# Your Next Week

## Tuesday April 14

*6:30 PM*
— **DUE Class 09 Mock Interviews**
— **DUE Class 09 Lab**
— **DUE Class 10 Reading**
— Class 10A

## Wednesday April 15

*6:30 PM*
— Class 10B

*MIDNIGHT*
— **DUE Class 10 Learning Journal**

## Thursday April 16

*6:30 PM*
— Co-working

## Friday April 17

## Saturday April 18

*9 AM*
— ~~**DUE Class 10 Code Challenge**~~
— **DUE Class 10 Lab**
— **DUE Class 11 Reading**
— Class 11
— Instructor Syncs

*MIDNIGHT*
— **DUE Class 11 Learning Journal**

## Sunday April 19

*MIDNIGHT*
— **DUE Career Coaching Personal Pitch**
— **DUE Career Coaching Resume**
— **DUE Career Coaching Stage Fright**
— **DUE Career Coaching Your Why**
— **DUE Class 10-11 Feedback**

## Monday April 20

*6:30 PM*
— Career Coaching Workshop #1 (Mandatory)

## Tuesday April 21

*6:30 PM*
— **DUE Class 11 Lab**
— **DUE Class 12 Reading**
— Class 12A

# Lab 09 Review

# Mock Interviews Review

# Class 10

# Authentication

seattle-javascript-401n16

# Authentication

- The process of determining if someone is who they say they are
- Important if you want to save user data
  - Especially if that data is sensitive
- Client asks for data access, server checks for **authentication**
- Client provides username/password or some other authentication string


- **Client**: My name is Bill, my username is `bUser` and my password is `bPass`
- **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 using the same method I used with my stored passwords
- **Server**: Let me search for a user that matches

  ```
  { username: bUser, password: encrypt(bPass) }
  ```
- **Server**: Hmm… I can’t find any users that match that
- **Server**: 401 Error - Unauthorized
- **Client**: Darn….

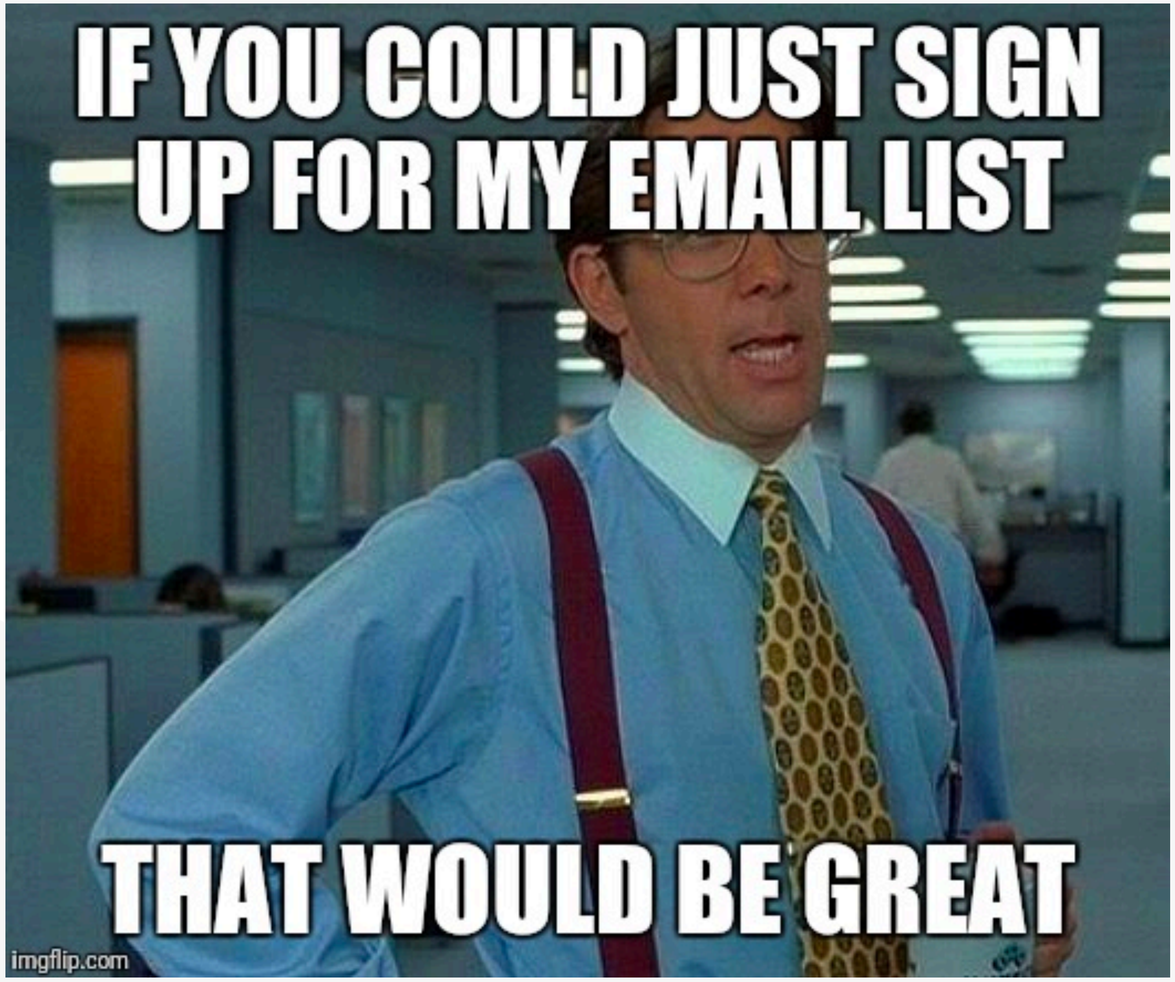
# Authorization

- **Authentication** = you are who you say you are
- **Authorization** = you are able to see what you ask to see
- Both are often tied together (called **Auth/Auth**)
- Authentication failed = 401 Error
- Authorization failed = 403 Error

- **Client**: My name is Bill, give me my wife Mary's data. My username is `bUser` and my password is `bPass`
- **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 you're 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 - Unauthorized
- **Client**: Darn….

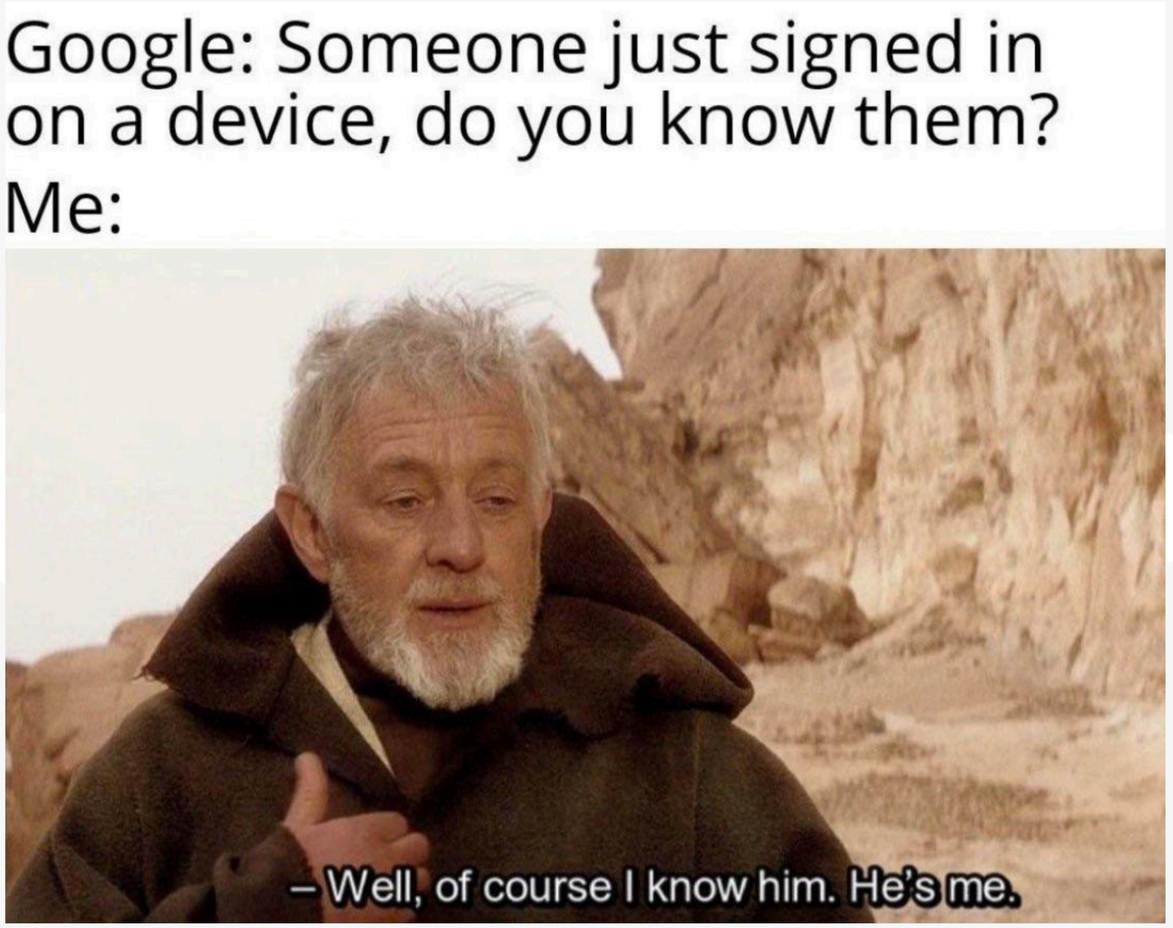
# Sign Up

- If you have an authenticated application, you need a way to create users
- Client should send a username and password for a user to be created
- Client either:
  - Sends username and password strings in `request.body`
  - Sends **base 64 encoded** string in `request.headers.authorization`



# Sign In

- If you have an authenticated application, you need a way to sign in users
- Client should send a username and password for a user to be validated
- Client sends **base 64 encoded** string in `request.headers.authorization`
- Very rare to see the client send username and password in the `request.body`



# Base 64 Encoding

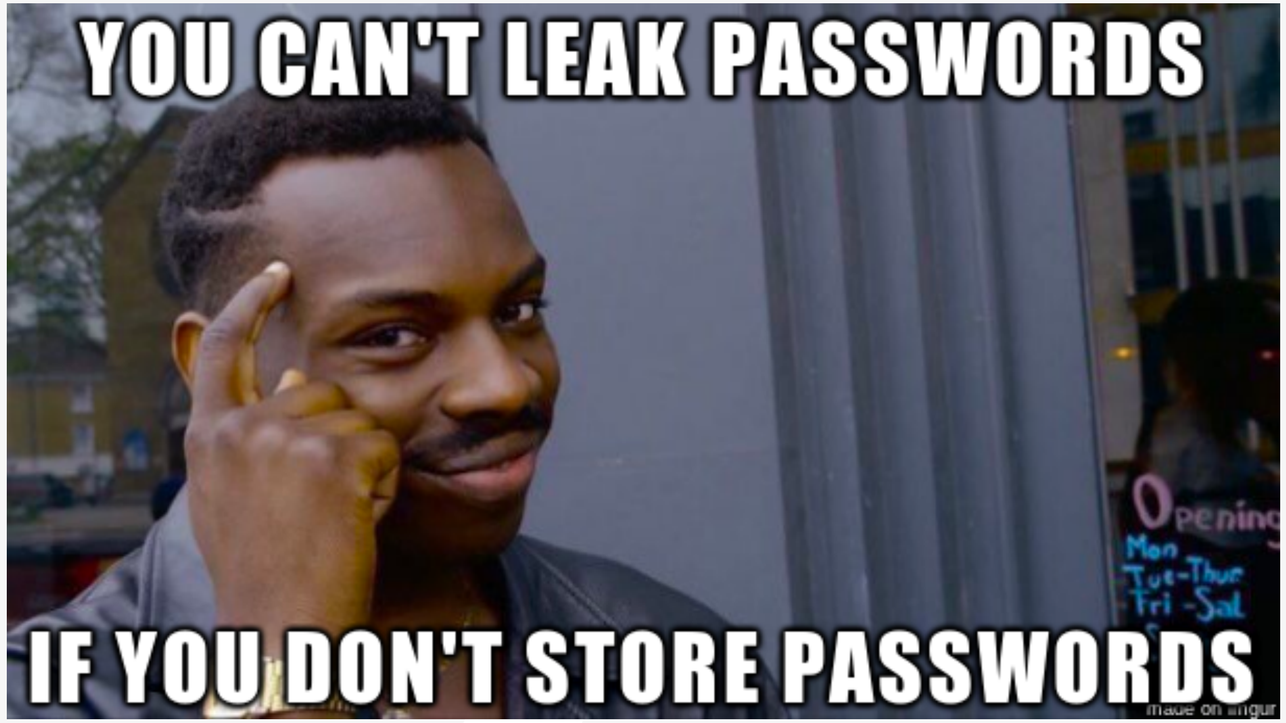
- Clients encode data sent to servers
- This protects against special characters in usernames / passwords
- **Encoding** is NOT encryption
- Encoding is just a simple language translation
  - Anyone who knows the “language” can reverse the translation
  - The data is not actually protected

# Basic Auth

- **Basic authentication** is the simplest way a client sends authentication credentials to a server
- A string of the format: `'Basic ' + base64encode(username + ':' + password)`
  - Example: `'Basic YlVzZXI6YlBhc3M='`
- Important to note that this string is **NOT SECURE**
- However hackers don't often intercept client -> server communication
  - They mostly intercept either the client or the server itself
  - **HTTPS** also protects client -> server communication more

# Password Security

- When a user “signs up” we have to store that user in our database
- Store their password as a plain string?
    - If our database gets hacked, those passwords will be up for grabs!
- We want to **encrypt** or **hash** the password before we store it

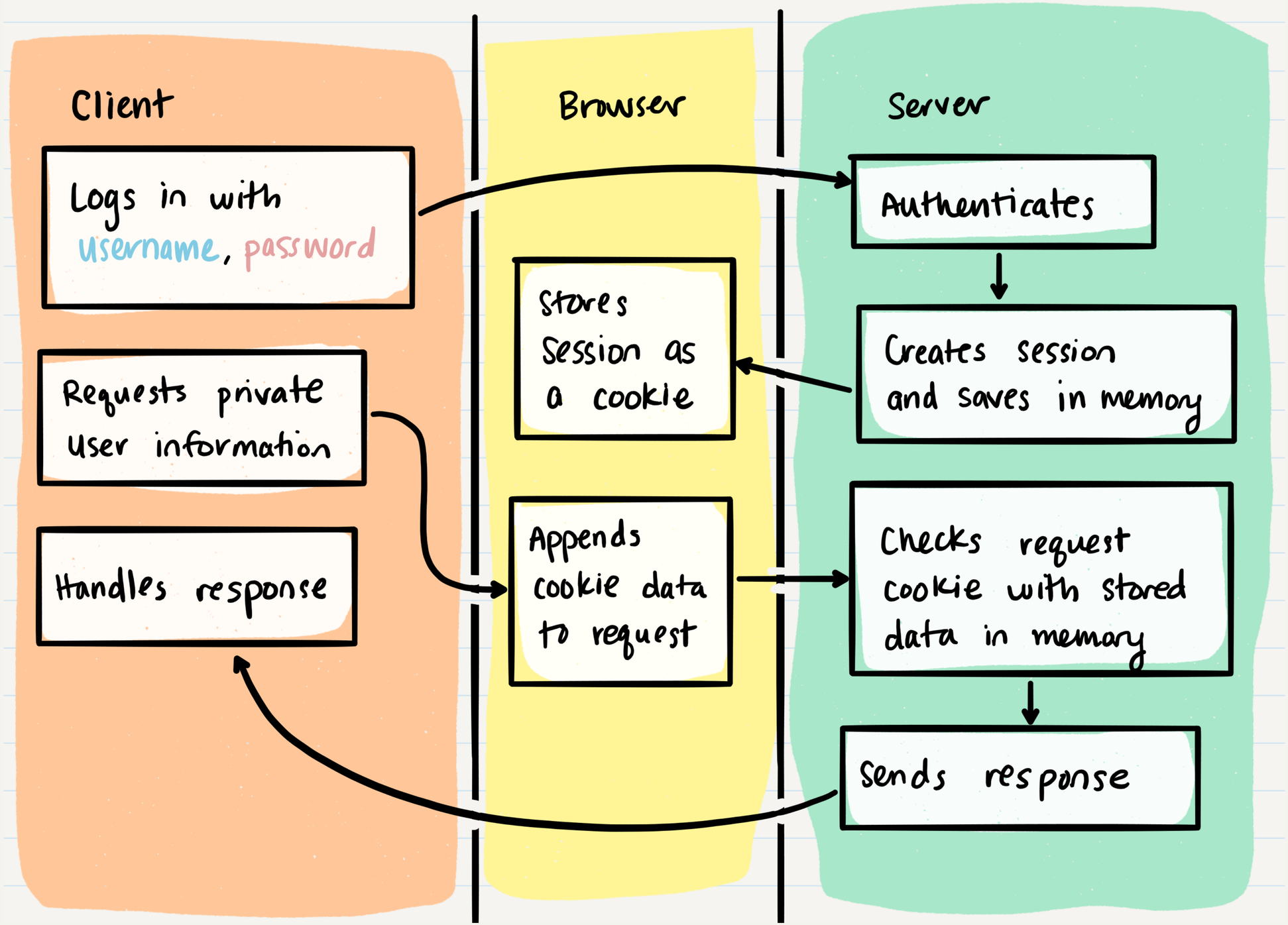


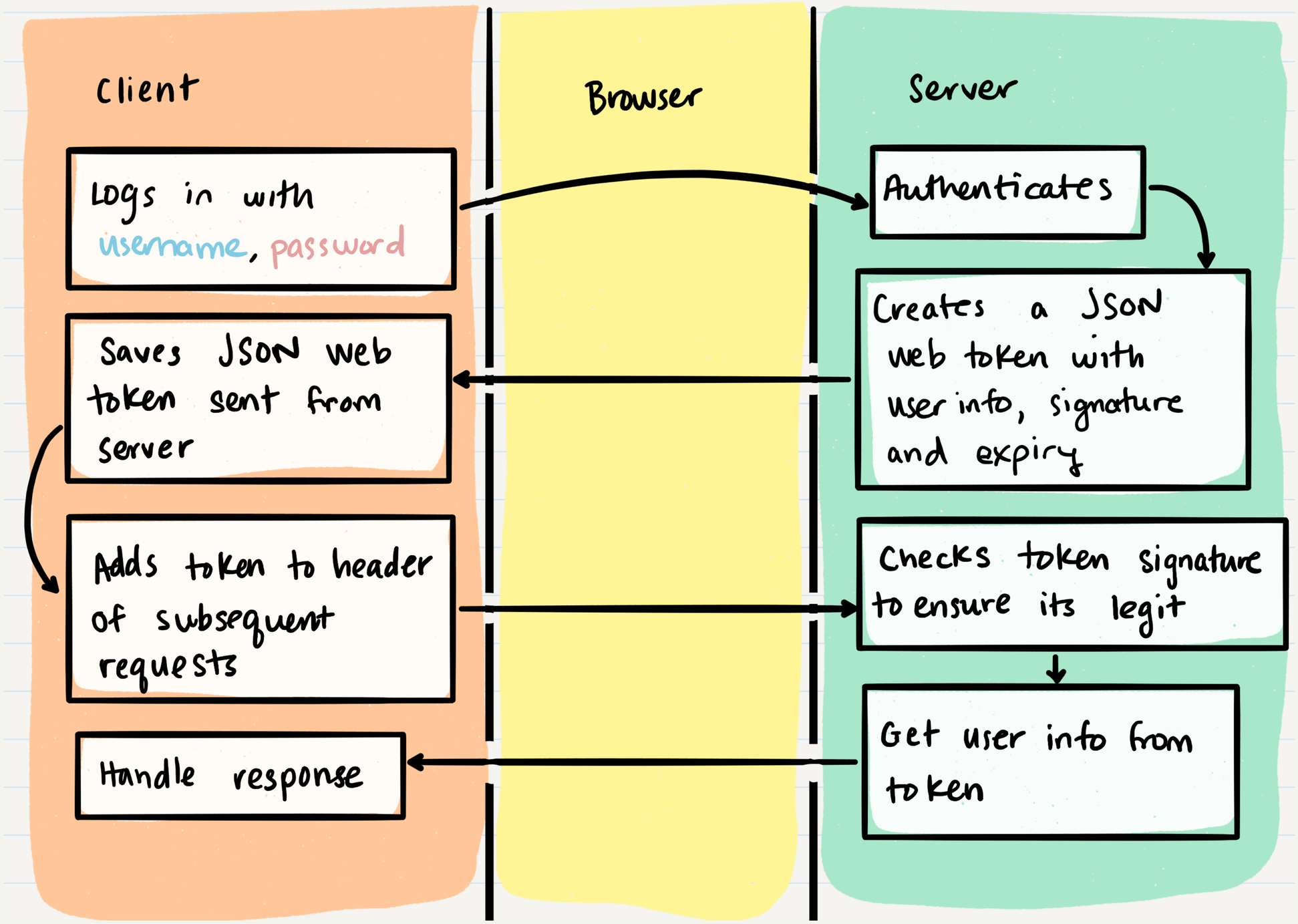
# Encryption and Hashing

- **Encryption** = two-way
  - I use a **key** to go from string to gibberish and back string if needed
- **Hashing** = one-way
  - I use a hashing algorithm to go from string to gibberish
  - Can’t go back
  - The **same string** will always hash to the **same gibberish**

# Sessions and Tokens

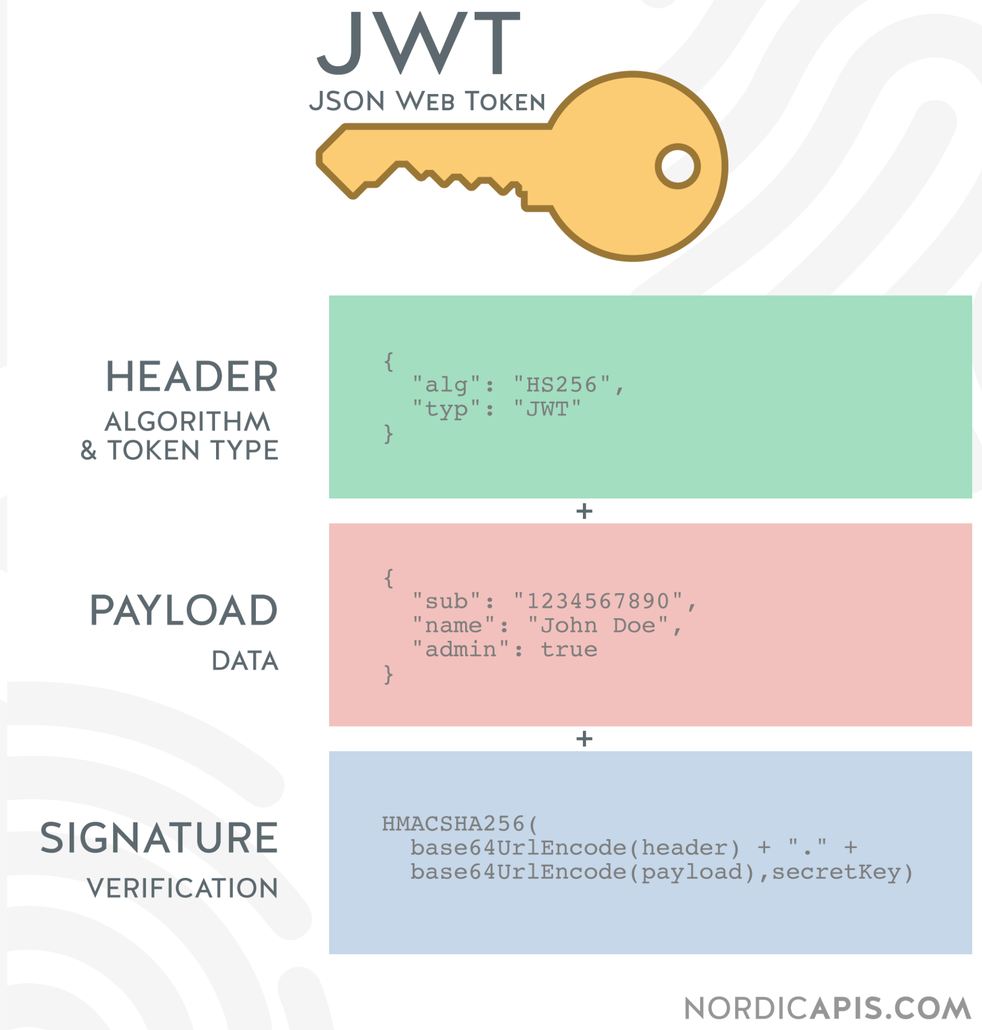
- How does an application “remember” that you were logged in?
- Server either creates a **session**, or gives the client a **token**
  - Session - something the server tracks by using a **cookie**
  - Token - something the server gives the client to pass along in any future requests


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

- A very common way to keep a client logged in
- The server generates a unique token
  - This is an encrypted string
  - Only the server has the encryption key
  - The data encrypted is typically some unique reference to the current user (user id for example)
- **JSON Web Token** is a package that generates tokens for us



# Lab 10 Overview