CHAPTER 10

Using cgi-lib.sol

OVERVIEW

In writing the applications for this book, we found ourselves duplicating a number of routines in the scripts. To make our application more modular, we felt it would be a good idea to create a library of these routines. The result is **cgi-lib.sol**.

One function we added, `counter`, is used to create a unique ID number for every item in a database. Every added item must have a unique identification number if we are to search or modify the database, because the search or modification routines must have a way to identify each database row apart from all others. If a client says, "Delete item number 34 in the database," the script must be able to find item number 34 and delete it. If there were two item 34s or if the script had no way of knowing what item 34 meant, the script would not know which row to remove.

`open_error` returns a useful debugging note if a CGI script is having a problem opening a file. The script sends a note to the Web browser explaining the problem and specifying the problem file. Much as `CgiDie`

in **cgi-lib.pl** does, `open_error` provides useful debugging information because it helps identify where the problem is occurring.

`header_tags` outputs the basic HTML header for every page:

```
<HTML><HEAD><TITLE>Insert your own title</TITLE></HEAD>
```

This routine does the same basic thing as `HtmlTop` in **cgi-lib.pl** except that `header_tags` isolates the header within the header tags; it outputs HTML only to `</HEAD>`. It does not force a `<BODY>` tag or a document header, which may need to differ from one library usage to another. By isolating the "true" document header, the function becomes more modular.

`table_header` is used to generate table headers using the `<TH>XXX</TH>` tags. Using a loop, this routine makes creating table headers much more elegant and generalizable to any table. `table_header` generates a series of table headers in HTML format when passed an array of headers.

`table_rows` is used like `table_header`, but it generates table rows instead (`<TD>Item1</TD>`).

`select_tag` is used to generate HTML `<SELECT>` tags for various input forms that are used in the scripts. The `select_options` routine is used to create the lists of select `<OPTION>` tags for each input field.

`build_input_form` takes information submitted by the main routine and creates an input form with multiple fields of multiple types. `make_form_row` is used by `build_input_form` to build the `<INPUT>` tags for an input form. If we give it information as to which tag we want and the desired name and value information, the subroutine can create an entire input form by itself with very little HTML programming on our part.

`get_database_rows` goes through a database file and gathers each line as a separate database row into an array. This array can then be manipulated in the main routine.

`GetFileLock` and `ReleaseFileLock` are used together to create and delete a lock file. Lock files are used to make sure that no more than one person modifies a datafile at any one time. A lock file is vital for the integrity of your data. Imagine what would happen if two or three people were using the same script to modify a shared database and each person accessed the shared database at the same moment. At best, the data

entered by some of the users would be lost. Worse, the conflicting demands could possibly result in the corruption of the database file.

Thus, it is crucial to provide a way to monitor and control access to the database. This is the goal of the lock file routines. When a database modification script wants to access a shared database or other file protected by the lock file routines, it must first check for the existence of a lock file by using the file lock checks in `GetFileLock`. If `GetFileLock` determines that there is an existing lock file, it instructs the script that called it to wait until the lock file disappears. The script then waits and checks back with `GetFileLock` after some amount of time. If the lock file is still there, the script continues to wait.

If, on the other hand, the lock file has disappeared, the script asks `GetFileLock` to create a new lock file and then edits the database. When the script is finished editing the database, the lock file is erased using `ReleaseFileLock`.

INSTALLATION AND USAGE

Maintaining A Counter File

The `counter` routine is fairly simple. To use it, you need only call it from the main script as follows:

```
&counter("./Counter/counter.file");
```

Notice that the routine requires one parameter, the location of the counter file, which is created before the main script calls this subroutine. In this case, the `counter` routine is instructed to access a file called **counter.file** in the **Counter** subdirectory.

The `counter` routine keeps track of the current database ID number by maintaining a *counter file*, which is a file with the current unique number. By continually incrementing the number in the counter file by 1, the `counter` routine is able to keep generating new unique identification numbers.

The counter routine returns a variable named $item_number, which equals the value of the incremented counter. The script can use this number for any new database additions.

Handling Errors When Opening Files

Like counter, the open_error routine also takes one parameter, the name of the file the script is trying to open. Its usage follows this format:

```
open (DATABASE, ">>dat.file") || &open_error("dat.file");
```

In this example, if the script has trouble opening the file **dat.file**, it accesses the subroutine open_error, passing it the name **dat.file**. open_error then sends a useful error message to the Web browser.

Creating the HTML Header Tags

header_tags outputs the HTML header tags with any title submitted by the main program. For example, if you called the routine this way,

```
print &header_tags ("Selena Sol's Database Manager");
```

your script would output the following:

```
<HTML>
<HEAD>
<TITLE>Selena Sol's Database Manager</TITLE>
</HEAD>
```

Creating Multiple Table Headers:

table_header is used to create table header fields for HTML tables. For example, suppose you pass to the routine the following information:

```
print &table_header("Cell One", "Cell Two", "Cell Three");
```

You would receive the following table header:

```
<TH>Cell One</TH>
<TH>Cell Two</TH>
<TH>Cell Three</TH>
```

You can send `table_header` as many table headers as you want by adding elements to the list. This routine is particularly helpful if you are creating several tables throughout one script.

Building a Select Tag

`select_tag` is used to create a `<SELECT>` input tag. The routine is called with the following syntax:

```
&select_tag ("City", "3" ,"Multiple");
```

The three arguments reflect the values of the select arguments that you want included in the final tag. Hence, the `<SELECT>` tag generated from the preceding line would be

```
<SELECT NAME = "City" SIZE = "3" MULTIPLE>
```

We also need a routine to generate the `<SELECT>` tag options. We pass the `select_options` routine a list of options using this format:

```
print &select_options ("Los Angeles", "Burbank",
                       "Pasadena");
```

The routine returns the `<SELECT>` options as well as closes out the `<SELECT>` tag, returning to the main routine the following:

```
<OPTION>Los Angeles
<OPTION>Burbank
<OPTION>Pasadena
</SELECT>
```

Building an HTML Input Form

`build_input_form` automates the generation of form inputs even further. This function requires several variables to be set before it is called. Because we will

generate a form field, we need a list of form input titles, the variable names to be associated with those titles, and the type of form arguments used to gather the data for each title. In the end, we want a form input like the following:

```
<TH>Last Name:</TH>
<TD><INPUT TYPE = "text" NAME = "last_name" SIZE = "32"
           MAXLENGTH = "100"></TD>
```

To have the script generate these lines, we feed it the appropriate information. The following arrays provide that data.

```
@field_names = ("Last Name", "First Name", "Email");
%FIELD_ARRAY = ( 'Last Name', 'last_name',
                 'First Name', 'first_name',
                 'Email', 'email');
%FORM_COMPONENT_ARRAY = ( 'Last Name', 'text|SIZE =
                          "32" MAXLENGTH = "100"',
                          'First Name', 'text|SIZE =
                          "32" MAXLENGTH = "100"',
                          'Email', 'text|SIZE = "32"
                          MAXLENGTH = "100"');
```

`@field_names` defines the titles, `%FIELD_ARRAY` associates those titles with their variable names, and `%FORM_COMPONENT_ARRAY` associates them with the type of input boxes they are associated with.

`%FORM_COMPONENT_ARRAY` is complex enough to warrant more explanation. The previous example shows how you would generate a text input. The pipe-delimited list includes two fields: the name `text` and a listing of all the options as they would normally appear in your HTML.

`<TEXTAREA>` input boxes are generated in much the same way. Consider a line such as this one in the `%FORM_COMPONENT_ARRAY` associative array definition:

```
'Last Name', 'textarea|ROWS = "5" COLS = "40"'
```

That line would yield a TEXTAREA input box like the following:

```
<TH>Last Name:</TH>
<TD><TEXTAREA NAME = "last_name" ROWS = "5" COLS = "40">
</TEXTAREA></TD>
```

The <SELECT> tag is a bit more complex. Instead of using a pipe to delimit only two fields (type of input and arguments), the <SELECT> tag requires a third field: a list of options. Thus, a call such as

```
' Last Name', 'select|3|MULTIPLE|Flintstone|Rubble'
```

would produce a <SELECT> tag as follows:

```
<TH>Last Name</TH>
<TD><SELECT NAME = "last_name" SIZE = "3" MULTIPLE">
<OPTION>Flintstone
<OPTION>Rubble
</SELECT></TD>
```

To call the build_input_form from the main routine, you use syntax similar to the following:

```
print "<TABLE>";
```

A foreach loop is used to parse through the array:

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

Building a form automatically is somewhat complicated, and so is the subroutine call. In fact, build_input_form uses another routine, make_form_row.

make_form_row is a supplementary routine that builds each individual row assigned to build_input_form. The first thing it needs is a list of input field names. In the preceding example, the routine was asked to create a form with three input fields—last name, first name, and E-mail—which were stored in @field_names. However, to generate a full <INPUT> tag, the routine requires that a variable name be associated with each input field for the NAME attribute of the <INPUT> tag. To do this, the associative array %FIELD_ARRAY was defined to match the fields in @field_names with associated variable names.

Finally, we communicate the type of <INPUT> tag to be used on the form. Will it be a text box, a text area, or a select tag, for example? Furthermore, what options, if any, will modify those input tags? The final array, %FORM_COMPONENT_ARRAY, defines such values.

When the subroutine is called, it processes each type of input field separately so that every input field created will have output a separate input description and input field. For every $field_name in our array, the routine goes through the process of creating the table row. The final output is a full-fledged input form embedded in a table:

```
<TABLE>
<TR>
<TH>Last Name</TH>
<TD><INPUT TYPE = "text" NAME = "last_name" SIZE = "32"
           MAXLENGTH = "100" ></TD>
</TR><TR>
<TH>First Name</TH>
<TD><INPUT TYPE = "text" NAME = "first_name" SIZE = "32"
           MAXLENGTH = "100" ></TD>
</TR><TR>
<TH>Email</TH>
<TD><INPUT TYPE = "text" NAME = "email" SIZE = "32"
           MAXLENGTH = "100" ></TD>
</TR><TR>
<TH>Email</TH>
<TD><INPUT TYPE = "text" NAME = "email" SIZE = "32"
           MAXLENGTH = "100" ></TD>
</TR><TR>
</TABLE>
```

make_form_row does most of the work for build_input_form by taking each input field and creating an input tag based on the parameters set in %FORM_COMPONENT_ARRAY. make_form_row is meant to be used from within the build_input_form subroutine but can be accessed from another program using this call:

```
&make_form_row ("$field_name",
                "$variable_name", "$form_type");
```

The make_form_row routine takes a name, a variable name, and a form input type (where type includes a pipe-delimited list of options) and trans-

lates them into an input field. Thus, `make_form_row` builds an `<INPUT>` tag as follows:

```
<INPUT TYPE = "text" NAME = "email" SIZE = "32"  MAXLENGTH = "100" >
```

At the time of this writing, only three types of input fields are supported by the routine: TEXT, SELECT, and TEXTAREA.

Getting Rows from a Database

`get_database_rows` creates a list array of all the rows from a database. `get_database_rows` takes one argument, the name of the datafile it is using. An example call would look something like the following:

```
&get_database_rows ("data.file");
```

This code returns an array containing all the database rows in the ASCII text file **data.file**.

Outputting Multiple HTML Table Rows

`table_rows` does much the same thing as `table_header` except that it outputs table rows. Thus, the syntax:

```
&table_rows ("Bob", "Fred", "John");
```

generates the following output:

```
<TD>Bob</TD>
<TD>Fred</TD>
<TD>John</TD>
```

Manipulating Lock Files

Both `GetFileLock` and `ReleaseFileLock` take the name of the file in use and a parameter. Thus, you call the routines using

```
&GetFileLock ("lock.file");
```

and

```
&ReleaseFileLock("lock.file");
```

respectively. `GetFileLock` creates the **lock.file**, and `ReleaseFileLock` deletes it. It is not necessary for you to create the **lock.file** or delete it yourself. It is only necessary that you give the script the permissions to create and delete the file.

Design Discussion

The `counter` subroutine is used to create a unique ID number for every item in a database.

```
sub counter
  {
```

The subroutine first assigns to the local variable `$counter_file` the filename that was passed to it from the main script.

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

Next, the subroutine opens the counter file. If the counter file cannot be opened, however, it accesses the `open_error` routine, passing it the filename.

```
open (COUNTER_FILE, "$counter_file") || &open_error($counter_file);
```

The, `counter` checks to see what number the counter is currently on and assigns that value to `$item_number`. `counter` is written to use the `while` method; if you want to, you can append to the counter file rather than write over it.

```
while (<COUNTER_FILE>)
  {
```

```
  $item_number = "$_";
  }
close (COUNTER_FILE);
```

Next, the subroutine adds one to the current counter number.

```
$item_number += 1;
```

Then it updates the counter file so that the file reflects the new number.

```
open (NOTE, ">$counter_file") || &open_error($counter_file);
print NOTE "$item_number\n";
close (NOTE);
```

Finally, the subroutine returns the number to the main script.

```
return $item_number;
}
```

The open_error Subroutine

`open_error` sends a note to the person debugging the script if it cannot open a file.

```
sub open_error
{
```

`open_error` begins by assigning to the local variable `$filename` the filename that was passed to it from the referencing routine.

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

Next, the subroutine lets the client know that the script was unable to open the requested file and exits so that the routine does not continue.

```
print "I am really sorry, but for some reason I was
unable to open <P>$filename<P>Please make sure that the filename is
correctly defined, actually exists, and has the right permissions
relative to the Web browser. Thanks!";
```

```
exit;
}
```

The header_tags Subroutine

`header_tags` outputs the basic HTML header for every page.

```
sub header_tags
{
```

The subroutine first assigns to the local variable `$title` the value submitted by the main routine. Then it sends back the HTML header tags with the title inserted.

```
local ($title) = @_;
$header_tags = "<HTML>\n<HEAD>\n<TITLE>$title</TITLE>\n</HEAD>\n";
return $header_tags;
}
```

The table_header Subroutine

`table_header` is used to generate table headers.

```
sub table_header
  {
```

`table_header` assigns to the local array `@headings` the headings that were passed to it from the referencing routine. It also specifies that `$table_header` will be a variable local to this routine.

```
local (@headings) = @_;
local ($table_header);
```

Next, the subroutine dumps the HTML arguments into the `$table_tag` variable and returns `$table_header` to the main routine once it has been filled.

```
foreach $table_field (@headings)
  {
```

```
  $table_header .= "<TH>$table_field</TH>\n";
  }
$table_header .= "\n";
return $table_header;
}
```

The select_tag Subroutine

select_tag is used to generate HTML <SELECT> tags for input forms.

```
sub select_tag
 {
```

First, select_tag assigns to the local variables $name, $size, and $multiple the values that were passed to it from the referencing routine. It also defines $select_tag, $select_argument, @select_arguments, @select_values, and %SELECT_ARGUMENTS as local variables.

```
local ($name, $size, $multiple) = @_;
local ($select_tag, $select_argument);
local (@select_arguments, @select_values, %SELECT_ARGUMENTS);
```

Next, the associative array %SELECT_ARGUMENTS is created using the values supplied from the main routine, and then arrays are created for the keys and values.

```
%SELECT_ARGUMENTS = ("NAME", "$name",
                     "SIZE", "$size",
                     "MULTIPLE", "$multiple");
@select_arguments = keys %SELECT_ARGUMENTS;
@select_values = values %SELECT_ARGUMENTS;
```

Then $select_tag—which will be used to build and contain the entire select tag—and $select_argument—which will be used to keep track of the arguments—are initialized.

```
$select_tag = "";
$select_argument = "";
```

select_tag then creates the $select_tag variable, adding the <SELECT to the select tag.

```
$select_tag .= "<SELECT ";
```

Then the arguments are added to the select tag. If this is going to be a MULTIPLE select, the subroutine adds that argument. Otherwise, it adds each argument (size or name) and its value to the growing select tag.

```
foreach $select_argument (@select_arguments)
  {
  if ($select_argument eq "multiple")
    {
    $select_tag .= "MULTIPLE ";
    }
  elsif ($SELECT_ARGUMENTS{$select_argument} ne "")
    {
    $select_tag .= "$select_argument =
    \"$SELECT_ARGUMENTS{$select_argument}\" ";
    }
  }
```

Finally, the select tag is completed and returned to the main routine.

```
$select_tag .= ">\n";
return $select_tag;
}
```

The select_options Subroutine

select_options is used to create the lists of <OPTIONS> tags for each input field.

```
sub select_options
{
```

First, the options passed to this subroutine are assigned to the local array @select_options. Then $select_options, which will be used to keep track of the final options list, is initialized.

```
local (@select_options) = @_;
local ($select_options);
$select_options = "";
```

Next, the options that were put into the `@select_options` array are added to the growing variable `$select_options`.

```
foreach $option (@select_options)
  {
  $select_options .= "<OPTION>$option\n";
  }
```

Finally, the subroutine completes the HTML necessary for the select tag and returns the variable to be printed by the main routine.

```
$select_options .= "</SELECT>\n";
return $select_options;
}
```

The build_input_form Subroutine

`build_import_form` creates an input form.

```
sub build_input_form
{
```

`build_input_form` assigns to the local variables `$variable_name`, `$form_type`, and `$field_name` the values that were passed to it from the referencing routine. It also creates the local variable `$input_form`, which is used to create the entire form and which is initially set to “”.

```
local ($variable_name, $form_type, $field_name) = @_;
local ($input_form);
$input_form = "";
```

Then the subroutine creates the input form within a table and adds the input fields. First, it uses the value of `$field_name` for the header cell and then uses the subroutine `make_form_row` to build the input form element

for the input cell. It then passes the make_form_row subroutine the field name, the variable name, and the form type so that it will be able to build the input tag for that cell. Finally, the subroutine closes the table row and returns it to the main routine to print.

```
$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";
return $input_form;
}
```

The make_form_row Subroutine

make_form_row is used to build the <INPUT> tags for an input form.

```
sub make_form_row
{
```

make_form_row assigns to the local variables $name, $variable_name, and $type the values that were passed to it from the referencing routine. Then it initializes the local array @options_and_arguments, which is used to temporarily store options and arguments to each input field, and the variable $form_row, which is used to store the growing form input tag.

```
local ($name, $variable_name, $type) = @_;
local (@options_and_arguments);
@options_and_arguments = ();
local ($form_row);
$form_row = "";
```

It then splits the $type variable into the type of input form to be created and the associated arguments that will modify it.

```
($type, @options_and_arguments) = split(/\|/, $type);
```

Next, it checks to see what type of input field the routine is being asked to generate and creates the appropriate tag.

```
if ($type eq "text")
  {
  $form_row .= "<INPUT TYPE = \"text\"
                       NAME = \"$variable_name\" ";
  foreach $argument (@options_and_arguments)
    {
    $form_row .= "$argument ";
    }
  $form_row .= ">";
  }

if ($type eq "textarea")
  {
  $form_row .= "<TEXTAREA NAME = \"$variable_name\" ";
  foreach $argument (@options_and_arguments)
    {
    $form_row .= "$argument ";
    }
  $form_row .= "></TEXTAREA>";
  }
```

In the case of a field that has been designated invisible, the subroutine outputs a note to the viewer of the input form that this is not modifiable data.

```
  if ($type eq "invisible")
      {
   $form_row .= "Not modifiable data";
   }
```

If the subroutine is being asked for a select input field, it references the `select_tag` and `select_options` routines to generate the `<SELECT>` tag and `<OPTION>` tag, respectively.

```
if ($type eq "select")
  {
  $number = shift (@options_and_arguments);
  $multiple = shift (@options_and_arguments);
  $form_row .= &select_tag("$variable_name", "$number");
```

```
  $form_row .= &select_options(@options_and_arguments);
  }
  return $form_row;
}
```

The get_database_rows Subroutine

`get_database_rows` gathers each line in a database into an array.

```
sub get_database_rows
{
```

First, `get_database_rows` assigns to the local variable `$datafile` the value that was passed to it from the referencing routine.

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

It then opens the datafile and gets every row that is not a comment line. (Comment lines are defined as having the word `COMMENT:` flush against the left margin.) Then it sends back the final array of all of the database rows.

```
 open (DATABASE, "$datafile");
 while (<DATABASE>)
   {
   unless ($_ =~ /^COMMENT:/)
    {
    push (@database_rows, $_);
    }
   }
@database_rows;
close (DATABASE);
}
```

The table_rows Subroutine

`table_rows` generates table rows.

```
sub table_rows
  {
```

table_rows assigns to the local array @row the row that was passed to it from the referencing routine. It also initializes the local variable $table_cell.

```
local (@row) = @_;
local ($table_cell);
```

Next, it dumps the HTML arguments into the $table_tag variable.

```
foreach $table_field (@row)
  {
  $table_field =~ s/~~/\|/g;
  $table_cell .= "<TD>$table_field</TD>\n";
  }
$table_cell .= "\n";
return $table_cell;
}
```

The GetFileLock Subroutine

The GetFileLock subroutine assigns to the local variable $lock_file the value passed to it from the main routine. It also initializes the local variable $endtime as 60 seconds. $endtime can be set to however long you want the routine to wait before it decides to create the lock file again. After 60 seconds, it is assumed that the original program instance that created the file either crashed or was killed before releasing the lock.

```
sub GetFileLock {
local ($lock_file) = @_;
local ($endtime);
$endtime = 60;
$endtime = time + $endtime;
```

As long as the lock file exists, and until the routine counts to 60 seconds, the routine does nothing. However, when all is clear, the routine creates the lock file.

```
while (-e $lock_file && time < $endtime) {
}
```

```
open(LOCK_FILE, ">$lock_file");
# flock(LOCK_FILE, 2); # 2 exclusively locks the file
} # End of get_file_lock
```

If you wish to use `flock` (if your system has it), switch the comment sign from the open line to the `flock` line.

The ReleaseFileLock Subroutine

`ReleaseFileLock` works much the same as `GetFileLock` but deletes the lock file instead of creating it.

```
sub ReleaseFileLock
  {
  local ($lock_file) = @_;
  # 8 unlocks the file if you use flock
  # flock(LOCK_FILE, 8);
  close(LOCK_FILE);
  unlink($lock_file);
  } # end of ReleaseFileLock
```