ADVANCED WEB TECHNIQUES

GOALS FOR THIS PRESENTATION

Overview of alternatives for doing web-based applications
Basic concepts behind all application development
Specific instructions for forms and counters, without requiring programming
Introduction to techniques that do require programming

BOOKS

O'Reilly has excellent books on CGI programming, Javascript, and Perl.

COMMONLY USED ADVANCED TECHNIQUES

Canned: forms and counters
Programming environments
  Client-side: embedded languages: Java, Javascript
  Server-side: embedded languages: PHP, Javascript, Filemaker Pro/Lasso
  Server-side: application languages: Visual Basic, Perl, Java
Environments at RU:
  Counter.rutgers.edu: one counter per person, special arrangements for depts
  Form program on RCI for simple form handling
  Filemaker Pro, embedded language with database; easy on a PC/Mac server
    where security isn't an issue; not currently planning a central service, but
    possibly could do if necessary
  VB talking to Access or Oracle; will announce service shortly, probably could do
    now if necessary
  Perl under Unix as server-side CGI language: common on Unix systems where
    security isn't an issue, both Sun and Linux
    by summer, plan to support PHP and minisql on RCI

THE CGI ENVIRONMENT

This is for classic Unix servers, but other environments will be similar.

Introduction to HTTP

As far as a normal user is concerned, you send a URL and get back a document.
There's a bit more going on: The request and response both send a good deal of
info about themselves and various supported options, and there is a specific
command/response protocol.
The most common command is GET, which normally retrieves a web page. Here's
an example:
Browser sends:
GET /~hedrick/testform.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (WinNT; I)
Host: geneva.rutgers.edu
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png,*/*
Accept-Language: en
Accept-Charset: iso-8859-1,* utf-8

Response from server:
HTTP/1.0 200 OK
Server: SWS-1.0
Date: Mon, 02 Mar 1998 22:51:07 GMT
Content-Type: text/html
Last-Modified: Sun, 01 Mar 1998 18:23:58 GMT
Connection: Keep-Alive
Content-Length: 245

<html>
...

For forms work, POST comamnd. Used to send the data from a form. Example:

The browser sends this:
POST /cgi-bin/form HTTP/1.0
Referer: http://geneva.rutgers.edu/~hedrick/testform.html
Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (WinNT; I)
Host: geneva.rutgers.edu
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png,*/*
Accept-Language: en
Accept-Charset: iso-8859-1,* utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 46

var1=aa&var2=bb&FORM_DATAFILE=%7Ehedrick%2Ffoo

Here is the response from the server:
HTTP/1.0 200 OK
Server: SWS-1.0
Date: Mon, 02 Mar 1998 22:51:12 GMT
Content-Type: text/html
Connection: Keep-Alive
Content-Length: 312

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
...

 
Requests for an application look just like requests for any other document. The server is configured to know that a certain area is actually CGI. So you might request http://foo.rutgers.edu/cgi-bin/finger. The server config file says that /cgi-bin is where CGI scripts are stored. So all requests to there are treated specially. They execute the thing and show the results, rather than just sending it as a file.

There are several kind of interface. All produce the same result: a program is called, some input is passed from the web transaction. The program generates either a complete header and then a web page, or a partial header and a web page. Here’s a partial header:

```
Content-type: text/html

<html>
This is a page
</html>
```

The whole header is as shown above. (To specify the entire header, you must call the script nph-xxx, for many servers.)

It is also possible for the program to say

```
Location: http://foo.rutgers.edu/~hedrick/foo.html
```

which will redirect to that URL.

The CGI interface: when your program runs, it is passed a list of standard env variables, e.g. REMOTE_HOST, HTTP_USER_AGENT. It could produce different document for internal and external users, or different web browsers. REMOTE_USER can be set if authentication is configured in the server.

PASSING DATA TO AN APPLICATION

There are three different ways to pass data. The first two use the URL:

**Path**

If the URL has components after the script name, they are passed as PATH_INFO, e.g.

```
http://foo.rutgers.edu/cgi-bin/foo/bar/baz
```

would call the script foo, and set PATH_INFO to /bar/baz. It also sets PATH_TRANSLATED to the path which would be used if /bar/baz were presented as a URL. So if you did

```
http://foo.rutgers.edu/cgi-bin/foo/~hedrick/foo
```

PATH_TRANSLATED would be /export/home/hedrick/public_html/foo
Query

If the URL has a ?, data after that is passed through QUERY_STRING, e.g.

http://foo.rutgers.edu/cgi-bin/finger?hedrick

would pass QUERY_STRING=hedrick.

Typically you would configure finger so that if QUERY_STRING is null, it displays

```html
<HTML>
<ISINDEX PROMPT="User: ">
This is a gateway to "finger". Type a username
</ISINDEX>
</HTML>
```

<ISINDEX> will generate a box, and send the result back to the same URL with ?.

With forms, the query has the form var=value&var=value...

Post

Post is intended primarily for use with forms. Sends data in the request after the header. In the case of a form, it's send just as in the query string:

```
var=value&var=value....
```

When your program is called, CONTENT_LENGTH is set to the amount of data.

Forms

```html
<FORM ACTION="/cgi-bin/form" METHOD=POST>
Var 1:  <INPUT TYPE="text" name="var1"><br>
Var 2:  <INPUT TYPE="text" name="var2"><br>
<INPUT TYPE="hidden" name="FORM_DATAFILE" value="~hedrick/foo">
<INPUT TYPE="submit">
```

Must have action and submit, and normally method. When you click on the submit button, it generates a POST transaction, with a line containing all the variables and values. In this case

```
var1=xxx&var2=yyy
```

Here are the various type of INPUT tag:

```
<input type="text" name="varname" size=40 maxlength=80 value="default">
name is required. value initializes the field with a default.
```

```
<input type="password" name="pw">
just like text, but doesn't echo
```

```
<input type="checkbox" name="sex" value="male" checked>
```
<input type="checkbox" name="sex" value="female"> checkbox. checked sets it by default. You get sex=male in the variables.
Warning: it's possible to check both male and female.

<input type="radio" name="sex" value="male" checked>
<input type="radio" name="sex" value="female"> similar, but you can just set one. To guarantee exactly one, you must initialize one. Otherwise you may get none.

<input type="submit" name="submitted" value="yes"> value will be used for the button label. If you include a name, the name and value will be included

<input type="image" src="foo.gif" name="map" align=top> clicking in the image acts like a submit. It will add two properties, name.x and name.y, in this case map.x and map.y. These get the location in the image.
Align works like align in IMG

<input type="reset" value="start over"> resets all fields to initial values. Value used for button label.

<input type="file" name="fn"> normally sends the filename just as text. however if you add ENCTYPE="multipart/form-data" to the FORM tag, it will upload the file itself. Your script must of course be prepared to accept it. I'm not describing the procedure here.

<input type="hidden" name="foo" value="stuff"> includes foo=stuff. Used to pass state.

<select name="flavor">
<option value="vanilla" selected>vanilla
<option value="strawberry">strawberry
</select>
sends flavor=vanilla or whatever. selected means this one starts out selected.
The text after option is what goes in the menu.

<select name="flavor" multiple size=2>
allows you to select multiple options. (With PC, you have to hold control.)
size=2 says 2 options visible in scroll box.
<textarea name=address rows=4 cols=40>
your address here
</textarea>
will send address=... with the whole text. CR will be encoded as %0D%0A.

The program “forms”. Man page and example files are enclosed. Installed on rci.

COUNTERS

A counter is simply a CGI returning a GIF. That allows for inline. Can also return text, but then need server side include to use it.
To set up a single counter, go to http://counter/counter.html.
Create a counter using user@rci-dialup. What it actually creates is a file containing a number which it increments, and the last host a request came from.
To use it:

<IMG SRC="http://counter.rutgers.edu/cgi-bin/counter?df=username@rci-dialup&dd=57chevy">
This calls a CGI script that increments the counter specified by df and returns a GIF showing the count. It saves the hostname, and will not increment if the request comes from the same hostname.
The web page shows many other options, but df and dd are the minimum useful. Two dd's that I recommend are 57chevy and goudy. Goudy is a nice-looking, dignified font.
Here's a nice conservative option:

```
<img src="http://counter/cgi-bin/counter?df=clh@eden-dialup&dd=goudy&ft=2&frgb=000000">
```

**SECURITY ISSUES**

At least with Unix, it is important not to allow users from the net to do anything other than the specific options you intended. Particularly important not to let them get a shell.

CGI scripts have a history of problems. If you write in shell or perl, it is easy to end up executing the text that the user supplied. Then he can give you a query like `http://foo/cgi-bin/appl?thing;rm -rf /` stuff in C has a history of buffer overruns.

There is such a consistent history of this that we don't allow users to do their own CGI programs on shared machines such as RCI. The form program was written by an expert and reviewed by another expert. The design is paranoid. There are very few other CGI programs that we have any confidence in.

PHP has special features to limit what it can do. So even if the program is buggy, it can only cause problems for one user.

It's hard to say much about security other than that either you have to use canned applications done by someone competent, or you must use a programmer who really knows what he is doing.

**APPLICATIONS PROGRAMMING**

Examples of applications in Javascript and PHP are included. Both are embedded languages. The difference is that Javascript is typically run on the client side, PHP on the server side.