CHAPTER 14

The Classified Ad Manager

OVERVIEW

The classified ad manager simulates a classified ad newspaper section, allowing clients to browse a master database of classified ads using a simple Web-based user interface. Clients browse any one of several categories according to keyword or other search parameters such as a price range or a post date. For example, a client might run a search for Ampeg Bass Amps priced less than $1,000 and posted within the last week.

The classified ad manager also allows clients to post, modify, and delete their own ads. Omitting the classified ad intermediaries from the equation, this application allows clients to directly modify their ads at any hour and as often as they want. By using the authentication algorithms discussed in Chapter 9, this application protects the integrity of every client's data by refusing to allow anyone except the poster to modify or delete the ad.

Together, ad searching and ad management create an environment in which clients buy, sell, and trade their wares with efficiency, privacy, and ease. The classified ad manager is also a fine example of how you might reconfigure the database management and searching algorithms discussed in Chapters 11 and 12 to handle other creative projects.

INSTALLATION AND USAGE

This script should be expanded into a directory from which the Web server is allowed to execute CGI scripts. Once unarchived, it will expand into the root directory **Classified_ad**. Figure 14.1 shows the directory structure along with a description of how permissions should be set graphically.

Classified_ad, the root directory, must have its permissions set to be readable and executable by the Web server. It contains two files (**class_ad.cgi** and **class_ad.setup**) and four subdirectories (**Databases**, **Images**, **Library**, and **Session_files**).

class_ad.cgi is the main script for the classified ad manager and should have its permissions set to be readable and executable by the Web server. The specifics of the script will be discussed in the design discussion.

class_ad.setup is the setup file that **class_ad.cgi** uses to gather server-specific information and obtain authentication options. It must have its permissions set to be readable by the Web server. It is discussed in greater detail in the "Server-Specific Setup and Options" section.

Databases is a subdirectory containing each of the classified ad databases and their associated setup files as well as the counter and user files. The **Databases** subdirectory must be readable writable, and executable by the Web server. The datafiles, user files, and counter file must be readable and writable by the Web server, and the setup files must be readable by the Web server.

class_ad.counter is a text file used to store unique classified ad database ID numbers. Initially, this file should contain the number 1 on the first line and nothing else. As time goes by, **class_ad.cgi** will increment this number by 1 for every new classified ad posted.

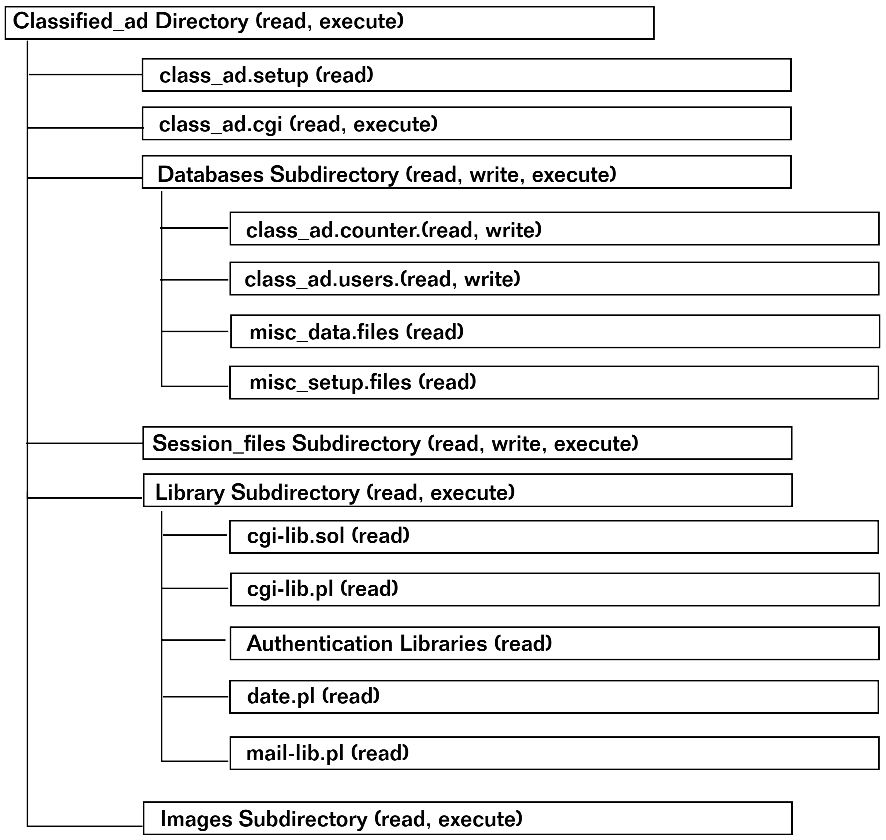

Classified_ad Directory (read, execute)
class_ad.setup (read)
class_ad.cgi (read, execute)
Databases Subdirectory (read, write, execute)
class_ad.counter.(read, write)
class_ad.users.(read, write)
misc_data.files (read)
misc_setup.files (read)
Session_files Subdirectory (read, write, execute)
Library Subdirectory (read, execute)
cgi-lib.sol (read)
cgi-lib.pl (read)
Authentication Libraries (read)
date.pl (read)
mail-lib.pl (read)
Images Subdirectory (read, execute)

Figure 14.1 *Directory structure of the classified ad manager.*

class_ad.users is the list of users who have been authorized to manipulate ads in the database. Chapter 9 includes an in-depth discussion of the use of the user file and the authentication libraries, so they will not be discussed here.

The datafiles and their associated setup files are discussed in greater detail in the "Server-Specific Setup and Options" section.

Images is a subdirectory containing the image map used for the example on the accompanying CD-ROM. The directory must be executable by the Web server, and any graphics within it must be readable.

Library is a subdirectory containing the CGI libraries (discussed in Chapter 9) that this script needs. The libraries used by this script include the following: **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**, **date.pl**, and **mail-lib.pl**. Each of these files must be readable by the Web server, and the directory itself must be readable and executable.

Session_files is the subdirectory used by the authentication libraries to store session files (as discussed in Chapter 9) as well as the lock file and temporary files used by the script during operation. Initially, this directory should be empty, but if it is made to be readable, writable, and executable by the Web server, the script will continually add to and prune this subdirectory as part of its daily usage.

NOTE

Never create the lock file or temporary file by yourself. As long as you configure the setup file correctly, the script will create and delete those files as it needs them.

Server-Specific Setup and Options

class_ad.setup is the setup file that **class_ad.cgi** uses to gather server-specific information and obtain authentication options. Within **class_ad.setup**, the following variables must be set to their server-specific values.

`$user_file` is the location of the file that contains the list of users who are authorized to use this script.

`$counter_file` is the path of the file that you are using to keep track of unique ID numbers. To make deletions and modifications, each item must have a unique ID number so that this script can determine which database item to delete. These ID numbers should always be the last field in any database row.

`$session_file_directory` is the location of the directory that temporarily holds session files. These session files are used to validate users and to keep track of their information should we need it.

`$database_manager_script` is the location of **class_ad.cgi**.

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

`$data_file` is the location of the flatfile ASCII text database that is being managed.

`$temp_file` is a file that **class_ad.cgi** uses to temporarily store various data at different times.

`$lock_file` is a file that **class_ad.cgi** uses to make sure that only one person can modify the database at any given time.

Authentication variables are defined and explained in Chapter 9, so they will not be discussed here.

The setup file included on the accompanying CD-ROM is shown next as an example of usage.

```
$counter_file = "./Databases/class_ad.counter";
$user_file = "./Databases/class_ad.users";
$temp_file = "./Session_files/class_ad.temp";
$lock_file = "./Session_files/class_ad.lockfile";
$session_file_directory = "./Session_files";
$database_manager_script = "./class_ad.cgi";
$database_manager_script_url = "class_ad.cgi";
$auth_lib = "$lib";
$auth_server = "off";
$auth_cgi = "on";
$auth_user_file = "./Databases/class_ad.users";
$auth_alt_user_file =  "";
$auth_default_group = "user";
$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 log on 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");
```

Each datafile follows the format discussed in Chapters 11 through 13, so we will not repeat the discussion here. The accompanying CD-ROM includes the following examples, which can be explored separately: **employment.data**, **housing.data**, **misc.data**, **personals.data**, and **vehicles.data**.

Every datafile comes with an accompanying setup file that defines each database's specific features. This chapter's example of a setup file explains the server-specific options that must be included with every datafile. The first example is from **vehicles.setup**.

`$data_file` is the location of the datafile associated with each setup file.

`$price` communicates whether price is one of the possible search parameters when a client searches this classified database. If this variable is set to `yes`, an extra input box will appear on search forms. The script also requires that you define `$price_field_num`.

`$price_field_num` is the location of the database field that contains the price information. It is essential that you provide the script with this location, because if you change the fields in the database, the script will have no idea which field should be compared to the client-defined price range. When defining this variable, remember that arrays start counting from zero.

`$date_field_num` is the array location of the date field. We need to identify this field to compare and search on date. When defining this variable, remember that array counting starts at zero.

`%FIELD_ARRAY` communicates the makeup of the database and specifies which fields are associated with which header and variable names. `@field_names` and `@field_values` are the ordered keys and values arrays for `%FIELD_ARRAY`.

@field_names_user_defined defines which fields the clients can submit when adding a new entry, whereas @field_names_non_user_defined defines which fields are supplied by the script.

As always, %FORM_COMPONENT_ARRAY describes which database fields are associated with which types of form input fields so that when we create forms for adding and searching, each database field will get an appropriate form input type.

Following is the text of **vehicles.setup** as it appears on the accompanying CD-ROM:

```
$data_file = "Databases/vehicles.data";
$price = "yes";
$price_field_num = "6";
$date_field_num = "8";
%FIELD_ARRAY = ( 'Last Name', 'last_name',
                 'First Name', 'first_name',
                 'Email', 'email',
                 'Phone Number', 'phone',
                 'Category', 'category',
                 'Location', 'location',
                 'Price', 'price',
                 'Your Ad', 'ad',
                 'Time', 'time',
                 'Id', 'id');
@field_names = ("Last Name", "First Name", "Email",
                "Phone Number", "Category", "Location",
                "Price", "Your Ad", "Time", "Id");

@field_names_user_defined = ("Category", "Phone Number",
                             "Location","Price",
                             "Your Ad");

@field_names_non_user_defined = ("Last Name",
                                 "First Name", "Email",
                                 "Time", "Id");

@field_values = ("last_name", "first_name", "email",
                 "phone", "category", "location",
                 "price", "ad", "time", "id");
```

```
%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"',
'Category', 'select|||Automobiles|Auto Parts|T
rucks|Vans|Motorcycles|4X4s|RVs|Mopeds|Water Craft|Air Craft|Other',
'Location', 'text|SIZE = "32" MAXLENGTH = "100"',
'Price', 'text|SIZE = "32" MAXLENGTH = "100"',
'Your Ad', 'textarea|ROWS = "4" COLS = "30"',
'Time', 'invisible',
'Id', 'invisible');
```

Running the Script

Once you have configured the setup files, created your own databases, and set the permissions, you can access the classified ad manager with a hyperlink such as this one:

```
<A HREF = "http://www.foobar.com/cgi-
bin/Classified_ad/class_ad.cgi">Classified Ad Manager</A>
```

Design Discussion

The logic of the classified ad manager is depicted in Figure 14.2.

As always, this script begins by calling on the Perl interpreter to print the HTTP header.

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

Loading the Supporting Libraries

Next, the `$lib` variable fixes the path of your current library.

```
$lib = "Library";
```

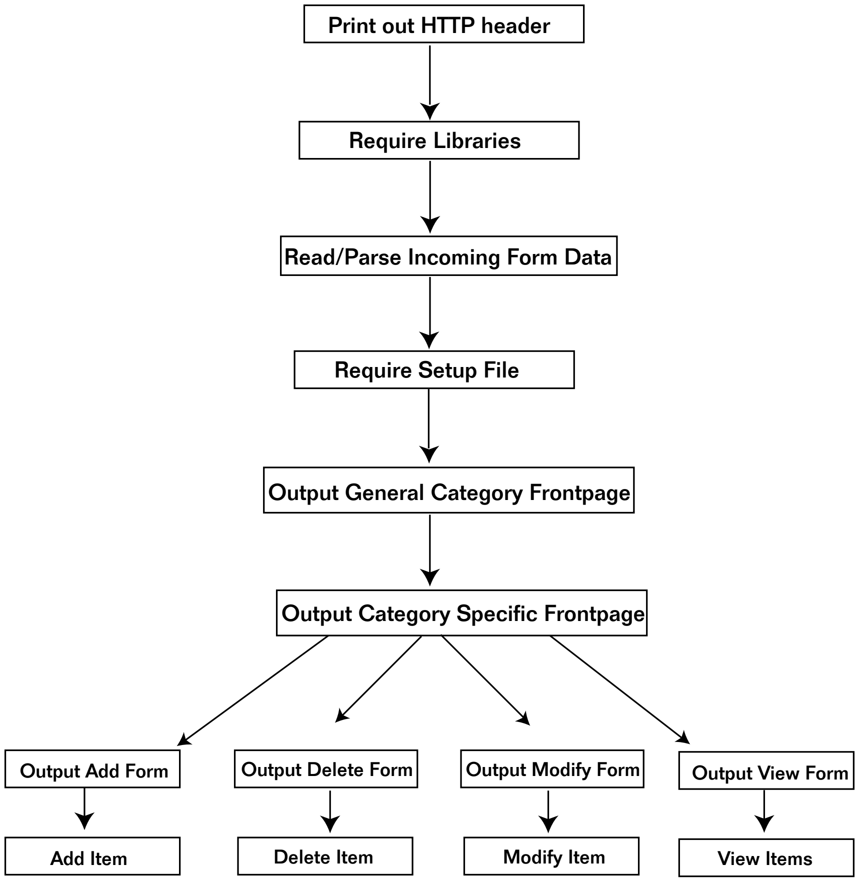


Figure 14.2 *Script logic for the classified ad manager.*

NOTE

By default, all library files used by this script have been placed in the Library subdirectory. Eventually, however, the best thing to do is to put them in your "real" CGI library and reference that path here.

Then the script adds the libraries to the beginning of the `@INC` array so that they will be read before any other libraries that may have routines with the same name.

```
unshift (@INC, "$lib");
```

At this point, **class_ad.cgi** `requires` the necessary files using `CgiRequire`, the subroutine at the end of this script. This subroutine is used so that if there is a problem with the `require`, the script will be able to send the client a meaningful error message.

```
&CgiRequire("$lib/cgi-lib.pl", "$lib/cgi-lib.sol",
            "$lib/auth-lib.pl", "./class_ad.setup",
            "$lib/date.pl");
```

Reading and Parsing Incoming Form Data

Next, the script uses **cgi-lib.pl** to parse the incoming form data, passing the subroutine `ReadParse (*form_data)` so that the variable will come out as `$form_data{'key'}` instead of `$in{'$key'}`.

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

Loading the Setup File

Once the form input has been parsed, the script determines which database the client is asking it to display. This script should have been called with the name of the datafile appended to the URL. For example, we may have linked to this script using the following hyperlink:

```
<A HREF = "class_ad.cgi?database=vehicles.setup">Vehicles Database</a>
```

In this example, the `%form_data` associative array contains the variable `database` with its associated value **vehicles.setup**. If there is such a value, the script assigns that to the variable `$setup_file`. If the value is empty and the person called this script without the parameter, the script assigns **basic.setup** to `$setup_file` instead.

```
if ($form_data{'database'} ne "")
  {
  $setup_file = $form_data{'database'};
  }
```

```
else
  {
  $setup_file = "basic.setup";
  }
```

Once the script has determined which setup file to use, it uses `CgiRequire` to `require` the setup file that it was asked for.

```
&CgiRequire("Databases/$setup_file");
```

Reformatting Variables

Next, the script reformats the name of the setup file so that it can display the name of the datafile in a user-friendly way on subsequent pages. If the user asked to see **vehicles.setup**, for example, the script should reformat the name to "Vehicles" so that it can later use the reformatted value to output something like, "Add an Item to the Vehicles Database" instead of "Add an Item to the vehicles.setup database."

So, if the script was given a database name, it first `splits` the name into a variable for the word *name* and a variable for the word *setup*. Thus, what was once **name.setup** becomes `name` and `setup`. Then the script assigns the first letter of the name to `$first_letter`, and the rest of the word to `$rest_of_the_word`. So `$first_letter` equals "n" and `$rest_of_the_word` equals "ame." Then the script turns the `$first_letter` into an uppercase letter using the translate (`tr`) function so that `$first_letter` now equals "N" instead of "n." Finally, the script splices the variables using (.). That's a lot of work for such a small change, but it makes the client GUI much nicer.

```
if ($form_data{'database'} ne "")
  {
  ($name, $junk) = split (/\./, $form_data{'database'});
  $first_letter = substr($name,0,1);
  $rest_of_the_word = substr($name,1);
  $first_letter =~ tr/a-z/A-Z/;
  $database = $first_letter . $rest_of_the_word;
  }
```

The script also defines the `$session_file` variable if one is coming in as form data. We will talk more about session file information later.

```
if ($form_data{'session_file'} ne "")
{
$session_file = $form_data{'session_file'};
}
```

Displaying the General Category Front Page

Now the script is ready to print the front page. This, however, will happen in only two cases. First, the script will output the front page if no database has been defined in the incoming form data (`$form_data{'database'} eq ""`) or (`||`) if the script is being asked specifically to return to the front page (`$form_data{'return_to_frontpage'} ne ""`). Second, the script outputs the front page if no values have yet been assigned to (`$ENV{'CONTENT_LENGTH'} eq ""`). If `CONTENT_LENGTH` is equal to zero, it means that this script is being accessed from an outside hyperlink rather from a script-generated HTML page.

```
if (($form_data{'database'} eq "" ||
     $form_data{'return_to_frontpage'} ne "")
     &&
    ($ENV{'CONTENT_LENGTH'} eq ""))||
($form.data{‘return.to.frontpage’}ne""))
{
```

The routine shown next prints the basic front page. However, notice that the image map hyperlinks reference this script with

```
?database=xxx&session_file=$session_file.
```

It’s important to always remember to pass this information so that the client does not get lost. As discussed in Chapter 9, the session file is used to maintain state. By passing the name of the session file from HTML page to HTML page, this script ensures that it will “remember” who the client is.

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

```
<HEAD>
<TITLE>The Classified Ad Manager</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<CENTER>
<IMG SRC = "/Graphics/class_ad_title.gif">
<P>
<A HREF = "/cgi-bin/imagemap/Graphics/classified.gif">
<IMG SRC = "/ Graphics/classified.gif"
 ISMAP USEMAP = "#map" BORDER = "0"></A>
<MAP NAME = "map">
<AREA COORDS = "11,18 191,53" HREF = "$datab
ase_manager_script_url?database=employment.setup&session_file=$ses-
sion_file">
<AREA COORDS = "9,
71 192,107" HREF =
"$database_manager_script_url?database=housing.setup&session_file=$ses
sion_file">
<AREA COORDS = "11
,124 191,160" HREF =
"$database_manager_script_url?database=misc.setup&session_file=$ses-
sion_file">
<AREA COORDS = "28
2,18 463,54" HREF =
"$database_manager_script_url?database=personals.setup&session_file=$s
ession_file">
<AREA COORDS = "28
4,72 463,107" HREF =
"$database_manager_script_url?database=vehicles.setup&session_file=$se
ssion_file">
<AREA COORDS = "28
4,125 465,160" HREF = "mailto:selena\@eff.org">4,125 </MAP>
<P>
<A HREF = "$database_manager_script_url?database=employment.setup&ses-
sion_file=$session_file">Employment</A>
| <A HREF = "$database_manager_script_url?database=housing.setup&ses-
sion_file=$session_file">Housing</A>
| <A HREF = "$database_manager_script_url?database=misc.setup&ses-
sion_file=$session_file">Misc. For Sale</A>
| <A HREF =
"$database_manager_script_url?database=personals.setup&session_file=$s
ession_file">Personals</A>
| <A HREF = "$database_manager_script_url?database=vehicles.setup&ses-
sion_file=$session_file">Vehicles</A>
</CENTER></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the general category front page looks like Figure 14.3.

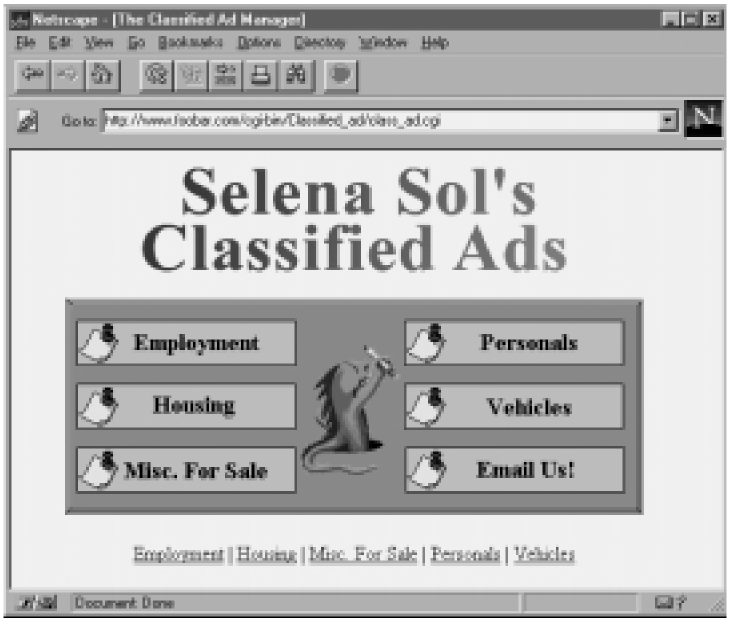

Netscape - [The Classified Ad Manager]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://www.foobar.com/cgi-bin/Classified_ad/class_ad.cgi
Selena Sol's Classified Ads
Employment
Personals
Housing
Vehicles
Misc. For Sale
Email Us!
Employment | Housing | Misc. For Sale | Personals | Vehicles
Document: Done

Figure 14.3 *General category front page.*

We have included the cheesy little graphic as an example in the Images subdirectory on the accompanying CD-ROM.

Displaying the Specific Category Front Page

If the client has clicked on one of the links from the front page image map, the client has submitted a request via `GET` rather than `POST` (which is used for all the forms throughout the rest of this script). If the request method is `GET`, the script knows that the client must be asking for the database-specific front page.

```
if ($ENV{'REQUEST_METHOD'} eq "GET")
{
```

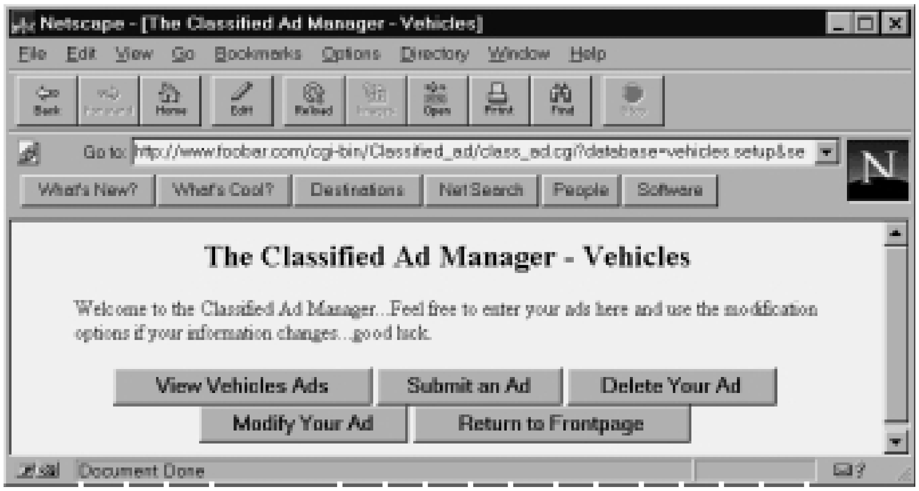
In response, the script prints the database-specific front page of the requested database, displaying several options.

```
print <<"end_of_html";
<HTML>
<HEAD>
<TITLE>The Classified Ad Manager - $database</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>The Classified Ad Manager - $database</H2>
</CENTER>
<BLOCKQUOTE>
Welcome to the Classified Ad Manager...Feel free to enter your ads
here and use the modification options if your information
changes...good luck.
</BLOCKQUOTE>
<FORM METHOD = "post" ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View $database Ads">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Submit an Ad">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete Your Ad">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify Your Ad">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
end_of_html
exit;
}
```

NOTE

Here is where we use the `$database` variable that we worked so hard to create. Also, notice that we are passing database and session IDs as hidden variables.

On the Web, the category-specific front page might look like Figure 14.4.


Netscape - [The Classified Ad Manager - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Go to: http://www.foobar.com/cgi-bin/Classified_ad/class_ad.cgi?database=vehicles.setup&se
What's New? What's Cool? Destinations Net Search People Software
View Vehicles Ads
Submit an Ad
Delete Your Ad
Modify Your Ad
Return to Frontpage
Document Done

The Classified Ad Manager - Vehicles

Welcome to the Classified Ad Manager...Feel free to enter your ads here and use the modification options if your information changes...good luck.

Figure 14.4 *Category-specific front page.*

Displaying the Classified Ad Add Form

Next, the script checks to see whether the client has asked for the form to add an entry to the database. If so, it logs the client in to be assigned a session ID number that the script can pass through the authentication routine if required. It is one thing to view the database; it is quite another to modify it!

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

To authenticate a client, the script passes the subroutine `GetSessionInfo`, which is contained in **auth-lib.pl**, three parameters: the `$session_file` value, which will be nothing if one has not yet been set, the name of this script so that it can provide links; and the associative array of form data we got from **cgi-lib.pl**. This process is discussed in greater depth in Chapter 9.

```
($session_file, $session_username, $session_group,
$session_first_name, $session_last_name, $session_email) =
&GetSessionInfo($session_file, $database_manager_script_url,
*form_data);
```

Because we also want to keep track of the date when new entries are made, the script also uses the `get_date` subroutine at the end of this script to get the current date.

```
&get_date;
```

Next, the script prints the header of the add form.

```
print <<"end_of_html";
<HTML><HEAD><TITLE>Add a Classified Ad - $database</TITLE></HEAD>
<BODY>
<CENTER><H2>Add a Classified Ad - $database</H2></CENTER>
<FORM METHOD = "post" ACTION = "$database_manager_script_url">
<CENTER>
end_of_html
```

It also creates an input form using the subroutine `create_input_form` at the end of this script. This subroutine creates an input field for each of the fields in the database and presents it in table format.

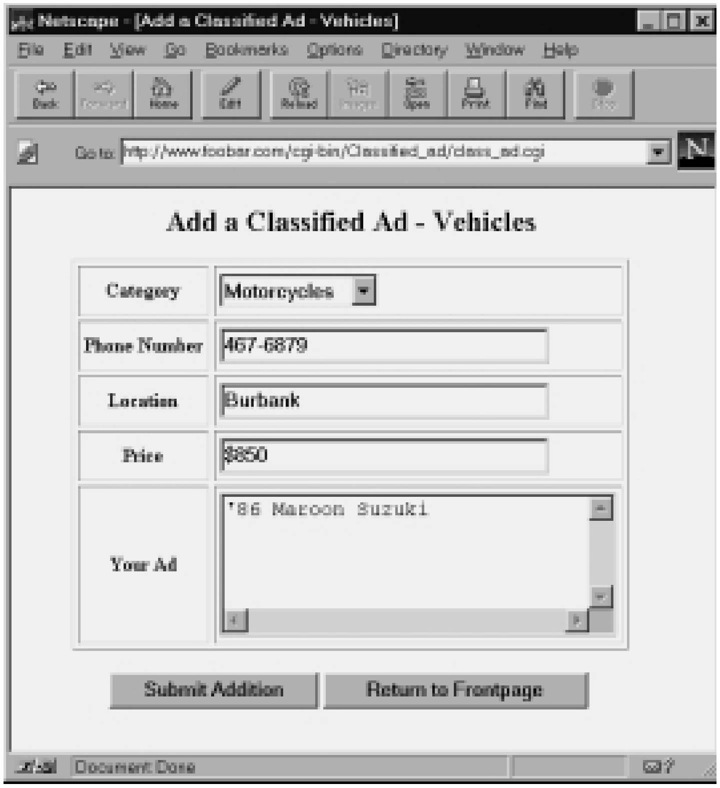
```
&create_input_form;
```

Finally, the script prints the page footers as it did for the front page.

```
print <<"end_of_html";
</TABLE><CENTER><P>
<INPUT TYPE = "hidden" NAME = "first_name"
       VALUE = "$session_first_name">
<INPUT TYPE = "hidden" NAME = "last_name"
       VALUE = "$session_last_name">
<INPUT TYPE = "hidden" NAME = "email"
       VALUE = "$session_email">
<INPUT TYPE = "hidden" NAME = "time" VALUE = "$date">
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<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;
}
```

Figure 14.5 shows the Add Form for the vehicles database that comes on the accompanying CD-ROM.


Netscape - [Add a Classified Ad - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Go to: http://www.foobar.com/cgi-bin/Classified_ad/class_ad.cgi
Submit Addition
Return to Frontpage
Document: Done

Add a Classified Ad - Vehicles

Category	Motorcycles
Phone Number	467-6879
Location	Burbank
Price	$850
Your Ad	'86 Maroon Suzuki

Figure 14.5 *The classified ads add form.*

Displaying the Classified Ad Delete Form

If the client wants to delete an ad, the script sends a form so that the client can specify which database item to delete.

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

First, the script passes the client through the security check using the `GetSessionInfo` subroutine in **cgi-lib.sol**.

```
($session_file, $session_username, $session_group,
$session_first_name, $session_last_name, $session_email) =
&GetSessionInfo($session_file, $database_manager_script_url,
*form_data);
```

Before it can begin deleting, however, the script must find out which item to delete. To do that, the script needs some information from the client—specifically, which item to delete—so it must be able to tell the client which items are available to delete. However, the script cannot simply output all the items in the database; the Web browser might run out of memory. Instead, the client gives the script one or more search terms so that it can put together a reasonably sized list from which the client can choose.

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

The script creates an input form using the subroutine `create_input_form` at the end of this script. The client uses this form to input keywords.

```
&create_input_form;
```

Then the script adds a form <input> tag for "exact match" and the page footer.

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

On the Web, the delete search form looks like the form shown in Figure 14.5 except for the delete-specific information.

Searching for Items to Delete

The script also needs a routine to accept the client-defined search term(s) and search the database, presenting a dynamically generated list of "hits."

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

The process begins with a security check.

```
($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 prints the page header.

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

Then the script begins searching the database by using the subroutine `search_database`, which can be found at the end of this script. The script passes to `search_database` one parameter, `delete`, so that the subroutine will know how to output the results of the search.

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

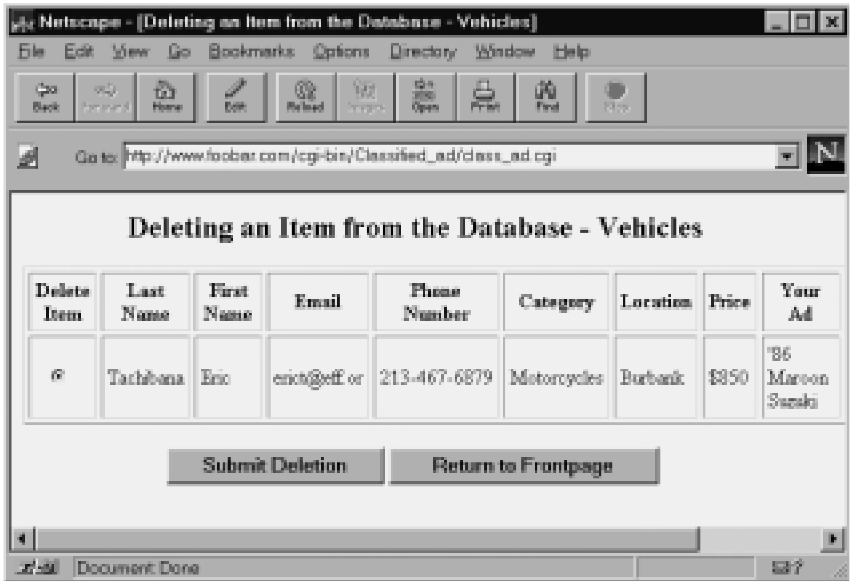
Finally, the script prints a list of hits in a table format. `$search_results`, returned from the subroutine `search_database`, contains all the hits in the form of table rows. The client then chooses which item to delete and deletes it.

```
print "<TABLE BORDER = \"1\" CELLPADDING = \"4\"
              CELLSPACING = \"4\">";
print &table_header ("Delete<BR>Item");
print &table_header (@field_names);
print "</TR>\n";
print "$search_results";

print <<"end_of_html";
</TABLE><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$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 deletion screen looks similar to Figure 14.6.


Netscape - [Deleting an Item from the Database - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Go to: http://www.foobar.com/cgi-bin/Classified_ad/class_ad.cgi
Submit Deletion
Return to Frontpage
Document: Done

Deleting an Item from the Database - Vehicles

Delete Item	Last Name	First Name	Email	Phone Number	Category	Location	Price	Your Ad
●	Tachibana	Eric	erict@eff.or	213-467-6879	Motorcycles	Burbank	$850	'86 Maroon Suzuki

Figure 14.6 *Deletion screen example.*

Displaying the Modify Item Form

Next, the script repeats the same process, this time for modification.

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

As usual, the client is authenticated first.

```
($session_file, $session_username, $session_group,
$session_first_name, $session_last_name, $session_email) =
&GetSessionInfo($session_file,
$database_manager_script_url, *form_data);
```

Then the usual header is printed.

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

Next, the script creates the input form using the subroutine `create_input_form` at the end of this script.

```
&create_input_form;
```

Finally, the script adds the HTML footer as it did for delete.

```
print <<"end_of_html";
<TH>Exact Match?</TH><TD>
<INPUT TYPE = "checkbox" NAME = "exact_match" CHECKED>
</TD></TR></TABLE><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$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 modify search form looks exactly like Figure 14.5 except for the modification-specific information in the header.

Searching the Database for Items to Modify

As it did for delete, the script prints the results of the query for modification after a security check.

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

First, the script prints the page header.

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

Then the script begins searching the database using the subroutine `search_database` at the end of the script as it did for delete.

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

Next, it prints a list of hits in a table format. You will recall that `$search_results`, returned from the subroutine `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>";
print &table_header (@field_names);
print "</TR>\n";
print "$search_results";
print "</TABLE><P>";
```

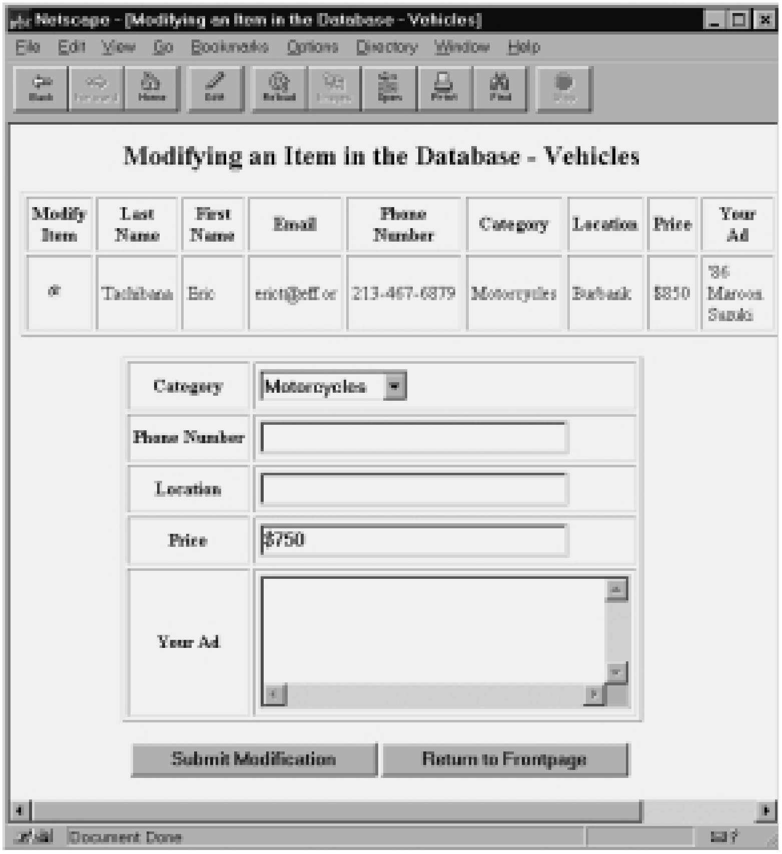
Furthermore, the script uses the subroutine `&create_input_form` to print the same form that was used for the "Add an Item" form. The client can now specify an item to modify in the table and then type new information to update the database fields.

```
&create_input_form;
```

Finally, the script prints the usual footer.

```
print <<"end_of_html";
</TABLE><CENTER><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$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 screen looks like Figure 14.7.


Netscape - [Modifying an Item in the Database - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Document Done

Modifying an Item in the Database - Vehicles

Modify Item	Last Name	First Name	Email	Phone Number	Category	Location	Price	Your Ad
●	Tachibana	Eric	erict@eff.or	213-467-6879	Motorcycles	Burbank	$850	'86 Maroon Suzuki

Category	Motorcycles
Phone Number	
Location	
Price	$750
Your Ad	

Submit Modification Return to Frontpage

Figure 14.7 *Modification screen example.*

Displaying the View Form

The next routines, beginning with the printing of the page header, allow a client to view the database.

```
if ($form_data{'search_form_view'} ne "")
{
print <<"end_of_html";
<HTML>
<HEAD>
<TITLE>Classified Ad Search Engine - $database</TITLE>
</HEAD>
```

```
<BODY>
<CENTER>
<H2>Classified Ad Search Engine - $database</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<TABLE BORDER = "1" CELLSPACING = "4"
       CELLPADDING = "4">
end_of_html
```

Next, the script creates an HTML form so that the client can submit keywords with which to search. For each field in the database—except for database ID and time fields, which are set by this script and not by the administrator—the script generates an input field. It gets the list of database fields from the `@field_names` array defined in the setup file.

However, to build the form, the script must send 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, SELECT). We don't want to send it the price or time of submission, though, because we will create new input boxes for these items based on maximum and minimum acceptable prices and posting date. Thus, these two fields must be removed from the array before it is passed to the `build_input_form` subroutine.

```
$id = pop (@field_names);
$time = pop (@field_names);
```

NOTE

Pay close attention to this if you are going to add other search features. Here is the first occurrence of a special customizing area.

Then the script uses `build_input_form` in **cgi-lib.sol** to create a form input field for every client-defined database field.

```
foreach $field_name (@field_names)
  {
  if ($field_name ne "Price")
    {
    if ($field_name ne "Time of Submission")
```

```
      {
 print &build_input_form("$FIELD_ARRAY{$field_name}",
 "$FORM_COMPONENT_ARRAY{$field_name}",
 $field_name);
      }
   } # End of if ($field_name ne "Price")
 } # End of foreach $field_name (@field_names)
```

Next, it adds the special new search boxes. In the setup file for each database is a variable called `$price`. If this variable is set to `yes`, it means that you want the client to be able to search this type of database by price.

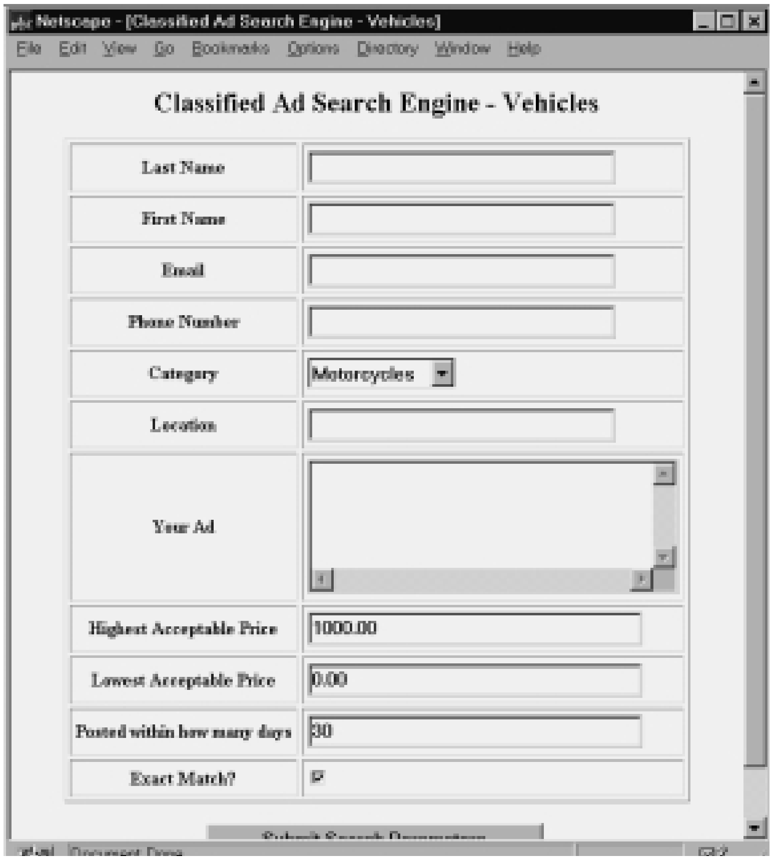
```
if ($price eq "yes")
{
print <<"end_of_html";
<TH>Highest Acceptable Price</TH>
<TD><INPUT TYPE = "text" NAME = "price.high"
           SIZE = "35" MAXLENGTH = "35">
</TD></TR><TR>
<TH>Lowest Acceptable Price</TH>
<TD><INPUT TYPE = "text" NAME = "price.low" SIZE = "35"
           MAXLENGTH = "35" VALUE = "0.00">
</TD></TR><TR>
end_of_html
}
```

Clients are also allowed to search by database row age. The script prints this second input field and finishes with a form input for exact match and the form, body, and ending HTML tags.

```
print <<"end_of_html";
<TH>Posted within how many days</TH>
<TD><INPUT TYPE = "text" NAME = "num_days_ago"
           SIZE = "35" MAXLENGTH = "35" VALUE = "30">
</TD></TR><TR>
<TH>Exact Match?</TH>
<TD><INPUT TYPE = "checkbox" NAME = "exact_match"
           CHECKED></TD>
</TR></TABLE><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "search_database_view"
```

```
       VALUE = "Submit Search Parameters">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the view form looks like Figure 14.8.


Netscape - [Classified Ad Search Engine - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Document Done

Classified Ad Search Engine - Vehicles

Last Name	
First Name	
Email	
Phone Number	
Category	Motorcycles
Location	
Your Ad	
Highest Acceptable Price	1000.00
Lowest Acceptable Price	0.00
Posted within how many days	30
Exact Match?	✓

Submit Search Parameters

Figure 14.8 *View form.*

Searching the Classified Ad Database for Items to View

Next, the script searches the database for items to view. All this is exactly the same as we've done before.

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

This time, 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 is to view the database.

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

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

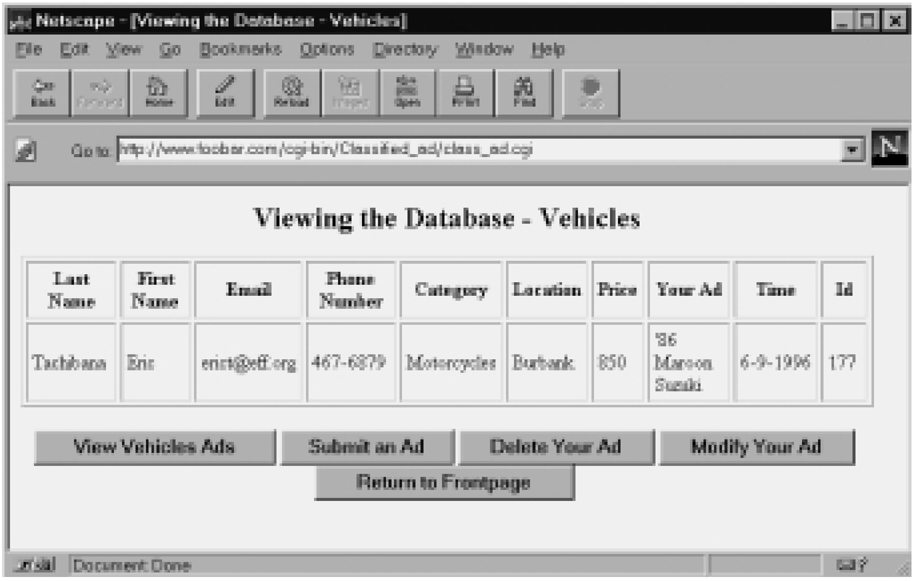
```
print <<"end_of_html";
<TABLE BORDER = "1" CELLPADDING = "4" CELLSPACING = "4">
end_of_html

print &table_header (@field_names);
print "</TR>\n";
print "$search_results";

print <<"end_of_html";
</TABLE><CENTER><P>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View $database Ads">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Submit an Ad">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete Your Ad">
<INPUT TYPE = "submit" NAME = "search_form_modify"
```

```
       VALUE = "Modify Your Ad">
<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 for a view might look like Figure 14.9.


Netscape - [Viewing the Database - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Go to: http://www.foobar.com/cgi-bin/Classified_ad/class_ad.cgi
View Vehicles Ads
Submit an Ad
Delete Your Ad
Modify Your Ad
Return to Frontpage
Document: Done

Viewing the Database - Vehicles

Last Name	First Name	Email	Phone Number	Category	Location	Price	Your Ad	Time	Id
Tachibana	Eric	erict@eff.org	467-6879	Motorcycles	Burbank	850	'86 Maroon Suzuki	6-9-1996	177

Figure 14.9 *Viewing the database.*

Adding an Item to the Classified Ad Database

The following routines direct the script when a client asks to add a new item to the database.

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

First, the client is passed through security.

```
($session_file, $session_username, $session_group,
$session_first_name, $session_last_name, $session_email) =
&GetSessionInfo($session_file, $database_manager_script_url,
*form_data);
```

In the case of an addition, the script assigns the new database entry a unique database ID number. It accesses the counter file, which keeps track of the last database ID number used. The `counter` subroutine, located in **cgi-lib.sol**, sends us a new number (incremented by 1) and then adjusts the counter file appropriately. The `counter` routine takes one parameter: the location of the counter file.

```
&counter($counter_file);
```

Then the ID value is slipped into the `%form_data` associative array so that the following routine will add it to the new database row along with all the other information.

```
$form_data{'id'} = "$item_number";
```

Next, the script formats the incoming form data the way the database is set up to understand (delimited by `|` and ending with a newline). It also makes sure to substitute new lines and line breaks with the corresponding HTML so that they will be displayed correctly.

The script also takes out any occurrences of `|`. If entered as part of the data, this pipe character would destroy our ability to read the database, because the script would interpret it as a field delimiter.

```
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;
```

Also, the script formats the price entered by the client so that it can later be compared numerically. The script removes any dollar signs ($) or commas (,) as well as any words such as “or best offer” (`[a-zA-Z]`) or spaces (`\s`).

```
if ($field eq "Price" ||
    $field eq "Cost (per month for rentals")
 {
 $form_data{$value} =~ s/\$//g;
 $form_data{$value} =~ s/[a-zA-Z]//g;
 $form_data{$value} =~ s/,//g;
 $form_data{$value} =~ s/\s//g;
 }
```

To improve the table presentation, the script changes any blank fields with `<CENTER>-</CENTER>` so that when the table is displayed, it won't have an ugly, empty-cell look.

```
if ($form_data{$value} eq "")
{
$form_data{$value} = "<CENTER>-</CENTER>";
}
```

Finally, the script creates and then appends `$new_row` with the value, thus creating a database row such as `field1|field2|field3|`. When all the fields have been spoken for, the script takes off the final pipe symbol and the database row is complete.

```
 $new_row .= "$form_data{$value}|";
} # End of foreach $field (@field_names)
chop $new_row; take out the last |
```

Next, the script creates a lock file so that it can edit the database file. If two people are trying to edit the locked datafile at one time, one person will not destroy the modifications made by the other person. The lock file is created using the subroutine `GetFileLock` in **cgi-lib.sol**, passing it one parameter: the location of the lock file used by this program.

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

Once the database is protected, the script opens it for appending (>>) and writes to it the value of `$new_row`.

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

```
print DATABASE "$new_row";
print DATABASE "\n";
```

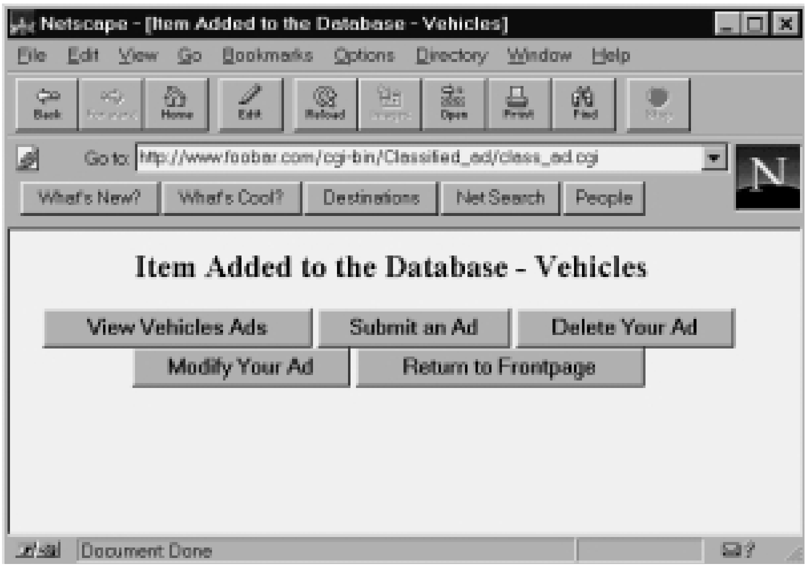
Then, after the change has been made, the script closes the database and deletes the lock file so that others may access the datafile.

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

Finally, the script sends a note telling the client that the item was added to the database.

```
print <<"end_of_html";
<HTML>
<HEAD>
<TITLE>Item Added to the Database - $database</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>Item Added to the Database - $database</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View $database Ads">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Submit an Ad">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete Your Ad">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify Your Ad">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

Figure 14.10 shows the client response on the Web.


Netscape - [Item Added to the Database - Vehicles]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Go to: http://www.foobar.com/cgi-bin/Classified_ad/class_ad.cgi
What's New? What's Cool? Destinations Net Search People
Document Done

Item Added to the Database - Vehicles

View Vehicles Ads | Submit an Ad | Delete Your Ad | Modify Your Ad | Return to Frontpage

Figure 14.10 *Client response for an addition.*

Deleting an Item from the Classified Ad Database

Next, the script checks to see whether it is being called on to make a deletion.

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

If it is, it passes the client through security.

```
($session_file, $session_username, $session_group,
$session_first_name, $session_last_name, $session_email) =
&GetSessionInfo($session_file, $database_manager_script_url,
*form_data);
```

Then it opens the database and reads it one line at a time.

```
open (DATABASE, "$data_file") ||
      &open_error($data_file);
while (<DATABASE>)
{
```

If the script finds a comment line, it adds it directly to `$new_data`, a variable used to store all the nondeleted database rows.

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

If it is not a comment row, however, the script `splits` the row by database fields and `pops` the item ID, which should be the last field in the row.

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

If the item ID does not match the one submitted by the client, the script adds the whole row to `$new_data`. If it is the same, however, the database row will not be added to `$new_data` and thus will be deleted by default.

```
 unless ($item_id eq "$form_data{'delete'}")
  {
  $new_data .= "$_";
  }
 } # End of else
} # End of while (<DATABASE>)
```

Next, the script closes the database, opens a lock file as it did for the add routine, and creates a temporary file that contains all the rows we dumped into `$new_data`.

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

It then copies the temporary file over the old database file, thereby deleting the entry, because it was not added to the temporary file. Then the script releases the lock file so that someone else can use it.

```
rename ($temp_file, $data_file);
&ReleaseFileLock ("$lock_file");
```

Finally, it prints the usual footer.

```
print <<"end_of_html";
<HTML>
<HEAD>
<TITLE>Your Item has been deleted - $database</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>Your Item has been deleted - $database</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View $database Ads">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Submit an Ad">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete Your Ad">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify Your Ad">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the client response looks like Figure 14.10 except for the delete-specific header.

Modifying an Item in the Database

Last, but not least, the script takes care of any modifications asked of it.

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

As usual, the script begins with a security check.

```
($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 makes sure that the client clicked one of the radio buttons on the modification table; without that information, it would not be able to know which item to modify. `$form_data{'modify'}` will be equal to the database ID number of the item if one of the radio buttons has been selected.

```
if ($form_data{'modify'} eq "")
{
print <<"end_of_html";
<HTML>
<HEAD>
<TITLE>Modifying an Item in the Database -
$database</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>Modifying an Item in the Database - $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 so 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;
}
```

Next, the script opens the database and reads through it one line at a time, adding the comment lines to `$new_data`.

```
open (DATABASE, "$data_file") ||
      &open_error($data_file);
while (<DATABASE>)
  {
```

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

If the line is not a comment line, however, the script `pops` the `item_id` as it did for deletion. But this time, it also `pushes` the `item_id` back into the array after it has read it, because we still want to have the ID in the row when the routine is finished. We'll need the complete row for the final modified row.

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

If the item ID from the database equals the one submitted by the client, the script creates a new array, `@old_fields`, equal to the current fields in the row.

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

If the two values are not equal, however, the script adds the row to `$new_data`. By the end, the script will have copied every line in the database to `$new_data` except for the item that the client wanted modified. Fortunately, the script saved that line in the `@old_fields` array.

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

To add the modified line to the database, the script begins by initializing a few variables. `$new_line` will contain the modified database row, and `$counter` will be used to count database fields in the row.

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

Then the script goes through each of the fields in `@field_values`, which is a list containing each of the database fields as defined in the setup file.

```
until ($counter >= @field_values)
{
```

For each field, the script initializes the variable `$value` and sets it equal to an incremented field according to the current value of `counter`.

```
$value = "";
$value = "$field_values[$counter]";
```

If that field, as represented by the form input, is equal to zero (the client did not wish to modify that field), the script takes the old value (as stored in the `@old_fields` array) and adds it to `$new_line`, appending a pipe (|) at the end to denote the end of the field.

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

On the other hand, if the user wanted to edit that field, the script formats the incoming data as it did for the add routine and adds the new data 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 ($field eq "Price" ||
     $field eq "Cost (per month for rentals")
  {
  $form_data{$value} =~ s/\$//g;
  $form_data{$value} =~ s/[a-zA-Z]//g;
  $form_data{$value} =~ s/,//g;
  $form_data{$value} =~ s/\s//g;
  }

 if ($form_data{$value} eq "")
  {
  $form_data{$value} = "<CENTER>-</CENTER>";
  }

 $new_line .= "$form_data{$value}|";
 } # End of else
$counter++;
} # End of until ($counter >= @field_values)
chop $new_line; Take off the final |
```

Next, the script opens the temporary file and prints the `$new_data` lines as well as the modified database line contained in `$new_line`. Then it closes the temporary file, copies the temporary file over the old database file, and releases the lock.

```
&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 footer.

```
print <<"end_of_html";
<HTML>
<HEAD>
<TITLE>Your Item has been Modified - $database</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>Your Item has been Modified - $database</H2>
</CENTER>
<FORM METHOD = "post"
      ACTION = "$database_manager_script_url">
<CENTER>
<INPUT TYPE = "hidden" NAME = "database"
       VALUE = "$form_data{'database'}">
<INPUT TYPE = "hidden" NAME = "session_file"
```

```
       VALUE = "$session_file">
<INPUT TYPE = "submit" NAME = "search_form_view"
       VALUE = "View $database Ads">
<INPUT TYPE = "submit" NAME = "add_form"
       VALUE = "Submit an Ad">
<INPUT TYPE = "submit" NAME = "search_form_delete"
       VALUE = "Delete Your Ad">
<INPUT TYPE = "submit" NAME = "search_form_modify"
       VALUE = "Modify Your Ad">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
} # End of if ($form_data{'submit_modification'} ne "")
```

On the Web, the client response looks like Figure 14.10 except for the modify-specific header.

The CgiRequire Subroutine

`CgiRequire` checks to see whether the file that the script is trying to `require` exists and is readable by it. This subroutine provides developers with an informative error message when they're attempting to debug the scripts.

```
sub CgiRequire
{
```

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

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

`CgiRequire` then checks to see whether the files exist and are readable by the script. If so, the files are `required`.

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

If there is a problem, however, `CgiRequire` sends an error message that will help the developer isolate 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 $file are set to allow me
  access?";
  exit;
  }
 } # End of foreach $file (@require_files)
} # End of sub require
```

The create_input_form Subroutine

The `create_input_form` subroutine is used to generate the input forms that the client will use to input new data for an addition or keywords for a search.

```
sub create_input_form
{
```

First, the table header is printed.

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

Then the subroutine is used to create an HTML form. In the end, we must have one input field for each field in the database (except the database ID field, which is set by this script and not by the client).

The subroutine gets the list of database fields from the `@field_names` array defined in the setup file and, for every element in the array, creates an input field. To do so, it sends the `build_input_form` subroutine in **cgi-lib.sol** a few parameters: 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, SELECT).

```
foreach $field_name (@field_names_user_defined)
 {
```

```
 $variable_name = $FIELD_ARRAY{$field_name};
 $form_type = $FORM_COMPONENT_ARRAY{$field_name};
 local ($input_form);
 $input_form = "";
 $input_form .= "<TR>\n";
 $input_form .= &table_header ("$field_name");
 $input_form .= "<TD>";
 $input_form .= &make_form_row ("$field_name",
                               "$variable_name",
                               "$form_type");
 $input_form .= "</TD></TR>\n";
 print"$input_form";
 }
} # End of sub create_input_form
```

The search_database Subroutine

The `search_database` subroutine is used to search the database for keyword matches.

```
sub search_database
{
```

First, the local variable `$submit_type` is set equal to the value sent to us from the main routine. This will be either `modify`, `delete`, or, in the case of a view search, `none`.

```
local($submit_type) = @_;
```

Next, the subroutine opens the database file and begins checking each field in every row against the keywords submitted, disregarding comment lines (`COMMENT:`).

```
open (DATABASE, "$data_file")||
      &open_error($data_file);
while (<DATABASE>)
 {
 $database_row = $_;
 unless ($database_row =~ /^COMMENT:/)
  {
```

$did_we_find_a_match is initially set equal to no to make sure that it has not been initialized elsewhere. The $did_we_find_a_match 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 yes.

```
$did_we_find_a_match = "no";
```

Next, the subroutine splits the database row into the @row array and creates some variables based on values specified in the setup file.

```
@row = split(/\|/,$database_row);
$row_price = "$row[$price_field_num]";
$row_date = "$row[$date_field_num]";
$last_name_field_number = "0";
$row_last_name = "$row[$last_name_field_number]";
($month,$day,$year) =
    split (/-/, $row[$date_field_num]);
```

It then uses the **date.pl** library to convert the row date into a Julian date that it uses to compare to today's date and generate the number of days ago that the row was posted.

```
$julian_day = &jday($month,$day,$year);
($today_month,$today_day,$today_year) = split (/-/,
       &get_date);
$today = &jday($today_month,$today_day,$today_year);
$posted_days_ago = ($today - $julian_day);
```

Next, search_database checks to see whether the client asked us to weed out by price or age. By the way, the if tests will pass if the user did not submit any pruning value.

```
if (($row_price <= $form_data{'price.high'} ||
     $form_data{'price.high'} eq "")
&&
($row_price >= $form_data{'price.low'} ||
$form_data{'price.low'} eq "")
&&
($form_data{'num_days_ago'} >= $posted_days_ago ||
      $form_data{'num_days_ago'} eq ""))
{
```

Then, for each key in the `%form_data` associative array, the script sets `$field_number` equal to –1. We do this because both the pesky “submit” and “exact match” key/value pairs may come in along with the rest of the form data. We do not want the script to search the database for those fields, because they don’t exist! By setting `$field_number` equal to –1, the script will be able to filter such non–field keys with the following routine:

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

The subroutine first 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 script offsets 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 = $y - 1;
  last; # Exit out of the for loop because we have
        # verified field
  }
} # End of for ($y = 1; $y <= @field_values; $y++)
```

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

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

Next, the script performs the match, checking the database information against the submitted keyword for that field. First, it performs an exact match test. If `$form_data{'exact_match'}` is not equal to `on`, it means that the exact match check box was not checked. The match is straightforward. If the keyword string (`$form_data{"$form_data_key"}`) matches (`=~`) a string in the field that the subroutine is searching (`$row[$field_number]`) with case insensitivity off (`/i`), then the script knows it got a hit.

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

However, before we get too excited, the script must make sure either that the client is an administrator (and is allowed to see all rows) or that the row the client got a hit on is actually a row that the client initially entered; it also must make sure that this is not a general view request. The first two `if` tests are obvious. The reason for the last test is that if the client is asking to view, we don't care whether the client entered the row. We want him or her to see everything. Only in the case of modifying and deleting do we want the extra level of filtering.

```
if (($session_group eq "admin" ||
     $row_last_name eq "$session_last_name") &&
     $submit_type ne "none")
  {
  $did_we_find_a_match = "yes";
  last; # Exit out of ForEach keys in FormData
}
```

The subroutine also handles a general view request. It is basically the same routine, but it covers whatever was left by the previous `if` test.

```
 if ($submit_type eq "none")
  {
  $did_we_find_a_match = "yes";
  last; # Exit out of ForEach keys in FormData
  }
 } # End of if (@row[$field_number]....
} # End of if ($form_data{'exact_match'} eq "")
```

On the other hand, the client may have clicked the exact match check box. This time, the script proceeds with an exact match using the `\b` switch, keeping it case-insensitive (`/i`). The same `$did_we_find_a_match` setting and `if` tests apply as before.

```
elsif ($row[$field_number] =~
/\b$form_data{$form_data_key}\b/i)
  {
  if (($session_group eq "admin" ||
       $row_last_name eq "$session_last_name") &&
       $submit_type ne "none")
   {
   $did_we_find_a_match = "yes";
   last; # Exit out of ForEach keys in FormData
   }
 if ($submit_type eq "none")
   {
   $did_we_find_a_match = "yes";
   last; # Exit out of ForEach keys in FormData
   }
  } # End of elsif ($row[$field_number]....
 } # End of if ($field_number > -1)
} # End of foreach $form_data_key (keys %form_data)
```

If the script finds a match (`$did_we_find_a_match` equals yes), it needs to create a table row for the output. `$search_results` is used to collect all the hits formatted as table rows for the output. A hit adds each of the fields of the database row to a table row. The subroutine also creates a variable called `$hit_counter` to remind it that it got a hit. If `$hit_counter` is never set to 1, the script knows that it must tell the client that her keyword turned up nothing.

```
if ($did_we_find_a_match eq "yes")
{
$search_results .= "<TR>";
$hit_counter = "1";
```

Then the subroutine gathers the database ID row number so that it can use it and then puts it back into the `@row` array.

```
$db_id_number = pop (@row);
push (@row, $db_id_number);
```

The subroutine then begins creating the database rows created by the HTML. If this is not a view and if it is a row that satisfies security, the script creates a first column for the radio button that the client will use to select a database row to modify or delete.

```
if (($session_group eq "admin" ||
     $row_last_name eq "$session_last_name")
&&
($submit_type ne "none"))
 {
 $search_results .= "<TD ALIGN = \"center\">\n";
 $search_results .= "<INPUT TYPE = \"radio\"
                 NAME = \"$submit_type\"
                VALUE = \"$db_id_number\">";
 $search_results .= "\n</TD>\n";
} # End of if ($session_group eq "admin" ||....
```

Then the subroutine fills in the HTML database table row. In the case of viewing, it adds every row, and in the case of modification and deletion, it gives them only the appropriate rows.

```
foreach $field (@row)
  {
  if ($submit_type eq "none")
    {
    $search_results .= "<TD>$field</TD>\n";
    }
    elsif ($row_last_name eq "$session_last_name" ||
           $session_group eq "admin")
    {
    $search_results .= "<TD>$field</TD>\n";
    }
    } # End of foreach $field (@row)...
   } # End of if ($did_we_find_a_match eq "yes")....
  } # End of if (($row_price <=....
 } # End of unless ($database_row =~ /^COMMENT:/)
} # End of while (<DATABASE>)
close (DATABASE);
```

Next, the subroutine provides an algorithm to handle 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 script sends a note to the client with a link to the search form.

N O T E

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

Also, the script exits the routine here if no hits were found. We do not want it to print an empty table.

```
if ($hit_counter ne "1")
 {
 print "I'm sorry, I was unable to find a match for the
        keyword(s) that you specified in a database row
        that you are authorized to see. Feel free to
        <A HREF = \"class_ad.cgi\">try again</A>";
 print "</CENTER></BODY></HTML>";
 exit;
 }
} # End of sub search_database
```

The get_date Subroutine

The get_date subroutine is used to get the current date of each added classified ad database row.

```
sub get_date
 {

 @days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday',
          'Thursday', 'Friday', 'Saturday');
 @months = ('January', 'February', 'March', 'April',
            'May', 'June', 'July', 'August',
            'September',  'October',  'November',
            'December');
```

The localtime command is used to get the current time, splitting it into variables.

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
localtime(time);
```

Then the variables are formatted and assigned to the final `$date` variable.

```
if ($hour < 10) { $hour = "0$hour"; }
if ($min < 10) { $min = "0$min"; }
if ($sec < 10) { $sec = "0$sec"; }
$date = "$mon-$mday-19$year";
}
```