Lab 10

New Attempt

- Due Apr 22 by 11:59pm
- Points 100
- Submitting a file upload
- File Types zip

CS-546 Lab 10

Authentication and Middleware

For this lab, we will be creating a basic server with a user sign-up and user login. You will be storing a user's first name, last name, username, hashed password, favoriteQuote, themePreference ("light" or "dark") and their role ("admin" or "user"). In addition, we will be creating a very basic logging middleware.

We will be using two new npm packages for this lab:

- **bcrypt** (https://www.npmjs.com/package/bcrypt) : a password hasing library. If you have problems installing that modules (since it uses C++ bindings), you can also try **bcrypt.js** (https://www.npmjs.com/package/bcryptjs) , which has the same API but is written in 100% JS.
- **express-session** (https://www.npmjs.com/package/express-session) : a simple session middleware for Express.

YOU MUST use the directory and file structure in the code stub, or points will be deducted. You can download the starter template here: **Lab10_stub.zip** (https://sit.instructure.com/courses/71954/files/13301862?wrap=1) (https://sit.instructure.com/courses/71954/files/13301862/download?download_frd=1)

PLEASE NOTE: THE STUB DOES NOT INCLUDE THE PACKAGE.JSON FILE. YOU WILL NEED TO CREATE IT! DO NOT ADD ANY OTHER FILE OR FOLDER APART FROM PACKAGE.JSON FILE. DO NOT FORGET THE START COMMAND OR THE TYPE MODULE PROPERTY IN THE PACKAGE.JSON

Database Structure

You will use a database with the following structure:

- The database will be called **FirstName_LastName_lab10**
- The collection you use will be called `users`

A sample of the user schema as it will be stored in the database:

```
[
  {
    _id: ObjectId("615f5211445eac188610ecbe"),
```

```
    firstName: 'Patrick',
    lastName: 'Hill',
    username: 'graffixnyc',
    password: '$2b$16$Vm/Xqc.2eyi3y3IqewuhjOTXeoxt4SaN1dcAfPwEPUrzA5Kgm1HFW',
    favoriteQuote: "We have two lives, the 2nd begins when you realize you only have one.",
    themePreference: "light",
    role: 'admin'
  },
  {
    _id: ObjectId("615f5211445eac188610ecc0"),
    firstName: 'Aiden',
    lastName: 'Hill',
    username: 'aidenflash',
    password: '$2b$16$SHQUG43PoIHoTHvkeDBczewvurYf3l.XKMRhrRomB.iVMcvldsq8m',
    favoriteQuote: 'Whatchu talking about Willis?",
    themePreference: "dark"
    role: 'user'
  }
];
```

You will have one data module in your data folder named: `users.js` that will only export two functions:

registerUser(firstName, lastName, username, password, favoriteQuote, themePreference, role)

You must do full input checking and error handling for the inputs in this function.

1. All fields must be supplied or you will throw an error
2. For `firstName`, it should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 2 characters long with a max of 25 characters. If it fails any of those conditions, you will throw an error.
3. For `lastName`, it should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 2 characters long with a max of 25 characters. If it fails any of those conditions, you will throw an error.
4. `username` should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 5 characters long with a max of 10 characters. If it fails any of those conditions, you will throw an error.
5. The `username` should be case-insensitive. So "PHILL", "phill", "Phill" should be treated as the same username.
6. YOU MUST NOT allow duplicate usernames in the system. If the `username` supplied is already in the database you will throw an error stating there is already a user with that username.
7. For the `password`, it must be a valid string (no strings with just spaces and no spaces but can be any other character including special characters) and should be a minimum of 8 characters long. If it fails any of those conditions, you will throw an error. **The constraints for password will be: There needs to be at least one uppercase character, there has to be at least one number and there has to be at least one special character: for example: Not valid: test123, test123$, foobar, tS12$ Valid: Test123$, FooBar123*, HorsePull748*%**
8. For `favoriteQuote`, it should be a valid string (no strings with just spaces) and should be at least 20 characters long with a max of 255 characters. If it fails any of those conditions, you will throw an error.

9. For `themePreference`, the ONLY two valid values are "light" or "dark". Your function should accept it in any case, but it should be stored as lowercase in the DB. If there is any other value supplied, throw an error.
10. For `role`, the ONLY two valid values are "admin" or "user". Your function can accept it in any case, but it should be stored as lowercase in the DB. If there is any other value supplied, throw an error.

In this function you will hash the password using bcrypt. You will then insert the user's first name, last name, username, **hashed** password, favoriteQuote, themePreference, and role into the database.

If the insert was successful, your function will return: `{signupCompleted: true}`.

loginUser(username, password)

You must do full input checking and error handling for the inputs in this function.

1. Both `username` and `password` must be supplied or you will throw an error
2. `username` should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 5 characters long with a max of 10 characters. If it fails any of those conditions, you will throw an error.
3. The `username` should be case-insensitive. So "PHILL", "phill", "Phill" should be treated as the same username.
4. For the `password`, it must be a valid string (no strings with just spaces and no spaces but can be any other character including special characters) and should be a minimum of 8 characters long. If it fails any of those conditions, you will throw an error. The constraints for password will be: There needs to be at least one uppercase character, there has to be at least one number and there has to be at least one special character: for example: Not valid: test123, test123$, foobar, tS12$ Valid: Test123$, FooBar123*, HorsePull748*%

In this function, after you validate the inputs you will:

1. Query the db for the `username` supplied, if it is not found, throw an error stating "Either the username or password is invalid".
2. If the `username` supplied is found in the DB, you will then use bcrypt to compare the hashed password in the database with the `password` input parameter.
3. If the passwords match your function will return the following fields of the user: `firstName`, `lastName`, `username`, `favoriteQuote`, `themePreference, role` which will be stored in the session from the route, **(DO NOT RETURN THE PASSWORD!!!!)**
4. If the passwords do not match, you will throw an error stating "Either the username or password is invalid"

Routes

GET `/`

The root route of the application. This route will never be hit at all. You will use a middleware function (described below) for this route. If the user is authenticated AND they have a role of `admin`, the

middleware function will redirect them to the /admin route, if the user is authenticated AND they have a role of `user`, you will redirect them to the /user route. If the user is NOT authenticated, you will redirect them to the GET /login route. Again, this route will never be accessed really, the middleware function will redirect based on the conditions above and as noted in the middleware section below.

GET `/login`

This route of the application will render a view with a login form. The form will contain two inputs, one for the username and one for the password, and a button to submit the form. The form will be used to submit a POST request to the `/login` route on the server and **must** have an `id` of `signin-form`. The input for the username must have both a `name` **and** `id` attribute of `"username"` and should be an input `type` of `text`. The input for the password must have both a `name` **and** `id` of `"password"` and should be an input `type` of `password` The button will have a `name` AND `id` attribute of `"submitButton"`

You will also have a link on this page that links to `/register` and has the text "Click here to register!"

Do not forget to use labels for your inputs! For the labels, you will use the method of using the `for` attribute referencing the id of the input it belongs to. You should have a label for both the username and password inputs referencing their id's in the `for` attribute of the label

An authenticated user should not ever see the login screen!!!

GET `/register`

This route will render a view with a sign-up form. The form will contain 6 inputs and two select elements (dropdown). For the inputs, you will have one for the `First Name` which is type `text`, one for the `Last Name` which is type `text`, one for the `Username` that is type `text`, one for `Password` with type of `password`, one for `Confirm Password` with type of `password` (you will check to make sure the password and confirm passwords match via client-side validation and in the routes, you do not have to send/check both passwords to the data function), and one input for the Favorite Quote which is type text. For the first select dropdown, you will have two options one that displays `"Light"` with a `value` attribute of `"light"` and one option that displays `"Dark"` with a `value` attribute of `"dark"`. For the 2nd select dropdown, you will have two options one that displays "Admin" with a value attribute of "admin" and one option that displays "User" with a value attribute of "user". You will also have a button on the form to submit the form.

The input for the First Name must have a `name` **AND** `id` attribute of `"firstName"`. The input for the Last Name must have a `name` **AND** `id` attribute of `"lastName"`. The input for the username must have a `name` **AND** `id` attribute of `"username"`. The input for the password must have a `name` **AND** `id` attribute of `"password"`. The input for the confirm password must have a `name` **AND** `id` attribute of `"confirmPassword"`, The input for Favorite Quote must have a name and id attribute of `"favoriteQuote"`, the select for Theme Preference must have a name AND id attribute of `"themePreference"` the select for role but have a `name` **AND** `id` attribute of `"role"`. The button will have a `name` AND `id` attribute of `"submitButton"`

The form will be used to submit the POST request to the `/register` route on the server and **must** have a `name` AND `id` attribute of `"signup-form"`.

Do not forget to use labels for your inputs! For the labels, you will use the method of using the `for` attribute. You should have a label for ALL inputs on your form (First Name, Last Name, Username, Password, Confirm Password, Favorite Quote, Theme Preference and Role) referencing the id's of the inputs in the `for` attribute of the label

You will also have a link on this page that links to `/login` and has the text "Already have an account? Click here to log-in"

An authenticated user should not ever see the sign-up screen!!!

POST `/register`

You must do full input checking and error handling for the inputs in the routes.

1. You must make sure that `firstName`, `lastName`, `username`, `password`, `confirmPassword`, `favoriteQuote`, `themePreference` `role` are supplied in the req.body. If any are missing you will re-render the form with a 400 status code explaining to the user which fields are missing.
2. For `firstName`, it should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 2 characters long with a max of 25 characters. If it fails any of those conditions, you will respond with an error and 400 status code.
3. For `lastName`, it should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 2 characters long with a max of 25 characters. If it fails any of those conditions, you will respond with an error and 400 status code.
4. `username` should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 5 characters long with a max of 10 characters. If it's not, you will respond with an error and 400 status code.
5. For the `password`, it must be a valid string (no strings with just spaces and no spaces but can be any other character including special characters) and should be a minimum of 8 characters long. If it fails any of those conditions, you will throw an error. The constraints for password will be: There needs to be at least one uppercase character, there has to be at least one number and there has to be at least one special character: for example: Not valid: test123, test123$, foobar, tS12$ Valid: Test123$, FooBar123*, HorsePull748*% If it doesn't meet those conditions, you will respond with an error and 400 status code.
6. For `confirmPassword`, you simply have to make sure that it's the same value as `password`, if it's not, you will respond with an error and 400 status code.
7. For `favoriteQuote`, it should be a valid string (no strings with just spaces) and should be at least 20 characters long with a max of 255 characters. if it's not, you will respond with an error and 400 status code.
8. For `themePreference`, the ONLY two valid values are "light" or "dark". If it's not either value, you will respond with an error and 400 status code.
9. For `role`, the ONLY two valid values are "admin" or "user". If it's not either value, you will respond with an error and 400 status code.

If it fails the error checks, or your DB function throws an error, you will render the sign-up screen once again, and this time showing an error message (along with an HTTP 400 status code) to the user explaining what they had entered incorrectly.

Making a POST request to this route you will call your `registerUser` db function passing in the fields from the `request.body`. (You do not have to pass confirmPasswordInput to the DB function)

If your database function returns `{signupCompleted: true}` you will then redirect the user to the `/login` page so they can log in. If your DB function does not return this but also did not throw an error (perhaps the DB server was down when you tried to insert) you will respond with a status code of 500 and error message saying "Internal Server Error"

POST `/login`

You must do full input checking and error handling for the inputs in the routes.

1. You must make sure that `username` and `password` are supplied in the req.body if not, you will respond with an error and 400 status code.
2. `username` should be a valid string (no strings with just spaces, should not contain numbers) and should be at least 5 characters long with a max of 10 characters. If it's not, you will respond with an error and 400 status code.
3. The `username` should be case-insensitive. So "PHILL", "phill", "Phill" should be treated as the same `username`.
4. For the `password`, it must be a valid string (no strings with just spaces and no spaces but can be any other character including special characters) and should be a minimum of 8 characters long. If it fails any of those conditions, you will throw an error. The constraints for password will be: There needs to be at least one uppercase character, there has to be at least one number and there has to be at least one special character: for example: Not valid: test123, test123$, foobar, tS12$ Valid: Test123$, FooBar123*, HorsePull748*% if not, you will respond with an error and 400 status code.

If it fails the error checks or your DB function throws an error, you will render the login screen once again, and this time showing an error message (along with an HTTP 400 status code) to the user explaining what they had entered incorrectly.

This route is simple: making a POST to this route will attempt to log a user in with the credentials they provide in the login form.

You will call your `loginUser` db function passing in the `username` and `password` from the `request.body`. If your DB function returns the user (meaning the DB function found the user and the passwords match),

You will have a cookie named `AuthenticationState` (this is the name of the session in app.js). This cookie must be named `AuthenticationState` or your assignment will receive a major point deduction.

You will store the following information about the user in the `req.session.user` property: the user's first name, the user's last name, the user's username, favoriteQuote, themePreference and the user's role. for example: `req.session.user= {firstName: "Patrick", lastName: "Hill", username: "phill", favoriteQuote: "We have two lives, the 2nd begins when you realize you only have one.", themePreference:, "light", role: "admin"}`. You will then redirect them based on their role, if they

are an admin, you will redirect them to `/admin` if they are a normal user, you will redirect them to `/user`.

If the user does **not** provide a valid login, you will render the login screen once again, and this time show an error message (along with an HTTP 400 status code) to the user explaining that they did not provide a valid username and/or password.

GET `/user`

This route will be simple, as well. This route will be protected your own authentication middleware to only allow valid, logged in users to see this page.

If the user is logged in, you will make a simple view that displays the following information (please note to give the div and all the other elements below the EXACT `id` attributes shown in the example below):

```
<div id="userView">
<p id="welcome">Welcome {{firstName}} {{lastName}}, the time is now: {{currentTime}}. Your role in the sy
stem is: {{role}}."</p>
<p id="quote">Favorite Quote: {{favoriteQuote}}</p>
<!-- link here for admin, ONLY if they are an admin user <a id="adminLink" href="/admin">Admin Page</a> !
-->
<a id="logoutLink" href="/logout">Logout</a>
</div>
```

If they are an admin user, you will also display a link inside the div (after the favorite quote P) to the `/admin` route (**if they are not an admin user, they should not see this link at all!**) give this link an `id` attribute of `"adminLink"`

Also, you will need to have a hyperlink at the bottom of the div to `/logout` give this link an `id` attribute of `"logoutLink"`

GET `/admin`

This route will be simple, as well. This route will be protected your own authentication middleware to only allow valid, logged in users who are admins to see this page.

If the user is logged in and they are an admin user, you will make a simple view that displays the following information ((please note to give the div and all the other elements below the EXACT `id` attributes shown in the example below):

```
<div id="adminView">
<p id="welcome">Welcome {{firstName}} {{lastName}}, the time is now: {{currentTime}}. You get super secre
t admin access since you are an admin. Remember, with great power comes great responsibility!!</p>
<p id="quote">Favorite Quote: {{favoriteQuote}}</p>
<a id="userLink" href="/user">User Page</a>
<a id="logoutLink" href="/logout">Logout</a>
</div>
```

You will also display a link on this page to the /user route so an admin can access the user route if they desire.

Also, you will need to have a hyperlink at the bottom of the page to `/logout`.

GET `/logout`

This route will expire/delete the `AuthenticationState` and inform the user that they have been logged out. It will provide a URL hyperlink to the `/login` route.

Middlewares

You will have the following middleware functions:

1. This middleware will apply to the root route `/` (note, a middleware applying to the root route is the same as a middleware that fires for every request) and will do one of the following:

A. This middleware will log to your console for every request made to the server, with the following information:

- Current Timestamp: `new Date().toUTCString()`
- Request Method: `req.method`
- Request Route: `req.originalUrl`
- Some string/boolean stating if a user is authenticated

There is no precise format you must follow for this. The only requirement is that it logs the data stated above.

An example would be:

```
[Sun, 14 Apr 2019 23:56:06 GMT]: GET / (Non-Authenticated User)
[Sun, 14 Apr 2019 23:56:14 GMT]: POST /login (Non-Authenticated User)
[Sun, 14 Apr 2019 23:56:19 GMT]: GET /user (Authenticated User)
[Sun, 14 Apr 2019 23:56:44 GMT]: GET / (Authenticated User)
```

B. After you log the request info in step A, if the request path is the root "/" and the user is authenticated **AND** they have a role of `admin`, the middleware function will redirect them to the /admin route, if the user is authenticated **AND** they have a role of `user`, you will redirect them to the /user route. If the user is NOT authenticated, you will redirect them to the GET /login route. If the path is not the root, then call next.

2. This middleware will `only` be used for the GET `/login` route and will do one of the following: If the user is authenticated AND they have a role of `admin`, the middleware function will redirect them to the /admin route, if the user is authenticated AND they have a role of `user`, you will redirect them to the /user route. If the user is NOT authenticated, you will allow them to get through to the GET /login route. A logged in user should never be able to access the login form.

3. This middleware will `only` be used for the GET `/register` route and will do one of the following: If the user is authenticated AND they have a role of `admin`, the middleware function will redirect them to the /admin route, if the user is authenticated AND they have a role of `user`, you will redirect them to the /user route. If the user is NOT authenticated, you will allow them to get through to the GET /register route. A logged in user should never be able to access the registration form.

4. This middleware will `only` be used for the GET `/user` route and will do one of the following:

1. If a user is not logged in, you will redirect to the GET /login route.
2. If the user is logged in, the middleware will "fall through" to the next route calling the `next()` callback. (Users with both roles admin or user should be able to access the /user route, so you simply need to make sure they are authenticated in this middleware.)

5. This middleware will `only` be used for the GET `/admin` route and will do one of the following:

1. If a user is not logged in, you will redirect to the GET /login route.
2. If a user is logged in, but they are not an admin user, you end the response right in the middleware function and render a HTML error page saying that the user does not have permission to view the page, and the page must issue an HTTP status code of `403`. (provide a link to the /user page, since they are logged in, just not an admin)
3. If the user is logged in AND the user has a role of `admin`, the middleware will "fall through" to the next route calling the `next()` callback.
4. ONLY USERS WITH A ROLE of `admin` SHOULD BE ABLE TO ACCESS THE /admin ROUTE!

6. This middleware will `only` be used for the GET `/logout` route and will do one of the following:

1. If a user is not logged in, you will redirect to the GET /login route.
2. if the user is logged in, the middleware will "fall through" to the next route calling the `next()` callback.

See this reference (https://expressjs.com/en/guide/writing-middleware.html) in the express documentation to read more about middleware.

Styling

You will create a CSS stylesheet in `/public/css/styles.css`; this file should have *at least 5 rulesets.* You will need to utilize all 5 rulesets on each of the pages (not every page has to use all 5 rulesets each but all 5 rulesets must be utilized by the various pages in your application). So you can have some rulesets for one page, some for another page but there needs to be at least 5 total rulesets that are used between all the pages. There are also two specific rulesets you need to create **in addition to the 5 other rulesets**: `.light` and `.dark`. Depending of the user's theme preferences stored in the session, your pages will be displayed using the user's theme preferences. If their theme preference is "light" then the background color of the page should be `white` and the text color of the page should be `black`. If their theme preference is "dark" then the background color of the page should be `black` and the text color should be `white`. HINT: You can just set the `body` element's class to either `dark` or `light` depending on their `themePreference` stored in the session. obviously, the theme preference only is active after a user is logged in. ALL pages must have the theme applied when a user is logged in.

Client-Side Validation For All Forms!

You will create a client-side Javascript file in `/public/js/client_side_validation.js`.

In this file, you must perform all client-side validation for every single form input (also the theme preference & role dropdowns) on your pages. The constraints for those fields are the same as they are for the data functions and routes. Using client-side JS, you will intercept the form's submit event when the form is submitted and If there is an error in the user's input or they are missing fields, you will not allow the form to submit to the server and will display an error on the page to the user informing them of what was incorrect or missing. You must do this for ALL fields for the register form as well as the login form. If the form being submitted has all valid data, then you will allow it to submit to the server for processing. Don't forget to check that password and confirm password match on the registration form!

Using `express-session`

This middleware package does one (fairly simple) thing. It creates a cookie for the browser that will be used to track the current session of the user, after we verify their login. We will expand on the `req.session` field to store information about the currently logged in user. You can see an example using `req.session` **here (https://github.com/expressjs/session#reqsession)** .

To initialize the middleware, you must do the following:

```
// Your app.js file

import session from 'express-session';

...

app.use(session({
  name: 'AuthenticationState',
  secret: 'some secret string!',
  resave: false,
  saveUninitialized: false
}))
```

You can read more about session's different configuration options **here (https://github.com/expressjs/session#options)** . For the sake of this lab, the above configuration is all you will need.

Requirements

1. All previous lab requirements still apply.
2. You must remember to update your package.json file to set app.js as your starting script!
3. **Your HTML must be valid (https://validator.w3.org/#validate_by_input)** or you will lose points on the assignment.
4. Your HTML must make semantical sense; usage of tags for the purpose of simply changing the style of elements (such as i, b, font, center, etc) will result in points being deducted; think in terms of content first, then style with your CSS.
5. You can be as creative as you'd like to fulfill front-end requirements; if an implementation is not explicitly stated, however you go about it is fine (provided the HTML is valid and semantical). Design is not a factor in this course.
6. All inputs must be properly labeled!

7. All inputs should be trimmed!