Server-Side Web Programming Intro

To make a Web server based "program"

- You have to get data in (from user-agent to server)
- Then process the data, perform some task, etc.
- You have get data out (from server to user-agent)
  - Input: <form> data, URLs, HTTP headers (browser type, IP, cookie, etc.)
  - Output: Some form of data usually HTTP, GIF, JPEG, etc. and the appropriate header (MIME type ,cookie, etc.)
- You will need to fix the stateless nature of HTTP to do anything meaningful
  - Sessionization via cookies, URLs, hidden fields, etc.
- You should provide features to make programming easy not only do the previous things simply but allow for access to DBs, output HTML and Web site widgets, etc.

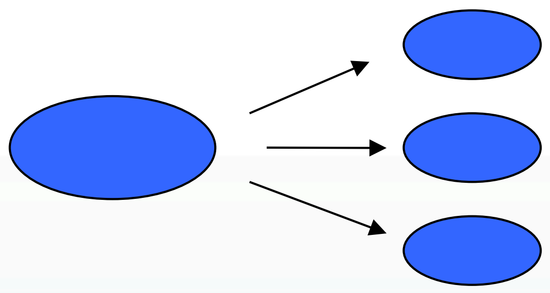
The “Hamburger” Model

- A funny way you might think of this is that your “program” – the thing you do is the “meat” of your hamburger and the buns are the input and the output
  - Input - ”Top Bun” (Msg body, query string, environment variables)
  - The Program – “Meat” (what does the work takes the input and does some calc, fetches data, etc.)
  - Output – “Bottom Bun” (The response + headers sent back)

Redux -3 Server-Side Programming Models

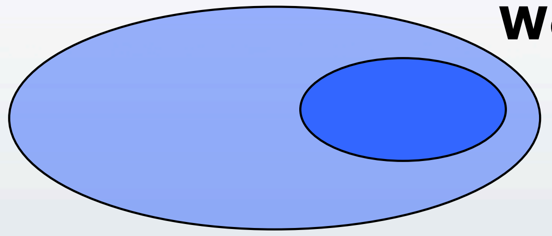
- Revisit this again because it is that important – there are just 3 models at play here really
  - #1) Classic CGI model – “fork and exec”
    - Web server creates new child process, passing it request data as environment variables
    - CGI script issues response using standard I/O stream mechanisms
  - #2) Server API model
    - Web server runs additional request handling code inside its own process space
  - #3) Web application frameworks
    - Web server calls API application, which may manage request within its own pool of resources and using its native objects

3 Server-Side Programming Models Redux

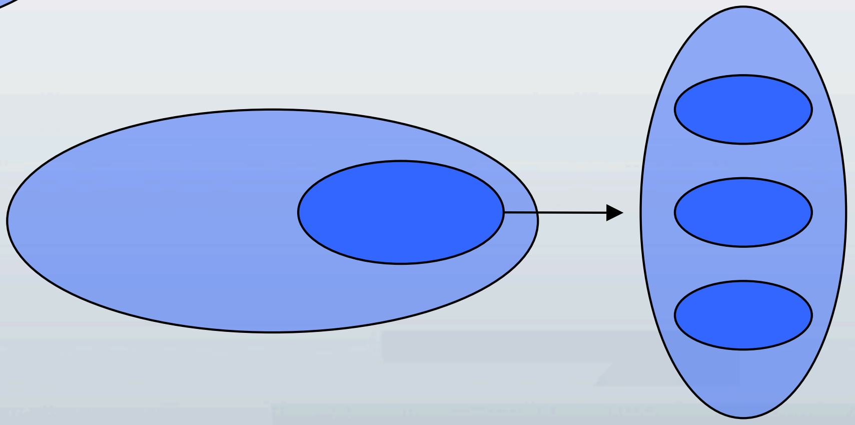


Classic CGI "fork and exec"

Server API running inside Web server's address space



Web application framework running inside Web server process but managing its own pool of resources via IPC



3 Server-Side Programming Models

- Each model has its pros and cons
  - Classic CGI model
    - Pro: isolation means easiest in principle to secure, least damaging if something goes wrong
    - Con: isolation makes it slow & resource intensive
  - Server API model
    - Pro: very fast & low overhead if written properly
    - Con: hard to write; blows up server if done wrong
  - Web application frameworks
    - Pro: ideally combines efficiency of API model with safety of CGI; adds helpful encapsulation of routine tasks like state management
    - Con: built-in tools can be resource hogs in wrong hands; ease of use may encourage carelessness

3 Server-Side Programming Models

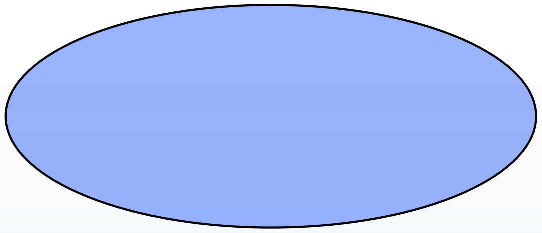
- Many examples of each
  - Classic CGI
    - Scripts written in Perl
    - Programs written in C
  - Server API
    - Apache modules
    - ISAPI filters and extensions
  - Web application frameworks
    - All descended from Server Side Includes (SSI), original “parsed HTML” solution that allowed interspersing of executable code with markup
    - ASP, ASP.NET, Cold Fusion, JSP/Servlets, Python, PHP, etc.

Theoretical Trade-offs

- As we saw there are many approaches to accomplishing Web programming
  - No one approach works for all situations
- Speed of ISAPI vs. simplicity of a PHP script
  - Some show you low-level details like headers, query strings, etc. very explicitly and some do not
  - Some are harder to work with than other
  - Trade-off: Configurability vs. Simplicity
  - We also see that some higher level frameworks will trade-off ease of programmer effort for overhead and thus speed or scale
  - Portability or reliance to a particular platform is also seen
- We start with low level old style CGI to see the underlying sameness of all server-side Web coding

Model 4?

The Web Server is the app / the app is the Web server?



Common in the NodeJS world our code defines routes and the app is the server. This is both a good idea (limited surface area) and an insane idea (you have no idea what serving HTTP really means) Suggestion; Use Node behind a HTTP proxy or in a co-server model

Interestingly this is an old model that was abandoned and is now back

CGI (Common Gateway Interface)

- Simple standard defining the way for external programs to be run on Web servers
- In short CGI defines the way that data is read in by external programs and what is expected to be returned
  - “Hamburger” model
- Useful CGI related resources
  - http://www.cgi-resources.com
  - http://www.w3.org/CGI/
  - O’Reilly’s CGI Book - http://www.oreilly.com/catalog/cgi2/

CGI and Perl

- CGI is often mentioned in the same breath as Perl, this is somewhat misleading
- CGI does not specify the usage of a particular language. If this is the case why Perl?
  - Perl was commonly found on UNIX machines which were the first Web servers
  - Perl is very good at string manipulation which is one of the main tasks when writing a CGI program
  - Perl is pretty easy to hack around with
  - Of course as an interpretted language you can see Perl partial to blame for CGI's speed problems
    - Soluion: Use mod_perl
    - Best Solution: Recompile as a C program

First Example in Perl

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "<html><head><title>Hello!</title>”;
print "</head><body bgcolor='yellow'>\n";
print "<h1>Hello from CGI</h1>\n”
print "</body>\n</html>";
```

- Notice the Content-type: header. That’s 50% of CGI’s “magic”
- In this case we use a .pl extension though you might want to use .cgi or even better something else or even nothing.
- We also probably place the code in the cgi-bin directory
  - Good practice in some ways, but again change the name

Note: We probably should use the #!/usr/bin/perl -wT to invoke Perl. The w flag shows warnings and the T flag turns on taint checking which checks incoming data a little more carefully.

First Example in C

- As said before language is irrelevant in CGI, so we recode helloworld in C as a demo

```
#include <stdio.h>
main()
{
 printf("Content-type: text/html \n\n");
 printf("<html><head><title>Hello World from CGI in
   C</title></head><body>");
 printf("<h1 align='center'>Hello World!</h1>");
 printf("</body></html>");
}
```

Don't Reinvent the Wheel

- There isn't much to do in CGI
  - Main tasks read in user submitted data and write out HTML and headers
  - The real programming is outside of this!
- Plenty of room to screw-up
  - Assuming that what is sent in is correct or safe
  - Not handling error conditions or exposing sensitive mechanisms.
- Rather than reinventing the wheel you could utilize a CGI library such as Lincoln Stein's CGI.pm module for Perl http://search.cpan.org/dist/CGI.pm/

Input/Output with CGI

- There are two forms of input to CGI programs
  1. User supplied data
  2. Environment supplied data

- User data generally comes from form fill-outs, clicks, etc. and is passed either using the GET or POST method while environment data is related to the environment in which the program is run and is mostly related to the various HTTP headers passed by the invoking user agent in combination with some local server variables.

- As shown by the previous example, output from CGI is a Content-type: header with the appropriate type (often text/html) followed by the specified content (often HTML)

Environment Variables

- These variables correspond to HTTP headers submitted to the running program:
  - HTTP_ACCEPT, HTTP_USER_AGENT, HTTP_REFERER REMOTE_HOST, REMOTE_ADDR
- Submitted data
  - QUERY_STRING, CONTENT_TYPE, CONTENT_LENGTH, REQUEST_METHOD
- Server or program related data
  - SERVER_NAME, SERVER_SOFTWARE, SERVER_PROTOCOL, SERVER_PORT, DOCUMENT_ROOT, SCRIPT_NAME
- There may be numerous others depending on the server in play.
- In the case of Perl, environment variables are stored in a hash named %ENV so we can easily access or print them as shown in the next example

Environment Variables Example without CGI.pm

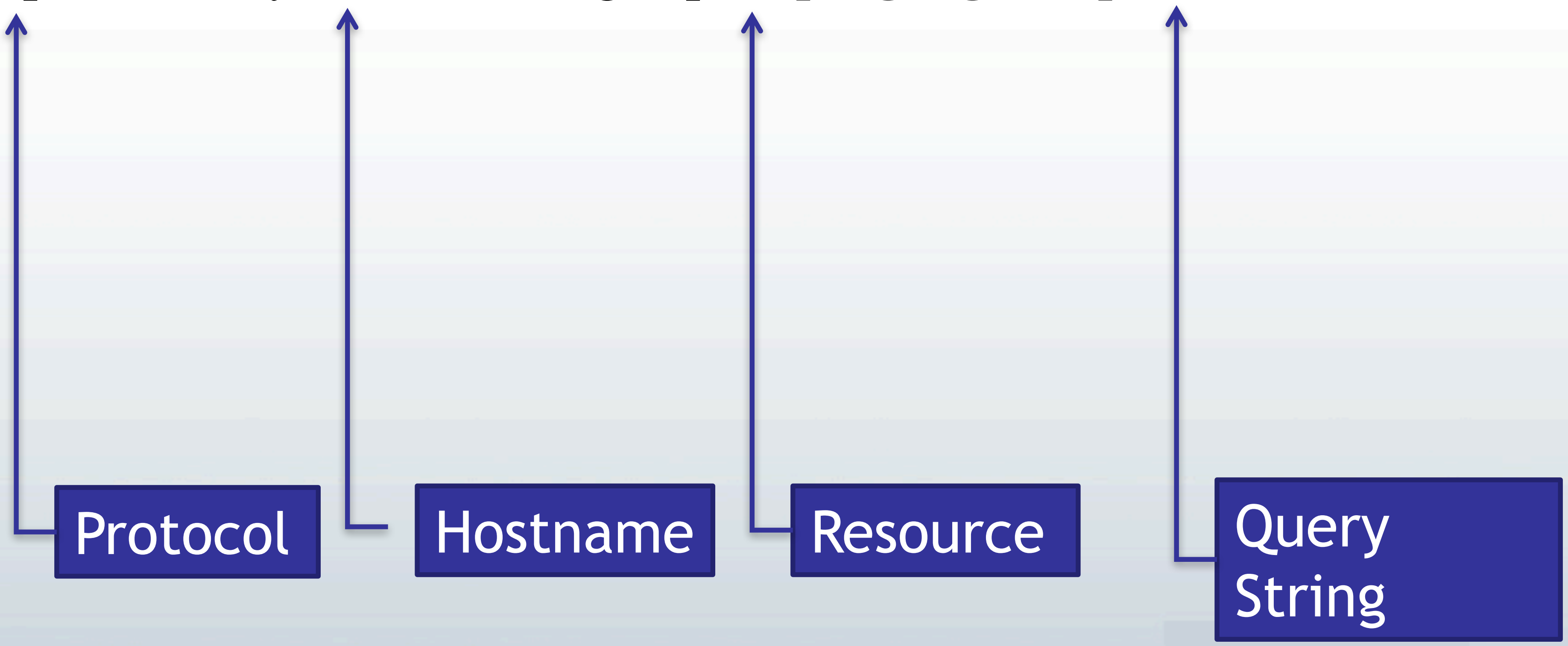
```
#!/usr/bin/perl
# print HTTP response header(s)
print "Content-type: text/html \n\n";

# print HTML file top
print <<END;
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html><head><title>Environment Variables</title>
</head><body><h1 align="center">Environment Variables</h1><hr />
END

# Loop over the environment variables
foreach $variable (sort keys %ENV) {
  print "<b>$variable:</b> $ENV{$variable}<br />\n";
}
# Print the HTML file bottom
print <<END;
</body></html>
END
```

Environment Variables Example with CGI.pm

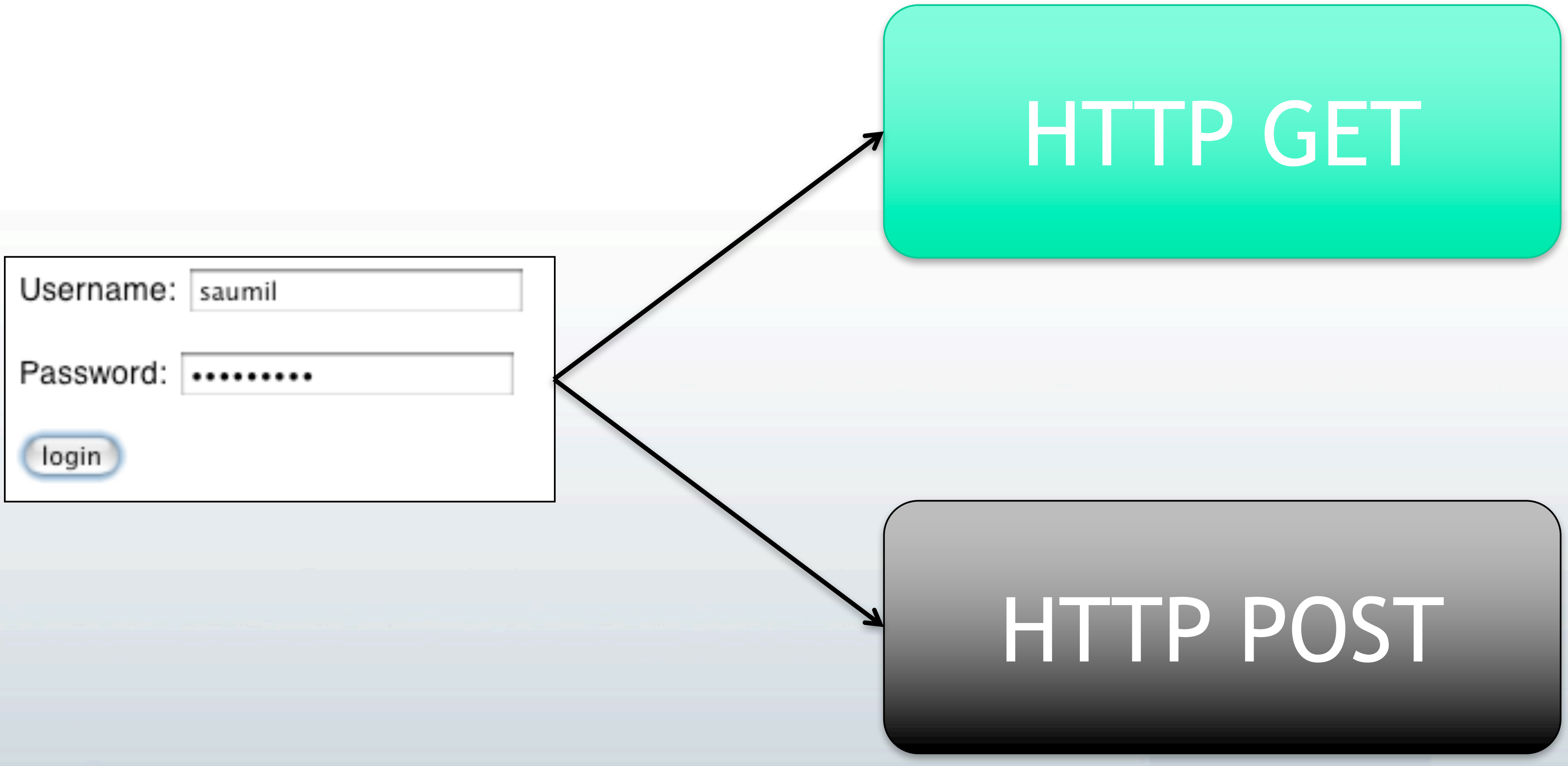
```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);
print header;
print start_html("Environment Variables");
print "<h1 align='center'>Environment Variables</h1><hr
   />";

foreach my $key (sort(keys(%ENV))) {
   print  "$key = $ENV{$key}<br />\n";
}

print end_html
```

Back to the URL

http://www.xyz.com/catalog.asp?dept=gadgets&product=7



HTML Forms


Username: saumil

Password: ••••••••••

login

HTTP GET Data Pass

```
GET /printvars.php?login=thomas+powell&password=not+secret HTTP/1.1
Host: demo.example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:13.0)
Gecko/20100101 Firefox/53.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://demo.example.com/examples.html
..
```

And the Response

```
GET /printvars.php?login=thomas+powell&password=not+secret HTTP/1.1
Host: demo.example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:13.0)
Gecko/20100101 Firefox/52.0.1
......
```

```
HTTP/1.1 200 OK

Date: Mon, 30 Jul 2016 01:25:48 GMT

Server: Apache/1.3.39 (Unix) PHP/5.2.5 mod_ssl/2.8.30 OpenSSL/0.9.8g

X-Powered-By: PHP/5.2.5

Keep-Alive: timeout=15, max=94

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: text/html


..... HTML data follows .....
```

HTTP POST Data Pass

```
POST /printvars.php HTTP/1.1
Host: demo.example.com
User-Agent: Mozilla/5. (Macintosh; Intel Mac OS X 10.7; rv:13.0) Gecko/20100101
Firefox/53.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://phpdemo.example.com/forms.html
Content-Type: application/x-www-form-urlencoded
Content-Length: 31

login=thomas+powell&password=not+secret
```

Response was same so omitted

Reading Data from Users

- Libraries like CGI.pm make it easy to read data in, they put it in a hash for us (no matter the method) by name/value pairs.

- We'll see in scripting languages like PHP it gets even easier!
- So given

```
<html>
<head><title>Simple Form</title></head>
<body><h1>Form Test</h1><hr>
<form action="/cgi-bin/getdata.cgi" method="get">
Name: <input type="text" name="username"><br>
Password: <input type="password" name="password"><br>
Magic Number: <input type="text" name="magicnum" size="2"
   maxlength="2"><br>
<input type="submit" value="send"></form>
</body>
</html>
```

Reading Data from Users Contd.

- We parse the data like so

```
    #!/usr/bin/perl -wT
use CGI qw(:standard);
use strict;

print header;
print start_html("Form Result");
print "<h1 align='center'>Form Result</h1><hr>";

my %form;
foreach my $p (param()) {
   $form{$p} = param($p);
   print "$p = $form{$p}<br>";
}
print end_html;
```

Reading Data from Users Contd.

- Using a library like CGI.pm handling post'ed data is performed in the same way
- We can also query directly for known name value pairs rather than just access what is passed up
  - Obviously this type of idea of just taking whatever the user passes us seems a tad insecure. You should generally only look at name-value pairs you know of and meet your required type/size constraints
  - Letting an environment immediately create global vars (or reset the contents of existing vars) might be really unsafe

CGI Troubles

- Since we are on the topic of security, why do people think CGI is insecure?
  - Often CGI programs run at higher permissions than they should be
  - Sometimes people use the shell
    - imagine form data being executed at a shell!
  - Programmers trust input way too much
    - Cross site scripting, code injection, etc.
    - **This isn't unique to CGI,** but the architecture particularly if they use shell scripts seems to make things worse

CGI Troubles

- Speed Problems
  - Every run of a CGI program server sets up environment, loads script/program, and runs it and then rips it down when done
  - Multiple users will equal many CGIs running
  - Often programmed in an interpreted language like Perl.
- Solutions
  - Keep the script loaded in memory and running as a co-process (e.g. FastCGI - www.fastcgi.com)
  - Use an embedded script interpreter (mod_perl)
  - Move to a server API program (e.g. ISAPI extension or Apache module)

CGI Troubles

- Problem with apparent complexity
  - A lot of details are still exposed to the programmer
  - Using a library like CGI.pm much can be hidden but you still have to understand headers, environment variables, etc. more than you might see in other environments
  - Some areas like session management are not abstracted as well in CGI programming as in more modern frameworks

  - Solution: Move to a scripting environment like PHP that hides these details better - but is their a trade-off there?

- Yet despite being very archaic CGI it is still heavily used by certain types of Web developers and it is useful as it demonstrates the details of server-side programming which is always there regardless of framework in play – dress it up, call it what you want but it all comes down to the same inputs, outputs, and network characteristics – better to know that than not!