

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

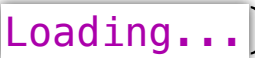


- Add to page before starting a (long?) async action
- Remove when complete
- If something breaks and you don't remove it
- ...it keeps spinning
- ...does NOT indicate anything is "thinking".
- It is just an animated image

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: )
- Can be pure CSS

All cases: HTML changed to show

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 (Chaos!)
- 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 INF06250 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 based on current state
- Event listener doesn't "know" the rendered HTML
- `render()` doesn't know action just happened

Not "knowing" = **decoupled**

- Can **find** relevant code easily
- Can **make changes** without breaking assumptions

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
 - Separates "Getting data" from "Displaying"
- Ex: Structured Errors
 - Separates "Noticing error" and "Displaying"

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
- Coders lack confidence - hoping and guessing

"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
 - OVERconfidence isn't real confidence
- It should not STAY that way
 - At least not when working with quality code

Client - Server Synchronization

- The server is the source of truth
- State on the client
 - Potentially out of date
 - As soon as you get it!
- Double state-changing actions
 - Ex: Double-click on button?
- Multiple clients interacting with same data
 - Clients unaware of changes after their load

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

- Option: Trust the user to manage
 - Make user responsible for knowing
 - Common if user editing their own data
 - Common if new data not based on old data
 - "add 1" vs "set to 9"
- Option: Keep a hash/timestamp of state
 - When client sends wrong/old value
 - Server refuses certain actions
 - Client must refresh data and try again

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?
 - OR Do you reload the list from server?

The "Back" Button w/SPA

A SPA has issues with the browser "Back" button

- SPA is a single changing page
- "Back" completely leaves that page
- "Forward" may reload the page as new
- 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: Load More
- Option 3: Infinite Scroll

Results Pagination

- Like service pagination, 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

Load More Results

- User must take action to load more data
- Adds to data shown on screen
- Good: No unnecessary data loading
- Bad: More data in client vs client pagination
- Bad: Potential slower experience
 - Must wait for new data to load

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

So what is Best Way to show user a lot of data?

- A source of much debate
- Conflicting "Benefits"
 - Loading time
 - Bytes of data set
 - Client Memory consumed
 - Reliance on Javascript
 - Accessibility
 - Option to return to this state
- There can be no single "best"

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 common
- We'll do periodic polling to learn the concepts
 - ...not because it is better

Polling methods

- Periodic Polling ("Basic", "Regular")
 - Periodic web requests
- "Long Polling"
 - Server keeps response open
 - Finishes response 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" not "Are there updates"
- Server does NOT complete response right away
 - Once it has an update, will finish response
- Client auto times out connection (0.5-5 min?)
 - 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
 - With multiple back-forth messages
- 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, msec);
```

- Very similar to `setTimeout()`
 - Call callback
- But repeats *every* msec (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() {  
  return fetchCats()  
    .then( cats => {  
      state.names = cats;  
      render();  
    })  
  // omitted error reporting for space  
}  
  
function pollCats() {  
  refreshCats().then( () => { // fetch and use data  
    setTimeout( pollCats, 2000 ); // AFTER previous call  
  });  
}
```

CORS on Client-side

Request a Cross-Origin Service is easy

- Just use URL with a different origin
- Browser enforces CORS
 - Client can't do much to fix!
 - Common source of confusion/frustration
- 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 silently not sent Cross-Origin by default

- `fetch()` must use `credentials: include` option
 - Same origin defaults to included credentials
 - Only client-side control we have for this
- Service must send CORS header
 - `Access-Control-Allow-Credentials: "true"`
- Or browser CORS error when sent credentials

Client Can't Change CORS

If Client sends correct request

- And includes any required cookies/Auth

Then any CORS issues are up to server-side to fix

- Assuming server-side WANTS to allow CORS
 - Often they block it for valid reasons
 - Such as client sending a private key/token
 - Nothing in front end is private from user!

This Course will not use CORS

- Our sites and services are on Same-Origin
 - Not Cross-Origin
- I provide info on CORS for future reference
 - Only the core concepts
- Do NOT write your services here to use CORS
 - Follow the lessons I teach
 - Not random code from Google
 - Bad habit to open security unintentionally!
- We will revisit CORS in next unit after Project 2