Part 1: Review of lecture content (slides 1-5)

1. What is Node / why are we using it?
    a. Many of you likely used it in 031
    b. A JavaScript environment used for web development
    c. Many packages written by other people that use Node, some for very specific tasks but lots for the kind of generic tasks that lots of web developers need (like how you find yourself wanting to reuse a lot of the concepts from Twitter). When making a website you will likely end up installing multiple such packages to do some of the grunt work for you
    d. Why are we using Node?
        i. Convenience, like we just said
        ii. Moving to JavaScript:
            1. Static websites: every user sees the same content, each page is "manually" generated (this is what the _site folder is)
            2. Dynamic websites: pages are generated on the fly and can depend on the user
            3. Static websites do have benefits! Can load much faster, often much easier to spin up (hence using Jekyll). But dynamic websites have a wider range of functionality
                a. Can't make a static site for every data point in every database we'd want to display
2. Quick overview of Express and routes (from lecture)
    a. In lecture we learned about Express, which is a web serving framework built on top of node. Its main functionality is that it handles routing; it's also modular so you can plug other packages into it, which we'll talk more about later.
    b. What is routing? What does it look like in Express?
        i. Routing: when the server gets a request, send it to a particular place based on the resource being asked for
        ii. You call a function on the router object which is the RESTful verb (one of the 5 we learned last recitation, like get or post), and pass in the RESTful noun
    c. Toy example (slide 5)
        i. In the first line you can see that the verb is post and the noun is /user, so we can guess that this function will create a new user
        ii. In the second line we can see boilerplate passing in the request and response
        iii. In the third line, we access the body parameters of the request to get out the username and password of the new user (we could also access the query parameters if there was something in there) - we are just assuming those parameters are there since this is a new user request, later we'll get to validation
        iv. The rest of the lines are making and sending the response, which sends an HTTP status code (201: 2xx = success) and a success message

Part 2: Middleware (slides 6-10)

1. Intro the concept of middleware
    a. What is middleware? Officially, anything that happens in the middle, between the request being sent by the client and the response being sent by the server
    b. What might you want to do between those two things? E.g. check if the user is logged in, since if not, that might change your response dramatically
    c. Another point - you might have a lot of routes where before you respond, you want to check if the user is logged in! Having dedicated middleware to do that keeps your code dry
    d. You call middleware from a route, e.g. the route to make a post, or to change your settings, might call the “isUserLoggedIn” middleware
    e. Note: The UI can also provide validation, for example graying out the Tweet button if the tweet is too long, but you want code validation as well to prevent people from hacking into your site / changing the URL to do something the UI won’t allow them to do
2. Walk through an example of middleware (slide 7) - explain how it works in detail
    a. Async function
    b. Takes the request and the response, as well as a next function
    c. The next function allows multiple middleware functions to be called from the same route - at the end of every middleware function, you have to call next so that the next function can run and the route can eventually return its own response
        i. If you want to throw an error instead of going to the route, you can return after sending the error and that will prevent next() from being run
    d. Having access to both the request and the response lets you do anything that you could do in your route: in this case, the middleware is checking the session (which we’ll get to in just a moment) to see if the user is logged in, and if they’re not, it sends an error and terminates.
        i. If the user is logged in, nothing happens except for next() – this is desired behavior because it means the rest of the middleware, and eventually the route if those middlewares also succeed, will just start executing
    e. Incorporating the middleware into a route (slide 8)
        i. Import the middleware, usually at the top of the page
        ii. Add a list of all the middleware you want to call as another parameter in the router function right after the RESTful noun
        iii. In this case, before we let the user create a new account, we check that the username they want to use is valid, check that it’s not taken, and check that the user is logged out
        iv. Note: picking good names for your middleware makes your routes a lot easier to understand! These are just functions like the “middleware” one we saw on the last page but the names are more helpful
        v. Also note: these will get called in the order they are listed, and if e.g. the first one returns an error, the rest won’t get run. So sometimes you may want to order them strategically

3. Exercise (slide 9): write middleware to check if a user's proposed username is valid
    a. Solution on slide 10

Part 3: Sessions and Cookies (slides 11-12)

1. Sessions
    a. A *session* is a set of requests to your website from the same client (i.e. computer) within a set time period (you could think of the time period as a session, which is a more natural terminology).
        i. Side note: is the same client always the same person? Often, but not always! That can have security implications. Ex: when a public computer tells you not to click "remember my login"
    b. All websites have users. Lots of them want to be able to identify their users. Lots of ways to do this (accounts, etc.). No matter how you choose, you'll likely want some amount of *persistence* (so you don't have to make your users log in over and over)
        i. HTTP is *stateless*, which means that each request is not associated with any other requests, so by default there's no way to figure out which ones came from the same place. Sessions and cookies let us keep track of that
    c. How many requests? How long of a time period? Tricky! Complicated factors go into this decision, which we will outsource
2. Slides: Within a session, we want persistence. How do we do that? Cookies
    a. Not the yummy ones :(
    b. You've probably heard of this kind of cookie, even if only in the context of clicking "accept all" while you're trying to read an article
    c. A cookie is a little piece of data that a website makes and puts on your computer, so that the next time your computer reaches out to that website, it can recognize the cookie
        i. Useful for keeping track of the session, which has benefits discussed earlier
        ii. Also useful for keeping track of if you're the person who was interested in a particular item in an online shop, or clicked on a particular type of article, or set the website to dark mode or the font size larger
            1. Useful to who? You, sometimes. The website and whoever profits from it, often

Part 4: Demo & exercises (slides 13-14)

1. TA demos an example website implementing basic express-sessions (user creates an account, can then log out and in). Then TA walks through the code thoroughly showing how it works and taking any questions. Points covered in the walkthrough:
    a. You will be given starter code extremely similar to this (just less simplified) for A5 so worth paying attention!
    b. Lots of stuff going on here but because it will all be given to you, you don't need to know how to replicate it. Plus we'll give more helpful tips when A5 comes out.

So for now just focus on the files I'm showing you and the things in them that I'm focusing on

c. api/index.ts: where the app is initialized
    i. Ignore almost all of it and only look at about api.use(session)
        1. This is where we tell the app to use sessions, once we do this there will always be a req.session object available anywhere req is available and we can store things in it
        2. Make sure to look at the place right underneath this where we set the session userId variable!
d. models/user.ts: where the user accounts and functions to access it are defined
    i. Don't go into any detail but look at the names of the functions you can access
e. routes/users.ts
    i. Don't go into any detail but look at the names of the routes and what you would call them for
f. routes/sessions.ts
    i. Look at the names of the routes
    ii. See how we set req.session to the user's id when they log in, and then set it to undefined when they log out - then in other routes/middleware we can check if they're logged in by checking req.session.userId
    iii. Reinforce concept from last recitation - this is still CRUD because we are representing users being logged in and out with sessions being created and deleted
g. routes/middleware.ts
    i. This is a fake middleware, you would obviously not really want this
    ii. You can match up each part of it to the middleware we saw earlier
    iii. Note that it returns two different errors depending on the condition

2. Rest of rec: activity
    a. Starter activity: Add a variable to sessions to count the number of times the user has viewed the page and report it back to them in the response
    b. Rest of rec: Give a variety of prompts for what students could add to extend the project and let them pick.
        i. Add middleware that checks if the user's password is correct and if their account exists, and don't let them log in if not
        ii. Add middleware that checks if the user is logged in/logged out, and don't let them log out/log in otherwise
        iii. Add middleware that checks if the user's username is taken and don't let them create an account if it is