Front End Considerations

When calling services

- What does the Front End need to consider?

The Biggest Lie in Web Dev

- We use "spinners" to tell the user to wait
- Does NOT indicate actual computer activity

Using a Spinner

- Add to page before starting a (long?) async action
- Remove when complete
- If something breaks and you don't remove it
- ...it keeps spinning
- ...does NOT indicate anything is "thinking".
- It is just an animated image

Spinner Example

```
appEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('show-cats') ) {
    state.isLoadingCats = true;
    render();  // Shows spinner
    fetchCats()  // Starts async
      .then( cats => {   // Runs after fetch delay
        state.cats = cats;
        state.isLoadingCats = false;
        render(); // shows data, not spinner
      })
      .catch( err => {
        state.error = err;
        state.isLoadingCats = false;
        render(); // shows error, not spinner
      });
  }
});
```

What is the Spinner?

- Can be image
    - With or without CSS to "spin"
- Can be text (Ex: `Loading...`)
- Can be pure CSS
    - **https://css.gg/app?s=spinner**

When to show Spinner/Loading Indicator

- To signal user that something is coming
- Async calls can take surprising time!
- Don't show misleading state
- Users get
    - Frustrated
    - Confused
    - And worst, BORED
        - Start clicking
- May show one spinner for multiple loads
    - Rather than many small spinners

Separating Service Call Concerns

Complexity can drag your code down

- Hard to find bugs
- Hard to fix bugs
- Hard to add features

Write it clean from the start

- Good, meaningful names
- Keep concerns separate

Example HTML for poor JS coding

Sample HTML for a simple TODO list

```
<ul class="tasks">
  <li class="task">Do INFO6250 work ONTIME</li>
</ul>

<div class="to-add">
  <input name="taskName" class="task-to-add">
  <button class="add-task">
</div>
```

Handling a user action?

- Attach a listener
    - Start spinner
    - Gather any user data for service call
    - Make service call
        - On success:
            - Stop spinner
            - Show updated results
            - Update form fields
        - On failure
            - Stop spinner
            - Show error message

That's a LOT!

Doing a lot makes it hard to

- Skim code to find the part you are looking for
- Know what all the effects are
- Make changes with confidence of results

Breaking into functions

Breaking the code into smaller functions

- Helps readability
- Doesn't inherently reduce complexity!

Many functions with "side-effects" is BAD!

- Even harder to find coder
- Even harder to make changes with confidence

Separation of Concerns

- We are using Model View Controller (MVC)
    - One form of Separation of Concerns (SOC)
- Goal is to reduce "side-effects"
    - All changes related to one purpose
    - Without impacting different purposes
    - "Side effects" is hard to define
- "Decoupled" code
    - Can change one part of code without requiring changes in other parts
    - One part "does not know" about other part

SOC is why MVC is valuable in service calls

- We have state
- We have functions to update state
- Event listener calls functions to update state
- Render() shows HTML corresponding to state
- Event listener doesn't "know" the rendered HTML
- Render() doesn't know what action just happened

Service Call Wrapper Functions

We made service call wrappers

- Not part of MVC
- Are Separation of Concerns (SOC)

Wrapper function

- Passed any needed data
- Calls fetch()
- Returns promise that resolves with parsed data
    - Or rejects with formatted error object

Service Call Wrapper Function is Decoupled

- Doesn't read HTML
- Doesn't change HTML
- Doesn't read/change state
- Doesn't "know" what event just happened
- Is reusable in other parts of app
- Handling of resolved/rejected object
    - Left to calling function

This is an example of Separation of Concerns

- Decoupled code
- Separates all fetch()/http code

Code "Responsibility"

- Consider "the responsibility" of some code
    - Don't change values outside responsibility
    - Pass in needed values outside responsibility
- Ex: functions that fetch and transform results
    - return promise of results or error
    - Separate concerns of "getting data" and "displaying data"
- Ex: structured Errors
    - Separate concerns of "deciding error" and "handling error"

Code that "works" isn't enough

Quality code allows for

- Ease of understanding
- Ease of finding the correct part of code
- Ease of making change
- Confidence that change won't impact other code
- Confidence that you know all needed change(s)

Poor quality code gets worse over time

- Changes exponentially get more complex

"It's working, I'm afraid to touch it"

- Common when you are starting
- REMAINS common if poor quality code!
- Coding shouldn't be guessing

As you start, it will ABSOLUTELY feel that way

- That's fine and normal
- It should not STAY that way

Client - Server Synchronization

- The server is the source of truth
- State on the client
    - potentially out of date
- Double state-changing actions
    - Double-click on button?

Why does it matter?

- Deleting an element already gone?
    - VERY BAD if using array index!
        - Delete unintended element
- Increment/Decrement too much?
- Pay $ based on inaccurate total?
    - Pay twice?
- Overwrite values on server?

Options to deal with data desync?

"Correct" answer depends on app

- Just trust the client
    - Make user responsible for knowing
- Keep a hash/timestamp of state
    - When client sends wrong value
        - Server refuses certain actions

Updating Client After Action

When do you update client state

- To be more likely in-sync with server?

Important to consider after sending a change

- Updates mean loading time!

Example:

- Load list from server
- Tell server to delete an item?
- Do you delete the item in the local state?
- Do you reload the list from server?

The "Back" Button

A SPA has issues with the browser "Back" button

- SPA is a single changing page
- "Back" completely leaves that page
- "Forward" reloads the page
- State before you hit "Back" is lost

This is a notable problem!

- But we will ignore until React
    - Solutions don't require React
- "Deeplinking" will be the solution

Long Results

Too much data to show user at once

- Option 1: Pagination of Results
- Option 2: Infinite Scroll

Results Pagination

- Like with service, but visible on screen
- Specific page numbers
    - And/Or "Previous"/"Next"
- Changes shown data
    - May or may not have to RETRIEVE data
    - May have more data than shown
        - Ex: Already have prev/next pages data
        - Show immediately on change
            - Start loading NEW data
            - Creates impression of speed

Infinite Scroll

- As user approaches end of displayed content
    - (scrolling)
    - Load additional data
    - Append to HTML
- May need to "remove" HTML from top
    - Otherwise it gets slow
- Accessibility problems
- Hard to save/link where you are at
- Best to use only on temporary data

Polling for Updates

The web request/response cycle:

- means the client has to ASK for an update
- ...even if there isn't one yet

This can feel (and be) inefficient

- But is also very common
- We'll do periodic polling because it's simple
- ...not because it is better

Polling methods

- Periodic Polling ("Basic", "Regular")
    - Periodic web requests
- "Long Polling"
    - Server keeps res open
    - Server finishes res once there is an update
    - Client immediately opens new request
        - On success or error
- Websockets
    - Not HTTP
    - A different protocol started *from* HTTP
    - Allows server "push" actions

Long Polling

- Client makes request
    - "Give me updates" vs "Are there updates"
- Server does NOT respond right away
    - Once it has an update, will respond
- Client auto times out connection (2-5 mins?)
    - Client will try again
- When Client gets response with data
    - Use data
    - Make new long polling request

All Client request behavior is in JS code

- Not automatic

Websockets

- Not HTTP: `ws://`, `wss://`
    - JS code requests a WebSocket connection
    - From server, with `ws` url
- Longer-lasting connection
- Not request/response
- Allows server to send unprompted messages
    - Once connection exists

Regular Polling

All I expect for this course

Pros

- Easy to implement
- Easy to understand

Cons

- Generates a lot of requests

`setTimeout()`

```
const timeoutId = setTimeout(callback, milliseconds);
```

- Calls passed **callback**
- No sooner than **milliseconds** from now
    - But could take longer (event queue!)
- Returns a `timeoutId`
    - Used by `clearTimeout(timeoutId)` to cancel

`setInterval()`

```
const intervalId = setInterval(callback, msecs);
```

- Very similar to `setTimeout()`
    - Call callsback
- But repeats *every* msecs (ish)
    - Not just once
- canceled by `clearInterval(intervalId);`

Implementing Regular Polling

- use `setInterval()` or `setTimeout()`
  - `setTimeout()` must schedule next run
  - `setInterval()` automatically schedules
    - Watch out for results taking too long
      - Coming back in wrong order
- **callback** issues a `fetch()`
  - and sets callbacks for `.then()`

Regular Polling Example

```
function refreshCats() {
  fetchCats()
  .then( cats => {
    state.names = cats;
    render();
  })
  // omitted error reporting for space
}

function pollCats() {
  refreshCats(); // fetch and use data
  setTimeout( pollCats, 2000 );
}
```

CORS on Client-side

Request a Cross-Origin Service is easy

- Just use URL with a different origin
- Browser enforces CORS
- Server is responsible for including CORS headers

When you get a CORS error

Use DevTools-Network

- Confirm URL is correct
- Confirm Server sent a success
- Check Response has CORS headers
- Fix Request or Server

Don't waste time trying to "turn CORS off"

- Don't use "no-cors"

Cookies not sent by default when Cross-Origin

- `fetch()` must use `credentials: include` option
  - Will silently not send cookies otherwise
  - Only client-side control we have for this
- Cookie NOT created with `SameSite=Strict` option
  - Default is `Lax`, which does work
- Service must send `access-control-allow-credentials: true` header
  - Otherwise browser will give CORS error when credentials sent