Code Fellows

Class 11
OAuth

seattle-javascript-401n14

Grading Recap

Just a Reminder:

- If Code Challenges require tests, they also require Travis!
- When you deploy to Heroku
    - Make sure you also push your local data to your remote database
    - Test your Heroku app functionality — it should be the same as your localhost
- Focus on getting setup + small requirements first - those are worth a lot of points!

Lab 10 Review

Code Challenge 10 Review

Local Elections are NEXT TUESDAY!!!

Not a meme

Please Vote

Vocab Review!

What is authentication?

What is authorization?

What is encryption?

What is hashing?

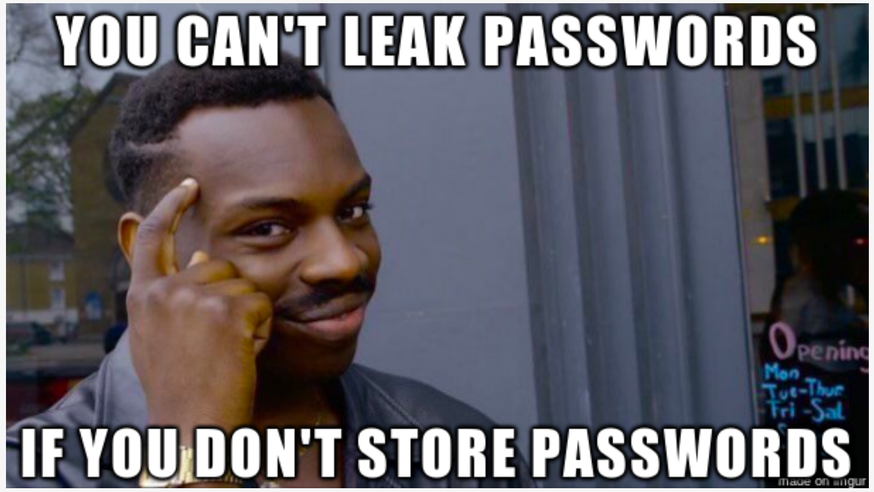
What is a session?

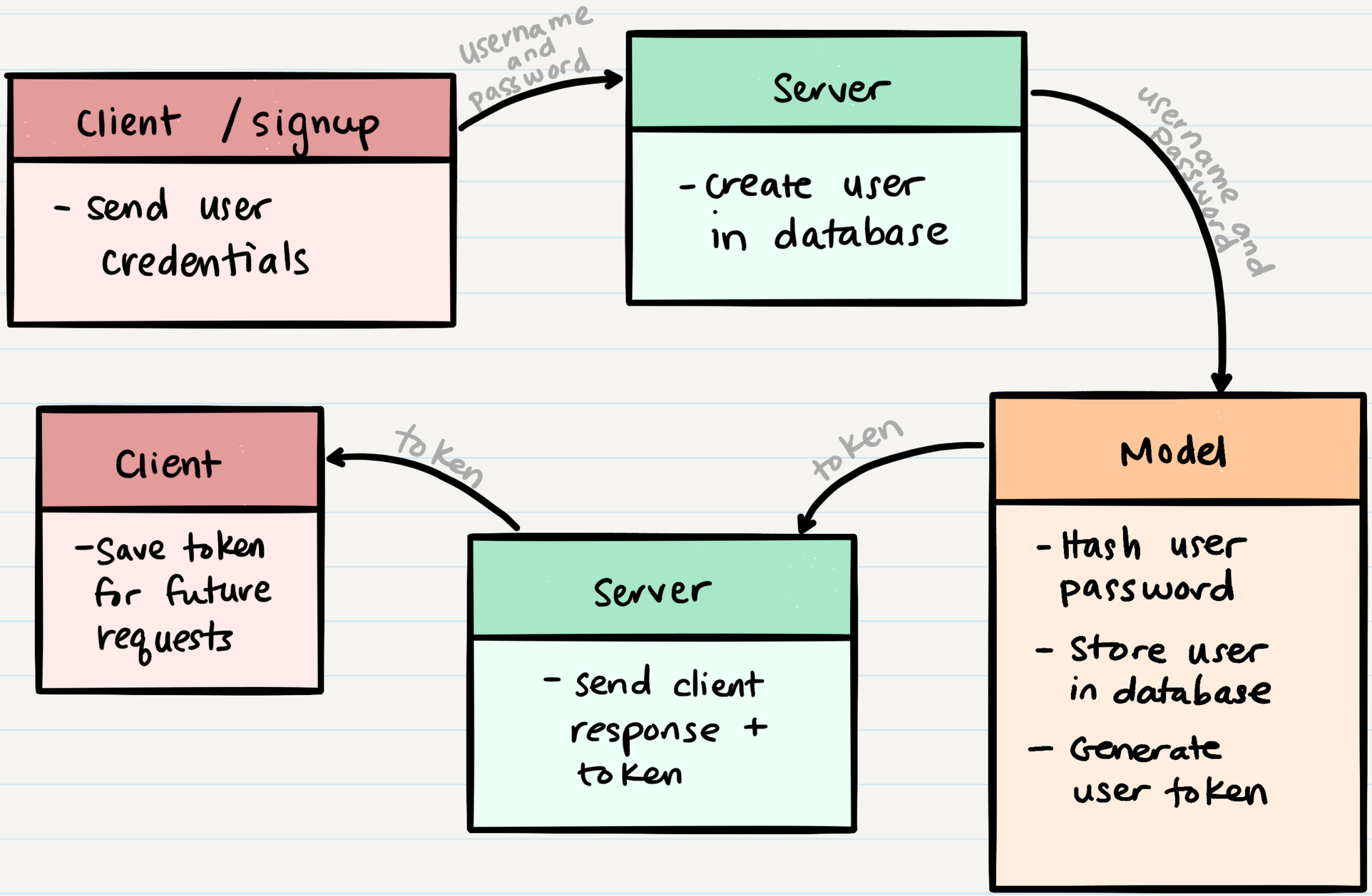
What is a token?

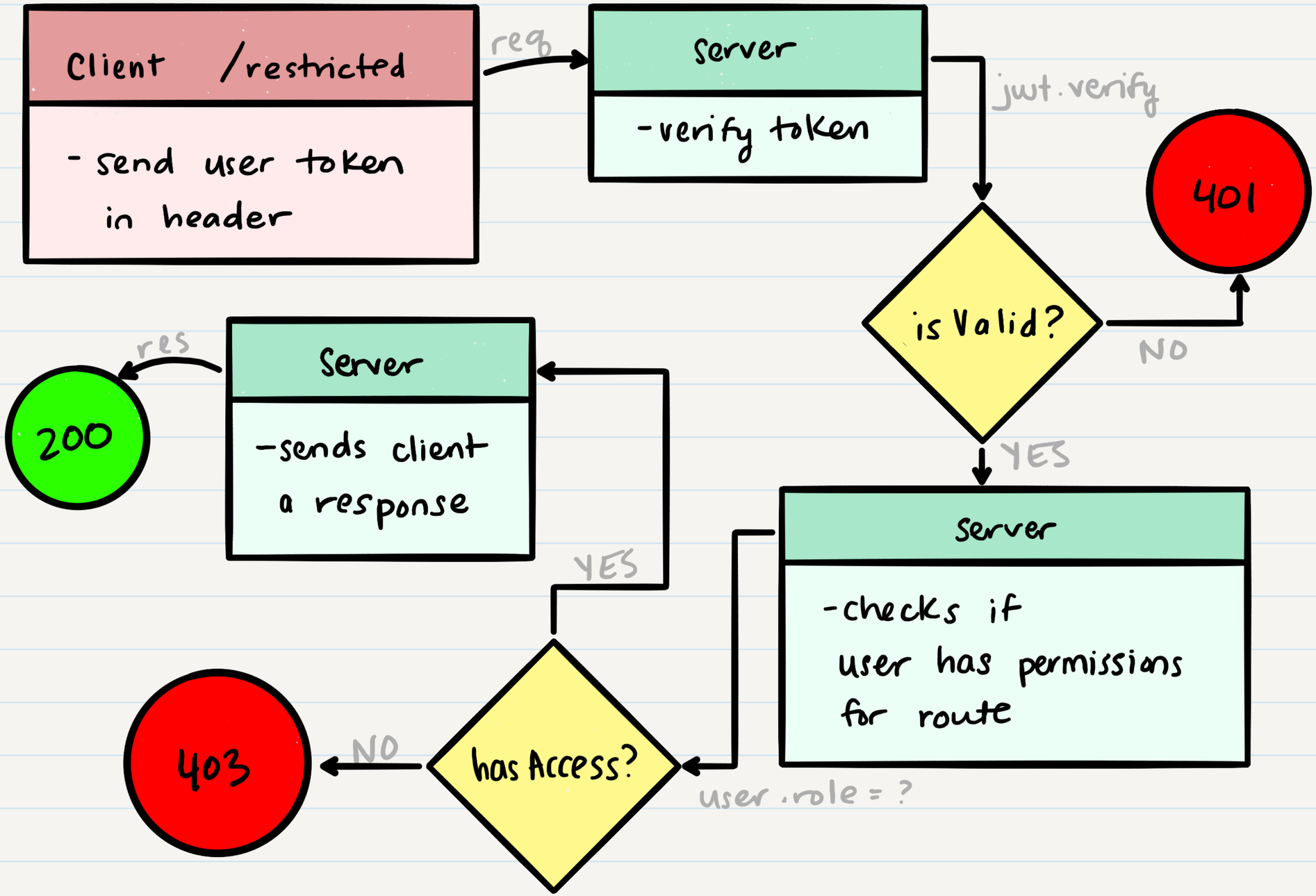
What is a secret?

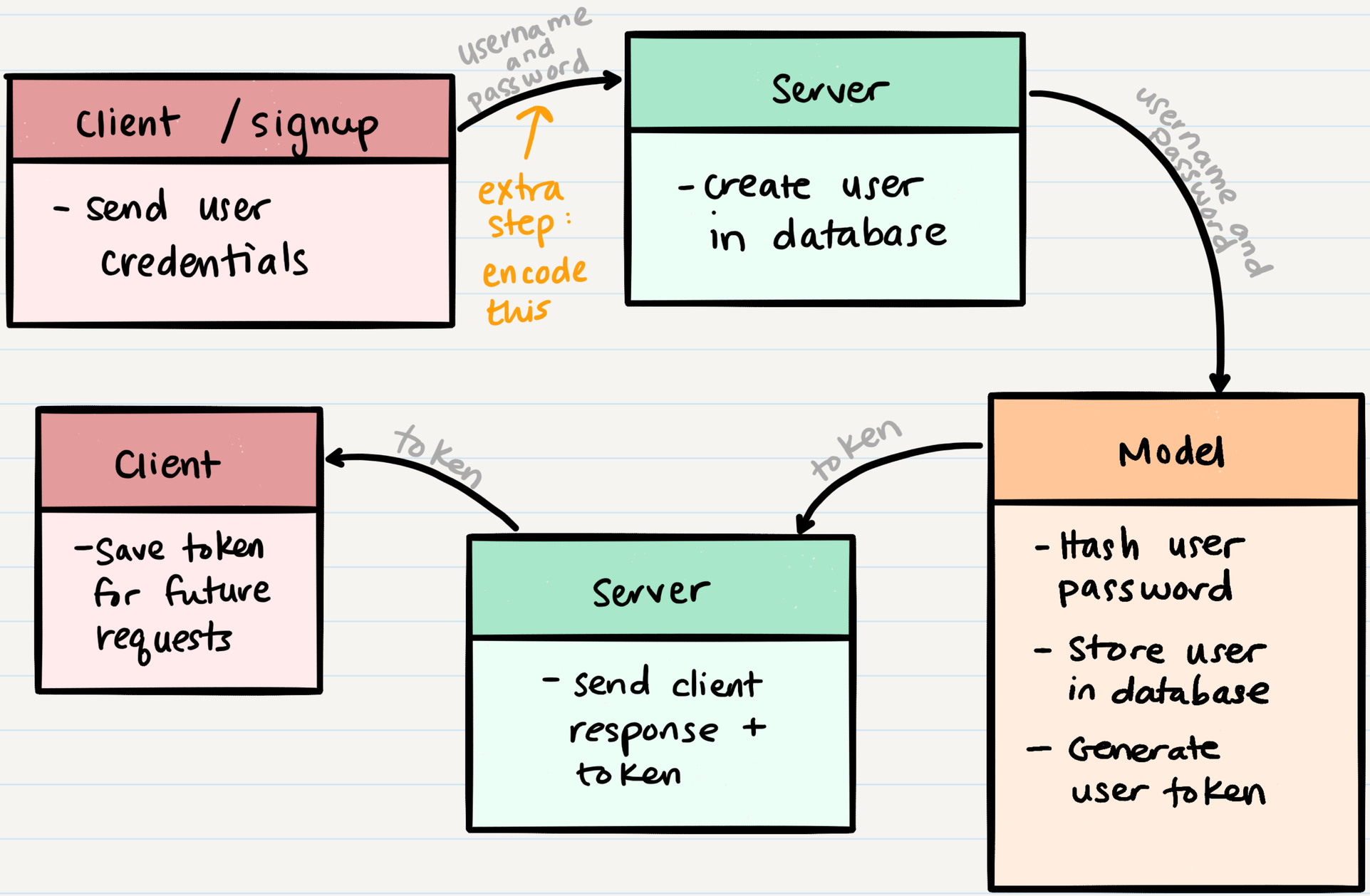
What we've learned

- We want to do our best to keep data secure when moving from client to server
- Clients need to **authenticate** that they are who they say they are
- Authenticated clients have access only to things they are **authorized** to have access to
- Servers need some way to let clients remain authenticated for subsequent requests
    - The server either maintains a **session** (stateful) or sends the client a **token** (stateless)




Client /signup
- send user credentials
username and password
Server
- create user in database
username and password
Model
- Hash user password
- Store user in database
- Generate user token
token
Server
- send client response + token
token
Client
-Save token for future requests


Client /restricted
- send user token in header
req
Server
-verify token
jwt.verify
401
is Valid?
NO
YES
Server
-checks if user has permissions for route
user.role = ?
has Access?
NO
403
YES
Server
-sends client a response
res
200


Client /signup
- send user credentials
Username and password
extra step: encode this
Server
- create user in database
username and password
Model
- Hash user password
- Store user in database
- Generate user token
token
Server
- send client response + token
token
Client
-Save token for future requests

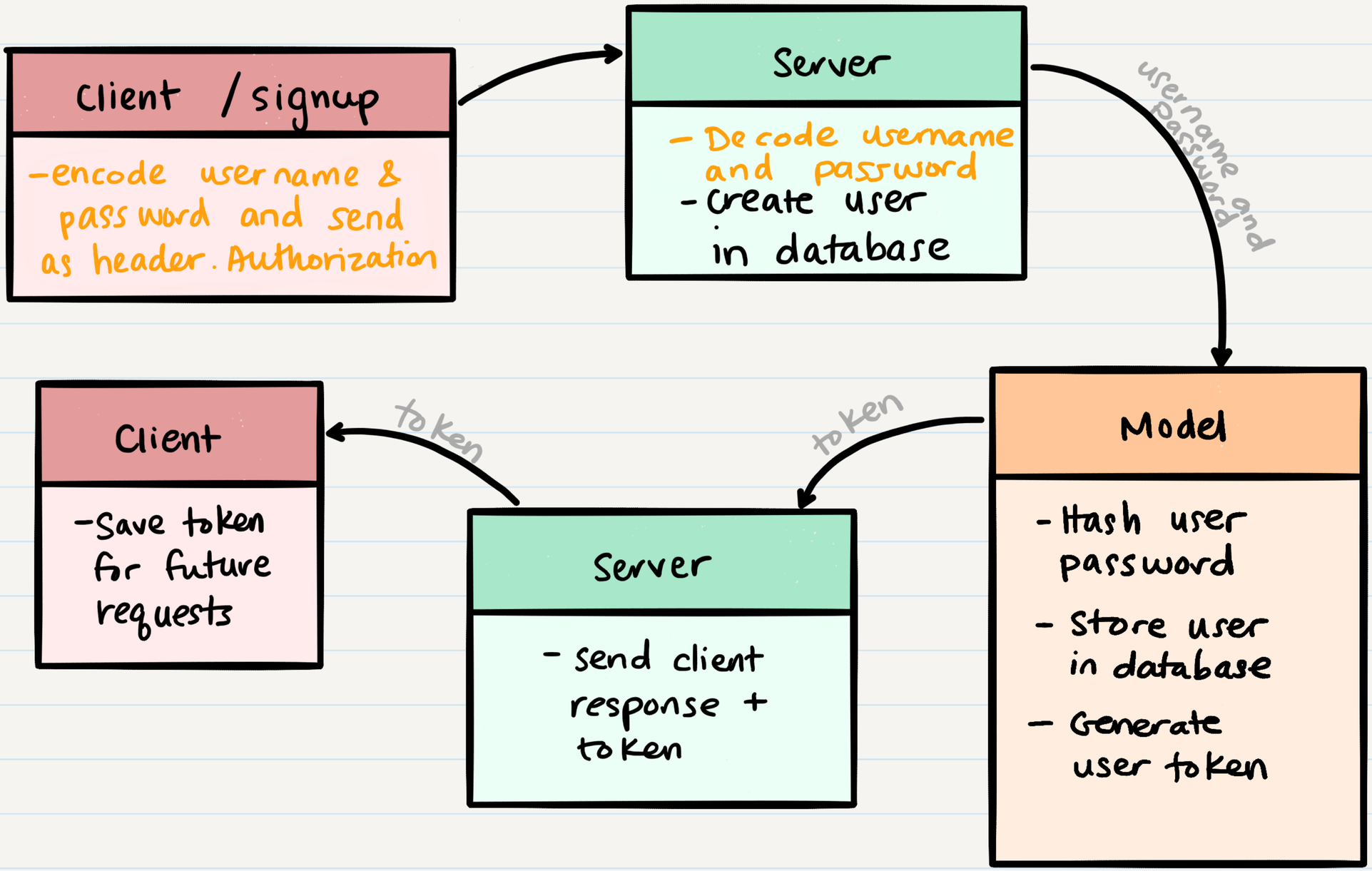
Encoding

- It’s a common standard to **encode** username and password data using **base64 encoding**
- This has no security benefit, it’s mainly so that:
    - The username and password are bundled together
    - We are treating the username and password as a Buffer instead of strings
    - This lets the username and password have special characters that JSON or string content might not like

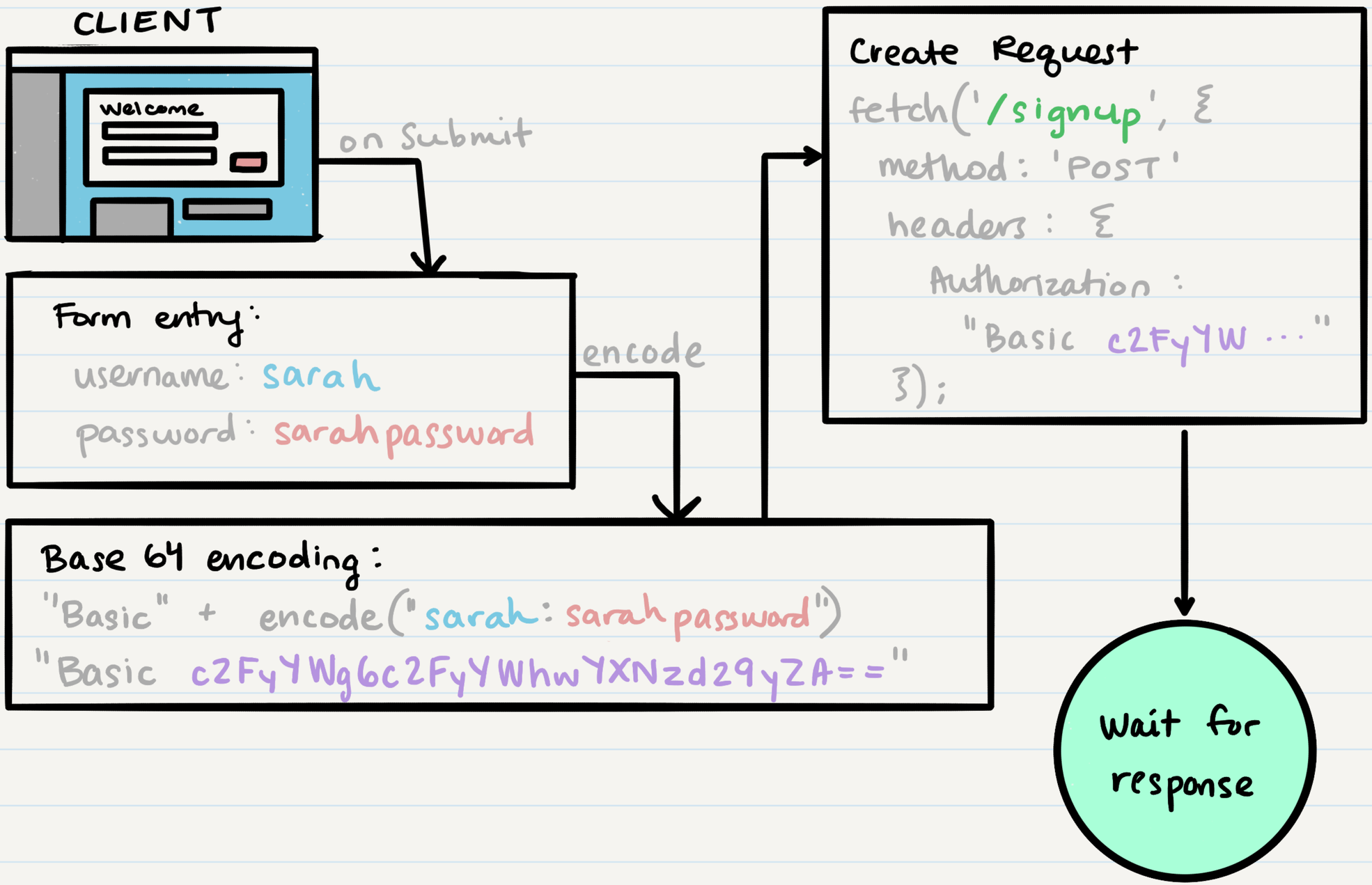
Base64 Encoding

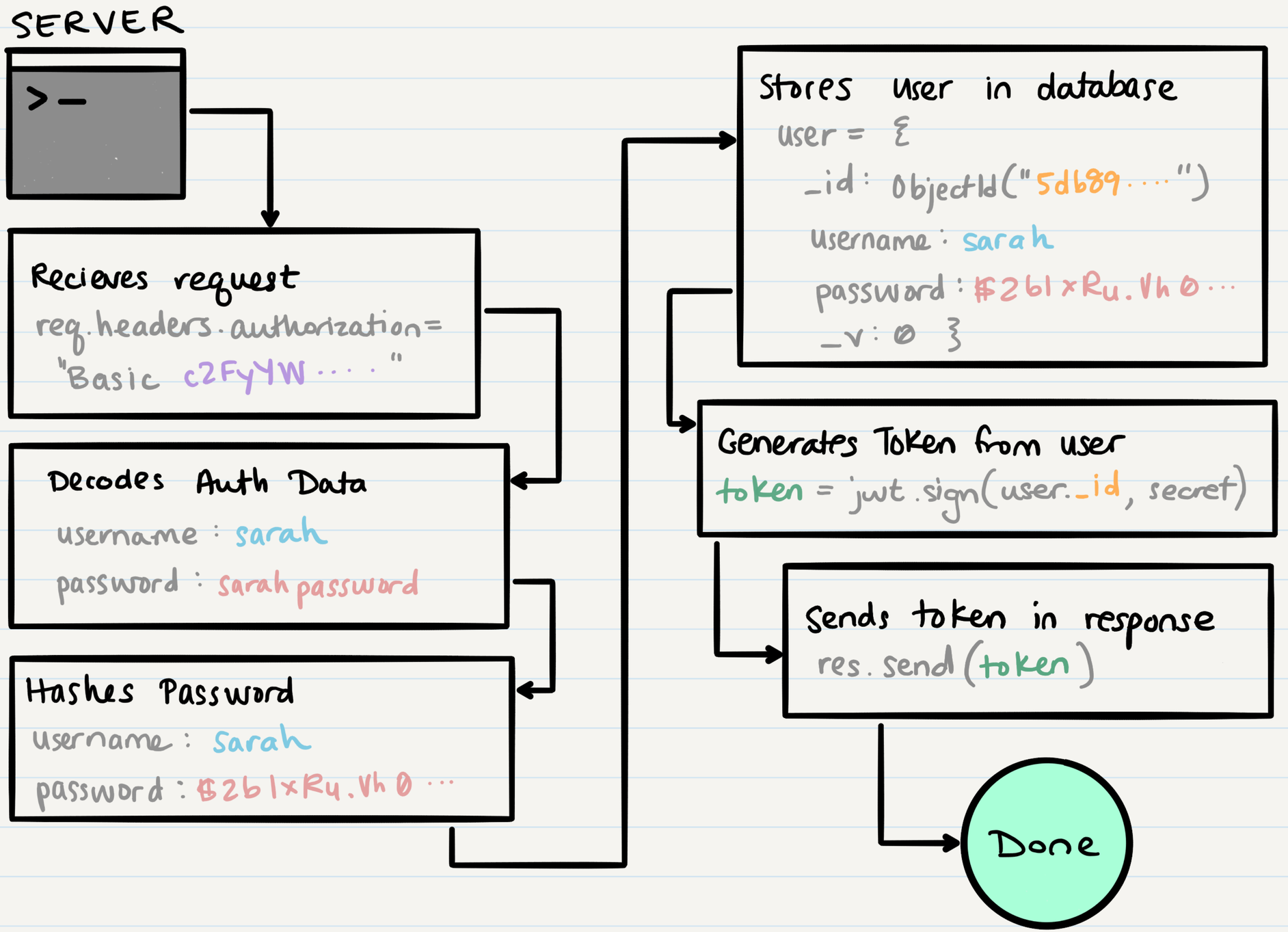
```
let username = 'myusername';
let password = 'mypassword';

// combine into 'myusername:mypassword'
let combined = username + ':' + password;

// encode using base64
let encoded = new Buffer(combined).toString('base64');

// encoded = bXl1c2VybmFtZTpteXBhc3N3b3Jk
console.log(encoded);
```

Using Base64 as “encryption”:

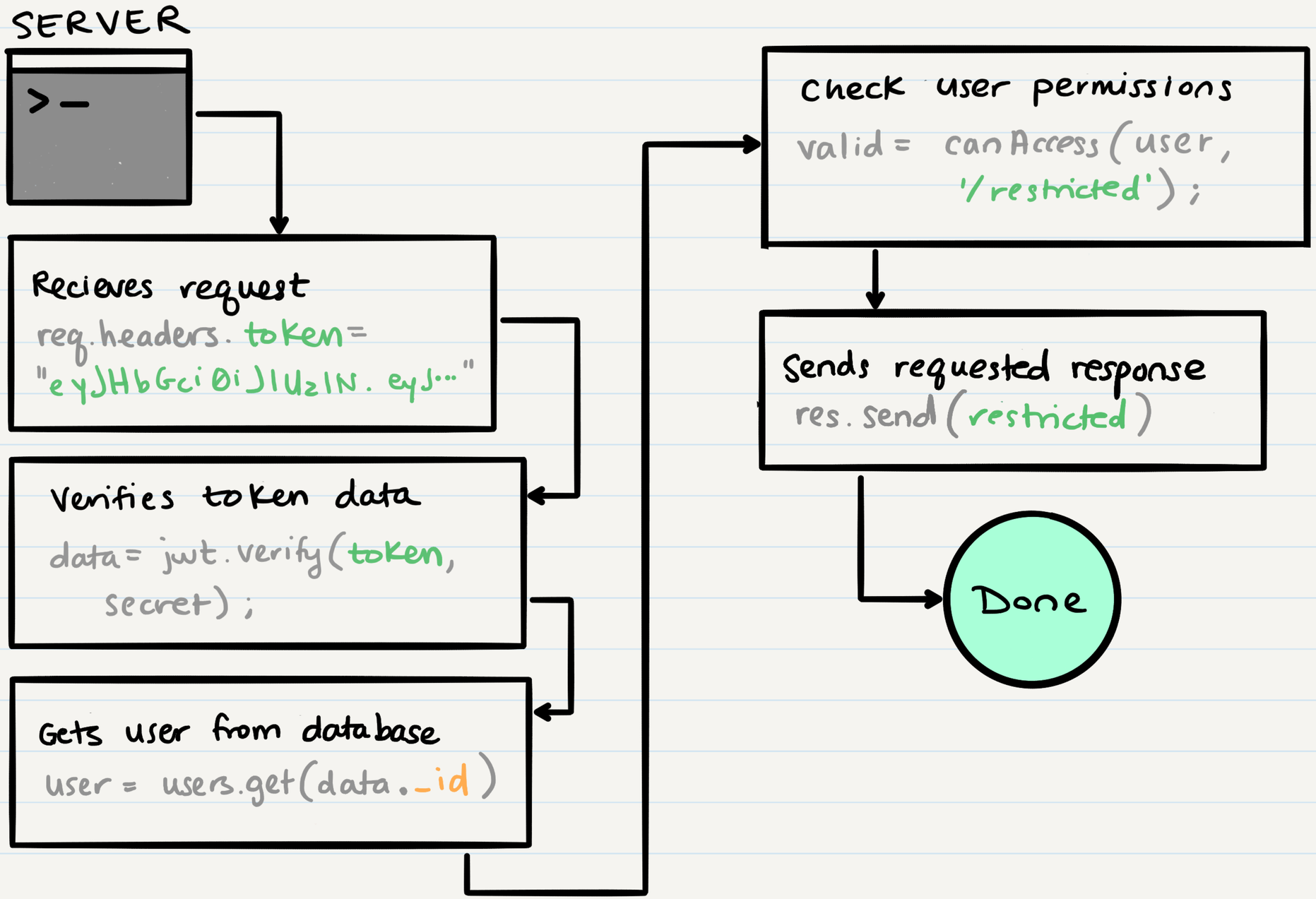

Client /signup
-encode username & password and send as header. Authorization
Server
- Decode username and password
- create user in database
username and password
Model
- Hash user password
- Store user in database
- Generate user token
token
Server
- send client response + token
token
Client
-Save token for future requests

Let’s review all of our data flow


CLIENT
Welcome
on Submit
Form entry:
username: sarah
password: sarahpassword
encode
Base 64 encoding:
"Basic" + encode("sarah:sarahpassword")
"Basic c2FyYWg6c2FyYWhwYXNzd29yZA=="
Create Request
fetch('/signup', {
method: 'POST'
headers: {
Authorization:
"Basic c2FyYW ..."
});
Wait for response

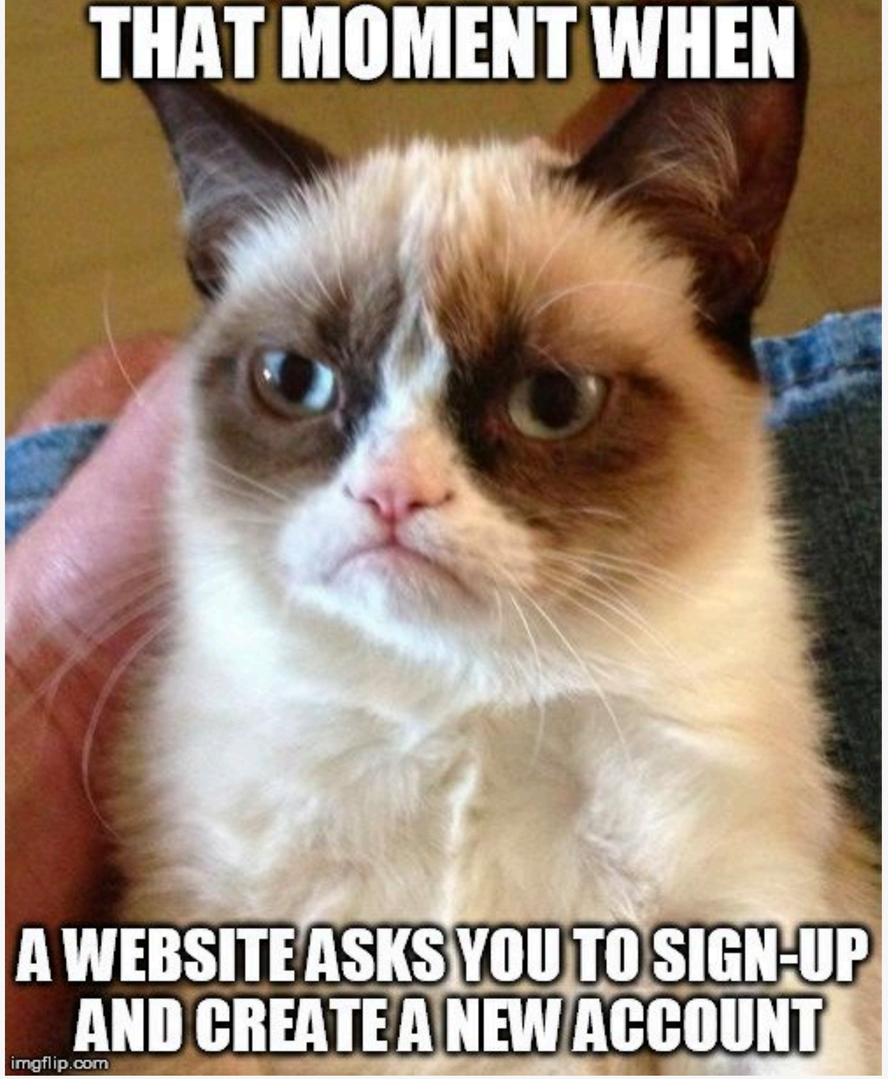

SERVER
>_
Recieves request
req.headers.authorization=
"Basic c2FyYW ... "
Decodes Auth Data
username : sarah
password : sarah password
Hashes Password
username : sarah
password : $2b1xRu.Vh0 ...
Stores user in database
user = {
_id : ObjectId("5d689...")
username : sarah
password : $2b1xRu.Vh0...
_v : 0 }
Generates Token from user
token = jwt.sign(user._id, secret)
Sends token in response
res.send(token)
Done


CLIENT
Get Private Data?
Create Request
fetch('/restricted', {
method: 'GET'
headers: {
token: token
});
Wait for response


SERVER
>_
Recieves request
req.headers.token=
"eyJHbGciOiJIUzIN.eyJ..."
Verifies token data
data= jwt.verify(token,
secret);
Gets user from database
user = users.get(data._id)
Check user permissions
valid = canAccess(user,
'/restricted');
Sends requested response
res.send(restricted)
Done

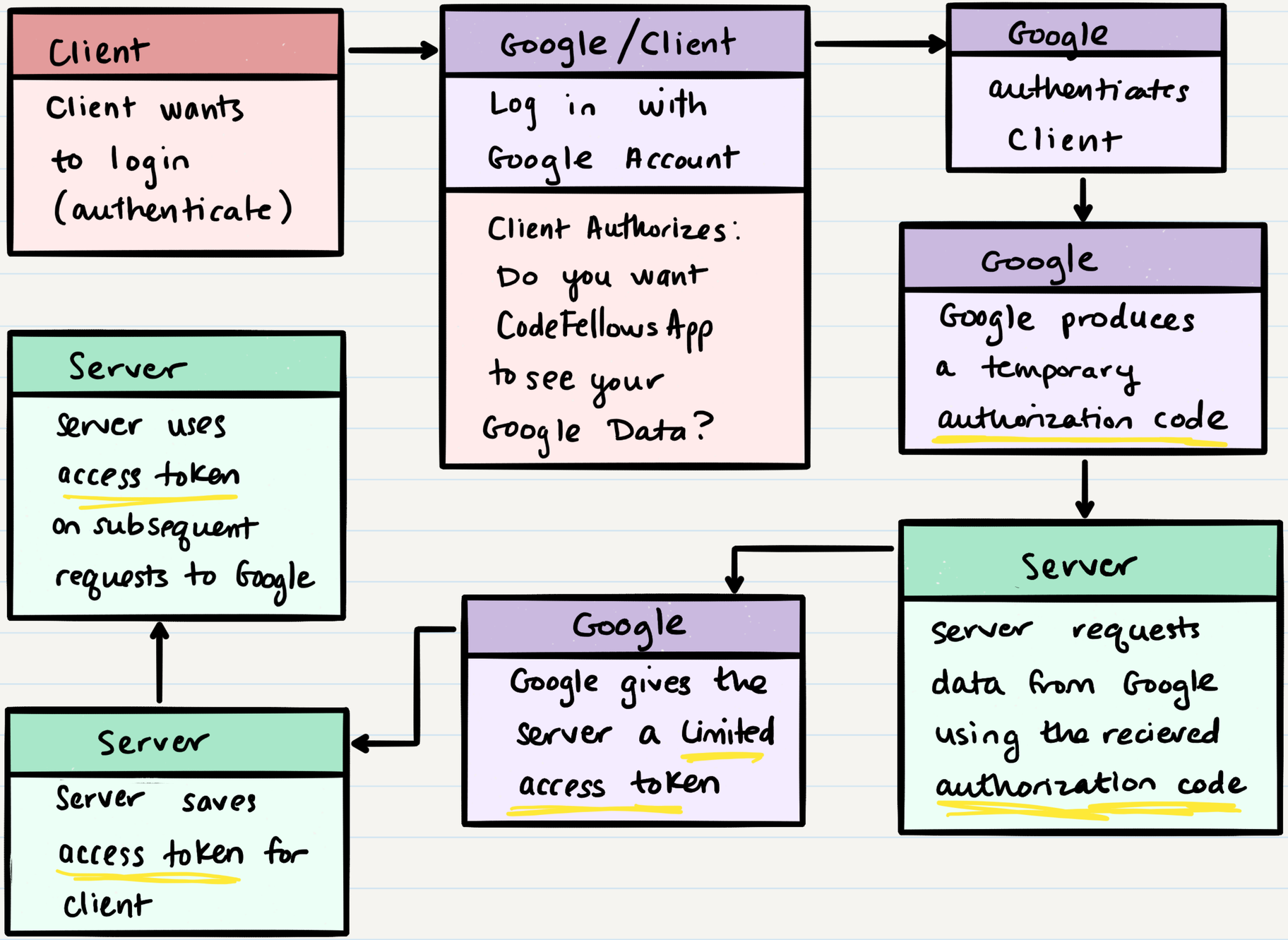
Adding Complexity

- Now we should have a rough idea of how to:
    - Create new users from a client request
    - Generate tokens for quicker client authentication
    - Check client authorization to load restricted data
- Let’s branch out even further:
    - Can we create new users from Gmail logins?
    - Can we make our server respond to more kinds of client requests? Not just encoded basic auth!



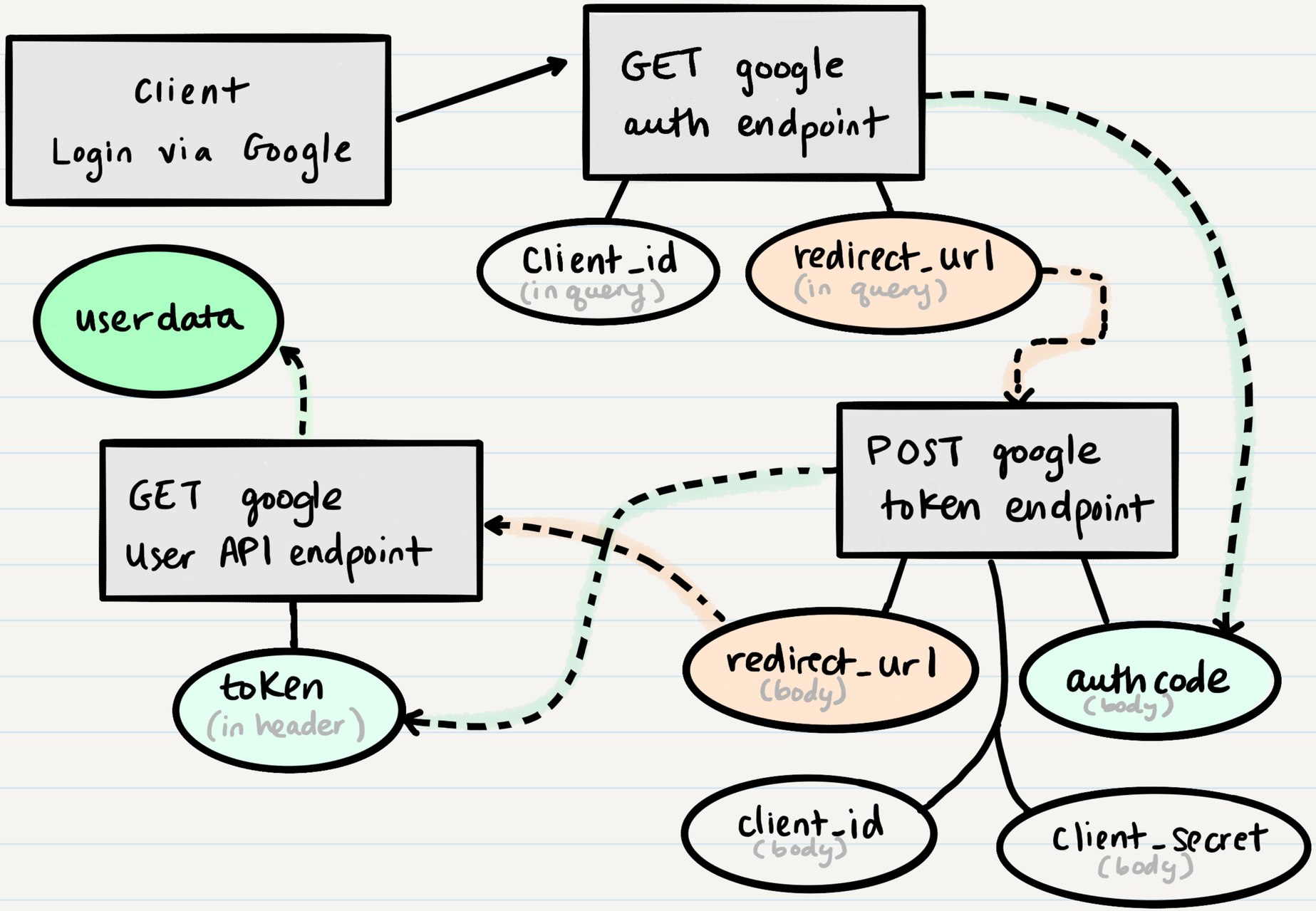
Introducing OAuth

- When two unrelated website are trying to accomplish something on behalf of a **shared user**, we use OAuth
- Sarah Smalls has an account on Gmail, and wants to use that to create an account on our CodeFellowsApp
    - Gmail authenticates Sarah
    - Gmail has a lot of data on Sarah, but doesn't want to send all of it. Instead, it sends a **limited access token**, giving CodeFellowsApp a small piece of Sarah's data
    - CodeFellowsApp creates its own database entry for Sarah, and stores the limited access token


Client
Client wants to login (authenticate)
Google / Client
Log in with Google Account
Client Authorizes: Do you want CodeFellows App to see your Google Data?
Google
authentiates Client
Google
Google produces a temporary authorization code
Server
server requests data from Google using the recieved authorization code
Google
Google gives the server a Limited access token
Server
Server saves access token for client
Server
server uses access token on subsequent requests to Google

What you need for OAuth

- An external web server that has OAuth features. A lot of large-scale websites (Twitter, Facebook, Google, Amazon) have OAuth capabilities
- Some way to set up a developer project with the external web server. Google has developers.google.com
- A **client ID** and **client secret** from this external web server. Your web application should store these in some environment variables typically.
- An **authentication endpoint** to send the user so that they can log in (usually this is a website with a lot of query parameters to it knows how to get back to your application)
- A **token generation endpoint** - once the client is authenticated and authorized, where to get a token
- An **API endpoint** - once I have a token, where can I go to load authenticated user data?


Client
Login via Google
GET google
auth endpoint
Client_id
(in query)
redirect_url
(in query)
userdata
GET google
User API endpoint
POST google
token endpoint
token
(in header)
redirect_url
(body)
auth code
(body)
client_id
(body)
Client_secret
(body)

OAuth is Generic

- Like REST, OAuth is a collection of rules on how to connect to external applications
    - OAuth is just specific to the use-case of authenticating a user
- Major pieces:
    - Client want to log in
    - OAuth server has an endpoint the client can go to
    - Client goes to endpoint with query params telling where to go after
    - Our Server is hit after client hits “log in” - OAuth server as given us an authorization code
    - We use the code to ask OAuth server for a token
    - We use the token to get user data from OAuth server

What's Next:

- Due by Midnight Tonight:
  - **Learning Journal Class 11**
- Due by Midnight Sunday:
  - **Feedback Week 6**
  - **Instructor Sync Reflection**
  - **Career Coaching**
    - **Personal Pitch**
    - **Resume**
- Due by Midnight Monday:
  - **Code Challenge 11**
- Due by 6:30pm Tuesday
  - **Lab 11**
  - **Reading Class 12**
- Next Class: **Class 12 - Bearer Authorization**

Questions?