CGI WITH perl

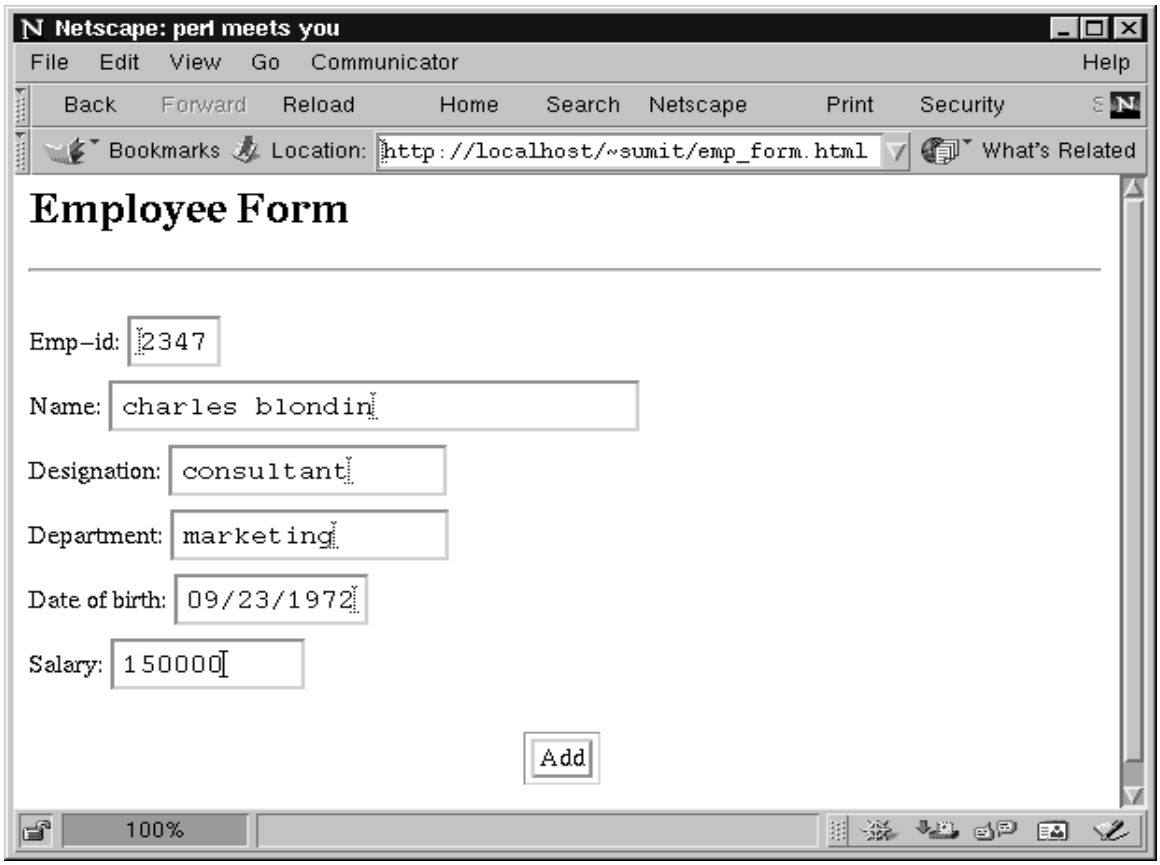
perl is ideally suited for use in CGI programming, so we'll develop an application to do two simple things:

- Filter form data.
- Generate HTML.

The CGI application also uses a similar employee database that was presented in Section 12.1 except this time we'll access this database from our Web browser. Before doing that, you need to know some HTML to understand the significance of its tags. You should then be able to make the CGI **perl** program generate the right tags when it sends back data to the browser. The HTML code presented below specifies a form with the <form> tag:

```
$ cat emp_form.html
<html> <head>
           <title>The Employee Database</title>          Appears on title bar
       </head>
       <body>
            <h1> Employee Form </h1>                     Appears in a large bold font
            <hr>                                         Adds a horizontal rule
        <form action="http://localhost/~sumit/cgi-bin/emp_add.pl" method=get>
           Emp-id: <input type="text" name="empid" size=4> <br>
           Name: <input type="text" name="ename" size=30> <br>
           Designation: <input type="text" name="desig" size=15> <br>
           Department: <input type="text" name="dept" size=15> <br>
           Date of birth: <input type="text" name="dtbirth" size=10> <br>
           Salary: <input type="text" name="salary" size=10> <br> <br>
           <center>
              <input type=submit value="Add">            The Add button is centered
           </center>
       </form>
       </body>
</html>
```

The action attribute of this tag specifies a URL pointing to a **perl** program (**emp_add.pl**) on the server. This program adds a line to the database. The form that accepts user input is shown next.



`emp_form.html`: An HTML form viewed by Netscape

Every HTML document consists of some header and footer code. Since **`perl`** has to generate these lines in most CGI applications, we'll create two subroutines to be used by our CGI programs.

The body of the HTML document specifies a form enclosed by the `<form>` and `</form>` tags. There are six text boxes here which accept the six fields of the employee database. The values entered into these fields are paired with their corresponding variable names—`empid`, `ename`, `desig`, and so on. The `<br>` tag is required to place each text box in a separate line.

The `action` attribute of the `<form>` tag specifies a URL on the localhost itself. This URL points to a **`perl`** program **`emp_add.pl`** in `cgi-bin` (the directory where CGI programs are generally kept). The program is executed the moment the button labeled *Add* and of type `submit` is clicked with the mouse.

The Query String

The browser sends data to the server through its request header. To understand how form data is structured, consider a form that has only three fields with names `empid`, `ename` and `desig` (the `name` attribute of the `<input>` tag). Let's put the values `1234`, `henry higgins` and `actor` into these three fields. On submission, the browser strings together the entire data as *name=value* pairs into a query string in this manner:

```
empid=1234&ename=henry+higgins&desig=actor
```

This single string is sent to the server specified in the URL. The `&` here acts as the delimiter of each *name=value* pair. Note that the browser has encoded the space character to a `+`.

To be able to use this data, **`perl`** has to split this string twice—once to extract all *name=value* pairs and then to separate the names from their values. This can be done in two ways depending on the method specified.

GET and POST: The Request Method

The `<form>` tag shows another attribute: `method`. This signifies the way data is transmitted to the server. Generally, the query string shown above is sent in two ways:

- **GET** This method appends the query string to the URL using the `?` as the delimiter. With this string, the URL will now look like this:

  ```
  http://localhost/cgi-bin/emp_add.pl?empid=1234&ename=henry+higgins&desig=actor
  ```

The server parses the `GET` statement in the request header and stores the data following the `?` in its environment variable, `QUERY_STRING`. This variable can be used by any CGI program.

- **POST** With this method, the browser precedes this string with a number signifying the number of characters the string holds. The server stores this number in the `CONTENT_LENGTH` variable. It supplies the string as standard input to the CGI program. **`perl`** reads this data with its **`read`** function, and reads just as much as specified by `CONTENT_LENGTH`.

The method itself is available as the `REQUEST_METHOD` variable in the server's environment. Our sample HTML form uses GET as the method. GET has the limitation that the string size is restricted to 1024 characters. If you have a lot of data to transmit, then use POST. However, the structure of the query string is the same in both cases, so the **`emp_add.pl`** script should be able to handle the data using both methods. There was no compelling reason to choose GET rather than POST for this example.

The CGI program, **emp_add.pl** (shown later), has to parse the data in QUERY_STRING (for GET) or STDIN (for POST). It then has to combine the extracted data into a single line and add it to a text file acting as the database. To better understand what's going on, we'll print the contents of the important CGI environment variables on the browser window. The CGI program must be able to generate the HTML required to display these messages.

Creating the Subroutines for Header and Footer

Since all HTML documents have a common header and footer segment, let's first frame two subroutines for them. **Htmlheader** prints the header:

```
sub Htmlheader {
  local ($title, $h1) = @_;
   print << "MARKER";                          A here document
   <html>
       <head>
           <title> $title </title>             Variable substitution enabled
       </head>
       <body>
       <h1> $h1 </h1>
MARKER                                         Don't indent this!
}
```

Htmlheader accepts two arguments into the placeholders $title and $h1. This means you have the option of specifying the title and first level header when calling the subroutine. Here, we have used a single **print** statement like a here document. The double quotes surrounding the marker tag ensure that variable substitution is enabled (required for $title and $h1).

The subroutine for the footer is simpler still:

```
sub Htmlfooter {
   print "</body></html>\n";
}
```

Place these two subroutines (and another one that we'll be discussing shortly) in a separate file, **web_lib.pl**. They will be required at runtime by the CGI program. Make sure you add the statement 1; at the end of the file so that it always returns a true value.

emp_add.pl: The Main CGI Program

Before we take up the third subroutine, let's have a look at the CGI program **emp_add.pl** specified in the URL. Apart from printing some messages on the browser window, the program appends a line built from form data to a text database:

```
$ cat emp_add.pl
#!/usr/bin/perl
#
require "web_lib.pl" ;

open (OUTFILE, ">>/home/sumit/public_html/emp_out.lst") ;
&Parse(*field) ;
print "Content-type: text/html\n\n";
&Htmlheader("Testing Query String", "The QUERY_STRING Variable") ;

print "The query string is $ENV{'QUERY_STRING'}<br>\n" ;
print "The method of sending data to server is $ENV{'REQUEST_METHOD'}<br>\n" ;
print "THE content length is $ENV{'CONTENT_LENGTH'}<br>\n" ;
print OUTFILE "$field{'empid'}|$field{'ename'}|$field{'desig'}|$field{'dept'}|
$field{'dtbirth'}|$field{'salary'}<br>\n" ;
print "A record has been added <a href=\"http://localhost/cgi-bin/emp_query.pl\"
>Click here to see the records</a><br>\n" ;

&Htmlfooter ;
close (OUTFILE);
```

Since this program generates HTML, it has to explicitly spell out its Content-Type and then leave a blank line (`\n\n`) before sending back data. The HTML header is then printed with the **`Htmlheader`** subroutine. The footer is printed at the end with **`Htmlfooter`**.

Observe that the array `%field` is passed by reference with the `*` prefix to the **`Parse`** subroutine (discussed next). The next three **`print`** statements following `&Htmlheader` display the contents of the server's environment variables on the browser window. The fourth one uses the filehandle `OUTFILE` to add a line to the database (the file `emp_out.lst`), using the `|` as the field delimiter. The final **`print`** statement prints a completion message and offers a hyperlink (with `A HREF`) to the **`emp_query.pl`** program. You should be able to see all lines of `emp_out.lst` when you click on this link.

The child HTTP process that communicates form data to the server runs as an ordinary user. For the process to be able to create the file `emp_out.lst`, the directory `public_html` must be world-writable (with **`chmod 777`**). This is necessary for entering the first detail. Once the file is created, the directory can have its old permissions.

The `Parse` Subroutine

`Parse` is the third subroutine that we need to use here. **`Parse`** makes each *name=value* pair available as separate entries in the associative array, `%field`. To understand how **`perl`** makes the form values available to the main program, we need to study this subroutine:

```
sub Parse {
  local (*in) = @_ ;
  local ($i, $key, $val) ;                          # Local variables

  if ($ENV{'REQUEST_METHOD'} eq "GET") {            # Takes care of both GET
     $in = $ENV{'QUERY_STRING'} ;
  } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {      # ... and POST
     read(STDIN, $in, $ENV{'CONTENT_LENGTH'}) ;
  }                                                 # Query string in $in

  @in = split(/&/, $in) ;                       # Break up into name=value pairs
  foreach $i (0 .. $#in) {
     $in[$i] =~ s/\+/ /g ;                          # Decode a + to a space
     ($key, $val) = split(/=/, $in[$i], 2) ;        # Splits on the first =
     $key =~ s/%(..)/pack("c",hex($1))/ge ;
     $val =~ s/%(..)/pack("c",hex($1))/ge ;
     $in{$key} = $val ;                    # Name and value in associative array
  }
  return %in ;
}
```

Parse here accepts an array by reference. This array is copied to %in inside the subroutine. We used the associative array %ENV to evaluate the server's three environment variables we have previously discussed. The query string is assigned to the variable $in irrespective of the method used. POSTed data is also read into $in, but from standard input (STDIN is the filehandle) with the **read** function. The number of characters read is determined by **read**'s third argument (the content length).

The query string uses the & as the delimiter of the *name=value* pairs. The first **split** stores every such pair in the scalar array, @in. The **s** function decodes every + in the string to a space as encoding (space to a +) takes place whenever there are spaces in the data. Many characters are encoded into hexadecimal strings because they have special significance in the URL string. For instance, the / that separates the elements of the date field is used to delimit directories in the URL string. As you can see in the figure below, it is encoded to %2F before the query string is sent to the server.

The **pack** function converts these hex values back to their original ASCII characters. Note how the **s** function identifies these characters with the pattern %(..) that uses a TRE. The ge flags ensure that **pack** is interpreted as an expression and not treated literally.

The **foreach** loop picks up each *name=value* pair from the array @in. After decoding, each element of the array is split again, this time on the =, and stored in the variables $key and $val. The first is set as the key and the other as the value in the associative array %in. This array is returned to the calling program. Note that in this program we used in as a variable ($in), as a scalar list (@in) and an associative array (%in) without conflict.

Output of CGI program **emp_add.pl**

emp_query.pl: The Query Program

The figure above shows the output of **emp_add.pl** on the browser window after you have pressed the *Add* button of the form shown previously. Note the hyperlink which offers to show all lines of the database. A click here runs the **emp_query.pl** program and displays the list of people as elements of a table. Here's the listing of the program:

```
$ cat emp_query.pl
#!/usr/bin/perl
require "web_lib.pl" ;

open (OUTFILE, "/home/sumit/public_html/emp_out.lst") ;
print "Content-type: text/html\n\n";
&Htmlheader("Retrieving from Database", "Result of Query:") ;
print "<table border=1 bordercolor=magenta bgcolor=cyan>" ;
print "<tr><th>Emp-id</th><th>Full Name</th><th>Designation</th>" ;
print "<th>Department</th><th>Date of Birth</th><th>Salary (\$)</th></tr>" ;

while (<OUTFILE>) {
    ($empid, $ename, $desig, $dept, $dtbirth, $salary) = split (/\|/) ;
    print "<tr><td>$empid</td><td>$ename</td><td>$desig</td>" ;
    print "<td>$dept</td><td>$dtbirth</td><td>$salary</td></tr>" ;
}
print "</table>" ;
&Htmlfooter ;
```

The table headers are printed with the `<tr>` and `<th>` tags. The program picks up each line of `OUTFILE`, splits it and then prints it as a table row with the `<tr>` and `<td>` tags.

Because of its powerful text manipulation capabilities, **`perl`** is the most widely used language for CGI programming on the Internet. However, CGI is a security threat on the Internet as a result of which the server administrator often disables CGI operation by individual users. In case you find this restriction on your system, contact the administrator.

Output of CGI program **`emp_query.pl`**