CSE135

- A Review of Some Fundamental Ideas
- Generation of App Patterns
- HTTP Overview
- Web Servers Overview

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, cookies, etc.)
    - Output: Some form of data usually HTTP, GIF, JPEG, etc. and the appropriate header (MIME type, cookies, 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
  - #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
    - Pro: isolation makes it slow & resource intensive
    - Con: isolation makes it slow & resource intensive
    - Scripts written in Perl
    - Programs written in C
  - #2) Server API model
    - Web server runs additional request handling code inside its own process space
    - Pro: very fast & low overhead if written properly
    - Con: hard to write; blows-up server if done wrong
    - Apache modules
    - ISAPI filters and extensions
  - #3) Web application frameworks
    - Web server calls API application, which may manage request within its own pool of resources and using its native objects
    - 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
    - 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.
  - 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 very cool idea (no need to learn and an insane idea (you have no idea what servicing HTTP really means) Suggestion: Use Nginx behind a HTTP proxy or in a co-server model
    - Interestingly this is an old model that was abandoned and is now back
- Theoretical Trade-offs
  - 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 some of programmer effort for speed 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

CGI (Common Gateway Interface)

- Don't Reinvent the Wheel
  - Simple standard defining the way for external programs to be run on Web servers
    - Useful to study to understand the sameness of server-side web programming
    - not useful to implement as production
  - In short CGI defines the way that data is read in by external programs and what is expected to be returned
    - "Hamburger" model
  - Archaic but useful CGI related resources
    - http://www.cgi-resources.com
    - http://www.w3.org/CGI/
    - O'Reilly's CGI Book - https://www.oreilly.com/catalog/cgi/
  - 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 interpreted language you can see Perl partial as blame for CGI's speed problems
        - Solution: Use mod_perl
        - Best Solution: Recompile as a C program
    - The jargon mix-up is not unique to CGI/Perl
      - see Ruby on Rails for example
  - Example
    - First Example in Perl
      - 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 "use strict;" and "use" to invoke Perl. This will flag things and the -T flag turns on taint checking which checks incoming data a little more carefully.
    - First Example in C
    - There really 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 libraries like Lincoln Stein's CGI.pm module for Perl were introduced.
      - These things went away, but the pattern of abstracting the common started at the dawn of the web!
- Input/Output with CGI
  - There are two forms of input to CGI programs.
    - 1. User supplied data
      - User data generally comes from form fill-outs, clicks, etc. and is sent either using the GET or POST method
    - 2. Environment supplied data
      - 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.
  - 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
    - Environment Variables Example without CGI.pm
    - Environment Variables Example with CGI.pm
      - environment Variables Example with CGI.pm (DEPRECATED) - used as demonstrative example

HTML Forms

- Back to the URL
- HTTP-GET
  - Data Pass
  - Response
- HTTP-POST
  - Data Pass

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
  - We parse the data like so
- Using a library like CGI.pm handling post/cut 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 bad 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

- 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**. Sure the architecture particularly if they use shell scripts seems to make things worse
- 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 - https://en.wikipedia.org/wiki/FastCGI)
    - Use an embedded script interpreter (mod_perl)
    - Move to a server API program (e.g. ISAPI extension or Apache module)
- Problem with apparent complexity
  - A lot of details are still exposed to the programmer
  - Using an abstraction 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?

Why should we care?

- despite being very archaic CGI still exists a little bit
- the new function of server-side programming is always there regardless of framework in play
- 我们可以把CGI那些然后称之为任何名字，但是
  - **it all comes down to the same inputs, outputs, and network characteristics** - better to know that than not!