Overview

In this project, you will exercise the React, HTML, CSS, JavaScript, and Git knowledge that you've gained. You'll build a front-end user interface (UI) that manages customer information. Unless otherwise instructed, each student will work on the project independently.

Core Development

Create an initial React app project.

- Create a new React project.
- Download dependencies.
- Start the development server (npm run start)
- View the application: **http://localhost:3000/.**

Create a static version of the application.

Implement a static version of the application:

- Create an app design,
- Implement static versions of each component.
- The customers list should include (min): "name", "email" and "password" for each customer.
- CSS styles can be minimal.
- Button handlers can output basic messages to console.
- All work can be done inside the App component.

Update the application to include dynamic capabilities.

Add styling and dynamic capabilities to the application.

- Make the app look good - add CSS Styling
- Data for the customer list should come from a hard-coded array.
- Enable navigation between components (if needed)

Add List-Item Selection:

When users click on a customer it should show as selected:

- Items in the list are selected by clicking on them.
- Only one item can be selected at a time.
- The selected list item should appear in bold lettering.

- The useState() hook can be used to manage selected state.

Add List-Item De-Selection:

List items should be de-selectable.

- Clicking on an already selected item should de-select it.

Setup basic customer management operations

The application should allow for basic data operations.

- Allow users to add customers.
- Allow users to edit individual customers.
- Allow users to delete customers.
- Actions should be cancellable.

Upgrade to memdb.js

The **ProjectAssets\memdb.js** file implements an in-memory data store. Updating your code to use the in-memory data store will make changing over to a REST data source later on easier.

- Copy memdb.js into your project.
- Import it where needed.
- Use the methods in memdb to manage create, update, and delete operations.
- Retrieve data using the useEffect hook.

Check Requirements and Re-Test the App

Test implemented application features.

- Decide how you are going to verify each feature you've implemented.
- Check completed features.
- Note what is working and what is not working.
- Fix code so that each check passes.

Implement the Delete Button

Make the delete button work as intended.

- Give users a way to select a customer for deletion.
- Give users a way to execute the deletion.
- Add logic that processes the delete request.
- Make sure the list is updated so that the deleted record is removed.

Add and Update Features

Complete the Add and Update features.

- Decide if you will use separate forms for Add and Delete
- Implement the forms.
- Implement Add and Update logic to set up the forms.
- Make sure text typed into input fields updates the screen.
- Implement logic that Adds or Updates data.
- Make sure the list is updated after Add and Update operations.

Check Requirements and Re-Test the App

Now that you've added or updated features you should test your app again.

- Save your code, restart development server if needed.
- Re-run each existing test.
- Add and execute tests for the new functions.
- Fix found issues.

Refactor the App into Separate Components

At this point, if your multiple components are implemented in a single file (App.js) you should separate them out into individual files.

- Save your code (commit if using Git)
- Create new files for each component you want to separate out.
- Move code for a component to its new file.
- Import the new file and adjust code where needed.
- Test that the component is working and fix any issues.
- Do the same for the second component that was moved.
- Repeat until the desired components have been moved and are again working.

Refactor the App to use REST Server

The application is currently set up to use an in-memory array inside of memdb.js as a data store. Real world applications are more likely to use a REST API as their data source. In this part of the project, you should update your application to use a typical REST API.

- Set up the provided REST server:
  1. Unzip REST server from **ProjectAssets\back-end-rest-server.zip.**
  - Install dependencies: **npm install.**
  - Start the server: **npm run start.**
  - Open a browser to: **http://localhost:4000/customers**.
  - Use Postman REST client to verify the REST server's Get, Insert, Update and Delete operations.
  - Test that the new code is working.
  - Make sure that the list is updated after data operations.

Remember to commit and tag your code before moving on.

Where to go from here

At this point you may have some features of your own that you'd like to implement. If so, then go ahead.

For each feature:

- Save and commit your existing application code.
- Before coding a feature, you should have a clear idea of how you are going to implement it.
- Implement the feature.
- Test the feature and make sure no previous features broke during its implementation.
- When its working, save and commit your code.
- Move on to your next feature.

If you don't have any of your own features to implement you can work on the following:

Feature01 - Update the application to work with larger data sets.

Up until now you have been working with data sets that include just a few customer records. What if you had 100 or 1000 customers? What issues might appear based on the size of the data? For this feature you'll want to:

- Add a large number of records to the app.
- Test the app to see what's working and what can be improved (performance?)

Try enhancing the application based on the requirements set out below. Feel free to adjust the requirements or add to them as you get to know more about the issues.

Ideas for supporting larger data sets:

- Display the number of records your app loads.
- Maybe add scroll bars for easier navigation
- Maybe add buttons users can use to jump or page through the records.
- Allow users to enter a search term to narrow down the list.

Hints:

- An online test data generator like Mockaroo (**https://www.mockaroo.com/**) can be used to create larger data sets.
- Use JavaScript's Array.filter() method to filter the list using a search term.

Feature02 – Use Separate pages for the customer list and the add/update form.

In you want to update the application to use more than one page. For example, when the app starts it will display the customer list. The user would then click an "Add" or "Update" button to navigate to a separate add/update page. To do this you will need the React router.

Try enhancing the application based on the requirements set out below. Feel free to adjust the requirements or add to them as you get to know more about the issues.

Requirements for separate list and edit pages.

- The initial page should hold the customers list.
- The initial page should have an "add" and an "update" button.
- The add and update buttons should navigate to the second page.
- The second page should show in add/update form as well as "save", "cancel" and "delete" buttons.
- "cancel" navigates the user back to the initial list.
- "save" adds or updates a customer record depending on the form's current mode.
- The "delete" button deletes the currently selected customer when the form is in update mode.

Rough work steps

- Add the React router to the project (install and import it)
- Create an EditPage component to hold the addUpdate component.
- Add routing code to the App component.
- Pass the currently selected customer when navigating to the EditPage.

- If no customer is passed, then the AddUpdate form should come up in "add" mode.
- Disable the "Update" button when no customer is selected.

Hint:

- Use the App component as the initial page that shows the list.