Web Frameworks

Web Frameworks

- Banned for homework assignments

- Now that you're starting your project where you can use these
    - Let's talk about it

Web Frameworks

- 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

Web Frameworks

- Today we'll talk about these features that you've developed in your assignments

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/

```
GET / HTTP/1.1
```

http://localhost:8000/blog

```
GET /blog HTTP/1.1
```

- A server must decide how to handle requests depending on the type and path
  - We call this routing

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
- Use method calls and passing functions avoids the giant if statement

```
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"

```
app.use(express.static('public'));
```

Parsing Query Strings

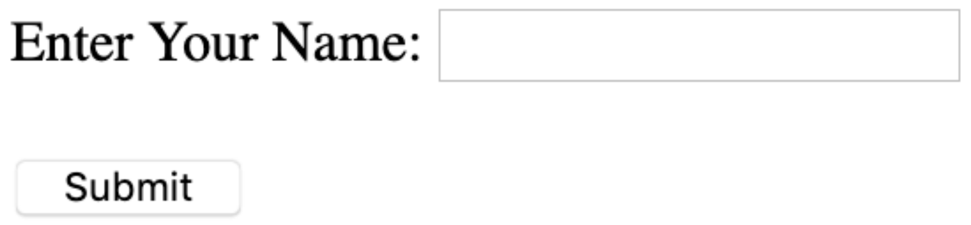
http://localhost:8000/search/q=content&key=123456

- 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 accessing 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

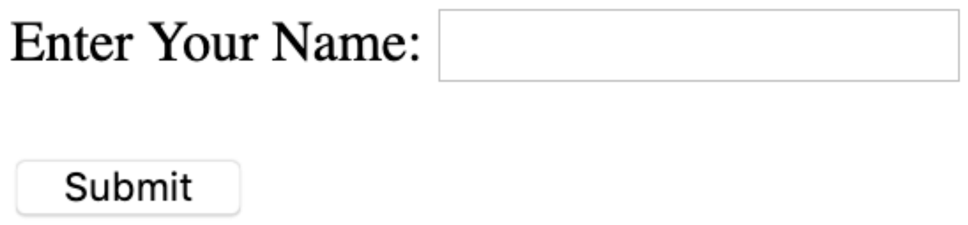
```html
<form action="/form" method="POST">
    Enter Your Name:
    <input type="text" name="user_name">
    <br/><br/>
    <input type="submit" value="Submit">
</form>
```



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

```html
<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