Using JS

Only experience can teach

- All the options available
- How to break down a problem

But...

- Some best practices can save you a lot of time

State

An application has "state"

- The current values for all things that can change

A chat application

- Are you logged in?
- As who?
- Are there messages?
- What are they?
- Are you typing a message?

How to store state

Store your state in variables/object

Use those to update the screen as needed

Do not read the HTML (DOM) to recapture the state

How NOT to store state

Example: You show a list of users on the screen.

To get the list of users, should you read the DOM?

No. Why?

- The screen is the visual output
- if you alter the display, you change how to get the list that way
- As your display gets more complicated, so does all your state interaction

Model-View-Controller (MVC)

MVC is a common best-practice pattern for many situations:

- Something manages your data (model)
- Something the flow of the application (controller)
- Something translates the data to output (view)

You can see this is in the chat-mpa assignment

We will change a lot, but that breakdown will remain

Debugging JS

When something isn't working

- There are a few ways to tackle the problem
- There are a few things NOT to do

Narrow the scope

FIRST: make sure you know what's wrong

- Validate you know up to which point things work
- Check for error messages
- Check values of fields and properties

Don't spend time fixing the wrong "problem"

Which line isn't working as you expect?

Checking for errors

In Node, check the console output for error messages

In Browser, check the console

- console erased on page load unless "preserve log"
    - redirects are page loads

Network subtab holds info on network calls

- check for errors
- separate "preserve log" option here

Check values

Inspect Element

- Are CSS classes and HTML properties correct?
- Is your CSS selector matching? overidden?

Network

- Check to see that form fields were passed
- Verify the correct method (GET/POST/etc) is used
- Check status code
- Check values in response

Console.log

"old" saying, basically:

```
When I was a new coder, I relied on console.log

When I was an senior coder, I relied on debuggers

When I was a master coder, I relied on console.log
```

`console.log` is fine **IF**

- You clean it up before submitting (!)
- You know the triggering state

Browser Debugger

"Sources" subtab

- "Watch" a variable visible to the current scope
- "Breakpoints" to stop on
    - Can make conditional stops
    - Requires page reload if code already done
- "Scope" to see other variables
- "pretty print" minified code (lower left `{}`)

Client side storage

Sometimes you want to store information outside of the page *on the browser*

- Cookies
- localStorage
- IndexedDB

BE CAREFUL

- Limited security
- Users will change browsers/machines
- Can get changed/deleted by user/browser
- Not all clients are browsers

Cookies

"Cookies" are just an HTTP header

- Special is how browsers treat them
- Browser sends cookies along with each request

Cookies are text-based key/values pairs

- limited to a URL and descendant paths
- might have expiration date
- might (should) require HTTPS
- might not be accessible to JS
- shared between tabs

When to use cookies

Most Common:

- Store a random key that is IS also server-side
    - a "session" identifier
- request with key lets server read extra data
- Depends on that random number staying secret
- Cookie/session should NOT hold **application** state
    - because user might be using multiple tabs
    - each page/tab has its own application state
    - session data is useful regardless of state

When not to use cookies

DO NOT use cookies to store:

- Sensitive data (CC numbers, passwords)
- Personal data (addresses, etc)
- Application state
- Big data
- Data hard to represent in short bits of text

Local Storage

localStorage and sessionStorage

- key/value
- client-side only (not sent to server)
- JS only (no JS, no using localStorage)
- Store bigger values than cookies
- localStorage is shared between tabs
    - sessionStorage is NOT
- localStorage does not expire
    - sessionStorage lasts until browser quits
- Still domain-limited
    - Not path limited

When to use localStorage

- Store JS-applicable preferences
- When data too awkward for cookies
- When user switching devices isn't a problem
- To keep tabs in sync with choices

Rarely want sessionStorage

- Lack of tab-sharing causes confusion

When NOT to use localStorage

- Cookie security restrictions still apply
  - Sensitive data (CC numbers, passwords)
  - Personal data (addresses, etc)
- If the data is needed without JS

IndexedDB

Browser-side object-based DB

- NOT relational, NOT table-based

Asynchronous

- Like a click handler: response will happen later

JS-only

Stores larger data, non-expiring

- Browser can limit and/or delete without warning

When to use IndexedDB

Fairly few cases

Transactions

Larger data, but unreliable storage

Non-trivial to use

When NOT to use IndexedDB

- Cookie security restrictions still apply
    - Sensitive data (CC numbers, passwords)
    - Personal data (addresses, etc)
- If the data is needed without JS
- If you don't want the complexity

What is a Polyfill?

Polyfills add newer functionality to older JS

Example:

- `forEach()` is a method on Arrays
- takes a callback, calls that callback with each element in turn

You can write this in JS versions prior to it being standard

How do Polyfills work?

- Check to see if the feature exists
- If not, add the new function to the prototype

Why all methods in MDN refer to `Foo.prototype.`

- The **only** time you modify native prototypes
- Someone else has done this for you

JS Tools

JS ecosystem has many tools beyond the engine

- linters
- minifiers
- bundlers
- transpilers

Linters

Linters (not JS-specific): programs to check syntax

- For purely stylistic preferences
- For patterns that are technically correct
    - but tend to lead to errors

Formatting is long debated

Linters can help find unintended errors

`eslint` is the most common JS linter

- Many IDEs have linting built-in

Prettier

- Newer tool (JS only?)
- Auto-formats code to a common style
- Popular among those that don't want to argue

Minifiers

- Removes unneeded whitespace
- Replaces variable names with short ones
    - where possible

Reduces file size of JS/CSS/HTML

Makes them harder to read/debug

Is NOT security

Smaller size CAN matter

Bundlers

Frontend struggles to handle multiple JS files well

"bundlers" convert multiple files into one

- Some use NodeJS `require()` syntax
- Others use the newer standard `import` command

Common bundlers:

- Webpack
- Rollup
- Browserify

Browersify - an example bundler

```
// Commands
mkdir b-ify
cd b-ify
npm init -y
npm install browserify
```

```
// foo.js
const bar = require('./bar');
console.log(`The other file says ${ bar() } successfully`);
```

```
// bar.js
module.exports = function() {
  return `"I like cats"`;
};
```

```
// Commands
browserify foo.js -o bundle.js
```

```
// index.html
<script src="bundle.js"></script>
```

Transpilers

Transpilers are "**trans**forming com**pilers**"

- input (something)
- output JS

Examples:

- Input typescript, output JS
- Input clojurescript, output JS
- Input modern JS, output older JS
- Input **future** JS, output modern JS

Example: See Babel at **https://babeljs.io/**

Hot reloading

During Front end development, it is common to have a setup that will reload your changes easily

- great during development
- not great for when the product is shipped

In This Course

- We will start without tools
  - Your IDE might have linting
  - We will add some tools
  - Section 3 will use a few
- Tools make things easy
  - But understand the concepts without them
  - You aren't lost if they aren't working

BUT: You may think WebDev is annoying