CS 295R
Lab 5 – Blog Application
Context and Home Page

The objective of this lab is to allow you to continue to develop your understanding of the use of context as an alternative to props drilling in React applications. You will also have the opportunity to use axios to make an AJAX get, post, put and delete requests. Jest test files as well as an image of a sample home page have been included to help you in completing the lab.

Introducing the Blog App

You'll build a simple blogging application in labs 5, 6 and 7. The application uses json-server to "mock" the backend of the application. A db.json file with sample user information and sample blog post information has been provided. The application allows the user to:

- View a "card" with 12 of the most recent blog posts on the home page of the site.
- See the complete post for one of the "cards" by clicking on the line in the card.
- Log in or create a new account by clicking on the login button on the nav bar on the home page. Once a user is logged in he/she/they can
    - View "cards" for all their blog posts on the home page.
    - Delete a blog post by clicking on the delete button on the "card" for a post.
    - Edit an existing blog post by clicking on the edit button on the "card" for the post. This will display the Edit Post page of the application with the existing post information displayed in the form.
    - Create a new blog post by clicking on the add button on the nav bar on the home page. This will display the Edit Post page of the application with blank values.
    - Edit his/her/their user profile by clicking on the user picture on the nav bar on the home screen. This will display the Edit User Profile page of the application with the existing profile information displayed in the form.

Setting Up

1. Use create-react-app to create a template for the application.
2. Add axios and bootstrap to the application. You might also want to use react-icons and/or react-bootstrap-icons to add icons to the NavBar. Import bootstrap into the index.js file in the usual way.
3. Add json-server to the application.
4. Make the following technical changes to your package.json file:
    a. devDependencies should contain `"@babel/plugin-proposal-private-property-in-object": "^7.21.11".` Run `npm install` to add this node module to your project.

b. test in the scripts section should be replaced with `"test": "react-scripts test --transformIgnorePatterns \"node_modules/(?!axios)/\""`

c. add apiServer to the scripts section `"apiServer": "json-server --watch db.json --port 5000"`

5. Copy db.json to the main folder for your application from the starting files.
6. Add a .env file to the main folder for your application. It should define the following 2 environment variables.

    a. `PUBLIC_URL=./`

    b. `REACT_APP_SERVER_URL=http://localhost:5000`

7. Copy setupTests.js to the src folder for your application from the starting files.
8. Remove all the unnecessary parts of the generated code.
9. Add a components folder to your project. Add a file for the Header, LoginForm, NavBar, PostCard and PostsList components to the folder. Define a functional component in each of these files that displays some identifying information on the screen as a placeholder for use during the testing process.
10. Add a pages folder to your project. Add a file for the Home page to the folder.
11. Add a context folder to your project. Add a file for the PostsContext and UserContext to the folder. Copy posts.test.js and user.test.js to the folder from the starting files. These 2 files contain unit tests that I have written for each context.
12. Start the application.
13. Start the apiServer.
14. Familiarize yourself with the structure of the categories, users and posts lists in the db.json file. Test the following get requests in a browser and make sure that you understand each URL pattern.

    a. http://localhost:5000/posts?_expand=user&_sort=datetime&_order=desc&_start=0&_end=12

    b. http://localhost:5000/posts?userId=${userId}&_expand=user&_sort=datetime&_order=desc

    c. http://localhost:5000/categories?_sort=name&_order=asc

    d. http://localhost:5000/users?userid=duckdonald&password=password

Creating Context

15. Create the PostsContext and the custom Provider that exposes data and functions in the context to other parts of the application in the posts.js file in the context folder. The context

is very similar to the books context from Modern React and contains the following data and functions:

a. featuredPosts – an array of the 12 most recent posts. This list is read only.

    i. fetchFeaturedPosts – the function that makes a get request to retrieve the 12 most recent post objects and stores them in the state variable. When you're making the call to axios.get, the URL for the json server instance, use process.env.REACT_APP_SERVER_URL rather than hard coding http://localhost:5000. When you move your application to citstudent, this url will access citweb.

b. categories – an array of category objects. This list is read only.

    i. fetchCategories – the function that makes a get request to retrieve the categories objects and stores them in the state variable.

c. posts – an array of post objects for one user. This list will be maintained by the application when the user is logged in.

    i. fetchPosts – the function that makes a get request to retrieve the post objects for the logged in user and stores them in the state variable. It takes one parameter, an integer, that represents the id of the logged in user.

    ii. deletePostById - the function that makes a delete request to delete a post object. It takes 1 parameter, an integer that represents the id of the post. It updates the posts state variable in memory after deleting the individual post in the db.json file.

    iii. editPostById – the function that makes a put request to update a post object. It takes 2 parameters, an integer that represents the id of the post and a json object that includes all of the properties of the post excluding the id. It updates the posts state variable in memory after updating the individual post in the db.json file.

    iv. createPost - the function that makes a post request to add a post object. It takes 2 parameters, a json object that includes all of the properties of the post excluding the id and a json object representing the user who created the post. The id will be added automatically as a result of the post request. It adds the user as a property to the post object and then updates the posts state variable in memory after inserting the individual post in the db.json file.

16. Create the UserContext and the custom Provider that exposes data and functions in the context to other parts of the application in the user.js file in the context folder. The context contains the following data and functions:

    a. user – an object that contains the properties of the logged in user.

        i. fetchUser – the function that makes a get request to retrieve the user object and stores it in the state variable. It takes 2 parameters, both strings, that

represent the userid and password of the user who is trying to log in. When the get request does not return an array of exactly 1 user object, the user state variable should be set to null. You might want to return the user object for use "synchronously" in your application when the user logs in.

ii. editUserById – the function that makes a put request to update a user object. It takes 2 parameters, an integer that represents the id of the user and a json object that includes all of the properties of the user excluding the id. It updates the user state variable in memory after updating the individual user in the db.json file.

iii. createUser - the function that makes a post request to add a user object. It takes 1 parameter, a json object that includes all of the properties of the user excluding the id. The id will be added automatically as a result of the post request. It updates the user state variable in memory after inserting the individual user in the db.json file.

iv. resetUser – a function that sets the user state variable to null. This function is used when the user logs out. It does not interact with json-server in any way.

17. Test each of the functions exposed by the custom context provider using the post.test.js file provided. Testing a function involves uncommenting the test component and the test for the function you're testing in the file and running npm run test in another console window. Begin by running the test for each fetch function in BOTH contexts. Run each of the tests that edit data in the db.json file with the db.json file open in your editor and undo the change to the file between each test. I have created a screencast to walk you through testing several functions. You MAY have to edit some test data if I tested with users or posts that are not in the version of db.json on your machine.

18. Update index.js to make both custom context providers available to all of the components in the application. "Wrap" or nest the App component in each provider. The order of the providers does not matter.

Creating the Home Page

19. Finish the App component.
    a. It should call fetchFeaturedPosts and fetchCategories once on first render.
    b. It should render the NavBar component and then the Home page component.

20. Finish the Home page component.
    a. It should get the user from the user context.
    b. It should display the Header component followed by a heading element that says either My Posts or Featured Posts depending on whether a user is logged in. It should also display the PostsList component.

21. Finish the Header component. It should display an appropriate image across the top of the page.
22. Create a first version of the PostsList component.
    a. It should get the user from the user context and the posts and featured posts from the posts context.
    b. It should declare variable postsToRender and set it to either posts or featured posts depending on whether a user is logged in.
    c. It should use a console.log to display postsToRender.
23. Create the NavBar component.
    a. It should get the user as well as the reset user function from the user context.
    b. Because it will render the login form when the user clicks the login button, its interaction with the LoginForm component will be similar to BookShow's interaction with BookEdit in the books application. It should
        i. Define a state variable showLogin and a corresponding setter function. showLogin should be initialized to false.
        ii. Define a function expression handleClick that fires when the user clicks the login buton. When the showLogin state variable is false and there is no logged in user, set the showLogin state variable to true. When the showLogin state variable is already true, set it to false. Otherwise, call the resetUser function.
        iii. Define a handleLoginSubmit function expression that will get passed as a prop to the LoginForm component. It should set the showLogin state variable to false.
    c. It should render a navbar with:
        i. A brand or logo that links to the root of the application.
        ii. A new post button or icon that is visible only when a user is logged in. Eventually this will link to the NewPost page but set its href attribute to # in this version.
        iii. An update user profile button or icon that is visible only when a user is logged in. Eventually this will link to the UpdateUserProfile page but set its href attribute to # in this version.
        iv. A button or link that shows login or logout depending on whether or not a user is logged in. It should have an onClick event handler that calls the handleClick function.
    d. It should also render the Login form when the showLogin state variable is set to true. The LoginForm should have a prop called onSubmit that references the handleLoginSubmit function expression.

e. Test. When the app loads you should see the NavBar and the HomePage (including the Header as well as a heading that says Featured Posts and the PostsList component) on the screen. In the console, you should see a list of featured posts. You should be able to click the login button and see the LoginForm on the screen as well.

24. Create the LoginForm component.

a. It should define 3 state variables for the userid, password and error message.

b. It should access fetchUser from the user context and fetchPosts from the posts context.

c. It should define a function expression handleSubmit. It must be async. It should

i. Prevent the default form submit behavior.

ii. Call fetchUser passing userid and password as its arguments. Await the return. It will be either a user object or null.

iii. When the return value is null, set the error state variable to true.

iv. When the return value is not null, call fetchPosts passing the id of the user object as its argument. Then set the userid and password state variables to an empty string and call the function passed in to the LoginForm component as the onSubmit prop (which will close the login form).

d. It should render a form.

i. The onSubmit handler for the form should call a function expression handleSubmit.

ii. The form should include div tag that displays an error message when the state variable error is set to true.

iii. The form should include a controlled input element for the userid. You can either define an onChange handler inline in the control.

```
<input id="userid" type="text" value={userid}
      onChange={(event)=> {setUserid(event.target.value);}}
      placeholder="user id" />
```

iv. The form should include a controlled input element for the password.

v. The form should include a submit button.

e. Test. You should be able to log in using the userid and password for one of the users from the db.json file. duckdonald and password should be a valid login. The heading on the Home page should change to My Posts and the console should display just the posts for donald duck.

25. Finish the PostsList component.

a. It should declare a variable renderedPosts and set it equal to a call to the map function on the postsToRender array. For each post a PostCard component should be created passing the id of the post as the key prop and the post object as the post prop.

b. It should render a div element containing the value in the variable renderedPosts.

c. Test. You should see several PostCard components on the Home page. When a user logs in successfully, the number of PostCard components should be the same as the number of posts for that user in the db.json file. Remove the console.log statement that displays the postsToRender.

26. Finish the PostCard component.

    a. It should receive a value called post as one of its props.

    b. It should get deletePostById from the posts context and user from the user context.

    c. It should define a function expression handleDeleteClick. The body of the function should call deletePostById passing the id of the post as its argument.

    d. It should declare a variable date and assign it to a new Date object created from the datetime property of the post.

    e. It should render div styled as a bootstrap card that contains a summary of the post.

        i. It might include: the image, category, title, a portion of the content, the name of the user who created the post as well as the date.

            1. Install html-react-parser to handle post content that contains html. In your code, import parse from 'html-react-parser' and call parse to render the content. I chose to display just the first 100 characters of the content in the card BUT when the content contains html, the text is less than 100 characters. That's fine.

            2. The image is stored in the db.json file as a base64 string. You can use an img element to display the image. Its source attribute will look something like: `src={` ${post.image}`}`

        ii. It must include a link that allows a user to delete the post only if the post was created by the logged in user. This link should call handleDeleteClick.

        iii. It must include a link that allows a user to edit a post only if the post was created by the logged in user. Eventually this link will display the EditPost page but set its href attribute to # for this version.

        iv. It must include a link that allows the user to view the entire post. Eventually this link will display the Post page but set its href attribute to # for this version.

    f. Test the functionality of the Home page of the application.

27. Create a production version of the app. Deploy the production version to citstudent.
    a. You already have .env file for the project.
    b. Change the value of the environment variable REACT_APP_SERVER_URL to use one of the free ports assigned to you on citweb.lanecc.net.
    c. Use the instructions in moodle to set up an instance of json-server running on that port on citweb. (You set up another instance of json-server running on citweb for the books application earlier in this lab.)
    d. Create a production version of the app in the usual way.
    e. Test the application from the build folder on your machine. This version will be using the instance of json-server running on citweb.
    f. Deploy the contents of the build folder to citstudent and test the application again.