CS142 Project 7: Sessions and Input

Due: Thursday, May 26, 2022 at 11:59 PM

In this project, you will extend your work on Project #6 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).

Setup

You should have MongoDB and Node.js installed on your system. If not, follow the installation instructions now.

Like in Project #6, you start by copying the files of your previous project's directory (`project6`) into a new directory named `project7` (**make sure that any hidden files are transferred as well, including the .babelrc and .eslintrc.json files**). Extract the contents of this zip file into the `project7` directory. This zip file will add several new files. Do all of your work for this project in the `project7` directory.

In order to do this assignment you need to fetch some new node modules and add them to your web server and React application. To fetch the module use the command `npm install --save <module>` which will fetch the module name `<module>` into your `node_modules` directory and add the dependency to your `package.json` file. Use a `require` function to add the module to your web server.

You will need to fetch the following Express middleware modules into your node modules:

- `express-session` - Express session is an Express middleware layer than handles session management for you. It was described in lecture.
- `body-parser` - `body-parser` is an Express middleware layer for parsing the body of HTTP requests. You can use it to parse the JSON encoded POST request bodies we use in our server API. For example, if you pass a request with a body consisting of JSON object with a property `parameter_name` it will show up in the Express request handler as `request.body.parameter_name`.
- `multer` - `multer` is another Express middleware body parser that is capable of handling the multi part forms we need to upload photos.

On most systems the following command should fetch the above modules:

```
npm  install --save express-session
npm  install --save body-parser
npm  install --save multer
```

Add these to your `webServer.js` using require statements like:

```
const session = require('express-session');
const bodyParser = require('body-parser');
const multer = require('multer');
```

Finally you need to add the `express-session` and `body-parser` middleware to express with the Express `use` like so:

```
app.use(session({secret: 'secretKey', resave: false, saveUninitialized: false}));
app.use(bodyParser.json());
```

where `'secretKey'` is the secret you use to cryptographically protect the session cookie. We will use the `multer` middleware in the photo upload code.

If you did Problem 2 of Project #6, you should be using `axios` to fetch models. See the `axios` documentation for more details if you didn't implement it for project 6.

Like in the previous assignment we provide a Mocha test of the server API so you can have confidence in your backend before you implement the front end code. **In exchange for this nicety, you are limited to implementing the server API we specify. This means making sure that your code passes our provided tests before submitting. See the Testing section further below for details.** If you extended your API while doing the Project #6 Extra Credit, you will need to patch the tests to handle your extended API. If you decide to do the extra credit for this project, you will need to pass additional provided tests as well.

As in the previous project you will need to start your MongoDB instance. Start MongoDB by running command:

```
mongod (the exact arguments depend on where you placed the database)
```

and load the photo app data set by running the command:

```
node loadDatabase.js
```

Note that the version of this command we distribute with Project #7 loads the user object with a login_name (lowercase version of their last_name) and password of "weak". These properties will only appear when you run loadDatabase.js after you added the fields to the schema (in problem 1).

Start the Node.js web server

Once you have the database up and running you will need to start the web server. Although this can be done with the same command as the previous assignments (e.g. `node webServer.js`), it is more convenient to start the web server using a program that will automatically restart it when you change the `webServer.js`. Otherwise, you will spend time restarting the web server after each change you make or wondering why your change didn't work when you forget to restart it.

The command

```
npm install -g nodemon
```

will install a program named nodemon that does this automatic restarting (you may need to run the command with `sudo`). Note that the `-g` flag installs the program globally so that you can run it anywhere. Start your web server with the command from your `project7` directory:

```
nodemon webServer.js
```

The command will run the web server and restart it every time the `webServer.js` file changes. If you have an error (e.g. JavaScript syntax error) that causes the Node to exit, nodemon will wait until you change the `webServer.js` file and then try to restart it. If you want to restart the web server without changing the file you can restart it manually by typing the two character command `rs` at the nodemon command. You will probably want to run this in its own window so you can see the logging messages from your Node.js code.

After updating your Photo Share App with the new files from Project #7 and starting the database and web server make sure the app is still working before continuing on to the assignment.

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 in and out a user. 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 where it can be checked by other request handlers that need to know whether a user is logged in.
- `/admin/logout` - A POST request with an empty body to this URL will logout the user by clearing the information stored in the session. 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.

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 labelled "Add Photo" that allows the current logged in user to upload a photo to the app. We will provide you with an example of how to upload files using HTML in the Hint section below.

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 (see hint below). 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. See the Hint section for help with this.

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 labelled "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. This is horribly insecure. See the Extra Credit below if you can't bring yourself to implement something so insecure.

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

- `/user` to allow 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. If the information is valid, then a new user is created in the database. 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`.

Extra Credit #1: Salted Passwords (5 points)

Enhance the security of your password mechanism by implementing *salting*. The salting mechanism is described in the next few paragraphs. The problem with the clear text password mechanism we implemented for Problem 4 is if someone is able to read the database (for example, a rogue system administrator) they can easily retrieve all of the passwords for all users.

A better approach is to apply a message digest function such as SHA-1 to each password, and store only the message digest in the database. SHA-1 takes a string such as a password as input and produces a 40-character string of hex digits (called a *message digest*) as output. The output has two interesting properties: first, the digest provides a unique signature for the input string (there is no known way to produce two different strings with the same digest); second, given a message digest, there is no known way to produce a string that will generate that digest. When a user sets their password, you must invoke Node crypto package's `createHash` function to generate the SHA-1 digest corresponding to that password, and store only the digest in the database; once this is done you can discard the password. When a user enters a password to login, invoke createHash function to compute the digest, and compare that digest to what is stored in the database. With this approach, you can make sure that a user types the correct password when logging in, but if someone reads the digests from the database they cannot use that information to log in.

However, the approach of the previous paragraph has one remaining flaw. Suppose an attacker gets a copy of the database containing the digests. Since the SHA-1 function is public, they can employ a fast *dictionary attack* to guess common passwords. To do this, the attacker takes each word from a dictionary and computes its digest using SHA-1. Then the attacker checks each digest in the database against the digests in the dictionary (this can be done very quickly by putting all of the dictionary digests in a hash table). If any user has chosen a simple dictionary word as their password, the attacker can guess it quickly.

In order to make dictionary attacks more difficult, you must use password salting. When a user sets their password, compute a random number and concatenate it with the password before computing the SHA-1 digest (the crypto package `randomBytes` function with a length of 8 will return a suitable random number. The random number is called a *salt*. Then store both the salt and the digest in the database. When checking passwords during login, retrieve the salt from the database, concatenate it to the password typed by the user, and compute the digest of this string for comparison with the digest in the database. With this approach an attacker who has gained access to the login database cannot use the simple dictionary attack described above; the digest of a dictionary word would need to include the salt for a particular account, which means that the attacker would need to recompute all of the dictionary digests for every distinct account in the database. This makes dictionary attacks more expensive.

To implement this, remove the `password` property from the User schema and replace it with two new string properties: `password_digest` and `salt`. Update the user register and login to use this mechanism.

You should create and write your implementation in a node module file named `cs142password.js`. This module should export two functions:

```
/*
 * Return a salted and hashed password entry from a
 * clear text password.
 * @param {string} clearTextPassword
 * @return {object} passwordEntry
 * where passwordEntry is an object with two string
 * properties:
 *      salt - The salt used for the password.
 *      hash - The sha1 hash of the password and salt
 */
function makePasswordEntry(clearTextPassword) {
```

and

```
/*
 * Return true if the specified clear text password
 * and salt generates the specified hash.
 * @param {string} hash
 * @param {string} salt
 * @param {string} clearTextPassword
 * @return {boolean}
 */
function doesPasswordMatch(hash, salt, clearTextPassword) {
```

We provide a Mocha test file `test/cs142passwordTest.js` that tests this interface. Please make sure that you pass the tests within this file before submitting. You will need to change the `package.json` scripts test line to `mocha serverApiTest.js sessionInputApiTest.js cs142passwordTest.js` so that running `npm test` runs these tests too. Changing this line also serves as indication that you've done this extra credit. You will also need to update the `loadDatabase.js` script to require `cs142password.js` and use it to generate the correct password properties in the new user objects that the script creates.

Extra Credit #2: Handle Browser Refresh (5 points)

For simplicity in the regular parts of this assignment we allow you to keep the application's session state in JavaScript memory. Although this makes implementation easier it means that a browser refresh causes the application to forget who is logged in.

Extend your application to handle browser refresh like it did before you added the login session support. Your scheme should allow a user to do a browser refresh yet stay logged in. You are free to use whatever implementation techniques you want but they must:

- Maintain backward compatibility with the other parts of this assignment, including the Mocha tests.
- Not mess up the security of the application.
- Work when submitted using the class assignment submission mechanism.

Style Points (5 points)

These points will be awarded if your problem solutions have proper MVC decomposition, follow the software stack conventions, and ESLINT warning-free JavaScript. In addition, your code and templates must be clean and readable, and your Web pages must be at least "reasonably nice" in appearance and convenience.

Testing

For testing, we:

- provide new test file named `test/sessionInputApiTest.js` that provides some test coverage of the new API calls.
- update the existing `test/serverApiTest.js` to login a user and provide the session information on each API call.

Our update overwrites the file `serverApiTest.js` so any changes you added in Project #6 as part of the extra credit will need to be backported. To help with the backporting (or if you're just curious), note that we we added a Cookie header key-value pair to the request options of each http request call. See the test file for details.

Just like project #6, please make sure to pass the provided tests before submitting, as a portion of your grade will depend on how many of these tests you pass.

Note the tests assume that the database has only the objects from `loadDatabase.js`. You should run `loadDatabase.js` before running the test.

Before running any tests you should make sure your `test/node_modules` is up to date by typing the command: `npm install` in the `project7/test` directory. You may get some warning about 2 packages being deprecated (namely `request` and `har-validator`). You can safely ignore those as these are warnings caused by the fact that one of the libraries we use for testing is currently in maintenance. As before you can run the tests with the command `npm run test`.

Hints

Now that you are writing code that updates the MongoDB database, bugs in your code can corrupt the database contents. Rerunning the `loadDatabase.js` program will reset your database to a clean state. The tests assume you have done this before each run.

Problem 1

This problem requires you to get the notion of a logged in user in both the React application and in the web server. You will need to get your pattern down for generating POST requests from your app to the web server and setup up the session state. The server-side functionality has tests you can run against it.

Assuming you got your LoginRegister component written, added to your photo app, and routed to using the path "/login-register", we need to make that view be the one displayed if the user is not logged in. React Router provides a useful way of handeling redirects: React Router Redirect. You can modify your Route so that if the user is not logged in, it redirects to /login-register. For example, if you have a Route like:

```
<Route path="/users/:id" component={UserDetail} />
```

and a component property this.userIsLoggedIn, you can have it go to /login-register by wrapping it in a conditional like:

```
{
    this.userIsLoggedIn ?
      <Route path="/users/:id" component={UserDetail} />
    :
      <Redirect path="/users/:id" to="/login-register" />
}
```

Since Problem 4 enhances the code you write for Problem 1 you might want to understand its needs before building your Problem 1 solution.

Problem 2

This problem will reuse most of what you needed to figure out for Problem 1. One difference is comments are not standalone objects in our schema. They are embedded inside the photo object so you are required to perform an object update rather than an object creation.

Problem 3

Uploading files is hard in JavaScript frameworks since the browsers don't want JavaScript code to be able to read arbitrary files. Browsers do provide an interface to allow the user to select a file and then submit its contents to a web server. We can leverage that mechanism to allow our photo app to upload photos.

React supports using HTML's input with type="file" to have the user select files. The following line added to your photo app

```
<input type="file" accept="image/*" ref={(domFileRef) => { this.uploadInput = domFileRef; }} />
```

will get the browser to add an ugly button labelled "Choose File" that the user can push to select a local file. When the user selects a file this.uploadInput will contain the DOM FileList from the input element from which the DOM File can be taken and sent to the web server. You will need to also have a button that the user presses to actually submit the photo file to the server as decribed below.

Once we have the selected file in the DOM we can add it to a DOM form and send it in a POST request to the web server. The following code provides an example of doing this. It assumes that the DOM fileList from the input element was put in this.uploadInput.

```
    //this function is called when user presses the update button
    handleUploadButtonClicked = (e) => {
       e.preventDefault();
       if (this.uploadInput.files.length > 0) {

        // Create a DOM form and add the file to it under the name uploadedphoto
        const domForm = new FormData();
        domForm.append('uploadedphoto', this.uploadInput.files[0]);
        axios.post('/photos/new', domForm)
          .then((res) => {
            console.log(res);
          })
          .catch(err => console.log(`POST ERR: ${err}`));
    }
  }
```

Express using body-parser can not handle a POST request with form containing a file but can with a middleware named `multer`. Insert the following line after the require of multer:

```
const processFormBody = multer({storage: multer.memoryStorage()}).single('uploadedphoto');
```

`processFormBody` is a function we can use in our post request handler for `/photos/new`. `processFormBody` will look at the form for a field named "uploadedphoto" and pull the file out of it and place the information is a property named `file` on the request object. The following code gives you an idea of how to call it in your post request handler:

```
processFormBody(request, response, function (err) {
    if (err || !request.file) {
        // XXX -  Insert error handling code here.
        return;
    }
    // request.file has the following properties of interest
    //      fieldname      - Should be 'uploadedphoto' since that is what we sent
    //      originalname:  - The name of the file the user uploaded
    //      mimetype:      - The mimetype of the image (e.g. 'image/jpeg',  'image/png')
    //      buffer:        - A node Buffer containing the contents of the file
    //      size:          - The size of the file in bytes

    // XXX - Do some validation here.
    // We need to create the file in the directory "images" under an unique name. We make
    // the original file name unique by adding a unique prefix with a timestamp.
    const timestamp = new Date().valueOf();
    const filename = 'U' +  String(timestamp) + request.file.originalname;

    fs.writeFile("./images/" + filename, request.file.buffer, function (err) {
      // XXX - Once you have the file written into your images directory under the name
      // filename you can create the Photo object in the database
    });
});
```

The above code fragment uses the Node.js package `fs` so you will need to bring it into your webServer.js with:

```
const fs = require("fs");
```

Problem 4

This problem combines the techniques you needed for the previous problems.

Deliverables

Use the standard class submission mechanism to submit the entire application (everything in the `project7` directory). Please clean up your project directory before submitting, as described in the submission instructions. In addition delete any images you uploaded into your images directory. If you kept the same naming convention for uploaded images that we had in the hints section you can delete them running the command: `rm -f images/U*` from your project7 directory.