What to do after INFO6250?

- You have been web developers for weeks
- Always more to learn
- Easy to forget details
- Practice!
    - Github repos of original code
        - more value to you and to them
- Expand
- Tap into info sources

Practice

Best way to improve as a coder is to code

- Know "right" through experience
- Get experience by doing it "wrong"
- Plus debugging skills

Make a public repo

- Contribute over time
- **https://blog.patricktriest.com/host-webapps-free/**

Code changes over time

- Teach you more
    - "It depends"...on what?
    - Revisiting old code
        - Reveals poor communication
- Give you an interview topic
    - Not just "how", but "why"
- Show value to employers
    - Primary skill not taught in schools
        - Because "over time"
        - Did my best with revisit

This was just one course

- We've covered a LOT of material
  - Industry is so much larger
  - Expect to never stop learning
  - But remember to enjoy!
- Here are topics as options
  - Some covered not well enough
  - Some not covered at all
  - You WILL NOT learn them all
  - Treat as a menu, not demands

Not Doing Web Dev?

Valid! "Knowing" web dev is useful anyway!

- Dev, but Non-Web
- Project/Product Managers
- Management/Leadership
- Any other skill expertise

There are still some lessons to take from course

Lessons for non-web developers

- Debugging
  - Not Magical!
  - A methodical process
  - Find *where* it goes wrong
- Communication!
  - It's about people, not processors
    - Even if we don't like other people
- Code for Change
  - Change is the real problem we solve for
  - "Make it work" isn't the real ask

More General Lessons

- Abstract to manage complexity
    - DRY
    - Focus on "contract" (input/output)
- Abstraction has a cost!
    - YAGNI, AHA
- Minimize coupling to allow easy changes
- Data Model (state) drives everything
    - State decides presentation
    - Events change state
- Data Model allows abstraction of concepts

Unit Testing - Wish I could squeeze in!

- But isn't small topic
  - Particularly with Web UI
- Will be used everywhere
- **Tests offer confidence about change**

You should start writing tests!

- Server-side logic easier
- Focus on "pure" logic, not integrated
    - Where you test input vs output
    - Functions
- Jest or Mocha+Chai are most common
- Components are functions too!
    - See example in create-react-app

What are you testing?

You want confidence about change

- Avoid "brittle" tests
    - Don't match exact output (for HTML)
    - Match impact of input on output

Unit tests are NOT integration

- Test the "units" in isolation
- Does this test add confidence?

Lots of Coverage Debate

What percentage of code lines are tested?

- Much debate on target: 80%? 100%?
- Gets weird with 3rd party code
- Focus on the confidence
    - Coverage will follow

Testing is Communication

- Tests document the expected behavior
- **Write tests to communicate**
    - Strings in test code matter!
    - BAD: `it('input', () => {...});`
    - GOOD: `it('validates input', () => {...});`
- Balance abstraction in tests
    - Keep tests easy to understand and change
    - Keep tests as communicative
    - Delicate balance
        - Needs Practice!

There ARE integration tests

Tests combining units

- Slower, more brittle
- But closer to testing real product
- Don't repeat tests from units
    - Already tested!
- Test the assumptions when units integrated
- Test the flow of output to input
- Some argue we should do mostly integration
    - Common concept is to do mostly unit
    - **https://kentcdodds.com/blog/write-tests**

Test Driven Development (TDD)

Practice of writing tests, then code

- Red-Green-Refactor cycle
- Easier when covering a mostly-known interface
- Harder when doing exploratory code
- Builds good habits
    - Create code that is easily testable
- Takes time to adjust to
    - You will code slower for ~6 months
    - Very rough estimate!
    - Then you are as fast or faster!
- Generates a lot of confidence

Writing Tests After Writing Code

- Fairly common practice
  - Write code until it mostly works
  - Then write tests
    - Check for all cases
    - Detect problems in the future
- Can have problems if your code is "hard to test"
  - Ex: uses hidden/private/internal methods
  - Ex: needs state only set by series of steps
- Allow time for this when estimating!
  - Most places: code not done until tested

Accounts

- We did simplified "create account on login"
    - I don't want to encourage poor security
    - Takes time
    - You have the theoretical skills
        - Build a user registration system
            - Good for final!
- But theory isn't practice
- Still avoid managing passwords - use experts!
- If you must manage passwords, remember
    - Proper, modern, hash+salting
    - Never store passwords

Tokens other than cookies

- We covered session id cookies
- Didn't use other tokens
    - JWT, API keys, etc
- Can be sent as header in fetch()
    - NOT automatic like cookies are
    - You know how to set headers
    - Theory isn't practice
- Often hidden from client JS
    - Cookie with "Secure" flag
    - Not SPA friendly

OAuth is an entire thing

- Redirect user to 3rd-party site for login
- They redirect back to your site with a code
    - You send code to 3rd-party to get the token
- You never see the password

Many libraries or do it directly

- Every OAuth Provider is a little different
- Great way to avoid user password management
- But work to set up

Websockets for polling for changes

- You can do a lot without websockets
    - I've never written one outside of test code
    - b/c I haven't written apps that poll a lot
- But they are great for performant polling code
    - Vital if your app polls frequently

GraphQL

- Growing in popularity
- Simplifies backend endpoint design
    - Endpoints for data categories
    - Not endpoints for data interactions
- Increases network design concerns
    - Caching, versioning, etc
    - REST used HTTP conventions
- Often involves libraries/frameworks
    - Both frontend/backend

We only skimmed Input Validation

- Deeply complex topic
- Done wrong more often than right
- Initial solutions often recreated (poorly)
    - email addresses
    - phone numbers
    - names
- Timing of validation
    - as entered?
    - field loses focus?
    - form submit attempted?

Git deserves more attention

- `git` is a core tool
- It should be more than "type these commands"
    - On problem, wipe repo/branch and recreate
    - Instead, should understand, choose fix
- Very frustrating to learn while solving a problem
    - Learn in depth when NOT having a problem
- `git` is history of code!
    - Allows backing up and trying again
    - Allows multiple "what if" experiments
    - Allows you to find trends in errors
- You should be a master of your core tools

We only covered CORS from front end

- CORS rules are defined by server response
    - Enforced by browser
- Many headers to control rules
    - Server has to emit them
- You can set headers on server
- You can define OPTIONS route
- But we didn't dive into the choices
- Commonly poorly understood!
    - Backend doesn't need it
    - Frontend can't change it

Javascript

- We covered a lot of JS
    - Needed for frontend!
- But this isn't a Javascript course
- May need more for interviews
- Will want more if JS is primary language

JS Topics to explore

- JS Prototypes
    - All inheritance is through prototypes
- JS Classes
    - Not fundamental like some languages
    - Common in some places + circumstances
- Closures
- `await/async`
    - Different syntax for promise management
    - Very, very popular!
    - Easier to learn well if you know Promises well
        - But still a new syntax

We didn't touch TypeScript

MANY Employers are shifting to TypeScript

- And individual fans like it anyway
- Build-time check of type safety
    - Increases confidence of change
    - Improves hinting from IDE
- Expansion of JS syntax
    - Requires you know JS
- Additional build step to do/configure

Truly understanding JS

To learn more about how JS really works

- "You Don't Know JS Yet" by @getify
- **https://github.com/getify/You-Dont-Know-JS**

Automated e2e(end-to-end) testing

Mosts tests are unit or integration

- e2e testing is slow, brittle
- But e2e testing has a place

Different toolsets

- selenium, webdriverio, cypress, puppeteer
- They run browser
    - You talk to them, not to browser code
        - Mostly

We only developed and tested in Chrome

- Most popular browser
- We stuck to well behaved standards

I lived through the Browser Wars

- (long stare into the distance)
- We don't want that again
    - But it's becoming a real risk
- You should validate against all major browsers
- Firefox is very popular among devs for DevTools

We only used Webpack as a bundler

...and we didn't LEARN it

- Webpack has many options and plugins
  - Tree shaking
  - Code splitting + lazy loading
- Other bundlers exist
  - Rollup, Snowpack, Parcel

We haven't discussed copyright + licenses

Being on the web does NOT let you use it for free!

- Copyright is a legal protection
  - Details vary by country
  - In US: Automatic protection on creation
- Need permission to use
  - Even in private
- Covers "derived works" as well
  - Works based on other works
- "I'm not charging money"
  - Not a defense!

A "license" is a permission to use

- Different licenses have different rules
  - How to give credit
  - Can you change/derive?
  - What license can YOU use on derived work?
  - Can you sell?
- Some common licenses
  - GPL (v2 + v3)
  - MIT
  - BSD (a few)
  - Apache
  - Creative Commons licenses (multiple)

You can "get away with" ignoring for a while

...but you risk a really painful experience

- You might be taking someone's livelyhood
- Or annoying a corp looking to make an example
- US Law allows for triple damages
    - "damage" is not limited by your profit!
- Employers really frown on problems
    - Costly, Embarassing, Frightening
- Can be very bad socially
- Better to learn, understand, and follow

Other "Intellectual Property"(IP) Concepts

- Patents are a separate concept
    - Govern processes, not works
    - Turbulent recent history in tech
    - Companies LOVE getting patents
        - May push their devs to get some
- Trademarks are a separate concept
    - Govern identifying mark, name, look, etc
    - Don't copy the look of a site too much!
    - Avoid names that are "confusingly" similar
    - This is where enforcement is required
        - Not all "IP"

We didn't discuss i18n and l10n

- i18n is "internationalization"
    - Because 18 letters between "i" and "n"
    - Programmers are really lazy :)
    - Providing multiple languages
        - Including orientation
- l10n is "localization"
    - Managing different dates/numbers
- Surprisingly complex!
    - Different pluralizations
    - Space for text
- Automatic translation is still garbage

a11y is "accessibility"

- Usable with disabilities
    - Color blindness
    - Blindness
    - Poor fine motor control
    - Temp or lasting injury
    - Holding a baby/pet
    - Using phone while on bouncing bus
- Very important field
- Often neglected field
    - Opportunities!

a11y has a lot of controls available in HTML/JS

- And a lot of ways to ruin the experience
    - Particularly with JS
- Devs need to learn how to do well
    - And what not to do
- Most general tutorials ignore this

We didn't discuss Deploying

How to put the code on a "real" website?

- Need a "host"
- Static file hosting is easier/cheaper
    - Github pages
    - Amazon S3
- Server code is a bit more involved
    - Vercel
    - Render.com
    - Netlify
    - Amazon Amplify

Our examples for class are harder to deploy

Most free hosts don't have one or more of

- Stable memory (they shutdown and restart often)
- Filesystem access
- Ability to deploy server code

To show off work like ours, I recommend one of:

- Using heroku or render and accept frequent reset
- Use remote database for persistence (change, $?)
- Use filesystem to persist data (change, $)

Separating the parts

- We combined our static server and services server
- This impacts deployment!
    - Servers w/CRA hosting only cover static side
        - But will build for you
    - Anything covering services covers static
        - But you need to arrange built files
            - Commit or ensure build step
- Consider if you want to separate to deploy

We didn't use HTTPS

Using HTTPS on a host is usually easy

- They usually manage the certificate
- Code doesn't change
- But configuring the DNS is daunting
    - Also have to wait for changes
    - "TTL" = "Time To Live"
    - Frustrating if you make mistake

A workplace may have CI System

CI = "Continuous Integeration"

- Means "Runs tests on merged code"
    - Before actual merge is allowed
- Many systems
    - Jenkins
    - Travis CI
    - Github Actions
- CI/CD is a step further
    - CD = "Continuous Deployment"
    - If tests pass and merge happens
        - So does deployment!

We didn't discuss Documentation

- Most devs want to "code", not "document"
- Programming is Communication
    - Poor documentation is poor programming
- Many docs based on audience
    - New devs to team? (Onboarding)
    - Users of the product (Product Docs)
    - Coders using the code (Functional Reference)
- Documentation is often neglected
    - This has a cost!
- Out of date documentation is worse than none!
    - Not an excuse not to document

Documentation is one form of "Tech Debt"

Tech Debt is resistance to change of code

- Updates that need to be made
- Fixing bugs
- Refactoring code
- Updating libraries
- Writing/Updating Docs
- Missing/failing tests

Tech Debt CANNOT be avoided

- But you CAN minimize

Tech Debt accrues interest

Over time, tech debt accrues "interest"

- Additional work necessary to stay where you are
    - Even MORE work necessary to improve
- Don't do the work, get MORE tech debt

Projects absolutely die from tech debt

- Rewrites are often not successful
- Why libraries aren't always a win

We haven't discussed Performance

Core questions:

- What are you optimizing for?
    - Projects often optimize for developer time!
- What is the major *real* slow down?
    - Never trust your instincts - measure!
    - Easy to waste time/make things worse!

Optimizing for Dev Time

No one WANTS to have sub-optimal performance

- But is a detail worth it?
- Developer time is very limited!
    - Much to code
    - Few (and expensive) devs
- Is the optimal code harder to change?

Is this performance a problem?

Benchmarking lets you know

- If something is actually slow
- If it is the slow that matters
- If a change is actually beneficial

> *Premature optimization is the root of all evil*
> *-- Donald Knuth*

- Your guess of inefficiency will often be wrong
- Your guess of what impacts real world time also

Don't act on guesses, measure with a benchmark

What to Benchmark on a web page/app

- TTFB (Time to First Byte)
- FCP (First Contentful Paint)
    - User is able to see useful data on the page
- TTI (Time to Interactive)
    - When the user can take meaningful action
- Load time - When all assets are loaded

Each is impacted by different details

- Network, server, frontend code, images, etc
- Make sure you're trying to solve the right problem

Most common performance problem is not code

Images are the most common source of slow loading

- Designers will often create unoptimized images
- Developers often just use any images they get
- You should size to actual size they will be used at
- You should optimize to balance filesize/quality
    - This is an entire skill

Where is the code issue?

One place I worked: ~15ms TTFB on service calls Another place: ~200ms TTFB on service calls

If we wanted a <1s TTI, that adds up!

- Services were much faster (~10ms)
- A network issue
    - No amount of coding would fix

Even as code issue, don't trust your instincts

Benchmarking shows where time is really spent

Consider:

- `function1()` called once taking 10ms
- `function2()` called 1000 times at 2ms each time
    - 2000ms total

Better to improve `function2`!

- Despite `function2()` being faster by itself

Profiling the Page Load

- Chrome Devtools has
    - Performance Insights
    - Performance
        - Generally useless with React code, as it benchmarks React itself

Use these to benchmark the page itself

- For non-code assets
- Or vanilla JS code

Profiling React

React DevTools extension

- Adds "Profiler" for React pages
- Can get a flame graph of components
- Many options

Options to improve React performance

- "window" large arrays of data
  - Only render visible or near-visible rows
  - Reduces rerendering of data user can't see
- Prevent unnecessary rerendering
  - `useMemo()` and `useCallback()`
  - Don't use if not needed!
  - **https://kentcdodds.com/blog/usememo-and-usecallback**
- Memoize pure components!
  - `React.memo()` (not same as `useMemo()`)

Service calls are often a source of delay

- Avoid repeated/unnecessary calls
- Perform in background after load
- Cache results
  - `react-query`?

Other React Hooks

- `useRef()`
- `useMemo()`
- `useCallback()`
- Several others at
    - **https://react.dev/reference/react**

`useRef()`

- Stores a value like `useState()`
- Doesn't trigger rerender when value changes
- Often used to hold reference to an element
- We skipped to avoid overuse

`useMemo()`

- "memoizes" a value
    - Based on a function and other variables
    - Recalcs when other values change
- Used to avoid repeated expensive calculations
- We skipped to avoid overuse

`useCallback()`

- "memoizes" a callback
    - Based on variables
    - Recalcs when variables change
- Used to avoid unneeded rerendering
    - Functions defined in a component are NEW
        - Even if the same
        - Reads as new prop, triggers rerender
- We skipped to avoid overuse

We avoided even common React-compatible libs

We avoided to learn core concepts

- Redux
- `react-router` / `@tanstack/router`
- `react-query`
    - Used to manage service data
    - Highly recommended
- Many, many conditional libraries
    - Forms and validations
    - Page flows
    - Service interactions
    - UI features

We avoided component-based styling

We used classical semantic CSS

- Many newer concepts exist
    - Ex: CSS Modules
    - Ex: styled-components
    - Ex: CSS-in-JS (losing favor?)
    - Ex: Tailwind, utility CSS
    - We avoided these to focus
        - Not because these are bad
    - Many opinions, few the same
        - Still being evaluated/argued over

Progressive Web Apps (PWA)

- Can be installed like an app
    - May look more "native"
- Can operate with cached data
    - Work "offline"
- Browser/OS support varies
- CRA has some built in support!
    - Managing the offline aspect is intricate

Electron allows actual apps

Electron wraps standalone Chrome with your app

- Slack is an Electon App
- Not great for performance
- Great for webapp/mobile/desktop

Simple Client Side SPAs are not only option

Exciting developments in moving React to server-side

- Ex: Next.js, Remix

Better performance for client

- Static Site Generation (SSG)
    - Build HTML/CSS
    - No client side JS to render
- Server Side Rendering (SSR)
    - Does bulk of rendering on server
    - Sends result to client-side JS

CRA not only option

Example:

- `create-next-app` for Next.js
- `create-wmr` for WMR and Preact (React alternative)
- Vite for React or Vue

React is not your only option

- Lots of server-side templating languages
- Preact (React syntax)
- Vue
- Svelte
- Angular

Web Components can be built in multiple languages

JS is not your only language option

- Front end still mostly just JS
    - A few "built" languages
        - Clojurescript
    - WebAssembly not for DOM interaction yet
- Non-JS Backend languages exist :)
    - Will have libraries/frameworks
    - Java, .NET
    - Python, Ruby, PHP
    - Go, Rust,
    - Perl, C++, C

Front End Devs may need to run Backend

- Common to have to run "local webserver"
- Not coding the backend language
    - But have to run a server with it
    - Frequently "update" the local server
        - Pulling in changes from backend
    - And perhaps occasionally debug

Databases

Backend may interact with different databases

- SQL vs NoSQL
    - ACID discussions
    - Backups, failover
    - Lots of SQL/NoSQL overlap nowadays
- Details can trip you up!
    - inner join vs outer join
- Huge performance impacts
- Data security, Data reliability

InfoSec is an entire field

Vast Persistence and Variety in flaws

Quality Assurance is an entire field

More common for QA to become dev

- But Devs can move to QA
- A certain style of thinking
- Satisfaction counts for a lot
- A QA/Test Eng is a master of domain knowledge
    - Wide skill net

Information Sources

Web Tech changes rapidly

- How do you learn about new options?
- How do you learn new best practices?

My experience: You have to CHASE this

- Easy to get bubbled into what you do now
- New devs generate a lot of intro "noise"
    - Great for newbie perspective
    - Often contradictory
    - Sometimes bad advice

Finding Good Sources

- Sometimes good people get celebrity
- Other times you have to find them
    - Both tech and search engines have biases!
    - Keep your eyes open
- When you find someone good
    - How will you know their next bit of info?
    - Subscribe, follow, whatever
- "Curate" your sources often
    - Sources will get stale over time

Twitter! Oh no!

Previous best source was Twitter

- Now in meltdown/exodus
- See where people go

Samples from my list:

```
@geekgalgroks  @cassidoo
@zeldman       @laurieontech
@jaffathecake  @JoshWComeau
@rmurphey      @b0rk
@meyerweb      @TerribleMia
@jen4web       @darkpatterns
@RachelTobac   @Mappletons
@whitep4nth3r  @shehackspurple
```

Blogs and RSS

Longform content is still important

I use RSS to keep track of blogs

- **https://overreacted.io/**
- **https://www.joshwcomeau.com/**
- **https://css-tricks.com/**
- **https://adrianroselli.com/**

Videos, Podcasts, and Streams

I'm a sucker for text, but others prefer other formats

- Same principles: Filter, Follow, and Curate

Samples:

- HTTP 203 Podcast
- Ladybug Podcast (paused, but good backlog)
- **https://www.youtube.com/kevinpowell**
- **https://youtu.be/wPcv9Rp4lHk**

Conferences

- A few days dedicated a common topic
    - Great sources of info and inspiration
    - So many topics!
    - So good for extroverts!
        - Networking and friends
- Many conferences stream their talks
- Many conferences later post them online for free
    - Even for those that didn't attend in any way
    - So good for introverts!

Bumps Ahead

- Imposter Syndrome
- Gatekeeping
- JS Hate
- Burnout

Imposter Syndrome

- Too much to know
- The more you know
    - You recognize your ignorance more
- Cocky, overconfident people exist
- Real experts in an area exist

Easy to feel like you aren't worthy of respect given

- "when they find out, my career is over"

Gatekeeping

Knowledge and expertise is a social currency

- Some max their value by putting others down
- Some improve the collective wealth
- "JS/HTML/CSS isn't a REAL language"
- "Web dev is trash programming"
- "You don't know X?"
- "You aren't cut out for programming"

Gatekeeping is about shutting out, not being right

JS Hate

Not as prevalent as it once was

- Still very present
- Always a target

Margaret Hamilton saved the Apollo 11 landing

- When "software" was considered trash
    - not even science
- Part of group that named "software engineering"

"Haters" don't improve things

Burnout

- Not talking about "need a week off"
- Traumatic mental experience
- Diminishes capability to do things
    - Can induce panic
    - Pushing through makes it worse

Caring more makes you more susceptible

Be Nice

It can be nasty out there

- Particularly for non-cis men, non-whites
    - But no one is totally safe
- Protect yourself
- Never think it is deserved
- Don't add to it
    - But still stand up for yourself
        - And for others
            - Esp if you're not marginalized