Computer Science 3100 **Winter Semester 2019**
Dr. Rod Byrne **J. House, G. Mensah, J. Patel**

Presentation Outline

What is the React Library?

Gillian

(1) **HTML: What HTML?**

React projects serve one HTML file, `/index.html`. [Show this file.] This file includes one (rather large) JavaScript file created by WebPack which encompasses all the modules and libraries used in the app.

(2) **JSX: Adding logic to markup.** React's "JavaScript" is more than vanilla JavaScript. Importing `React` from the `react` library allows us to write markup HTML elements (and custom derivatives) in JavaScript — no we don't mean huge strings containing the markup. [Show `frontent/src/components/PostCreatePage.jsx`.] These files are commonly saved with a `.jsx` extension.

These can be objects, extending `Component` from the `react` library — in which case they can contain *state* — or they can be functions, which do not have state. These *components*, as they are called, allow for modularity.

(3) **React Router: Route Awareness on the Frontend.**

React is a frontend JavaScript library that creates a Single Page Application — the first page load is the only one where the browser receives HTML and the page "flashes"; subsequent requests fetch at most a JSON object. To allow a logical flow, and to make pages bookmarkable, SPAs need to manipulate the URI in the address bar of the browser.

We use a library called `react-router-dom` to do this. It pushes "pages" into the history stack as the user navigates the website, and includes JavaScript to get the browser URI and render a particular DOM layout

sort-of dynamically based on the route. [Show `frontend/src/App.jsx`.]

Build Features

Jacob

(1) **Node Inception: The Project within a Project.**

[Begin with view of project structure.] We have two Node "projects": the frontend and backend. All of our React codebase is in the frontend. [Show `react` in `frontend/package.json`.] All of our Express, SQLite, etc. code is in the backend. [Show `express` in `package.json`.]

How do we combine these? We run `make build` which installs all required packages in the frontend and backend, and then runs WebPack to export a production-ready *static* site (HTML, JS, images, fonts, etc.) to `frontend/build/`. [While that is compiling, open `app.js`.] Our Express app, based in `app.js`, begins with routes we know exist: `/robots.txt` for site crawlers, `/api/*` for all of our asyncronous frontend-backend JSON communication (as well as image serving!), as well as anything in the `frontend/build/` folder like static JS, font, and image assets.

But the React router includes routes such as `/login` which the Express router doesn't know about! We need hits to the web server for such routes to always serve `frontend/build/index.html` so we use one final Express wildcard route for this.

START THE APP

(2) **Helmet: Express App Hardening**

[Ensure the Express default app is running on port 8000.] If we run `curl -i localhost:8000 | less`, we see the headers that Express sends to clients. Running `curl -i localhost:3100 | less` shows *our* instance of Express' headers.

Security through obscurity is no security at all, but advertising your attack surface unnecessarily isn't advisable either.

OPEN APP IN BROWSER AND INTRODUCE IT

(3) **Web Sockets: Pulling it all Together**

As we can see, our test user has a few unread notifications. As stated in class, updating such a list of real-time events could mean poling a server. This is neither elegant nor scalable. So we used Web Sockets.

One thing we did differently is this... [Open `frontend/src/App.jsx`.] On the browser *close* event, we can tell that our socket has disconnected for some reason — perhaps network access was lost. When this occurrs, we open a *new* connection to the server, without the user needing to refresh the page and without any major user experience disruption.

Demo

Jay

(1) **Gatekeeper: Securing NumHub**

Due to time constraints, we won't show registration, just login with pre-made demo users.

OPEN SECOND WINDOW OF THE APP

If a user forgets their username [Demonstrate this.], we show them that they made a mistake. If they forget their password, we allow an option to reset it by answering a pre-defined security question [Demonstrate this but do not proceed; go back to password form.].

If they were to attempt to sign in 10 times (our test user already tried 8 times), a user locks their account. From this point, only an administrator can reset their logon attempt count. [Go to the first window and demonstrate this, then log on in the second window with the now-reset account.]

(2) **"I have a question!": Creating new Posts**

We've split the set of all posts on our website intoa collection of disjoint subsets. In other words, we have separate "sub-sites" for primary school questions, elementary school, middle schol, high school, undergraduate, and graduate-level questions.

So, all the questions we see here are at the *Undergraduate* level. If we

expnd one of these questions and open it, notice the styling: LaTeX and Markdown has been used to typeset the content.

When you post a new question [Open the post page paste in a question from the question document.], you will need to tag your question as well. This makes the site easier to use and makes finding relevant questions faster.

(3) **Catering to the Know-It-Alls: Answering Questions**

Of course we also have a way to answer questions posted to the site. [Go back to the home page and choose the tag for *Trigonometry* and tick *Unanswered*. The sample question should match this search.]

If we know an answer to a question on the site, it's easy to post it. But what if the clarity of an answer (or question) is impeded without visual assets? If we'd like to include a graph, for example.

[Put in the image for the answer to the trig question.]

We've developed an API that can upload, compress, store, and retrieve images for this purpose. If we inspect the post preview, we see that the image references the `/api` route of our site.

Each image that's uploaded is compressed on the server, hashed, and then a URI is created in Express which maps this hash to a bas-64-encoded image stored in the database. As shown in class, Express can easily serve images from a filesystem; doing so from another source is no different.