CHAPTER 11

The Database Manager

OVERVIEW

The database manager script provides a Web interface with which to manipulate databases stored in ASCII text files (flatfiles). Such manipulations include the ability to add to, modify, and delete from multiple databases based on keyword recognition.

For example, suppose a company maintains a database of its employees' personal information (such as home phone number and address) and that employee John Doe moves to a new address and gets a new phone number. Using our script, the database administrator can hop on the Web, call up the script, choose the **modify** option, select **John Doe** as the search word, and modify the database row to reflect the new information. No great understanding of the operating system or the database formats is necessary, because the Web provides the interface. The database administrator, perhaps a secretary familiar with using the Web, simply points and clicks using a Web browser.

To protect the possibly sensitive information in the database, however, this script also implements the password authentication algorithms discussed in Chapter 9. Furthermore, the script supports multiple administrators by incorporating *lock file* routines so that no one can make changes to the datafile while someone else is modifying it.

Installation and Usage

This application takes advantage of multiple files and directories that must work together as a team. Thus, the first step in installing these scripts is to copy them into a CGI-executable directory on your Web server. When the scripts are unarchived, they expand into the default directory structure beginning with the root directory, **Database_manager**. Figure 11.1 outlines the expanded directory structure and the permissions needed for the search to operate.

The root **Database_manager** directory must have permissions that allow the Web server to read and execute and should contain three directories (**Databases**, **Library**, and **Session Files**) and one text file (**db_manager.cgi**).

db_manager.cgi is the meat of the application and contains most of the Perl code to make it run. This file, which should be readable andexecutable by the Web server, will be discussed in detail in the design discussion.

Databases is a subdirectory containing "pairs" of database files: a setup file and a datafile for every database that has the database manager as a front end. It also contains the user and counter files discussed later. The datafiles must be readable and writable by the Web server, and the setup files must be readable. The directory itself must be readable, writable, and executable.

db_manager.users contains the list of users who are allowed to use the application. This user file is formatted exactly the same as the default user file discussed in Chapter 9: a pipe-delimited database containing the encrypted password, username, group, last name, first name, and E-mail address. This file should be readable and, if you want users to be added via the Web interface (not necessarily recommended), should be writable by the Web server. It should also initially be empty.

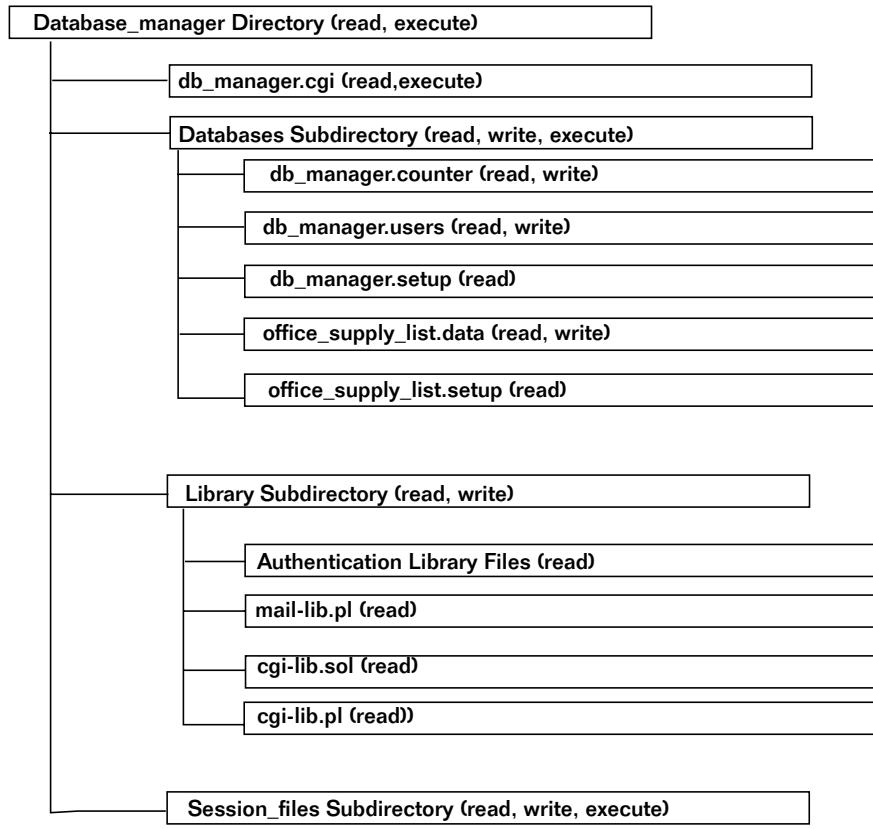

Figure 11.1 *The directory layout for the database manager application.*

db_manager.counter is a text file that is used to keep track of unique database ID numbers that have been assigned to database rows in the data file. Initially, this file should contain the number 1 on the first line and nothing else. As time goes by, **db_manager.cgi** will increment this number by 1 for every new database entry. Thus, the file should be readable and writable by the Web server.

The database manager can manipulate several databases at one time. To do this, however, it must have both a datafile and a setup file for every database.

Library is a subdirectory containing the CGI libraries (discussed in Chapter 9) that **db_manager.cgi** uses. The following library files are used for this script: **auth-extra-html.pl**, **auth-extra-lib.pl**, **auth-lib-fail-html.pl**, **auth-lib.pl**, **auth-server-lib.pl**, **auth_fail_html.pl**, **cgi-lib.pl**, **cgi-lib.sol**, and **mail-lib.pl**. This subdirectory must be executable and readable by the Web server, and its files must be readable.

Session_files is the subdirectory used by the authentication libraries to store session files, as discussed in Chapter 9. Initially, the directory will be empty, but if it is set to be readable, writable, and executable, the Web server will continually fill up and prune the directory as part of its daily use of session files, lock files, and temporary files.

Server-Specific Setup and Options

The Setup File

There are three primary functions of the setup file: defining server-specific variables, determining the names of database fields and their relative display orders, and configuring authentication options. Each setup file sets the following variables.

`$counter_file` is the path of the counter file that is used to keep track of unique ID numbers. By default, this variable is set to `./Databases/db_manager.counter`, because on the accompanying disk, **db_manager.counter** is located in the subdirectory databases.

`$user_file` is the location of the file that contains the list of users who are authorized to use **db_manager.cgi**.

`$data_file` is the location of the default ASCII database file that is being managed.

`$temp_file` is a file that we'll use to temporarily store various data at various times. It's mainly used in modifying and deleting from the database, as explained later.

`$lock_file` is a file that we'll use to make sure that only one person can modify the database at any given time.

`$session_file_directory` is the location of the directory that will temporarily hold session files, the temporary file, and the lock file.

`$database_manager_script` is the local absolute path of **db_manager.cgi**.

`$database_manager_script_url` is the URL of **db_manager.cgi**.

`%FIELD_ARRAY` is an associative array that is used to define database fields and their associated variable names.

`@field_names` is a list of fields in our database. Because of the hash table functions for associative array handling, if we used

```
@field_names = (keys %FIELD_ARRAY);
```

the field names would not come up in the order that we chose to assign them in `%FIELD_ARRAY`. Thus, we must manually specify the keys in the order that we want them in the `@field_names` array and the `@field_values` array.

`@field_values` is an array that contains only the variable names associated with the database fields. Whereas the `@field_names` elements are descriptive and can include spaces and special characters (such as single quotation marks or exclamation marks), the `@field_values` elements are used to generate form variables and hence should be in a variable name format.

`%FORM_COMPONENT_ARRAY` is an associative array that matches database fields with the kind of input type and the kinds of input type arguments the fields are associated with. Arguments and options are pipe-delimited so that type is always the first in the list, usually followed by a second item that includes the input arguments. In the case of a `<SELECT>`, however, options—including multiple and size options—are also added and are separated by pipes. For more details, see the discussion of **cgi-lib.sol** in Chapter 10.

Authentication variables are explained in the discussion of authentication in Chapter 9.

As an example, here is the complete text of **db_manager.setup**, which we will use for the chapter example.

```
#!/usr/local/bin/perl
$counter_file = "./Databases/db_manager.counter";
$user_file = "./Databases/db_manager.users";
$data_file = "./Databases/db_manager.data";
$temp_file = "./Session_files/db_manager.temp";
$lock_file = "./Session_files/db_manager.lockfile";
$session_file_directory = "./Session_files";
```

```
$database_manager_script = "./db_manager.cgi";
$database_manager_script_url = "db_manager.cgi";

%FIELD_ARRAY = ( 'Last Name', 'last_name',
                 'First Name', 'first_name',
                 'Email', 'email',
                 'Phone Number', 'phone',
                 'Address', 'address');
@field_names = ("Last Name", "First Name", "Email",
               "Phone Number", "Address");
@field_values = ("last_name", "first_name", "email",
                "phone", "address");


%FORM_COMPONENT_ARRAY = (
'Last Name', 'text|SIZE = "32" MAXLENGTH = "100"',
'First Name', 'text|SIZE = "32" MAXLENGTH = "100"',
'Email', 'text|SIZE = "32" MAXLENGTH = "100"',
'Phone Number', 'text|SIZE = "32" MAXLENGTH = "100"',
'Address', 'textarea|ROWS = "4" COLS = "30"');

$auth_lib = "$lib";
$auth_server = "off";
$auth_cgi = "on";

$auth_user_file = "./Databases/db_manager.users";
$auth_alt_user_file = "";
$auth_default_group = "view:add:modify:delete";
$auth_add_register = "on";
$auth_email_register = "off";
$auth_admin_from_address = "selena\@foobar.com";
$auth_admin_email_address = "selena\@foobar.com";
$auth_session_length = 2;
$auth_session_dir = "./Session_files";
$auth_register_message = "Thanks, you may now logon with
      your new username and password.";
$auth_allow_register = "on";
$auth_allow_search = "on";
$auth_generate_password = "off";
$auth_check_duplicates = "on";
$auth_password_message =
    "Thanks for applying to our site, your password is";
@auth_extra_fields = ("auth_first_name",
                      "auth_last_name",
                      "auth_email");
@auth_extra_desc = ("First Name",
                    "Last Name",
                    "Email");
```

The Datafile

Every datafile should be a pipe-delimited text file:

```
COMMENT: Item|Category|Description|Unique ID #
COMMENT:
100a|Bik Pen (Black)|10.00|1
100c|Mooky Stapler|12.00|3
100b|Bik Pen (Red)|11.00|2
```

Notice that the database manager scripts allow comment lines in the database. If you use comments, make sure that you follow the specific format for commenting. Any comment line must be flush against the left margin and must be preceded by `COMMENT:`.

Notice also that fields are delimited with the pipe symbol (|). Thus, you must not have any pipe symbols embedded in your data. The script has been configured by default to translate pipes into two tildes (~~), but if you make any manual additions, you must be careful to make the translation yourself. You are free to use a different delimiter by using search and replace to modify each occurrence of pipes in the scripts and datafiles, but we recommend the pipe symbol, because it is rarely found in user data.

You may have as many fields or rows as you want, but you must always allow the database manager to add a final field for the unique database ID number. To make sure that modifications and deletions affect only one database row, we need to make sure that every row is uniquely identifiable. Thus, the add routine in the database manager assigns a unique number as the last field of every row. If you wish to manually add data rows by editing the datafile directly, you must remember to insert ID numbers at the end of each of the rows as well as increment the counter file appropriately.

Running the Script

Once you have configured the setup file to the specifics of your server and databases, it is time to rev up the motor of the application, the **db_manager.cgi** script. If you have configured your setup file correctly and have set all the permissions for files and subdirectories properly, all you need to do is to create a hyperlink to the script. For example:

```
<A HREF = "http://www.foobar.com/cgi-
bin/Database_manager/db_manager.cgi">Database manager</A>
```

Design Discussion

When a user clicks on the link, the main script will run, taking advantage of your default setup configurations. Figure 11.2 documents the logic of this script as it manages the needs of the client with the abilities of the server.

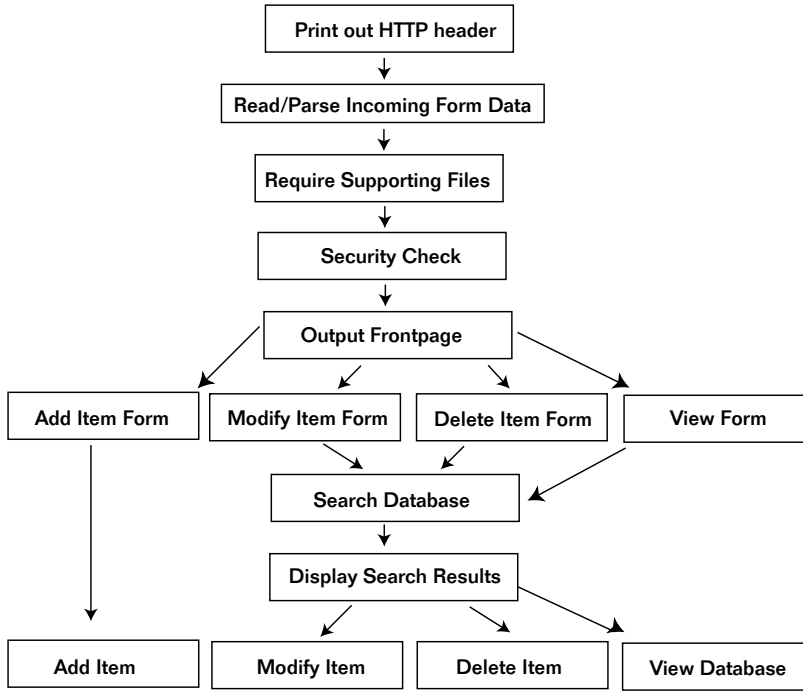

Figure 11.2 *The script logic.*

The script begins by starting the Perl interpreter and printing the HTTP header.

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
```

Loading the Libraries

Next, the **Library** subdirectory is added to the `@INC` array used by Perl to locate supporting libraries. By `unshift`ing the subdirectory into the array, the script makes sure to add it to the beginning of the `@INC` array so that any duplicate subroutines in other libraries are not used. Notice that if you move the library files to a directory other than **Library**, you will need to change this hard-coded variable.

Once the **Library** directory has been included in the `@INC` array, the necessary supporting files are loaded using the `CgiRequire` subroutine at the end of this script. The `CgiRequire` subroutine returns a meaningful error message in case the script has a problem requiring one of the libraries.

```
$lib = "Library";
unshift (@INC, "$lib");
&CgiRequire("$lib/cgi-lib.pl", "$lib/cgi-lib.sol",
"$lib/auth-lib.pl");
```

Reading and Parsing the Incoming Form Data

Next, the script uses **cgi-lib.pl** to parse the incoming form data, passing the `ReadParse` subroutine the parameter `*form_data` so that the returned incoming form data associative array will be named `%form_data` instead of `%in`.

```
&ReadParse(*form_data);
```

In this script, we chose to name the associative array returned by cgi.lib.pl form_data rather than the default %in because it is easier to remember what the variable is when we have many associative arrays in one script.

Loading the Setup File

Once the form data has been processed, the script checks to see whether the client has asked to see a database other than the default database.

Because **db_manager.cgi** can be used to manage several databases at once, we need a way of communicating to the script which database it should use and which setup file is associated with that database. Thus, when you're not accessing the default database, you must append some special information to the `<A HREF>` tag in the HTML that calls the script. You should use the following format:

```
<A HREF = "http://www.foobar.com/cgi-
bin/Db_manager/db_manager.cgi?database=office_supply_list.setup">Offic
e Supply Database</A>
```

The question mark signifies that you are passing parameters to **db_manager.cgi**. In the example, the variable `database` is set equal to `office_supply_list.setup` and passed to the script.

That the setup file is determined at this point is the reason we had to hard-code `$lib`. We still have not used the setup file, because we are not yet sure which setup file to use. Once we parse the data, if there was an incoming variable named `database`, we will know to use its value as the name of the database setup file to use. If no value came in, we can use (`require`) the default database's setup file.

Notice also the use of the hard-coded `Databases`. This could cause problems when you are customizing this script to your site if you are changing directory relationships. If you change the directory relationships, be aware that you will have to modify the `require` subroutine call to reflect the change.

```
if ($form_data{'database'} ne "")
  {
  $setup_file = $form_data{'database'};
  }
else
  {
  $setup_file = "db_manager.setup";
```

```
  }
&require("Databases/$setup_file");
```

Authenticating Users

After the script has gathered together all the supporting files, it must check to see whether the client is authorized to access or modify the datafile.

To do this, the script passes three parameters to the subroutine `GetSessionInfo`, which is contained in **auth-lib.pl**. `$session_file` will be `null` if a session file has not been set; otherwise, it will be the value associated with the `session_file` variable coming in from the form. As you'll see later, we will pass this value as a hidden variable once it has been assigned. `GetSessionInfo` also requires the name of this script so that it can provide links to this script. Finally, the script must pass to `GetSessionInfo` the associative array containing the form data returned from **cgi-lib.pl**.

In exchange, the routine returns to us several bits of authentication information; the main script passes this information as hidden form variables throughout the following routines to keep track of the user and his or her privileges with the database. If you are still unclear about this process, it is covered in greater detail in Chapter 9.

```
if ($form_data{'session_file'} ne "")
  {
  $session_file = $form_data{'session_file'};
  }
($session_file, $session_username, $session_group,
$session_first_name, $session_last_name, $session_email) =
&GetSessionInfo($session_file, $database_manager_script_url,
*form_data);
```

Next, the script `splits` the `$session_group` information so that it can keep track of what type of permission the user has. The `session_group` variable will look something like `view:add:modify:delete` (as defined in the setup file), so the script `splits`, based on the colons, to see what levels of security the user is cleared for.

```
@group_array = split (/:/, $session_group);
```

Displaying the Front Page

Next, the script checks to see whether the client is requesting the first page. The script knows that the client needs to see the first page if the client has just clicked on an input button named **return_to_frontpage** or has just made it through security and the `$session_file` is coming from the authentication routines rather than as hidden form input from other pages deeper in this script. The || means "or."

```
if ($form_data{'return_to_frontpage'} ne "" ||
    $form_data{'session_file'} eq "")
    {
```

If either condition is true, the script prints the database manager front page, which gives the client four choices of actions (add, modify, delete, and view).

```
print <<"    end_of_html";
<HTML>
<HEAD>
<TITLE>Database Manager Front Page</TITLE>
</HEAD>
<BODY>
<CENTER><H2>Database Manager Front Page</H2></CENTER>
<BLOCKQUOTE>
Thanks for checking out the Database Manager Scripts.  Feel free to
make any modifications that are necessary.
</BLOCKQUOTE>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Add an Item">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete an Item">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify an Item">
```

```
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View the Database">
end_of_html
exit;
}
```

In the routine, we chose to use the `end_of_html` method for printing to make it easier for administrators to customize the look and feel of the HTML. However, be careful to escape Perl special characters such as @ within the HTML.

On the Web, the front page will look like Figure 11.3.

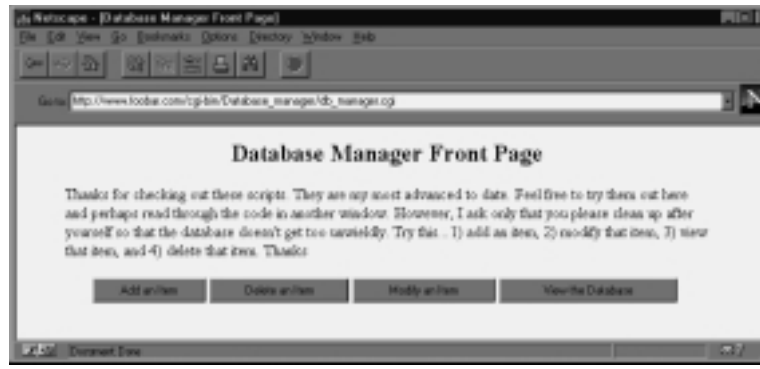

Figure 11.3 *The database manager front page.*

Displaying the Add Form

If the client asked for the form to add an entry to the database, the script prints that form instead.

```
if ($form_data{'add_form'} ne "")
  {
```

Notice that throughout these `if` tests, the standard `if submit button ne ""` syntax is used. Thus, you can set the value of the submit buttons to whatever you want. An add button might be `Add New Employee Records` instead of `Add Item`. When the user clicks on the button, regardless of the value, this script will evaluate the `if` test as `true`. This is because if the submit button is pressed, its form value will not be null (`ne ""`).

The script prints the page header, body, document header, and form tags as it did for the front page–generating routine.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Adding an Item to the Database</TITLE>
</HEAD><BODY>
<CENTER><H2>Adding an Item to the Database</H2></CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

Using the subroutine `create_input_form` at the end of this script, the script creates an input form that the client uses to submit the information for the new database row. This subroutine creates an input field for each of the fields in the database and presents it in table format.

```
&create_input_form;
```

Finally, the script prints the page footers.

```
print <<"    end_of_html";
</TABLE><CENTER><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "hidden" NAME = "id"
       VALUE = "$item_number">
<INPUT TYPE = "submit" NAME = "submit_addition"
       VALUE = "Submit Addition">
```

```
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the add form looks something like Figure 11.4.

Figure 11.4 *Database manager add form.*

Displaying the Search Form for Deletion

On the other hand, perhaps the client wants to delete an item from the database. If so, the script outputs a form so that the client can specify which item to delete.

```
if ($form_data{'search_form_delete'} ne "")
  {
```

Before the script can begin deleting, it must find out which item to delete. To do that, the client must have a way of telling the script which item to delete and the script must tell the client which items are available to delete. However, the script cannot spew out all the items in the database, because the browser might run out of memory; also, you may not want the client to get all the information in the database. Instead, the client must be able to give the script one or more search terms so that it can put together a reasonably sized list from which the client can then choose.

This communication is created by using an HTML input form that includes an input field for every database field so that the client can search by keyword and field. The script begins by printing the HTML header for the HTML form.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Query Database for Deletion</TITLE>
</HEAD><BODY>
<CENTER><H2>Query Database for Deletion</H2></CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

Then the script outputs the form using the subroutine `create_input_form` at the end of this script. The logic is the same as for the equivalent call for add.

```
&create_input_form;
```

Next, the script adds a form input for "exact match" and the form, body, and HTML footers and tags. The exact match option lets users search by word boundary so that the script will return only rows with an exact hit.

```
print <<"    end_of_html";
<TH>Exact Match?</TH><TD>
<INPUT TYPE = "checkbox" NAME = "exact_match">
</TD></TR></TABLE><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
```

```
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "search_database_delete"
       VALUE = "Submit Search Term">
<P><BLOCKQUOTE>To get a full view of database, submit "no" keywords.
But beware, if there are too many items in your database, you will
exceed the memory of your browser.
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

All searches are case-insensitive, but the exact match matches on word boundary so that it will not match gen'eric' if the search is for eric, whereas the nonexact search will find the item.

On the Web, the previous routine will yield a page that looks exactly like Figure 11.4, except that the title will be specific to deletion and the exact match check box will appear on the page.

Searching the Database for Possible Items to Delete

The script also needs a routine to accept the client-defined search term(s) and search the database so that it can present a dynamically generated list of hits from which the client can choose items to delete.

```
if ($form_data{'search_database_delete'} ne "")
  {
```

The script conducts the search and prints the results using the following routines. First, the script prints the page header.

```
print <<"  end_of_html";
<HTML><HEAD>
<TITLE>Deleting an Item from the Database</TITLE>
</HEAD><BODY>
<CENTER>
<H2>Deleting an Item from the Database</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

The script then begins searching the database by using the subroutine `search_database`, which can be found at the end of this script. The `search_database` subroutine is given one parameter, `delete`, so that it knows how to output the results of the search.

```
  &search_database ("delete");
```

Finally, the script prints a list of hits in table format. `$search_results`, returned from `&search_database`, contains all the hits in the form of table rows. The client chooses which item to delete and deletes it with the `if ($form_data{'submit_deletion'} ne "")` routine.

```
print "<TABLE BORDER = \"1\" CELLPADDING = \"4\"
      CELLSPACING = \"4\">";
print &table_header ("Delete<BR>Item");
push (@field_names, "Id");
print &table_header (@field_names);
print "</TR>\n";
print "$search_results";
print <<"  end_of_html";
</TABLE><P>
$search_notice<P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "submit_deletion"
       VALUE = "Submit Deletion">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the delete form appears as shown in Figure 11.5.

Displaying the Search Form for Modification

Next, we repeat the same logic, adapting it for the modification operation.

```
if ($form_data{'search_form_modify'} ne "")
  {
```

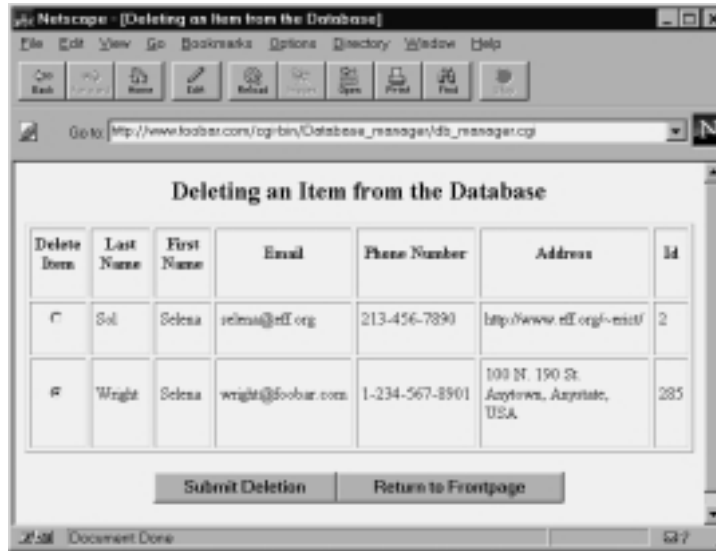

Figure 11.5 *Itemized delete form.*

The script begins by printing the familiar page header.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Query Database for Modification</TITLE>
</HEAD><BODY>
<CENTER>
<H2>Query Database for Modification</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

As it did in the delete routines, the script creates the input form using the subroutine `create_input_form`.

```
&create_input_form;
```

It then adds the footer.

```
print <<"    end_of_html";
<TH>Exact Match?</TH><TD>
```

```
<INPUT TYPE = "checkbox" NAME = "exact_match">
</TD></TR></TABLE><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "search_database_modify"
       VALUE = "Submit Search Term">
<P><BLOCKQUOTE>To get a full view of database, submit "no" keywords.
But beware, if there are too many items in your database, you will
exceed the memory of your browser.
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the modification screen looks like Figure 11.4 except for the new title and the addition of the exact match check box.

Searching the Database for Possible Rows to Modify

Just as it did for delete, the script prints the results of the query for modification.

```
if ($form_data{'search_database_modify'} ne "")
  {
```

The script prints the page header.

```
print <<"  end_of_html";
<HTML><HEAD>
<TITLE>Modifying an Item in the Database</TITLE>
</HEAD><BODY>
<CENTER>
<H2>Modifying an Item in the Database</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

Next, the script searches the database using the subroutine `search_database`.

```
&search_database ("modify");
```

Now the script prints a list of hits in table format. Recall that `$search_results`, returned from `&search_database`, contains all the hits in the form of table rows.

```
print "<TABLE BORDER = \"1\" CELLPADDING = \"4\"
              CELLSPACING = \"4\">";
print "<TH>Modify<BR>Item</TH>";
push (@field_names, "Id");
print &table_header (@field_names);
print "</TR>\n";
print "$search_results";
print "</TABLE><P>";

print "$search_notice<P>";
```

The script uses the subroutine `create_input_form` to print the same form that was used for adding items. Now the client can specify an item in the created table and can type new information with which to update the database row.

```
&create_input_form;
```

Finally, the script prints the usual footer.

```
print <<"    end_of_html";
</TABLE><CENTER><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "submit_modification"
       VALUE = "Submit Modification">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the modification form and table look like Figure 11.6.

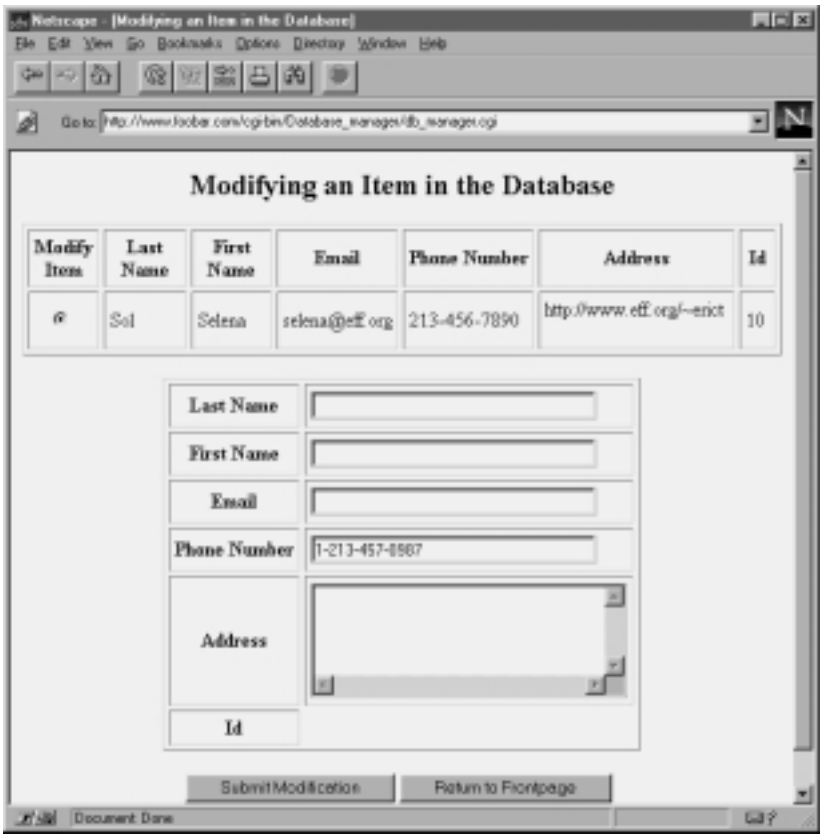

Figure 11.6 *Itemized modification screen.*

Displaying the Search Form for Viewing the Database

The script uses the same routines to produce a view of the database.

```
if ($form_data{'search_form_view'} ne "")
  {
```

The script begins by printing the usual header.

```
print <<"    end_of_html";
<HTML><HEAD>
```

```
<TITLE>Query Database for Viewing</TITLE>
</HEAD><BODY>
<CENTER><H2>Query Database for Viewing</H2></CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

Then the script creates the input form using the subroutine `create_input_form` at the end of this script, exactly as we've done before.

```
&create_input_form;
print <<"    end_of_html";
<TH>Exact Match?</TH><TD>
<INPUT TYPE = "checkbox" NAME = "exact_match">
</TD></TR></TABLE><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "search_database_view"
       VALUE = "Submit Search Term">
<P><BLOCKQUOTE>To get a full view of database, submit "no" keywords.
But beware, if there are too many items in your database, you will
exceed the memory of your browser.
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

The previous routine produces the usual screen.

Searching the Database for Items to Display for Viewing

Now the script must search the database for the client to view. The process is the same as we've done before.

```
if ($form_data{'search_database_view'} ne "")
  {
  print <<"    end_of_html";
  <HTML><HEAD><TITLE>Viewing the Database</TITLE></HEAD>
  <BODY>
  <CENTER><H2>Viewing the Database</H2></CENTER>
  <FORM METHOD = "post"
```

```
        ACTION = "$database_manager_script_url">
  <CENTER>
  end_of_html
```

This time, however, the script sends the subroutine a parameter of `none`, because we do not want any radio buttons in the resulting table. There will be no items to select, because all the client wants to do is to view the database.

```
&search_database ("none");
```

Finally, the script prints a list of hits in a table format.

```
print <<"  end_of_html";
<TABLE BORDER = "1" CELLPADDING = "4" CELLSPACING = "4">
end_of_html
push (@field_names, "Id");
print &table_header (@field_names);
print "</TR>\n";
print "$search_results";
print <<"  end_of_html";
</TABLE><CENTER><P>
$search_notice<P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Add an Item">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete an Item">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify an Item">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the search results page looks like Figure 11.7.

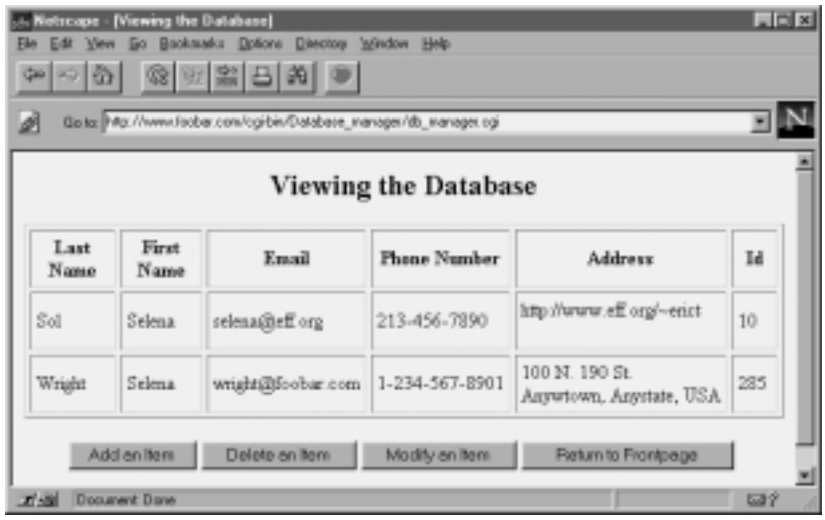

Figure 11.7 *Search results page.*

Adding an Item to the Database

Now that the user has the ability to use script-generated HTML forms to specify an addition, modification, or deletion, the script must be prepared to make those changes. The script begins by providing for a database addition.

```
if ($form_data{'submit_addition'} ne "")
  {
```

In the case of an addition, the script must assign the new database entry a unique database ID number. To do this, it accesses a counter file that keeps track of the last database ID number used. The `counter` subroutine, located in **cgi-lib.sol**, returns a new number (incremented by 1) and then adjusts the counter file appropriately. The unique ID number is essential for modifying the database; without it, the script would have no way of identifying unique rows. The `counter` routine takes one parameter, the location of the counter file.

Before the subroutine can begin editing the counter file, the script must make sure that no one else is incrementing the file at the same time. If that happened, we'd have two database rows with the same "unique" ID number, and that wouldn't do. The script uses the `GetFileLock` routine in **cgi-lib.sol** to create a lock file that prevents anyone else from accessing the counter file while it is being incremented. When the script is finished with the counter file, it uses the `ReleaseFileLock` subroutine to release the lock.

```
&GetFileLock ("$lock_file");
&counter($counter_file);
&ReleaseFileLock ("$lock_file");
```

The script then takes the incoming form data and formats it the way the database is set up to understand (delimited by | and ending with a newline). In the process, the script also translates all hard returns into `<BR>` and paragraph breaks into `<P>` so that the database row will display correctly as HTML information. The script also changes all pipe characters (|) into two tildes (~~) so that input pipe data will not confuse the database manager later. Recall that the database manager stores database rows using the pipe character by default. If the user submitted a pipe character as part of the database information, the pipe character would confuse the database manager into thinking that there were too many database fields.

```
foreach $field (@field_names)
  {
  $value = "$FIELD_ARRAY{$field}";
  $form_data{$value} =~ s/\n/<BR>/g;
  $form_data{$value} =~ s/\r\r/<P>/g;
  $form_data{$value} =~ s/\|/~~/g;
```

Finally, if the user did not submit any data at all, the script changes the empty value into a centered "-" so that when the database row is displayed in table format, it will not look ugly.

```
  if ($form_data{$value} eq "")
    {
```

```
    $form_data{$value} = "<CENTER>-</CENTER>";
    }
```

For each `$field`, the script appends each formatted field to `$new_row`, taking off the last pipe when it is finished building the new database row.

```
  $new_row .= "$form_data{$value}|";
  }
$new_row .= "$item_number";
```

Next, the script creates a lock file while it edits the database file. Then it writes in the new row. Notice also that the script appends (`>>`) to the data file so that all the old data remains intact. Also, the script uses the subroutine `open_error` in **cgi-lib.sol** if there is a problem opening the datafile.

```
&GetFileLock ("$lock_file");
open (DATABASE, ">>$data_file") || &open_error($data_file);
print DATABASE "$new_row\n";
close (DATABASE);
```

When the script is finished adding the item, it deletes the lock file so that other instances of the script may access the datafile.

```
&ReleaseFileLock ("$lock_file");
```

Finally, the client is sent an HTML page letting her know that the item has been added and, once again, giving the basic administrative options.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Item Added to the Database</TITLE>
</HEAD><BODY>
<CENTER><H2>Item Added to the Database</H2></CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "add_form"
```

```
       VALUE = "Add an Item">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete an Item">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify an Item">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View the Database">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the response page takes the standard form depicted in Figure 11.8.

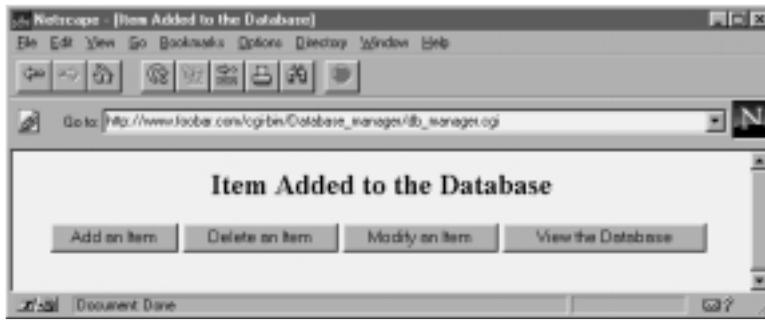

Figure 11.8 *Standard response after successful addition.*

Deleting an Item from the Database

If the client has asked to delete an item, the script uses the following routines to delete from the database:

```
if ($form_data{'submit_deletion'} ne "")
  {
```

The routine begins by opening the datafile using the `open_error` routine in **cgi-lib.sol** in case there is a problem.

```
  open (DATABASE, "$data_file")
       || &open_error($data_file);
```

Then the script reads the datafile one line at a time.

```
while (<DATABASE>)
  {
```

If the line is a database comment line, the script adds it to a variable called `$new_data`. This variable will eventually contain every line in our datafile, because we continually append to it (`.=`). In this case, the script simply adds to `$new_data` all the comment lines before further processing.

```
if ($_ =~ /^COMMENT:/)
  {
  $new_data .= "$_";
  }
```

If the current line is not a comment line, the script `splits` the line into the `@fields` array using a pipe (`|`) as the element delimiter. The script then `pops` out the unique database ID number.

```
else
  {
  @fields = split (/\|/, $_);
  $item_id = pop(@fields);
```

`Pop` is a Perl function that snips off the very last element in an array. In this case, in the setup file we defined the last element in every database row to be the unique database ID number gathered from the `counter` routine.

Now the script compares the database ID number of the current row with the database ID number submitted by the client by way of the delete form. If the two numbers are not equal, the script adds the row to the growing `$new_data` variable. If they are equal, however, the script does not add the row.

Thus, by the end of the `while` loop, every row from the old database will have been added to `$new_data` except the line that the script was asked to delete.

```
unless ($item_id eq "$form_data{'delete'}")
  {
```

```
        $new_data .= "$_";
        }
      } # End of else
    } # End of  while (<DATABASE>)
  close (DATABASE);
```

At this point, the script opens a temporary file and copies to it the old database rows contained in `$new_data`. Then it uses the `rename` command to copy the temporary file over the old database file to finalize the deletion.

```
  &GetFileLock ("$lock_file");
  open (TEMPFILE, ">$temp_file")
       || &open_error($temp_file);
  print TEMPFILE "$new_data";
  close (TEMPFILE);
  rename ($temp_file, $data_file);
  &ReleaseFileLock ("$lock_file");
```

Finally, the script prints the usual HTML response, which looks like Figure 11.8 except for the deletion-specific header.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Your Item has been deleted</TITLE>
</HEAD><BODY>
<CENTER><H2>Your Item has been deleted</H2></CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Add an Item">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete an Item">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify an Item">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View the Database">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

Modifying an Item in the Database

Finally, the script checks to see whether the client is asking to make a modification.

```
if ($form_data{'submit_modification'} ne "")
  {
```

Before the script can make a modification, however, it must be sure that the client has submitted new information as well as a database row for modification. If the client has filled in the new information but has forgotten to select an item by using the radio buttons, the script must warn him or her to choose a database row.

```
  if ($form_data{'modify'} eq "")
    {
    print <<"      end_of_html";
    <HTML><HEAD>
    <TITLE>Modifying an Item in the Database</TITLE>
    </HEAD>
    <BODY>
    <CENTER>
    <H2>Modifying an Item in the Database</H2>
    </CENTER>
    <BLOCKQUOTE>
    I'm sorry, I was not able to modify the database
    because none of the radio buttons on the table was
    selected and I was not sure which item to modify.
    Would you please make sure that you select an item
    "and" fill in the new information.  Please press the
    back button and try again.  Thanks.
    </BLOCKQUOTE>
    end_of_html
    exit;
    }
```

Once we're sure that the user has selected an item and has typed new information into the text input boxes, the script modifies the database. As usual, the script opens the datafile for reading.

```
  open (DATABASE, "$data_file")
       || &open_error($data_file);
```

As it did for deletion, the script adds comment lines to `$new_data`.

```
while (<DATABASE>)
  {
  if ($_ =~ /^COMMENT:/)
    {
    $new_data .= "$_";
    }
```

The script also gets the `item_id` number from each database row. However, this time the script pushes the number back onto the end of the `@fields` array because we want that information to be stored with the rest of the database row for further processing.

```
  else
    {
    @fields = split (/\|/, $_);
    $item_id = pop(@fields);
    push (@fields, $item_id);
```

This time, if the item ID is equal to the item number submitted by the client, the script renames the fields array to `@old_fields`.

```
    if ($item_id eq "$form_data{'modify'}")
      {
      @old_fields = @fields;
      }
```

Once the row to be modified has been identified, the script dumps all the other rows into the `$new_data` array, as was done for delete.

```
    else
      {
      $new_data .= "$_";
      }
    } # End of else
  } # End of while (<DATABASE>)
```

The script then initializes a couple of variables. `$counter` is used to keep track of the number of fields in the database row, and `$new_line` is used to build the new database row. Both variables will be discussed later.

```
$counter = 0;
$new_line = "";
```

Next, the script goes through the list of `field_values` as defined in the setup file. Recall that arrays begin at zero, so the first array element is `$arrayname[0]`. The script assigns the current element in our count of field values to `$value`, thus going through every field in the database.

```
until ($counter >= @field_values)
  {
  $value = "";
  $value = "$field_values[$counter]";
```

If the `form_data` variable associated with that field does not have a value, the script adds the "old" field value stored in `@old_fields` to the `$new_line` variable.

```
  if ($form_data{$value} eq "")
    {
    $new_line .= "$old_fields[$counter]|";
    }
```

On the other hand, if the client submitted new information, the script formats the information as it did for the add routine and adds the resulting value to `$new_line`.

```
  else
    {
    $form_data{$value} =~ s/\n/<BR>/g;
    $form_data{$value} =~ s/\r\r/<P>/g;
    $form_data{$value} =~ s/\|/~~/g;
    if ($form_data{$value} eq "")
      {
      $form_data{$value} = "<CENTER>-</CENTER>";
      }
    $new_line .= "$form_data{$value}|";
    }
    $new_line .= "$sold_fields[$counter]\n";
```

Next, the script increments the counter by 1 so that the loop goes through every field in a database row. Once it is finished with the loop, the script closes the database.

```
    $counter++;
    } # End of until ($counter >= @field_values)
 close (DATABASE);
```

Finally, as before, the script locks the datafile and opens a temporary file. It then prints the `$new_data` variable (containing all the old, unmodified database rows) and `$new_line` (containing the modified row) to the temp file. Then the script copies over the old datafile with the temp file and removes the lock. This action finalizes the modification.

```
  chop $new_line;
  &GetFileLock ("$lock_file");
  open (TEMPFILE, ">$temp_file")
       || &open_error($temp_file);
  print TEMPFILE "$new_data";
  print  TEMPFILE "$new_line";
  close (TEMPFILE);
  rename ($temp_file, $data_file);
  &ReleaseFileLock ("$lock_file");
```

Finally, the script prints the usual HTML response, which looks like Figure 11.8 except for the modification-specific title.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Your Item has been Modified</TITLE>
</HEAD><BODY>
<CENTER><H2>Your Item has been Modified</H2></CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$setup_file">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$form_data{'session_file'}">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Add an Item">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete an Item">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify an Item">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View the Database">
</CENTER></FORM></BODY></HTML>
```

```
end_of_html
exit;
}
```

The CgiRequire Subroutine

The first supporting subroutine, `CgiRequire`, checks to see whether the file that we are trying to exists and is readable by us. This subroutine provides developers with an informative error message when they're attempting to debug the scripts.

```
sub CgiRequire
  {
```

First, the `@require_files` array is defined as a local array and is filled with the filenames sent from the main routine.

```
local (@require_files) = @_;
```

The subroutine then checks to see whether the files exist and are readable. If they are, the files are `loaded`.

```
foreach $file (@require_files)
  {
  if (-e "$file" && -r "$file")
    {
    require "$file";
    }
```

If any of the files are not readable or do not exist, the subroutine sends back an error message that identifies the problem.

```
else
  {
  print "I'm sorry, I was not able to open
  $file.  Would you please check to make sure
  that you gave me a valid filename and that the
  permissions on $require_file are set to allow me
  access?";
  exit;
  }
```

```
 } # End of foreach $file (@require_files)
} # End of sub CgiRequire
```

The create_input_form Subroutine

The `create_input_form` subroutine is used to create the input forms for the add, modify, and delete user interfaces so that the user can search by keyword and enter the needed data.

```
sub create_input_form
  {
```

First, the subroutine prints a table header.

```
print "<TABLE BORDER = \"1\" CELLPADDING = \"4\"
              CELLSPACING = \"4\">";
```

Next, `create_input_form` makes an HTML form so that administrators can add items to the database. This form has one input field for each field in the database except the database ID field, which is set by this script and not by the administrator. The subroutine gets the list of database fields from the `@field_names` array and, for every element in the array, creates an input field. The subroutine sends the `build_input_form` subroutine in **cgi-lib.sol** a few things: the name of the field, the variable to be associated with that name, and the type of input we are going to use (TEXTAREA, TEXT, or SELECT).

```
foreach $field_name (@field_names)
  {
  print &build_input_form("$FIELD_ARRAY{$field_name}",
        "$FORM_COMPONENT_ARRAY{$field_name}",
        $field_name);
  }
}
```

The search_database Subroutine

The `search_database` subroutine is used to search the database based on keyword input from the client. The `main` routine passes to this subroutine

the type of search (modify, delete, or view) for which the results should be formatted.

Notice that the routine uses the keyword none **as the submit type for viewing the data. A view is not a submission as much as it is a simple request.**

First, the subroutine assigns the submit type value to the local variable `$submit_type`.

```
sub search_database
  {
    local($submit_type) = @_;
```

Next, the subroutine opens the database file and begins checking each field in every row against the keywords submitted. It also initializes the `$number_of_hits` variable, which is used to keep track of how many hits come up in the search.

```
$number_of_hits = "0";
open (DATABASE, "$data_file");
while (<DATABASE>)
  {
  $database_row = $_;
```

Then `$not_found_flag` is defined as zero to make sure that it has not been initialized elsewhere. The `$not_found_flag` variable is used to keep track of whether a hit was made based on the client-submitted keyword. If a match was found, the variable will equal 1, but more on that in a bit.

```
$not_found_flag = "0";
```

The next line ensures that the search disregards any line that is a database comment line.

```
unless ($database_row =~ /^COMMENT:/)
  {
```

The subroutine then `splits` the database row into the `@row` array.

```
@row = split(/\|/,$database_row);
```

For each key in the `%form_data` associative array, the script sets `$field_number` equal to –1. That's because both the pesky "submit" and "exact match" key/value pairs may come in along with the rest of the form data. We do not want the script to search the database for those fields, because they don't exist! By setting `$field_number` equal to –1, we filter out such nonfield keys using the following routine:

```
foreach $form_data_key (keys %form_data)
  {
  $field_number = -1;
```

Then the subroutine goes through the fields in the database (`@field_values`), checking to see whether there is a corresponding value coming in from the form (`$form_data_key`). However, because arrays are counted from zero rather than from 1 and because `@field_values` gives us a count of the array starting at 1, the subroutine must offset the counter by 1. Thus, if the form "key" submitted is indeed a field in the database, `$field_number ($y - 1)` will be its actual location in the array.

```
for ($y = 1; $y <= @field_values; $y++)
  {
  if ($form_data_key eq @field_values[$y-1] &&
      $form_data{"$form_data_key"} ne "")
    {
    $field_number = $y - 1;
    last; # Exit out of the for loop because we have
          # verified field
    }
  } End of for loop
```

Next, the script checks the submitted value against the value in the database. It also makes sure that the value is not `on` or `submit keyword`. If `$field_number` is still equal to –1, it means that the form data key did not match an actual database field, so we should try the next key. Otherwise, the script knows that it has a valid database field to check against. Again, `$field_number` must be less than or equal to –1, because array starts from zero.

```
if ($field_number > -1)
  {
```

Now the subroutine performs the match, checking the database information against the submitted keyword for that field. If it is requested to do so, the script performs an exact match test. If `$form_data{'exact_match'}` is equal to nothing, then the exact match check box was not checked. The match is straightforward. If the keyword string (`$form_data{"$form_data_key"}`) matches (`=~`) a string in the field being searched (`@row[$field_number]`) with case insensitivity (`/i`), the script knows that it has found a hit!

```
if ($form_data{'exact_match'} eq "")
  {
  unless (@row[$field_number]
          =~/$form_data{"$form_data_key"}/i)
  {
```

If there was no match, the subroutine sets `$not_found_flag` equal to 1. In the end, if it gets through all the fields and still has not found a match, the script will be able to tell the client that no matches were found.

```
$not_found_flag = "1";
last; # Exit out of ForEach keys in FormData
} # end of unless
} # end of if ($form_data{'exact_match'} eq "")
```

On the other hand, the client may have clicked the exact match check box.

```
else
  {
```

This time, the subroutine proceeds with an exact match using the `\b` switch to define the word boundary of the keyword while still keeping the search case-insensitive (`/i`). The same `$not_found_flag` setting applies.

```
unless (@row[$field_number] =~/\b$form_data{"$form_data_key"}\b/i)
    {
```

```
    $not_found_flag = "1";
    last; # Exit out of ForEach keys in FormData
     } # end of unless
    } # end of else
  } # End of if ($field_number > -1)
} # End of ForEach keys in FormData
```

If a match was found (`$not_found_flag` still equals 0), the script checks for group privileges.

```
if ($not_found_flag eq "0")
  {
  $number_of_hits++;
  if ($number_of_hits > "25")
    {
    $search_notice .= "This search engine will only
                      return 25 hits, and, your
            search turned up more than that.
            Would you please refine your
       search?";
     last;
     }
  $hit_counter = "1";
  $db_id_number = pop (@row);
  push (@row, $db_id_number);
```

In the case of a deletion, the subroutine checks to see whether the client is authorized to make a deletion. Recall that the script defined this `@group_array` when it initially made the security check with `GetSessionInfo`. If the user is not allowed to delete, the element in `group_array` will not have a value and the subroutine will set the variable `$permission` equal to `no`.

```
if ($submit_type eq "delete")
    {
    $group_test = "$group_array[3]";
    if ($group_test eq "")
      {
      $permission = "no";
      }
    } # End of if ($submit_type eq "delete")
```

The subroutine then does the same for modification.

```
if ($submit_type eq "modify")
  {
  $group_test = "$group_array[2]";
  if ($group_test eq "")
    {
    $permission = "no";
    }
  }
```

If the subroutine has gone through those last three routines and at some point `$permissions` has been set to `no`, clients are told that they are not allowed to go forward.

```
if ($permission eq "no")
  {
  print <<"    end_of_html";
  </CENTER><BLOCKQUOTE>
  I am sorry, it appears that you do not have \
  permission to $submit_type.  Please contact the
  database administrator to find out how to gain
  access.  Sorry about that!
  </BLOCKQUOTE><CENTER>
  <INPUT TYPE = "hidden" NAME = "database"
         VALUE = "$setup_file">
  <INPUT TYPE = "hidden" NAME = "session_file"
         VALUE = "$session_file">
  <INPUT TYPE = "submit" NAME = "add_form"
         VALUE = "Add an Item">
  <INPUT TYPE = "submit" NAME = "search_form_delete"
         VALUE = "Delete an Item">
  <INPUT TYPE = "submit" NAME = "search_form_modify"
         VALUE = "Modify an Item">
  <INPUT TYPE = "submit" NAME = "search_form_view"
         VALUE = "View the Database">
  end_of_html
  exit;
  }
```

Otherwise, the subroutine creates database rows to display. First, the subroutine appends to `$search_results` an initial `<TR>`.

```
$search_results .= "<TR>";
```

Then, if the client has asked for a modify or delete form, the subroutine creates a radio input button so that the client can select a database row.

None **is the value used if there is no need to create a submit radio option. If the client wants only to view the database and does not need to select an item for further processing, we should return a table without a radio button.**

The radio button has a value equal to the `database_id` number so that each row can be identified when its corresponding radio button is clicked.

```
  if ($submit_type ne "none")
     {
     $search_results .= "<TD ALIGN = \"center\">\n";
     $search_results .= "<INPUT TYPE = \"radio\"
                         NAME = \"$submit_type\"
                         VALUE = \"$db_id_number\">";
     $search_results .= "\n</TD>\n";
     }
```

Next, the subroutine adds to the `$search_results` variable the individual database fields for every row and adds a closing `</TR>` for every row.

```
foreach $field (@row)
     {
     $search_results .= "<TD>$field</TD>\n";
     }
   } # End of if ($not_found_flag eq "0")
 $search_results .= "</TR>\n";
 } # End of unless ($database_row =~ /^COMMENT:/)
} # End of while (<DATABASE>)
close (DATABASE);
```

Finally, the subroutine handles the possibility that a client-submitted keyword may turn up nothing in the search. At this point, if `$hit_counter` is not equal to 1, it means that the search did not turn up a hit. Thus, the subroutine must send a note to the client with a link to the search form.

Notice that we use a hyperlink rather than a submit button. We want the client to access this script without a `CONTENT_LENGTH` so that the form will pop up.

If no hits were found, the subroutine exits. We do not want to print an empty table.

```
if ($hit_counter ne "1")
  {
  print "I'm sorry, I was unable to find a match for the
  keyword that you specified in the field that you
  specified.  Feel free to
  <A HREF = \"db_manager.cgi\">try again</A>";
  print "</CENTER></BODY></HTML>";
  exit;
  }
          }
```