Project 6: Complete the app

In this project, you will extend your work on Project 5 by adding the ability for users to login, comment on photos, and upload new photos. Note that these new feature additions are **full stack** in that you will need to modify both the front end (React app) and the back end (Node web server and MongoDB database).

Problem 1: Simple Login (15 points)

Extend your photo app to have the notion of a user being logged in. If a user is logged in, the toolbar should include a small message "Hi <firstname>" where <firstname> is the first name of the logged-in user. The toolbar should also contain a button displaying "Logout" that will log the user out.

If there is no user logged in, the toolbar should display "Please Login" and the main view of your application should display a new view component named LoginRegister. The LoginRegister view should provide a way for a user to login and, as part of Problem 4 below, register as a new user. All attempts to navigate to a different view (e.g., deep links) should result in the display being redirected to the LoginRegister view if no user is logged in. (See the hints section if you are unsure how to implement this.) In addition, the user list on the left should not be populated if the current user is not logged in. (See the section below about modifying the server endpoints to return a status of 401 (Unauthorized).)

When a user logs in successfully, the view should switch to displaying the user's details. If the user login fails (e.g., no user with the login\_name) the view should report an appropriate error message and let the user try again.

Extend your backend implementation to support the photo app's notion of logged in users. In making this change you will need to change both the database schema and the web server API.

Extend the Mongoose schema for User to add a new property login\_name. This property is a string containing the identifier the user will type when logging in (their "login name").

Modify the web server API to support 2 new REST API calls for logging a user in and out. Like in the previous assignment we will use HTTP requests with JSON-encoded bodies to transmit model data. The API uses POST requests to:

* /admin/login - Provides a way for the photo app's LoginRegister view to login in a user. The POST request JSON-encoded body should include a property login\_name (no passwords for now) and reply with information needed by your app for logged in user. An HTTP status of 400 (Bad request) should be returned if the login failed (e.g., login\_name is not a valid account). A parameter in the request body is accessed using request.body.parameter\_name. Note the login register handler should ensure that there exists a user with the given login\_name. If so, it stores some information in the Express session (or generate a JWT token if using token-based authentication) where it can be checked by other request handlers that need to know whether a user is logged in. The return body of POST /admin/login should be the properties of the logged in user your web app needs. Our tests require the \_id property but you can include any other user object properties. Returning the entire user object is discouraged because of what is added in Problem 4.
* /admin/logout - A POST request with an empty body to this URL will logout the user by clearing the information stored in the session (or delete the JWT token on the client if using token-based authentication) . An HTTP status of 400 (Bad request) should be returned in the user is not currently logged in.

As part of updating the web server to handle login/logout, you need to update all requests (except to /admin/login and /admin/logout) to reject the request with a status of 401 (Unauthorized) if the session state does not report a user is logged in (or does not have valid token in request header if using token-based authentication).

Problem 2: New Comments (15 points)

Once you have implemented user login, the next step is to implement the ability to add comments to photos. In the photo detail view where you display the comments of a photo, add the ability for the currently logged in user to add a comment to the photo. You get to design the user interface (e.g., popup dialog, input field, etc.) for this feature. It should be obvious how to use it and what photo the comment is about. The display of the photo and its comments should be updated immediately to reflect the newly added comment.

For the backend support extend the web server API with the following HTTP POST API:

* /commentsOfPhoto/:photo\_id - Add a comment to the photo whose id is photo\_id. The body of the POST requests should be a JSON-encoded body with a single property comment that contains the comment's text. The comment object created on the photo must include the identifier of the logged in user and the time when the comment was created. Your implementation should reject any empty comments with a status of 400 (Bad request).

Problem 3: Photo Uploading (15 points)

Allow users to add new photos. When a user is logged in, the main toolbar should have a button labeled "Add Photo" that allows the current logged in user to upload a photo to the app. When a photo is successfully posted, it should show up automatically on the user photos page, or at the very least it should have some indication that the photo was added.

Extend the web server to support POST requests to the URL:

* /photos/new - Upload a photo for the current user. The body of the POST request should be the file. The uploaded files should be placed in the images directory under an unique name you generated. The unique name, along with the creation data and logged in user id, should be placed in the new Photo object you create. A response status of 400 should be returned if there is no file in the POST request.

Problem 4: Registration and Passwords (15 points)

Enhance the LoginRegister view component to support new-user registration and passwords. Extend the login portion to add a password field. Add a registration section that allows all the fields of the User object to be filled in. To reduce the chance that the user typos that password the view should contain a an additional copy of the password field and the view should only allow the user to be created if the two password fields are identical. Good security practice requires that the passwords typed by the user shouldn't be visible in the view. Registration should be triggered by a button at the bottom of the page labeled "Register Me". When the button is pushed either an error should be reported explaining **specifically** why it didn't work or a success message should be reported and the register form input fields should be cleared.

For the backend, extend the User object schema with a string field password that will store the password.

Extend the web server to support POST requests to the URL:

* /user - Allows a user to register. The registration POST takes a JSON-encoded body with the following properties: (login\_name, password, first\_name, last\_name, location, description, occupation). The post request handler must make sure that the new login\_name is specified and doesn't already exist. The first\_name, last\_name, and password must be non-empty strings as well (the other fields may be empty). If the information is valid, then a new user is created in the database. The response body should be the properties your web app needs. Our tests require the login\_name property be present. If there is an error, the response should return status 400 and a string indicating the error.

Enhance the LoginRegister view to support logging in with a password and check it as part of the post request to /admin/login.