Web Request Broker Programmer's Reference™

Release 2.1

Part No. [[Part Number]]

ORACLE®

Enabling the Information Age

Web Request Broker Programmer's Reference, 2.1

Part No. [[Part Number]]

Primary Author: Kennan Rossi

Contributors: Seshu Adunuthula, Mala Anand, Jeffrey Caesar

Contents

CHAPTER

1 WRB Overview

The Web Request Broker is the central component of the Oracle WebServer. 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.

Currently, you use the C language to write WRB cartridges. In the future, however, the WRB API will also be available in other languages.

WRB Architecture

The WRB consists of these components:

- The WRB Dispatcher
- 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 *WRB Dispatcher* is a program that provides the interface between Web Listeners and WRB cartridges. The WRB 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 WRB Dispatcher. The Web Listener uses its configuration data to map URLs to WRB cartridges.

When the Dispatcher receives a request, it determines the WRB cartridge to which the request is directed, and dispatches the request to an execution instance of that cartridge. 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. It provides the interface between WRB cartridges and the WRB dispatcher, directs WRB cartridge flow of control, and provides services for WRB cartridges to use.

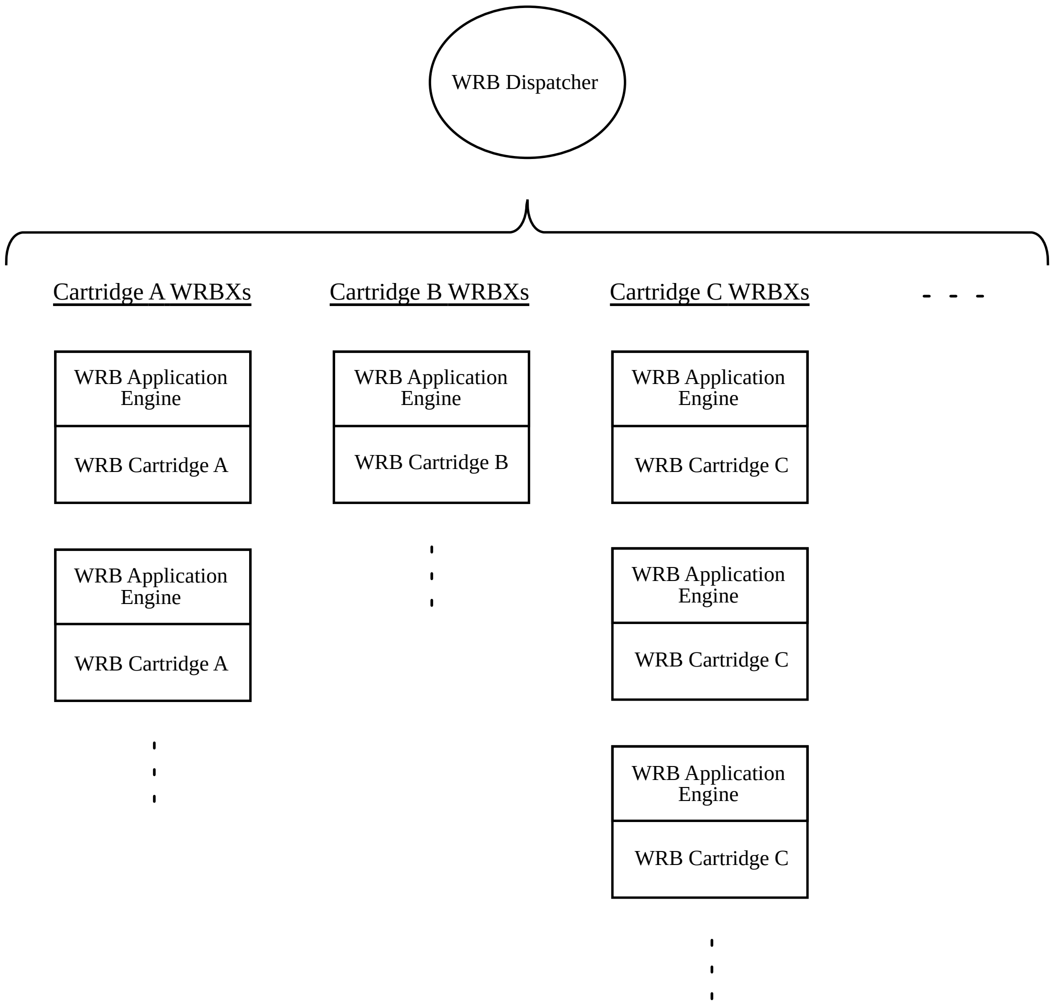

Figure 1-1: The WRB Dispatcher manages execution instances of WRB cartridges

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).

The WRB application engine functions are declared in the header **wrb.h**. The function names all begin with “`WRB`.”

CHAPTER

2

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. It's divided into these sections:

- Writing a WRB Cartridge
- Compiling, Linking, Configuring, and Running a WRB cartridge

For a general overview of the Web Request Broker architecture under which cartridges run, see WRB Overview in *Introduction to the Web Request Broker*...

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.
- 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.

Though 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's 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 cartridge function.)

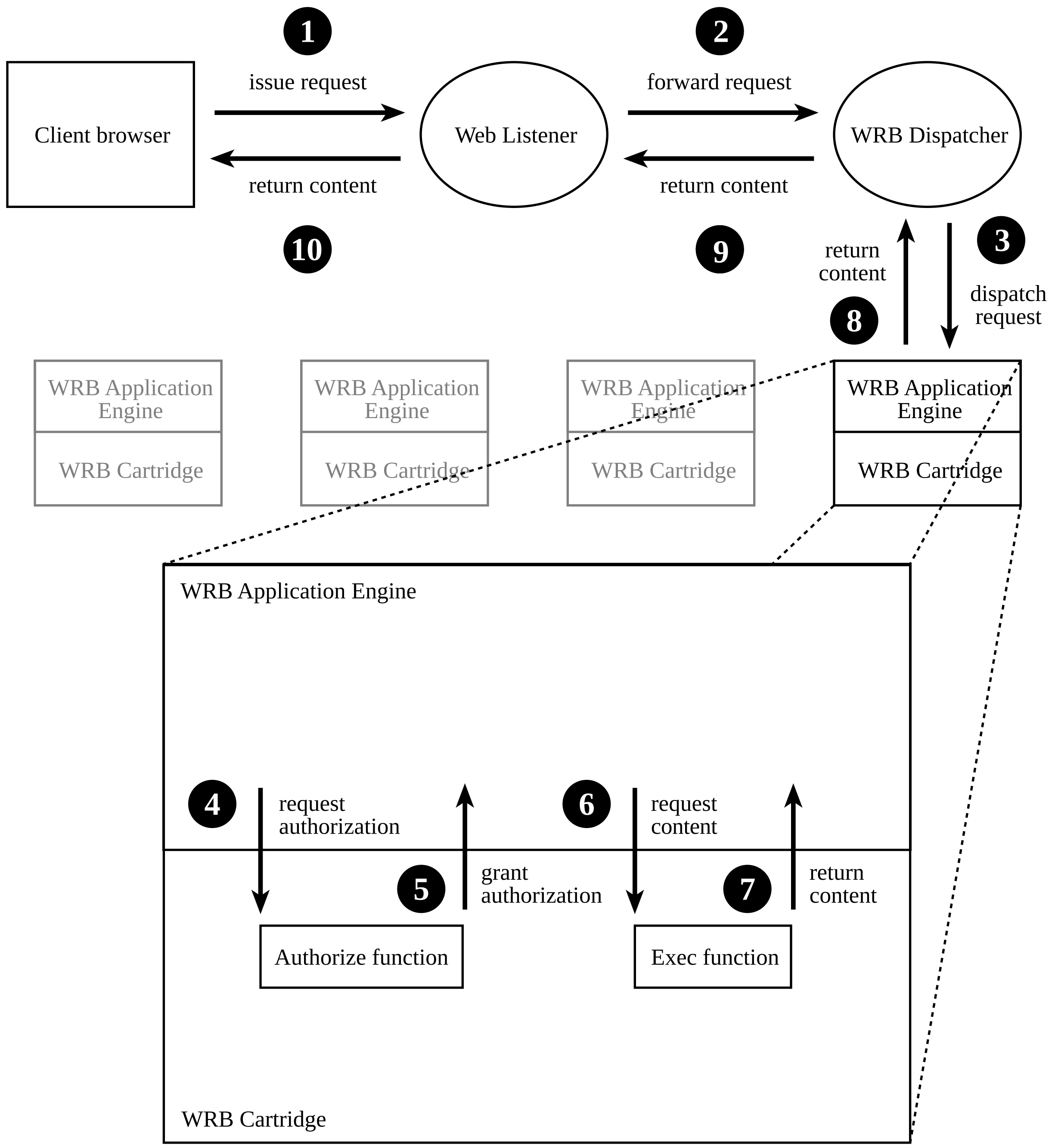

Figure 2-1: A request successfully handled by a WRB cartridge

Compiling, Linking, Configuring, and Running a WRB cartridge

When you want to begin testing your cartridge, you'll need to compile and link it as a shared library and configure a Web Listener on your WebServer 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/ows21/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.

The following sample makefile is taken from the MyWRBApp cartridge:

```
#Makefile for building WRB Cartridges
#===================================

TOP     =   $(ORACLE_HOME)/ows21
LIBHOME =   $(TOP)/wrbsdk/lib
INCHOME =   $(TOP)/wrbsdk/inc
LDCOM   =   -g -xs -L$(LIBHOME)
DESTDIR =   $(TOP)/sample/wrbsdk/mywrbapp
SLIBS   =   -lnsl -lm -lsocket -ldl -laio
# DFLAGS must be -DDEBUG to compile a version that logs debug output
DFLAGS  =   -DDEBUG

all: mywrbapp.so

mywrbapp.o: mywrbapp.h mywrbapp.c
    $(CC) -c $(DFLAGS) -o $@ -g -I$(INCHOME) mywrbapp.c

#The link line for the final .so dynamic library is given below
mywrbapp.so: mywrbapp.o
    $(CC) $(LDCOM) -o $@ -G mywrbapp.o $(SLIBS)

clean:
    rm -f mywrbapp.so mywrbapp.o

install: all
    cp mywrbapp.so $(DESTDIR)
```

Configuring a Web Listener to use your cartridge

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

1. Go to the Web Request Broker Administration page and click the Modify link for the listener of your choice.
2. Go to the Applications and Objects section and create an entry specifying the location of your cartridge shared library and the name of your Entry function.
3. Go to the Applications and Directories section and create an entry defining any virtual directory mappings that your cartridge needs.
4. Go to the top of the form and follow the link to the Web Listener configuration page for the listener.
5. Go to the Directory Mappings section to set up the necessary virtual mappings that request URLs will use to access any static files related to your cartridge such as splash or registration pages.
6. Go to the Web Listener home page and stop and restart the Listener you have just configured.
7. Use your web browser to issue a request to your cartridge—your URL must specify the port number of the configured Listener and a virtual pathname assigned to your cartridge.

Debugging your WRB cartridge

To debug your cartridge, you can use the `WRBLogMessage()` WRB API function in your cartridge functions to log debugging messages to log files.

As you test your cartridge, you can monitor its log file to see what it's doing:

1. **cd $ORACLE_HOME/ows21/log**
2. **ls -lt | more**

   This lists the files in the directory in order of modification time, with the most recently modified files at the top. Near the top of the listing you should see a filename of the form **wrb_*cartridge_proc-id***, where *cartridge* is the name of your cartridge and *proc-id* is the process ID of the WRBX in which the cartridge is running. There may be many files in the directory with names like this—if so, choose the most recently modified file.
3. **more wrb_*cartridge_proc-id***

   If you repeat this step after every HTTP request, you can see the new logging output your cartridge generates in handling each request. You might want to repeat step 2 occasionally to make sure you're still looking at the right log file.

Each time you edit and recompile your cartridge, you must stop and restart the Listener you're using for testing and repeat these steps.

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'll specify the full path to your cartridge shared library and the name of your entry-point function. It's 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

Parameter	Type	Description
WRBcalls	WRBCallbacks *	A pointer to the cartridge function table to be filled in.

Table 2-1: Entry-point function parameters

Return values

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

Return value	Description
WRB_DONE	Successful completion
WRB_ERROR	An error occurred—the request cannot be completed
WRB_ABORT	An error occurred from which the application cannot recover—terminate the application

Table 2-2: Entry-point function return values

Example

See the MyWRBApp sample function `MyWRBApp_Entry()`.

The Init cartridge 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.

Syntax

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

Parameters

Parameter	**Type**	**Description**
WRBCtx	void *	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	void **	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.

Table 2-3: Init cartridge function parameters

Return Values

Your Init function must return a value of type `WRBReturnCode` (see Entry-point function return values).

Example

See the MyWRBApp sample function `MyWRBApp_Init()`.

Related information

See The Shutdown cartridge function.

The Authorize cartridge 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 Authorize function must then pass back a boolean value indicating whether the client is authorized to issue the request. You can use the `WRBGetUserID()`, `WRBGetPassword()`, and `WRBGetClientIPAddress()` functions to determine the client's privileges.

You can also use the `WRBSetAuthorization()` function to set up an authentication realm for the client to use in prompting the user for username and password.

Note: this cartridge function is optional. If you do not implement it, access to your WRB cartridge can be regulated only by through the Web Listener and WRB configuration pages....

Syntax

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

Parameters

Parameter	Type	Description
WRBCtx	void *	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	void *	A pointer to the application context structure allocated your Init function.
bAuthorized	boolean *	Indicates whether the client is authorized to issue the request.

Table 2-4: Authorize cartridge function parameters

Return values

Your Authorize function must return a value of type `WRBReturnCode` (see Entry-point function return values).

Example

See the MyWRBApp sample function `MyWRBApp_Authorize()`.

Related information

See `WRBGetUserID()`, `WRBGetPassword()`, `WRBGetClientIP()`, and `WRBSetAuthorization()` in the Web Request Broker API Reference.

The Exec cartridge 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 WRBGetURL() WRB API function to get the request URL.

Syntax

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

Parameters

Paramet er	Type	Description
WRBCtx	void *	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	void *	A pointer to the application context structure allocated your Init function.

Table 2-5: Exec function parameters

Return values

Your Exec function must return a value of type `WRBReturnCode` (see Entry-point function return values).

Example

See the MyWRBApp sample function `MyWRBApp_Exec()`.

The Shutdown cartridge 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

The parameters for Shutdown are the same as for Exec—see Exec function parameters.

Return values

Your Shutdown function must return a value of type `WRBReturnCode` (see Entry-point function return values). Returning a value of WRB_DONE implies that your cartridge is ready to exit at any time.

Example

See the MyWRBApp sample function `MyWRBApp_Shutdown()`.

The Reload cartridge 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 `WRBGetAppConfig()` or `WRBGetConfigVal()`.

Syntax

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

Parameters

The parameters for Reload are the same as for Exec—see Exec function parameters.

Return values

Your Reload function must return a value of type `WRBReturnCode` (see Entry-point function return values).

Example

See the MyWRBApp sample function `MyWRBApp_Reload()`.

Related information

See `WRBGetAppConfig()` in the Web Request Broker API Reference.

The Version cartridge 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.

Note: 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"
```

Related information

See The Version_Free cartridge function.

The Version_Free cartridge 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.

Note: 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.

Related information

See The Version cartridge function.

CHAPTER

3

Web Request Broker API Reference

The Web Request Broker (WRB) API Reference provides a lookup reference for programmers using the WRB API to write server-side web applications. The WRB Overview section of *Introduction to the Web Request Broker*... describes WRB applications, called *cartridges*, and gives a general overview of the WRB architecture.

To learn how to develop WRB cartridges, see Writing Applications Using the Web Request Broker API.

The WRB API Reference is divided into these sections:

WRB Application Engine Functions

- `WRBClientRead()`—Read POST data from the client
- `WRBClientWrite()`—Write response data to the client
- `WRBLogMessage()`—Complete HTTP response header
- `WRBGetAppConfig()`—Get cartridge configuration parameters
- `WRBGetCharacterEncoding()`—Get the client's preferred character sets
- `WRBGetClientIP()`—Get the client'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 client'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 Data Types

- WRB Return Codes
- WRB Error Codes
- The WRBCallbacks Structure
- The WRBEntry Structure

WRBClientRead()

`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()`.

Syntax

```
ssize_t WRBClientRead( void *WRBCtx, char *szData, int nBytes );
```

Parameters

Parameter	**Type**	**Description**
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
szData	char *	A pointer to the buffer where you want the function to put the data it reads from the client.
nBytes	int	The size in bytes of the buffer pointed to by szData.

Return Values

`WRBClientRead()` returns the number bytes read from the client.

Usage

You can call `WRBClientRead()` from your Exec function to get the POST data associated with the current request. `WRBClientRead()` is especially useful for buffering raw POST data.

If the POST data is in the form of name-value pairs, however, it's 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.

Related Information

See also `WRBClientWrite()`, `WRBGetContent()`, and `WRBGetParsedContent()`.

WRBClientWrite()

`WRBClientWrite()` writes a specified number of bytes from a specified buffer to a client in response to the current HTTP request.

Note Currently, you must make all `WRBClientRead()` calls for a given HTTP request before your first call to `WRBClientWrite()`.

Syntax

```
ssize_t WRBClientWrite( void *WRBCtx, char *szData, int nBytes );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
szData	char *	A pointer to the buffer containing the data to be written to the client.
nBytes	int	The number of bytes to be written to the client.

Return Values

`WRBClientWrite()` returns the number of bytes successfully written to the client.

Usage

You can call `WRBClientWrite()` from your Exec function to send data to a client in response to the current request. You can use `WRBClientWrite()` to send both HTTP data, such as `Content-type:` and `Set-Cookie:` headers, and actual content.

You can also use `WRBClientWrite()` in generating error messages in response to invalid requests, in generating redirection messages, or when cartridge errors occur (see `WRBReturnHTTPError()` and `WRBReturnHTTPRedirect()`).

Examples

See the MyWRBApp sample function `formatUserData()`.

Related Information

See also `WRBClientRead()`, `WRBReturnHTTPError()`, `WRBReturnHTTPRedirect()`, and `WRBCloseHTTPHeader()`.

WRBCloseHTTPHeader()

`WRBCloseHTTPHeader()` finishes writing an HTTP header to a client after previous calls to `WRBReturnHTTPError()` or `WRBReturnHTTPRedirect()` for which the `close` parameter was set to `FALSE`.

Syntax

```
ssize_t WRBCloseHTTPHeader( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBCloseHTTPHeader()` returns the number bytes successfully written to the client.

Usage

You can call `WRBCloseHTTPHeader()` from your Authorize or Exec function if you have previously called `WRBReturnHTTPError()` or `WRBReturnHTTPRedirect()` with the close parameter set to FALSE.

Examples

This example shows how to send an HTTP status code of 204 (No Content) to the client, write additional HTTP header data, and call `WRBCloseHTTPHeader()` to complete the header.

```
WRBReturnHTTPError(WRBCtx, (WRBErrorCode)204, NULL, TRUE);
/* use WRBClientWrite() to output additional HTTP headers */
WRBCloseHTTPHeader(WRBCtx);
```

Related Information

See also `WRBReturnHTTPError()` and `WRBReturnHTTPRedirect()`.

WRBGetAppConfig()

`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. See The Reload cartridge function.

`WRBGetAppConfig()` returns a pointer to an array of pointers, each of which 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 pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char **WRBGetAppConfig( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetAppConfig()` returns a pointer to the first element in an array of pointers to name-value pairs constituting the configuration data for the calling cartridge.

Usage

If you implement a Reload function, it can call `WRBGetAppConfig()` to load the updated configuration data for your cartridge.

Examples

This example shows how to loop through your cartridge configuration parameters:

```
char **appcfg;
char *varp;

appcfg = WRBGetAppConfig(WRBCtx);

for (varp = *appcfg; varp; appcfg++, varp = *appcfg) {
    /* varp points to a “name=value” pair */
}
```

Related Information

See also `WRBGetConfigVal()`.

WRBGetCharacterEncoding()

`WRBGetCharacterEncoding()` returns a comma-separated list of character-set identifiers indicating the character sets that the client can accept in response to the current request.

Syntax

```
char *WRBGetCharacterEncoding( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetCharacterEncoding()` returns a pointer to a comma-separated list of character-set identifiers indicating the character sets that the client can accept in response to the current request.

Usage

If your cartridge can serve content using more than one character set, your Exec function can call `WRBGetCharacterEncoding()` to get a list of the character sets that the client prefers. Exec should then choose the first character set in this list in which your cartridge can generate a response.

Examples

This example shows how you might choose a character set to use in responding to a request:

```
#include <string.h>

/*
 * Note: this static array is just a convenience for this example.
 * A better way to store this info would be in your application
 * context structure.
 * You could also make the supported languages configurable, and
 * get this list using WRBGetConfigVal().
 */
static char *mycharsets[] = {“iso-8859-1”, “iso-2022-jp”, NULL};

char **mcsp;
char *clientcharsets;
char *charsetp;
char *ep;
boolean match = FALSE;

clientcharsets = WRBGetCharacterEncoding(WRBCtx);

while (clientcharsets && *clientcharsets) {
    for (mcsp = mycharsets, charsetp = *mcsp;
         charsetp;
         mcsp++, charsetp = *mcsp) {
        if (!strncmp(clientcharsets, charsetp, strlen(charsetp))) {
            match = TRUE;
            break;
        }
    }

    if (match)
        break;

    clientcharsets = strchr(clientcharsets, ‘,’);
    clientcharsets++;
}

if (match) {
    /* charsetp points to chosen character set */
}
```

Related Information

See also `WRBGetLanguage()`.

WRBGetClientIP()

`WRBGetClientIP()` returns a the IP address of the client who issued the current HTTP request in the form of an unsigned 32-bit integer, each byte of which encodes a quad of the IP address.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
ub4 WRBGetClientIP( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetClientIP()` returns the IP address of the client who issued the current HTTP request.

Usage

You can call `WRBGetClientIP()` to from your Authorize or Exec function. Usually, Authorize calls `WRBGetClientIP()` to make sure that a client issued the current request from a trusted host.

Examples

This example shows how to get the IP address of the client who issued the current request and convert it to dotted quad notation:

```
#include <stdio.h>

ub4 clientIP;
char clientIPquads[18];

clientIP = WRBGetClientIP(WRBCtx);
```

```
sprintf(clientIPquads, “%u.%u.%u.%u”,
    (clientIP & 0xFF000000) >> 24,
    (clientIP & 0x00FF0000) >> 16,
    (clientIP & 0x0000FF00) >> 8,
    (clientIP & 0x000000FF));
```

Related Information

See also `WRBSetAuthorization()`, `WRBGetUserID()`, and `WRBGetPassword()`.

WRBGetConfigVal()

`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.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetConfigVal( void *WRBCtx, char *name );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
name	char *	The name of the configuration parameter of which you want the value.

Return Values

`WRBGetConfigVal()` returns a pointer to the value of the configuration parameter specified by `name`.

Usage

You can call `WRBGetConfigVal()` from any cartridge function whenever you need the current value of a configurable parameter. Your cartridge therefore does not need to maintain local copies of these parameters in static data or in its application context structure.

Examples

See the MyWRBApp sample function `MyWRBApp_Reload()`.

Related Information

See also `WRBGetAppConfig()`.

WRBGetContent()

`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.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetContent( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetContent()` returns a pointer to the query string or POST data associated with the current HTTP request. It returns NULL if the client browser sent no content.

Usage

You can call `WRBGetContent()` 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.

Examples

This example shows how to get a request's query string or POST data for parsing:

```
char *datap;
```

```
datap = WRBGetContent(WRBCtx);
while (datap && *datap) {
    /* parse the content */
}
```

Related Information

See also `WRBGetParsedContent()`.

WRBGetEnvironment()

`WRBGetEnvironment()` retrieves the environment of the Web Listener process. It returns a pointer to an array of pointers, each of which points to a character string of 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 pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char **WRBGetEnvironment( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetEnvironment()` returns a pointer to the first element in an array of pointers to name-value pairs that define the Web Listener environment.

Usage

You can call `WRBGetEnvironment()` from any cartridge function to retrieve the entire Web Listener environment.

To get the value of a specific environment variable, however, it's usually more convenient to call `WRBGetEnvironmentVariable()` or `WRBGetORACLE_HOME()`.

Examples

This example shows how to loop through all environment variables:

```
char **envp;
char *varp;

envp = WRBGetEnvironment(WRBCtx);

for (varp = *envp; varp; envp++, varp = *envp) {
    /* varp points to a “name=value” string */
}
```

Related Information

See also `WRBGetORACLE_HOME()` and `WRBGetEnvironmentVariable()`.

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.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetEnvironmentVariable( void *WRBCtx, char *szEnvVar );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
szEnvVar	char *	The name of environment variable of which you want the value.

Return Values

`WRBGetEnvironmentVariable()` returns a pointer to the value of the Web Listener environment variable specified by `szEnvVar`. It returns NULL if the specified variable is not in the Web Listener environment.

Usage

You can call `WRBGetEnvironmentVariable()` from any cartridge function when you need the value of a specific Web Listener environment variable.

If you need the value of `ORACLE_HOME`, however, it's more convenient to call `WRBGetORACLE_HOME()`.

Examples

See the MyWRBApp sample function `getCatalog()`.

Related Information

See also `WRBGetORACLE_HOME()` and `WRBGetEnvironment()`.

WRBGetLanguage()

`WRBGetLanguage()` returns a comma-separated list of language identifiers indicating the natural languages in which the client prefers to receive a response to the current request.

Syntax

```
char *WRBGetLanguage( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetLanguage()` returns a pointer to a comma-separated list of language identifiers indicating the languages that the client can accept in response to the current request.

Usage

If your cartridge can serve content in more than one natural language, your Exec function can call `WRBGetLanguage()` to get a list of the languages that the client prefers. Exec should then choose the first language in this list in which your cartridge can generate a response.

Examples

This example shows how you might choose a language in which to respond to a request:

```
#include <string.h>

/*
 * Note: this static array is just a convenience for this example.
 * A better way to store this info would be in your application
 * context structure.
 * You could also make the supported languages configurable, and
 * get this list using WRBGetConfigVal().
 */
static char *mylangs[] = {“en”, “fr-FR”, “jp-JP”, NULL};

char **mlp;
char *clientlangs;
char *langp;
boolean match = FALSE;

clientlangs = WRBGetLanguage(WRBCtx);
while (clientlangs && *clientlangs) {
    for (mlp = mylangs, langp = *mlp; langp; mlp++, langp = *mlp) {
        if (!strncmp(clientlangs, langp, strlen(langp))) {
            match = TRUE;
            break;
        }
    }

    if (match)
        break;

    clientlangs = strchr(clientlangs, ‘,’);
    clientlangs++;
}

if (match) {
    /* langp points to chosen language */
}
```

Related Information

See also `WRBGetCharacterEncoding()`.

WRBGetMimeType()

`WRBGetMimeType()` returns a pointer to the MIME type that the WRB application engine associates with a specified filename extension.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetMimeType( void *WRBCtx, char *extension );
```

Parameters

Parameter	**Type**	**Description**
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
extension	char *	The filename extension for which you want to get the corresponding MIME type.

Return Values

`WRBGetMimeType()` returns a pointer to the name of the MIMEType corresponding to the filename extension specified by `extension`. If the specified filename extension does not correspond to any MIME type listed in the WRB configuration, `WRBGetMimeType()` returns `"text/html"`.

Usage

You can call `WRBGetMimeType()` in your Exec function to get the MIME type that the client 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.

Examples

This example shows how to extract the filename extension from the current URI and use it to determine the request MIME type:

```
#include <string.h>

char *URI;
char *ext;
char *MIMEtype;

URI = WRBGetURI(WRBCtx);
ext = strrchr(URI, '.');
ext++;
MIMEtype = WRBGetMimeType(WRBCtx, ext);
```

Related Information

See also `WRBGetReqMimeType()`.

WRBGetNamedEntry()

`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.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetNamedEntry( char     *entryName,
                        WRBEntry *WRBEntries,
                        int       numEntries );
```

Parameters

Parameter	Type	Description
entryName	char *	The name of the POST data entry of which you want the value.
WRBEntries	WRBEntry *	The pointer to the parsed content array passed back by WRBGetParsedContent().
numEntries	int	The total number of POST data entries passed back by WRBGetParsedContent().

Return Values

`WRBGetNamedEntry()` returns the value of the parsed content entry specified by `entryName`. It returns NULL if the specified entry is not found.

Usage

You can call `WRBGetNamedEntry()` 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.

Examples

See the MyWRBApp sample function `storeInfo()`.

Related Information

See also `WRBGetParsedContent()` and The WRBEntry Structure.

WRBGetORACLE_HOME()

`WRBGetORACLE_HOME()` returns a pointer to the value of the `ORACLE_HOME` environment variable that the Web Listener process inherited when it was started.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetORACLE_HOME( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetORACLE_HOME()` returns a pointer to the value of the Web Listener environment variable `ORACLE_HOME`.

Usage

It's often convenient for a WRB cartridge to store data files, scripts, or other files it needs in the file system under the `ORACLE_HOME` directory on the WebServer machine. This allows the cartridge to store files in a central place without using hard-coded pathnames. To do this, you can use the value returned by `WRBGetORACLE_HOME()` as a path prefix.

Examples

See the MyWRBApp sample function `storeInfo()`.

Related Information

See also `WRBGetEnvironment()` and `WRBGetEnvironmentVariable()`.

WRBGetParsedContent()

`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.

`WRBGetParsedContent()` 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. `WRBGetParsedContent()` 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.

Syntax

```
WRBReturnCode WRBGetParsedContent( void      *WRBCtx,
                                   WRBEntry **WRBEntries,
                                   int       *numEntries );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
WRBEntries	WRBEntry **	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	int *	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`.

Usage

You can call `WRBGetParsedContent()` from your Exec function to get query string or POST data that takes the form of name-value pairs, such as data entered into an HTML form.

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.

Examples

See the MyWRBApp sample function `storeInfo()`.

Related Information

See also `WRBGetContent()`, `WRBGetNamedEntry()`, and The WRBEntry Structure.

WRBGetPassword()

`WRBGetPassword()` returns a pointer to the password that the user of the client browser entered in response to an authentication challenge from the cartridge (see `WRBSetAuthorization()`).

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetPassword( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetPassword()` returns a pointer to the password that the user entered in response to a challenge from the cartridge. It returns NULL if no password is found.

Usage

You can call `WRBGetPassword()` from your Authorize or Exec function. Your Authorize function can call `WRBGetPassword()` and `WRBGetUserID()` to authenticate the client who issued the current request.

Your Exec function might call `WRBGetPassword()` if your cartridge maintains user accounts.

Examples

See the MyWRBApp sample function `MyWRBApp_Authorize()`.

Related Information

See `WRBSetAuthorization()` and `WRBGetUserID()`.

WRBGetReqMimeType()

`WRBGetReqMimeType()` returns a pointer to the MIME type that the WRB application engine associates with the filename extenion of the current HTTP request URI .

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetReqMimeType( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetReqMimeType()` returns a pointer to the MIME type for the current HTTP request.

Usage

You can call `WRBGetReqMimeType()` in your Exec function to get the MIME type that the client is requesting. You can then use this value to set the MIME type of your response.

Examples

This example shows how to get the MIME type being requested by the client and set the type of the response to this type:

```
#include <stdio.h>

char *MIMEtype;
char buf[1024];
int len;
```

```
MIMEtype = WRBGetReqMimeType(WRBCtx);

len = sprintf(buf, “Content-type: %s\n”, MIMEtype);
WRBClientWrite(WRBCtx, buf, len);

/* output rest of header info and content */
```

Related Information

See also `WRBGetMimeType()`.

WRBGetURI()

`WRBGetURI()` returns a pointer to the Uniform Resource Identifier (URI) for the HTTP request currently being handled by the calling WRB cartridge.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetURI( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetURI()` returns a pointer to the URI of the current HTTP request.

Usage

You can call `WRBGetURI()` 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's usually more convenient to call `WRBGetContent()` or `WRBGetParsedContent()` to do this.

Related Information

See also `WRBGetURL()`.

WRBGetURL()

`WRBGetURL()` returns a pointer to the Uniform Resource Locator (URL) for the HTTP request currently being handled by the calling WRB cartridge.

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetURL( void *WRBCtx );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetURL()` returns a pointer to the URL of the current HTTP request.

Usage

Your can call `WRBGetURL()` from your Authorize or Exec function. Your Authorize function, for example, can apply different authorization checks for different URLs. Your Exec function can call `WRBGetURL()` to determine how to satisfy the current request.

Although you can extract the query string for a GET request from its URL , it's usually more convenient to call `WRBGetContent()` or `WRBGetParsedContent()` to do this.

Examples

See the MyWRBApp sample function `MyWRBApp_Exec()`.

Related Information

See also `WRBGetURI()`.

WRBGetUserID()

`WRBGetUserID()` returns a pointer to the username that the user of the client browser entered in response to an authentication challenge from the cartridge (see `WRBSetAuthorization()`).

The pointer returned refers to WRB application engine memory that your cartridge should not modify.

Syntax

```
char *WRBGetUserID( void *WRBCtx );
```

Parameters

Parameter	**Type**	**Description**
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.

Return Values

`WRBGetUserID()` returns a pointer to the username that the user entered in response to a challenge from the cartridge. It returns NULL if no username is found.

Usage

You can call `WRBGetUserID()` from your Authorize or Exec function. Your Authorize function can call `WRBGetUserID()` and `WRBGetPassword()` to authenticate the client who issued the current request.

Your Exec function might call `WRBGetUserID()` if your cartridge maintains user accounts.

Examples

See the MyWRBApp sample functions `MyWRBApp_Authorize()` and `getUserData()`.

Related Information

See `WRBSetAuthorization()` and `WRBGetPassword()`.

WRBLogMessage()

`WRBLogMessage()` appends a specified character string to the log file for calling WRBX, which resides in the directory **$ORACLE_HOME/ows21/log** and is named **wrb_*cartridge*_*proc-id***, where *cartridge* is the cartridge name (defined using the Web Request Broker administration... pages), and *proc-id* is the process ID of the WRBX.

Syntax

```
void WRBLogMessage( void *WRBCtx, char *message, int nSeverity );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
message	char *	A pointer to the buffer containing the message to be written to the log file.
nSeverity	int	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 Values

None.

Usage

You can call `WRBLogMessage()` from any cartridge function to debug your WRB cartridge, and to keep a log of cartridge transactions.

Examples

See the MyWRBApp sample function `MyWRBApp_Init()`.

Related Information

See Debugging your WRB cartridge.

WRBReturnHTTPError()

WRBReturnHTTPError() sends a standard HTTP error to a client in response to the current HTTP request. The nErrorCode parameter must specify a standard HTTP error code of type WRBErrorCode. You may specify a custom error message to be displayed to the client. If you pass a NULL error message pointer, WRBReturnHTTPError() uses the standard HTTP error message for the specified error code.

If you set the close parameter to TRUE, the function sends a complete HTTP header to the client. You can set this parameter to FALSE if you want to write additional HTTP header information, such as Set-Cookie: headers, before completing the HTTP header.

If you set close to FALSE, you must subsequently call WRBCloseHTTPHeader() to finish writing the HTTP header to the client.

Note Currently, 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

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
nErrorCode	WRBErrorCode	An HTTP status number identifying the nature of the error.
szErrorMessage	char *	A pointer to the error message string that you want to display to the client.
close	boolean	When this parameter is set to TRUE, WRBReturnHTTPError() outputs a marker that indicates the end of the HTTP header data. If you specify FALSE, you must call WRBCloseHTTPHeader() explicitly.

Return Values

`WRBReturnHTTPError()` returns the number bytes successfully written to the client.

Usage

You can call `WRBReturnHTTPError()` from your Authorize or Exec function in response to an invalid request, or to notify the client of a cartridge error.

Examples

See the MyWRBApp sample function `MyWRBApp_Exec()`.

Related Information

See also `WRBCloseHTTPHeader()` and `WRBReturnHTTPRedirect()`.

WRBReturnHTTPRedirect()

`WRBReturnHTTPRedirect()` redirects the current HTTP request to a specified URI.

If you specify `TRUE` for the `close` parameter, the function sends a complete HTTP header to the client. You can specify `FALSE` for this parameter if you want to write additional HTTP header information, such as `Set-Cookie:` headers, before completing the HTTP header.

If you specify `FALSE` for `close`, you must subsequently call `WRBCloseHTTPHeader()` to finish writing the HTTP header to the client.

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

Parameter	**Type**	**Description**
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
szURI	char *	The URI to which to redirect the request.
close	boolean	When this parameter is set to TRUE, WRBReturnHTTPRedirect() outputs a marker that indicates the end of the HTTP header data. If you specify FALSE, you must call WRBCloseHTTPHeader() explicitly.

Return Values

`WRBReturnHTTPRedirect()` returns the number bytes successfully written to the client.

Usage

You can call `WRBReturnHTTPRedirect()` from your Authorize or Exec function to send a standard HTTP redirection response to a client 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.

Examples

This example shows how to redirect an HTTP request from an old URI to a new URI:

```
/*
 * These static character arrays are just a convenience for this example.
 */
static char oldURI[] = "/test/bin/confirm";
static char newURI[] = "/test/bin/accept";

char buf[4096];
char *bp = buf;
char *URI;

URI = WRBGetURI(WRBCtx);
if (!strncmp(URI, oldURI, strlen(oldURI))) {
    /* outdated URI--redirect */
    strcpy(buf, newURI);

    /*
     * append query string or POST data to URI
     * because WRBReturnHTTPRedirect() uses GET
     */
    bp = buf + strlen(newURI);
    *bp = '?';
    bp++;
    strcpy(bp, WRBGetContent(WRBCtx));

    WRBReturnHTTPRedirect(WRBCtx, buf, TRUE);
}

return (WRB_DONE);
```

Related Information

See also `WRBCloseHTTPHeader()` and `WRBReturnHTTPError()`.

WRBSetAuthorization()

`WRBSetAuthorization()` sends a challenge to the client browser 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.

WRB Authentication Schemes

These are the possible values for the `nScheme` parameter:

Scheme	**Description**
WRB_AUTH_BASIC_EXIST	Specifies an existing basic authentication scheme defined in the Web Listener configuration.
WRB_AUTH_BASIC_NEW	Specifies an new basic authentication scheme.
WRB_AUTH_DIGEST_EXIST	Specifies an existing digest authentication scheme defined in the Web Listener 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 Web Listener administration... pages.
WRB_AUTH_IP	Specifies IP-based restriction using an existing group of IP addresses defined by the Web Listener administration... pages.

Syntax

```
WRBReturnCode WRBSetAuthorization( void *WRBCtx,
                                   WRBAuthScheme nScheme,
                                   char *szRealm,
                                   boolean bAndOrFlag );
```

Parameters

Parameter	Type	Description
WRBCtx	void *	The pointer to the opaque WRB context object that the WRB application engine passed to your cartridge function.
nScheme	WRBAuthScheme	The type of authentication or restriction you want to set up with the client (see WRB Authentication Schemes).
szRealm	char *	The name of an authentication realm for the client browser to use in prompting the user for a username and password. This parameter must not be NULL if WRBAuthScheme specifies authentication. If nScheme specifies restriction, this parameter must specify an existing group of domains or IP addresses defined in the Web Listener configuration.
bAndOrFlag	boolean	If set to FALSE in both of two successive calls to WRBSetAuthorization(), this parameter specifies that clients must satisfy both specified schemes. If set to TRUE in both calls, this parameter specifies that clients need only satisfy one of the specified schemes.

Return Values

`WRBSetAuthorization()` returns a value of type `WRBReturnCode`.

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, call `WRBSetAuthorization()` from Authorize—if, for example, you want to assign different authentication schemes or realms to different URIs. If you should 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 Web Listener administration... pages.

Similarly, if you specify `WRB_AUTH_IP`, `szRealm` must specify an existing group of IP addresses defined in the Web Listener 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 clients. If you set `bAndOrFlag` to `FALSE` in both calls, the client must satisfy both schemes to be authorized; if you set `bAndOrFlag` to `TRUE`, the client need only satisfy one of the two schemes.

Note Both specified schemes must be existing schemes defined in the Web Listener configuration.

Examples

This example illustrates how to set up authentication for your cartridge to require clients to satisfy an existing authentication scheme *and* an existing restriction scheme that are defined in the Web Listener configuration:

```
WRBReturnCode ret;

ret = WRBSetAuthorization(WRBCtx,
                          WRB_AUTH_BASIC_EXIST,
                          “goodguys”,
                          FALSE);
if (ret != WRB_DONE)
    return (ret);

ret = WRBSetAuthorization(WRBCtx,
                          WRB_AUTH_IP,
                          “goodaddrs”,
                          FALSE);
if (ret != WRB_DONE)
    return (ret);
```

See also the MyWRBApp sample function `MyWRBApp_Authorize()`.

Related Information

See also `WRBGetUserID()`, `WRBGetPassword()`, and `WRBGetClientIP()`.

Related Information

See also `WRBReturnHTTPError()`, `WRBReturnHTTPRedirect()`, `WRBLogMessage()`, and `WRBClientWrite()`.

WRB 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.

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
```

Related Information

See `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)();
   sb4           (*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;
```

CHAPTER

4

The MyWRBApp Sample Cartridge

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

- Setting up and initializing a WRB cartridge
- Cleaning up after a WRB cartridge before terminating
- Sharing data among multiple execution instances (WRBXs) of a cartridge
- Collecting registration data from users
- Authenticating registered users
- Parsing URLs and other HTTP data
- Parsing CGI data
- Maintaining persistent client state data (using cookies)
- Creating HTML documents dynamically and sending them to clients

You can find the source code for MyWRBApp in the files `mywrbapp.h` and `mywrbapp.c` on your WebServer machine in the directory **$ORACLE_HOME/ows21/sample/wrbsdk/mywrbapp**.

Trying out MyWRBApp

Before diving into the code, you might want to play with the MyWRBApp cartridge to see what it does. By default, any new Listener you create using the WebServer Manager is automatically configured to allow you to run the WebServer sample cartridges, including MyWRBApp, so you should be able to point your browser at this URL to use the cartridge:

http://*your-server-machine*:*port*/sample/wrbsdk/mywrbapp

where *your-server-machine:port* is the hostname / port combination that identifies a running Listener on your WebServer machine. If this doesn't work, see Configuring a Web Listener to use your cartridge to learn how to configure your Listener to run MyWRBApp.

Monitoring debugging output

While running the MyWRBApp cartridge, you can monitor its log files to see what the cartridge is doing. Here's an easy way to do this:

1. **cd $ORACLE_HOME/ows21/log**
2. **ls -lt | more**

   This lists the files in the directory in order of modification time, with the most recently modified files at the top. Near the top of the listing you should see a filename of the form **wrb_MYAPP_*proc-id***, where proc-id is the process ID of the WRBX in which the cartridge is running. There may be many files in the directory with names like this—if so, choose the most recently modified file.
3. **more wrb_MYAPP_*proc-id***

   If you repeat this step after every HTTP transaction, you can see the new logging output the cartridge has generated for each transaction. You might want to repeat step 2 occasionally to make sure you're still looking at the right log file.

Programming Tasks Illustrated by the MyWRBApp Sample Cartridge

Setting up and initializing 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.

Cleaning up after 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()`.

Sharing data among multiple execution instances (WRBXs) of a cartridge

Because several copies of a WRB cartridge can run at once (in multiple WRBXs), any data that a cartridge can modify that is shared among WRBXs must be protected from corruption.

MyWRBApp uses a memory-mapped data file to store the usernames and passwords of registered users. All WRBXs of the MyWRBApp cartridge share this data file. To protect the integrity of the data file, MyWRBApp uses the advisory file and record locking facilities of the host operating system. The cartridge encapsulates calls to these facilities in special-purpose locking macros, defined in **MyWRBApp.h**.

Several MyWRBApp functions use these macros to lock sections of the data file for reading or writing. `editUserData()`, for example, uses them to lock a single data block for reading and writing, while `mapdata()` uses them to lock the entire file for reading and writing.

Collecting 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 a memory-mapped data file in the host file system that is shared among WRBXs. The `newUser()`, `getUserData()`, and `editUserData()` functions, and the functions that they call, manipulate this data file.

See also Authenticating registered users and Sharing data among multiple execution instances (WRBXs) of a cartridge.

Authenticating 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.

Parsing 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

Parsing CGI 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 `WRBGetParsedContent()` and `WRBGetNamedEntry()`.

Maintaining persistent client state data (using cookies)

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 *cookies*, data objects that are created by the cartridge, stored in the client browser, and passed back to the cartridge with subsequent requests.

The MyWRBApp cartridge uses cookies to store information about the items that a client orders from a catalog. When a user marks an item on the MyWRBApp catalog form and submits the form, the cartridge sends back to user's browser a cookie of the form *part-number=quantity*, where *part-number* and *quantity* identify an item and the quantity of that item that the user has marked.

The user may then leave and return to the catalog page many times, adding or removing items from the order, and the cookies keep track of the order. When the user finally confirms the order, MyWRBApp signals the browser to delete the cookies its local storage.

The MyWRBApp functions `getCatalog()`, `placeOrder()`, `confirmOrder()`, and the functions they call illustrate how to send cookies to the client, how to get cookies from the Listener, and how to parse the cookies themselves.

Creating HTML documents dynamically and sending 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

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

/* maximum string lengths */
#define MAXNAMELEN 12
#define MAXPASSLEN 32
#define PARTNUMLEN 9

#define MAXITEMS 170        /* max items allowed in orderitems array */
#define TSIZE 128           /* number of bins in hash table--tune to taste */
```

The datablock structure

MyWRBApp uses a `datablock` structure to store authentication data for each registered user. The cartridge stores these data files in a large array in a shared, memory-mapped data file, and uses a hash table also stored in the data file to access the data blocks quickly. The `datablock` structure contains strings of hard-coded length because the cartridge stores data blocks in the data file, which different WRBXs can in principle map into memory at different addresses. The data file, therefore, can contain no pointers, as pointers are process-specific.

Also for this reason, the hash table contains indeces, rather than pointers, into the `datablocks` array. The `nextidx` field of the data block, therefore, contains the index into the `datablocks` array of the next data block in the data block’s hash bin.

```
/*
 * datablock stores authentication data for a user--
 * allocate arrays rather than pointers because the datafile must
 * contain no pointers, as each process can map the file at a
 * different address
 */
typedef struct datablock {
    int nextidx;        /* index into datablocks array of next datablock */
    char username[MAXNAMELEN];
    char password[MAXPASSLEN];  /* encrypted */
    /* more to come? */
} datablock;
```

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 {
    char partnum[PARTNUMLEN + 1];
    char qty[5];
} orderitem;
```

The myappctx structure

MyWRBApp uses the `myappctx` structure primarily to access sections of the memory-mapped data file. `MyWRBApp_Init()` calls `mapdata()` to allocate and set up this structure.

```
/*
 * application context--set up by Init function and subsequently passed
 * to each cartridge function by the WRB application engine
 */
typedef struct myappctx {
    int *table;         /* hash table of indeces into datablocks array */
    int *firstfree;     /* index of first free datablock--currently unused */
    int *datacount;     /* current number of datablocks, including free */
    datablock *datablocks;      /* first datablock struct in large array */
    int datafd;         /* data file descriptor */
    off_t end;          /* offset of current EOF */
    orderitem *items;   /* pointer to ordered items array */
    int numitems;       /* number of orders currently in array */
    struct flock lock;  /* lock structure used in locking the datafile */
    char *state;        /* our home state, read from configuration data */
```

```
        double taxpct;      /* tax in our area, read from configuration data */
    } myappctx;
```

Data file section offsets and sizes

MyWRBApp uses these constants to specify data file sections in calls to the locking macros. For example, to lock the entire data file for reading, the cartridge uses this syntax:

```
RDLOCK(FILE_START, FILE_SIZE, ctx)
```

where ctx is a pointer to the application context structure.

```
/*
 * datafile section offsets
 */
#define FILE_START 0
#define TABLE_OFF FILE_START
#define FIRSTFREE_OFF (TABLE_OFF + (TSIZE * sizeof(int)))
#define DATACOUNT_OFF (FIRSTFREE_OFF + sizeof(int))
#define DATABLOCKS_OFF (DATACOUNT_OFF + sizeof(int))

/*
 * datafile section sizes
 */
#define FILE_SIZE 0 /* FILE_SIZE of zero means “to EOF” in locking calls */
#define TABLE_SIZE (TSIZE * sizeof(int))
#define FIRSTFREE_SIZE sizeof(int)
#define DATACOUNT_SIZE sizeof(int)

/* DATABLOCKS_SIZE of zero means “to EOF” in locking calls */
#define DATABLOCKS_SIZE 0

/* minimum datafile size--tune to taste */
#define MINFILESIZE (DATABLOCKS_OFF + (TSIZE * sizeof(datablock)))

/* DBLOCK_OFF calculates the offset of a datablock from beginning of file */
#define DBLOCK_OFF(data, ctx)   (DATABLOCKS_OFF + \
        (data - ctx->datablocks)*sizeof(datablock))
```

The THASH macro

MyWRBApp uses THASH to assign data blocks to and retrieve them from the hash table in the shared data file. The algorithm is a simple string hash cribbed from Kernighan and Ritchie (*The C Programming Language, Second Edition*; Prentice Hall Software Series, 1988; p. 144).

```
/*
 * hash macro
 */
#define PRIME 157
#define THASH(count, cp)    for (count = 0; *cp; cp++) \
                                count = *cp + PRIME * count; \
                            count %= TSIZE;
```

Locking macros

MyWRBApp uses the locking macros to apply advisory read locks and write locks on sections of the shared data file. The macros are wrappers that provide MyWRBApp with a simplified interface to the operating system advisory file and record locking mechanism.

```
/*
 * locking macros--used to maintain the integrity of the shared datafile
 */
#define LOCK(type, off, len, ctx)   ctx->lock.l_type = type; \
        ctx->lock.l_whence = SEEK_SET; \
        ctx->lock.l_start = off; \
        ctx->lock.l_len = len; \
        fcntl(ctx->datafd, F_SETLKW, &(ctx->lock));

#define RDLOCK(off, len, ctx)   LOCK(F_RDLCK, off, len, ctx)
#define WRLOCK(off, len, ctx)   LOCK(F_WRLCK, off, len, ctx)
#define UNLOCK(ctx) ctx->lock.l_type = F_UNLCK; \
        fcntl(ctx->datafd, F_SETLKW, &(ctx->lock));

#define UPGRADE(ctx)    UNLOCK(ctx) \
        ctx->lock.l_type = F_WRLCK; \
        fcntl(ctx->datafd, F_SETLKW, &(ctx->lock));

#define DOWNGRADE(ctx)  UNLOCK(ctx) \
        ctx->lock.l_type = F_RDLCK; \
        fcntl(ctx->datafd, F_SETLKW, &(ctx->lock));

/*
 * 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, void *WRBCtx);

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

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

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

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

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

WRBReturnCode addUser(const datablock* newdata, myappctx* ctx,
  void *WRBCtx);

WRBReturnCode rehash(const char *oldname, const datablock* newdata,
  myappctx* ctx, void *WRBCtx);

WRBReturnCode formatUserData(const datablock *data, myappctx *ctx,
  void *WRBCtx);

void userDataError(const char *message, const char *URL, myappctx *ctx,
  void *WRBCtx);

WRBReturnCode storeInfo(const char* user, myappctx *ctx, void *WRBCtx);

datablock *getDataBlock(const char *user, myappctx *ctx, void *WRBCtx);

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

boolean taxable(char *user, myappctx *ctx, void *WRBCtx);

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

myappctx *mapdata(const int fd, myappctx *ctx, void *WRBCtx);

WRBReturnCode remapdata(myappctx *ctx, void *WRBCtx);

WRBReturnCode growdata(myappctx *ctx, void *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 opens the data file and calls the `mapdata()` function to map the data file into memory and allocate the application context structure. `MyWRBApp_Init()` then completes initialization of the context structure and passes it back to the WRB application engine when it returns.

```
WRBReturnCode
MyWRBApp_Init(void *WRBCtx, void **appCtx)
{
    int fd;
    myappctx *ctx;
    char datafile[256];
    char *ohome;

    ohome = WRBGetORACLE_HOME(WRBCtx);
    sprintf(datafile, “%s/ows21/sample/wrbsdk/mywrbapp/datafile”, ohome);
    fd = open(datafile, O_CREAT | O_RDWR, 0664);
    if (fd == -1) {
        /* bad news--can’t get to datafile--shut down app */
#ifdef DEBUG
        sprintf(debug, “MyWRBApp_Init: failed to open datafile: %s”,
          strerror(errno));
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        return (WRB_ABORT);
    }

    /* map datafile into memory */
    if (!(ctx = mapdata(fd, NULL, WRBCtx))) {
        return (WRB_ABORT);
    }
    /* allocate order items array */
    ctx->items = (orderitem *)malloc((MAXITEMS + 1) * sizeof(orderitem));
    ctx->numitems = 0;

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

#ifdef DEBUG
    sprintf(debug, “MyWRBApp_Init: context structure fields are:”);
    WRBLogMessage(WRBCtx, debug, 1);
    sprintf(debug, “table: %X\nfirstfree: %X\ndatacount: %X\ndatablocks: %X”,
      ctx->table, ctx->firstfree, ctx->datacount, ctx->datablocks);
    WRBLogMessage(WRBCtx, debug, 1);
    sprintf(debug, “datafd: %d\nend: %u\nitems: %X\nnumitems: %d”,
      ctx->datafd, ctx->end, ctx->items, ctx->numitems);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    /* 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(void *WRBCtx, void *appCtx)
{
    myappctx *ctx = (myappctx *)appCtx;

    WRLOCK(FILE_START, FILE_SIZE, ctx)

    /* sync data file */
    msync((char *)ctx->table, ctx->end, MS_SYNC | MS_INVALIDATE);

    UNLOCK(ctx)

    /* unmap and close data file */
    munmap((char *)ctx->table, ctx->end);
    close(ctx->datafd);

    /* 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 doesn't 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 API function `WRBSetAuthorization()` 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 the WRB API function `WRBGetUserID()` to get the username that the user entered in the client browser's authentication dialog. If `WRBGetUserID()` returns NULL, it's 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 isn't 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 `WRBGetUserID()` will succeed.

```
WRBReturnCode
MyWRBApp_Authorize(void *WRBCtx, void *appCtx, boolean *bAuthorized)
{
    char *URL, *req, *user, *password;
    datablock *data;
    myappctx *ctx = (myappctx *)appCtx;
    int cmp;

    /* get request URL */
    URL = WRBGetURL(WRBCtx);
    URL = strdup(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 = strrchr(URL, ‘/’);
    req++;
    if (!strncmp(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
     */
    WRBSetAuthorization(WRBCtx, WRB_AUTH_BASIC_NEW, “MyWRBApp”, FALSE);

    /* get the username provided by the client */
    user = WRBGetUserID(WRBCtx);
    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: WRBGetUserID failed”);
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        return (WRB_DONE);
    }

    /* copy username to local memory */
    user = strdup(user);

    /* get data block for this user */
    data = getDataBlock(user, ctx, WRBCtx);
    if (!data) {
        /* unknown user - not authorized */
        *bAuthorized = FALSE;
#ifdef DEBUG
        sprintf(debug, “MyWRBApp_Authorize: unknown user %s not authorized”,
          user);
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        return(WRB_DONE);
    }

    /* get the password provided by the client and copy to local memory */
    password = WRBGetPassword(WRBCtx);
    password = strdup(password);

    RDLOCK(DBLOCK_OFF(data, ctx), sizeof(datablock), ctx)

    /* compare stored password with password given */
    cmp = strcmp(password, /* XXX decrypt(*/data->password/*)*/);
    UNLOCK(ctx)

    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(void *WRBCtx, void *appCtx)
{
    char *URL, *req;
    myappctx *ctx = (myappctx *)appCtx;

    /* get request URL */
    URL = WRBGetURL(WRBCtx);
    URL = strdup(URL);

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

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

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

    /* bad URL--send HTTP error */
    WRBReturnHTTPError(WRBCtx, (WRBErrorCode)400, NULL, TRUE);

    return(WRB_DONE);
}
```

MyWRBApp_Reload()

`MyWRBApp_Reload()` uses the WRB API function `WRBGetConfigVal()` to update the application context structure’s copies of the cartridge’s configurable parameters.

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

    /* get configuration parameters */
    ctx->state = WRBGetConfigVal(WRBCtx, “state”);
    tax = WRBGetConfigVal(WRBCtx, “tax”);

    if (tax) {
        /* remove percent sign from tax, if any */
        cp = strchr(tax, ‘%’);
        if (cp) {
            cp = ‘\0’;
        }
        ctx->taxpct = atof(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/ows21/sample/wrbsdk/mywrbapp** directory on your WebServer machine).

newUser() uses WRB API function WRBGetParsedContent() to parse the CGI data from the form. WRBGetParsedContent() 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 WRBGetNamedEntry() to extract the values of specific input fields.

```
WRBReturnCode
newUser(myappctx *ctx, void *WRBCtx)
{
    char *user, *password, *confirm, *cp;
    boolean confirmed;
    datablock newdata;
    WRBReturnCode ret;
    WRBEntry *entries;
    int numentries;
    char wbuf[1024];
    int len;

    /* get username from form data */
    WRBGetParsedContent(WRBCtx, &entries, &numentries);
    user = WRBGetNamedEntry(“username”, entries, numentries);

    /* check whether this name is in use */
    if (getDataBlock(user, ctx, WRBCtx)) {
        /*
         * The given username is already in use--
         * respond with an error page
         */
        userDataError(“The username you entered is already registered to another user. P
lease choose another username.”,
          “/sample/wrbsdk/mywrbapp/newuser.html”, ctx, WRBCtx);
        return (WRB_DONE);
    }

    /* start filling in newdata */
    strcpy(newdata.username, user);

    /* get password from form data */
    password = WRBGetNamedEntry(“password”, entries, numentries);

    /* get password confirmation and compare with password */
    confirm = WRBGetNamedEntry(“confirm”, entries, numentries);
    if (!strcmp(password, confirm)) {
        /* password and confirmation string match--fill in password */
        confirmed = TRUE;
        strcpy(newdata.password, password);     /* XXX encrypt? */
    }
    else {
        /* they don’t match--null out password for now */
        confirmed = FALSE;
        newdata.password[0] = ‘\0’;
    }

    /* add this user to the datafile */
    ret = addUser(&newdata, ctx, WRBCtx);
    if (ret != WRB_DONE) {
        return (ret);
    }

    ret = storeInfo(user, ctx, WRBCtx);
    if (ret != WRB_DONE) {
        return (ret);
    }

    if (!confirmed) {
        /*
         * confirmation doesn’t match password--generate error page
         */
        userDataError(“The password you entered does not match your confirmation string.
 Please reenter your password”,
          “/mywrbapp/bin/getuserdata”, ctx, WRBCtx);

        return (WRB_DONE);
    }

    /* done--go to catalog */
    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, void *WRBCtx)
{
    char *user;
    datablock *data;
    WRBReturnCode ret;

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

    /* get datablock for this user */
    data = getDataBlock(user, ctx, WRBCtx);
    if (!data) {
        /*
         * this shouldn't happen--if the user is authenticated, there
         * should be a datablock
         */
        return (WRB_ERROR);
    }

    RDLOCK(DBLOCK_OFF(data, ctx), sizeof(datablock), ctx)

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

    UNLOCK(ctx)

    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, void *WRBCtx)
{
    char *user, *password, *confirm;
    boolean confirmed, nameInUse = FALSE;
    char wbuf[1024];
    int len;
    datablock newdata;
    datablock *data;
    WRBEntry *entries;
    int numentries;

    /* get data from form */
    WRBGetParsedContent(WRBCtx, &entries, &numentries);
    user = WRBGetNamedEntry(“username”, entries, numentries);
    password = WRBGetNamedEntry(“password”, entries, numentries);

    /* get password confirmation and compare with password */
    confirm = WRBGetNamedEntry(“confirm”, entries, numentries);
    if (!strcmp(password, confirm)) {
        /* password and confirmation string match */
        confirmed = TRUE;
    }
    else {
        confirmed = FALSE;
    }

    /* look up currently authenticated user */
    data = getDataBlock(WRBGetUserID(WRBCtx), ctx, WRBCtx);

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

    RDLOCK(DBLOCK_OFF(data, ctx), sizeof(datablock), ctx)

    if (strcmp(user, data->username)) {
        /* new usernamee */
        if (!getDataBlock(user, ctx, WRBCtx)) {
            /* new username not in use--copy it to newdata */
            nameInUse = FALSE;
            strcpy(newdata.username, user);
        }
        else {
            /* new name is not in use--leave username unchanged */
            nameInUse = TRUE;
            strcpy(newdata.username, data->username);
        }

#ifdef DEBUG
        sprintf(debug, “editUserData: old username: %s\n\tnew username %s”,
          data->username, user);
        WRBLogMessage(WRBCtx, debug, 1);
#endif
```

```
        if (confirmed) {
            strcpy(newdata.password, password);     /* XXX encrypt? */
        }
        else {
            /* confirmation failed--leave password unchanged */
            strcpy(newdata.password, data->password);
        }

        if (rehash(data->username, &newdata, ctx, WRBCtx) != WRB_DONE) {
            UNLOCK(ctx)

            return (WRB_ERROR);
        }
    }
    else {
        UPGRADE(ctx)

        /* change password only if confirmation matched password */
        if (confirmed) {
            strcpy(data->password, password);       /* XXX encrypt? */
        }

        /* sync datafile */
        msync((char *)ctx->table, ctx->end, MS_SYNC | MS_INVALIDATE);

        DOWNGRADE(ctx)
    }

    /* store the rest of the form data */
    storeInfo(data->username, ctx, WRBCtx);

    UNLOCK(ctx)

    if (nameInUse) {
        /* specified username is already in use--generate error page */
        userDataError(“The username you entered is already registered to another user. P
lease choose another username.”,
          “/mywrbapp/bin/getuserdata”, ctx, WRBCtx);

        return (WRB_DONE);
    }

    if (!confirmed) {
        /* confirmation doesn’t match password--generate error page */
        userDataError(“The password you entered does not match your confirmation string.
 Please reenter your password”,
          “/mywrbapp/bin/getuserdata”, ctx, WRBCtx);

        return (WRB_DONE);
    }

    /* done--go to catalog */
    return(getCatalog(ctx, WRBCtx));
}
```

getCatalog()

`MyWRBApp_Exec()` calls `getCatalog()` to handle “getcatalog” requests. It’s 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.

To show any pending order data associated with the client, `getCatalog()` calls `WRBGetEnvironmentVariable()` to extract the client’s cookie data from the Web Listener environment. For each catalog item, `getCatalog()` then searches the cookie string for that item’s part number; if it finds the item in the cookie, 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, void *WRBCtx)
{
    FILE *cat;
    char buf[1024];
    char wbuf[1024];
    int len;
    char *cp, *tp;
    char part[PARTNUMLEN + 1];
    char qty[5] = “\0”;
    char *qp;
    char catfile[256];
    char *ohome;

    /* open catalog */
    ohome = WRBGetORACLE_HOME(WRBCtx);
    sprintf(catfile, “%s/ows21/sample/wrbsdk/mywrbapp/catalog.txt”, ohome);
    cat = fopen(catfile, “r”);
    if (!cat) {
#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 */
    WRBClientWrite(WRBCtx, “Content-type: text/html\n\n”, 25);

    len = sprintf(wbuf,
      “<HTML><HEAD><TITLE>Hot Goods Catalog</TITLE></HEAD>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “<BODY BGCOLOR=\”FFFFFF\”>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);
    len = sprintf(wbuf, “<H1>Hot Goods Catalog</H1>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “<FORM ACTION=\”/mywrbapp/bin/placeorder\” METHOD=\”POST\”>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    /* get cookie from the client */
    cp = WRBGetEnvironmentVariable(WRBCtx, “HTTP_COOKIE”);
#ifdef DEBUG
    if (!cp) {
        WRBLogMessage(WRBCtx, “getCatalog: no cookie found”, 0);
    }
#endif

    /*
     * Loop through the catalog generating HTML for each line--
     * for each part number listed in the catalog, check the
     * cookie to see if the user has already entered an order quantity
     * for the item
     */
    while (fgets(buf, 1024, cat)) {
        /* write the description line to the client */
        WRBClientWrite(WRBCtx, “<P>”, 3);
        WRBClientWrite(WRBCtx, buf, strlen(buf));
        WRBClientWrite(WRBCtx, “</P>\n<P>”, 8);

        /* get part number line and write it out without the newline */
        fgets(buf, 1024, cat);
        WRBClientWrite(WRBCtx, buf, strlen(buf) - 1);

        /* the part number is the first PARTNUMLEN chars of buf */
        strncpy(part, buf, PARTNUMLEN);
        part[PARTNUMLEN] = ‘\0’;

        if (tp = cp) {
            /* search through cookie for part number */
            qp = qty;
            while (tp) {
                if (!strncmp(tp, part, PARTNUMLEN)) {
                    /* found part number--now get quantity */
                    tp = strchr(tp, ‘=’);
                    tp++;
                    while (*tp && (*tp != ‘;’)) {
                        *qp++ = *tp++;
                    }
                    break;
                }
                tp = strchr(tp, ‘;’);
                if (tp) {
                    tp += 2;
                }
            }
            *qp = ‘\0’;
        }
        /*
         * complete the HTML line with an input box and set its value to
         * the quantity stored in the cookie (if any)
         */
        len = sprintf(wbuf,
          " Order quantity: <INPUT TYPE=\"text\" NAME=\"%s\" ", part);
        WRBClientWrite(WRBCtx, wbuf, len);

        len = sprintf(wbuf, "SIZE=\"4\" VALUE=\"%s\"></P>\n", qty);
        WRBClientWrite(WRBCtx, wbuf, len);
    }

    /* finish HTML page */
    len = sprintf(wbuf, "<INPUT TYPE=\"submit\" NAME=\"submit\" ");
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, "VALUE=\"Place Order\">\n");
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, "</FORM>\n</BODY>\n</HTML>");
    WRBClientWrite(WRBCtx, wbuf, len);

    fclose(cat);
    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, void *WRBCtx)
{
    WRBReturnCode ret;
    WRBEntry *entries;
    int numentries, item, ent;
    char wbuf[1024];
    int len;
    char *cp, *tp;
    char *qp;
    orderitem *items;

    item = 0;
    items = ctx->items;

    /* get cookie from the client */
    cp = WRBGetEnvironmentVariable(WRBCtx, “HTTP_COOKIE”);
    if (cp) {
        /* put cookie name-value pairs into items array */
        tp = cp;
        while (item < MAXITEMS) {
            /*
             * include only names that start with a part number
             */
            if (isdigit(*tp)) {
                strncpy(items[item].partnum, tp, PARTNUMLEN);
                items[item].partnum[PARTNUMLEN] = ‘\0’;

                /* the value is the order quantity */
                tp = strchr(tp, ‘=’);
                tp++;
                qp = items[item].qty;
                while (*tp && (*tp != ‘;’)) {
                    *qp++ = *tp++;
                }
                *qp = ‘\0’;
                item++;
            }
            else {
                /* not a numeric name--skip this name-value pair */
                tp = strchr(tp, ‘;’);
            }
            if (!tp || (*tp == ‘\0’)) {
                /* reached the end of the cookie */
                break;
            }
            tp += 2;
        }
    }
#ifdef DEBUG
    else {
        WRBLogMessage(WRBCtx, “placeOrder: no cookie found”, 0);
    }
#endif

    /* save number of items found in cookie */
    ctx->numitems = item;
#ifdef DEBUG
    sprintf(debug, “placeOrder: numitems: %d”, ctx->numitems);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    /* get form data */
    WRBGetParsedContent(WRBCtx, &entries, &numentries);
    for (ent = 0; ent < numentries; ent++) {
        /*
         * for each name-value pair from the form, search through
         * part numbers found in the cookie for a duplicate--
         * if found, replace the cookie data with the form data
         */
        for (item = 0; item < ctx->numitems; item++) {
            if (!strcmp(entries[ent].name, items[item].partnum)) {
                /* found a duplicate */
                strcpy(items[item].partnum, entries[ent].name);
                strcpy(items[item].qty, entries[ent].value);
                entries[ent].name[0] = ‘\0’;
                entries[ent].value[0] = ‘\0’;
                break;
            }
        }
    }

    /* transfer any remaining form data to items array */
    item = ctx->numitems; /* take up where we left off */
    for (ent = 0; (ent < numentries) && (item < MAXITEMS); ent++) {
        if (isdigit(entries[ent].name[0]) && entries[ent].value[0] &&
          strcmp(entries[ent].value, “0”)) {
            /* the name is a part number with a non-zero value (quantity) */
            strcpy(items[item].partnum, entries[ent].name);
            strcpy(items[item].qty, entries[ent].value);
            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 */
    WRBClientWrite(WRBCtx, “Content-type: text/html\n”, 24);
    item = 0;

    /* write cookie data */
    while (item < ctx->numitems) {
        if (items[item].qty[0] && strcmp(items[item].qty, “0”)) {
            /* item has non-zero quantity--put it in the cookie */
            len = sprintf(wbuf, “Set-Cookie: %s=%s; path=/\n”,
              items[item].partnum, items[item].qty);
            WRBClientWrite(WRBCtx, wbuf, len);
        }
        else {
            /*
             * the item has a zero quantity--delete it from the cookie
             * by setting past expiration date
             */
            len = sprintf(wbuf,
              “Set-Cookie: %s=0; path=/; “, items[item].partnum);
            WRBClientWrite(WRBCtx, wbuf, len);

            len = sprintf(wbuf,
              “expires=Monday, 01-Jan-96 00:00:00 GMT\n”);
            WRBClientWrite(WRBCtx, wbuf, len);
        }
        item++;
    }

    /* remember the empty line before “<HTML>” */
    WRBClientWrite(WRBCtx, “\n”, 1);

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

    len = sprintf(wbuf, “<HTML><HEAD><TITLE>Order Form</TITLE></HEAD>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “<BODY BGCOLOR=\”FFFFFF\”>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “<H1>Placing an Order</H1>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “<P>These are the items and the quantities you have selected. “);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “To change the quantity of an item you want to order, “);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “just change the value in the quantity box for that item. “);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “If you decide you don’t want to order an item after all, “);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “just delete the contents of the item’s quantity box. “);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “To add more items to your order, you can “);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “<A HREF=\”/mywrbapp/bin/getcatalog\”>return to the catalog</A>.</P>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);
```

```
    len = sprintf(wbuf, “<FORM ACTION=\”/mywrbapp/bin/confirm\”>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

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

    len = sprintf(wbuf, “<INPUT TYPE=\”submit\” VALUE=\”Confirm Order\”>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “</FORM>\n</BODY>\n</HTML>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    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.

Like `getCatalog()`, `confirmOrder()` calls `WRBGetEnvironmentVariable()` to extract the cookie data associated with the client from the Web Listener environment.

```
WRBReturnCode
confirmOrder(myappctx *ctx, void *WRBCtx)
{
    WRBReturnCode ret;
    WRBEntry *entries;
    int numentries, item, ent;
    char wbuf[1024];
    int len;
    char *cp, *tp;
    char part[PARTNUMLEN + 1];
    orderitem *items;

    /* begin response */
    WRBClientWrite(WRBCtx, “Content-type: text/html\n”, 24);

    /* get cookie from the client */
    cp = WRBGetEnvironmentVariable(WRBCtx, “HTTP_COOKIE”);
    if (cp) {
        /*
         * the order is now placed--delete cookie entries by setting
         * past expiration date
         */
        tp = cp;
        while (tp && *tp) {
            if (isdigit(*tp)) {
                /* this name-value pair is an order item--delete it */
                strncpy(part, tp, PARTNUMLEN);
                part[PARTNUMLEN] = ‘\0’;

                len = sprintf(wbuf, “Set-Cookie: %s=0; path=/; “, part);
                WRBClientWrite(WRBCtx, wbuf, len);

                len = sprintf(wbuf, “expires=Monday, 01-Jan-96 00:00:00 GMT\n”);
                WRBClientWrite(WRBCtx, wbuf, len);
            }

            /* find the next name-value pair */
            tp = strchr(tp, ‘;’);
            if (!tp) {
                /* reached the end of the cookie */
                break;
            }
            tp += 2;
        }
    }

    /* remember the blank line */
    WRBClientWrite(WRBCtx, “\n”, 1);

    len = sprintf(wbuf, “<HTML><HEAD><TITLE>Thank You</TITLE></HEAD>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);
    len = sprintf(wbuf, “<BODY BGCOLOR=\”FFFFFF\”>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);
    len = sprintf(wbuf, “<H1>Thank you for your order</H1>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);
    len = sprintf(wbuf,
      “<P>These are the items and the quantities you have ordered.</P>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    /* get form data and fill in ordered items array */
    WRBGetParsedContent(WRBCtx, &entries, &numentries);
    items = ctx->items;
    item = 0;
    for (ent = 0; (ent < numentries) && (item < MAXITEMS); ent++) {
        if (isdigit(entries[ent].name[0]) && entries[ent].value[0]) {
            strcpy(items[item].partnum, entries[ent].name);
            strcpy(items[item].qty, entries[ent].value);
            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) {
        return (WRB_ERROR);
    }

    len = sprintf(wbuf,
      "<P><A HREF=\"/mywrbapp/bin/getcatalog\">Return to the catalog");
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, "</A>.</P>\n</BODY>\n</HTML>\n");
    WRBClientWrite(WRBCtx, wbuf, len);

    return (WRB_DONE);
}
```

addUser()

`newUser()` calls `addUser()` to allocate a new data block in the data file for the user specified in the data block pointed to by `newdata`.

```
WRBReturnCode
addUser(const datablock* newdata, myappctx* ctx, void *WRBCtx)
{
    unsigned hashval, idx;
    char *cp;
    datablock *data;

    WRLOCK(FILE_START, FILE_SIZE, ctx)

    if (*(ctx->firstfree)) {
        /* there's an unused datablock--take it off the free list */
        idx = *(ctx->firstfree);
        data = ctx->datablocks + idx;
        *(ctx->firstfree) = data->nextidx;
#ifdef DEBUG
        sprintf(debug, "addUser: taking block of free list, index %u", idx);
        WRBLogMessage(WRBCtx, debug, 1);
        sprintf(debug, "firstfree is now %u", *(ctx->firstfree));
        WRBLogMessage(WRBCtx, debug, 1);
#endif
    }
    else {
        /*
         * add a new datablock to the end of the datablocks array--
         * pre-inc datacount to assure no datablock has index 0--
         * index 0 means "end of linked list"
         */
        (*(ctx->datacount))++;

        if (DATABLOCKS_OFF + (*(ctx->datacount) * sizeof(datablock))
          > ctx->end) {
            /* datafile is full--extend it */

            UNLOCK(ctx)
            if (growdata(ctx, WRBCtx) != WRB_DONE) {
                return (WRB_ABORT);
            }

            WRLOCK(FILE_START, FILE_SIZE, ctx)
        }

        /*
         * calculate address of the new datablock by adding the
         * new datacount to the base address of the array
         */
        idx = *(ctx->datacount);
        data = ctx->datablocks + idx;
#ifdef DEBUG
        sprintf(debug, “addUser: adding new datablock for %s, index %u”,
          newdata->username, idx);
        WRBLogMessage(WRBCtx, debug, 1);
#endif
    }

    /* fill in the new datablock from the data passed in */
    strcpy(data->username, newdata->username);
    strcpy(data->password, newdata->password);

    /* hash on username--add new datablock to hash table */
    cp = data->username;
    THASH(hashval, cp)
#ifdef DEBUG
    sprintf(debug, “addUser: hashval for %s is %d”, data->username, hashval);
    WRBLogMessage(WRBCtx, debug, 1);
#endif
    data->nextidx = ctx->table[hashval];
    ctx->table[hashval] = idx;
#ifdef DEBUG
    sprintf(debug, “addUser: putting new datablock for %s in bin %d”,
      data->username, hashval);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    /* sync datafile */
    msync((char *)ctx->table, ctx->end, MS_SYNC | MS_INVALIDATE);

    UNLOCK(ctx)

    return (WRB_DONE);
}
```

rehash()

`editUserData()` calls `rehash()` when a user changes his or her username. (Because data blocks are hashed by username, a change of username requires the data block to be rehashed and placed in a different hash bin.)

```
WRBReturnCode rehash(const char *oldname, const datablock* newdata,
  myappctx* ctx, void *WRBCtx)
{
    unsigned idx, hashval;
    datablock *data, *lastdata;
    char *cp;
    off_t size;

    WRLOCK(FILE_START, FILE_SIZE, ctx)

    /*
     * check to see if another WRBX has extended the datafile--
     * if so, remap the file
     */
    size = lseek(ctx->datafd, 0, SEEK_END);
    if (size > ctx->end) {
        /* file has grown--remap it */

        UNLOCK(ctx)

        if (remapdata(ctx, WRBCtx) != WRB_DONE) {
            return (WRB_ABORT);
        }

        WRLOCK(FILE_START, FILE_SIZE, ctx)
    }

    /* hash on oldname--retrieve datablock from hash table */
    cp = strdup(oldname);
    THASH(hashval, cp)
    lastdata = NULL;
    idx = ctx->table[hashval];
    data = ctx->datablocks + idx;
    while (data != ctx->datablocks) {
        /*
         * end of list not yet reached--compare datablock username
         * with oldname--break out if match
         */
        if (!strcmp(oldname, data->username)) {
            break;
        }

        lastdata = data;
        idx = data->nextidx;
        data = ctx->datablocks + idx;
    }

    if (data == ctx->datablocks) {
        /* end of hash list reached--no datablock found for given username */
        return (WRB_ERROR);
    }

    /* unlink data block */
    if (lastdata) {
        lastdata->nextidx = data->nextidx;
    }
    else {
        ctx->table[hashval] = data->nextidx;
    }

    /* copy new data into datablock */
    strcpy(data->username, newdata->username);
    strcpy(data->password, newdata->password);

    /* hash on username--relocate datablock in hash table */
    cp = data->username;
    THASH(hashval, cp)
#ifdef DEBUG
    sprintf(debug, “rehash: hashval for %s is %d”, data->username, hashval);
    WRBLogMessage(WRBCtx, debug, 1);
#endif
    data->nextidx = ctx->table[hashval];
    ctx->table[hashval] = idx;
#ifdef DEBUG
    sprintf(debug, “rehash: putting new datablock for %s in bin %d”,
      data->username, hashval);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    /* sync datafile */
    msync((char *)ctx->table, ctx->end, MS_SYNC | MS_INVALIDATE);

    UNLOCK(ctx)

    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(const datablock *data, myappctx *ctx, void *WRBCtx)
    {
        char buf[1024];
        char wbuf[1024];
        char *bp, *ep;
        FILE *info;
        int len;
        char infofile[256];
        char *ohome;

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

        WRBClientWrite(WRBCtx, “Content-type: text/html\n\n”, 25);

        len = sprintf(wbuf,
          “<HTML>\n<HEAD><TITLE>Registration Form</TITLE></HEAD>\n”);
        WRBClientWrite(WRBCtx, wbuf, len);

        len = sprintf(wbuf, “<BODY BGCOLOR=\”FFFFFF\”>\n”);
        WRBClientWrite(WRBCtx, wbuf, len);

        len = sprintf(wbuf,
          “<FORM ACTION=\”/mywrbapp/bin/edituserdata\” METHOD=\”POST\”>\n”);
        WRBClientWrite(WRBCtx, wbuf, len);

        /* username */
        len = sprintf(wbuf, “<P>Username: <INPUT TYPE=\”text\” NAME=\”username\” “);
        WRBClientWrite(WRBCtx, wbuf, len);

        len = sprintf(wbuf, “VALUE=\”%s\” MAXLENGTH=\”10\” SIZE=\”10\”></P>\n”,
          data->username);
        WRBClientWrite(WRBCtx, wbuf, len);

        /* password */
        len = sprintf(wbuf,
          “<P>Password: <INPUT TYPE=\”password\” NAME=\”password\” “);
        WRBClientWrite(WRBCtx, wbuf, len);

        len = sprintf(wbuf, “VALUE=\”%s\” MAXLENGTH=\”10\” SIZE=\”10\”>\n”,
          data->password);
        WRBClientWrite(WRBCtx, wbuf, len);

        /* confirm password */
        len = sprintf(wbuf,
          “ Confirm Password: <INPUT TYPE=\”password\” NAME=\”confirm\” “);
        WRBClientWrite(WRBCtx, wbuf, len);

        len = sprintf(wbuf, “VALUE=\”%s\” MAXLENGTH=\”10\” SIZE=\”10\”></P>\n”,
          data->password);
        WRBClientWrite(WRBCtx, wbuf, len);

    #ifdef DEBUG
        WRBLogMessage(WRBCtx, “formatUserData: opening infofile”, 0);
    #endif

        /* open infofile and write its data to the client */
```

```
ohome = WRBGetORACLE_HOME(WRBCtx);
sprintf(infofile, “%s/ows21/sample/wrbsdk/mywrbapp/%s.info”, ohome, data->username);
info = fopen(infofile, “r”);

/* fullname */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>Full Name: <INPUT TYPE=\”text\” NAME=\”fullname\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”%s\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

/* address */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>Street Address: <INPUT TYPE=\”text\” NAME=\”address\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”%s\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

/* city */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>City: <INPUT TYPE=\”text\” NAME=\”city\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”%s\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

/* state */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>State: <INPUT TYPE=\”text\” NAME=\”state\” “);
```

```
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”%s\” MAXLENGTH=\”2\” SIZE=\”2\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

/* zip */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>Zip: <INPUT TYPE=\”text\” NAME=\”zip\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”%s\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

/* country */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>Country: <INPUT TYPE=\”text\” NAME=\”country\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”%s\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

/* email */
fgets(buf, 1024, info);
bp = strchr(buf, ‘:’);
bp += 2;
ep = strchr(bp, ‘\n’);
*ep = ‘\0’;

len = sprintf(wbuf,
  “<P>Email Address: <INPUT TYPE=\”text\” NAME=\”email\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “ SIZE=\”40\” VALUE=\”%s\”></P>\n”, bp);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “<INPUT TYPE=\”submit\” NAME=\”submit\” “);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf, “VALUE=\”Save Changes\”>\n</FORM>\n”);
WRBClientWrite(WRBCtx, wbuf, len);

len = sprintf(wbuf,
  “<P><A HREF=\”/mywrbapp/bin/getcatalog\”>Go to the catalog</A></P>\n”);
```

```
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “</BODY>\n</HTML>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

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

    fclose(info);

    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 char *message, const char *URL, myappctx* ctx, void *WRBCtx)
{
    char wbuf[1024];
    int len;

    WRBClientWrite(WRBCtx, “Content-type: text/html\n\n”, 25);

    len = sprintf(wbuf, “<HTML><HEAD><TITLE>Error</TITLE></HEAD>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “<BODY BGCOLOR=\”FFFFFF\”>\n<H1>Error</H1>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “<P>%s</P>\n”, message);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf,
      “<P><A HREF=\”%s\”>Return “, URL);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “to the registration form</A>.</P>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    len = sprintf(wbuf, “</BODY>\n</HTML>\n”);
    WRBClientWrite(WRBCtx, wbuf, len);

    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(const char *user, myappctx *ctx, void *WRBCtx)
{
    int fd;
    char buf[256];
    char *cp;
    WRBEntry *entries;
    int numentries;
    char infofile[256];
    char *ohome;

    /* create infofile for this user */
    ohome = WRBGetORACLE_HOME(WRBCtx);
    sprintf(infofile, “%s/ows21/sample/wrbsdk/mywrbapp/%s.info”, ohome, user);
    fd = open(infofile, O_CREAT | O_TRUNC | O_RDWR, 0664);
    if (fd == -1) {
        return (WRB_ABORT);
    }

    /* get form data */
    WRBGetParsedContent(WRBCtx, &entries, &numentries);

    cp = WRBGetNamedEntry(“fullname”, entries, numentries);
    sprintf(buf, “Full Name: %s\n”, cp);
    write(fd, buf, strlen(buf));

    cp = WRBGetNamedEntry(“address”, entries, numentries);
    sprintf(buf, “Street Address: %s\n”, cp);
    write(fd, buf, strlen(buf));

    cp = WRBGetNamedEntry(“city”, entries, numentries);
    sprintf(buf, “City: %s\n”, cp);
    write(fd, buf, strlen(buf));

    cp = WRBGetNamedEntry(“state”, entries, numentries);
    sprintf(buf, “State: %s\n”, cp);
    write(fd, buf, strlen(buf));

    cp = WRBGetNamedEntry(“zip”, entries, numentries);
    sprintf(buf, “Zip Code: %s\n”, cp);
    write(fd, buf, strlen(buf));

    cp = WRBGetNamedEntry(“country”, entries, numentries);
    sprintf(buf, “Country: %s\n”, cp);
    write(fd, buf, strlen(buf));

    cp = WRBGetNamedEntry(“email”, entries, numentries);
    sprintf(buf, “Email Address: %s\n”, cp);
    write(fd, buf, strlen(buf));

    close(fd);
    return (WRB_DONE);
}
```

getDataBlock()

`getDataBlock()` retrieves the data block for the specified user from the data file and returns it to the caller.

```
datablock *
getDataBlock(const char *user, myappctx *ctx, void *WRBCtx)
{

    unsigned hashval;
    char *cp;
    datablock *data = NULL;
    off_t size;

    RDLOCK(FILE_START, FILE_SIZE, ctx)

    /*
     * check to see if another WRBX has extended the datafile--
     * if so, remap the file
     */
    size = lseek(ctx->datafd, 0, SEEK_END);
    if (size > ctx->end) {
        /* file has grown--remap it */

        UNLOCK(ctx)

        if (remapdata(ctx, WRBCtx) != WRB_DONE) {
            return (NULL);
        }

        RDLOCK(FILE_START, FILE_SIZE, ctx)
    }

    /* hash on username--retrieve datablock from hash table */
    cp = strdup(user);
    THASH(hashval, cp)
    data = ctx->datablocks + ctx->table[hashval];
    while (data != ctx->datablocks) {
        /*
         * end of list not yet reached--compare datablock username
         * with given value--break out if match
         */
        if (!strcmp(user, data->username)) {
            break;
        }
        data = ctx->datablocks + data->nextidx;
    }

    UNLOCK(ctx)

    if (data == ctx->datablocks) {
        /* end of hash list reached--no datablock found for given username */
        return (NULL);
    }

#ifdef DEBUG
    sprintf(debug, “getDataBlock: data block found for user %s”,
      data->username);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    return (data);
}
```

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)
{
    FILE *cat;
    char lbuf[1024];
    char *lp;
    char wbuf[1024];
    int len;
    orderitem* items;
    int item = 0;
    double itemprice;
    double quantity;
    double total = 0;
    char catfile[256];
    char *ohome;

    /* open catalog */
    ohome = WRBGetORACLE_HOME(WRBCtx);
    sprintf(catfile, “%s/ows21/sample/wrbsdk/mywrbapp/catalog.txt”, ohome);
    cat = fopen(catfile, “r”);
    if (!cat) {
#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 */
        fgets(lbuf, 1024, cat);
        if (!fgets(lbuf, 1024, cat)) {
            break;
        }
        if (!strncmp(lbuf, items[item].partnum, PARTNUMLEN)) {
            /*
             * found the catalog entry for current item--
             * write it out to the client
             */
            WRBClientWrite(WRBCtx, “<P>”, 3);
            WRBClientWrite(WRBCtx, lbuf, strlen(lbuf));

            if (invoice) {
                /* calculate and print price, add to subtotal */
                lp = strchr(lbuf, ‘$’);
                lp++;
                itemprice = atof(lp);
                quantity = atof(items[item].qty);
                len = sprintf(wbuf, “ Quantity: %s @ $%.2f = $%.2f</P>\n”,
                  items[item].qty, itemprice, itemprice * quantity);
                WRBClientWrite(WRBCtx, wbuf, len);

                total += itemprice * quantity;
            }
            else {
                /*
                 * not an invoice--write quantity out as an editable
                 * input field
                 */
                len = sprintf(wbuf, “ Order quantity: <INPUT TYPE=\”text\” “);
                WRBClientWrite(WRBCtx, wbuf, len);

                len = sprintf(wbuf, “NAME=\”%s\” “, items[item].partnum);
                WRBClientWrite(WRBCtx, wbuf, len);

                len = sprintf(wbuf,
                  “SIZE=\”4\” VALUE=\”%s\”></P>\n”, items[item].qty);
```

```
                    WRBClientWrite(WRBCtx, wbuf, len);
                }
                item++;
            }
        }

        if (invoice) {
            if (taxable(WRBGetUserID(WRBCtx), ctx, WRBCtx)) {
                /* print subtotal */
                len = sprintf(wbuf, "<P>Subtotal: $%.2f</P>\n", total);
                WRBClientWrite(WRBCtx, wbuf, len);

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

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

        fclose(cat);

        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(char *user, myappctx *ctx, void *WRBCtx)
    {
        FILE *info;
        char *ohome;
        char buf[1024];
        char *cp = NULL;

    #ifdef DEBUG
        WRBLogMessage(WRBCtx, "taxable: entering", 0);
    #endif
        /* open user's info file */
        ohome = WRBGetORACLE_HOME(WRBCtx);
        sprintf(buf, “%s/ows21/sample/wrbsdk/mywrbapp/%s.info”, ohome, user);
        info = fopen(buf, “r”);

        /* get user’s home state */
        while (fgets(buf, 1024, info)) {
            if (!strncmp(buf, “State:”, 6)) {
                cp = strchr(buf, ‘:’);
                cp += 2;
                break;
            }
        }

        /* compare it with our home state */
        if (cp && ctx->state && !strncmp(cp, ctx->state, 2)) {
            /* 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(((orderitem *)item1)->partnum,
      ((orderitem *)item2)->partnum));
}
```

mapdata()

MyWRBApp_Init() and remapdata() call mapdata() to map into memory the data file specified by fd, and if ctx is NULL, to allocate an application context structure for the calling WRBX.

```
myappctx *
mapdata(const int fd, myappctx *ctx, void *WRBCtx)
{
    off_t size;
    if (!ctx) {
        /* NULL context means allocate context */
        if (!(ctx = (myappctx *)malloc(sizeof(myappctx)))) {
#ifdef DEBUG
            WRBLogMessage(WRBCtx, “mapdata: context malloc failed”, 0);
#endif
            return (NULL);
        }
    }

    RDLOCK(FILE_START, FILE_SIZE, ctx)

    /* fd refers to the open datafile--find its size */
    size = lseek(fd, 0, SEEK_END);
#ifdef DEBUG
    sprintf(debug, “mapdata: datafile size is %u”, size);
    WRBLogMessage(WRBCtx, debug, 1);
#endif

    if (size < MINFILESIZE) {
        /* data file too small--it was probably just created */

        UPGRADE(ctx)

        /* extend file */
        size = lseek(fd, MINFILESIZE - 1, SEEK_END);
        if (write(fd, “\0”, 1) != 1) {
#ifdef DEBUG
            sprintf(debug, “mapdata: failed to extend file:”);
            WRBLogMessage(WRBCtx, debug, 1);
            sprintf(debug, “\t%s”, strerror(errno));
            WRBLogMessage(WRBCtx, debug, 1);
#endif
            goto error;
        }

        size = lseek(fd, 0, SEEK_END);
#ifdef DEBUG
        sprintf(debug, “mapdata: extended datafile to size: %u”, size);
        WRBLogMessage(WRBCtx, debug, 1);
#endif

        DOWNGRADE(ctx)
    }

    /* map entire file into memory */
    ctx->table = (int *)mmap(NULL, size, PROT_READ | PROT_WRITE,
      MAP_SHARED, fd, (off_t)0);
    if (ctx->table == MAP_FAILED) {
#ifdef DEBUG
        sprintf(debug, “mapdata: failed to mmap datafile:”);
        WRBLogMessage(WRBCtx, debug, 1);
        sprintf(debug, “%s”, strerror(errno));
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        goto error;
    }

    /* set up rest of context--defines data file format */
    ctx->firstfree = (int *)(ctx->table + TSIZE);
    ctx->datacount = (int *)(ctx->firstfree + 1);
    ctx->datablocks = (datablock *)(ctx->datacount + 1);
    ctx->datafd = fd;
    ctx->end = size;

    UNLOCK(ctx)

    return (ctx);

error:
    UNLOCK(ctx)

    return (NULL);
}
```

remapdata()

`growdata()` and `rehash()` call `remapdata()` to unmap the data file and remap it into memory. This is necessary when the datafile grows to make room for more data blocks.

```
WRBReturnCode
remapdata(myappctx *ctx, void *WRBCtx)
{

    /* unmap data file */
    if (munmap((char *)ctx->table, ctx->end) == -1) {
        return (WRB_ABORT);
    }

    /* remap data file */
    if (!mapdata(ctx->datafd, ctx, WRBCtx)) {
        return (WRB_ABORT);
    }

    return (WRB_DONE);
}
```

growdata()

`addUser()` calls growdata() to extend the data file when the data file is full. It calls `remapdata()` to unmap and remap the data file into memory.

```
WRBReturnCode
```

```
growdata(myappctx *ctx, void *WRBCtx)
{
    off_t size;

    WRLOCK(FILE_START, FILE_SIZE, ctx)

    /* extend file by seeking past the end and writing a byte */
    size = lseek(ctx->datafd, MINFILESIZE + 1, SEEK_END);
#ifdef DEBUG
    sprintf(debug, “growdata: extended datafile to size: %u”, size);
    WRBLogMessage(WRBCtx, debug, 1);
#endif
    if (write(ctx->datafd, “\0”, 1) != 1) {
#ifdef DEBUG
        sprintf(debug, “growdata: failed to extend datafile: %s”,
          strerror(errno));
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        goto error;
    }

    /* sync datafile */
    if (msync((char *)ctx->table, ctx->end, MS_SYNC | MS_INVALIDATE) == -1) {
#ifdef DEBUG
        sprintf(debug, “growdata: failed to msync datafile: %s”,
          strerror(errno));
        WRBLogMessage(WRBCtx, debug, 1);
#endif
        goto error;
    }

    UNLOCK(ctx)

    /* remap datafile */
    if (remapdata(ctx, WRBCtx) != WRB_DONE) {
        return (WRB_ABORT);
    }

    return (WRB_DONE);

error:
    UNLOCK(ctx)

    return(WRB_ABORT);
}
```