#### **Javascript Notes**

* Javascript file linked to HTML document lets us add interactivity
* In JS file linked to HTML document: we have access to “document” object, representing the HTML file
* Document Object Model (DOM)
  + Represents a HTML document as a tree
  + Document object is the root node; each part of the HTML doc (elements, text, comments) is a node, with parent-child relationships according to the HTML document
  + Each element is represented (modeled) by an object, which the linked Javascript file can interact with/modify
  + **DOM serves as an interface through which the linked Javascript file can modify the HTML document**
    - JS file makes changes to the DOM, and changes in the DOM are immediately reflected in the display of the HTML doc
    - Essentially: think of the DOM as the “live HTML doc” that dictates what’s rendered
* Access an HTML element: const element = document.getElementbyId(id)
  + Returns object representing HTML element with corresponding element
* Add HTML element:
  + Create the element: const newElement = document.createElement(tag)
  + Set attributes and content
    - Attributes: newElement.attribute = value
      * Note: for class, you do .className
    - Text Content: newElement.textContent = value
      * Sets the text content
    - Inner HTML: newElement.innerHTML = expression
  + Add the element to the DOM
    - Append to existing element: parentElement.appendChild(newElement)
      * Where parentElement is an object representing a parent element
      * Appends newElement to the end of parentElement’s content
* Remove HTML element: elementToRemove.remove()
  + Where elementToRemove = corresponding DOM element to remove
* We can also modify existing HTML elements!
  + Text content: element.textContent = value
  + Inner HTML: element.innterHTML = expression
  + Attributes: element.setAttribute(name, value), element.removeAttribute(name)
  + Classes: element.classList.add(name), element.classList.remove(name)
* Note: distinction between text content and inner HTML!

* Event handling
  + We can add an event listener to an element in the DOM
  + **Syntax:** element.addEventListener(eventType, eventHandlerFunction, [useCapture])
    - eventType = string representing type of event to listen for
    - eventHandlerFunction = callback function that is executed once event occurs; takes in one argument, an Event object representing the event
    - useCapture: optional parameter; if true, event listener is triggered during capturing phase; if false, event listener is triggered during bubbling phase
      * Default value = false.
  + Capturing vs bubbling phase
    - When an event occurs on an element (called the target element), it is also registered for all its ancestor elements
    - The event then propagates in two phases: 1) capturing phase, where the event travels from the root of the DOM (this root is the document object) to the target element along the path between them. And 2) bubbling phase, where the event travels from the target element back up to the root of the DOM.
  + Common eventTypes: “click”, “mousedown”, “keydown”, “keyup”, “keypress”
* Note on HTML forms: traditionally, when we make a HTML form, we have an input element of type submit that upon being clicked, submits the form. After that, an HTTP request is sent to the provided link.
  + However, this is annoying because we have to make a backend to deal with HTTP requests.
* But we can also handle form submissions directly in the front-end using Javascript!
  + Idea is: have a “submit button” so that when it’s clicked, our JS code fetches the value of the input field and clears it.
    - Note: in a HTML <input> element of type=text or number:
      * The value attribute sets the element’s initial/default value
      * **At any point, the value property of the corresponding DOM element reflects the user’s current input**
* Note: if a Javascript file is linked to two HTML documents: at any point, Javascript code will interact with the DOM of whichever document is open.
* Web Storage API: Easy way to send data between different pages (in Vanilla HTML/JS)
  + Allows us to store data on one page and retrieve it on another
  + Basically an interface for a JS script to interact with the browser’s memory; hence why it’s called an API
    - Supported by modern browsers; no need for additional imports in JS
  + Two types of storage: localStorage and sessionStorage, both store key-value pairs
    - localStorage has no expiration date, data persists even when browser closed/reopened
    - sessionStorage stores data for the duration of the page session: i.e. as long as the current tab is open
      * So: if we navigate to another page on the same tab, sessionStorage persists
      * But if we make a new tab, there’s a new sessionStorage
      * And if we close the tab (or window), sessionStorage is gone
  + Syntax:
    - Store data: localStorage.setItem(key, value)
      * Or: localStorage.key=value
    - Retrieve data: let value = localStorage.getItem(key)
      * Or: let value = localStorage.key
    - Remove data: localStorage.removeItem(key)
    - Exact same syntax for sessionStorage
  + Other notes:
    - Generally, setItem/getItem preferred to dot notation
    - localStorage.length returns # of key-value pairs stored (same for sessionStorage)
      * Don’t store length as a key
    - values in localStorage/sessionStorage can only be strings; so if we want to store objects, convert to JSON
* JSON (Javascript Object Notation)
  + Basically, a notation to encode objects/arrays into a string (objects = collection of key-value pairs)
  + Key idea: We can serialize (“encode”) an object/array into a JSON string, and then uniquely parse (“decode”) a JSON string to obtain the corresponding object/array
    - This works for complex nested objects/arrays as well
    - This allows us to send data involving objects in the form of a string, which is important because many in many cases we can only send strings (e.g. the query for a HTTP GET request)
  + Another important property: easy for humans to read and write JSON
  + Note: “JSON object” refers to a Javascript object obtained from parsing a JSON string
* Note on linking Javascript files in an HTML document:
  + Typically, we link Javascript files either in the head or at the end of the body
    - Either way, any linked JS file will have access to the entire DOM
  + Each Javascript file we link has access to the global scope in all previously linked Javascript files
    - So if we define a variable (say, x) in the global scope of a previously linked JS file, we can access that x in our current code
    - Important: **load order matters!** Each file has access to global scope in all files linked earlier, but not files linked later (since link order = load order)
* URLs/links using Javascript:
  + window.location object represents current url
  + To tell browser to navigate to newLink: window.location.href = newLink
    - We can both to relative or absolute path
    - Relative path: goes relative to current page
    - Absolute path: goes relative to overall base url
* Note on some bugs:
  + If our Javascript file interacts with the DOM, we generally want to link at the end of the body
    - This way, HTML loads before JS file, so we don’t run the risk of trying to access a DOM element that’s yet to be loaded
  + By default, a button inside a form will submit a form, causing a page reload
    - This means buttons inside forms that navigate to another page won’t work
    - To fix this, either use preventDefault, or just don’t use forms
  + In general, it’s just easier to not use HTML forms and let JS handle submissions.
* Note: traditionally, uninitialized variables are equal to undefined in Javascript; but in strict mode, accessing an *undeclared* variable will throw an error (we can still access declared but unassigned variables)
  + However, even in strict mode, we can always access any property of an object that’s been declared; nonexistent properties will evaluate to undefined
* Note: Javascript functions are flexible with handling arguments; won’t throw an error when there’s the wrong number of arguments.
  + We can call a function with fewer arguments than it has parameters. The remaining parameters will then be set to undefined.
  + We can call a function with more arguments than it has parameters. The extra arguments will just not be bound to any parameter. (they can still be accessed using the arguments object)
* Using Web Lab’s utilities.js:
  + Make HTTP request in client: get(endpoint, params), post(endpoint, params)
    - Both times: params is an object. (Note: for the key-value pairs in params, the values should be simple primitives, not objects; JSON can’t parse objects.)
    - get and post functions both return a promise that resolves to the response object the server sends (note: server must do res.send(object))
      * If an error occurs in the client-side processing of the response (e.g. response is not an object): it goes to the .catch (in the get/post request!)
  + Process HTTP request in server:
    - For GET request, access params with req.query
    - For POST request, access params with req.body
* IMPORTANT NOTE: For HTTP GET requests, we should have query be a simple object consisting of key-value pairs of strings. So: in query = {key1:value1, …}, each value should be a string!!!!
  + Don’t store objects as values in the query parameter, because they don’t parse properly.
  + If we need to search for something specific / need identity, pass ID as a query parameter or serialize it with JSON, and let backend handle the rest.

#### **ReactJS Notes**

* Main benefit of React JS: very good at abstraction, allows us to create “custom HTML elements” essentially.
* In order to even work with React in the first place, you need to:
  + Make a package.json file for dependencies/scripts/version control
  + Go to terminal in that directory and do “npm install”
  + Run “npm start” to open a local server with the page
* In React: first decide what the components are and what props will be passed.
  + This is very important!
* IMPORTANT: a component can have multiple states!!!
  + Can have multiple states, each set by a useState hook
  + Whenever one state changes, the component re-renders but the other states stay intact.
* Important: always set state, never assign!
  + And: setState must change the state’s *identity* (memory address) for a re-render!
* To have components update:
  + For each content/attribute that needs to update, make it either part of the component’s state, or have the component receive it as a prop.
  + If part of state: re-renders when state updates.
  + If it’s received as a prop: re-renders when that prop changes (in parent component).
* Important note: re-rendering is asynchronous, and only occurs once the current execution context (i.e. frame) is done. So if we call setState(X) (the state setter) in a function, then the rest of the function will first execute before the component is re-rendered.
* Also: whenever a (functional) component is re-rendered, its entire function executes again, from the start!
  + Only difference: useState(X) hook only *initializes* the state when the component is first rendered
  + Also note: a component’s setter function is stable (i.e. stays the same) throughout the component’s lifecycle!
* To have a child component update a parent component:
  + Pass down the updater function for the parent, as a prop.
* To have a parent component update a child:
  + Use “lifting the state up” approach!
  + Alternatively: make the child’s setter a global variable, so the parent component can access it.
* “Lift the state up” approach
  + When information needs to be shared between components in different branches: best approach is to “lift the state up” to their least common ancestor.
    - Also works for a child giving its parent info.
  + Approach: include the shared info in the LCA’s state, and have the LCA pass down both its state and state setter as props.
    - This way: components can all access and change this info.
  + Best when shared info needs to update a lot and trigger re-renders.
  + And this is easy, since components can have multiple states!
* Global variable approach:
  + When you need to share info among components in different branches or have a child component give info to its parent, easy way to do this is:
    - Just to give the info to a global variable!
    - (Always updated when re-rendering)
  + Best for when the info is stable/static (e.g. a setter method)
* UseEffect Hook:
  + Syntax: useEffect(func, *optional* dependency array)
  + func = effect function
  + Dependency array = array of variables; calls func each time a variable in dependency array changes
    - Dependency array can consist of anything, not just states!
  + Effect function **guaranteed** to be called upon mounting
  + If no dependency array: effect function called every render (incl mount)
  + If dependency array is empty: effect function only called upon mounting
  + If nonempty dependency array: effect function called upon mounting, and whenever a dependency changes. (NOT called every render)
  + In general: better to have useEffect hook than code after a setState call
  + Cleanup function
    - If the effect function returns a function, then the returned function is used as a clean-up function.
    - The clean-up function is called just before each time the effect function runs, and also when the component unmounts.
    - Important for cleaning up event listeners we set up in the effect function.
* Rendering arrays
  + React can render an array of JSX elements
  + If you have React render an array of JSX elements, it just maps each element in order. (alternatively, we can put them into a list)
    - Note: if component returns array of JSX elements, each one should have a unique key attribute.
* Routers
  + React is meant for making single-page applications, so use routers to handle this. Particularly, Reach Router.
  + Note: in Reach Router, we can have url parameters!
    - Syntax: path/:var\_name
    - Then, when you visit path/:value, Reach Router passes down var\_name=value as a prop to the component.
* Less important notes:
  + To pass down props to a child component in JSX, put it as attributes. Then, React will put all of them into an object.
  + Consider breaking down a component into smaller parts when:
    - The component is getting too long/hard-to-read
    - The component has many parts with their own functionalities
    - The component is handling too many responsibilities
  + Conditional rendering: “condition ? resultIfTrue : resultIfFalse”
    - Expression that evaluates to “resultIfTrue” if condition is true, and “resultIfFalse” if condition is false.
    - Reduces if/else statement to a one liner
  + Note: effect function in useEffect runs AFTER the component has finished rendering, not during rendering phase!
    - So if the effect function has a state update and we call useEffect with no dependency array, then there will be an infinite loop.
* Running React: easiest way is to use Create React App (CRA), which abstracts away all the configuration with Wolfpack, Babel, etc
* Integrating CSS into React:
  + In each JS file with React components, import the corresponding CSS stylesheet, so the stylesheet applies to the components in this file.
    - But note: CSS cascades down from parent to child components. So any CSS on a parent component will also apply to child components, unless overridden.
* IMPORTANT NOTE: The state setter function for a React component remains stable (i.e. stays the same) throughout the entire lifecycle of the component
  + Remains stable in both behavior **and identity** (i.e. memory location)
* To manipulate/access DOM in React: use refs
  + Refs are basically references to DOM nodes that let us access/manipulate them
  + Syntax: const myRef = useRef(null)  
     <Component ref={myRef}/>
    - useRef(x) returns a ref object with a “current” property, initially set to x
    - When we set “ref” attribute of a JSX element to a ref object, this ref object is connected to that element upon rendering
      * Then, ref.current (“current” property of the ref object) is equal to that element’s DOM node, through which we can access/modify attributes
  + Summary: create ref with useRef hook, set it to the “ref” attribute to connect it to a JSX element, then use ref.current to access the DOM node
* Context
  + Problem: oftentimes we want to pass down some information many layers deep, and we want components to re-render when it changes
    - It’s very annoying to straight-up pass it as props; instead, we use context
  + Context is essentially global information that can be changed or accessed anywhere
  + **Creating context:** first create a context instance.

import {createContext} from “react”;

const MyContext = createContext(defaultValue)

* + - defaultValue = context value that components access when they don’t have a matching provider (i.e. provider of the same context instance)
  + **Providing context:** create a context provider associated with the context instance
    - const WrapComponent = (props) => {

return <myContext.Provider value={props.val}> <MyComponent/> </myContext.Provider>

}

* + - Basically: this provides a mechanism for <MyComponent/> to access the value of myContext, which is always equal to props.val
    - And <MyComponent/> renders within <WrapComponent/> as a child
  + **Consuming context:** For a (functional) component to consume context, use the useContext hook
    - import {useContext} from “react”;

const myComponent = () => {

const value = useContext(myContext)

return <stuff prop={value}/>

}

* + - Essentially: myComponent consumes the context from myContext; basically value is a prop, and the component re-renders when its corresponding context provider changes values
  + Note: we can have multiple context instances. And a single context instance can have multiple context providers with different values scattered throughout the DOM tree; each component, when consuming context, does so wrt to the nearest ancestor context provider in the DOM tree.

#### 

#### **Promises/Async Notes**

* Javascript is synchronous and single-threaded
* Javascript has a call stack: basically, a stack where we push the function when we start execution, and pop it when we finish execution.
* Idea of asynchronous function is: we want the current call stack to clear / the function to execute later, not right now.
  + However, sometimes other functions are dependent on the result of this asynchronous function!
  + So when another function is dependent on an asynchronous function, we must include the dependent function as a callback in the asynchronous function! (We must do this to guarantee the dependent function runs right after.)
  + So when we have a chain of dependencies of asynchronous functions, we’ll either have nested callbacks (callback hell), or we’ll have the functions specifically tailored to a particular sequence (reducing flexibility, and allowing errors to propagate).
    - Both of these are worse than using promises and async/await.
* Remember: API requests are asynchronous!
* Promise = object that represents the eventual completion/failure of an asynchronous operation
  + Promise constructor: new Promise(executor)
    - executor is a callback function we write, that takes 2 arguments provided by Javascript: resolve and reject (in order)
      * resolve(x) resolves the promise with fulfillment value x
      * reject(x) rejects the promise with rejection value x (often an error)
  + .then() method, handles fulfillment:
    - Syntax: promise.then(handler)
      * handler is a callback function; JS gives it 1 argument, the promise’s fulfillment value
    - .then() method returns/evaluates to a new promise
    - If the handler returns a normal value: the new promise resolves to that value
    - If the handler returns a promise: the new promise is set to that promise
    - If the handler throws an error at some point: the new promise is rejected
    - Note: to reject the new promise returned by .then, the handler must either throw an error or return another promise that’s rejected
    - Key idea: **handler is guaranteed to run AFTER promise resolves!** (but we don’t know how long after; .then() method is also asynchronous)
  + Since .then() method returns/evaluates to a new promise, we can chain .then()’s to have a chain of dependent asynchronous operations!
    - If any one fails: it will go to the .catch() at the bottom
  + .catch() method, handles rejection
    - Syntax: promise.catch(handler)
    - Also returns a new promise; evaluates the same way as handler in .catch!
      * Handler in .catch gets 1 arg from JS: the error value/reject reason
  + We can have multiple promise.then() chains; each chain will run independently. This is good if we need to handle the promise in multiple independent ways.
  + In general, it’s best to keep the .then()/.catch() chain until there are no more dependencies.
    - We *could* extract the fulfillment value/rejection reason from a promise by assigning to an outer variable in the .then/.catch handlers, but this is bad practice, since we don’t know when these will be assigned.
  + And remember: when you create a promise object, you’re running the code/asynchronous processes inside it.
  + Promise.all(arr):
    - Takes an array “arr” of promises, and creates a new “combined” promise.
    - This new promise “runs” each individual promise, and resolves to an array of the resolutions of the individual promises (if they are all fulfilled)
      * If any individual promise is rejected: the combined promise is rejected
  + Important note: a promise will still be resolved/rejected and the .then called, even after the function it’s defined in has been exited!

* Misc. notes:
  + Note: a Javascript function that doesn’t return anything automatically returns “undefined”, by default.
    - So: if the handler function in a .then() method doesn’t return anything, the new promise (i.e. promise returned by the .then()) resolves to “undefined”
* Async/Await
  + async/await keywords: lets us work with promises more easily
  + async keyword: append in front of function declaration
  + async function: either returns a promise (which can be either fulfilled or rejected), or returns a promise wrapped around the “normal return value”
    - I.e. if the normal function returns some value, the async function returns a promise that has resolved to that value
  + await keyword: can only be used in an async function
    - Syntax: let x = await someAsyncOperation()
      * someAsyncOperation() is a function that returns a promise
    - await keyword: waits for the promise to be fulfilled, then “extracts” its resolution value
      * Key idea: the execution of the function is paused until the promise is resolved/rejected!
      * So in let x= await someAsyncOperation():
        + We don’t execute the rest of the function till someAsyncOperation() is completed
        + In this case: x is set to the fulfillment value of the promise returned by someAsyncOperation()
    - HOWEVER: await doesn’t directly let us handle errors; so when using await keyword, we must do try/catch for error handling.
  + Async function that throws an error returns a rejected promise, with rejection reason set to the error that was thrown.
  + So: we can do .then()/.catch() on async functions, since they return a promise!

#### **Express JS Notes**

* ExpressJS = web framework for NodeJS (built on top of Node.js and extends it)
  + Express.js lets us build the backend, Node.js lets us run it on our machine
* Route handlers:
  + Handle HTTP requests to routes
  + Syntax: app.METHOD(path, handler)
    - app = Express.js server
    - method= get, post, put, delete
    - path= route we’re writing the code for
    - handler= callback function that takes 2 args (req, res), then handles it
      * Req, res are both automatically provided by express
      * req = object representing the HTTP request
      * res = object representing our response
        + res.send(x) sends back a Response object, containing both the metadata and content
        + What we send as our response is what the API call (a promise) resolves to
        + Note: API call = HTTP request to an API
      * Note: handler also has optional next parameter, in case we want to pass control to the next middleware.
  + Note: order route handlers from most specific (first) to most general (last)!
    - ExpressJS tries to match the route handlers in sequential order, so we don’t want a handler for a more general route catching what’s meant for a more specific route
      * (However, ExpressJS does only match exact patterns, not prefixes)
  + Catch-all route handler: have a route handler with path=”\*” at the end, so it catches all paths that we didn’t explicitly define
  + Essentially: route handlers are special middleware for specific routes (HTTP request + path)
  + Can also do app.all(path, handler) to handle all HTTP requests to path
* Middleware:
  + Basically, a function that helps handle a request to a server. Runs after the request is made, but before the response is sent to the client.
    - Often use: processing requests/preparing responses
  + Note: route handlers are also middleware!
  + Syntax: app.use(path, handler), to register a middleware function with Express.
    - path = what path/route we want this middleware to apply to; optional arg, will apply to all paths is left out
    - handler = the actual middleware function.
      * Syntax: (req, res, next) ⇒ body
      * When middleware executes, expressJS automatically passes 3 arguments: req, res, next
      * req / res = request / response objects
      * next = callback function that hands control to the next middleware
      * We can also have two parameters: (req, res)
        + Then Express passes in request & response but not next
  + In middleware function, when we call next(), Express finds the next middleware function with a matching route to req, and executes it with the same req/res objects (so we can modify req/res to process requests/prepare responses)
    - Note: “next middleware” means in sequential order in the code (i.e. first to last)
  + When we call next(err) where err=Error object, we instead go to the next (matching-route) error handler middleware.
  + Error-handling middleware: same syntax (app.use(path, middleware\_function), where path is optional), except the middleware function has 4 args.
    - Middleware function: (err, req, res, next) => body
    - err = error object when calling next(err)
    - Express JS distinguishes error-handling vs normal middleware from # of parameters in middleware function: 3 args is normal, 4 args is error-handling
  + Note: route handlers are special middleware designed to handle a specific route!
    - ExpressJS recognizes which one to use via the method/path in the request; so NEVER CHANGE METHOD/PATH IN THE REQUEST!
  + Note: each middleware function either needs to send a response or pass control to the next middleware.
    - Never do both! That will cause bad behavior.
  + In general, request-response cycle: first, client sends a request to a specific route in the server, then a series of middleware functions handles the request, then a response is sent to the client.
* Routers
  + Basically a “mini-express application” for handling a certain subset of requests
  + Syntax: const router = express.Router(), to make
    - app.use(router), to use (after import)
  + Syntax for middleware/route handling for routers is the exact same as for the express app itself
    - router.use(path, handler)
    - router.method(path, handler)
  + When we do app.use(router), we’re basically copy/pasting all its middleware. So ExpressJS goes through middleware first in order of routers, second in order within each router.
  + We can mount a router on a base path, by doing: app.use(prefix, router)
    - This essentially adds a prefix to all paths in the router
* **IMPORTANT:** Normal middleware functions match prefixes, but route handlers match exact paths!
  + I.e. app.use(*path*, *handler*) applies to all paths prefixed by *path*
  + Whereas app.[HTTP method](*name, handler)* only applies to exactly *path*
* General order of ExpressJS code: first middleware, then route handlers, and last error handlers.
* And at the very end of the code, start the server.
  + Syntax: app.listen(port, [hostname], [callback])
    - port is required, hostname and callback are optional arguments
    - Binds the server to listen at the specified post/hostname, then calls the callback function
    - Note: app.listen(port, hostname) and app.listen(port, callback) both work!
      * ExpressJS distinguishes which one it is via type checking.
* Note: we can also chain middleware!
  + I.e. do app.use(path, middleware1, middleware2, …)
  + This executes a sequence of middleware in a given order for a certain route
    - (Works for both router handlers and general middleware functions)
    - Works for both overall server and router objects
* Note: use require(filename) to import stuff
* When we visit a webpage, our browser (the client) makes a HTTP GET request to the server for the HTML/CSS/Javascript files, which the client then renders.
  + In ExpressJS: express.static function enables the server to send static files from a given directory to a client
  + Syntax: app.use(express.static(directory));
    - This makes it so that when the client makes a GET request to “server\_url/name”, the server will send the file “directory/name”
      * directory can be a relative or absolute path
      * If no file “directory/name”: moves onto next middleware
    - When client encounters linked files in the file served (e.g. linked CSS stylesheet or JS script in a HTML file), it makes HTTP GET requests to those addresses
      * We should use relative paths for these! (So client can figure out where to make a request)
    - So we want to have one directory consisting of **all** static files we send to the client, and then do app.use(express.static(directory)).
      * If we’re using a bundler like Webpack: make sure to put bundle.js into the public directory (that we do express.static on)
      * Note: if we’re using Webpack, we only need to do app.use(express.static(client\_dir)), where client\_dir is client/dist or client/public
      * This is because all the client-side JS and CSS files get bundled into one file by Webpack, called bundle.js; so that’s the only static JS/CSS file that is served to the client!
        + Note: CSS can get bundled into bundle.js via, for instance. injection through the <style> element
  + Important: we should avoid any overlap between our static file paths and paths that handle other GET requests! (e.g. API endpoints)
  + Note: static files are just files that are served to the client as is (i.e. the same for all clients / sent without server-side processing)
    - HTML, CSS, and Javascript files (including w/ React) are static files
    - Even though JS file may interact with the backend, *that file itself* is sent as is to all clients!

#### **MongoDB Notes**

* Structure/hierarchy:
  + MongoDB instance (has one or more databases)
    - Database (group of collections, typically corresponds to one web app)
      * Collection (collection of documents with the same structure)
        + Document (basically, each object)

Field (some attribute of a document)

* Mongoose: wrapper/library to “connect” with MongoDB in NodeJS
* In Mongoose, we have *models*
  + Think of each model as a constructor/representative for a collection
  + We can use the model to access elements from the collection, or create a new document (of the model’s type) and save it to the collection
  + (We want one model per collection)
  + Syntax for setting up a Mongoose model:
    - const mongoose = require(“mongoose”);

const schemaName = new mongoose.Schema({field1: type1, ...});

module.exports = mongoose.model(collectionName, schemaName);

* + - Note: Mongoose automatically pluralizes and lowercases collectionName; then either finds the collection or creates a new one
    - To have a default value: do “key: {type: typeName, default: defaultValue},”
      * defaultValue can either be a value or a function
      * If it’s a function, Mongoose calls it to set the default value
* **Creating a document (MongoDB object)**
  + Syntax: const newDocument = new myModel({field1: value1, …});
    - myModel is a Mongoose model; serves as a constructor that creates a document according to its schema
* **Saving a document**
  + Syntax: newDocument.save()
  + This returns a promise: resolves to the saved document if successful
    - Can handle using .then/.catch
  + Document is saved to the collection through it was created
* **Finding documents**
  + Syntax: myModel.find(query)
    - Finds all documents satisfying query in collection corresponding to myModel
    - Returns a promise that resolves to an array of documents
    - If nothing found: promise resolves to [] (empty array)
  + Syntax: myModel.findOne(query) / myModel.findById(id)
    - Finds one document in the collection that satisfies query/id
    - Returns a promise that resolves to that one document
    - If nothing found: promise resolves to null
  + Most common query: filter by property values
    - If query=undefined (i.e. no query argument): finds all documents
  + Find + sort: myModel.find(query).sort({field1: ±1, …})
    - Returns a promise to the sorted array. In order of priority, sorts by field1, then field2, … (1 = ascending order, -1 = descending)
* **Updating documents**
  + Syntax: myModel.update(query, update, [options], [callback])
    - Query: same format for filtering as for .find
    - update = {$set: {field1:value1, …}}
    - Options: optional arg. If we want to update all documents found, set options = {multi: true}; by default, it will just update one document
  + More clear methods:
  + myModel.updateOne(query, update, [options], [callBack])
    - Updates one document matching the query
  + myModel.updateMany(query, update, [options], [callBack])
    - Updates all documents matching the query
  + Note: modifying a Mongoose document object (obtained using .find) will NOT update the corresponding document in MongoDB!
* **Deleting documents**
  + myModel.deleteOne(query)
    - Deletes document matching the query
  + myModel.deleteMany(query)
    - Delete all documents matching the query
  + myModel.findByIdAndDelete(id)
    - Deletes the document with a given id
* IMPORTANT: **All** the update and delete methods return a promise that resolves upon the completion of the operation (no error if nothing was updated/deleted)
* If our schema doesn’t have an \_id property, an \_id property is automatically created upon saving documents to MongoDB
  + However if our schema does have one, it overrides the default \_id property
  + Default \_id property created by Mongoose is an ObjectId type
    - This is an object, so we can’t compare equality using ===; instead, use the .equals method: id1.equals(id2)
      * id1.equals(id2): returns true if equal, false is not
    - Alternatively: to work with ObjectId’s more easily, convert them to strings using: const str\_id = id.toString() (where id is an ObjectId instance)
    - model.findById still works with string version of id!
* Note on MongoDB documents: MongoDB document objects are stored in BSON, but we can access properties using dot notation. Can also convert to a normal Javascript object
  + Convert to plain JS object: const JSdoc = document.toJSON();
  + We can use normal object destructuring on MongoDB documents
  + When we do res.send(Mongoose document), it automatically converts to JSON before sending! (In the process, \_id and other ObjectId fields are stringified)
* Note: in Mongoose, we can also filter by sub-properties for retrieving data!
  + Particularly: do {“property.subproperty”: value} to filter by nested subproperties
    - Quotations are optional for single-level fields but required for nested fields
  + Dot notation lets us drill down an arbitrary number of levels
* In Mongoose, generally try to avoid filtering by {property: object}, since that can be tricky to keep track of. Instead, filter by {“property.subproperty”: primitive}.
* Note: best to convert Mongoose docs to objects before sending thru HTTP req/res

#### **Sockets/SocketIO Notes**

* The normal HTTP communication protocol is the client making a request and the server responding; the server can’t initiate communication.
* WebSockets are another type of communication protocol that enables both the client and server to make requests/responses.
* To implement WebSockets: use Javascript SocketIO library
* Server-side:
  + Setup/integration:
    - const express = require("express");

const { createServer } = require("http");

const { Server } = require("socket.io");

const app = express();

const server = createServer(app);

const io = new Server(server);

* + - This makes io, a new SocketIO server instance
      * SocketIO server is basically an interface for the server to talk to/listen to the client
    - In code above: we bind a socketIO server to a HTTP server, which lets them listen on the same port
      * This way: the application can handle both HTTP requests (through Express), and WebSocket connections (through SocketIO)
  + Important concept: sockets
    - A *socket* (aka server-side socket) is an object (accessible to the SocketIO server) representing a single client’s connection to the SocketIO server
    - Sockets have methods that enable the server to communicate through that particular connection with the corresponding client
    - Each socket has a unique id (type string)
      * Access using socket.id
  + Server event listener: io.on(eventName, callBack)
    - Calls callBack function upon event occurring
      * This only works for predefined events (most commonly: eventName = “connection” and “disconnect”
    - **IMPORTANT: This does not work for custom events!**
      * To listen for a custom event, must put it in the listener of an individual socket, rather than the SocketIO server.
    - When eventName=”connection”, callBack is given 1 argument by socketIO: the socket representing the connection that triggered it
  + Socket event listener: socket.on(eventName, callBack)
    - Server listens to eventName from a particular socket
    - When eventName occurs, callBack function is executed
    - callBack function receives arguments = data sent by the client
  + io.emit(eventName, [data1], [data2], …, [acknowledgementCallback])
    - Emits an event with specified name and data to all clients
    - Often used to send a message
  + socket.emit(eventName, [data1], [data2], …)
    - Emits an event to the client of a particular socket
  + Using .emit, we can send both primitives and objects/arrays (SocketIO automatically serializes them to JSON, then deserializes upon reception)
    - But when sending mongoose documents, we should first convert to an object before emitting
      * Syntax: const obj = doc.toObject()
  + Important concept: rooms
    - Rooms are used for the server to group the sockets (think of chatrooms)
    - Server can have many rooms, & each socket can be in multiple rooms
    - Each room is uniquely identified by its name, a string
    - socket.join(roomName), socket.leave(roomName)
      * Methods for sockets to join/leave a room
      * Take an optional callback function with an optional error argument
    - socket.rooms: returns Set of names of all rooms that socket is in
      * Note: every socket is automatically in the room with name = its own ID!
    - io.socketsJoin(roomName), io.socketLeave(roomName)
      * Makes all sockets (connected to io) join/leave a room
    - io.to(roomName).emit(event, data)
      * Emits an event to all the sockets in a given room
    - Io.sockets.adapter.rooms = mapping from room names to sockets in it
      * io.sockets.adapter.rooms.get(roomName):

returns a Set of all the socketIDs in room with given name

* + - Note: once a socket disconnects, it automatically leaves all its rooms.
* Client-side:
  + Setup:
    - Include the SocketIO client library
      * If we’re doing “vanilla” HTML/CSS/JS: link to it in our HTML doc
        + Either download the library and host it locally, or include it using a CDN (content delivery network)
        + Then just put our JS file after it, and it’ll work (since our JS code will have access to the functions in the library)
* If we’re using a module bundler like webpack or framework like React: just install SocketIO client library via “npm install socket.io-client”, then import it in Javascript file.
  + Default export of SocketIO client library: function that creates a client socket
    - const socket=io(url);
      * Here, io is the default import from the SocketIO library; it’s a function that creates a client-side socket object
      * So io() creates a client-side socket representing the connection of this client to the SocketIO server at url
        + Client-side socket is distinct from the server-side socket!
      * If url left blank: connects to server that serves/hosts the page
        + This works because we’ve bound the SocketIO server to the HTTP server, so they’re on the same port/address!
  + Event listening: socket.on(eventName, callback)
    - Listen to eventName emitted by the server, then executes callback
    - callback takes args = data sent from the server
  + Event emitting: socket.emit(eventName, [data1], [data2], …)
* Key idea/functionality in sockets: event listening/emitting between server and client
  + Serverside emit/client listen:
    - Server: serverside.emit(eventName, data1, data2, …)
      * Serverside = io or some room or some socket
    - Client: socket.on(eventName, callBack)
      * socket = Client socket, and callBack takes args data1, data2, …
  + Client emit/serverside listen:
    - Client: socket.emit(eventName, data1, data2, …)
    - Server: serverside.on(eventName, callBack)
      * Where callBack takes args data1, data2, …
  + So: syntax for listening/emitting is exactly symmetrical for client and server!
* Note: on the server-side, the only way we can access the (server-side) socket through which was sent, is inside the “connection” event listener
  + So: if we need any server-side event listener to have access to the socket, put it inside the “connection” event listener
    - And inside connection event listener, we should only have socket-specific event listeners (since they apply directly to the socket that connected)
* Note: socket communication is asynchronous!
* Acknowledgement callback functions
  + emit actually has an optional last parameter = acknowledgement callback
  + Syntax: something.emit(eventName, data1, …, ackCallBack)
    - First arg = eventName, middle args = data, last arg = ack. callback
    - SocketIO figures out if the last arg is a piece of data or the acknowledgement callback by checking if it’s a function
      * (We can’t send functions via .emit; we can only send primitives and objects/arrays)
  + Note: if the event emitter sends an acknowledgement callback, the (corresponding) event listener’s handler function is automatically given both the data and the acknowledgement callback.
    - So the listener’s handler function must take the acknowledgement callback as the last parameter!
    - To acknowledge having received the event: the listener (handler) function calls ackCallBack and passes response in; then, ackCallBack(response) is executed on the emitter’s side (remember, we can’t pass functions through sockets, so receiver doesn’t know what ackCallBack is)
  + Full syntax:
    - Server: serverside.emit(eventName, data1, …, data\_n, ackCallBack)
    - Client: socket.on(eventName, handler)
      * handler takes args: (data1, …, data\_n, ackCallBack)
      * handler calls ackCallBack(response), to execute ackCallBack(response) on server-side
      * Note: handler can also just call ackCallBack(), with no passed in response
  + Acknowledgement callbacks: useful for if we want to emit a number of events in sequence.
    - One way: inside the acknowledgement callback, emit the next event
      * But this can lead to lots of nested callbacks, aka “callback hell”
    - Better way: Make a Promise that resolves once the acknowledgement is received! (put .emit inside Promise’s executor)
      * We can make a helper function that does this.
      * **To emit a number of events in sequence: just wrap a Promise around all of them (using helper function), then put them in an async function using await.**
    - Note: if we wrap a Promise around an serverside emitter that emits to multiple clients, it resolves *as soon as one of the clients acknowledges it.*
      * Important property of promises: once it’s resolved/rejected, it’s no longer mutable, and all attempts to change its state are ignored.
* IMPORTANT: the client socket and server socket for a given connection **both have the same id!** (Upon connection, server makes an id and sends it to the client too)
  + Thus: when a user connects to the SocketIO server, they can send us their user info and client socket id, so our server has a mapping between user/socket!
* Note: we remove a socket event listener using socket.off(eventName, handler)
  + handler must be *the same in identity* as the original handler in socket.on!!!
* When we want to have a React component use a socket event listener:
  + Put the socket.on in a useEffect hook to make it only setup upon mount
  + Add a cleanup function to delete the event listener upon dismount
  + Make sure that the handler function is defined INSIDE useEffect! So that the cleanup function uses the original instance of the handler function.

#### 

#### **Other Stuff**

* Mantine: library of tons of different types of React components
* Typescript: Javascript but statically/strongly typed
* Tailwind CSS: CSS library that provides lots of utility classes for styling
  + Utility classes: CSS classes that do just one thing
* Bootstrap CSS: CSS library that provides component templates, results in uniform/similar-looking components
  + Also a framework, where components come with HTML/CSS/JS
* Cheat sheets
  + Front-end JS cheat sheet: <https://drive.google.com/file/d/1TaTQlxP1PBgpbQBx8Ae7fzZe1yDaXfCl/view?usp=sharing>
  + Git cheat sheet: <https://drive.google.com/file/d/19aWrCYUr0q82jhh755scC-1yp6SiLjv8/view?usp=sharing>
  + Databases cheat sheet: [https://drive.google.com/file/d/1VWm2W0sgGjCkCgYxV0m0YSCdqmbB7hXh/view?usp=sharingdatabases-cheatsheet.pdf](https://drive.google.com/file/d/1VWm2W0sgGjCkCgYxV0m0YSCdqmbB7hXh/view?usp=sharing)
* Summary of how a website runs:
  + Server is hosted somewhere; may or may not connect to a database.
    - Server is controlled by back-end code.
    - Server hosting = giving the server a device/place to run
  + When a user navigates to website, the browser (client) makes an HTTP GET request for the static content (HTML/CSS/Javascript files), then the server sends them and the client renders it.
  + Key point: **rendering is done client-side!**
    - So the user has access to all the front end files (including React components); so all sensitive logic/data should be handled in the backend
  + If there’s authentication: the server authenticates the client before sending user-specific data (which is then rendered by the client)
* Remember: in Javascript, a function’s scope solely depends on where it was defined, and not on where it’s called. So regardless of where it’s executed, it still has access to the variables in the scope in which it was defined!
* Important implementation strategy to resolve circular dependencies: **wrappers**
  + Let’s say we want file1 to run some functions from file2, but those functions need information from file1.
  + We need to import file2 in file1, but we can’t also import file1 in file2, otherwise there will be a circular dependency.
  + Solution:
    - In file2, rather than directly exporting the functions, export “wrapper” functions that take in information and run the functions based on that.
    - In file1, import these wrappers and pass in the necessary information.
  + Key idea: **Import wrappers and then pass in information!**