Lab 10

New Attempt

Due Apr 11 by 11:59pm **Points** 100 **Submitting** a file upload

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 usernames and hashed password. 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.

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"),
    username: 'graffixnyc',
    password: '$2b$16$Vm/Xqc.2eyi3y3IqewuhjOTXeoxt4SaN1dcAfPwEPUrzA5Kgm1HFW'
  },
  {
    _id: ObjectId("615f5211445eac188610ecc0"),
    username: 'phill',
    password: '$2b$16$SHQUG43PoIHoTHvkeDBczewvurYf3l.XKMRhrRomB.iVMcvldsq8m'
  }
];
```

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

createUser(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. For `username`, it should be a valid string (no empty spaces, no spaces in the username and only alphanumeric characters) and should be at least 4 characters long. 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. YOU MUST NOT allow duplicate usernames in the system. If the username is already in the database you will throw an error stating there is already a user with that username
5. For the password, it must be a valid string (no empty spaces and no spaces but can be any other character including special characters) and should be at least 6 characters long. If it fails any of those conditions, you will throw an error.

In this function you will hash the password using bcrypt. You will then insert the username and **hashed** password into the database.

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

checkUser(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. For `username`, it should be a valid string (no empty spaces, no spaces in the username and only alphanumeric characters) and should be at least 4 characters long. 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 empty spaces and no spaces but can be any other character including special characters) and should be at least 6 characters long. If it fails any of those conditions, you will throw an error.

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 `{authenticated: true}`
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 will do one of two things:

1. If the user is authenticated, it will redirect to `/private`.
2. If the user is not authenticaed, it will render a view with a login form. The form will contain two inputs, one for username and one for password. The form will be used to submit a POST request to the `/login` route on the server and **must** have an `id` of `login-form`. The input for the username must have a `name`/ `id` of `username`; the input for the password must have `name`/ `id` of `password` and should be an input type of password.

You will also have a link on this page that links to `/signup` and has the text "Need to register? Click here to sign-up"

Do not forget to use labels for your inputs!

An authenticated user should not ever see the login screen.

GET `/signup`

1. If the user is authenticated, it will redirect to `/private`.
2. If the user is not authenticated, this route will render a view with a sign-up form. The form will contain two inputs, one for the username and one for password.
3. The form will be used to submit the POST request to the `/signup` route on the server and **must** have an `id` of `signup-form`. The input for the username must have a `name`/ `id` of `username`; the input for the password must have `name`/ `id` of `password` and should be an input type of password.

Do not forget to use labels for your inputs!

You will also have a link on this page that links to `/` 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 `/signup`

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
2. For username, it should be a valid string (no empty spaces, no spaces in the username and only alphanumeric characters) and should be at least 4 characters long and should be case-insensitive.
3. For the password, it must be a valid string (no spaces, no empty spaces but can be any other character including special characters) and should be at least 6 characters long.

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 createUser db function passing in the `username` and `password` from the `request.body`.

If your database function returns `{userInserted: true}` you will then redirect the user to the `/` 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
2. For `username`, it should be a valid string (no empty spaces, no spaces in the username and only alphanumeric characters) and should be at least 4 characters long and should be case-insensitive.
3. For the `password`, it must be a valid string (no spaces, no empty spaces but can be any other character including special characters) and should be at least 6 characters long.

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 checkUser db function passing in the `username` and `password` from the `request.body`. If your DB function returns `{authenticated: true}`, You will have a cookie named `AuthCookie`(this is the name of the session in app.js). This cookie must be named `AuthCookie` or your assignment will receive a major point deduction. You will also store the username of the user in the session so you can display the value in the `/private` route. After logging in, you will redirect the user to the `/private` route.

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 `/private`

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 username (which you stored in the session when they logged in) for the currently logged in user.

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

GET `/logout`

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

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

const session = require('express-session')

...

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

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.

Authentication Middleware

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

1. If a user is not logged in, you will return an HTML page saying that the user is not logged in, and the page must issue an HTTP status code of `403`.
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.

Logging Middleware

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 /private (Authenticated User)
[Sun, 14 Apr 2019 23:56:44 GMT]: GET / (Authenticated User)
```

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!