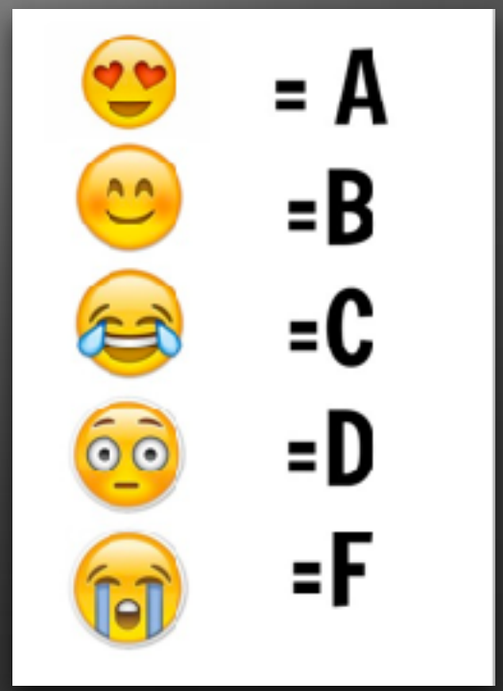
# Design Problem: Stateful CLI Calculator

CIS 322
Feb 16, 2017

Today I'm hoping to demystify cookie/session a little bit and show connecting to the database through collaboratively implementing a strange little calculator application. I'm hoping we can do this in 60 minutes or less.

# Midterm Points

- 24
- 7
- 1
- 0
- 0



# From the Dailies

- Finished my 2nd code review, really liked his style of code and going to implement some of it into my future programs.
- Asked a bunch of questions on piazza! got my work cut out for me before the due date of project 6
- Fixed git repo; pushed create_user route/htmls and login route/htmls. Plan to post a couple clarifying questions to piazza and finish with dashboard tomorrow.
- I have two other midterms Thursday and Friday so the remainder of my week will be spent studying for those. I plan on starting assignment 6 Friday after my midterm.

I wanted to share some dailies with you all again. I'm glad to hear that you're getting value out of the code reviews. I'm also glad that many of you have had time to read the requirements and start posting questions. I have added a FAQ to the end of the assignment to capture some of the questions received and answers given. This has clearly been a busy week for most of us.

# GET and POST

- HTTP 1.1 is specified using a Request For Comment (RFC)
- Specification in 1999 RFC2616
- GET
  - Sometimes cache-able
  - Parameters in URL
  - Flask - request.args
- POST
  - Not cached
  - Parameters in request body
  - Flask - request.form

I've been asked a few times what the difference between GET and POST requests are… I must admit to ignorance though I have had some suspicions… So, I looked up an RFC this morning to get a better idea.

It looks like the big protocol difference between GET and POST in HTTP 1.1 was around cache behavior for GET vs POST. POST never caches and is intended to support sending data to the server. GETs behavior is complicated but seems to be built around the underlying resources being the same each time a specific URL is requested.

An anecdotal difference comes from the way web servers may have been implemented in the past. The first time I ran a graphical web browser, it was on a Macintosh LC II; 4MB of ram total (system and video), 40 MB HD, 16 MHz processor with no cache, 12 inches of 256 color, and 9600 bps of bandwidth. One of the tricks to make the computer run faster was to switch from color to b/w since that would free up a bunch of memory and processor that would otherwise be spent on graphics. As fast as technology moves, we still need to acknowledge that there is a history that causes certain design decisions to become encoded in our solutions.

When we read an input in a language like C we need to decide upfront how much data to grab at a time. The first line of the HTTP request needs to give the route the request targets and requires some parsing; making it convenient to hold in memory all at once. When 8 MB of RAM is a lot of RAM for a computer, reserving 1k or more for each request being serviced could be problematic. After the request is parsed and can be routed the data can be streamed to the target in whatever chunk size makes sense.

From RFC 2616 section 3.2
Note: Servers ought to be cautious about depending on URI lengths above 255 bytes, because some older client or proxy implementations might not properly support these lengths.

# Software Engineer

- Just a synonym?
  - Programmer
  - Developer
- What should “engineer” in a title mean?
  Not every welder is a structural engineer…
- What should it mean to be a software engineer?
- Solution architect is also a fun title… What should that title mean?


Let’s talk about what we want the software engineer term to mean…

I think the major differentiator between a programmer and developer is the amount of independent problem solving effort that should be expected. Programmers and developers are both very tactical, these are people who are actually getting the work done and understand the pesky details of the software and hardware. I see software engineering as being more strategic; rather than directly deploying tactics, identification of the appropriate tactics and order of tactics to meet the objective become major roles. Solution and enterprise architects work at an operational level; given some nebulous collection of organizational objectives, identify a coherent collection of strategic activities to meet the organizational objectives.

# CLI Calculator Features

- Want an application to do basic calculator operations
- Must be a command line application
- Must be able to execute other command line programs between calculator operations
- Would like to support multiple concurrent calculations

Getting back to engineering… Here's a problem for us to work through. Here are the requested features from the user. What data can we pull out of this that will help us to design a solution?

Which features are required? Which features are optional? Can we use that information to help break up the problem and delivery into smaller phases?

Let's pick out our tech stack and strategy.

# What does this have to do with web apps?

- Each invocation of an application is stateless like each individual web request.
- Persistent storage can be used for the application to lookup state. Web applications can do the same thing.
- To have concurrent workflows some thing is needed to identify which of many saved states to load.
  - Our calc app will uses a nonce of some form
  - Web applications will use cookies (generally just a nonce)

I want to try motivating/understanding the Flask session by having us grapple with the system challenge the session tries to solve. Hopefully this exercise will provide some insight. The exercise should also provide a place for remaining tactical questions about python and the psycopg2 driver.

```
[osnapdev@osnap-image ~]$ calc = 0
40
[osnapdev@osnap-image ~]$ calc x 0 # Zero out the calculator
[osnapdev@osnap-image ~]$ ls -l
total 20
-rwxr--r-- 1 osnapdev osnapdev  627 Feb 10 10:42 calc
drwxr-xr-x 2 osnapdev osnapdev 4096 Feb  5 13:27 keys
drwxr-xr-x 2 osnapdev osnapdev 4096 Feb  5 21:59 logs
drwxr-xr-x 9 osnapdev osnapdev 4096 Feb  6 06:22 lost
drwxr-xr-x 9 osnapdev osnapdev 4096 Feb  6 17:04 wsgi
[osnapdev@osnap-image ~]$ ls -l | wc -l
6
[osnapdev@osnap-image ~]$ calc + 6
[osnapdev@osnap-image ~]$ ls -l | wc -l keys
wc: keys: Is a directory
0 keys
[osnapdev@osnap-image ~]$ calc x 0 # Zero out the calculator
[osnapdev@osnap-image ~]$ ls -l | wc -l
6
[osnapdev@osnap-image ~]$ calc + 6
[osnapdev@osnap-image ~]$ ls -l keys | wc -l
5
[osnapdev@osnap-image ~]$ calc + 5
[osnapdev@osnap-image ~]$ ls -l logs | wc -l
4
[osnapdev@osnap-image ~]$ calc + 4
[osnapdev@osnap-image ~]$ ls -l lost | wc -l
10
[osnapdev@osnap-image ~]$ calc + 10
[osnapdev@osnap-image ~]$ ls -l wsgi | wc -l
15
[osnapdev@osnap-image ~]$ calc + 15
[osnapdev@osnap-image ~]$ calc = 0
40
[osnapdev@osnap-image ~]$
```

Long form example usage to get the approximate number of directory entries in my home directory and immediate child directories

```
[osnapdev@osnap-image ~]$ calc x 0 # Zero out the calculator
[osnapdev@osnap-image ~]$ calc + `ls -l | wc -l`
[osnapdev@osnap-image ~]$ calc + `ls -l keys | wc -l`
[osnapdev@osnap-image ~]$ calc + `ls -l logs | wc -l`
[osnapdev@osnap-image ~]$ calc + `ls -l lost | wc -l`
[osnapdev@osnap-image ~]$ calc + `ls -l wsgi | wc -l`
[osnapdev@osnap-image ~]$ calc = 0
40
[osnapdev@osnap-image ~]$
```

Same computation using some shell magic to reduce the typing