CS142: Section 7

Sessions, Input, and Validation

Project Overview

0. Setup
1. Logging in (sessions)
2. Adding new comments (input)
3. Photo uploading (input)
4. Registration and passwords (sessions)

Setup

Unlike previous assignments: need to add some code fragments not just code files!

Make sure to copy files from project #6 and THEN replace with new project #7 zip files

You will then need to:

- Use npm to install need packages: `npm --save install <moduleName>`
- Add packages to webServer.js: `var module = require('moduleName');`
- Add Express middleware to webServer.js: `app.use(middleWare);`

Setup directions from detailed instructions

WARNING: Follow the setup steps for the project CAREFULLY! There is a lot of setup this time around and you don't want to be stuck debugging for hours because you forgot to add a line somewhere from the setup section

Setup

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:

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

middleware?!: https://hackernoon.com/middleware-the-core-of-node-js-apps-ab01fee39200

Project Overview - Full Stack

Each problem requires doing something on both:

The Node.js and MongoDB backend

Files: `webServer.js` and `schema/*`

The React app front end

Files: `photoShare.jsx, components/*`)

Do the backend (including getting the tests to run) then do the front end

Read the hints at the bottom of the detailed instructions before beginning

Project Overview - Full Stack

There's a lot of moving parts in this assignment. Make sure to rebuild components & restart the server properly! Commands to run:

To (re)set database contents: node loadDatabase.js (helpful between runs of npm test also!)

For *any* change to *.jsx: `npm run build`

or `npm run build:w`

For *any* change to webServer.js: re-run `node webServer.js`

or `nodemon webServer.js`

Also make sure your mongoDB instance is up and running!

Project Overview - Full Stack

My preferred style:

Run node loadDatabase.js when you want to reset

Have the following three commands **always running** on a terminal when you want to use your app or run tests:

1. `nodemon webServer.js`
2. `npm run build:w` (can combine these two with & in between)
3. `mongod` (lives somewhere else)

Problem 1: Logging in

Client Side

Server Side

What do you think needs to be done?

Problem 1: Logging in

Client Side

New component “`LoginRegister`” displayed if user isn’t logged in (Add in `project7/components`)

Hi <first_name> and logout in the toolbar if a user is logged in (`TopBar.js` and `photoShare.jsx`)

Submit a POST request to `/admin/login` with `login_name` as a parameter (`LoginRegister.jsx`)

Logout is on toolbar (`TopBar.js`)

Server Side

Don’t forget to change the schema (`user.js` in schema folder) - to add what?

Implement `/admin/login` and `/admin/logout` (POST) (`webServer.js, app.post()`)

Maintain session state (create and save information to session on login, destroy on logout)

Once this is done, run “`node loadDatabase.js`” for testing. The login names of each existing user is their lowercase last name (passwords are all “weak”, this will be used for testing problem 4)

Good things to know (Server)

`/admin/login` will be a `POST` request (`app.post` in express)

- You should check if the `login_name` exists, and set the session (`request.session`) here.
- Hint: It will be VERY helpful to store the `login_name` and `user_id` in the session

`/admin/logout` will also be a `POST` request

- Remove references with "`delete`", then call `request.session.destroy(`*`callback`*`)`

Remember: you can access the session object as long as the user is logged in (can access `request.session` properties)

POST vs GET:
https://stackoverflow.com/questions/3477333/what-is-the-difference-between-post-and-get

`var session = require('express-session');`

- ExpressJS has a middleware layer for dealing with the session state
  - Stores a sessionID safely in a cookie
  - Store session state in a session state store
  - Like Rails, handles creation and fetching of session state for your request handlers

- Usage:

```
app.use(session({secret: 'badSecret'}));
```

`secret` is used to cryptographically sign the sessionID cookie

```
app.get('/user/:user_id', function (request, response) …
```

`request.session` is an object you can read or write
(e.g. request.**session**.user_id = …)

Express session usage example

- Login handler route can store into `request.session.user_id`
- All other handlers read `request.session.user_id`
  - If not set error or redirect to login page
  - Otherwise we know who is logged in
- Can put other per-session state in `request.session`
- On logged out you want to destroy the session

```
request.session.destroy(function (err) { } );
```

very helpful description of sessions:
https://nodewebapps.com/2017/06/18/how-do-nodejs-sessions-work/

Requests from non-logged in user

How would we know?

What would we do?

When should we check?

Requests from non-logged in user

How would we know?

Check if session object has the stored properties it should have

What would we do?

When should we check?

Requests from non-logged in user

How would we know?

Check if session object has the stored properties it should have

What would we do?

Return a 401 (Unauthorized) status if this property is not found.

When should we check?

All endpoints which serve sensitive data (users, photos, …).

Good things to know (Client)

Put this code into your photoShare.jsx file to block all attempts to navigate to a different view (while not logged in):

```
<Switch>
     {
         isLoggedIn() ?
           <Route path="/users/:id" component={UserDetail} />
         :
           <Redirect path="/users/:id" to="/login-register" />
     }
     {/* do same for all routes except login-register */}
</Switch>
```

You will need to implement the *isLoggedIn* function yourself. Also check on server side.

Additional Problem 1 Tips

Read the entire assignment before beginning

It may be good to understand problem 4 requirements before deciding on your implementation design for problem 1.

Problem 2: Adding new comments

Client Side

Add a way for users to add comments to photos

Submit a POST request to `/commentsOfPhoto/:photo_id` with the new comment

Server Side

Implement a POST request handler for `/commentsOfPhoto/:photo_id`

We will have to get the corresponding photo (with `id` matching `request.params.photo_id`)

Append the comment to end of comments list (see `photo.js` in schema folder for why we have to do this)

Good things to know (Updating Photo)

How can we get a Photo from the database, and update it?

```
Photo.findOne({_id: photo_id}, function (err, photo) {
    // Update photo object
    photo.save();
});
```

So we can get the photo_id in the URL. How can we get the user_id to save in the comment object? (Hint: session, remember I said something would be VERY helpful a few slides ago?)

Problem 3: Photo uploading

Client Side

Add a way to select and upload a photo from the toolbar. The majority of the code is already provided for you in the hint section.

Server Side

Implement a `POST` request handler for `/photos/new`

All the tedious stuff has been implemented for you. You will need to create a new Photo in the database (this will be done in the `fs.writeFile` function)

MongoDB 3.6 with Mongoose 4.x Problem

Newer versions of MongoDB (3.6) no longer supports the $pushAll(), which was what Mongoose 4.x's array.push() maps to.

So don’t use array.push(element), use: arr = arr.concat([element])

If you are using Mongoose 5.1.0, then you are fine! Yeahhh!

Mongoose: Make Model from Schema

- Creating new database objects
- A **Model** in Mongoose is a constructor of objects
- May or may not correspond to a model of the MVC (in `schema/user.js`)

```
var User = mongoose.model('User', userSchema);
```

- Create objects from Model

```
User.create({ first_name: 'Ian', last_name: 'Malcolm'}, doneCallback);

function doneCallback(err, newUser) {
   assert (!err);
   newUser.save();
   console.log('Created object with ID', newUser._id);
}
```

IMPORTANT for Problems 3 & 4

Don’t forget to **reject** any attempts to access the /commentsOfPhoto/:photo_id and /photos/new endpoints if **there is no logged-in user** (just like we did in problem 1 with /user/list, /user/:id, etc.)

Problem 4: Registration and passwords

Client Side

Add a password field for logging in. As a hint, you can use `<input type=”password”...>` to get a password field (characters are hidden)

Implement a registration mechanism (Users will need to specify a `login_name`, `password`, `first_name`, `last_name`, `occupation`, `location`, and `description`).

Server Side

Update `/admin/login` to also verify the password

Implement a `POST` request handler for `/user`

You will have to check to see if the `login_name` exists.

There are other restrictions (`first_name` and `last_name` can’t be empty). THINK about where (client-side or server-side) you want to implement each restriction check.

Don’t forget to change the schema (`schema/user.js`)

Extra Credit

You’re on your own for this one! If you made it this far, the extra credit shouldn’t be that hard. Salted passwords123, in particular, are quite simple to implement and very important! Make sure to read the instructions carefully!

They’re super interesting. :)