CHAPTER 13

The Groupware Calendar

OVERVIEW

Many offices have a calendar tacked to the wall in a central location. Employees add items such as scheduled vacations, reservations of conference rooms, or notices about meeting, conferences, or seminars. By using a shared calendar, employees can more efficiently coordinate their work with that of others in the organization. The groupware calendar script provides a Web interface to a similar shared calendar. Every user can read what other people have added and, if the script is configured to do it, can modify the other entries. Calendar entries are stored in an ASCII database file.

The calendar has two primary views: the month view and the day view. In the month view, users can see calendar months for any year within a range specified at installation. Each day in the month displays the subjects of events scheduled for that day. In the day view, the events for a given day can be viewed in greater detail. Users can easily change the displayed month or year depending on their needs.

Varying levels of security are configurable. For example, the calendar can be made to be viewable by anyone or by authenticated users only. Items within the calendar database can be made to be modifiable or deletable only by the person who posts them or by everyone.

Installation and Usage

The application should be installed into the default directory, **Calendar**. Figure 13.1 outlines the expanded directory structure and the permissions of files and subdirectories.

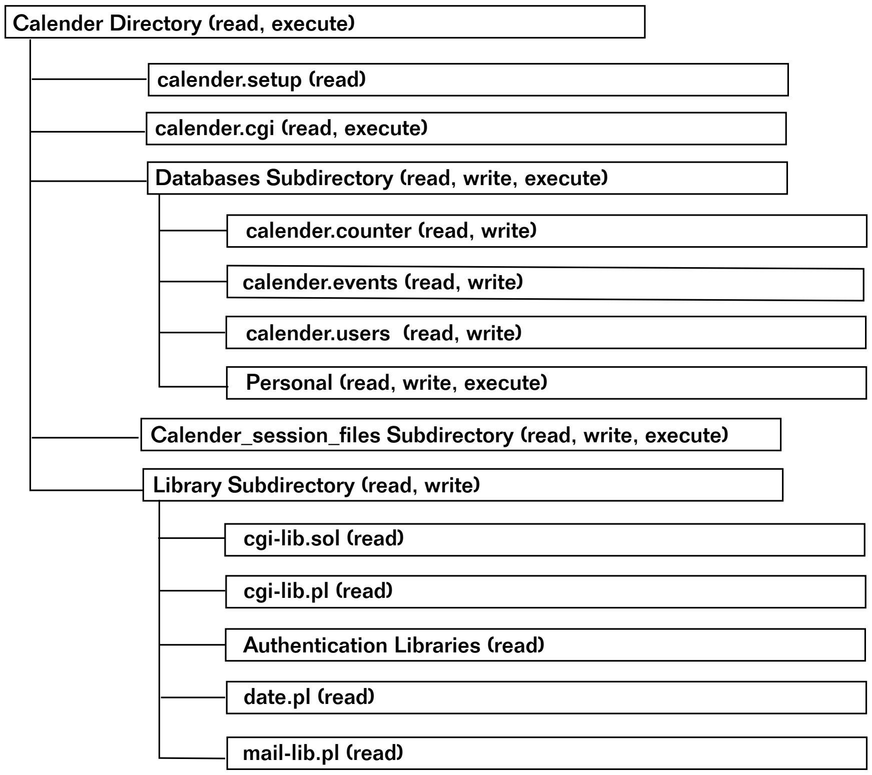

Calender Directory (read, execute)
calender.setup (read)
calender.cgi (read, execute)
Databases Subdirectory (read, write, execute)
calender.counter (read, write)
calender.events (read, write)
calender.users (read, write)
Personal (read, write, execute)
Calender_session_files Subdirectory (read, write, execute)
Library Subdirectory (read, write)
cgi-lib.sol (read)
cgi-lib.pl (read)
Authentication Libraries (read)
date.pl (read)
mail-lib.pl (read)

Figure 13.1 *Directory layout for calendar.*

This script should be placed in a directory that has permissions set to allow the Web server to read and execute and should expand to include three directories (**Calendar_session_files**, **Library**, and **Databases**) and two files (**calendar.cgi** and **calendar.setup**).

calendar.cgi is the application that provides most of the calendar processing. This file should be readable and executable by the Web server and will be discussed in greater detail in the design discussion.

calendar.setup is the file used to set the options and variables that pertain to your local setup. The file must be readable by the Web server and will be discussed in greater detail in the "Server-Specific Variables and Options" section.

Calendar_session_files is the subdirectory used by the authentication libraries to store session files, as discussed in Chapter 9. Initially the directory will be empty, but if it is set to be readable, writable and executable relative to the Web server, it will continually fill up and be pruned as a part of daily usage.

Library is a subdirectory containing the supporting CGI libraries, which are discussed in Part Two. The subdirectory should be readable and executable by the Web server and should contain the following libraries, which should be readable by the Web server: **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**.

Databases is a subdirectory containing the counter, events, and user files as well as any other calendar database files for alternative calendars. Because **calendar.cgi** must be able to write to this directory, its permissions should be set to be readable, writable, and executable.

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

calendar.events is the datafile used to store all the entered events. The format is exactly the same as for the databases we have discussed in previous chapters, including the protocol for comment lines and the use of pipe (|) as a field delimiter. If you have jumped right to this chapter and do not understand the database format, read the section on the datafile in Chapter 11.

The default fields for the events file are as follows: the day, month, year, username, first name, last name, E-mail address, subject, event time, event description, and ID number. The file must be readable and writable by the Web server.

calendar.users is a file containing the list of users who are allowed to modify the events database. This user file is formatted exactly the same as the default user file discussed in Chapter 9. This file should be readable and writable by the Web server and should initially be empty.

Personal is a sample subdirectory used to describe how to create separate calendar databases to be displayed by the same **calendar.cgi** script. The directory contains three files: **calendar.counter**, **calendar.events**, and **calendar.users**. These files define all the specific formats of each separate calendar.

Server-Specific Setup and Options

calendar.setup is the setup file used by **calendar.cgi** to define server-specific variables, configure authentication, and configure the events database. Defined variables are as follows:

`$this_script_url` is the location of **calendar.cgi**. Because we refer to it from here and because, theoretically, this file will be in the same directory, you need only state the name of the script. If that is the case and if you don't change the name of the file, don't bother changing this variable.

`$the_current_year` is pretty obvious. Set this to the current year.

`$greatest_year` is the highest-numbered year for which you want people to be able to submit calendar events on the Add Item form.

`$database_file` is the location of the file that contains the calendar database. Because this file can be defined by the user, we tag onto this variable the value of `$calendar_type`, which is given to us by the main script, **calendar.cgi**. In short, `$calendar_type` is a value set by the initial link via urlencoding

```
http://www.foobar.com/cgi-
bin/Calendar/calendar.cgi?calendar=Personal
```

Thus, the `$calendar_type` should be the directory name of each separate calendar.

NOTE **We don't recommend that you change this variable. The only reason to change it is if you don't like our file-naming conventions or if you are working with DOS and can have only 8.3 filenames.**

`$counter_file` is the path of the file that you are using to keep track of unique ID numbers. To make deletions and modifications, we must have a unique ID number so that the script can determine which database item to delete. These ID numbers should always be the last field in any database row. Again, because we need to isolate each of the calendars, we will reference the counter file including the `$calendar_type` variable.

`$temp_file` is a file that the script uses to temporarily store various data at various times. The file will be generated and deleted by the script.

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

Authentication variables are explained in Chapter 9.

`@day_names` is an array containing the names of the weekdays. A word of caution for all the arrays: the most common source of configuration errors is to forget to put a comma here or a quotation mark there, or even to forget to add one of the fields. So be very careful here; everything must be perfect!

`@month_names` is—yup, you got it—a list of month names.

`%MONTH_ARRAY` is an associative array that pairs month names with their numbers.

`%TIME` is an associative array that pairs time names with military time values.

`@time_values` is an ordered list of military time values.

`%FIELD_ARRAY` is an associative array that pairs database field names with their variable names.

`@field_names` is the list of database fields.

@field_values is the list of the variable names associated with the database fields in @field_names. By the way, the reason that we don't use key and value commands to define the arrays relative to all the associative arrays is that we want to predefine an order for them. If we used `keys` and `values`, we would lose our order in the hash table entry to the associative array.

`$field_num_time` is the element number in the array of the `event_time` field in the database. We need this value to sort the database by time so that when you click on a day view, the day events come up in order. When setting this variable, remember that arrays count from zero and not from 1.

As an example, the complete **calendar.setup** file is listed next:

```
$this_script_url = "calendar.cgi";
$the_current_year = "1996";
$greatest_year = "2011";
$database_file = "./Databases/$calendar_type/calendar.events";
$counter_file = "./Databases/$calendar_type/calendar.counter";
$temp_file =
"./Calendar_session_files/$calendar_type/calendar_temp.file";
$lock_file =
"./Calendar_session_files/$calendar_type/calendar_lock.file";
$auth_lib = "$lib";
$auth_server =  "off";
$auth_cgi =  "on";
$auth_user_file =  "./Databases/$calendar_type/calendar.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 = "./Calendar_session_files";
$auth_register_message = "Thanks, you may now logon with
       your new username and password.";
$auth_allow_register =  "on";
$auth_allow_search =  "on";
$auth_generate_password =  "off";
$auth_check_duplicates =  "on";
$auth_password_message = "Thanks for applying to our
      site, your password is";
@auth_extra_fields = ("auth_first_name",
                      "auth_last_name",
                      "auth_email");
@auth_extra_desc = ("First Name",
                    "Last Name",
                    "Email");
@day_names = ("Sunday", "Monday", "Tuesday",
              "Wednesday", "Thursday", "Friday",
              "Saturday");
@month_names = ("January", "February", "March", "April",
                "May", "June", "July", "August",
                "September", "October", "November",
                "December");
%MONTH_ARRAY = ('January', '1',       'February', '2',
                'March', '3',         'April', '4',
                'May', '5',           'June', '6',
                'July', '7',          'August', '8',
                'September', '9',     'October', '10',
                'November', '11',     'December', '12');
%TIME = ('01:00', '1 AM', '02:00', '2 AM', '03:00',
         '3 AM', '04:00', '4 AM', '05:00', '5 AM',
         '06:00', '6 AM', '07:00', '7 AM', '08:00',
         '8 AM', '09:00', '9 AM', '10:00', '10 AM',
         '11:00', '11 AM', '12:00', '12 Noon', '13:00',
         '1 PM', '14:00', '2 PM', '15:00', '3 PM',
         '16:00', '4 PM', '17:00', '5 PM', '18:00',
         '6 PM', '19:00', '7 PM', '20:00', '8 PM',
         '21:00', '9 PM', '22:00', '10 PM', '23:00',
         '11 PM', '24:00', '12 Midnight');
@time_values = ("01:00", "02:00", "03:00", "04:00",
                "05:00", "06:00", "07:00", "08:00",
                "09:00", "10:00", "11:00", "12:00",
                "13:00", "14:00", "15:00", "16:00",
                "17:00", "18:00", "19:00", "20:00",
                "21:00", "22:00", "23:00", "24:00");
%FIELD_ARRAY = ('Day', 'day',
                'Month', 'month',
                'Year', 'year',
                'Username', 'username',
                'First Name', 'first_name',
                'Last Name', 'last_name',
                'Email Address', 'email',
                'Subject', 'subject',
                'Event Time', 'time',
                'Body', 'body',
                'Database Id Number',
                'databse_id_number');
@field_names = ("Day", "Month", "Year", "Username",
                "First Name", "Last Name",
                "Email Address", "Subject",
                "Event Time", "Body",
                "Database Id Number");
@field_values = ("day", "month", "year", "username",
                 "first_name", "last_name", "email",
                 "subject", "time", "body",
                 "databse_id_number");
$field_num_time = "8";
```

Running the Script

Once you have configured the setup file to the specifics of your server and calendar datafile, it is time to try out your installation. To reference the script, use the following URL format.

```
<A HREF="http://www.foobar.com/cgi-
bin/Calendar/calendar.cgi">Calendar</A>
```

When clients click on this link, they should see the front page of your calendar with the default configurations. To reference separate calendar databases, the link to this script must have `?calendar=Some_subdirectory` added at the end of the URL. For example,

```
http://www.foobar.com/cgi-
bin/Calendar/calendar.cgi?calendar=Personal
```

DESIGN DISCUSSION

Figure 13.2 outlines the logic of the script as it manages the needs of the client and the demands of the server.

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

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

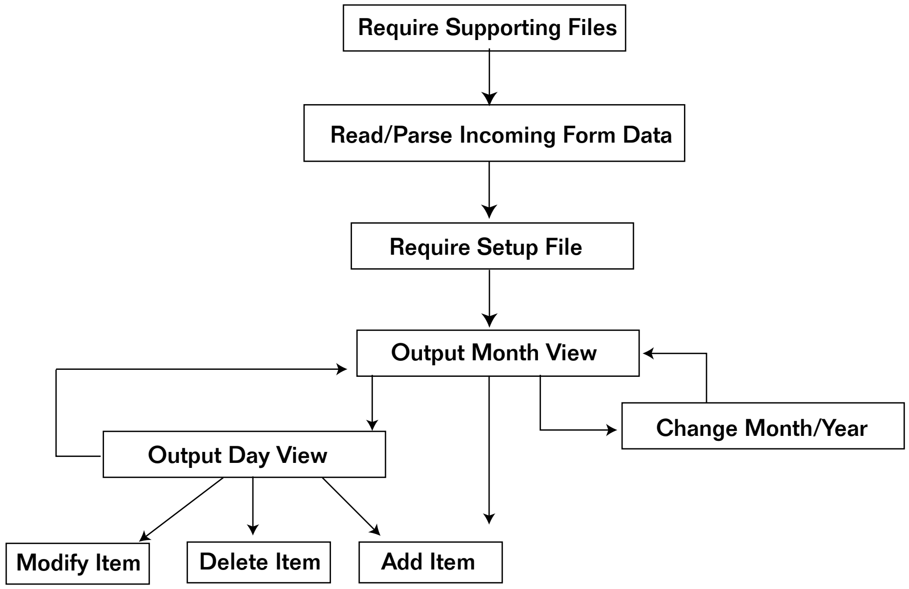


Figure 13.2 *The script logic.*

Loading the Supporting Libraries

Next, the script loads the necessary files using the subroutine `CgiRequire` at the end of this script. If there is a problem loading files, this subroutine returns a message that's valuable for debugging. `$lib` is the location of the **Library** directory, where these files are to be stored; because it is hard-coded in the main script, you must be careful to change it here if you move the library elsewhere. Keep `$lib` equal to `./Library` if you do not have a **Library** directory and are going to use the default directory included on the accompanying disk.

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

Reading and Parsing Incoming Form Data

cgi-lib.pl is used to parse the incoming form data. By passing it (`*form_data`), the subroutine returns an associative array that we reference as `$form_data{'key'}` instead of `$in{'$key'}`. In the end, the script will reference all the incoming form data as `$form_data{'variablename'}`.

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

Loading the Setup File

Once the incoming form data has been processed, the script determines which calendar database to use. If the calendar administrator has set up more than one calendar, each calendar database will be in a subdirectory, such as the example **Personal Database** that is included in the accompanying disk. To reference these separate databases, the link to this script must have `?calendar=Somesubdirectory` added at the end of the URL. For example:

```
http://www.foobar.com/cgi-
bin/Calendar/calendar.cgi?calendar=Personal
```

If the client does not submit a new database to use, the script will simply assign the default database and calendar. The `$calendar_type` variable will be used within the **calendar.setup** file, as addressed earlier.

```
if ($form_data{'calendar'} ne "")
  {
  $calendar_type = "./$form_data{'calendar'}";
  }
else
  {
  $calendar_type = "./";
  }
```

Then the script defines all the calendar-specific variables by using the setup file, which you should have customized for your site.

```
&require("calendar.setup");
```

Defining Intrascreen Variables

Next, we make sure that the script "remembers" the `$session_file` so that it can continually check for authentication and keep track of the current client. However, if the client has already logged on, the script does not need to revalidate the client, because it will be getting the `$session_file` as form data (the same hidden field we are about to define). Thus, the script renames `$form_data{'session_file'}` to `$session_file` so that in both cases (the client's first time at this point or subsequent script access by a continuing client) it will have the `session_id` in the same variable name form.

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

The script also renames some other variables using the same principle. In doing so, the script uses a couple of routines in **date.pl**, as discussed in Chapter 6, to manipulate date information so that it can use the date information in a standardized way.

```
if ($form_data{'year'} ne "")
  {
  $current_year = "$form_data{'year'}";
  }
else
  {
  $current_year = "$the_current_year";
  }
if ($form_data{'month'} eq "")
  {
  @mymonth = &make_month_array(&today);
  $current_month_name = &monthname($currentmonth);
  }
else
  {
  @mymonth =
&make_month_array(&jday($form_data{'month'},1,$current_year));
  $current_month_name = &monthname($form_data{'month'});
  }
```

Printing the Calendar for the Current Month

The script prints the dynamically generated calendar in two cases. First, the calendar is printed if the client has just logged on and is asking for the first page (`$form_data{'session_file'} ne ""`). Second, the calendar is printed if the client has already been moving through various pages and has asked to view the calendar again (`$form_data{'change_ month_year'} ne ""`). The || means "or." If either case is true, the script may proceed.

```
if ($form_data{'change_month_year'} ne "" ||
   $ENV{'REQUEST_METHOD'} eq "GET" &&
   $form_data{'day'} eq "")
   {
```

Displaying the Calendar Header

The script begins outputting the calendar by first printing the HTML calendar header.

```
&header ("Selena Sol's Groupware Calendar Demo:
$current_month_name - $current_year");
print <<"    end_of_html";
<CENTER>
<H2>$current_month_name - $current_year</H2>
</CENTER>
<TABLE BORDER = "2" CELLPADDING = "4" CELLSPACING = "4">
<TR>
end_of_html
```

NOTE

You can modify everything between the `print <<" end_of_html";` and the `end_of_html`, but be careful of illegal characters. For example, `@` must be preceded by a backslash (selena\@foobar.com).

The script then prints the table header (Weekdays). Essentially, for every day (`foreach $day`) in our list of days (`@day_names`), the script prints the day as a table header.

```
foreach $day (@day_names)
  {
```

```
  print "<TH>$day</TH>\n";
  }
print "</TR>\n<TR>\n";
```

Next, the script creates the variable `$count_till_last_day`, which it uses to make sure that it does not add too many `<TR>`s. Also, the script clears out a new variable, `$weekday`, which it uses to keep track of the two-dimensional aspect of the calendar: the script breaks the calendar rows after every seventh cell to represent a week. We will talk more about this later.

```
$count_till_last_day = "0";
$weekday = 0;
```

Displaying Calendar Days as Table Cells

For every day in the `mymonth` array, the script creates a cell for the calendar. The array `@mymonth`, if you recall, is an array returned from the subroutine `make_month_array`.

```
foreach $day_number (@mymonth)
  {
```

The script begins by incrementing the two counter variables: `$count_till_last_day` and `$weekday`.

```
  $count_till_last_day++;
  $weekday++;
```

The script must also make sure that it adds a break for every week to make the calendar two-dimensional. Thus, when it has gone through sets of seven days in this `foreach` loop, it resets `$weekday` to zero. In the following code, the script uses these values to determine where it should place the `</TR><TR>`, making a new calendar row. When `$weekday` is greater than 6, the script knows that it needs a `</TR><TR>`, so by setting the `$weekday` flag to zero, we notify the script a few lines later to insert the row break.

```
$weekday = 0 if ($weekday > 6);
```

Next, the script prints a table cell for each day. Because we want to make each of the numbers in each of the cells clickable so that someone can click on the number to see a day view, the script must manage a great deal of information here.

First, the script builds a variable called `$variable_list`, which is used to create a long URL appendix to transfer information using urlencoding. As we will discuss more specifically later, the routine that generates the day views needs to have the day, year, and month values if it is to bring up a day view. It must also have the `$session_file` value (as all the routines in this script must), the name of the calendar database, and the special tag `view_day=on`. So the script gathers all that information and appends it to the `$variable_list` variable.

```
$variable_list = "";
$variable_list = "day=$day_number&year=$currentyear";
$variable_list .= "&month=$currentmonth";
$variable_list .= "&session_file=$session_file";
$variable_list .= "&calendar=$form_data{'calendar'}";
$variable_list .= "&view_day=on";
```

Then the script creates the calendar cell. Notice that the number in each cell is made clickable by using urlencoding to tag the URL with all the variables we want passed.

```
print "<TD VALIGN = \"top\" WIDTH = \"150\">\n";
print "<A
HREF=\"$this_script_url?$variable_list\">$day_number</A
>\n";
```

Adding Subject Listings to Calendar Cells

The cell is not yet complete. The script must also grab from the calendar database the subject listings for all the entries on that day. In doing so, the script makes sure that, if it cannot open the database file, it sends a useful message to us for debugging. It uses the `open_error` subroutine in **cgi-lib.sol**, passing the routine the location of the database file.

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

If it successfully opens the database file, the script goes through each line, `split`ting the fields into their associated variables.

```
while (<DATABASE>)
    {
    ($day, $month, $year, $username, $first_name,
     $last_name, $email, $subject, $time, $body,
     $database_id_number) = split (/\|/,$_);
```

For every row searched, the script determines whether the day, month, and year of each item on that row are equal to the day, month, and year of the cell it is currently building.

```
if ($day eq "$day_number" && $month eq "$currentmonth"
&& $year eq "$currentyear")
   {
```

If it was able to answer `true` to all the preceding conditions, the script knows that it has found a match and prints the subject in that cell.

```
 print "<BR><FONT SIZE = \"1\">$subject</FONT>\n";
  } # End of if ($day eq "$day_number" && $month eq ...
} # End of while (<DATABASE>)
```

Once the script has checked all the lines in the database, it closes that cell and moves to the next cell.

```
print "</TD>\n";
```

If, however, the script reached the end of a week row, it must begin a new table row for the next week. If `$weekday` is equal to zero, then the script knows that it is time to begin a new row. Otherwise, it continues with the row.

```
if ($weekday == 0)
  {
  print "</TR>\n";
```

NOTE

By the way, here we use `==` instead of `=`. If we used `=`, Perl would interpret the part inside the `if ()` to be assigning the value of zero to `$weekday`, and it would evaluate the whole process as `true`. That would undercut the whole point of counting with `$weekday`.

Before the script blindly prints another table row, it makes sure that it has not reached the end of the month. If `$count_till_last_day` equals `@mymonth`, it knows that there are no more days left and it should not begin a new row. Notice that when we reference `@mymonth` without quotation marks, we receive the numerical value of the number of elements in the array.

```
unless ($count_till_last_day == @mymonth)
  {
  print "<TR>";
  } # End of unless ($count_till_last_day == @mymonth)
 } # End of if ($weekday == 0)
} # End of foreach $day_number (@mymonth)
```

Displaying Footer Information for the Month View

Once the script has finished making all the cells for the calendar, it prints the HTML footer.

```
print <<"    end_of_html";
</TABLE>
</CENTER>
<BLOCKQUOTE>
For day-at-a-glance calendar, click on the day number on the
calendar above.
<BR>
Or, to see another 1996 month, choose one
end_of_html
```

To let the user select a different month to view, the script creates a select box using the subroutine `select_a_month` at the end of this script.

```
&select_a_month;
print "<P>Or, to see another year, select one\n";
```

Likewise, it creates a select box that allows the client to choose a new year to view using `select_a_year` at the end of this script.

```
&select_a_year;
```

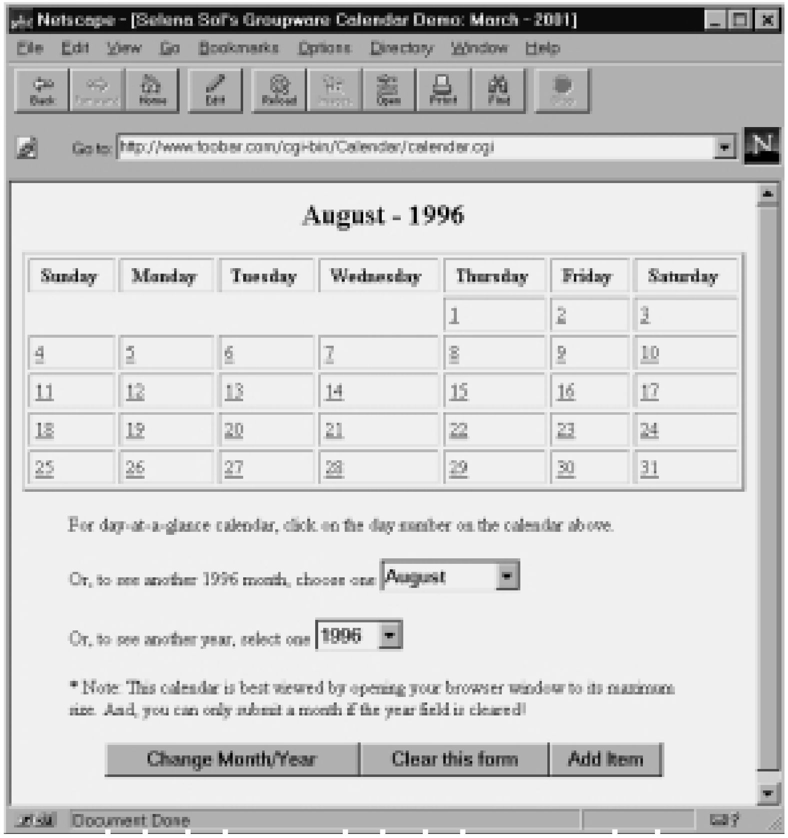
Then the script outputs the usual footer.

```
print <<"    end_of_html";
<P>
* Note: This calendar is best viewed by opening your
browser window to its maximum size. And, you can only submit
a month if the year field is cleared!<P>
</BLOCKQUOTE>
<CENTER>
<INPUT TYPE = "submit" NAME = "change_month_year"
        VALUE = "Change Month/Year">
<INPUT TYPE = "reset" VALUE = "Clear this form">
<INPUT TYPE = "submit" NAME = "add_item_form"
         VALUE = "Add Item">
</FORM>
</CENTER>
</BODY>
</HTML>
end_of_html
exit;
} # End of if ($form_data{'change_month_year'} ne "" ||...
```

On the Web, the front page looks like Figure 13.3.

Displaying a Day View

In the preceding routine, the script made every number in every cell of the calendar clickable so that the client could view the detailed descriptions of the events scheduled for that day. In the urlencoded string it built, the script included a tag `view_day = on`. Here is where that tag comes in handy. The following `if` test checks to see whether the person has clicked on a number; if the person has, the test will evaluate to `true`. If the test evaluates to `true`, the script prints the page header.


Netscape - [Selena Sol's Groupware Calendar Demo: March - 2001]
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/Calendar/calendar.cgi
Change Month/Year
Clear this form
Add Item
Document Done

August - 1996

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

For day-at-a-glance calendar, click on the day number on the calendar above.

Or, to see another 1996 month, choose one: August

Or, to see another year, select one: 1996

* Note: This calendar is best viewed by opening your browser window to its maximum size. And, you can only submit a month if the year field is cleared!

Figure 13.3 *The view month interface.*

```
if ($form_data{'view_day'} eq "on")
  {
  &header ("$current_month_name $form_data{'day'},
            $current_year");
  print <<"    end_of_html";
  <CENTER>
  <H2>$current_month_name $form_data{'day'},
      $current_year</H2>
  </CENTER>
  end_of_html
```

Next, the script opens the database again and looks for database rows that match the requested day, month, and year.

```
open (DAYFILE, "$database_file") ||
&open_error($database_file);
while (<DAYFILE>)
    {
```

Just as it did in the routine for generating subject lines for day cells, the script pays attention only to database rows that match the client-defined day, month, and year.

```
($day, $month, $year, $username, $first_name,
 $last_name, $email, $subject, $time, $body,
 $database_id_number) = split (/\|/,$_);
if ($day eq "$form_data{'day'}" &&
    $month eq "$form_data{'month'}" &&
    $year eq "$current_year")
        {
```

Next, the script sets the `$item_found` flag so that it can keep track of whether it has found an item in the database.

```
$item_found = "yes";
```

The script then prints an `<HR>`-delimited, detailed list of events of the day, followed by a standard HTML footer.

```
  print <<"        end_of_html";
  <B>Time:</B> $TIME{$time}<BR>
  <B>Subject:</B> $subject<BR>
  <B>Poster:</B>
  <A HREF = "mailto:$email">$first_name
             $last_name</A><BR>
  <B>Body:</B><BLOCKQUOTE>$body</BLOCKQUOTE>
  <P><CENTER><HR WIDTH = "50%"></CENTER><P>
  end_of_html
  } # End of if ($day eq "$form_data{'day'}" &&........)
} # End of while (<DAYFILE>)
```

If the script was not able to find any items in the database, it must let the client know. So if `$item_found` was never set to yes, the script sends the client a note of explanation.

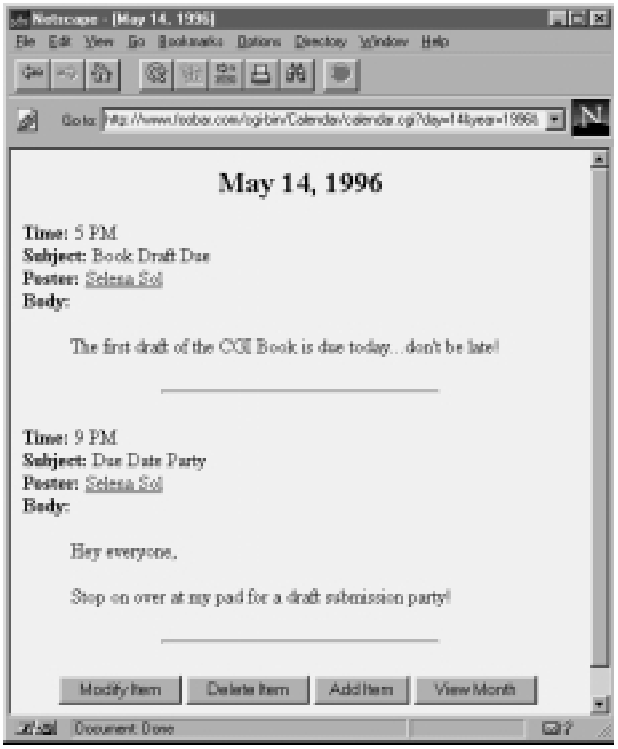
```
if ($item_found ne "yes")
  {
  print "<BLOCKQUOTE>It appears that there are no
        entries posted for this day.  Would you like to
        add one?</BLOCKQUOTE>";
  }
print <<"    end_of_html";
<CENTER>
<INPUT TYPE = "hidden" NAME = "day" VALUE = "$form_data{'day'}">
<INPUT TYPE = "hidden" NAME = "month"
            VALUE = "$form_data{'month'}">
<INPUT TYPE = "hidden" NAME = "year" VALUE = "$current_year">
<INPUT TYPE = "submit" NAME = "modify_item_form"
           VALUE = "Modify Item">
<INPUT TYPE = "submit" NAME = "delete_item_form"
           VALUE = "Delete Item">
<INPUT TYPE = "submit" NAME = "add_item_form"
           VALUE = "Add Item">
<INPUT TYPE = "submit" NAME = "change_month_year"
           VALUE = "View Month">
</FORM></CENTER></BODY></HTML>
end_of_html
exit;
}
```

Figure 13.4 shows the day view interface on the Web.

Authenticating the User

Allowing clients to view the calendar is one thing. Letting them modify the calendar database is another. The following routine checks to see whether the client is authorized to do anything in addition to viewing. The script passes `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**.

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


Netscape - [May 14, 1996]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://www.foobar.com/cgi-bin/Calendar/calendar.cgi?day=14&year=1996&
Modify Item
Delete Item
Add Item
View Month
Document: Done

May 14, 1996

Time: 5 PM
Subject: Book Draft Due
Poster: Selena Sol
Body:

The first draft of the CGI Book is due today...don't be late!

Time: 9 PM
Subject: Due Date Party
Poster: Selena Sol
Body:

Hey everyone,

Stop on over at my pad for a draft submission party!

Figure 13.4 *The day view.*

Displaying the Add Event Form

Once the client has been authenticated, the script determines the desired action. This first routine checks to see whether the client wants to add an item. If so, the script presents a form that the client uses to submit information for each of the database fields.

```
if ($form_data{'add_item_form'} ne "")
  {
  &header ("Add an Item to Selena's Groupware Calendar Demo");
```

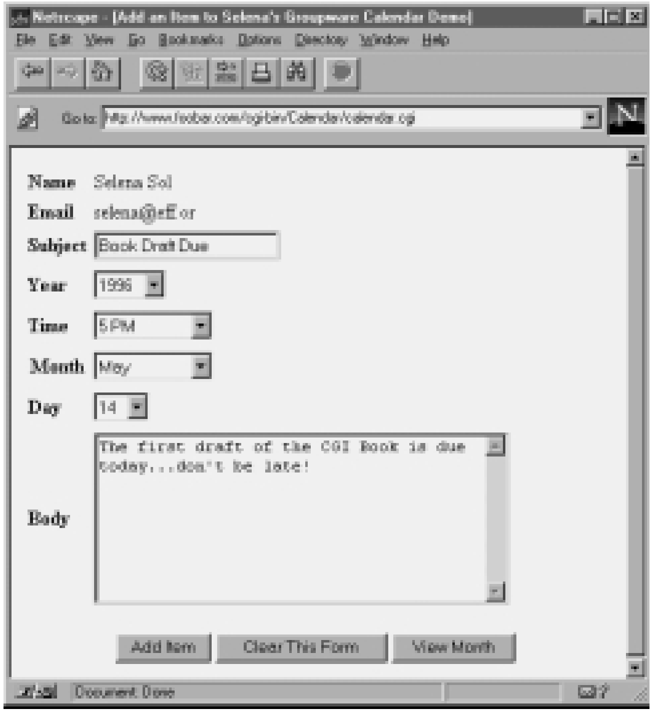
The subroutine `submission_form` at the end of this script is used to generate a form with input fields for every database item that can be manipulated by the client.

```
&submission_form;
```

The script then prints a standard HTML footer and quits.

```
print <<"    end_of_html";
<CENTER><P>
<INPUT TYPE = "submit" NAME = "add_item" VALUE = "Add
Item">
<INPUT TYPE = "reset" VALUE = "Clear This Form">
<INPUT TYPE = "submit" NAME = "change_month_year" VALUE
= "View Month">
</CENTER></BODY></HTML>
end_of_html
exit;
}
```

Figure 13.5 shows what the add form looks like on the Web.


Netscape - [Add an Item to Selena's Groupware Calendar Demo]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://www.foobar.com/cgi-bin/Calendar/calendar.cgi
Add Item
Clear This Form
View Month
Document: Done

Name Selena Sol
Email selena@eff.or
Subject Book Draft Due
Year 1996
Time 5 PM
Month May
Day 14
Body
```
The first draft of the CGI Book is due
today...don't be late!
```

Figure 13.5 The calendar add form.

Adding an Event to the Database

Once the client submits a new event, the script must be prepared to add the item to the calendar database. The following routine does just that.

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

The routine begins by printing the page header.

```
&header ("Adding an Item to the Calendar Database");
```

Next, the script makes sure that the client has filled out all the necessary fields in the submission form. The script gets a list of the variable names (keys) associated with the associative array `%form_data` given to us by **cgi-lib.pl**.

```
@form_data = keys (%form_data);
```

For every element in the list array `@form_data`, the script checks to see whether the associated value in `%form_data` has content. If it doesn't, the script sends an error message and quits.

```
foreach $variable_name (@form_data)
    {
    if ($form_data{$variable_name} eq "" &&
        $variable_name ne "calendar")
      {
      print <<"       end_of_html";
      <BLOCKQUOTE><FONT SIZE = "+3">
      I'm very sorry but you must enter something in
      the <B>$variable_name</B> input box.  Please press
      the back button and try again.
      </BLOCKQUOTE></BODY></HTML>
      end_of_html
      exit;
      }
    }
```

On the other hand, if the client entered data into all the fields, the script adds the new entry. First, it uses the subroutine `GetFileLock` in **cgi-lib.sol** to create a lock file to protect database integrity during modification.

The script passes as the sole parameter the location of the lock file used by this program. If the script makes it past the lock file routine, it means that the script is the sole owner of the database file and can safely make changes.

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

Before it can add a new entry, though, the script must acquire a unique number from the counter file by using the subroutine `counter` in **cgi-lib.sol**. `counter` receives as its one parameter the location of the counter file used by this program.

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

NOTE

The unique counter number is essential, because every row must be uniquely identifiable for modifications and deletions. The numbers don't need to be in any order, and there can be gaping holes between numbers (as when items are deleted), but they must be unique.

Next, the script writes the contents of the new entry to the database file, appending (>>) the new data to the end of the existing list of items.

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

Then it formats the incoming form data so that the new event will be confined to one database line. To do this, the script changes (`=~ s/`) all occurrences (`/g`) of newlines (`\n`) into `<BR>`, and all occurrences of two hard returns (`\r\r`) into `<P>`.

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

Finally, the script simplifies some of the variables and generates the new database row.

```
if ($session_first_name eq "")
  {
  $session_first_name =  "$form_data{'first_name'}";
  $session_last_name =  "$form_data{'last_name'}";
  $session_email = "$form_data{'email'}";
  }
$subject = "$form_data{'subject'}";
$event_time = "$form_data{'time'}";
$month = "$form_data{'month'}";
$day = "$form_data{'day'}";
$year = "$form_data{'year'}";
$body = "$form_data{'body'}";
$new_row = "";
$new_row .= "$day\|$month\|$year\|$session_username\|";
$new_row .=
"$session_first_name\|$session_last_name\|$session_email\|";
$new_row .= "$subject\|$event_time\|$body\|$item_number";
```

The script now safely adds the new database row to the database file and deletes the lock file so that someone else may modify the database file.

```
print DATABASE "$new_row\n";
close (DATABASE);
&ReleaseFileLock ("$lock_file");
```

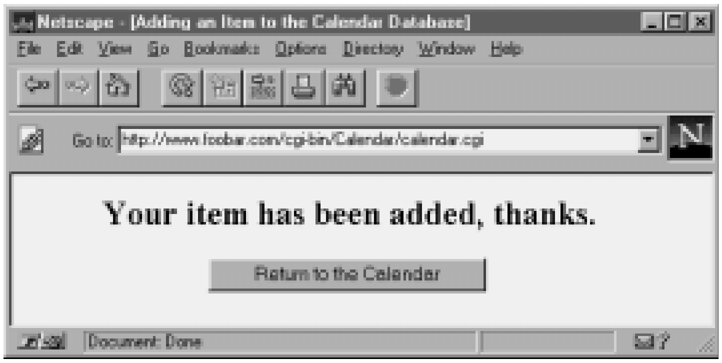
NOTE

Don't forget the newline at the end of the database row so that the next item entered will be on its own line.

Finally, the script prints the standard page footer.

```
print <<"    end_of_html";
<H2><CENTER>Your item has been added, thanks.</H2>
<INPUT TYPE = "hidden" NAME = "day" VALUE = "$form_data{'day'}">
<INPUT TYPE = "hidden" NAME = "month" VALUE = "$form_data{'month'}">
<INPUT TYPE = "hidden" NAME = "year" VALUE = "$current_year">
<INPUT TYPE = "submit" NAME = "change_month_year"
       VALUE = "Return to the Calendar">
</BODY></HTML>
end_of_html
```

Figure 13.6 shows the response on the Web.


Netscape - [Adding an Item to the Calendar Database]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://www.foobar.com/cgi-bin/Calendar/calendar.cgi
Return to the Calendar
Document: Done

Your item has been added, thanks.

Figure 13.6 Web response.

Sorting the Calendar Database by Time

We are not finished with the add quite yet. The script must also sort the entries in the database file so that when clients choose day views, their entries come out ordered by time. Again, the script creates the lock file so that no one else can modify the database file while it is being modified.

```
&GetFileLock ("$lock_file");
open (DATABASE, "$database_file") || &open_error($database_file);
```

The script adds every row in the database file to an array called `@database_fields`.

```
while (<DATABASE>)
  {
  @database_fields = split (/\|/, $_);
```

Next, it creates a variable called `$comment_row`, which is used to hold comment lines in the database file. We do not want them sorted along with the rest of the items.

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

If the database row is not a comment row, the script finds the field that has the time of the event and appends it to the front of the database row. (So it occurs twice: once at the beginning of the line and again in the middle somewhere.) The script also adds (pushes) the whole string (`$sortable_row`) into a growing array called `@database_rows`. (We'll explain why in the next paragraph.)

```
else
  {
  $sortable_row = "$database_fields[$field_num_time]~~";
  $sortable_row .= $_;
  push (@database_rows, $sortable_row);
  }
}
```

When all the "modified" rows have been added to the array `@database_rows`, the script sorts the array. This is why we appended the time to the beginning of each of the rows: the sort routine sorts all the database items by event time.

```
@sorted_temp_database = sort (@database_rows);
```

NOTE

There is no need to sort on the date, because database rows are already displayed by date. We need only sort the items by time within a day.

Now the script goes through `@sorted_temp_database` and takes out the extra `event_time` string at the beginning of each database row, and we're back where we started except that the rows are sorted. The script `splits` the string at ~~ and then `pushes` the part of the string that corresponds to the original database row back into the array `@final_sorted_database`.

```
foreach $database_row (@sorted_temp_database)
  {
  ($extra_event_time, $true_database_row) = split (/~~/,
   $database_row);
  push (@final_sorted_database, $true_database_row);
  }
close (DATABASE);
```

Next, the script modifies the original database file so that it represents the sorted order. To do this, it creates a temporary file to which it reprints all the comment rows stored in the variable `$comment_row`.

```
open (TEMPFILE, ">$temp_file") || &open_error($temp_file);
print TEMPFILE "$comment_row";
```

Then, for each of the database rows stored in `@final_sorted_database`, the script prints to the temporary file.

```
foreach $row (@final_sorted_database)
  {
  print TEMPFILE "$row";
  }
close (TEMPFILE);
```

Finally, the script copies the temporary file over the original database file using the `rename` command so that the resulting file represents the sort. Then the lock file is deleted.

```
rename ($temp_file, $database_file);
&ReleaseFileLock ("$lock_file");
exit;
}
```

Displaying the Modification Form

Next, if asked to do so, the script prints the modify event form.

```
if ($form_data{'modify_item_form'} ne "")
{
&header ("Modify and Item");
```

First, it prints the basic header, including the hidden fields, which must be transferred to the modification routines so that they will have all the user information necessary to re-create database rows.

Because the modification routines will compare incoming form data to database row information, this information must come in with the rest of the form data.

```
print <<"    end_of_html";
<INPUT TYPE = "hidden" NAME = "username"
       VALUE = "$session_username">
<INPUT TYPE = "hidden" NAME = "first_name"
       VALUE = "$session_first_name">
<INPUT TYPE = "hidden" NAME = "last_name"
       VALUE = "$session_last_name">
<INPUT TYPE = "hidden" NAME = "email"
       VALUE = "$session_email">
<CENTER>
<H2>$current_month_name $form_data{'day'}, $current_year</H2>
</CENTER>
end_of_html
```

The script then begins a table that will display all the items posted by the client on the day of interest. But, for the time being, instead of printing the table immediately, it builds it in a variable called `$table`.

```
$table .= "<TABLE BORDER = \"1\" CELLSPACING = \"2\"
           CELLPADDING = \"2\" WIDTH = \"1100\">\n";
$table .= "<TR>\n";
$table .= "<TH>Modify Item</TH>\n";
```

Similarly, the script adds the header row to `$table`.

```
foreach $name (@field_names)
  {
  $table .= "<TH>$name</TH>\n";
  }
$table .= "</TR>\n";
```

Then the script opens the database and checks for items that correspond to the user as well as the requested day, month, and year.

```
open (DAYFILE, "$database_file") ||
      &open_error($database_file);
while (<DAYFILE>)
  {
  chop $_; # Make sure to take out the newline.
```

Next, it `splits` the database row as usual, but this time it also creates the list array `@database_values`, which will be discussed soon.

```
($day, $month, $year, $username, $first_name,
$last_name, $email, $subject, $time, $body, $database_id_number) =
split (/\|/,$_);
@database_values = split (/\|/,$_);
```

The script is directed to pay attention only to items specific to user, day, month, and year.

```
if ($day eq "$form_data{'day'}" &&
    $month eq "$form_data{'month'}" &&
    $year eq "$form_data{'year'}" &&
    $session_username eq "$username")
        {
```

The script also flags the fact that it found an item.

```
$item_found = "yes";
```

Then the script continues adding to `$table` by adding the table row corresponding to the database row that was matched. Also, it adds a radio button so that the client can select the table row to modify.

```
$table .= "<TR>\n";
$table .= "<TD ALIGN = \"center\">";
$table .= "<INPUT TYPE = \"radio\" NAME =\"item_to_modify\"";
$table .= "VALUE=\"$database_id_number\"></TD>\n";
foreach $value (@database_values)
    {
     $table .= "<TD>$value</TD>\n";
     }
     $table .= "</TR>\n";
   }
 }
$table .= "</TR></TABLE><P><CENTER>\n";
```

If `$item_found` is still not equal to `yes`, it means that we did not match any items and that the script should send the client a note of explanation.

```
if ($item_found ne "yes")
  {
  print <<"      end_of_html";
  <BLOCKQUOTE>
  I'm sorry, you have not posted any items for this day,
  so there is nothing for me to modify.
  </BLOCKQUOTE><CENTER>
  <INPUT TYPE = "submit" NAME = "change_month_year"
         VALUE = "View Month"></BODY></HTML>
  end_of_html
  exit;
  }
```

If, however, `$item_found` equals yes, the script prints `$table`.

```
print "$table";
```

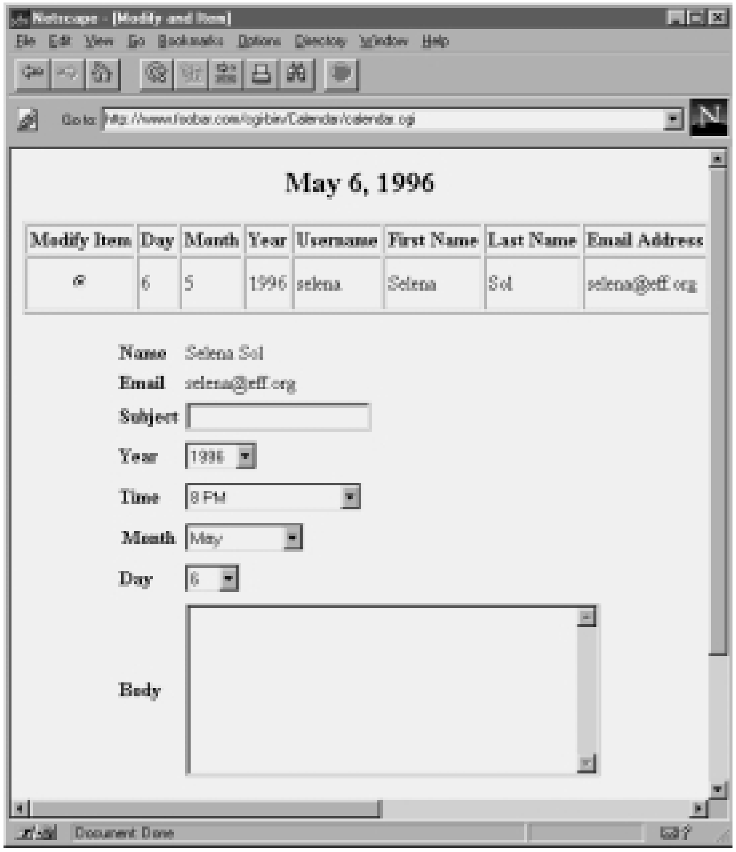
In the case of modification, the client also needs a form similar to the add form so that he or she can make any desired modifications. We use the `submission_form` subroutine at the end of this script, passing it the parameter `modify` so that it will know to output that form.

```
&submission_form("modify");
```

Finally, the script prints a standard footer and quits.

```
print <<"    end_of_html";
<CENTER><P>
<BLOCKQUOTE><I>Note: Make sure to select an item to modify using the
radio buttons on the top table.  Then change any of the form inputs
you want changed, leaving the others as they are.  Feel free to cut
and paste from the top table to the bottom table if you only need to
change a small amount of
text</I></BLOCKQUOTE>
<INPUT TYPE = "submit" NAME = "modify_item"
       VALUE = "Modify Selected Item">
<INPUT TYPE = "reset" VALUE = "Clear This Form">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the modify form looks like Figure 13.7.


Netscape - [Modify and Item]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://www.foobar.com/cgi-bin/Calendar/calendar.cgi
Document: Done

May 6, 1996

Modify Item	Day	Month	Year	Username	First Name	Last Name	Email Address
●	6	5	1996	selena	Selena	Sol	selena@eff.org

Name	Selena Sol
Email	selena@eff.org
Subject	
Year	1996
Time	8 PM
Month	May
Day	6
Body	

Figure 13.7 Calendar modify form.

Displaying the Delete Event Form

Next, the script prints a form for item deletion if requested by the client.

```
if ($form_data{'delete_item_form'} ne "")
  {
  &header ("Delete an Item");
  print "<CENTER>\n";
  print "<H2>$current_month_name $form_data{'day'},
             $current_year</H2>\n";
  print "</CENTER>\n";
```

Just as it did for the modify form, the script creates the `$table` variable and prints the delete form (or the error message if no items were found). First, the script outputs the header.

```
$table = "";
$table .= "<TABLE BORDER = \"1\" CELLSPACING = \"2\"
            CELLPADDING = \"2\" WIDTH = \"1100\">";
$table .= "\n<TR>\n";
$table .= "<TH>Delete Item</TH>";
foreach $name (@field_names)
  {
  $table .= "<TH>$name</TH>\n";
  }
$table .= "</TR>\n";
```

Then it creates the possible delete rows.

```
open (DAYFILE, "$database_file") || &open_error($database_file);
while (<DAYFILE>)
  {
  chop $_;
  ($day, $month, $year, $username, $first_name,
   $last_name, $email, $subject, $time, $body,
   $database_id_number) = split (/\|/,$_);

  @database_values = split (/\|/,$_);

  if ($day eq "$form_data{'day'}" &&
      $month eq "$form_data{'month'}" &&
      $year eq "$form_data{'year'}" &&
      $session_username eq "$username")
    {
    $item_found = "yes";
    $table .= "<TR>\n";
    $table .= "<TD ALIGN = \"center\">";
    $table .= "<INPUT TYPE = \"radio\"
                      NAME =\"item_to_delete\"";
    $table .= "VALUE=\"$database_id_number\"></TD>\n";

    foreach $value (@database_values)
      {
```

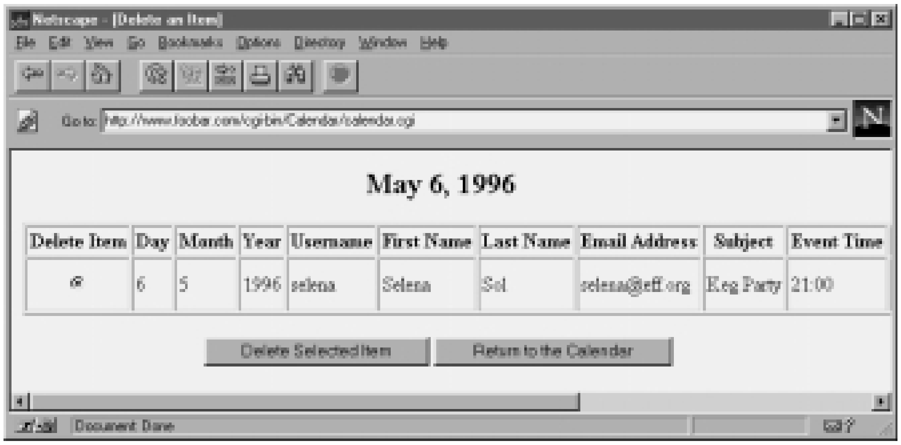
```
      $table .= "<TD>$value</TD>\n";
      }
      $table .= "</TR>\n";
    }
  }
```

Next, if necessary, the script prints an error message.

```
if ($item_found ne "yes")
  {
  print <<"      end_of_html";
  <BLOCKQUOTE>
  I'm sorry, you have not posted any items for this day,
  so there is nothing for me to delete.
  </BLOCKQUOTE><CENTER>
  <INPUT TYPE = "submit" NAME = "change_month_year"
         VALUE = "View Month"></BODY></HTML>
  end_of_html
  exit;
  }

print <<"    end_of_html";
$table
</TR></TABLE><CENTER><P>
<INPUT TYPE = "hidden" NAME = "day"
       VALUE = "$form_data{'day'}">
<INPUT TYPE = "hidden" NAME = "month"
       VALUE = "$form_data{'month'}">
<INPUT TYPE = "hidden" NAME = "year" VALUE = "$current_year">
<INPUT TYPE = "submit" NAME = "delete_item"
       VALUE = "Delete Selected Item">
<INPUT TYPE = "submit" NAME = "change_month_year"
       VALUE = "Return to the Calendar">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the delete form looks like Figure 13.8.


Netscape - [Delete an Item]
File Edit View Go Bookmarks Options Directory Window Help
Go to: http://www.foobar.com/cgi-bin/Calendar/calendar.cgi
Document: Done

May 6, 1996

Delete Item	Day	Month	Year	Username	First Name	Last Name	Email Address	Subject	Event Time
●	6	5	1996	selena	Selena	Sol	selena@eff.org	Keg Party	21:00

Delete Selected Item
Return to the Calendar

Figure 13.8 *The calendar delete form.*

Deleting an Event from the Database

If asked, the script deletes an item from the database.

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

The script must be sure that the client actually chose an item to delete with the radio buttons.

```
if ($form_data{'item_to_delete'} eq "")
  {
  &header ("Woopsy");
  print <<"      end_of_html";
  <CENTER><
  H2>Delete an Item in the Database Error</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 delete.
  Would you please make sure that you select an item
  \"and\" fill in the new information. Just press the
  back button.  Thanks.
  </BLOCKQUOTE>
  end_of_html
  exit;
  }
```

First, the script locks the database file as it did for the add item routines.

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

Then it creates a temporary file as before.

```
open (TEMP, ">$temp_file") || &open_error($temp_file);
close (TEMP);
```

If there is data in the database file, the script checks to see which item matches the deletion.

```
open (DATA, "$database_file") || &open_error($database_file);
while (<DATA>)
  {
  @grepfields=split(/\|/,$_);
```

To do so, the script gets the unique database ID for each database row and chops off the newline.

```
$database_id = pop (@grepfields);
chop $database_id;
```

If the unique database ID of the row is not equal to the database ID number submitted by the client, the script knows not to delete that row. Instead, it prints it to the temporary file.

```
if ($database_id ne "$form_data{'item_to_delete'}")
  {
  open (TEMP, ">>$temp_file") ||
        &open_error($temp_file);
  print TEMP "$_";
  close (TEMP);
  }
} # End of while (<DATA>)
```

Once it has gone through all the items in the database, the script copies the temporary file over the database file; the deletion will have been made, because the row that matched the database ID number will not have been printed to the temporary file. Then the script closes the database file and deletes the lock file so that others can modify the database.

```
close (DATA);
rename ($temp_file, $database_file);
&ReleaseFileLock ("$lock_file");
```

Finally, the script prints a standard footer.

```
&header ("Deleting an Item from the Calendar");
print <<"    end_of_html";
<CENTER>\n<FONT SIZE = \"+3\">Your item has been deleted
</FONT>\n<P>
<INPUT TYPE = "hidden" NAME = "day"
       VALUE = "$form_data{'day'}">
<INPUT TYPE = "hidden" NAME = "month"
       VALUE = "$form_data{'month'}">
<INPUT TYPE = "hidden" NAME = "year" VALUE = "$current_year">
<INPUT TYPE = "submit" NAME = "change_month_year"
       VALUE = "Return to the Calendar">
</CENTER></BODY></HTML>
end_of_html
exit;
}
```

Modifying an Event in the Database

The script can also be used to modify an item.

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

First, the script must be sure that the client chose an item to modify.

```
&header("Modify an Item in the database");
if ($form_data{'item_to_modify'} eq "")
```

```
  {
  print <<"      end_of_html";
  <CENTER><H2>Modifying an Item in the Database Error</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. Just press the back button.  Thanks.
  </BLOCKQUOTE>
  end_of_html
  exit;
  }
```

As it did before, the script creates the lock file and the temporary file.

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

And as it did for deletion, the script gets the unique database ID number for each row by popping it out of the `@fields` array. But this time, it makes sure to add the database ID number into the array so that it will have a whole array again (`push (@fields, $item_id)`). Finally, as usual, the script `chop`s off the newline.

```
while (<DATABASE>)
    {
    @fields = split (/\|/, $_);
    $item_id = pop(@fields);
    chop $item_id;
    push (@fields, $item_id);
```

If the item ID of the database row matches the one that the client submitted, the script renames the `@fields` array to `@old_fields`. Otherwise, it adds the line to the growing list of database rows in `$new_data`.

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

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

Once it gets through all the items in the database, the script should have found one that matched the item selected by the client, and the rest should have been stored in `$new_data`. Now the script prints the rows in `$new_data` to the temporary file.

```
print TEMPFILE "$new_data";
```

Then it prepares to substitute the new data submitted by the client for the old data that was in the database. First, the script initializes a couple of variables: `$counter` and `$new_line`. `$counter` will be used to keep track of the database fields that we have edited, and `$new_line` will be used to create the new database row.

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

Now the script begins going through the list of `field_values` as defined in the setup file.

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

NOTE

Recall that arrays begin with a zero so that the first array element is `$arrayname[0]`. In this case, the script assigns the current element in the count of field values to `$value`, thus going through every field in the database.

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

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

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

```
  else
    {
    $form_data{$value} =~ s/\n/<BR>/g;
    $form_data{$value} =~ s/\r\r/<P>/g;
    $form_data{$value} =~ s/\|/~:~/g;

    if ($form_data{$value} eq "")
      {
      $form_data{$value} = "<CENTER>-</CENTER>";
      }
    $new_line .= "$form_data{$value}|";
    } # End of else
```

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

```
  $counter++;
} # End of until ($counter >= @field_values)
chop $new_line; # take off last |
```

Next, the script closes everything, copies the temporary file over the original, and releases the lock file.

```
print  TEMPFILE "$new_line\n";
close (TEMPFILE);
close (DATABASE);
rename ($temp_file, $database_file);
&ReleaseFileLock ("$lock_file");
```

Then it prints the usual footer.

```
print <<"    end_of_html";
<CENTER><H2>Your Item has been Modified</H2>
<INPUT TYPE = "hidden" NAME = "day"
       VALUE = "$form_data{'day'}">
<INPUT TYPE = "hidden" NAME = "month"
       VALUE = "$form_data{'month'}">
<INPUT TYPE = "hidden" NAME = "year" VALUE = "$current_year">
<INPUT TYPE = "submit" NAME = "change_month_year"
       VALUE = "Return to the Calendar">
</CENTER></BODY></HTML>
end_of_html
```

Again, it is time to sort the entries in the database file so that when people choose day views, their entries are ordered by time. The script creates the lock file so that no one else can modify the database file while we are modifying it.

```
&GetFileLock ("$lock_file");
open (DATABASE, "$database_file") || &open_error($database_file);
```

Then the script adds every row in our database file to the list array `@database_fields` and creates `$comment_row` as before.

```
while (<DATABASE>)
  {
  @database_fields = split (/\|/, $_);
  if ($_ =~ /^COMMENT:/)
    {
    $comment_row .= $_;
    }
```

If the database row is not a comment row (`COMMENT:`), the script finds the field that has the time of the event and appends it to the front of the database row. It also adds (`pushes`) the whole string (`$sortable_row`) into a growing array called `@database_rows`.

```
else
  {
  $sortable_row = "$database_fields[$field_num_time]~~";
  $sortable_row .= $_;
  push (@database_rows, $sortable_row);
  }
}
```

When it has added all the modified rows to the array `@database_rows`, the script sorts `@database_rows`.

```
@sorted_temp_database = sort (@database_rows);
```

Next, the script goes through `@sorted_temp_database` and takes out the extra `event_time` string at the beginning of each database row.

```
foreach $database_row (@sorted_temp_database)
  {
  ($extra_event_time, $true_database_row) =
        split (/:/, $database_row);
  push (@final_sorted_database, $true_database_row);
  }
close (DATABASE);
```

Then the script creates the temporary file for the modification as it did for the addition.

```
open (TEMPFILE, ">$temp_file") ||
        &open_error($temp_file);
print TEMPFILE "$comment_row";
```

Next, for each of the database rows stored in `@final_sorted_database`, the script prints to the temporary file.

```
foreach $row (@final_sorted_database)
  {
  print TEMPFILE "$row";
  }
close (TEMPFILE);
```

Finally, the script copies the temporary file over the original database file so that the resulting file will represent the sort. Then the lock file is removed.

```
rename ($temp_file, $database_file);
&ReleaseFileLock ("$lock_file");
exit;
}
```

Displaying the Default Error

The script adds a default in case clients got through everything without finding what they wanted (probably because they pressed **Return** when typing into a text box).

```
&header("Wooopsy");
print <<"  end_of_html";
<BLOCKQUOTE>I'm sorry, you are not allowed to press the Return key
when typing in your subject.  Please press the back button and try
again.</BLOCKQUOTE><CENTER>
<INPUT TYPE = "submit" NAME = "change_month_year"
       VALUE = "Return to the Calendar">
</CENTER></BODY></HTML>
end_of_html
```

The make_month_array Subroutine

The `make_month_array` subroutine is used to generate the month arrays used by the main routine.

```
sub make_month_array
  {
```

First, the subroutine defines some variables that will be local to this subroutine.

```
local($juldate)  = $_[0];
local($month,$day,$year,$weekday);
local($tempjdate,$firstweekday,$numdays,$lastweekday);
local(@myarray);
```

Next, the subroutine defines variables based upon the passed parameter.

```
($month, $day, $year, $weekday) = &jdate($juldate);
```

Then `make_month_array` makes a new date based on the first of the month.

```
$tempjdate = &jday($month, 1, $year);
```

make_month_array also gets the weekday of the first of the month and then builds @myarray to be passed to the main routine.

```
($month, $day, $year, $weekday) = &jdate($tempjdate);
$firstweekday = $weekday;
$currentmonth = "$month";
$currentyear = "$year";
$month++;
if ($month > 12)
  {
  $month = 1;
  $year++;
  }
$tempjdate = &jday($month,1,$year);
$tempjdate—;
($month, $day, $year, $weekday) = &jdate($tempjdate);
$numdays = $day;
$lastweekday = $weekday;

for ($x = 0;$x < $firstweekday; $x++)
  {
  $myarray[$x] = " ";
  } # End of for

for ($x = 1; $x <= $numdays; $x++ )
  {
  $myarray[$x + $firstweekday - 1] = $x;
  }

 for ($x = $lastweekday; $x < 6; $x++)
  {
  push(@myarray,"");
  }

return @myarray;
}
```

The CgiRequire Subroutine

This subroutine checks to see whether the file that we are 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 is filled with the filenames sent from the main routine.

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

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

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

If any of the files are not readable or do not exist, the subroutine sends 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
```

The select_a_month Subroutine

The `select_a_month` subroutine is used to generate a select list of months that the client can use to select months in the various forms throughout the script. It is a straightforward routine with no new syntax.

```
sub select_a_month
  {
```

```
  print "<SELECT NAME=\"month\">\n";
  foreach $month (@month_names)
    {
    if ($month ne "$current_month_name")
      {
      print "<OPTION VALUE =
              \"$MONTH_ARRAY{$month}\">$month\n";
      }
    else
      {
      print "<OPTION SELECTED VALUE =
           \"$MONTH_ARRAY{$month}\">$month\n";
      }
    }
  print "</SELECT>\n";
  }
```

The select_a_year Subroutine

As with `select_a_month`, the `select_a_year` subroutine generates a select input tag and options for the client to select from a list of years. The only thing of note in this subroutine is that `$the_current_year` is defined in the setup file and must be changed annually.

```
sub select_a_year
  {
  print "<SELECT NAME = \"year\">\n";

  for ($i = $the_current_year; $i < $greatest_year; $i++)
    {
    if ($i eq "$currentyear")
      {
      print "<OPTION SELECTED VALUE = \"$i\">$i\n";
      }
    else
      {
      print "<OPTION VALUE = \"$i\">$i\n";
      }
    }
  print "</SELECT>\n";
  }
```

The submission_form Subroutine

The `submission_form` subroutine is used to generate a form that clients can use to submit new events to the database. As with the previous subroutines, the logic includes no new Perl tricks or syntax.

```
sub submission_form
  {
  local ($type_of_form) = @_;

  if ($session_first_name ne "")
    {
    print <<"      end_of_html";
    <TABLE BORDER = "0" CELLSPACING = "2"
           CELLPADDING = "2">
    <TR ALIGN = "LEFT">
    <TH>Name</TH>
    <TD>$session_first_name $session_last_name</TD>
    <TR ALIGN = "LEFT">
    <TH>Email</TH>
    <TD>$session_email</TD>
    </TR>
    end_of_html
    }
  else
    {
    print <<"      end_of_html";
    <TABLE BORDER = "0" CELLSPACING = "2"
           CELLPADDING = "2">
    <TR ALIGN = "LEFT">
    <TH>First Name</TH>
    <TD><INPUT TYPE = "text" NAME = "first_name"
               SIZE = "20" MAXLENGTH = "20"></TD>
    </TR>
    <TR ALIGN = "LEFT">
    <TH>Last Name</TH>
    <TD><INPUT TYPE = "text" NAME = "last_name"
               SIZE = "20" MAXLENGTH = "20"></TD>
    </TR>
    <TR ALIGN = "LEFT">
    <TH>Email</TH>
    <TD><INPUT TYPE = "text" NAME = "email" SIZE = "20"
               MAXLENGTH = "20"></TD>
    </TR>
    end_of_html
    }
  print <<"    end_of_html";
  <TR ALIGN = "LEFT">
  <TH>Subject</TH>
  <TD><INPUT TYPE = "text" NAME = "subject" SIZE = "20"
             MAXLENGTH = "20"></TD>
  </TR>
  <TR ALIGN = "LEFT">
  <TH>Year</TH>
  <TD>
  end_of_html
  &select_a_year;
  print <<"    end_of_html";
  </TD>
  </TR>
  <TR ALIGN = "LEFT">
  <TH>Time</TH>
  <TD>
  <SELECT NAME = "time">
  end_of_html
  if ($type_of_form eq "modify")
    {
    print "<OPTION VALUE = \"\">Don't Change Time\n";
    }
  foreach $time_value (@time_values)
    {
    if ($time_value ne "09:00")
      {
      print "<OPTION VALUE =
            \"$time_value\">$TIME{$time_value}\n";
      }
    else
      {
      if ($type_of_form ne "modify")
        {
        print "<OPTION SELECTED VALUE =
            \"$time_value\">$TIME{$time_value}\n";
        }
      }
    }

  print "</SELECT></TD></TR>\n";

  print "<TR>\n<TH>Month</TH>\n";
  print "<TD>\n";
  &select_a_month;
  print "</TD>\n";
```

```
  print "</TR>";

  print "<TR ALIGN=LEFT>\n";
  print "<TH>Day</TH>\n";
  print "<TD><SELECT NAME=\"day\">\n";
  for ($i = 1; $i < 32; $i++)
    {
    if ($i eq "$form_data{'day'}")
      {
      print "<OPTION SELECTED VALUE = \"$i\">$i\n";
     }
    else
      {
      print "<OPTION VALUE = \"$i\">$i\n";
      }
    }
  print <<"    end_of_html";
  </SELECT></TD>
  </TR>
  <TR ALIGN=LEFT>
  <TH>Body</TH>
  <TD><TEXTAREA WRAP = "virtual" NAME = "body" ROWS = "8"
                COLS = "40"></TEXTAREA></TD>
  </TR>
  </TABLE>
  end_of_html
  }
```

The Header Subroutine

The `header` subroutine is used by the main script to generate the HTML header common to every script-generated HTML page.

```
sub header
  {
  local ($title) = @_;
  if ($title eq "")
    {
    $title = "Selena Sol's Groupware Calendar Demo";
    }
  print <<"    end_of_html";
  <HTML><HEAD><TITLE>$title</TITLE></HEAD>
  <BODY>
  <FORM METHOD = "post" ACTION = "$this_script_url">
  <INPUT TYPE = "hidden" NAME = "session_file"
         VALUE = "$session_file">
  <INPUT TYPE = "hidden" NAME = "calendar"
         VALUE = "$form_data{'calendar'}">
  end_of_html
  }
```