Back End Development

Announcement

- I won't be hear next Tuesday
- Meet with your group and get started
- Tuesday's activity: Submit your project idea

Activity

Write 2 web apps

1. One written in Node.js/Express

2. One written in Python/Flask (If you prefer, you can use any other Python web framework)

Requirements for each app

-Serve an HTML document at the root path

-Serve at least 2 images that are rendered in your HTML

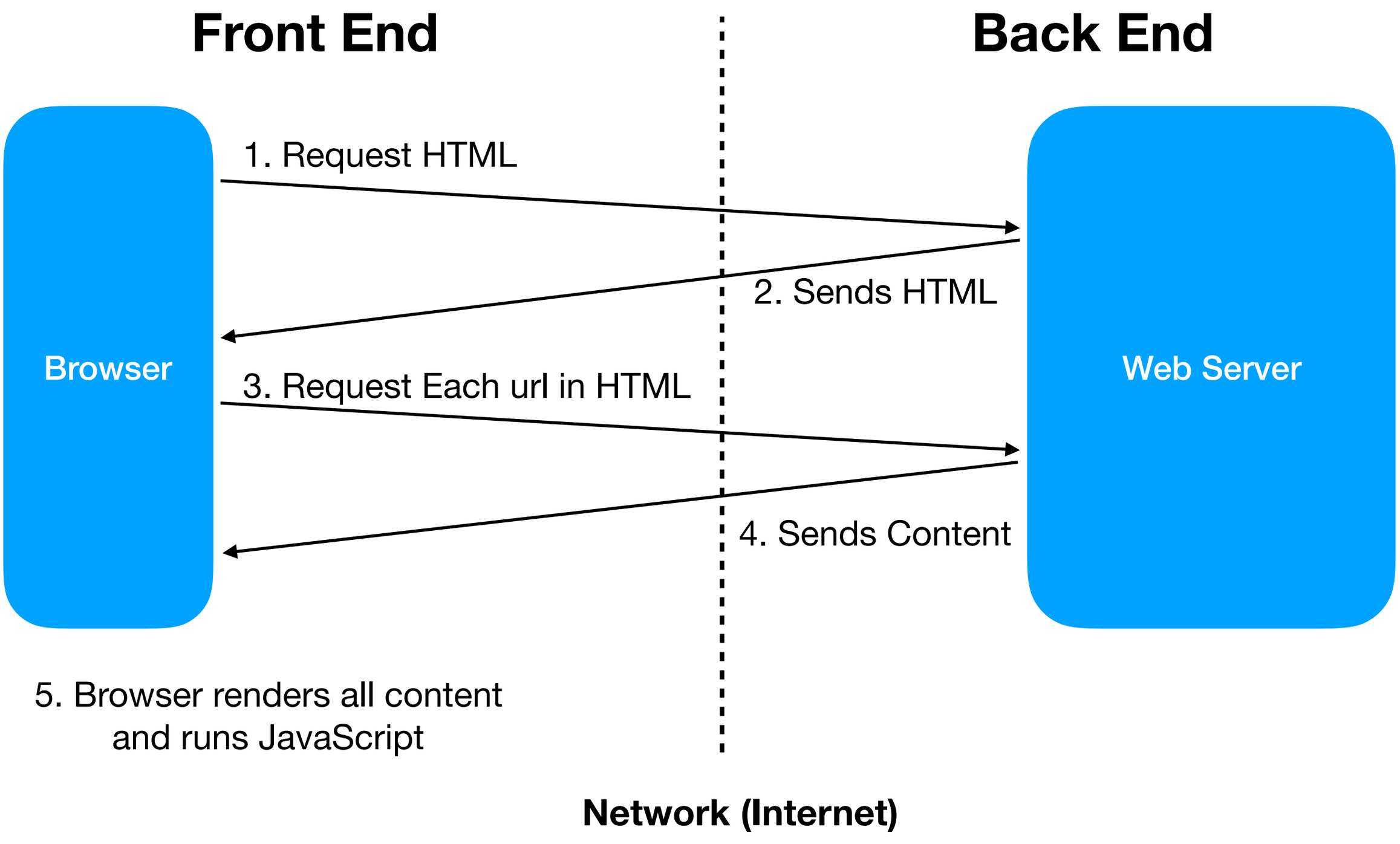
-Include a form where the user can enter a value and have the page reload using that value (Ex. "Enter your name" form and the page says hello <name> -or- A calculator where the user enters a dollar amount and the page shows that amount with an 8% sales tax added)

-Return a 404 if the client requests a path that doesn't have a route

Web Server

- So far we've been opening web pages by loading *.html files in the browser
    - This uses the 'file' protocol which loads a file from disk
- We eventually want to host our web pages/apps on the Internet
- For this we'll need a web server
    - Hardware/Software that listens for HTTP requests

Loading a Web Site

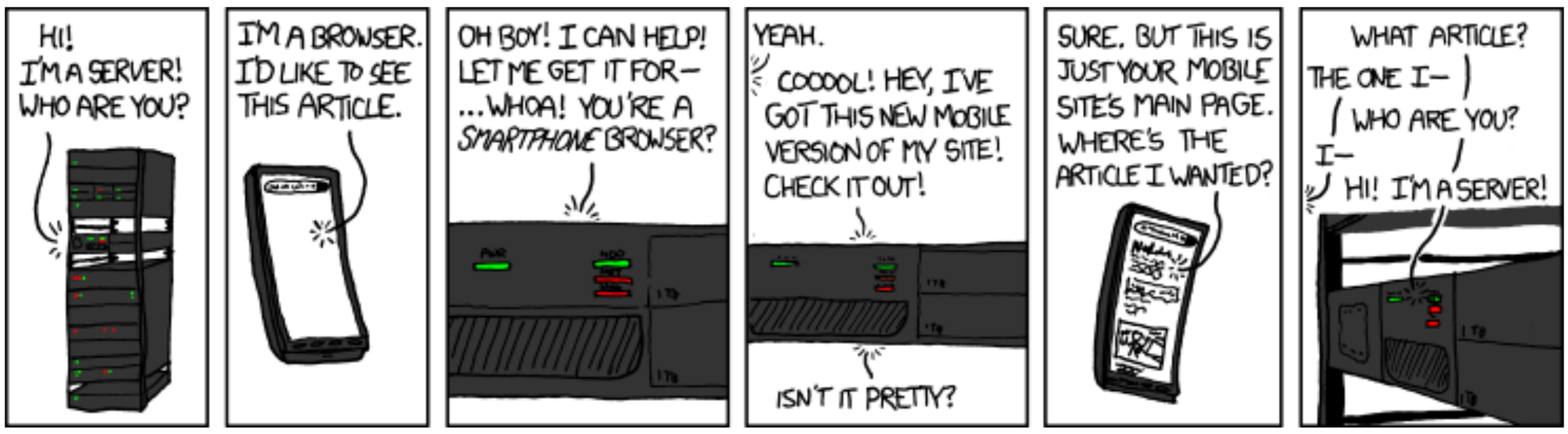


HTTP

- HTTP is a protocol used by the browser to load content from a web server
- Protocol: An agreed upon set of rules
  - HTTP: Defines the format of messages sent to/from a web server
- Request - Response protocol
  - Client makes request to server
  - Server returns a response
  - Ex. Request The latest tweets from a user. Twitter server returns the tweets in its response
- Response can require more requests

HTTP

- HTTP is a stateless protocol
  - Each request is handled in isolation even if a client just made another request
- If state is desired (ex. Login), the state must be sent with each request
  - Cookies
  - Tokens
- When handling an HTTP request, do not have to care who sent it



https://xkcd.com/869/

HTTP - Request

Protocol://host:port/path?query_string#fragment

- Each request is made to a specific URL (Uniform Resource Location)
  - A URL uniquely identifies a web resource and has the following parts
- **Protocol** - The protocol being used (ex. file, HTTP, HTTPS)
- **Host** - The IP address or domain name of the server
  - Used to route the request to appropriate machine
- **Port** - The TCP port number of the host server
  - Defaults to 80/443 for HTTP/HTTPS respectively
- **Path** - Specifies the specific resource being requested from the server

HTTP - Request

Protocol://host:port/path?query_string#fragment

- **Query String** - [Optional] Contains key-value pairs set by the client
- https://www.google.com/search?q=web+development
  - HTTPS request to Google search for the phrase "web development"
- https://duckduckgo.com/?q=web+development&ia=images
  - An HTTPS request to Duck Duck Go image search for the phrase "web development"

- **Fragment** - [Optional] Specifies a single value commonly used for navigation
- https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
  - HTTPS Request for the URI Wikipedia page
- https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Definition
  - HTTPS Request for the URI Wikipedia page that will scroll to the definition of URI

HTTP - Request

- Each HTTP request will contain the request type:
  - GET: Request information from a server
  - POST: Send information to a server
  - PUT (less common): Add information to a service
  - DELETE (less common): Delete information form a service
  - etc.
- We'll focus on GET/POST requests

HTTP - Request

- HTTP GET Request
  - Used when requesting content from a server
  - Contains only a URL [And HTTP headers]
  - All examples on the query string/fragment slide were GET requests
    - Whenever you click a link your browser makes a get request
    - Requesting HTML/CSS/Javascript/Images/etc are GET requests
- HTTP POST Request
  - Used when sending data to a website
  - Contains a URL and a body [And HTTP headers]
  - When you submit a form your browser [typically] makes a POST request
  - The contents of the form are sent in the body of the request

HTTP - Response

- When the server receives an HTTP request
  - Process the request and send a response to the client
- Whether the request was GET or POST, the response will have a body containing the requested content
- Response also contains HTTP headers including meta information about the response and the server
  - Headers often include caching information
- Response includes a response code indicating the status of the response

HTTP - Response

Common Response codes include:

200 OK

Request was handled as expected

301 Moved Permanently

Redirect to the new location

304 Not Modified

File in local cache can be used

403 Forbidden

You don’t have access to the requested page

404 Not Found

Requested data could not be found

500

Internal server error

Web Frameworks

Web Framework

- There are many common tasks that every web developer must accomplish on a regular basis

- Web frameworks are libraries that handle these common tasks

    - Allows web developers to focus on developing their apps

- Typically contain a simple web server for testing and development

    - These servers are limited (More Thursday)

Web Framework

- Today we'll cover these features of web frameworks

1. **Routing Paths**
2. **Serving Static Files**
3. **Parsing query strings**
4. **Handling POST requests / Forms**
5. **HTML Templates**

Routing Paths

Protocol://host:port/path?query_string#fragment

- When a client sends an HTTP request to you app each part of the URL should be handled

- Protocol, host, and port are used by the Internet and your web server to route the request to your app

- The first part your app needs to handle is the path (The specific resource being requested)

Routing Paths

http://localhost:8000/

- *Assuming your running your server on your laptop using port 8000
- This is the root path of your app "/"
- When a request for this path is received you should return the home page of your app

http://localhost:8000/blog

- This URL is request for the path "/blog"
- Your app should return a different page when it receives a request for this URL

Routing Paths

http://localhost:8000/

http://localhost:8000/blog

- Web frameworks will provide a way to handle these paths differently

- Typically a framework will let you specify a path as a string, then provide a function that will be called to serve that path

- Specify whether the route is for get or post requests

```
app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

https://expressjs.com/en/starter/basic-routing.html

Programming Side Note

- This code uses an anonymous function as an argument of a method call
- This has the same functionality if we define and name the function earlier
- Note that we are passing the entire function
    - Do not use ( ) since this will call the function
- The app will call this function each time a request is made for the root path

```
app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

```
function serverRoot(req, res) {
   res.send('Hello World!')
}

app.get('/', serverRoot);
```

https://expressjs.com/en/starter/basic-routing.html

Routing Paths

http://localhost:8000/

http://localhost:8000/blog

- By changing the string for the path we can define different behavior for each path we want to implement

```
app.get('/blog', function (req, res) {
  res.send('Welcome to my blog!')
})
```

https://expressjs.com/en/starter/basic-routing.html

Routing Paths

http://localhost:8000/blog/post1

http://localhost:8000/blog/post2

- We often don't want to hardcode every single path
- Frameworks give us a way to add variables in our paths
- Can use regular expressions to define more general paths

```
app.get('/blog/:post', function (req, res) {
  res.send('Welcome to my post titled: ' + req.params.post)
})
```

Static Files

http://localhost:8000/static/script.js

http://localhost:8000/static/mewtwo.png

- Instead of returning hardcoded content, we often want to serve entire files of content

- If the files are always sent as-is, we call them static files

- Most frameworks allow you to add all static files into a single directory (typically named "static/" or "public/") and tell the framework to allow clients to access any of those files by name

    - A specific path is used for all these files (ex. "static/:filename")

    - Save you the trouble of working with file io, converting the file to a byte stream, creating and sending the HTTP response

Static Files

http://localhost:8000/static/script.js

http://localhost:8000/static/mewtwo.png

- Be very careful if not using the the built-in way of serving static files for your framework
- The framework will prevent clients from accessing arbitrary files on your server
- Ex. If you simply take the provided filename and send it to the client
    - Client requests the path "/static/../../all_your_secrets.txt"

Parsing Query Strings

http://localhost:8000/search/q=content&key=123456

- If a request contains a query string, if must be parsed to read the key-value pairs
- Query strings follow a strict format which makes parsing the string possible
- Web Frameworks will parse these strings for you are store the key-values in the request
- For each framework, find their syntax for access the values by key
- *Same for fragments

Handling POST Requests

- Suppose we have the following form in our HTML
- When the user submits this form an HTTP POST request will be sent to the server with the path "/form"
- All form inputs will be in the body of the POST request as key-value pairs
    - The "name" attribute of each form input will be the key and whatever the user enters will be the value
    - In this example, the body will contain a key "user_name" with a value of whatever the user entered into the text field

```
<form action="/form" method="POST">
    Enter Your Name:
    <input type="text" name="user_name">
    <br/><br/>
    <input type="submit" value="Submit">
</form>
```

Handling POST Requests

- A web framework will provide a way to read these key-value pairs submitted from a form

- Access the valuable containing the body of the request

- Access the value at each key

- Many frameworks will parse the form responses for you and enter them into a data structure

- Return a response just like we did with GET requests

```
<form action="/form" method="POST">
    Enter Your Name:
    <input type="text" name="user_name">
    <br/><br/>
    <input type="submit" value="Submit">
</form>
```

HTML Templates

- HTML Templates add a significant amount of flexibility to our apps
- So far we've handled mostly static content and served the content requested by the client
    - We also read user inputs from a form, but how do we send a user a custom page made just for them?
- HTML templates allow us to add variables and control flow into our HTML
- The template defines the structure of the HTML
    - For each request, we fill in the content of the template

HTML Templates

- In this example we have an HTML file written using a template language (Handlebars)

- The template language adds more functionality to HTML
  Using { } and certain keywords

```
<div class="messages">
    {% for message in messages %}
    <div class="alert alert-info alert-dismissable">
        <a href="#" class="close" data-dismiss="alert" aria-label="close">&times;</a>
        {{message}}
    </div>
    {% end %}
</div>
```

```
self.render('view_templates/messages.html', messages=["Password not set", "Passwords don't match"])
```

HTML Templates

- We use a for loop to iterate over a list of messages and display them all on the page using handle bars syntax
    - {% for <var_name> in <data_structure> %}
    - {{ <var_name> }} to insert the value of a variable into our HTML
    - {% end %} to end the current control structure

```
<div class="messages">
    {% for message in messages %}
    <div class="alert alert-info alert-dismissable">
        <a href="#" class="close" data-dismiss="alert" aria-label="close">&times;</a>
        {{message}}
    </div>
    {% end %}
</div>
```

```
self.render('view_templates/messages.html', messages=["Password not set", "Passwords don't match"])
```

HTML Templates

- When we want to use the template, we use the function in our framework that renders a template
    - Provide all variables needed for the template in the call to render
    - Will return a 500 error if a variable is missing
    - Can think of rendering a template as calling a function that returns HTML

```
<div class="messages">
    {% for message in messages %}
    <div class="alert alert-info alert-dismissable">
        <a href="#" class="close" data-dismiss="alert" aria-label="close">&times;</a>
        {{message}}
    </div>
    {% end %}
</div>
```

```
self.render('view_templates/messages.html', messages=["Password not set", "Passwords don't match"])
```

HTML Templates

- Each framework will chose a default template language
    - Flask defaults to Jinja
    - Express encourages Pug in it's documentation
- Tons of choices
- Find one that works for you, or just stick to the defaults if you don't want to think about this yet

Running Your App

Running

- Most frameworks, including the ones we'll see in class, include a web server
- When you run this server on your laptop, it will run forever and wait for HTTP requests
- Each time it receives an HTTP request it will respond according to your code
- Run your server, then open a browser and access your app
    - URL will be something like "https://localhost:3000/"
    - Each framework has a different default port that you can change
    - Common default ports: 3000, 5000, 8000, 8080

Activity

Write 2 web apps

1. One written in Node.js/Express

2. One written in Python/Flask (If you prefer, you can use any other Python web framework)

Requirements for each app

-Serve an HTML document at the root path

-Serve at least 2 images that are rendered in your HTML

-Include a form where the user can enter a value and have the page reload using that value (Ex. "Enter your name" form and the page says hello <name> -or- A calculator where the user enters a dollar amount and the page shows that amount with an 8% sales tax added)

-Return a 404 if the client requests a path that doesn't have a route