Server-side Dynamic Pages

Slides by Andrew DeOrio

Agenda

- **Static pages vs. dynamic pages**
- Server-side dynamic pages
- Dynamic page URL routing

Static page example

- Example: https://andrewdeorio.com
- Check out the text
- Now, view source. You should be able to find all the text in the HTML.

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

Static content

- On the server side: HTTP servers are fileservers
- On the client side: browsers are HTML renderers

- Example
  - `python3 -m http.server`
  - Copies files

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

Server-side dynamic page example

- Example: https://github.com/mikecafarella
- Check out the text
- Now, view source. You can find the text.

- But, another user's github page is different, e.g., https://github.com/awdeorio
- The server generates these on-the-fly from a database

Agenda

- Static pages vs. dynamic pages
- **Server-side dynamic pages**
- Dynamic page URL routing

History

- In the old days (1997?), almost all requests were just disk loads
  - Static pages
- Computing the page dynamically was a **mind-blowing idea**; today it's assumed
  - Server-side dynamic pages

Server-side dynamic pages

- Server-side dynamic pages: Response is the output of a function.

1. Client makes a request
2. Server executes a function
   - Output is usually HTML
3. Server response is the output of the function

Server-side dynamic pages example

- Python/Flask library dynamically creates web pages

```
# 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()
```

Run `hello_world()` when client requests URL `/hello`

Return a string containing HTML

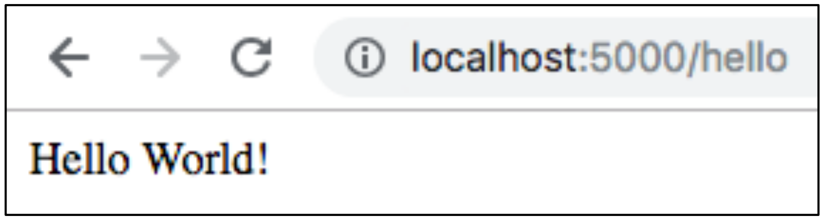
Debug server

Server-side dynamic pages example

- Run

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

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



Hello World!

- NOTE: `localhost` == `127.0.0.1`

Templating

- Do we have to hardcode HTML html in a string?
- Templates are a common way to generate server-side dynamic pages
- 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`
- Output is expanded HTML

Template example

- Template is an HTML file containing jinja2 syntax

```
<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 tempate syntax "filled in"

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

hello

world

</body>
```

Rendering templates with Flask

- Adapt Hello World example to render a template instead of returning hard coded HTML string
- Two files: Python program and HTML with template syntax
  - `hello.html` is identical to previous example

```
$ tree
.
├── hello.py
└── templates
    └── hello.html
```

Rendering templates with Flask

```
# 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()
```

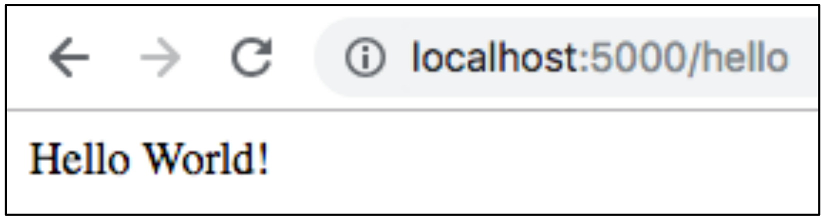
Render template, providing values from `context` dictionary

Run Flask example

- Run

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

- Browse to http://localhost:5000/hello
- Template is rendered and HTML text returned



Hello World!

- NOTE: `localhost` == `127.0.0.1`

Principal: data/computation duality

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

- Can substitute data for deterministic computation:
  - Pre-generated static pages instead of dynamic pages
  - Memoization, memcache storing SQL query results, ...

- Can substitute deterministic computation for data:
  - Dynamic pages instead of pre-generated static pages
  - MMU/virtual memory, dynamic web pages, ...

Project 2 = server-side dynamic content

Client specifies a URL

- This *looks* like a file path on the server
- 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

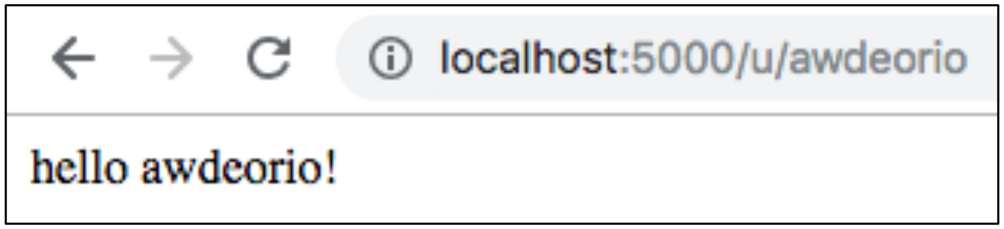
Agenda

- Static pages vs. dynamic pages
- Server-side dynamic pages
- **Dynamic page URL routing**

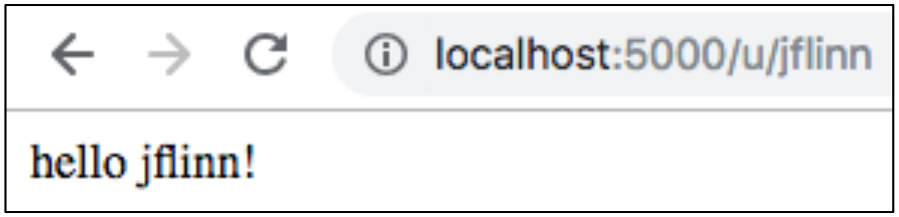
URL routing

- 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?

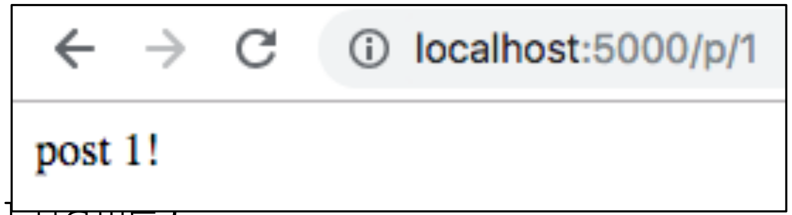
Routes with inputs


localhost:5000/u/awdeorio

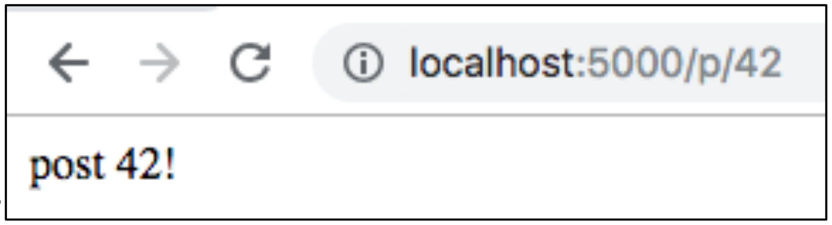
hello awdeorio!


localhost:5000/u/jflinn

hello jflinn!


localhost:5000/p/1

post 1!


localhost:5000/p/42

post 42!

```
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(pos

if __name__ == '__main__':
    app.run()
```

Static routes vs. dynamic routes

- Static pages: one URL maps to one file
  - Example: https://www.cse.umich.edu/index.html
    `index.html` is a plain text file on a server
- Dynamic pages: many URLs map to one function
  - Example: https://localhost:5000/u/awdeorio
    calls `show_user("awdeorio")`
  - Example: https://localhost:5000/u/jflinn
    calls `show_user("jflinn")`

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

URL routing

- Which URLs map to which functions?
- This is called *URL routing*
  - Not the same thing as TCP/IP packet routing

- A table mapping URL to function-reference
  - URL could be a pattern, e.g., '/u/**<username>**'

- Different libraries/frameworks describe routing with different techniques, let's see how Flask does it

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

```
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

Create a function at runtime

- Function object contains function execution "code"
  - Created when function is declared
- Function activation record contains references to function's local variables
  - Created when function is called

```
def show_user(username):
    return "hello {}!".format(username)

output = show_user("awdeorio")
print(output)
```

Visualize https://goo.gl/HtdUZp

Pass a function as input

- Pass a reference-to-function as a function input

```
def show_user(username):
    return "hello {}!".format(username)

def show_post(postid):
    return "post {}!".format(postid)

def show(func, slug):
    print(func(slug))

show(show_user, "awdeorio")
show(show_post, 1)
```

```
$ python3 test.py
hello awdeorio!
post 1!
```

Visualization https://goo.gl/EFWb7C

Return a function as output

- Return a reference-to-function as output

```
def get_view(url):
    if url.startswith("/u"):
        return show_user
    elif url.startswith("/p"):
        return show_post
    else:
        return None

def view(baseurl, slug):
    func = get_view(baseurl)
    print(func(slug))

view("/u", "awdeorio")
```

```
def show_user(username):
    return "hello ..."

def show_post(postid):
    return "post ..."
```

```
$ python3 test.py
hello awdeorio!]
```

Visualization https://goo.gl/oJKycP

Decorator

- A *decorator* is a function that transforms another function
- It can be used to add functionality

Simple decorators

```
def register(func):
    print("registered {}".format(func.__name__))
    return func

def show_user(username):
    return "hello {}!".format(username)

show_user = register(show_user)

print(show_user("awdeorio"))
print(show_user("michjc"))
```

What is the output?

Simple decorators

```
def register(func):
    print("registered {}".format(func.__name__))
    return func

def show_user(username):
    return "hello {}!".format(username)

show_user = register(show_user)

print(show_user("awdeorio"))
print(show_user("michjc"))
```

```
$ python3 test.py
register(show_user)
hello awdeorio!
hello michjc!
```

Visualization https://goo.gl/A9qkY4

Decorator syntactic sugar

```
def register(func):
    print("registered {}".format(func.__name__))
    return func

def show_user(username):
    return "hello {}!".format(username)

# "Manual" decorator happens here
show_user = register(show_user)

print(show_user("awdeorio"))
print(show_user("michjc"))
```

Decorator syntactic sugar

```
def register(func):
    print("registered {}".format(func.__name__))
    return func

@register # Python decorator syntax
def show_user(username):
    return "hello {}!".format(username)

~~show_user = register(show_user)~~

print(show_user("awdeorio"))
```

```
$ python3 test.py
registered show_user
hello awdeorio!
```

Visualization https://goo.gl/CmEfp3

Register pattern

- Use decorators to register functions
- Python/Flask creates a table of URL rule -> function
- Simplification: store function name instead of URL

```
VIEW_FUNCTIONS = dict()

def register(func):
    print("registered {}".format(func.__name__))
    VIEW_FUNCTIONS[func.__name__] = func
    return func
```

Register pattern

```
VIEW_FUNCTIONS = dict()
def register(func):
    print("registered {}".format(func.__name__))
    VIEW_FUNCTIONS[func.__name__] = func
    return func

@register
def show_user(username):
    return "hello {}!".format(username)

@register
def show_post(postid):
    return "post {}!".format(postid)
```

What is the output?

Register pattern

```
VIEW_FUNCTIONS = dict()
def register(func):
    print("registered {}".format(func.__name__))
    VIEW_FUNCTIONS[func.__name__] = func
    return func

@register
def show_user(username):
    return "hello {}!".format(username)

@register
def show_post(postid):
    return "post {}!".format(postid)
```

```
$ python3 test.py
registered show_user
registered show_post
```

Visualization https://goo.gl/42PHCJ

Server example

```
import sys
def run():
    while (True):
        # Fake request
        print("Function name:")
        fxn_name = sys.stdin.readline().strip()
        print("Function input:")
        fxn_input = sys.stdin.readline().strip()

        # Execute view function
        output = VIEW_FUNCTIONS[fxn_name](fxn_input)

        # Fake response
        print(output)

run()
```

```
$ python3 test.py
registered show_user
registered show_post
Function name:
show_user
Function input:
awdeorio
hello awdeorio!
```

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/