CSE 503

Introduction to Computer Science for Non-Majors

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Day 23
Web Servers (Part 2)

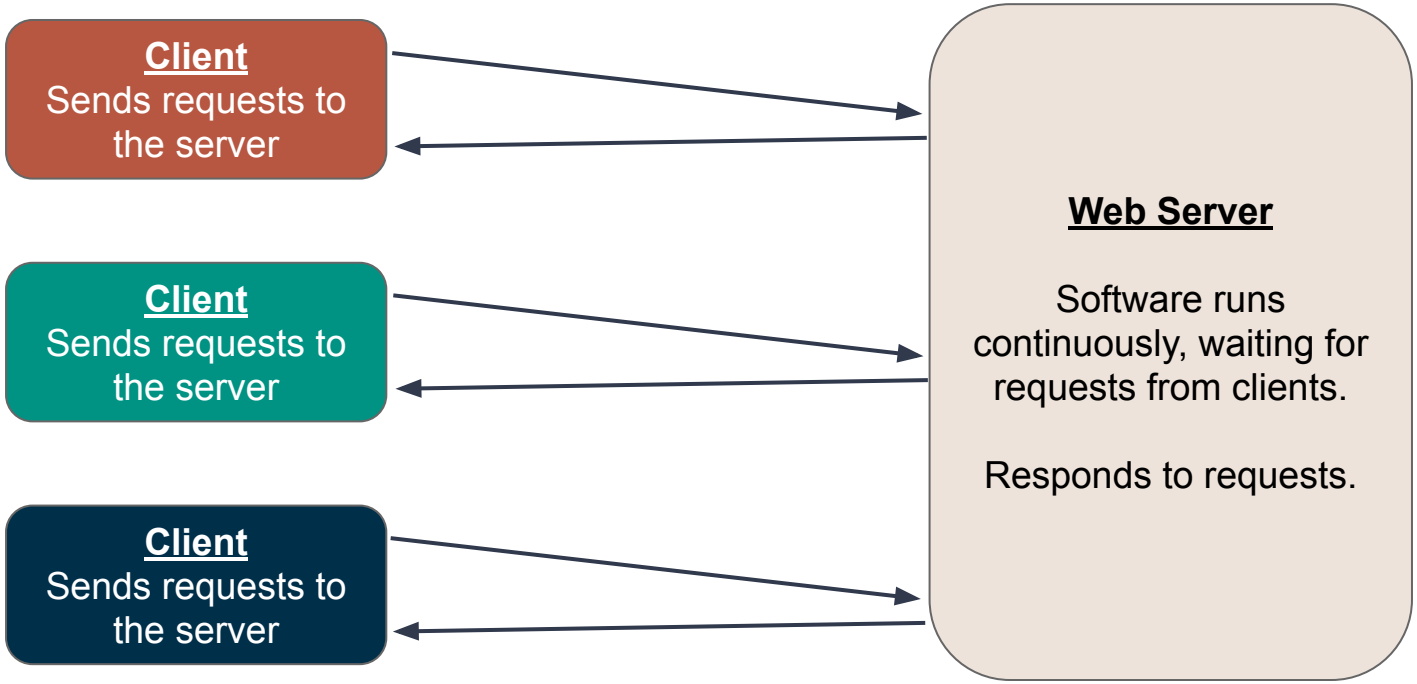
Announcements

- Lab #3 due tonight @ Midnight
  - Make sure you've submitted JS **and** PY versions
- Lab #4 will be released by tonight
  - Lab #4 and 5 are a two part sequence, 5 will build on 4
  - Lab #4 deals with reading and writing data with CSV files

Recap

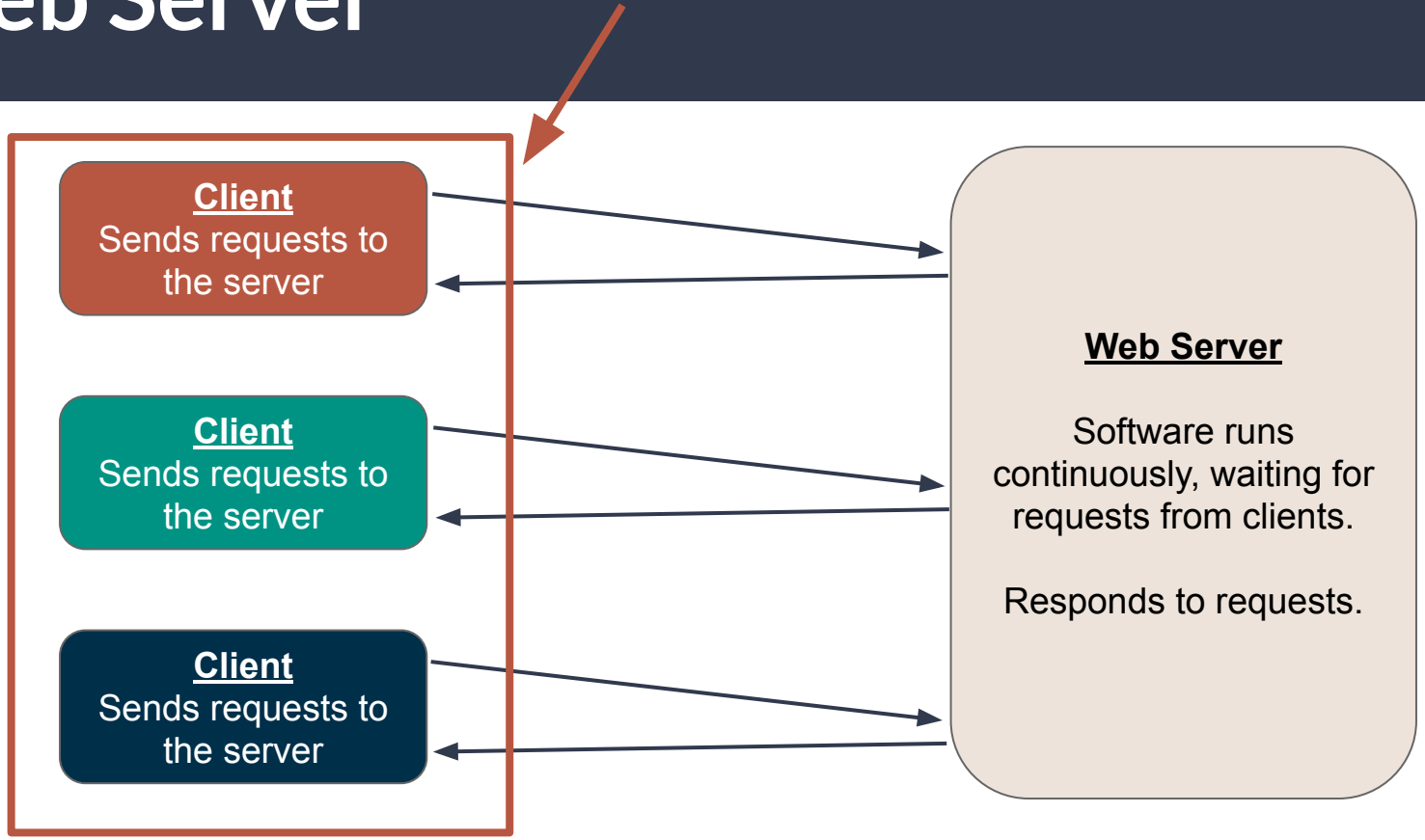
- Last time we used **`urllib`** to make HTTP requests in Python
  - Sometimes we requested HTML
  - Sometimes we made requests to a Web API that returns JSON
- **J**ava**S**cript **O**bject **N**otation **(JSON)** is a way to turn JavaScript objects into strings that can be communicated across the internet
  - Any other programming language can deal with strings
  - This will allow our JS front-end to communicate with Python back-end

Web Server


Client
Sends requests to the server
Client
Sends requests to the server
Client
Sends requests to the server
Web Server
Software runs continuously, waiting for requests from clients.
Responds to requests.

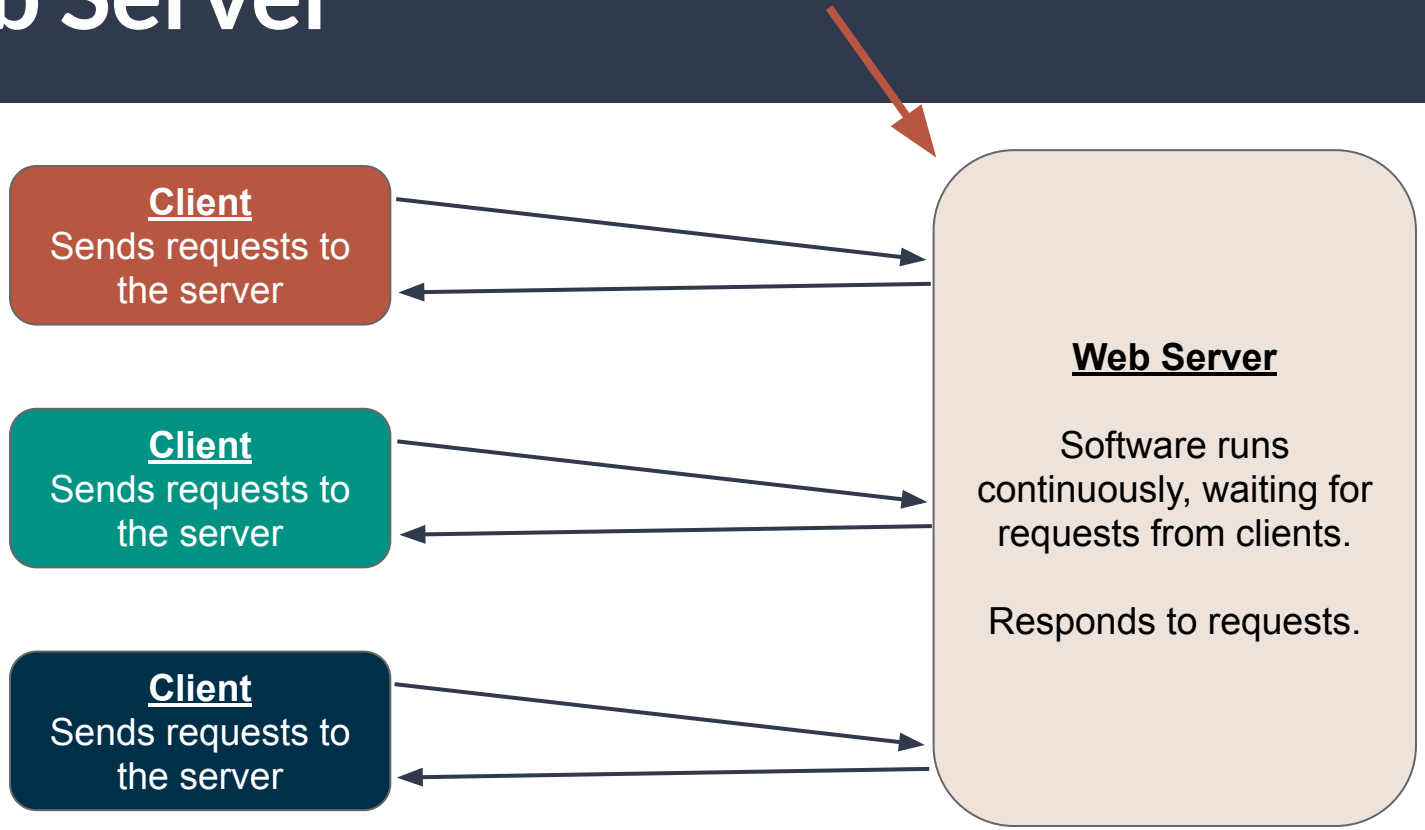
Web Server

Last time we wrote Python code that could make requests



Web Server

Today we will focus on how to write a web server in Python



Web Server

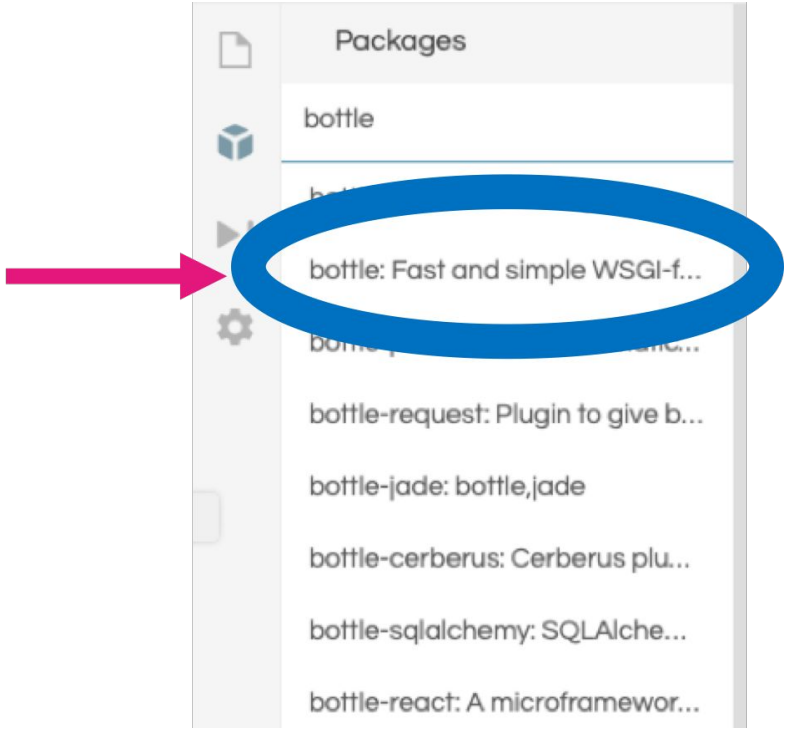
- Don't run all the code in the users browser
  - Some code runs on the Web Server instead
- Users make HTTP requests to get content from the server
- Users don't have access to the code/data on the server (hopefully)

Web Server - Bottle

- To create a Web Server, you will commonly use an existing web framework library
- In this course we will use **`bottle`**: https://bottlepy.org/docs/dev/
  - Other options include Django and Flask
- Bottle does not come with Python
  - We must install it in our REPLs on repl.it in order to use it

Installing Bottle

- Create a Python REPL on repl.it as normal
- On the left sidebar, click the Packager tab (third from the top, looks like a box)
- Search for bottle and click the + icon to install it



Our First Web Server

```
import bottle

@bottle.route("/")
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Our First Web Server

```
import bottle

@bottle.route("/")
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Import the bottle library

Our First Web Server

```
import bottle

@bottle.route("/")
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

This is called an annotation, it adds meta-information to a function…more on this later

Our First Web Server

```
import bottle

@bottle.route("/")
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

We've defined a function that returns an html string. Because of the annotation, bottle will call this function to respond to certain requests…

Our First Web Server

```
import bottle

@bottle.route("/")
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

bottle.run() starts our server

Our First Web Server

Before we go over the details, lets try running our server on repl.it

Once I start the server, you should be able to connect to it by going to:

`lec23.epmikida.repl.co`

Bottle Annotations

Annotations in Python start with `@`, and can be used to annotate functions with meta-information.

In this case, it allows us to tell bottle which functions to call to handle certain HTTP requests.

Bottle Annotations

Annotations in Python start with `@`, and can be used to annotate functions with meta-information.

In this case, it allows us to tell bottle which functions to call to handle certain HTTP requests.

`@bottle.route("/")` tells bottle that the annotated function should be called to respond to requests to the servers root (`"/"`)

Bottle Annotations

Annotations in Python start with `@`, and can be used to annotate functions with meta-information.

In this case, it allows us to tell bottle which functions to call to handle certain HTTP requests.

`@bottle.route("/")` tells bottle that the annotated function should be called to respond to requests to the servers root (`"/"`)

We can use different strings to respond to requests for different paths, ie `@bottle.route("/foo")` would handle requests to path `"/foo"`

Bottle Annotation

The function we annotate can have any name we want, and execute any code we want.

Whatever the function returns, will be the response sent to the client.

```
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response
```

Bottle Annotation

The function we annotate can have any name we want, and execute any code we want.

Whatever the function returns, will be the response sent to the client.

```
def any_name():
    response = "<html><body><p>"
    response = response + "Hello from the server!"
    response = response + "</p></body></html>"
    return response
```

In this case we return an HTML string for the requesting browser to display

Our First Web Server

- Our server is a program that runs continuously waiting for HTTP requests (in this case on localhost address 0.0.0.0, port 8080)
- If you change the code, the server must be restarted for the changes to take effect
- If the server stops running you will no longer have access to the server

Serving Static HTML Files

Creating HTML strings in Python for every request would be very tedious

Serving Static HTML Files

Creating HTML strings in Python for every request would be very tedious

The **`bottle.static_file()`** function can be used to return an HTML file:

```
@bottle.route("/")
def any_name():
  return bottle.static_file("index.html",root="")
```

Serving Static HTML Files

Creating HTML strings in Python for every request would be very tedious

The **`bottle.static_file()`** function can be used to return an HTML file:

```
@bottle.route("/")
def any_name():
  return bottle.static_file("index.html",root="")
```

Root specifies where the file is located with respect to the server, empty string means it is in the same directory

Using Templates to Modify the HTML

We can also use HTML files as a template, and have our Python code fill in some of the details.

Using Templates to Modify the HTML

We can also use HTML files as a template, and have our Python code fill in some of the details.

In the HTML file you want to serve, use **`{{key_name}}`** to specify where bottle can fill in values:

```
<p>Hello {{name}}!</p>
```

Using Templates to Modify the HTML

We can also use HTML files as a template, and have our Python code fill in some of the details.

In the HTML file you want to serve, use **`{{key_name}}`** to specify where bottle can fill in values:

```
<p>Hello {{name}}!</p>
```

Our Python code will be able to replace {{name}} with a new value. Our HTML file can have as many template placeholders as we want.

Using Templates to Modify the HTML

In our Python code, we can call the **`bottle.template()`** function to serve up a specific HTML file, and specify a dictionary that determines how the template placeholders are filled in:

```
@bottle.route("/")
def any_name():
    return bottle.template("hello.html", {"name": "World"})
```

Using Templates to Modify the HTML

In our Python code, we can call the **`bottle.template()`** function to serve up a specific HTML file, and specify a dictionary that determines how the template placeholders are filled in:

```
@bottle.route("/")
def any_name():
    return bottle.template("hello.html", {"name": "World"})
```

Return the hello.html file, but replace {{name}} with World

Utilizing Query Strings

Last lecture, we saw how query strings could be used to pass extra information in an HTTP request…so how can we get that information?

Utilizing Query Strings

Last lecture, we saw how query strings could be used to pass extra information in an HTTP request…so how can we get that information?

`bottle.request.query` is a dictionary containing the key value pairs passed in the query string:

```
@bottle.route("/hello")
def any_name():
  replacements = {}
  replacements["name"] = bottle.request.query.get("name","World")
  return bottle.template("hello.html",replacements)
```

Utilizing Query Strings

Last lecture, we saw how query strings could be used to pass extra information in an HTTP request…so how can we get that information?

`bottle.request.query` is a dictionary containing the key value pairs passed in the query string:

```
@bottle.route("/hello")
def any_name():
  replacements = {}
  replacements["name"] = bottle.request.query.get("name","World")
  return bottle.template("hello.html",replacements)
```

We can use bottle.request.query just like any other dictionary we've used

Utilizing Query Strings

Last lecture, we saw how query strings could be used to pass extra information in an HTTP request…so how can we get that information?

`bottle.request.query` is a dictionary containing the key value pairs passed in the query string:

```
@bottle.route("/hello")
def any_name():
  replacements = {}
  replacements["name"] = bottle.request.query.get("name","World")
  return bottle.template("hello.html",replacements)
```

We can use bottle.request.query just like any other dictionary we've used

We fill in template placeholders with our replacements dictionary, which we've filled in based on the query string