Project Instructions: Full Stack App with React and a REST API

Github

- Create a new repo for this project.
- Create a .gitignore and use it to make sure your node_modules folder is not stored in or tracked by your repo.
- Create a README.md file for your repo that explains what the project is and anything your user or fellow developers might need to know to use the project.

Understand what you are working with

- Have a basic understanding of React, JSX, Create React App, React Router, React Context API, React authentication, and working with APIs.

Download the project files:

- The markup folder contains a collection of HTML files that will show you how each page in the app should be structured.
- The styles/global.css file contains all the styles you will need for this project, but you are encouraged to experiment with things like colors, background colors, and fonts.
- The mockups folder contains a collection of PNG image files showing you how each page should look with the provided HTML and CSS applied.

Create your React project

- Use the create-react-app tool to set up and create your React project in a folder named client.
    - To do this, run the command npx create-react-app client from the root of your repo.

NOTE: npx is not a typo — it’s a package runner tool that comes with npm 5.2+.

Set up your REST API

- Add a folder named api to the root of your repo.
- Copy the REST API Express application from your unit 9 project into the api folder.

Add CORS support to your REST API

- When developing your React application, you'll be using the create-react-app development server, which will host your application (by default) at http://localhost:3000/. Your REST API, will be hosted separately from your React application at http://localhost:5000/. While both the React and REST API applications will be using the same hostname, localhost, their port numbers differ, so the browser will treat them as separate origins or domains.

- To successfully make a request from the React application's domain to the REST API's domain, you'll need to update your REST API application to support cross-origin resource sharing or CORS (see MDN web docs: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS for more information about CORS).
    - Add a middleware function to set the appropriate headers to support CORS.
    - Alternatively, you can install and configure the cors npm package (https://www.npmjs.com/package/cors).

Test calling your REST API from your React application

- Before going any further, let's ensure that your React and REST API applications are setup correctly and you can successfully call your REST API from your React application.
- Update the React App component (src/App.js file) to call the REST API to get a list of courses and render the results.
    - We're just confirming the setup of the applications, so just render the list of course titles using some simple markup (e.g. an unordered list or set of divs).
- Open a terminal or command window and start your REST API application.
    - Browse to the api folder and run the command npm start.
    - Once you've started the REST API application, you can typically just leave the app running in the background.
- Open another terminal or command window and start your React application.
    - Browse to the client folder and run the command npm start.
    - The create-react-app development server should start and open your application into your default browser. If the development server started but it didn't open in the browser, try manually browsing to it at http://localhost:3000/.

Build your app components

- Use the provided HTML files (see the markup folder in the project files download) as a guide while you create the components for this project.
- Use the App component (src/App.js file) that was generated by the create-react-app tool as your main container component.
- Create the following stateful class components:
    - Courses - This component provides the "Courses" screen by retrieving the list of courses from the REST API's /api/courses route and rendering a list of courses. Each course needs to link to its respective "Course Detail" screen. This component also renders a link to the "Create Course" screen.
    - CourseDetail - This component provides the "Course Detail" screen by retrieving the detail for a course from the REST API's /api/courses/:id route and rendering the course. The component also renders a "Delete Course" button that when clicked should send a DELETE request to the REST API's /api/courses/:id route in order to delete a course. This

component also renders an "Update Course" button for navigating to the "Update Course" screen.
    - UserSignIn - This component provides the "Sign In" screen by rendering a form that allows a user to sign using their existing account information. The component also renders a "Sign In" button that when clicked signs in the user and a "Cancel" button that returns the user to the default route (i.e. the list of courses).
    - UserSignUp - This component provides the "Sign Up" screen by rendering a form that allows a user to sign up by creating a new account. The component also renders a "Sign Up" button that when clicked sends a POST request to the REST API's /api/users route and signs in the user. This component also renders a "Cancel" button that returns the user to the default route (i.e. the list of courses).
    - CreateCourse - This component provides the "Create Course" screen by rendering a form that allows a user to create a new course. The component also renders a "Create Course" button that when clicked sends a POST request to the REST API's /api/courses route. This component also renders a "Cancel" button that returns the user to the default route (i.e. the list of courses).
    - UpdateCourse - This component provides the "Update Course" screen by rendering a form that allows a user to update one of their existing courses. The component also renders an "Update Course" button that when clicked sends a PUT request to the REST API's /api/courses/:id route. This component also renders a "Cancel" button that returns the user to the "Course Detail" screen.
- Create the following stateless functional components:
    - Header- Displays the top menu bar for the application and includes buttons for signing in and signing up (if there's not an authenticated user) or the user's first and last name and a button for signing out (if there's an authenticated user).
    - UserSignOut - This component is a bit of an oddball as it doesn't render any visual elements. Instead, it signs out the authenticated user and redirects the user to the default route (i.e. the list of courses).

Pro Tip: Resist the temptation to keep and manage the courses data as global state in the App component. Instead, allow the Courses and CourseDetail components to retrieve their data from the REST API when those components are mounted. Using this approach simplifies the management of the courses data and ensures that the data won't get out of sync with the REST API's persisted data.

<u>Set up your routes</u>

- Install React Router and set up your <Route> and <Link> or <NavLink> components.
- Clicking a link should navigate the user to the correct route, displaying the appropriate info.
- The current route should be reflected in the URL.
- Your app should include the following routes (listed in the format path - component):
    - `/` - `Courses`
    - `/courses/create` – `CreateCourse`
    - `/courses/:id/update` – `UpdateCourse`
    - `/courses/:id` – `CourseDetail`
    - `/signin` – `UserSignIn`

    - `/signup` – `UserSignUp`
    - `/signout` – `UserSignOut`

Add support for user authentication

- To prepare for implementing user authentication (i.e. user sign in and sign out), determine where you'll manage your application's global state.
    - One option, is to keep your global state in your App component. Using this approach, the authenticated user and the user sign in and sign out actions (i.e. methods) are made available throughout your application, by using props to pass references down through your component tree.
    - Another option, is to manage your global state using the React Context API. Using this approach, the authenticated user and the user sign in and sign out actions (i.e. methods) are defined using a Context API <Provider> component and made available throughout your application using Context API <Consumer> components.
- Create your signIn() method:
    - Your signIn() method should define emailAddress and password parameters.
    - To authenticate the user, make a request to the REST API's /users endpoint, using the emailAddress and password parameter values to set an Authorization header on the request using the Basic Authentication scheme.
    - If the request to the REST API succeeds (i.e. the server returns an "200 OK" HTTP status code), then you'll know that the supplied user credentials are valid. If the server returns a "401 Unauthorized" HTTP status code, then the supplied user credentials are invalid.
    - After validating the user's credentials, persist the returned user record and the user's password in the global state. Doing this will allow you to create and set the appropriate Authorization header on future REST API requests that require authentication.
- Create your signOut() method:
    - The signOut() method should remove the authenticated user and password from the global state.

Configure your protected routes

- Define a higher-order component (HOC) named PrivateRoute for configuring protected routes (i.e. routes that require authentication).
    - Use a stateless functional component to wrap an instance of the <Route> component.
    - Use the <Route> component's render property to define a function that renders the component associated with the private route if there's an authenticated user or redirects the user to the /signin route if there's not an authenticated user.
    - For an example of how this is done, see: https://reacttraining.com/react-router/web/example/auth-workflow in the React Router documentation.
- Update the following routes to use the PrivateRoute component
    - `/courses/create`
    - `/courses/:id/update`

Restrict access to updating and deleting courses

- On the "Course Detail" screen, add rendering logic so that the "Update Course" and "Delete Course" buttons only display if:
    - There's an authenticated user.
    - And the authenticated user's ID matches that of the user who owns the course.

Display validation errors

- Update the "Sign Up", "Create Course", and "Update Course" screens to display validation errors returned from the REST API.
- See the *create-course.html* file in the markup project files folder.

Add support for rending markdown formatted text

- Use npm to install the react-markdown package (see https://www.npmjs.com/package/react-markdown for more information).
- On the "Course Detail" screen, use the <ReactMarkdown> component to render the course description and materialsNeeded properties as markdown formatted text.

Add HTML and CSS

- Use the HTML files contained within the markup project files folder as a guide while you create the components for this project.
- Use the CSS contained within the global.css file in the styles project files folder for your application's styles.
    - Free free to experiment with modifying the colors, background colors, or fonts in order to personalize your application.

Add code comments!

Cross-Browser consistency:

- Google Chrome has become the default development browser for most developers. With such a selection of browsers for users to choose from, it's a good idea to get in the habit of testing your projects in all modern browsers.

EXTRA CREDIT

Display user friendly messages

- A well-designed application will display user-friendly messages when things go wrong. For example, when a requested page can't be found.
- Create the following stateless functional components:
    - `NotFound` - Display a message letting the user know that the requested page can't be found.
    - `Forbidden` - Displays a message letting the user know that they can't access the requested page.
    - `UnhandledError` - Display a message letting the user know that an unexpected error has occurred.
- Add the following routes (listed in the format `path` - `component`):
    - `/notfound` – `NotFound`
    - `/forbidden` – `Forbidden`
    - `/error` – `UnhandledError`
- Update the CourseDetail and UpdateCourse components to redirect users to the /notfound path if the requested course isn't returned from the REST API.
- Update your React Router configuration so that if a route isn't matched the NotFound component will be rendered.
- Update the UpdateCourse component to redirect users to the /forbidden path if the requested course isn't owned by the authenticated user.
- Throughout your application, redirect users to the /error path when requests to the REST API return a "500 Internal Server Error" HTTP status code.

<u>Persist user credentials</u>

- After successfully authenticating a user, persist their credentials using an HTTP cookie or local storage so that the user's authenticated state is maintained even if the application is reloaded or loaded into a new browser tab.

<u>Redirecting the user after successfully signing in</u>

- After a user successfully signs in, redirect them back to the previous screen (whatever that happens to be).
    - For example, if a user attempts to view the "Create Course" screen before they've signed in, they'll be redirected to the "Sign In" screen. After the user has successfully signed in, redirect them to the "Create Course" screen.