Code Fellows

Class 10
Authentication

seattle-javascript-401n14

Lab 09 Review

Code Challenge 09 Review

Local Elections are NEXT TUESDAY!!!

Not a meme

Please Vote

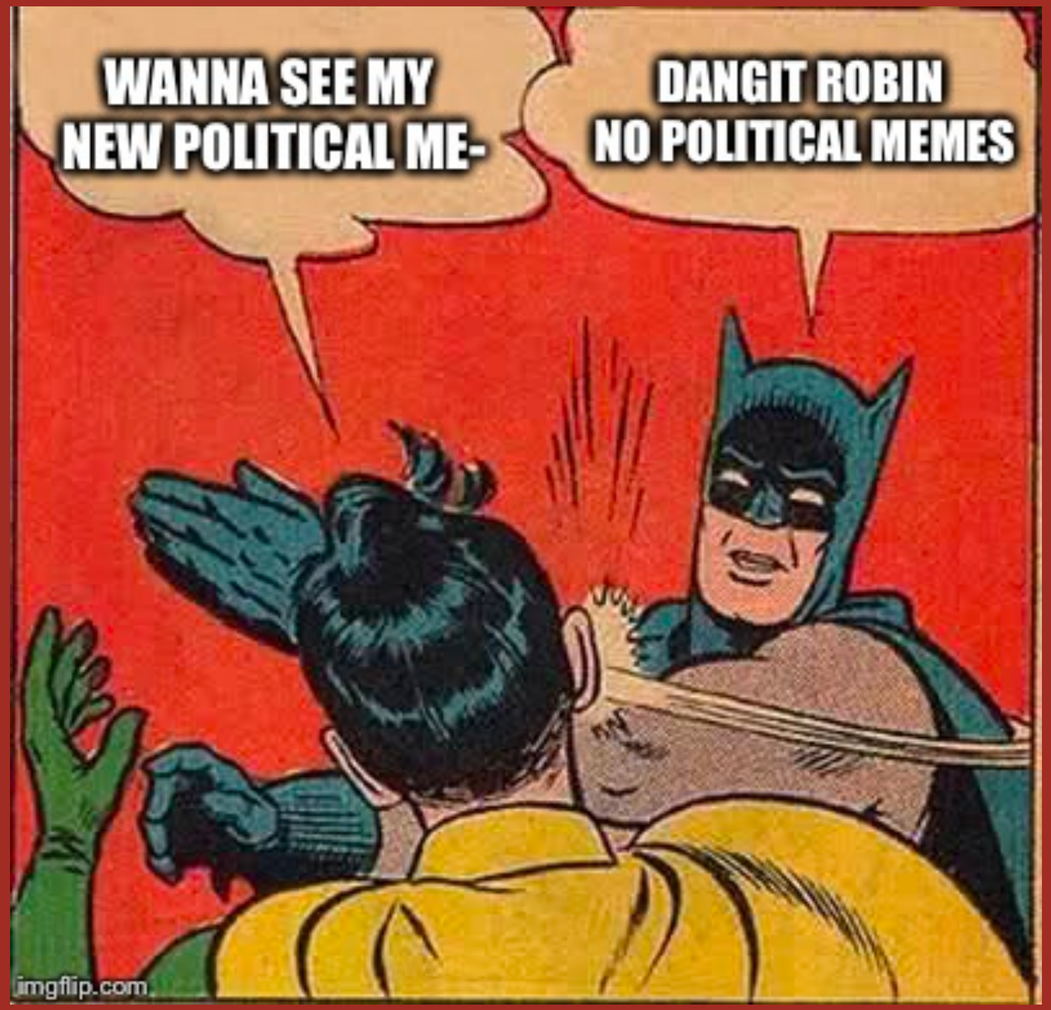
Elections on Tuesday Nov 5th

- Please register and vote if you are a Seattle + Washington Resident
- You have **WAY MORE INFLUENCE** in a local election than a national election
- Issues on the line:
  - Affordable Housing
  - Rent Control
  - Public Transit Support
  - Affirmative Action
  - Corporate Money / Interference in Politics

Elections on Tuesday Nov 5th

- On election day, Tuesday November 5th, **we will be starting class late**
    - **Class will start at 7:30pm**
- This is so that everyone feels they have enough time to fill out their ballot and mail it or drop it off at one of the many ballot drop-off locations.
- Those of you who don't need that extra time are welcome to come to Code Fellows; I will be there to help with any questions on labs / homework

Okay, back to lecture



Authentication

- The title of our class!
- **Authentication** is the process of determining whether something is what it declares itself to be
- Examples where we might need authentication:
    - User logs in to their private account
    - A client requests some private data
    - Transactions, data transfer, etc.

Authentication Example

- **Client**: My name is Bill, my username is `bill` and my password is `bill`
- **Server**: Let me authenticate that you are Bill…
- **Server**: I don’t store your password as a string, instead I encrypt it
- **Server**: Let me encrypt the password you gave me, `bill`, using the same method I used with my stored passwords
- **Server**: Let me search for a user that matches `{username: bill, password: encrypt(bill)}`
- **Server**: Hmm… I can’t find any users that match that
- **Server**: 401 Error - Login Failed
- **Client**: Darn….

Authorization

- Sounds very similar to Authentication, but it's not!
- **Authentication** tells us if you are who you say you are
- Once we know that, **Authorization** tells us what you are allowed to access
- There are various kinds of authorizations:
    - Read Access
    - Write Access

Authorization Example

- **Client**: My name is Bill, give me my wife Mary's data
- **Server**: Let me authenticate that you are Bill…
- **Server**: Good news! Based on your username and password, you seem to be Bill!
- **Server**: Now let me check if your authorized to access Mary's data
- **Server**: Hmm… it looks like Mary's not your wife so you don't have access!
- **Server**: 403 Error - Forbidden
- **Client**: Darn….

Storing Passwords

- We store passwords securely by using either **hashing** or **encryption**
    - **Hashing** is **one-way**: we have an algorithm that can get from `password string -> hash`, but we can't go back
    - **Encryption** is **two-way**: we have a key that can take us from `password string -> encryption -> password string`
- Hashing is generally safer because if a hacker figures out our hash algorithm, they still can't interpret stored hashed values. But if a hacker figures out our encryption key….

How to Hash

- When a user signs up, they will give us a password as a string. We'll hash it and save the hash in our database
- We are going to use bcrypt to hash our secure data
  - `npm install bcrypt`
  - In our server code:

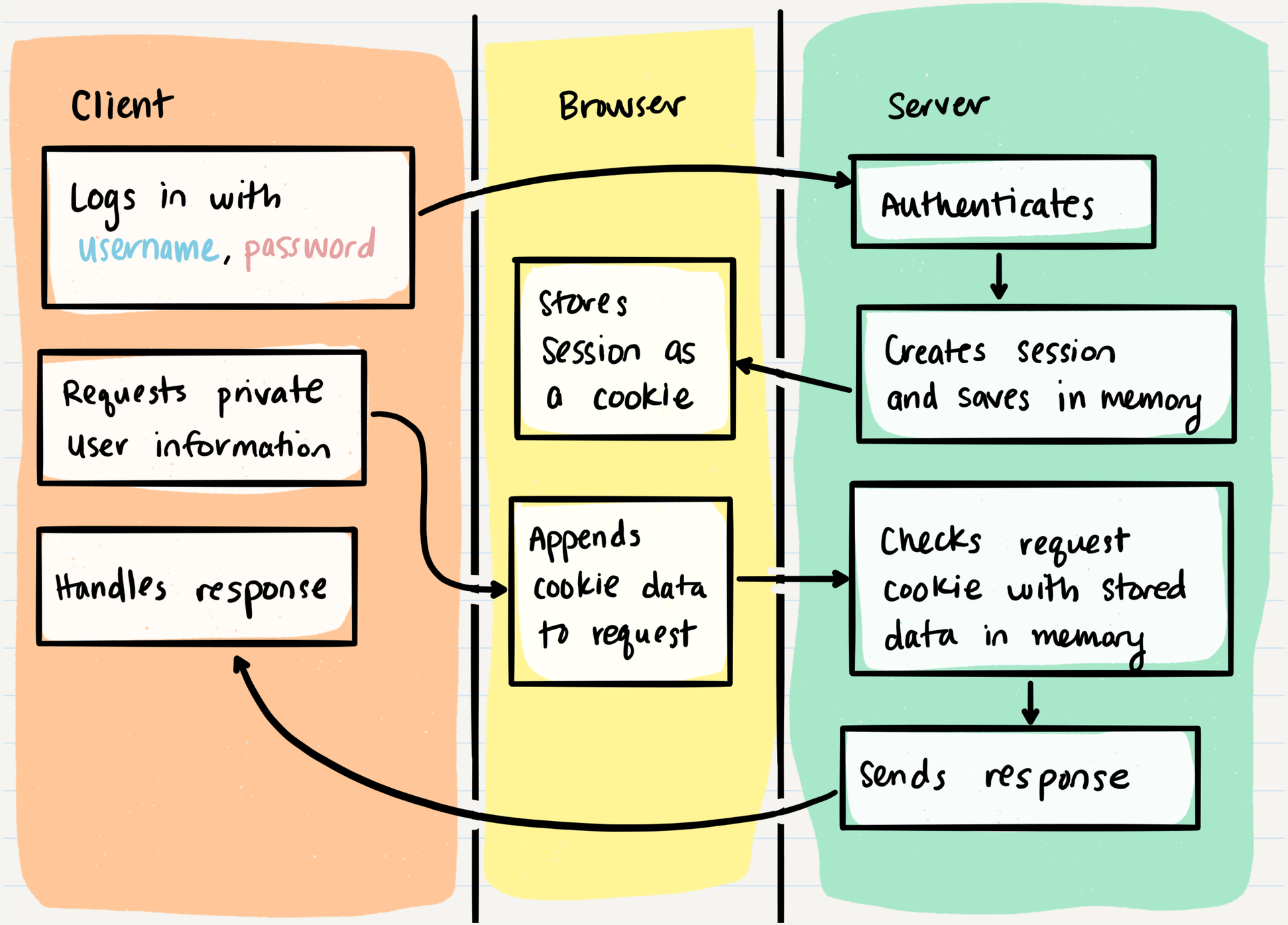
    ```
    let hashed = bcrypt.hash(password, 10);
    ```

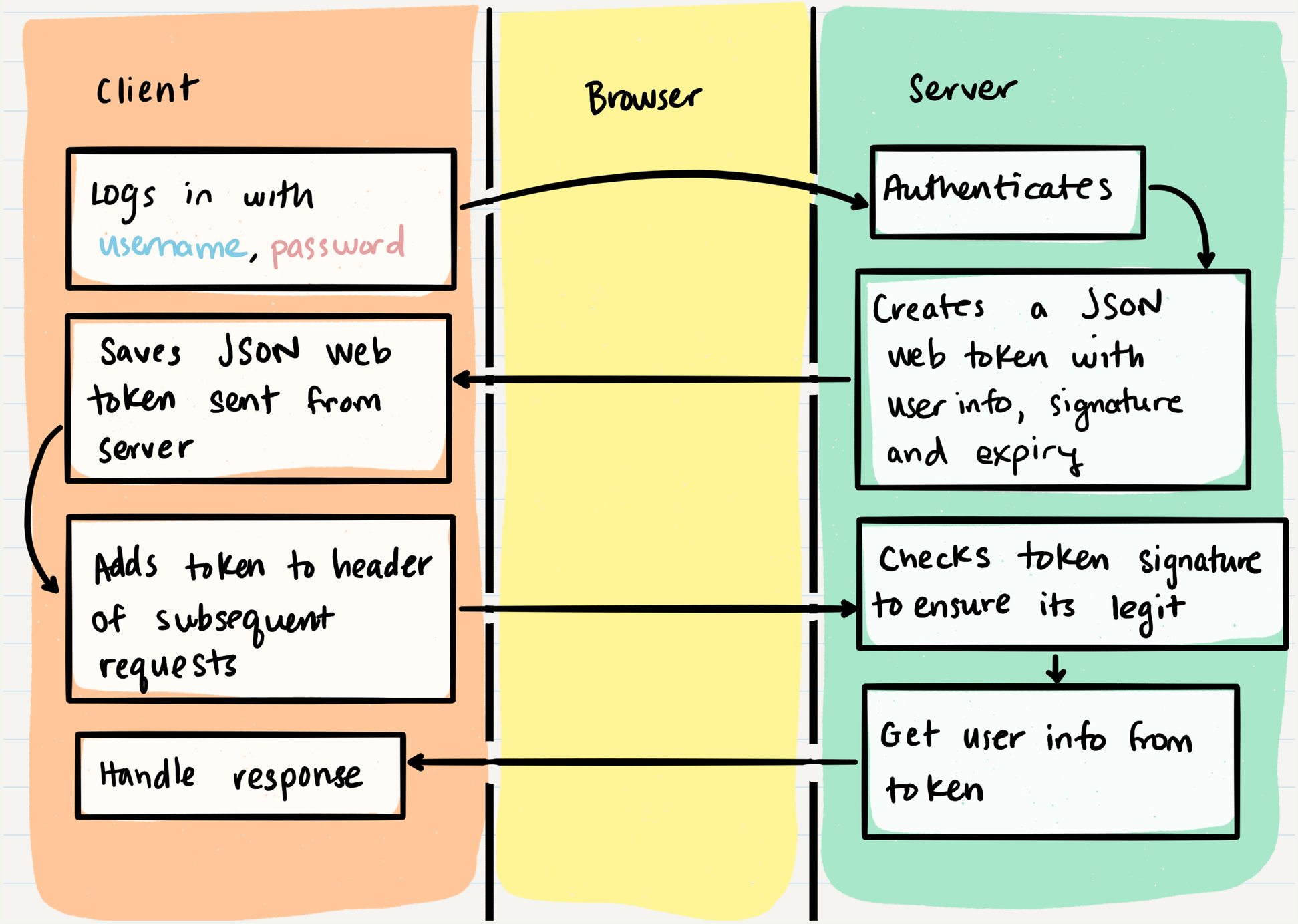
  - To check if a string password matches a hashed one:

    ```
    let isValid = bcrypt.compare(password, hashed);
    ```

Sessions and Tokens

- There's two major ways that we can handle an Authenticated and/or Authorized user:
  - **Sessions** save user information both on the server and in a **cookie** loaded into the browser. The session cookie is automatically added to any client request from that user
    - Sessions are **stateful**: the server has to save something about the state of the client-server interaction
  - **Tokens** are created by the server and then sent in a response to the client. The client now has to save the token and add it to every request
    - Tokens are **stateless**: the server doesn't have to remember anything about the client


Client
Browser
Server
Logs in with username, password
Authenticates
Stores Session as a cookie
Creates session and saves in memory
Requests private user information
Appends cookie data to request
Checks request cookie with stored data in memory
Handles response
Sends response


Client
Browser
Server
Logs in with username, password
Authenticates
Creates a JSON web token with user info, signature and expiry
Saves JSON web token sent from server
Checks token signature to ensure its legit
Adds token to header of subsequent requests
Get user info from token
Handle response

JSON Web Token

- When we create tokens, they’re really just JSON objects
- We can generate them using the package jsonwebtoken
    - Npm install jsonwebtoken
- To create a token, we need the data we want to store in our token, plus a special string that is unique to our application. We call this a `secret`, and it can be any string!

```
const jwt = require('jsonwebtoken');

jwt.sign(record, secret);
```

Cross-Origin Resource Sharing

- Clients can make frequent requests to our server
- Our server needs to check that these are legit requests
- Origin is where a request is coming from. Same-origin client requests come from the same **protocol** (http), **host** (localhost), and **port** (3000) as the server
- But that's super restrictive! **Cross-Origin** requests can come from any client (Postman, HTTPie, etc)
    - Now that we have multiple kinds of client accessing our server, we might want to put restrictions on certain clients

Cross-Origin Resource Sharing

- The CORS package can help us define what kind of clients (origins) can access what routes, assets, etc on our server
- Remember our REST request structure? We had some **headers:**

```
fetch(‘/‘, {
  method: ‘GET’
  headers: {
    Content-Type: ‘application/json’
  }});
```

Cross-Origin Resource Sharing

- The CORS package will automatically add headers to our requests
- These headers are called Access Control Headers
- How to allow any origin to do anything:

```
app.use(cors());
```

- How to only allow a specific origin:

```
const config = { origin: “http://localhost:3000”}
app.use(cors(config));
```

What's Next:

- Due by Midnight Tonight:
  - **Code Challenge 09**
- Due by 6:30pm Tuesday
  - **Lab 09**
  - **Reading Class 10**
- Due by Midnight Wednesday
  - **Learning Journal Class 10**
- Due by Midnight Thursday
  - **Code Challenge 10**
- Due by 9am Saturday
  - **Lab 10**
  - **Reading Class 11**
- Next Class: **Class 11 - OAuth**

Questions?