CHAPTER 12

The Database Search Engine

OVERVIEW

Another useful database function you can provide through the Web is a search engine that allows users to search a shared database by keyword but doesn't give them the power of a larger database manager script. Database administrators can provide a database management interface for themselves and a corresponding database search interface for other people in the company. Users are spared the confusion of having too many options (most of which they're not allowed to use anyway), and the sensitive database management functions are hidden behind yet another level of security.

The database search engine provides a client with text input boxes for each of the searchable fields in the database. The client types one or more keywords in any of the input boxes, submits the information to the CGI script, and receives from the script a list of database rows that match the keywords.

INSTALLATION AND USAGE

Taking advantage of multiple files and directories, this application is best installed by expanding it into the default directory structure with the root **Database_search**. Figure 12.1 outlines the directory structure and the permissions needed for the search to operate.

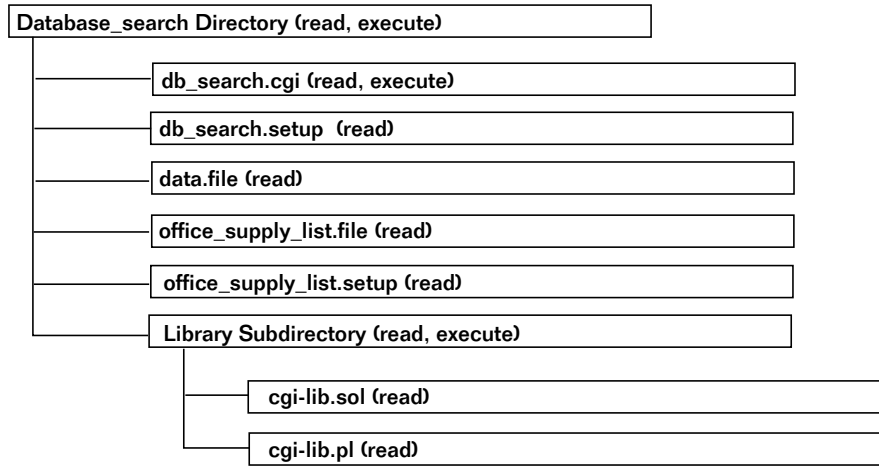

Figure 12.1 *Directory layout for the database search application.*

The root **Database_manager** directory must have permissions that allow the Web server to read and execute and should contain one directory (**Library**) and five text files (**data.file**, **db_search.cgi**, **db_search.setup**, **office_supply_list.file**, and **office_supply_list.setup**).

db_search.cgi is the application that searches the assigned databases. This file, which should be executable and readable, will be explained in detail in the design discussion.

data.file is the default database that is searched by **db_search.cgi**, and **office_supply_list.file** is an example "secondary" database file that can also be searched with the same **db_search.cgi**. These files must be readable by the Web server so that they can be searched.

db_search.setup and **office_supply_list.setup** are the setup files that communicate to **db_search.cgi** the specific properties of the related datafiles. These files must be readable by the Web server and are discussed in greater detail later.

Library is a subdirectory containing a couple of supporting library files. The library files used for this script are **cgi-lib.pl** and **cgi-lib.sol** (discussed in Part Two). Needless to say, this subdirectory must be executable and readable by the Web server, and its files must be readable.

Server-Specific Setup and Options

The Setup File

As with the setup file for **db_manager.cgi**, there are two functions performed by the setup file for **db_search.cgi**: defining server-specific variables and determining the names of database fields and their relative display orders. Each **setup** file sets the following variables.

`$database_search_script` is the location of **db_search.cgi** on your local system.

`$database_search_script_url` is the URL of **db_search.cgi**.

`$data_file` is the location of the default datafile. In some cases, this datafile will be located in a separate directory (such as the directory containing the databases for **db_manager.cgi**), but the accompanying disk contains a sample datafile for the purposes of explanation, and we have referenced this variable to that file. In the case of **office_supply_list.setup**, notice that `$data_file` points to the appropriate data file.

`$max_hits` specifies the number of hits that should be displayed at any one time. We set this variable to an amount that most browsers will have enough memory to handle.

`%FIELD_ARRAY` is an associative array that we use 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 array handling, if we used the line

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

the field names would not come up in the order that we chose to assign them in the database. Thus, we must manually assign 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 our database fields.

`%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 the size and multiple options—are also separated by pipes. The details can be found in Chapter 10.

As an example, here is the complete text of **db_search.setup**, the default setup file.

```
#!/usr/local/bin/perl
$database_search_script = “./db_search.cgi";
$database_search_script_url = “db_search.cgi";
$data_file = “./data.file";
$max_hits = “5";
%FIELD_ARRAY = ( 'Last Name', 'last_name',
                 'First Name', 'first_name',
                 'Email', 'email',
                 'Phone Number', 'phone',
                 'Address', 'address',
                 'Id', 'id');
@field_names = (“Last Name", “First Name", “Email",
           “Phone Number", “Address", “Id");
@field_values = (“last_name", “first_name", “email",
           “phone", “address", “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"',
'Address', 'textarea|ROWS = “4" COLS = “30"',
'Id', 'invisible');
```

The Data File

Every datafile should be a pipe-delimited text database and should be readable by the Web server. The text of **office_supply_list.file** is as follows:

```
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
```

As you can see, this datafile is formatted exactly the same as the datafiles used by the **db_manager.cgi** script discussed in Chapter 11. This is because both scripts are meant to work together. For more information about the datafile, see the design discussion in Chapter 11.

Running the Script

Once you have configured the setup file to the specifics of your server setup, you can test the search on the sample datafile. To access the script, create a hyperlink to the location of the main script as follows:

```
<A HREF = “http://www.foobar.com/cgi-
bin/Database_search/db_search.cgi">Database Search</A>
```

When you click on this link, you will run **db_search.cgi**, which will take advantage of the default setup file. Alternatively, you can use this same script to search another database on the system by using the following format:

```
http://www.foobar.com/cgi-
bin/Search/db_search.cgi?database=xxx
```

The xxx is the name of the alternative database to use, such as **office_supply_list**.

Design Discussion

Figure 12.2 summarizes the logic of the script as it responds to the demands of the client.

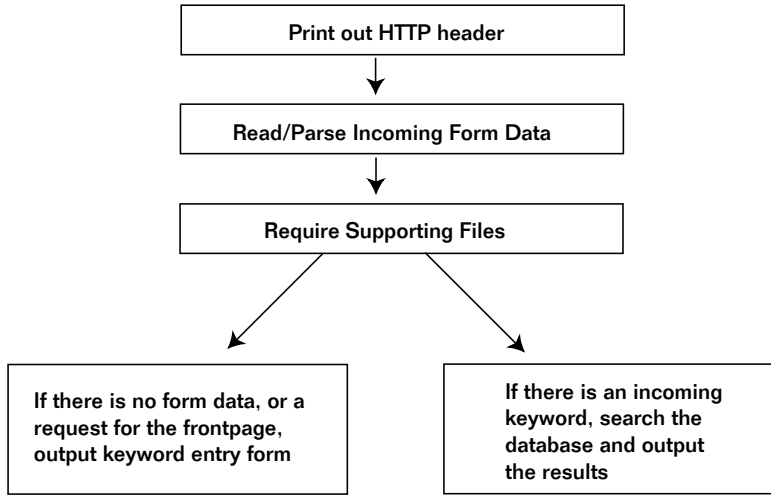

Figure 12.2 *The script logic.*

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 Library Files

Then the script gathers the necessary supporting library files. First, the script makes sure that the library is in the Web server's path and then it `loads` each library that it will use. To do so, the script adds the library to the array of directories (`@INC`) known to the Perl interpreter as valid libraries. It adds the directory to the beginning of the array using `unshift`; in case there is another library in another directory with the same name as ours, Perl will choose the one in **Library** first.

```
$library_path = "./Library";
unshift (@INC, "$library_path");
```

Next, the necessary library files are loaded using `CgiRequire`, the subroutine at the end of this script. If there is a problem with the `require`, a meaningful error message will be sent to the Web browser.

```
&CgiRequire("$library_path/cgi-lib.pl",
         "$library_path/cgi-lib.sol");
```

If there is an error, the most likely cause is a typo in the path location of the file to be `required`.

Reading and Parsing Incoming Form Data

Now **cgi-lib.pl** is used to parse the incoming form data. The script passes `*form_data` to **cgi-lib.pl** so that the associative array returned will come out as `%form_data` instead of `%in`.

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

Loading the Setup File

Once it has loaded the basic set of libraries, the script determines which database it should use. The determination depends on client-provided information coming in as url-encoded data. By default, the requested database is the main database:

```
http://www.foobar.com/cgi-bin/Search/db_search.cgi
```

Or the client may have asked to use this script to search a different database on the system:

```
http://www.foobar.com/cgi-
bin/Search/db_search.cgi?database=xxx
```

The following routine determines whether the requested database is other than the default one:

```
if ($form_data{'database'} ne “”)
 {
 $database_to_use = “$form_data{'database'}";
 }
else
 {
 $database_to_use = “db_search"; # Default setup file
 }
```

Once the script determines which database to search, it loads the associated setup file.

```
  &CgiRequire(“$database_to_use.setup");
```

Displaying the Database Search Form

Next, the script checks whether the client is asking for the form used to submit a keyword. There are two cases in which this test will return `true`: first, the client may have accessed this script for the first time, having “submitted” no information; content length should equal nothing. Second, the client may have pressed a **submit** button asking to return to the front page. The next line asks for either case 1 or (||) case 2.

```
if ($ENV{'CONTENT_LENGTH'} eq “” ||
$form_data{'return_to_frontpage'} ne “”)
  {
```

If this `if` test returns `true`, the script knows that it is being asked to display the front page. First, the script outputs the HTML page header.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Database Search Engine</TITLE>
</HEAD><BODY>
<CENTER>
<H2>Database Search Engine Front Page</H2>
</CENTER>
<FORM METHOD = “post"
```

```
      ACTION = “$database_search_script">
<CENTER>
<TABLE BORDER = “1" CELLSPACING = “4" CELLPADDING = “4">
end_of_html
```

Next, the script generates an HTML form so that the client can submit keywords. This HTML form has one input field for each field in the database except for the database ID field, which is an administrative field.

The script gets the list of database fields from the `@field_names` array and, for every element in the array, creates an input field. To create the HTML form, the script sends the `build_input_form` subroutine in **cgi-lib.sol** three 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);
      }
```

At the end of the form, the script tacks on a form input for exact match and the page 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 = “$database_to_use">
<INPUT TYPE = “submit"  NAME = “search_database"
       VALUE = “Submit Search Term">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
} # End of if ($ENV{'CONTENT_LENGTH'} eq “" ......
```

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 non-exact search will find the item.

Figure 12.3 shows what the query form looks like on the Web.

Figure 12.3 *The query form.*

Searching the Database for Client-Submitted Keywords

If the script gets past the previous `if` test, it means that the client has submitted a keyword. The script conducts the search and prints the results. As usual, the process begins by printing the page header.

```
print <<"    end_of_html";
<HTML><HEAD>
<TITLE>Database Search Engine Results</TITLE>
</HEAD><BODY>
```

```
<CENTER><H2>Database Search Engine Results</H2></CENTER>
<FORM METHOD = “post"
      ACTION = “$database_search_script">
<INPUT TYPE = “hidden" NAME = “database"
      VALUE = “$database_to_use">
<CENTER>
end_of_html
```

Next, the script searches the database. Searching begins by opening the database file and checking each field in every row against the keywords submitted.

The script also assigns to the variable `$initial_hits` the value of `$number_of_hits`, which may or may not be coming in as form data. The script uses this variable to keep track of how many screens' worth of search results the user has seen. So that we do not overwhelm the Web browser, this script has been built to display no more than a specific amount of hits at one time. In the distribution setup file, we have set this amount to 5.

```
$initial_hits = $form_data{'number_of_hits'};
open (DATABASE, “$data_file");
while (<DATABASE>)
  {
  $database_row = $_;
```

Then the script sets the variable `$not_found_flag` equal to zero to make sure that it has not been initialized elsewhere. The `$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. If, at the end of all the searching, `$not_found_flag` is still equal to zero, we tell the client the bad news.

```
$not_found_flag = “0";
```

The script must also disregard any line that is a database comment line. If you have not read Chapter 11 and don't understand this, it's a good idea to jump back and check out the description there.

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

Then the script splits the current database row into the @row array, assigning elements for every occurrence of a pipe character.

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

We escape the pipe character, because it has meaning other than as a delimiter for our database.

Next, 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;
```

Now the script 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 its 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
```

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

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

Then the script performs the match, checking the database information against the submitted keyword for that field. It performs an exact match test if requested to do so. If `$form_data{'exact_match'}` is equal to nothing, it means that the exact match check box was not checked. The match is straightforward. If the keyword string (`$form_data{"$form_data_key"}`) matches (`=~`) a string in the field that we are searching (`@row[$field_number]`) case insensitively (`/i`), then 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 script sets `$not_found_flag` equal to 1. In the end, if it gets through all the fields and has still 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 script proceeds with an exact match using the \b switch to delimit the keyword absolutely while still matching it with case sensitivity off (/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 the script finds a match ($not_found_flag still equals 0), it must create a table row for the output. The variable $search_results collects all the hits and formats them as table rows for the output. A hit adds each of the fields of the database row to a table row.

However, because the script may be working with a huge database having thousands of rows, we must make sure that the client-submitted keyword does not overwhelm the client's browser or our own server's memory. Thus, for every hit, the script increments $number_of_hits by 1. In addition, the script sets $adjusted_max_hits to $initial_hits plus $max_hits. That way, $adjusted_max_hits takes into account rows that have been seen by the client, who will have pressed the **submit** button to see the next group of hits. Then, provided it receives these values as incoming form data, the script can ignore all hits whose values are less than the number of hits already seen by the client. It can also ignore those greater than the number of $max_hits defined in the setup file plus the $initial_hits already seen by the client.

```
if ($not_found_flag eq "0")
  {
  $number_of_hits++;
  $adjusted_max_hits = $initial_hits + $max_hits;
  if ($number_of_hits > $initial_hits &&
      $number_of_hits <= $adjusted_max_hits)
    {
```

If the row is within the boundary, the script outputs the row to the Web-based table.

```
    $search_results .= “<TR>";
    $hit_counter = “1";
    foreach $field (@row)
      {
      $search_results .= “<TD>$field</TD>";
      }
    $search_results .= “</TR>";
    }
```

If, on the other hand, the script finds too many hits, we stop the printing of table rows and let the user choose to see more if desired.

```
   else
    {
    $search_notice .= “This search engine will only
          return $max_hits hits, and your
     search turned up more than that.
Would you please refine your
search?";
```

If it has displayed all the data, the script breaks out of the loop.

```
     if ($number_of_hits eq “$adjusted_max_hits")
       {
       last;
       }
     }
 } # End of unless ($database_row =~ /^COMMENT:/)
} # End of while (<DATABASE>), try the next row...
close (DATABASE);
```

But what if the client-submitted keyword turns up nothing in the search? At this point, if `$not_found_flag` is still equal to zero, it means that the search did not turn up a hit. Thus, the script must be prepared to send a note to the client with a link to the search form and exit.

```
if ($not_found_flag eq “0")
  {
```

```
  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-search.cgi\">try again</A>";
  print "</CENTER></BODY></HTML>";
  exit;
  }
```


NOTE

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 front page form will pop up.

Otherwise, the script prints a list of hits in a table format. `$search_results` contains all the hits, in the form of table rows. `table_header`, a subroutine in **cgi-lib.sol** that separates table headers out of the `@field_names` array, is used to generate the header. The script also prints `$search_notice` in case its search results exceeded the maximum number of hits defined in `$max_hits`.

```
print "<TABLE BORDER = \"1\" CELLSPACING = \"4\"
              CELLPADDING = \"4\">";
print &table_header (@field_names);
print <<"  end_of_html";
</TR>
$search_results
</TABLE><P>$search_notice<P>
end_of_html
foreach $form_data_key (keys %form_data)
  {
  unless ($form_data_key eq "database" ||
          $form_data_key eq "number_of_hits")
    {
     print "<INPUT TYPE = \"hidden\"
            NAME = \"$form_data_key\"
            VALUE = \"$form_data{$form_data_key}\"\n";
    }
  }
print <<"  end_of_html";
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Frontpage">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
```

Figure 12.4 shows what a sample search results page might look like.

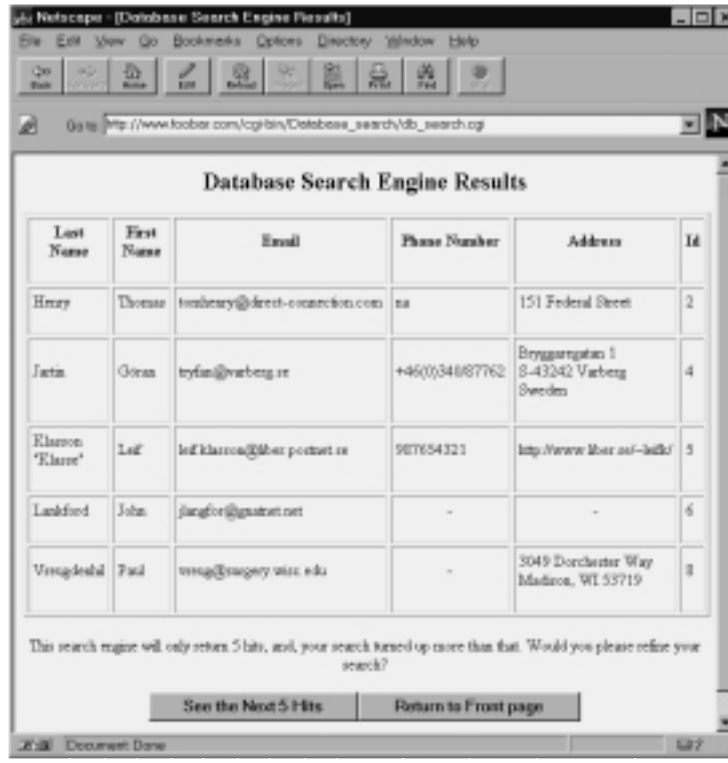

Figure 12.4 *Database search results.*

The CgiRequire Subroutine

`CgiRequire` checks to see whether the file that the script is trying to `require` 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 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 `required`.

```
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 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 $file are set to allow me
  access?";
  exit;
  }
 } # End of foreach $file (@require_files)
} # End of sub CgiRequire
```