# Lab 2: Single Page Applications

In this lab you will use [ReactJS](https://reactjs.org/) with [Material-UI](https://mui.com/) to create the beginnings of a photo-sharing web application. In the second half of this lab, 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 on Node.js website.

Create a directory photo-sharing-v1 and extract the contents of the zip file into the directory. The zip file contains the starter files for this lab. Your solutions for all of the problems below should be implemented in the photo-sharing-v1 directory.

Because the photo-sharing-v1 folder doesn’t have node modules installed in it, you need to run the npm install command in the photo-sharing-v1 directory to fetch them into the subdirectory node\_modules.

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

As starter code for your PhotoApp we provide you a skeleton that can be started using the URL [http://localhost:3000](http://localhost:3000/photo-share.html). The skeleton:

* Loads a [ReactJS](https://reactjs.org/) web application that uses [Material-UI](https://mui.com/) to layout a Master-Detail pattern as described in app. It has a header made from a [Material-UI App Bar](https://mui.com/components/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 component.
* Uses the [React Router](https://v5.reactrouter.com/) 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 [Router](https://v5.reactrouter.com/web/api/HashRouter), and [Route](https://v5.reactrouter.com/web/api/Route) in App.js 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 use our magic models hack to provide the model data so we display a pre-entered set of information. As before, the models can be accessed by importing the PhotoApp component. 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 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. Note: The date/time for photos and comments should be formatted as user-friendly strings and not as raw JavaScript dates.

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 TopBar 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 React Router 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

Model data is typically fetched from the webserver which retrieves the data from a database. To avoid having to set up a database for this lab we will give you a hardcode model data in a ReactJS component in /modelData folder. 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 function models.userModel(user\_id) returns the user object of the user with id user\_id. The function models.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 photo-sharing-v1/images). |
| comments: | An array of the comment objects representing the comments made on this photo. |

* The function models.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 object. In Problem 2, we remove this hack and have the app fetch models from the web server as would typically be done in a real application.

The API exported by server uses HTTP GET requests to particular URLs to return the model data. The HTTP response to these GET requests is encoded in JSON. The API is:

* /test/info - Returns models.schemaInfo(). This URL is useful for testing your model fetching method.
* /user/list - Returns models.userListModel().
* /user/:id - Returns models.userModel(id).
* /photosOfUser/:id - Returns models.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> wiil 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.

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

* components/UserDetail/index.jsx
* components/UserList/index.jsx
* components/UserPhotos/index.jsx

to use the fetchModel function to request the data from the server.

### 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](https://mui.com/), React components that implement Google's [Material Design](https://material.io/). We have used Material-UI's [Grid component](https://mui.com/components/grid/) to layout the Master-Detail pattern as described in class, and an [App Bar](https://mui.com/components/app-bar/) header for you. Although you don't need to build a fully Material Design compatible app, you should use [Material-UI](https://mui.com/) 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 TopBar should have a checkbox labeled "Enable Advanced Features" that is checked when "Advanced Features" is enabled and unchecked otherwise. Clicking the checkbox should enable/disable "Advanced Features."
* Your app should use the original photo view unless "Advanced Features" has 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.