2025 Theme: Branch Out

Notes taken by Catherine Tu, C.O. '28,

Intro

Website Basics:

- Accessing a website: Client (you) send a request to the server, server stores & serves, responds with webpage files, user can view the website
- HTML: actual website, CSS: style, JS: assets
- We'll be building a website from ground up with profile, game,

Milestones Due Dates:

- 0: Ideation — 1.8
- 1: Project Pitch + Feedback — 1.11
- 2: Minimum Viable Product (MVP) — 1.22
- 3: Final Website — 1.29

Judging Criteria:

- Functionality (technical components of core features — they'll play around with website)
- Usability (ease of use)
- Aesthetics
- Concept execution (applicability of solution to the problem)

Must Build:

- Dynamic website supported by back-end
- Personalized experience based on user accounts
- Minimum security requirements fulfilled
- Original design + implementation
- Use Git on web.lab GitHub repo
- Resizeable to different devices??

Cannot:

- Use website building softwares (Drupal, Wordpress, Squarespace)
- Use any part of previous project
- Outsource

Learning Resources:

- Piazza
- The course website weblab.mit.edu
- Resources compilation weblab.mit.edu/info
- Office hours (7-9 pm typically)
- Hackathon during week 2 + 3

Git

Git Cheatsheet: https://education.github.com/git-cheat-sheet-education.pdf
Problems that Git solves:

- Need independent local copies of codebase
- Need to be able to merge different people's changes together
- Need to keep track of versions
- Need to know which version is most up to date

GitHub is a giant remote server to push code on & collaborate

In **VSCode**:

- Terminal → New Terminal (way to interact with computer)
- ls → list out everything in the folder / directory
- cd <dir> → goes inside another directory
- cd .. → goes backwards
- mkdir <new dir> → makes a directory
- git init → turns current directory into git repo
- . = current directory
- .. = parent directory

Staging & Commit:

- git status (tells you what's happening; red = hasn't been added)
- git diff (shows changes between working copy and staged / committed copy)
- git add <filepath>
    - Adds filepath from working dir to staging area
- git commit -m '<message>'
- git log (shows the different commits in history)
    - Different ids for each commit
- git push

Branching:

- git checkout (switches from one branch to another)
- git branch (looks at current branches)
- git checkout -b <name> (creates a new branch)

Merge:

- Switch to main dir

- git merge <branch> (merges that branch into the dir you're on)

Cloning GitHub:

- git clone <git website url>
- git fetch (looks & updates whatever is on your computer with what's on GitHub)
    - Different from git pull bc does not automatically merge — more control
- git pull (update & merges local copy with yours)
- git push

Resetting:

- git reset --hard (wipes everything clean from local copy; takes version from main branch)
    - Resets the LOCAL version of your code on your computer to MATCH the last committed version on the main branch
    - Irreversibly deletes all local (uncommited) changes
    - Don't abuse this though!
- git checkout wX-stepY
    - Retrieves the branch "wX-stepY" from the github cloud

Intro To HTML / CSS

HTML:

- Hypertext Markup Language
- The skeleton, CSS is the aesthetics
- HTML = nested boxes (simple!)
    a. Box for the website, sub-boxes with info
- Have an opening and closing tab, with content in between
- Make sure it's nested properly
- **Don't just use too much**
    a. Semantics are better

Tags:

- groups block section tag of doc (line break)
- groups an inline section of a doc (select one thing)
- <html>
- <h1>
- <p>
-
- <section>

- <hr> (adds horizontal line)
- <tagname abc = 'xyz'> (adding attributes)
- ordered list
    a. There's also unordered lists, etc.
- <nav> (nav bar title changer)

Common Attributes:

- <a href = 'link'> href tells it what to look at
-
    - Self closing tab — similar to <img scr = 'img link'></img>
-
-
    -
    - Info
    - Ids must be unique in any given HTML doc

```
<!DOCTYPE html> heading — always include in website
<html> first element (opening tag)
    <head>
        <title> Title! </title> On the website tab
        <link rel = "stylesheet" Joins HTML and CSS
            href = "style.css" /> Can have multiple style sheets
    </head>
    <body>
        <div>
            <h1> Heading! </h1>
            <p> Paragraph! </p>
        </div>
    </body>
<html> first element (opening tag)
```

CSS:

- Cascading Style Sheets
- Go to website → developer tools → delete style sheets to see HTML only
- Adds aesthetics to HTML
- **Hierarchy**:
    a. Inline style
    b. ID Attributes #unique {...}

c. Classes .info {...}
d. Elements div {...}
e. ONLY USE FOR CSS STYLING!

```
div { selector
    color: red; property
    font-family: Arial;
    font-size: 24pt;
}

.info { selects a class called info
    color: red;
    font-family: Arial;
    font-size: 24pt;
}

.id { selects only things tagged with an id
    color: red;
    font-family: Arial;
    font-size: 24pt;
}
```

Also: :root

Margin:

- Has ordering
- Some CSS attributes can take multiple values, like **padding** and **margin**
  - Padding typically multiples of 8
  - padding: 8px 16px;

Workshop 0 Notes:

- Can just drag an html file into chrome new tab and preview what it looks like
- Good practice to separate into different sections <section>
- Utility classes denoted by .u- (classes with only one function)
- For h: The bigger the number, the smaller the text
- Fonts: use fonts.google.com → get embed code → @import → copy what's within the <style> tags → put into the styles css
- Use MDN Web Docs to determine which tag is best
- Right click + inspect to look at the void — can go to Computed to see margins
- Exercise: make buka buka perfectly round

- Flexbox: https://css-tricks.com/snippets/css/a-guide-to-flexbox/
    - Flexible box that lets you control direction, sizing, distribution, etc
- Flexbox learning: https://flexboxfroggy.com/
- Flex: http://www.flexboxdefense.com/
- Simple JS: https://www.jschallenger.com/javascript-practice

1/7/25:

JS

- Programming language that manipulates the content of web page (organs)
- Makes website interactive
- Not related to Java
- All web browsers know how to run JS
    - Cmd + Option + j
- Types: boolean, number, string, null, undefined
    - No distinction between float, int, etc (everything is number)
- Operators: ===, !== (to compare)
    - 2 == 2 → True
    - 2 == '2' → True (type coercion before comparing)

Basic Syntax:

```
Const greatestCommonDiv = (a, b) => {
        while (b != 0) {
                const temp = b;
                b = a % b;
                a = temp;
        }
        return a;
}
```

Defining Constants vs. Vars:

- let myBool = true ← variable that may change later
- const myBool = true ← constant that CANNOT change later
- Use camelCase
- let = block scoped
- var = function scoped

null vs. undefined:

- let firstName; ← currently undefined

- firstName = null ← can "empty" a variable with null

Commands:

- console.log(`a * b = ${a * b}`) ← operation
- alert('congrats') ← sends out a pop up on user screen
- Arrays can pop(), push(<el>), change elements console.log(pets[3])
- While loop as long as: while (<condition>) { … }
- For loops: for (let i = 0; i < pet.length; i++) { … }
    - const phrase = 'test'
    - console.log(phrase);
- for (const animal of pets) { ← loops through items of pets
- map(...) → creates new array by applying callback function to every element
    - const newArr = myArray.map((num) => (num * 3));
- filter(...) → filters out elements
    - let posVal = values.filter(x => x > 0);
    - const valid = staff.filter((name) => name !== 'Annabel')

Objects:

- Set of keys and values (similar to a dictionary)
    - console.log(myCar.model); ← retrieves value; same!
    - console.log(myCar['color']) ← retrieves value
- Object destructuring: shorthand to obtain multiple properties at once
    - const { make, model } = myCar

Copy Array:

- Shallow copy: let copyArr = arr;
- Deep copy: let copyArr = [...arr];

Functions:

- Multiple ways to define a function, but we're sticking to one
- (parameters) => { body}
- Callback function: function that calls another function
- setTimeout() ← calls a function when timer ends
- setInterval() ← calls a function at certain intervals

```
const celsiusToFah = (tempCel) => {
    const tempF = tempCel * 9/5 + 32;
    return tempF;
```

};

Array of C to F: two ways

```
(arrayT) => {
    const arr = []
    for(let i = 0; i < arrayT.length; i++) {
        let f = arr[i] * 9/5 + 32
        arr.push(f)
    }
    return arr
}

const tempC = (arrayT) => {
    const arr = [];
    for(const t of arrayT) {
        let f = t * 9/5 + 32;
        arr.push(f);
    }
    return arr;
}
```

modifyArray function F to C:

```
const modifyArray = (arr, transformFunc) => {
      const newArr = [];
      for(let i = 0; i < arrF.length; i++) {
            newArr.push(transformFunc(arr[i]));
      }
      return newArr
}
```

Shorthand notation:

```
const cToF = (tempC) => (tempC * 9/5 + 32);
```

Why Use Callback Functions:

- Callback functions = function passed as an argument to another function
- Callback is a reference to the function that is passed in
- Reusability (map and filter)
- Abstraction (‘when x happens, do this…’)

```
router.get(‘/comment’, (req, res) => {
      Comment.find({ parent: req.query.parent }).then((comments) => {
            res.send(comments);
```

```
    });
});
```

React

- React guide - https://docs.google.com/document/d/1Y1WYwqoho7cWCRRU4iYrfcZZ2tCR6BFoXGSBhLKgysQ/edit?tab=t.0
- React is a framework that lets you divide up your website into reusable components
- Simplifies (abstraction)
- Abstraction for a bunch of HTML, JS in one file
- Call on this component and it returns part of website
- Components of Facebook: <App />
    - Has components within it (broken down) and components within that
    - <NavBar /> <InfoBar /> photos, post, etc.
    - Component Tree
- Components = functions that take in props and returns what you want to render
- The idea: once we have our components, we can write any website with "one line of code"
- Link css to jsx by doing import "./NavBar.css";

Props:

- Helps generalize comments
- Comment Props: profile picture, author name, comment content, like, reply, date poster
- Parent (post) calls props then outputs a rendered comment
- Can be updated when parent component passes in a new prop
- Props are **immutable**
- < Post name='Kenneth' text="welcome to weblab!" />
- props = {name: "Samvit", text: "walcome to weblab};

State:

- Info maintained by a component
- Lets us control what is displayed in application
- Can be updated (**mutable**) by human input or automated
- Ex: counting number of comments, adding comments
- const [status, setStatus] = useState('busy');
- const [isOnline, setIsOnline] = useState(false);
- Always set state!! Never assign

- const [value, setValue] = useState(0)
- setValue(6); vs value = 42 (BAD)

Toggle thumbs up:

```
<button
      onClick = {() => {
            setIsLiked(!isLiked);
      }}>
      {isLiked ? “Liked” : “Like”}
</button>
```

CommentReply.js (generalized function)

```
import React, { useState } from ‘react’← import react libs
const CommentReply = (props) => { ← define component
      const [isLiked, setIsLiked] = useState(false) ← react state syntax
      return (
            <div className=’comment-text’>
                  <h5>{props.name}</h5>
                  <p>{props.content}</p>
                  <p>{isLiked ? “Liked” : “Like” }</p> ← conditional state
            </div>
      )
}

export default CommentReply ← export component
```

Slide 87 - end: Facebook example with props and states (customized)

Workshop 1 Notes:

- Use className instead of class in React
- **npm install → npm run dev** ← in terminal will see live page updates then navigate to **localhost:5173**
- Add a state to component: const [something, setSomething] = useState(default val)
- Go to console: right click + inspect

Component Tree: break down into smaller parts when…

1. Component code is getting too long or hard to read
2. Component contains parts or sections (usually reusable) that have their own functionality
3. Component is handling too many responsibilities
    - Ex: a button may require a lot of logic

- There's no single way to structure React applications — you are the architect
- Pass down props that may be used in multiple components
- Component rendering works like a restaurant (trigger aka mounting, render, commit)
    - Dismounting: no longer view the component

React Hooks: https://react.dev/reference/react/hooks

- Special functions provided by React to access parts of component lifecycle
- useState, useEffect

useState: Lets us add a state variable to our component

1. import React, { useState } from 'react';
2. Create state (name) and state variable (setName)
3. setState
4. Pass state as prop

```
import React, { useState } from 'react';

const ParentComponent = () => {
  const [name, setName] = useState('Alice');
  setName('Ben');

  return (
    <div>
      <ChildComponent name={name} />
    </div>
  );
};

const ChildComponent = ({ name }) => (
  <div>
    <h2>User Details</h2>
    <p>Name: {name}</p>
  </div>
);

export default ParentComponent;
```

useEffect Hook:

- Hook that does something when an event occurs
- Takes two variables: callback function and OPTIONAL array
- Runs after specific variable changes (response to state change)
- Typically used to synchronize with something external to React
    - Load external data into state
    - Call an API/perform some computation/etc at specific times

- useEffect(function, *optional* dependency array);
- We can do a lot of work that only needs to be done once
- Guarantee that we have the most up-to-date state value if we use the state as dependency
    - Counters the set function does not update right away

Examples:
useEffect(function, [var1, var2]) ← calls func on mount (1st render) & when var1/var2 change
useEffect(function, []) ← calls function only on mount (component is rendered for first time)
useEffect(function) ← calls function at every render (first component call + every state change)

Common Patterns in React:

1. Conditional rendering
    a. In JSX: using HTML tags to render stuff
    b. Ternary statement — condition ? resultIfTrue : resultIfFalse
2. Rendering an array of data
    a. Loop rendering
    b. map()
3. Fetching and sending data from a server

Stopwatch Example:

```
const Stopwatch = () => {
      const [time, setTime] = useState(0)

      useEffect(() => {
            const timer = setInterval(() => {
                  setTime((oldTime) => oldTime + 1);
            }, 1000);
            return () => clearInterval(timer);
      }, []);

      return <>Time: {time}</>;
};
```

Another Way:

```
const Stopwatch = () => {
      const [time, setTime] = useState(0);

      use Effect(() => {
            setInterval(() => {
                  setTime(time + 1);
            }, 1000);
      }, [time]);
```

```
        return <h1>{time}</h1>
}
```

useContext Hook:

- Contexts provides a way for higher-level components in our tree to communicate to all sub-components
- Great way to share state variables from one component to ALL of its descendants
- import React, { createContext } from 'react'
- const UserContext = createContext(); ← outside our component we create a context
- **Inside component, we create a useState name & variable [user, setUser]
- <UserContext.Provider value = {{user: user}}>
  <h1{`Hello ${user}!`}</h1>
  </UserContext.Provider> ← wrap content of component in a **Provider**
  - Provider gives a JSON object value
- In another component: const user = useContext(UserContext).user;
- If **useState** is like a piece of info, **useContext** is an empty book you can write in, and **Provider** is like a library you can check out the book from
- Every subcomponent of the thing the Provider is wrapped in can use the context
- Can have multiple contexts

React Components:

- More powerful HTML element
- Javascript functions
- In old React, components were classes; now, are functions
- Components are made of JSX (similar to HTML) elements and other components
- Can nest components within each other
- All components are recursively rendered until there are no more nested components

DOM: Document Object Model

- Data representation (model) of the objects that comprise the structure and content of the document
- At the very top: document, then <html>, etc
- Can be represented as a tree
- Can use DOM to represent react components and HTML
- React creates a virtual DOM tree that represents structure of your UI

Phases of Component's Lifestyle:

1. Mounting
   a. Process of adding a component to the DOM
2. Updating (loop)
   a. Trigger causes a re-render and changes the virtual DOM

   b. Component can stay in updating phase for a long time
   c. Caused by trigger, render, commit
3. Disnounting
   a. Process or removing a component from DOM

Triggers: causes component to enter different phase; happens when…

- React is telling the component to render for the initial render
- One of its ancestors re-renders (you'll re-render if ancestor does)
  - Theres a way to avoid this (different hooks)
- A state changes in the component
- A prop passed from parent to itself changes

Rendering: create the virtual DOM at each phase

- Runs the component's function, which triggers its children to be rendered
- Reacts looks at where the state resides so we can re-render as little as possible

Committing: React transfer changes from virtual DOM to our browser

- React changes the real DOM (from the browser) wherever it finds a difference between the virtual DOM and real DOM
- React only modifies differences — new real DOM reflects what the virtual DOM is
  - Painting converts it

Create a blank React app:

- **npm create vite@latest**
  - Name, framework (React), variant (JavaScript)
- **cd app-name**
- **npm install**
- **npm run dev**
- Visit **localhost:5173** in browser

React Router

Documentation: **https://reactrouter.com/en/6.28.1**

- URL = base URL + route path
- Need a **react-router-dom** in package.json file
  - Specify a version (min version allowed) or else can't do anything with router
- Allows us to display different React components based on what the current path is
- Outlets + hooks like useParams and useOutletContext help us pass info down our route tree

Most common imports:

```
import {
 createBrowserRouter,
 createRoutesFromElements,
 Route,
 RouterProvider,
} from 'react-router-dom'
```

- Creating the router lays out the routing for entire element:

```
const router = createBrowserRouter(
 createRoutesFromElements(
   <Route errorElement={<NotFound />} elements={<App />}>
     <Route path="/" element={<Feed />} />
     <Route path="/profile" element={<Profile />} />
   </Route>
 )
);
```

- "Tree" with App | NotFound at top and Feed & Profile below

```
ReactDOM.createRoot(document.getElementById("root")).render(
 <>
   <RouterProvider router={router} />
   {/* use Router to route between pages */}
 </>
);
```

React Router Setup:

- Imports many packages (will not be changed) from react-router-dom
- Render the router into the element (.root)
- <Outlet/> is a placeholder in App.jsx – when we render at /, will be replaces
- const myTestProp = **useOutletContext()**.myTestProp
  - Used to pass the same thing to all subcomponents from App.jsx
- The path to a route is the concatenation of relative routes along the root
  - Error elements catch when paths don't matche valid router paths

Dynamic Routing: useParams

- Want as many profiles as possible and dont want to hardcode every single one as React components
- Use the **useParams** method

```
const Profile = () => {
  const name = useParams().name;

  const [catHappiness, setCatHappiness] = useState(0);

  useEffect(() => {
    document.title = `${name}'s Profile Page`;
  }, []);
```

```
      <Route errorElement={<NotFound />} element={<App />}>
        <Route path="/" element={<Feed />}/>
        <Route path="/profile/:name" element={<Profile />}/>
      </Route>
    )
```

APIs and Promises

- Right now, our Catbook is static (does not change with user data)
- Frontend: interacts with user; backend: data storage/manipulation

HTTP - Hypertext Transfer Protocol

- Standardized form of requests and responses for website
- HTTPS is HTTP secure
- HTTP(S) Methods: get, post, put, delete

Request:

- Request Headers: provides context for the HTTP request (fancy)
  - Timestamp, language, etc
- Request Body: data associated with the request
  - Key-body pairs
- Open developer section → **cmd + shift + i → network** → see all the requests our website makes
- Typing a URL into any browser sets off a get request that often responds with HTML, JS, CSS

Response:

- Headers: info about the response
- Body: the response data

Respond - Status Codes:

- 404 (page not found)
- 400 (bad request)
- 500 (internal server error)
- 200 (ok – request successful)
- 1xx - informational

- 2xx - succeeded
- 3xx - redirect
- 4xx - you did something wrong
- 5xx - server did something wrong

APIs: Application Program Interface

- Set of endpoints of a service that allows you to make requests in order to perform a function
- Ex: Google Calendar API, Amazon Selling Partner API, Open AI LLM API, YouTube v3 API, OpenWeather API, The Dog API, Google Maps API, Twillio API, Twitter API, MIT People API
- You need to access data, but cannot access data on servers directly (inconvenient and security nightmare)
- Server forwards requests from our client to database and APIs
- Return type of a get function is a **promise**
    - Promises allow users to do things while server takes its time fulfilling the request

Promises in JavaScript:

- .then() → once promise is fulfilled, do stuff (call a callback function); returns a promise
```
get('/api/stories').then((storiesObj) => {
      setStories(storiesObj);
});
```
- .catch() → once the promise is rejected, do stuff (call a callback function)
```
get('/api/stories').then((storiesObj) => {
      setStories(storiesObj);
}) .catch((err) => {
      console.out('this is so sad', err.message);
});
```
- .all() → returns a promise that resolves once all promises in array resolve
```
Promise.all(promises).then((allResults) => { … })
```
- .race() → returns promise that fulfils or rejects with the first promise that fulfils or rejects
- .any() → returns a promise that resolves when any of the input promises fulfils

~Backend: Servers and Nodes

- Some computer that our client requests data from
- Need for server: want to request data from a central point (file access), centralization (true state multiple people can go to), security (don't want client to access database)

- A server binds to a port on a computer
    - Computer has multiple ports
    - Server will be on a certain port listening for requests
    - protocol://domain:port
    - HTTPS websites: 443, HTTP: 80
    - Most websites have a default port
- Every computer can run server code (run a program designed to actively listen to requests from other computers on a network)
    - Special domain localhost (sending request to own computer through a certain port)
- Frameworks: handle the logic of listening to ports and sending along to be handled (low level communications)
- Javascript we run on server is for the client; our computer doesn't understand Javascript, which is why we use Node.js
    - Node js = a JS runtime
    - Have already been using it → **npm** = Node Package Manager
    - package.json holds project metadata
- /client folder: contains all our React code, components, pages, utilities, etc (front end)
- /server: contains all our backend code
- Other folders: set up by staff to actually run the website
- Frontend: **npm run dev** → use localhost:5173
- Backend: **npm start** → use localhost:3000

API Endpoint:

```
const app = express();
app.get('/api/test', (req, res) => {
      res.send({message: 'Wow I made my first API'});
});
```

- HTTP method: `app.get`
- Express route: `'/api/test'`
- Parameters: request and response object
    - req = incoming request
    - res = server's response

Middleware:

- Run code in between receiving a request and running endpoint code
    - Workers in an assembly line
    - Called in order of definition
- Ex: console.log() on server that prints in terminal that runs npm runs start
- app.use() takes optional path and "middleware object" (often callback func)

```
app.use(req, res, next) => {
```

```
        console.log('time:' Date.now())
        next()
});
```

- This executed for every request to the router

```
app.use(err, req, res, next) => {
        ...
});
```

- Error middlewares take in four parameters and catch if endpoint code errors; defined last
- Catch All endpoints: app.get('*')
    - All endpoints which are not concretely defined will hit this
    - Log the error seen in terminal and send to client seen in browser

```
app.all("*", (req, res) => {
 console.log(`Route not found: ${req.method} ${req.url}`);
 res.status(404).send({ msg: "Route not found" });
});
```

Get Requests:

- **req.query**
- Ex: req.query.content, req.query.parent

Post Requests:

- **req.body**
- Ex: req.body.content

App vs Router:

- App: (server.js) represents your overall server (main application)
- Router: (cat.js, dog.js, etc) isolated groups of API endpoints (mini applications)
    - const router = express.Router();
    - Organization / modularity
- app in server.js → middleware to route /api paths → router in api.js

Workshop 3 Notes:

- Getting and setting stories in front end:

```
useEffect(() => {
  // TODO (step1): fetch the stories from the server
  get("/api/stories").then((storyObjs) => {
    const reversedStories = storyObjs.reverse();
    setStories(storyObjs);
  });
}, []);
```

- HTTP Request: get

- We use a .then() because we don't want to wait for the request to be fulfilled — we create a promise and .then() handles the content after it is resolved
- Express routing documentation: https://expressjs.com/en/guide/routing.html
- Any future endpoints we write can be put in api.js

Design, UI & UX, Figma

UI: User Interface (visuals)

- Fonts, color palettes, shapes + layout, reusable content
  - Adobe Fonts
  - Coolors (palette)
  - Think about color psychology
  - Web design museum (how websites have changed over time)
  - Consider user base
- UI changes over time — overall, we see a trend towards more minimal UI in past decade
  - Also varies across culture (ex: Yahoo in America vs. Japan)
- Make UI look good:
  - Consistency (use UI guidelines)
  - UI component library (ex: Mantine — can customize and reuse components)
  - Responsive design (go to **view → inspect** and choose dimensions to check resizing and how it looks)
  - Interactivity (allows users to interact with website)

UX: User Experience (usability)

- Use symbols, concepts, and colors that are commonly understood across cultures
- Contrast color checker: https://contrastchecker.com/
- Make it intuitive (large buttons)

Wireframing:

- Represent the skeletal framework / blueprint of website
- Use placeholders and focus on overall structure
- Can be drafty / mockup
- Make a wireframe:
  - **Figma** https://www.figma.com/
  - **Figma cheatsheet:** https://drive.google.com/file/d/1BJesvDGOprIPs-FtDjfRHVbBtPGYsVFW/view
  - Google slides
  - Pen & paper

Prototyping:

- Take a wireframe and bring it to life
- Improved UI, can further understand technical requirements of project

Databases

- Database (DB): Organized collection of data
  - Amazon Neptune (graph based), IBM IMS (hierarchical), influxdb (time series)
- Database Management System (DBMS): collection of functions that let you retrieve/add/modify/delete data
- Storing data in server as variable is wrong because:
  - Can run out of memory
  - All data is gone
- Can load data & write data to file
- Get: Frontend talks to server, server talks to DBMS, then DBMS retrieves and gives back to server
  - get("/api/stories")
  - DBMS.find(Storys)
  - post("/api/stories", {content: "new story"})
  - DBMS.add(Storys, {_id: 5, content: …})
- Write: server gives new data to DBMS

Kinds of Databases:

- Relational Database (SQL) → Stores data in a spreadsheet-like format (tables) with rows and columns
  - Relations between different tables
  - Problems: can be complicated to make relationships between tables; need overhead code for relations between tables
- Document Database (NoSQL) → documents, alike JSON objects
  - Don't need to have the same fields
  - Might want common objects living in the same collection (comments, stories)
  - Optimizes write speed, memory usage, query speed, and concurrency issues
  - Ex: MondoDB
- Run MongoDB on the 'cloud' (in case one fails)
  - Duplicate data across different hard drives for redundancy

MongoDB (Database)

Mongo Cheatsheet:
https://drive.google.com/file/d/1LI2XNX7lekOLdPccL1u9Eiy4KAGEgEsq/view

- Different clusters: comments, stories, users
- Can edit field and modify data directly
- Efficient when we need to write lots of data
- Structure of data is very prone to changes
- Relatively easy to use as programmer
- Structure:
    - Database: group of collections
    - Collection: group of very similar pieces of data
    - Document: single JSON or JS object
    - Field: attribute we want to record

Mongoose:

- Object Data Modeling (ODM) Javascript library
- Allows us to interact with MongoDB cluster
- Enforce schemas and models
- Creates documents
- Interacts with databases
- Every document is auto assigned a unique identifier (_id field)
- Useful when theres a relationship between documents

```
const mongoose = require(“mongoose”);
…
```

Schema:

- Map to a single MongoDB collection and define the structure of documents in that collection
- Define the keys (document fields) and types of values corresponding to keys
- Schema types: string, num, date, buffer, bool, mixed, object id, array

```
const UserSchema = new mongoose.Schema({
      name: String,
      age: Number,
      pets: [String]’
});
```

Models:

- Constructors that we define from a Schema and apply to MongoDB collection
    - Construct documents, query for documents, delete documents, update, etc

```
const User = mongoose.model(“User”, UserSchema)
```

Creating Documents:

```
const Tim = new User({name: “Time”, age: 21});
Tim.save()
   .then((student) => console.log(`added ${student.name}`)
```

Finding & Deleting Documents:

- First argument describes how to filter the collection
- To execute the query, we must explicitly invoke it
- Can add as many params as you want to filter

```
// Returns all documents
User.find({})
    .then((users) => console.log(`Found ${users.length} users`));

// Returns all users age 21
User.find({age: 21})
    .then((users) => console.log(`Found ${users.length} users`));

// Returns all users age 21 named Tim
User.find({name: "Tim", age: 21})
    .then((users) => console.log(`Found ${users.length} users`));
```

```
// Deletes the first user in the collection named Tim
User.deleteOne({"name": "Tim"})
    .then((err) => {
        if (err) return console.log("error 😟");
        console.log("Deleted 1 user! 🎉");
    });

// Deletes all users in the collection named Tim
User.deleteMany({"name": "Tim"})
    .then((err) => {
        if (err) return console.log("Couldn't delete 🤷‍♀️");
        console.log("Deleted all users! 😮")
    });
```

Promises & Await

- **Syncronous**: Processes happen one after the other (“one order at a time”)
    - Lots of time wasted
- **Asynchronous**: Processes can run at the same time (“multiple orders at a time”)
- After placing a delivery order, or creating a promise, they will have one of three statuses:
    - Fulfilled .them
    - Pending
    - Rejected (something went wrong) .catcjh

- If a promise is fulfilled, do stuff (callback function)

Await & Async:

- Only asynchronous functions can use await

```
const myFumc = async () => {
      console.log(await a + await b);
};
```

- Can use .then(), but it's uglier
- Waits for the promise to resolve and uses that value

```
console.log(await a + await b)
```

When to Use Async:

- Running background tasks without stopping the user from interacting with the front end
  - Fetching data
  - Downloads / uploads
  - Ex: Can still click around on other stuff as Spotify plays our music

Auth: Authorization and Authentication

- **Authorization**: Determines what a user can access and what actions they can perform
  - Verifying user permissions
- **Authentication**: How we are proving our identity to the website
  - Verifying user credentials
  - Bad to store user / pass info about each User in our database — password is not encrypted & hackers can easily read it
  - Solution: Hash functions → take in a string & mathematically generate a string (one way & deterministic) → bad because can very easily look up common hash codes and try them
  - Solution: Hash Salting → adding random strings at the end → people can still eventually guess
  - Solution: Google sign in → but… how to prove to out website that we logged in / already logged in?

- **Sessions**: user logs in, server stores the session & responds with a session ID
    - Secure because the server stores all the information about the user and only sends back session ID
    - Issues: multiple different servers = each server needs a different glocal lookup table for users
- **Tokens**:
    - User submits login form, server creates a JWT (JSON web token), browser puts JWT in local storage, signed JWT header validated on future requests

Server vs. Sessions:

	Sessions:	Server:
Stores authentication details:	Server	User
What users send to have req authorized:	Cookie	Token itself
Can server perform security actions?	Yes – all authentication details are stored on the server side	No — authentication details are stored on user side; server does not store authentication details

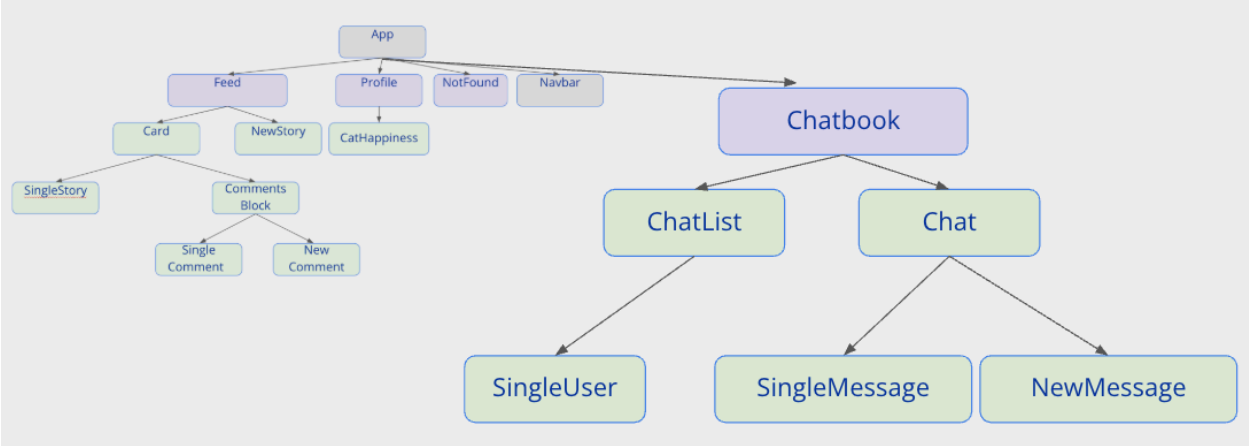
Catbook manage login:

- Seperate auth server & resource server
- Initial login: sign in with Google
- Staying logged in: Express js sessions
- Login to website → inspect → network → payload → receive success token
    - JWT.io → paste token you get, and once you decode, you have a lot of info that is stored in this token (email address, name, etc)
    - If you delete the cookie, you'll be signed out and automatically get logged out
    - Sends cookie as identity verification for subsequent requests

W5 Notes:

- Need the Google Client ID for front and back end (need to verify token)
- In index.jsx – OAuthProvider sends the client id
- Make a User model (edit user.js)
    - name: String
    - googleid: String
- auth.js → persisting user
- Need to add routes of login and logout on api.js

Chatbook



- **Backend**: what inputs to AI requests (req.query, req.body), and what API requests need to return to frontend (if any)
    - Get all of the messages
    - Send a message to everyone
- **Data Representation**:
    - ChatData
        - Messages = array of MessageObjects
        - Recipient: a UserObj
    - UserObj
        - _id: String
        - name: string
    - MessageObj
        - sender
        - content

Ex: Chatbook data representation:

```
UserObject: {
  _id: String,
  name: String
}

MessageObject: {
  sender: UserObject,
  content: String
}

ChatData: {
  messages: array of MessageObject,
  recipient: UserObject
}
```

Message schema (in the .js file) — workshop 6:

```
//define a message schema for the database
const MessageSchema = new mongoose.Schema({
 // TODO (step 3.1): Write the schema for a message
 sender: {
   _id: String,
```

```
  name: String,
},
recipient: {
  _id: String,
  name: String,
},
timestamp: {type: Date, default : Date.now()},
content: String,
});
```

Sockets: Socket.IO (uses WebSocket)

How to use sockets:
https://docs.google.com/document/d/1H3pie1d1yz3LrRPtKcEi1e5aF6LrKg0yQIfvqVb6nZk/edit?tab=t.0

- Sockets enable fast, live communication between the server and client, while API endpoints are for slow data communication
- We use a server socket to broadcast live updates to the clients, and use a client socket manager to receive update from the server
- Important part of live interaction/connection (game, chat)
- Supports many clients interacting with a game state at the same time
- Limitation of HTTP: client sends request to server, and server responds to client
    - Server can't send data to client unless a request is made
    - Could constantly poll the server (ask every x seconds if new requests are made), but this is very slow & inefficient
    - Solution: Teach server how to initiate conversations
- **Broadcast** a message from server to every user connected
    - socketManager.getIo.emit("event_name", data)
    - Title (channel) and the data
    - socketManager.getSocketFromUserId
    - Getes a specific user
- **Listen** for messages on client
    - socket.on("event_name", someFunction)
    - Title and what to do when you get a socket emit of that title
    - Function looks like (data) => {do something with data}
- server-socket.js exports functions for us to use, we can import it then start using
    - Documentation on socket.js functions: https://docs.google.com/document/d/1Q8_T7NEc1ROY7LhwvOTgXzr3SFFGVLWOFqTBOYKyCoE/edit?tab=t.0#heading=h.p4253amxfdiu

- Ex: sockets in chatbook → when a user sends a message, all other users will see that message; when a user joins, all other users will see them
    - DM others
    - See messages live
- **io.emit** is public to all sockets, but if we want the server to amit to a particular cient socket, we use **io.{specific client id}.emit**
- Server maintains 2 mappings: user id → socket & socket id → user
    - Server maps user id to socket and socket id to user
- addUser(name, socket_id) ← server maps name to socket id and now you can get a user's socket!
- socketManager.getSocketFromUserID(id) ← get a user's socket

Advanced CSS & Other Libraries

Tailwind: https://play.tailwindcss.com/
Slides with examples:
https://docs.google.com/presentation/d/1F_QJJjkFw9ZP9_mjTO88ENyxpk1BXC74jgyaHvLRTLQ/edit#slide=id.g1ee5fc8e84d_0_112

CSS:

- **CSS Combinators**: specifies relationships between CSS selectors, such as HTML tags (div, p, etc)
    a. Descendant selector (space)
        ■ Matches all elements that are descendants of the specified element
    b. Child selector (>)
        ■ Matches all elements that are direct childre of the specified element
    c. Adjacent sibling selector (+)
        ■ Selects a single element that is directly after another
    d. General sibling selector (~)
        ■ Selects all elements after another specific element
- **Display Types**: tells browser how to display an element and its child on page
    a. display: grid
        ■ Tells browser to display child elements in a 2-d layout
    b. grid-auto-flow
        ■ Row instructs browser to prioritize adding rows, columns vice versa
    c. grid-template-rows/grid-template-columns
        ■ Allows us to modify the width / height between rows / cols
    d. display: none

        - Tells browser to remove an element from the document
    e. visibility: hidden
        - Tells browser to hide an element, but it still takes up space
- **Content Overflow**: allows us to tell browser how to handle child elements that may exceed the size of parent element
    a. visible (will see the element)
    b. hidden (clips the content into the element)
    c. scroll (display a scroll bar always in the overflow)
    d. auto (display a scroll bar only if needed – if there is overflowing content)
- **Animations:** Give HTML elements some movement
    a. Keyframes
        - Describes the animation we're creating, and what will happen at different points of the animation (ex: opacity)
        - @keyframes fadeIn { … }
    b. Calling our animation
        - Can reference the name to call the animation
    c. Duration
        - Tells the browser how long the animation should last
    d. Delay
        - Tells the browser a delay before the animation is executed
    e. Timing functions
        - ease (default) → slow start, fast middle, slow end
        - ease-in → slow start
        - ease-out → slow end
        - ease-in-out → slow start & slow end
        - Linear → uniform speed

TailwindCSS:

- Utility-first CSS framework that utilizes pre-made classes to make development quicker
- Low level; can create different components even with the same utility classes
- Tailwind reduces CSS bundle sizes to the absolute minimum
    - Smaller CSS bundle sizes = faster load times
- Emphasizes responsive design
-

Games

- Complicated game logic and state
- Performance super important
- HTML5 Canvas is a good way to render animations on the front end

    - Origin is at top left, then increases
- Emit socket messages from both the client to the server and the server to the client
- Event listeners on the client allow the website to take in user input
- The game state is stored on the server, where the ground truth of the game should be stored
    - All game logic should be done on the server
- Upon a component unmount (event listener disconnect), or a client socket disconnect, we should clean up the user from a game

Typescript - statically typed

Incorperate **Typescript**: https://www.sitepoint.com/how-to-migrate-a-react-app-to-typescript/

- Language built on top of Vanilla JS that enforces static typing
- Validates that you cod works at compile-time
- Save your life when debugging
- Javascript = dynamically typed
    - Types are only associated with values, so a variable type can change during execution
- In Typescript, you need to declare the type for the function so users know what type gets passed
- Easily integratable with your projects
- Functions in Typescript are treated as a variable, so you can add them as a property

Static Typing Can Catch:

- Missing or unnecessary prop values
- Similarly named variables or functions
- Undefined & null value behavior
- Overloaded operators

RSC (React Server Components) and Next.js

- **Serverless** = way of running code so that code normally ran on server are bundled and ran individually when called
    - Server = living in house (need to manage your own load balancing, resource allocation, etc)
    - Serverless = living in hotel (cloud provider stores and runs your code for you)

Pros / Cons of serverless:

- P Scaling (auto provision of resources)
- P Lower costs (pay for what you can use)
- P Focus on development (indrastruvture management handles by provider)
- C Cold starts (latency with functions that are called for first time in a while)
- C Lack of global state (sockets won't work out of box)

Next.js vs. React

- Full stack framework using React as the frontend framework of choice
- Built in support for routing, filesystem based routing, while React needs React Router
- React is a single page application (SPA) while Next is a multi-page application (MPA)
- Next optimizes your site out of the box
- Next contains Middleware capabilities
- Next pre-renders our HTML document on server

When to use Next.js

- Great option for full stack applications, such as
    - Interact with database
    - Authentication
    - Dynamic (changing) data
    - API layer (inward / outward facing)

Single Page Application (React) → bundle of HTML and JS are downloaded by client

- Client then runs JS to render the app on client
- Downside: everything handled by browser (client) – must wait for entire JS bundle to download, data fetching dependent on user connection speed

Server Side Rendering (Next.js) → initial render of document is sent to client first; we can display HTML without even running any JS

- From there, we wait until JS bundle is downloaded for our site to be interactive
- Servers are much closer to data & more consistent / reliable
- Clients can vary in performance (unpredictable), and when building apps with sensitive data, cannot trust client

React Server Components (RSC) → split our code into client and server components

- Split in the middle & let each handle their strengths
- Hybrid approach gives us an overall better user experience
- Client components sent first (visual feedback), server makes request to database and then combines it with server components then sends to client → loading complete

How to Code Good

- Use prettier (VSCode extension)
    - **Either everyone on team uses it, or everyone does not use it**

Lag & Optimization:
- Minimize unnecessary & repetitive computations
- Bundle communication into packages

Games:
- Movement curves (acceleration, sustain, deceleration)
- Input buffers (some lag time for users to interact with game)
- Wall sliding (remove components of motion)

Documentation:
- Most important: API documentation, front end props

Debug:
- Different parts: mongo, node js, express api, src (server backend, client frontend)
    - Where along the stream of info is the code coming from?
- Check the browser console (command + option + j)
- console.log things (from front end sent to backend)
- Make small changes then test app incrementally
    - Keep functions short and modular

Git Hygiene:
- Always git pull
- Dangerous commands:
    - git add .
    - git push --force
    - git reset --hard
    - git commit --amend

Deployment

- Making your web app accessible to the world
- localhost:5173 → yourwebsite.com
- We're using render to deploy

- Slides: https://docs.google.com/presentation/d/1jnk_IfpU-d1El0xM42FuOjR_S2eJtXinPBpIE3ifVzo/edit

Last Lecture

- Full stack design
    - Design with all layers of app in mind
    - Front end, server side, database
    - “What data do we need to store? What user wants”
- Feature by feature
    - Design features independently
    - Visual display of info about user

Your Website:

- Front end, back end, database (almost like a bridge connecting front and back)
- Good documentation and communication
- Divide the work
- Quality > quantity

Criteria:

- Functionality
- Usability
- Aesthetics
- Concept execution
- NO CRASHING (“we will be attacking your website seeing if it crashes)

Special Prizes:

- Unique concept
- Responsive UI design
- Innovative UI feature
- Innovative backend feature
- Webby award
- Futuristic UI design
- Best Social Impact