CS142 Project 5: Single Page Applications

Due: Thursday, May 12, 2022 at 11:59 PM

In this project you will use ReactJS with Material-UI to create the beginnings of a photo-sharing web application. In the second half of this project, you'll also explore retrieving data from a server.

Setup

You should already have installed Node.js and the npm package manager to your system. If not, follow the installation instructions now.

Create a directory `project5` and extract the contents of this zip file into the directory. The zip file contains the starter files for this assignment.

This assignment requires many node modules that contain the tools (e.g. Webpack, Babel, ESLint) needed to build a ReactJS web application as well as a simple Node.js web server (ExpressJS) to serve it to your browser. It also fetches Material-UI which contain the React components and style sheets we will be using. These modules can be fetched by running the following command in the `project5` directory:

```
npm install
```

ReactJS and Material-UI are fetched into the `node_modules` subdirectory even though we will be loading it into the browser rather than Node.js.

Like the previous assignment, we can use npm to run the various tools we had it fetch. The following npm scripts are available in the `package.json` file:

- `npm run lint` - Runs ESLint on all the project's JavaScript files. The code you submit should run ESLint without warnings.
- `npm run build` - Runs Webpack using the configuration file `webpack.config.js` to package all of the projects JSX files into a single JavaScipt bundle in the directory `compiled`.
- `npm run build:w` - Runs Webpack like the "run build" command except it invokes webpack with `--watch` so it will monitor the React components and regenerates the bundle if any of them change.

Your solutions for all of the problems below should be implemented in the `project5` directory. As was done with on the previous project you will need to run a web server we provide for you by running a command in your `project5` directory:

```
node webServer.js
```

As in the last project, you can use the command:

```
node webServer.js & npm run build:w
```

to run the web server and webpack within a single command line window.

Problem 1: Create the Photo Sharing Application (40 points)

As starter code for your PhotoApp we provide you a skeleton (`photo-share.html` which loads `photoShare.jsx`) that can be started using the URL `"http://localhost:3000/photo-share.html"`. The skeleton:

- Loads a ReactJS web application that uses Material-UI to layout a Master-Detail pattern as described in class. It has a header made from a Material-UI App Bar accross the top, places a `UserList` component along the side, and has a content area beside it with either a `UserDetail` or `UserPhotos` components.
- Uses the React Router to enable deep linking for our single page application by configuring routes to three stubbed out components:
  1. `/users` is routed to the component `UserList` in `components/userList/`
  2. `/users/:userId` is routed to the component `UserDetail` in `components/userDetail/`
  3. `/photos/:userId` is routed to the component `UserPhotos` in `components/userPhotos/`

  See the use of HashRouter, and Route in `photoShare.jsx` for details. For the stubbed out components in `components/*`, we provide an empty CSS file and a simple render function that includes some description of what it needs to do and the model data to use.

For this problem, we will continue to use our magic `cs142models` hack to provide the model data so we display a pre-entered set of information. As before, the models can be accessed using `window.cs142models`. The schema of the model data is defined below.

Your assignment is to extend the skeleton into a working web app operating on the fake model data. Since the skeleton is already wired to either display components `UserList`, `UserDetail`, and `UserPhotos` with the appropriate parameters passed by React Router, most of the work will be implementing the stubbed out components. They should be filled in so that:

- `components/userList` component should provide navigation to the user details of all the users in the system. The component is embedded in the side bar and should provide a list of user names so that when a name is clicked, the content view area switches to display the details of that user.
- `components/userDetail` component is passed a `userId` in the `props.match` by React Router. The view should display the details of the user in a pleasing way along with a link to switch the view area to the photos of the user using the `UserPhotos` component.
- `components/userPhotos` component is passed a `userId`, and should display all the photos of the specified user. It must display all of the photos belonging to that user. For each photo you must display the photo itself, the creation date/time for the photo, and all of the comments for that photo. For each comment you must display the date/time when the comment was created, the name of the user who created the comment, and the text of the comment. The creator for each comment should be a link that can be clicked to switch to the user detail page for that user.

Besides these components, you need to update the `TopBar` component in `components/topBar` as follows:

- The left side of the `TopBar` should have your name.
- The right side of the `TopBar` should provide app context by reflecting what is being shown in the main content region. For example, if the main content is displaying details on a user the toolbar should have the user's name. If it is displaying a user's photos it should say "Photos of " and the user's name.

The use of ReactRouter in the skeleton we provide allows for deep-linking to the different views of the application. Make sure the components you build do not break this capability. It should be possible to do a browser refresh on any view and have it come back as before. Our standard approach to building components handles deep-linking automatically. Care must be taken when doing things like sharing objects between components. A quick browser refresh test on each view will show when you broke something.

Although you don't need to spend a lot of time on the appearance of the app, it should be neat and understandable. The information layout should be clean (e.g., it should be clear which photo each comment applies to).

Photo App Model Data

For this problem we keep the magic DOM loaded model data we used in the previous project. The model consists of four types of objects: `user`, `photo`, `comment`, and `SchemaInfo` types.

- Photos in the photo-sharing site are organized by user. We will represent users as an object `user` with the following properties:
  - _id: The ID of this user.
  - first_name: First name of the user.
  - last_name: Last name of the user.
  - location: Location of the user.
  - description: A brief user description.
  - occupation: Occupation of the user.

  The DOM function `window.cs142models.userModel(user_id)` returns the `user` object of the user with id `user_id`. The DOM function `window.cs142models.userListModel()` returns an array with all `user` objects, one for each the users of the app.
- Each user can upload multiple photos. We represent each photo by a `photo` object with the following properties:
  - _id: The ID for this photo.
  - user_id: The ID of the `user` who created the photo.
  - date_time: The date and time when the photo was added to the database.
  - file_name: Name of a file containing the actual photo (in the directory `project5/images`).
  - comments: An array of the `comment` objects representing the comments made on this photo.

  The DOM function `window.cs142models.photoOfUserModel(user_id)` returns an array of the `photo` objects belonging to the user with id `user_id`.
- For each photo there can be multiple comments (any user can comment on any photo). `comment` objects have the following properties:
  - _id: The ID for this comment.
  - photo_id: The ID of the `photo` to which this comment belongs.
  - user: The `user` object of the user who created the comment.
  - date_time The date and time when the comment was created.
  - comment The text of the comment.
- For testing purposes we have `SchemaInfo` objects have the following properties:
  - _id: The ID for this SchemaInfo.
  - __v: Version number of the SchemaInfo object.
  - load_date_time:The date and time when the SchemaInfo was loaded. A string.

Problem 2: Fetch model data from the web server (20 points)

After doing Problem 1, our photo sharing app front-end is looking like a real web application. The big barrier to be considered real is the fakery we are doing with the model data loaded as JavaScript into the DOM. In this Problem we remove this hack and have the app fetch models from the web server as would typically be done in a real application.

The `webServer.js` given out with this project reads in the `cs142Models` we were loading into the DOM in Problem 1 and makes them available using ExpressJS routes. The API exported by `webServer.js` uses HTTP GET requests to particular URLs to return the `cs142Models` models. The HTTP response to these GET requests is encoded in JSON. The API is:

- `/test/info` - Returns `cs142models.schemaInfo()`. This URL is useful for testing your model fetching method.
- `/user/list` - Returns `cs142models.userListModel()`.
- `/user/:id` - Returns `cs142models.userModel(id)`.
- `/photosOfUser/:id` - Returns `cs142models.photoOfUserModel(id)`.

You can see the APIs in action by pointing your browser at above URLs. For example, the links `"http://localhost:3000/test/info"` and `"http://localhost:3000/user/list"` will return the JSON-encoded model data in the browser's window.

To convert your app to fetch models from the web server you should implement a FetchModel function in `lib/fetchModelData.js`. The function should be declared as follows:

```
/*
 * FetchModel - Fetch a model from the web server.
 *   url - string - The URL to issue the GET request.
 * Returns: a Promise that should be filled
 * with the response of the GET request parsed
 * as a JSON object and returned in the property
 * named "data" of an object.
 * If the requests has an error the promise should be
 * rejected with an object contain the properties:
 *    status:  The HTTP response status
 *    statusText:  The statusText from the xhr request
 */
```

Although there many modules that would make implementing this function trivial, we want you to learn about the low-level details of AJAX. You may not use other libraries to implement FetchModel; you must write Javascript code that creates XMLHttpRequest DOM objects and responds to their events.

Your solution needs to be able to handle multiple outstanding FetchModel requests. To demonstrate your FetchModel routine works, your web application should work so that visiting `http://localhost:3000/photo-share.html` displays the version number returned by sending an AJAX request to the `http://localhost:3000/test/info` URL. The version number should be displayed in the `TopBar` component of your app.

After successfully implementing the FetchModel function in `lib/fetchModelData.js`, you should modify the code in

- `components/userDetail/UserDetail.jsx`
- `components/userList/UserList.jsx`
- `components/userPhotos/UserPhotos.jsx`

to use the FetchModel function to request the data from the server. There should be no accesses to `window.cs142models` in your code and your app should work without the line in `photo-share.html`:

```
<script src="modelData/photoApp.js"><script>
```

Style Points (5 points)

These points will be awarded if your problem solutions have proper MVC decomposition. In addition, your code and components must be clean and readable, and your app must be at least "reasonably nice" in appearance and convenience.

Note that we are using Material-UI, React components that implement Google's Material Design. We have used Material-UI's Grid component to layout the Master-Detail pattern as described in class, and a App Bar header for you. Although you don't need to build a fully Material Design compatible app, you should use Material-UI components when possible.

In addition, remember to run ESLint before submitting. ESLint should raise no errors.

Extra Credit (5 points)

The `userPhotos` component specifies that the display should include all of a user's photos along the photos' comments. This approach doesn't work well for users with a large numbers of photos. For extra credit you can implement a photo viewer that only shows one photo at a time (along with the photo's comments) and provides a mechanism to step forward or backward through the user's photos (i.e. a stepper).

In order to get extra credit on this assignment your solution must:

- Introduce the concept of "advanced features" to your photo app. On app startup "advanced features" is always disabled. The toolbar on the app should have a checkbox labelled "Enable Advanced Features" that displays the current state of "advanced features" (checked meaning advanced features is enabled) and supports changing the enable/disable state of the advanced features.
- Your app should use the original photo view unless the "advanced features" have been enabled by the checkbox. If enabled, viewing the photos of a user should use the single photo with stepper functionality.
- The user interface for stepping should be something obvious and the mechanism should indicate (e.g. a disabled button) if stepping is not possible in a direction because the user is at the first (for backward stepping) or last photo (for forward stepping).
- Your app should allow individual photos to be bookmarked and shared by copying the URL from the browser location bar. The browser's forward and back buttons should do what would be expected. When entering the app using a deep linked URL to individual photos the stepper functionality should operate as expected.

Warning: Doing this extra credit involves touching various pieces used in the non-extra credit part of the assignment. Adding new functionality guarded by a *feature flag* is common practice in web applications but has a risk in that if you break the non-extra credit part of the assignment you can lose more points than you could get from the extra credit. Take care.

Deliverables

Use the standard class submission mechanism to submit the entire application (everything in the `project5` directory). Please clean up your project directory before submitting, as described in the instructions. In particular, the node_modules directory is 180MB in size so you won't want to upload it to Canvas.