Goal: Multiple-Page Web Application (Chat)

We are going to write a chat application

- Start very simple
    - All messages from same one user
    - To anyone reading page
    - No login yet
    - Add more features later
- See previous messages on page load
- See all messages after entering new one
- All backend generated HTML
- Static CSS

CSS? Images?

- Any CSS and images will be loaded as **static files**
  - Exist as files under the **document root**
- But all HTML will be dynamically generated
  - NO `.html` files

State is a vital concept

state = all the key variables that can change

- **ui state** = data about visuals
  - varies by interaction (session/page/user)
- **data state** = data about the concepts
  - users may care about different parts
  - but data state is universal
- **derived state** = based soley on other state
  - NOT stored as state, could get out of sync
  - calculated as needed from state

State Examples

Examples:

- Is a section expanded/hidden? (**ui state**)
    - Different tabs will have different **ui state**
        - Even for same user
- User's profile data (**data state**)
    - Doesn't tend to change per tab
    - Diff for diff users, but from same pool of data
- Is there a new-to-user message? (**derived state**)
    - from message date + user's last read date
    - if stored as state, creates ways to be wrong

Server-side State

Server-side apps/pages have **state** only on the server

- Browser maintains a separate state

The duration of a request will

- Draw state from stored data
- Update state from request
- Generate page
- Send response
- Request is over (state ends)
- Stored Data remains

State for a Chat Application

Server stores

- List of logged in users
- List of messages

"current user"

- Only means something mid-request
- Can't store this on the server
- Multiple users, multiple requests

Programs are Data Flows

We start simple:

- Load Page
- Send Message

Respond to Flows

- Load Page
    - Generate Page from stored state
    - Response: Page
- Send Message
    - Update stored state
    - Response: Redirect to Load Page

Separation Of Concerns (SOC)

Design programs with **Separation of Concerns**

- A "concern" is a vague term
    - Break a set of problems into related groups
        - each group is a "concern"
    - Can repeat at different "zoom" levels
- A technique of abstraction
    - Allows **decoupling**

Law Of Demeter/Principle of Least Knowledge

Write your code to "know" as little as possible

- Everything the code assumes can be changed
    - Breaks the code/Creates bugs
- Changing anything means seeing who cares
    - Ignorance is bliss

SOC and Abstraction

SOC makes use of **abstraction**

Abstraction

- is vital to make programs understandable
- has a cost in execution (meh)
- has a cost in cognitive overhead (Wut?!)

Abstraction is always a cost-benefit calculation

- Sometimes an easy calculation
- Sometimes very tricky
- "It depends!"

"Dumb" is good

Don't be clever -- Programming Wisdom

"smart" or "clever" code

- Depends on a lot of things
- Easy to break
- Hard to debug

"dumb" code is better

- Decoupled
- Durable
- Modifiable

"Elegant" code is the ideal

Good parts of "dumb"

- Without tedium

Model-View-Controller (MVC)

MVC - software pattern for User Interfaces (UI)

- Involves Separation of Concerns

3 Parts:

- **Model** - the core logic separate from UI
- **View** - the UI
  - populated by Model
  - tied to messages to Controller
- **Controller** - the glue connecting them

MVC Variations and Implementations

- Many MVC variants (MVVM, MVA, MVP, etc)
- Diff Implementations based on language/framework

Core Tenent: Separate the Concerns of

- The Data (model)
- The Presentation (view)
- How changes are triggered (controller)

Simple server-side MVC Implementation

Just have different modules for different purposes

- **Controller** for our app is `server.js`
  - Decides what to respond with (**View**)
  - Can tell data to update (**Model**)
  - Can pass data (**Model**) to presentation (**View**)
- **Model** is **core logic without HTML** in a module
  - Also contains stored data (**state**)
- **View** is a module w/functions to return HTML
  - Must be passed data (passed the Model)

Frameworks may wrap/enforce, but concepts just exist

Putting this together

```
// server.js (Controller)
const chat = require('./chat'); // Model
const chatWeb = require('./chat-web'); // Views
// ....
app.get('/', (req, res) => {
  res.send(chatWeb.chatPage(chat));
});
```

- Code unseen, we know what these should do
  - `chat` is an object of all data + related methods
  - `chatWeb.chatPage()`
    - returns the HTML for the page
    - is passed the Model data

Making changes

```
app.post(
  '/chat',
  express.urlencoded({ extended: false }),
  (req, res) => {
    // Not shown - get data from request
    chat.addMessage({ sender, text }); // update Model state
    res.redirect('/');
  }
);
```

- model state persists between requests
- state about this message lives only in this callback

Summary - State

State is all the key variables that can change

- **ui state** - which is based on the UI
- **data state** - which is true despite UI

derived state isn't stored with state

- but is decided entirely BY state

Summary - Separation of Concerns (SOC)

Separation of Concerns

- break up problem sent into groups
    - Each group **decoupled** from others
    - All items in a group related to one another
    - Done at high- and low-levels of the problem

Law Of Demeter/Principle of Least Knowledge

- Each thing should know as little as possible
- **decoupling**

"Dumb" code is GOOD code

Summary - Model-View-Controller(MVC)

MVC Pattern

- A form of SOC for UI
- Separates
    - Data and non-UI logic (Model)
    - Presentation/UI (View)
    - Glue controlling the flow (Controller)
- Easier to change
- Breaks less often

MVC can be done in code w/o frameworks