` if you want to end the HTTP header that is sent to the requestor. Specify `FALSE` if you want to write additional HTTP header information, such as `Set-Cookie:` headers, before completing the HTTP header. You must subsequently call `WRBCloseHTTPHeader()` to finish writing the HTTP header to the requestor.

Return Values

The number of bytes successfully written to the requestor.

See Also

`WRBCloseHTTPHeader()` and `WRBReturnHTTPError()`

WRBSetAuthorization()

Sends a challenge to the requestor that issued the current HTTP request, and sets up an authentication realm for the browser to use in prompting the user for a username and password for the current request URI.

Syntax

```
WRBReturnCode WRBSetAuthorization( void *WRBCtx,
                                  WRBAuthScheme nScheme,
                                  char *szRealm,
                                  boolean bAndOrFlag );
```

Parameters

→ WRBCtx	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
→ nScheme	The type of authentication or restriction you want to set up with the requestor (see WRB Authentication Schemes).
→ szRealm	The name of an authentication realm for the requestor to use in prompting the user for a username and password. This parameter must not be NULL if *nScheme* specifies authentication. If *nScheme* specifies restriction, this parameter must specify an existing group of domains or IP addresses defined in the WRB configuration.
→ bAndOrFlag	If set to FALSE in two successive calls, this parameter specifies that requestors must satisfy both specified schemes. If set to TRUE in both calls, this parameter specifies that requestors need only satisfy one of the specified schemes.

Return Values

`WRBSetAuthorization()` returns a value of type `WRBReturnCode`.

WRB Authentication Schemes

The *nScheme* parameter can take these values:

Scheme	Description
WRB_AUTH_BASIC_EXIST	Specifies an existing basic authentication scheme defined in the WRB configuration.
WRB_AUTH_BASIC_NEW	Specifies a new basic authentication scheme.
WRB_AUTH_DIGEST_EXIST	Specifies an existing digest authentication scheme defined in the WRB configuration.
WRB_AUTH_DIGEST_NEW	Specifies a new digest authentication scheme.
WRB_AUTH_DOMAIN	Specifies domain-based restriction using an existing group of domains defined by the WRB administration pages.
WRB_AUTH_IP	Specifies IP-based restriction using an existing group of IP addresses defined by the WRB administration pages.

Usage

You can call `WRBSetAuthorization()` from your Init or Authorize cartridge function. To set the authentication scheme for your cartridge globally once and for all, call `WRBSetAuthorization()` from Init.

To control access to your cartridge in more detail (for example, to assign different authentication schemes or realms to different URIs), call `WRBSetAuthorization()` from Authorize. If you call `WRBSetAuthorization())` from both functions, calls made from Authorize supersede those made from Init.

You can also use `WRBSetAuthorization()` to apply a restriction scheme to the current request URI. If you specify `WRB_AUTH_DOMAIN` for *nScheme*, you must use the *szRealm* parameter to specify an existing group of domains defined by the WRB administration pages.

Similarly, if you specify `WRB_AUTH_IP`, *szRealm* must specify an existing group of IP addresses defined in the WRB configuration.

Using Two Schemes

You can call `WRBSetAuthorization()` twice in succession to specify that both an existing authentication scheme and a restriction scheme be applied in authorizing requestors. If you set *bAndOrFlag* to `FALSE` in both calls, the requestor must satisfy both schemes to be authorized; if you set *bAndOrFlag* to `TRUE`, the requestor need only satisfy one of the two schemes.

Note: Both specified schemes must be existing schemes defined in the WRB configuration.

See Also

`WRBGetUserID()`, `WRBGetPassword()`, and `WRBGetClientIP()`

WRB Cartridge Function Return Codes

These are the return codes of type WRBReturnCode that cartridge functions must return. Some WRB API functions also return these codes:

Return Code	Description
WRB_DONE	Indicates normal completion.
WRB_ERROR	Indicates that the request could not be completed because of an error.
WRB_ABORT	Indicates a severe error has occurred: the calling WRBX should terminate immediately.

Note: Some WRB API functions return values of type `WAPIReturnCode` instead. See WRB Error Codes.

WRB Error Codes

Variables of type `WRBErrorCode` can assume these standard HTTP status values:

Status Code	Meaning
200	OK
201	Created
202	Accepted
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

Currently, you specify WRB error codes as casts, for example:

```
(WRBErrorCode)400
```

See Also

`WRBReturnHTTPError()`

The WRBCallbacks Structure

The `WRBCallbacks` type defines the dispatch table that the WRB application engine uses to call your cartridge functions. See The Entry-Point Function for more information.

```
struct WRBCallbacks
{
   WRBReturnCode (*init_WRBCallback)();
   WRBReturnCode (*exec_WRBCallback)();
   WRBReturnCode (*shut_WRBCallback)();

   WRBReturnCode (*reload_WRBCallback)();

   char         *(*version_WRBCallback)();
   void          (*version_free_WRBCallback)();
   WRBReturnCode (*authorize_WRBCallback)();
};
typedef struct WRBCallbacks WRBCallbacks;
```

The WRBEntry Structure

The `WRBGetParsedContent()` and `WRBGetNamedEntry()` functions use the `WRBEntry` type to store name-value pairs extracted from the query string or POST data that accompanies an HTTP request.

```
struct WRBEntry
{
   char *name;
   char *value;
};
typedef struct WRBEntry WRBEntry;
```

Index

D

E

G

H

I

L

M

O

P

R

S

T

U

V

W