3 components
Client - src - Context.js
Client - src - Data.js

Client - src - components - CourseDetail

Key
Yellow - I've done
Red - Need help, components

- 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 this page on MDN 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).

- Additional note for macOS Monterey users.
  Apple introduced some changes with AirPlay when they launched macOS Monterey. Now, the AirPlay Receiver uses ports 5000 and 7000. You will need to follow these instructions to turn off the AirPlay receiver, to be able to use port 5000 with this project.
- 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 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 in 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 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 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.
- Set up your routes
    - 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 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 this page 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 rendering 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.
        - Feel free to experiment with modifying the colors, background colors, or fonts in order to personalize your application.
-