GOAL: MULTIPLE-PAGE WEB APPLICATION (CHAT)

Version 1:

- User will see chat page
- User can add message
- User will see updated messages
- A different user can load the page
- Other user will see updated messages
- Other user can add

BREAKING DOWN THE NEEDS:

- One "page"
  - We will add other pages to handle login later
- All dynamic HTML
- Static CSS
- Only server-side JS

ACTIONS

- Initial Load
- Post Message
- More cases later

DATA (STATE)

- List of users
- List of messages
  - Text
  - Sender
  - Timestamp

SETUP NEW SERVER

- Create a directory `chat-mpa`
  - `mkdir chat-mpa` / `md chat-mpa`
- Move into that directory
  - `cd chat-mpa`
- Make a new npm package
  - `npm init -y` (creates `package.json)
- Install `express`
  - `npm install express` (local install)

CREATE FILES

- create a `public/` directory for static files
- create a `server.js` file

both names are just names we pick

- could be different

```
const express = require('express');
const app = express();
const PORT = 3000;

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

// Last step:
app.listen(PORT, () => {
  console.log(`http://localhost:${PORT}`);
});
```

SKELETON OF PARTS

Best to confirm everything works in small steps

- If something doesn't, you only have a small amount of code to look for a bug in

"Stub" out a lot parts

- functions that don't do anything
- functions that return hardcoded values
- data that is created with some fake contents

PAGE SKELETON

First, create a dynamic **handler** for the route **/**

Longer variables names are NOT BAD

But the community convention is to use `req` and `res` instead of `request` and `response`

```
app.get("/", (req, res) => {
  res.send(`
    // Paste the chat html here
  `);
});
```

Add a `chat.css` to the public folder

Restart the server and confirm it works

NEXT - FILL HTML DYNAMICALLY

Again, do it in parts and confirm each part

- Break up the page into function
    - Each function returns bits of text
    - Each function is passed the state
- Separate the logic
    - Controls the routes (`server.js`)
    - Controls the data logic (`chat.js`)
    - Wraps the data in HTML (`chat-web.js`)

SEPARATION OF CONCERNS

Separation of Concerns is a programming strategy.

Parts of an application that don't NEED to be coupled (deeply tied together) AREN'T coupled

You want this separation because it makes changes more simple and easier to understand.

Imagine:

- Artificial kidney: Complex, hard to be confident in
- Water filter: Less complex, understandable

LAW OF DEMETER

Law of Demeter (Principle of Least Knowledge)

The less parts know about other parts, the more each part can change without requiring changes in other parts

SIDE EFFECTS

Something the function alters outside of return value

"Pure" means without side-effects - same inputs always generate same outputs

A Useful program must have side effects, but side-effects introduce complexity (the enemy!)

So always be careful in accepting side-effects.

Side effects should be

- Obvious from the function name
- Predictable