Dynamic Pages

This Many People Have Been to MY Page:

0123456789

WHEN DID YOU BECOME AN EXPERT IN PYTHON FLASK

THIS MORNING, IT WAS QUITE EASY ACTUALLY
Review: Markup Languages, URLs, and HTTP

• **Markup languages** help you define structure and appearance of a document
  • HTML <- you’ll use this a lot, plus CSS
  • XML <- generalization of HTML (btw, MS Office uses this for docx, pptx, etc.)

• **URLs** encode location information for accessing a resource in the request-response cycle
  • protocol://server:port/path?query#fragment

• **HTTP** is the standard protocol for communicating websites between servers and clients
  • **Requests** consist of **methods** like GET, POST, HEAD, etc.
    • Also, **headers** contain useful metadata for the server
  • **Response** contains a HTTP response code (e.g., 200, 404) as well as content
    • Headers also sent here to tell the client about the data being sent
Review: URLs

• Quiz 1.
  • https://dijkstra.eecs.umich.edu/kleach/eecs485/su20/potato.php?salty=true
  • There’s no fragment and there’s no port
    • The port is implied for websites, but it’s “missing” in the same way that specifying no fragment makes it blank (implicit “#”)

• URLs are an important artifact for web services
  • Web APIs are documented according to URL endpoints for various services
    • e.g., GitHub API: https://developer.github.com/v3/,
      Google search API: https://developers.google.com/custom-search/v1/overview,
      DialogFlow API: https://cloud.google.com/dialogflow/docs/reference/rest/v2-overview
Review: HTTP(s)

• HTTP (unsecure) and HTTPS (secure) are protocols for communication
  • curl –verbose http://umich.edu
  • telnet umich.edu 80
  • GET / HTTP/1.1
    • etc.
HTTP request methods

• The client’s request contains what it wants the server to do
• GET: request a resource
  • Example: load a page
• HEAD: identical to GET, but without response body
  • Example: see if page has changed
• POST: send data to server
  • Example: web form
HTTP request headers

• $ curl --verbose http://cse.eecs.umich.edu/ > index.html
  * Connected to cse.eecs.umich.edu (141.212.113.143) port 80 (#0)
  > GET / HTTP/1.1
  > Host: cse.eecs.umich.edu
  > User-Agent: curl/7.54.0
  > Accept: */*

• **Host** distinguishes between DNS names sharing a single IP address
  • Required as of HTTP/1.1
• **User-Agent**: which browser is making the request
• **Accept**: which content ("file") types the client will accept
User agent

• When a browser visits a page, it identifies itself with a User-agent string
  • For example, check yours out:
  • http://www.whatismyuseragent.net/

• Example from Google Chrome:
  • Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.103 Safari/537.36
  • Previously used to indicate compatibility with the Mozilla rendering engine
  • During the "browser wars", some web sites would only send advanced features to some user agents
HTTP status code

• Response starts with a status code
  • 1XX: Informational “hold on”
  • 2XX: Successful “here ya go”
  • 3XX: Redirection Error “go away”
  • 4XX: Error “u screwed up”
  • 5XX: Server Error “sry i screwed up lol”

• $ curl --verbose http://cse.eecs.umich.edu/
  > GET / HTTP/1.1
  < HTTP/1.1 200 OK

• $ curl --verbose http://cse.eecs.umich.edu/asdf
  > GET /asdf HTTP/1.1
  < HTTP/1.1 404 Not Found
HTTP response headers

• Headers accompany a response
• Most are optional

$ curl --verbose http://cse.eecs.umich.edu/
* Connected to cse.eecs.umich.edu
> GET / HTTP/1.1

... HTTP/1.1 200 OK
< Date: Tue, 12 Sep 2017 20:04:20 GMT
< Server: Apache/2.2.15 (Red Hat)
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
HTTP content type

• Content type describes the "file" type and encoding
  
  $ curl --verbose http://cse.eecs.umich.edu/
  * Connected to cse.eecs.umich.edu
  > GET / HTTP/1.1
  ...
  < HTTP/1.1 200 OK
  ...
  < Content-Type: text/html; charset=UTF-8
HTTP content type

• Content type describes the "file" type and encoding

```bash
$ curl --verbose
http://cse.eecs.umich.edu/eecs/images/CSE-Logo-Mobile.png > CSE-Logo-Mobile.png
* Connected to cse.eecs.umich.edu
> GET /eecs/images/CSE-Logo-Mobile.png HTTP/1.1
...
< HTTP/1.1 200 OK
< Content-Type: image/png
```
MIME Types

Content-Type: text/html; charset=UTF-8

• MIME: Multipurpose Internet Mail Extensions
• Way to identify files
• Browser can open or display content correctly
• <type>/<subtype>

Character encodings

Content-Type: text/html; charset=UTF-8

• Most people in the world use languages other than English
• ASCII -> char in C++

• 我太帅了 => ?
  • Encode with Unicode: \xE6\x88\x91\xE5\xA4\xAA\xE5\xB8\x85\xE4\xBA\x86
  • (think char x = 0xE6; in C; each of these is a byte in UTF-8)

• In olden times, you used to have to configure your browser ahead of time to know how to interpret a string – UTF-8 is a modern time saver!
Character Encodings

• Things look nasty if not specified!
POST request

• POST request sends data from the client to the server
• Commonly used with HTML forms

```html
<html>
<body>
    <form action="" method="post" enctype="multipart/form-data">
        <input type="text" name="username" placeholder="username"/>
        <input type="password" name="password" placeholder="password"/>
        <input type="submit" value="login"/>
    </form>
</body>
</html>
```
POST request

• No POST requests needed in project 1
• Example from project 2

$ curl

  --form 'username=awdeorio' \
  --form 'password=password' \
  --form 'submit=login' \
  localhost:8000/accounts/login/
HTTP versions

• See the HTTP version in the request and response
  
  $ curl --verbose http://cse.eecs.umich.edu/
  * Connected to cse.eecs.umich.edu
  > GET / HTTP/1.1
  ...
  < HTTP/1.1 200 OK

• Three versions:
  • HTTP/1.0 (old)
  • HTTP/1.1 (common)
  • HTTP/2 (new)
HTTP/1.0 vs HTTP/1.1

• How many TCP connections?

<html>
<body>
  <p>Block M</p>
  <img src="http://cse.eecs.umich.edu/eecs/images/CSE2017-hi-rev.png">
  <img src="http://cse.eecs.umich.edu/eecs/images/CSE2017-rev.png">
</body>
</html>

• HTTP/1.0: 3
HTTP/1.0 vs HTTP/1.1

• How many TCP connections?

• HTTP/1.1: 1
  • Reuse one HTTP connection
HTTP/2

• Methods, status codes, etc. same as HTTP/1.1
• One new feature: server push
  • Server supplies data it knows a web browser will need to render a web page, without waiting for the browser to examine the first response.

• Example: images in this page are loaded as PUSH commands are sent from the server after responding to the original GET /blah.html request
One-Slide Summary: Dynamic Pages

• **Static** web pages remain constant over time
  • **Static pages are boring**: No account management, no statefulness, etc.

• **Dynamic** web pages are built with *server software* that *generates* HTML or DOM subtrees *on the fly*!
  • Recall: HTTP is just sending strings back and forth...
    • Can’t you just write a script that prints a bunch of HTML out, then pipe it over HTTP? (Spoiler alert: yes, we use Python Flask in this class)

• Dynamic web pages come in two flavors
  • **Server-side dynamic**: Client gets an *entire new HTML page* on each request
  • **Client-side dynamic**: Client gets *DOM subtrees* on each request
    • Requires the use of JavaScript to modify the DOM on the fly in the client
Paleolithic era

• 1965 Gordon Moore proposes law
• 1966 Design of ARPAnet

• 1969 First ARPAnet msg, UCLA -> SRI
• 1970 ARPAnet spans country, has 5 nodes
• 1971 ARPAnet has 15 nodes
• 1972 First email programs, FTP spec
The internet ramps up

• 1983 ARPAnet uses TCP/IP; design of DNS; 1000 hosts on ARPAnet
The internet ramps up

• 1983 ARPAnet uses TCP/IP; design of DNS; 1000 hosts on ARPAnet
• 1985 symbolics.com (computer mfg) is first registered domain name

• 1988 Robert Morris accidentally takes over the Internet
• 1989 100K hosts on Internet
• 1990 Cisco goes public; Tim Berners-Lee creates WWW at CERN; 3M Internet users world-wide
Modern age

• 1993 WWW Wanderer
  • First crawler
• 1995 Yahoo, Amazon
• 1998 Google & PageRank
• 2003 Skype
Modern age

• 2004 Facebook founded
• 2006 Twitter founded
• 2010 Instagram founded
• 2019 Facebook has 2.4 B monthly active users
  • ~30% of humanity
  • https://investor.fb.com/investor-events/
Static content

• On the server side: HTTP servers are file servers
• On the client side: browsers are HTML renderers

• Example
  • `python3 -m http.server`
  • Copies files
Static vs. dynamic content

• Static content is the same every time
• Dynamic content changes
• Think of the things that are impossible with simple static pages
Static vs. dynamic content

• Static content is the same every time
• Dynamic content changes
• Think of the things that are impossible with simple static pages
  • Web search
  • Database lookups
  • Current time
  • # visitors to page
  • Everything
Project 1 = Static content

- Project 1: the pages are static

- Pages only change rarely via a manual process
  - You manually update data, templates (with a text editor)
  - Run `insta485generator` to produce new pages
  - Templates are a way to reduce the work of editing
  - Everybody gets same content until next manual update

- Generation of content not specific to each request
Aside: Jinja

• Project 1 has you build static HTML files from Jinja templates

```html
<!DOCTYPE html>
<html>
  <head>
    <title>{{ variable|escape }}</title>
  </head>
  <body>
    {% for item in item_list %}
      {{ item }}{% if not loop.last %},{% endif %}
    {% endfor %}
  </body>
</html>
```

```python
from jinja2 import Template
with open('example.html.jinja') as f:
    tmpl = Template(f.read())
print tmpl.render(
    variable = 'Value with <unsafe> data',
    item_list = [1, 2, 3, 4, 5, 6]
)
```
Aside: Jinja

• Project 1 has you build static HTML files from Jinja templates

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Value with &lt;unsafe&gt; data</title>
  </head>
  <body>
    1, 2, 3, 4, 5, 6
  </body>
</html>
```
Aside: Jinja... Still static!

• Project 1 has you build static HTML files from Jinja templates

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Value with &lt;unsafe&gt; data</title>
  </head>
  <body>
    1,
    2,
    3,
    4,
    5,
    6
  </body>
</html>
```

• The HTML files are built before being served up... What about dynamism?
Dynamic server content

• In the old days (1997?), almost all requests were just disk loads

• Computing the page dynamically was a mind-blowing idea; today it’s assumed

• Each request calls a function in the server software, which generates the response
Why is this useful?

• Most common pattern:
  • Store data in a database
  • Use jinja/templates to create HTML from that data
  • This way, users can change the data
Thought question

• All of these web pages are dynamic. What's the reason why each of these websites could not or were not implemented with static pages?

• Piazza
• Your course schedule on Wolverine Access
• Wikipedia articles
• cse.engin.umich.edu
Dynamic server content

• Project 2: what if we let users create posts?

• When a user creates a new post, save information in database

• When a user loads /u/deorio, call a function to look for newest posts for the user in the database
Dynamic server example

# hello.py
import flask
app = flask.Flask(__name__)

@app.route("/hello")
def hello_world():
    return "<html><body>Hello World!</body></html>"

if __name__ == "__main__":
    app.run()
Dynamic server example

• Run
$ python3 hello.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

• Browse to http://localhost:5000/hello

• NOTE: localhost == 127.0.0.1
Templating

• Instead of HTML in string, base on template

• Example: Python's jinja2 library

• Write an HTML file with special keywords
  • e.g., {% for post in posts %}

• Run it through a function, along with a data structure of values to fill in
  • e.g., template.render()
  • or flask.render_template() in project2
Template example

• Template is an HTML file containing jinja2 syntax

```html
<html>
<head><title>Hello world</title></head>
<body>
{% for word in words %}
{{word}}
{% endfor %}
</body>
</html>
```
Rendered template example

- Rendered template is a string with jinja2 template syntax "filled in"

```html
<html>
<head><title>Hello world</title></head>
<body>

hello

world

</body>
```
# hello.py
import flask
app = flask.Flask(__name__)

@app.route("/hello")
def hello_world():
    return "<html><body>Hello World!</body></html>"

context = {"words": ["Hello", "World!"]}
return flask.render_template("hello.html", context)

if __name__ == "__main__":
    app.run()
Principle: data/computation duality

• We think of data and computation as separate
• But, they are really two sides of the same coin

• Data instead of computation
  • Pre-computing and storing results

• Computation instead of data
  • Dynamically generating web pages
Project 2 = server-side dynamic content

Client specifies a URL
- This looks like a file path on the server (remember last lecture?)
- But server really runs a function, serves returned output

- How does function generate content?
  - State is stored in a database (SQLite)
  - Function issues SQL queries to get relevant state
  - Populates Python object
  - Renders template using object
  - Returns resulting HTML

- Generation of content specific to each request
Server-side vs. client-side

- With server-side dynamic pages:
  - must reload the page for the page to change
Server-side vs. client-side examples
Client-side dynamic pages

• Website is a program split into 2 parts
  • Client side (in web browser): sees what user wants to do, sends requests to server

  • Server side: responds to program in client side

• The web browser is the environment for the client-side program
How to tell client vs. server side dynamic

• Does the page reload?

• Does the page change after the server has sent it to the client?
  • e.g. a chat app must be client-side dynamic or you wouldn’t be able to see new messages after the page loaded
JavaScript

• The client-side dynamic part of a website is written in JavaScript

• Absolutely nothing to do with Java – Java was cool at the same time

• Web browsers have an interpreter/compiler for JavaScript
  • The speed of a web browser mostly depends on how fast its JavaScript engine runs
How JavaScript is used

• document API: change the DOM
  • Use JavaScript to form a small HTTP request to get a DOM subtree to insert in the current DOM

• Events: detect when user does something
  • Click
  • Mouse movement
  • Key press

  • You write an event handler in JS that takes action when such an event occurs!
    • “Asynchronous”
document API

• JavaScript can interact with the DOM
  • Recall: DOM is the data structure built from HTML

• The document API represents the web page loaded in the browser
  • Web page represented as a Document Object Model (DOM)

Document.body.html = ‘Declaration of Independence’;
How does JavaScript usually modify a page? Find a location in the DOM and modify it.

```html
<html><body>
  <div id="JSEntry">Loading ...</div>
  <script>
    document.getElementById("JSEntry").innerHTML = "Hello World!";
  </script>
</body></html>
```

<table>
<thead>
<tr>
<th>Before JS executes</th>
<th>After JS executes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading ...</td>
<td>Hello World!</td>
</tr>
</tbody>
</table>
DOM Diagram

```html
<html>
<head></head>
<body>
  <div id="JSEntry">Loading ...</div>
  <script>
    document.getElementById("JSEntry").innerHTML = "Hello World!";
  </script>
</body>
</html>
```
Thought question

• Does the document API change the HTML source of a page, or just the DOM?
Events

• Example: clicking a button

• Attach a function to a button
• Button causes an *event*
• *Event* runs a function
• Function modifies the DOM

• We'll talk about this later in the semester
URL routing

• Dynamic pages are created by executing a function at the time of a request

• When a client requests a URL, how does a server know which function to call?

• What about routes like a user page, where the function could have an input?
from flask import Flask
app = Flask(__name__)

@app.route('/u/<username>')
def show_user(username):
    return "hello {}!".format(username)

@app.route('/p/<postid>')
def show_post(postid):
    return "post {}!".format(postid)

if __name__ == '__main__':
    app.run()
Routing in Python/Flask

• Flask uses a Python *decorator* to describe routing
• A *decorator* is a function that changes the behavior of another function
• A decorator is a higher order function

```python
from flask import Flask
app = Flask(__name__)

@app.route('/u/<username>')
def show_user(username):
    return "hello {}!".format(username)
```
First class objects

• In Python, functions are *first class objects*
• Recall from EECS 280, that first class objects can be:
  • Passed as input
  • Returned as output
  • Created at runtime
  • Destroyed at runtime
Storing functions in variables

def fn():
    print("Hello!")

f = fn
f()
f()
Passing functions to other functions

def wrapper(fn):
    print("Start")
    fn()
    print("End")

def f():
    print("Hello")

wrapper(f)
Returning functions

def wrapper(fn):
    print("Wrapping")
    def wrapped():
        print("Start")
        fn()
        print("End")
    return wrapped

def f():
    print("Hello")

w = wrapper(f)
w()
Decorator pattern

def wrapper(fn):
    print("Wrapping")
    def wrapped():
        print("Start")
        fn()
        print("End")
    return wrapped

@wrapper
def f():
    print("Hello")

@wrapper
def g():
    print("Goodbye")
Decorators and Flask

```python
code snippet
from flask import Flask
app = Flask(__name__)

@app.route('/u/<username>')
def show_user(username):
    return "hello {}!".format(username)
```
Further reading

• Decorators can be extended to accept arguments
  @app.route('/u/<username>')
  def show_user(username):
      return "hello {}!".format(username)

• Flask's route() is a combination of two patterns:
  • Registering plugins with a decorator
    https://realpython.com/primer-on-python-decorators/#registering-plugins
  • Decorators with arguments
    https://realpython.com/primer-on-python-decorators/#decorators-with-arguments

• Tutorial on decorators
  https://realpython.com/primer-on-python-decorators/