WEB322 Assignment 6

Submission Deadline:

Monday, April 7 – 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Part A: Work with Client Sessions and data persistence using MongoDB to add user registration and Login/Logout functionality & tracking (logging)

Part B: Update the password storage logic to include "hashed" passwords (using bcrypt.js)

NOTE: Please refer the sample: https://web322-w25-a6.vercel.app/ when creating your solution. Once again, the UI does not have to match exactly, but this will help you determine which elements / syntax should be on each page. **You may copy** any HTML / CSS code from here if it helps with your solution.

Specification:

For this assignment, we will be allowing users to "register" for an account on your Climate Solutions App. Once users are registered, they can log in and gain access to the add / edit & delete functionality created in assignment 5. By default, this functionality will be hidden from the end user and unauthenticated users will only see the "projects" / "project" and "about" views / top menu links. Once this is complete, we will add bcrypt.js to our code to ensure that all stored passwords are "hashed"

NOTE: If you are unable to start this assignment because Assignment 5 was incomplete - email your professor for a clean version of the Assignment 5 files to start from.

Part A: User Accounts / Sessions

Step 1: Getting Started:

If you have not already done so, create a new account on https://www.mongodb.com/cloud/atlas to host our new MongoDB database:

- Follow the instructions from the course notes under "Setting up a MongoDB Atlas account"

- Continue following the instructions until you create a new database (named whatever you like) and connection string, to be used in the following steps.

Step 2: Adding a new "service" module to persist User information:

For our app to be able to register new users and authenticate existing users, we must create a convenient way to access this stored information. To accomplish this, we will need to **add a new module** called "**auth-service**". This module will be responsible for storing and retrieving user information (user & password) using our newly created **MongoDB database**:

1. Use npm to install **mongoose** (We will be using this ODM to connect to our new DB)

2. Create a new file at the within the "**modules**" folder, called "**auth-service.js**"

3. "**Require**" your new "**auth-service.js**" module at the top of your **server.js** file as "**authData**"

4. Inside your **auth-service.js** file write code to **require** the **mongoose** module and create a **Schema** variable to point to **mongoose.Schema** (**Hint**: refer to "Creating a Schema")

5. Add the "dotenv" module: **require('dotenv').config();**

6. Define a new "**userSchema**" according to the following specification:

Property	Mongoose Schema Type
userName	String (**NOTE:** this value must be **unique**)
password	String
email	String
loginHistory	[{ Property: Type, Property: Type }] NOTE: this will be an array of **objects** that use the following specification: `TABLE_PLACEHOLDER`

Property	Mongoose Schema Type
dateTime	Date
userAgent	String

7. Once you have defined your "**userSchema**" per the specification above, add the line:

    o **let User; // to be defined on new connection (see initialize)**

auth-service.js - Exported Functions

Each of the below functions are designed to work with the **User** Object (defined by **userSchema**). Once again, since we have no way of knowing how long each function will take**, every one of the below functions must return a Promise** that **passes the data** via its "**resolve**" function (or if an error was encountered, passes an **error message** via its "**reject**" function). When we access these functions from the server.js file, we will be assuming that they return a Promise and will respond appropriately with **.then()** and **.catch()** (or with async / await and try / catch).

initialize()

*   Much like the "initialize" function in our projects module, we must ensure that we are able to sync / connect to our MongoDB instance before we can start our application.

- We must also ensure that we create a new connection (using **createConnection()** instead of **connect()** - this will ensure that we use a connection local to our module) and initialize our "User" object, if successful

- Additionally, if our connection is successful, we must **resolve()** the returned Promise without returning any data

- If our connection has an error, we must, **reject()** the returned Promise with the provided error:

- To achieve this, **use the following code** for your new initialize function, where **process.env.MONGODB** is your **completed connection string** to your MongoDB Atlas database as identified above. **NOTE:** This will have to be added to your .env file, ie: MONGODB='**your completed connection string**'

```
return new Promise(function (resolve, reject) {
  let db = mongoose.createConnection(process.env.MONGODB);

  db.on('error', (err)=>{
     reject(err); // reject the promise with the provided error
  });
  db.once('open', ()=>{
    User = db.model("users", userSchema);
    resolve();
  });
});
```

registerUser(userData)

- This function is slightly more complicated, as it needs to perform some **data validation** (ie: **do the passwords match? Is the user name already taken?)**, return meaningful errors if the data is invalid, as well as saving **userData** to the database (if no errors occurred). To accomplish this:

  - You may assume that the **userData** object has the following properties: **.userName, .userAgent, .email, .password**, **.password2** (we will be using these field names when we create our **register** view). You can compare the value of the **.password** property to the **.password2** property and if they do not match, **reject** the returned promise with the message: "**Passwords do not match**"

  - Otherwise (if the passwords successfully match), we must create a new **User** from the **userData** passed to the function, ie: **let newUser = new User(userData);** and invoke the **newUser.save()** function (**Hint**: refer to the "MongoDB Crud Reference")

    - If the save operation was rejected with error (**err**) and its **err.code is 11000** (duplicate key), **reject** the returned promise with the message: "**User Name already taken**".

    - If the save operation was rejected with error (**err**) and its **err.code is not 11000**, **reject** the returned promise with the message: "**There was an error creating the user: err**" where **err** is the full error object

    - If the save operation resolved successfully, **resolve** the returned promise without any message

checkUser(userData)

- This function is also more complex because, while we may **find** the user in the database whose **userName property** matches **userData.userName,** the provided password (ie, **userData.password**) may not match (or the user may not be found at all / there was an error with the query). In either case, we must reject the returned promise with a meaningful message. To accomplish this:

- o Invoke the **find()** method on the **User** Object (defined in our initialize method) and filter the results by only searching for users whose **user** property matches **userData.userName**, ie: **User.find({ userName: userData.userName })** (**Hint**: refer to the "MongoDB Crud Reference")

  - If the **find()** promise resolved successfully, but **users** is an **empty array**, **reject** the returned promise with the message "Unable to find user: *user*" where *user* is the **userData.userName** value

  - If the **find()** promise resolved successfully, but the **users[0].password** (there should only be one returned user) **does not match userData.password**, **reject** the returned promise with the error "Incorrect Password for user: *userName*" where *userName* is the **userData.userName** value

  - If the **find()** promise resolved successfully and the **users[0].password matches userData.password**, then we must perform the following actions to record the action in the "loginHistory" array before we can resolve the promise with the **users[0]** object:

    - Check if there are 8 login history items (this is our maximum) and if there are, "pop" the last element from the array:

      ```
      if(users[0].loginHistory.length == 8){
          users[0].loginHistory.pop()
      }
      ```

    - Now that we have space in our loginHistory array, add a new entry to the front of the array using "unshift()":

      ```
      users[0].loginHistory.unshift({dateTime: (new Date()).toString(), userAgent: userData.userAgent});
      ```

    - Next, invoke the **updateOne** method on the **User** object where userName is **users[0].userName** and **$set** the **loginHistory** value to **users[0].loginHistory**. (**Hint**: refer to the "MongoDB Crud Reference" for a refresher on **updateOne**)

    - Finally, if the above was successful, **resolve** the returned promise **with the users[0] object**. If it was unsuccessful, **reject** the returned promise with the message: "**There was an error verifying the user: *err***" where *err* is the full error object

  - If the **find()** promise was rejected, **reject** the returned promise with the message "Unable to find user: *user*" where *user* is the **userData.userName** value

Step 3: Adding authData.initialize to the "startup procedure":

Once the code for **auth-service.js** is complete, we need to add its **initialize** method to the promise chain surrounding our **app.listen()** function call within our **server.js** file, for example:

Your code should currently look something like this:

```
projectData.initialize().then(function(){
    app.listen(HTTP_PORT, function(){
        console.log(`app listening on:  ${HTTP_PORT}`);
    });
}).catch(function(err){
    console.log(`unable to start server: ${err}`);
});
```

Since our server also requires **authData** to be working properly, we must add its **initialize** method (ie: **authData.initialize**) to the promise chain:

```
projectData.initialize()
.then(authData.initialize)
.then(function(){
   app.listen(HTTP_PORT, function(){
      console.log(`app listening on:  ${HTTP_PORT}`);
   });
}).catch(function(err){
   console.log(`unable to start server: ${err}`);
});
```

Step 4: Configuring Client Session Middleware:

Now that we have a back-end to store user credentials and data, we must download and "require" the "client-sessions" module using NPM and correctly configure our app to use the middleware:

1. Open the "Integrated Terminal" in Visual Studio Code and enter the command:
    npm install client-sessions

2. Be sure to "require" the new "client-sessions" module at the top of your **server.js file** as **clientSessions**.

3. Ensure that we correctly use the client-sessions middleware with appropriate **cookieName, secret, duration** and **activeDuration** properties (**HINT**: Refer to the "Middleware" notes under "Managing State Information")

4. Once this is complete, incorporate the following custom middleware function to ensure that all of your templates will have access to a "session" object (ie: {{session.userName}} for example) - we will need this to conditionally hide/show elements to the user depending on whether they're currently logged in.

    ```
    app.use((req, res, next) => {
      res.locals.session = req.session;
      next();
    });
    ```

5. Define a helper middleware function (ie: **ensureLogin** from the "Middleware" notes, in the "Practical Application" section) that checks if a user is logged in (we will use this in all of our post / category routes). If a user is not logged in, redirect the user to the "/login" route.

6. Update all routes that allow users to **add**, **edit**, or **delete** Projects (this should be 5 routes) to use your custom **ensureLogin** helper middleware.

Step 5: Adding New Routes:

With our app now capable of respecting client sessions and communicating with MongoDB to register/validate users, we need to create **routes** that enable the user to register for an account and login / logout of the system (above our 404 middleware function). Once this is complete, we will create the corresponding **views** (Step 6).

GET /login

- This "GET" route simply renders the "**login**" view with the data: { errorMessage: "", userName: "" } (See **login.ejs** under Adding New Routes below)

GET /register

- This "GET" route simply renders the "**register**" view with the data: {errorMessage: "", successMessage: "", userName: ""} (See **register.ejs** under Adding New Routes below)

POST /register

- This "POST" route will invoke the **authData.RegisterUser(userData)** method with the POST data (ie: **req.body**).

  - If the promise resolved successfully, **render** the **register** view with the following data:
    { errorMessage: "", successMessage: "User created", userName: "" }

  - If the promise was rejected (**err**), **render** the **register** view with the following data:
    { errorMessage: err, successMessage: "", userName: req.body.userName } - **NOTE:** we are returning the userName back to the page, so the user does not forget the **userName value** that was used to attempt to register with the system

POST /login

- Before we do anything, we must set the value of the client's "User-Agent" to the **request body**, ie:

  req.body.userAgent = req.get('User-Agent');

- Next, we must invoke the **authData.CheckUser(userData)** method with the POST data (ie: **req.body**).

  - If the promise resolved successfully, add the returned user's **userName, email & loginHistory** to the session and redirect the user to the "/solutions/projects" view, ie:

    authData.checkUser(req.body).then((user) => {
    ** req.session.user = {**
    **   userName: // authenticated user's userName**
    **   email: // authenticated user's email**
    **   loginHistory: // authenticated user's loginHistory**
    ** }**

    ** res.redirect('/solutions/projects);**
    })

  - If the promise was rejected (ie: in the "**catch**"), **render** the **login** view with the following data (where **err** is the parameter passed to the "**catch**": **{errorMessage: err, userName: req.body.userName}** - **NOTE:** we are returning the user back to the page, so the user does not forget the **user value** that was used to attempt to log into the system

GET /logout

- This "GET" route will simply "reset" the session (**Hint**: refer to the "Route Updates (Logic)" section) and redirect the user to the "/" route, **ie: res.redirect('/');**

GET /userHistory

- This "GET" route simply renders the "**userHistory**" view without any data (See **userHistory.ejs** under Adding New Routes below). **IMPORTANT NOTE:** This route (like the **5 others** from above) must also be protected by your custom **ensureLogin** helper middleware.

Step 6: Updating / Adding New Views:

Lastly, to complete the register / login functionality, we must update/create the following **.ejs** files (views) within the **views** directory.

partials/navbar.ejs

- To enable users to register for accounts, login / logout of the system, and conditionally hide / show menu items, we must make some small changes to our navbar.ejs.

- To begin, remove the "Add Project" Link (we will be moving it to a new location)

- Next, ensure that the following dropdown is rendered in the navbar (and responsive navbar) if **there is** an active session (ie: if(session.user){ ... }):

```
<li>
  <details>
    <summary>Account: <%= session.user.userName %></summary>
    <ul class="p-2 right-0">
      <li><a class="<%= (page == "/solutions/addProject") ? 'active' : '' %>" href="/solutions/addProject">Add Project</a></li>
      <li><a class="<%= (page == "/userHistory") ? 'active' : '' %>" href="/userHistory">User History</a></li>
      <li><a href="/logout">Log Out</a></li>
    </ul>
  </details>
</li>
```

- Similarly, we must also ensure that the following dropdown is rendered in the navbar (and responsive navbar) when **there is not** an active session:

```
<li>
  <details>
    <summary>Account</summary>
    <ul class="p-2 right-0">
      <li><a class="<%= (page == "/login") ? 'active' : '' %>" href="/login">Login</a></li>
      <li><a class="<%= (page == "/register") ? 'active' : '' %>" href="/register">Register</a></li>
    </ul>
  </details>
</li>
```

login.ejs

- This (new) view must consist of the "login form" which will allow the user to submit their credentials (using **POST**) to the **"/login"** POST route:

input type	Properties	Value
text	name: "userName" placeholder: "User Name" required	**userName** if it was rendered with the view. Refer to the "/login" POST route above for more information

password	name: "password" placeholder: "Password" required	
submit (button)	text / value: "Login"	

- Below the form, we must have a space available for error output: Show the [daisyUI Error Alert](#):

  <div class="alert alert-error">

   <svg xmlns="http://www.w3.org/2000/svg" class="stroke-current shrink-0 h-6 w-6" fill="none" viewBox="0 0 24 24"><path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M10 14l2-2m0 0l2-2m-2 2l-2-2m2 2l2 2m7-2a9 9 0 11-18 0 9 9 0 0118 0z" /></svg>

   <%= errorMessage %>

  </div>

  only if there is an errorMessage rendered with the view.

- For layout guidelines/elements used to create the form, refer to the HTML code available here: https://web322-w25-a6.vercel.app/login . When complete, the form should look like this:

register.ejs

- This (new) view must consist of the "register form" which will allow the user to submit new credentials (using **POST**) to the **"/register"** POST route. **IMPORTANT NOTE:** this form is **only visible** if **successMessage** was **not** rendered with the view (refer to the "/register" POST route above for more information). If **successMessage** was rendered with the view, we will show different elements.

input type	Properties	Value
text	name: "userName" placeholder: "User Name" required	**userName** if it was rendered with the view. Refer to the "/register" POST route above for more information
password	name: "password" placeholder: "Password" required	
password	name: "password2" placeholder: "Confirm Password" required	
email	name: "email" placeholder: "Email Address" required	
submit (button)	text / value: "Register"	

- Below the form, we must have a space available for error output: Show the daisyUI Error Alert:

<div class="alert alert-error">

  <svg xmlns="http://www.w3.org/2000/svg" class="stroke-current shrink-0 h-6 w-6" fill="none" viewBox="0 0 24 24"><path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M10 14l2-2m0 0l2-2m-2 2l-2-2m2 2l2 2m7-2a9 9 0 11-18 0 9 9 0 0118 0z" /></svg>

  <%= errorMessage %>

</div>

only if there is an errorMessage rendered with the view.

- Additionally, we must also have a space available for success output: Show the daisyUI default Alert:

<div class="alert">

  <svg xmlns="http://www.w3.org/2000/svg" class="stroke-current shrink-0 h-6 w-6" fill="none" viewBox="0 0 24 24"><path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M9 12l2 2 4-4m6 2a9 9 0 11-18 0 9 9 0 0118 0z" /></svg>

  <%= successMessage %>

</div>

  Proceed to Log in

only if there is a successMessage rendered with the view (this will be rendered **instead** of the form).

- For layout guidelines/elements used to create the form, refer to the HTML code available https://web322-w25-a6.vercel.app/register . When complete, the form should look like this:

userHistory.ejs

- This (new) view simply renders the following table using the globally available **session.user.loginHistory** object

Column	Value
Login Date/Time	This will be the **dateTime** value for the current **loginHistory** object formatted to show the date & time
Client Information	This will be the **userAgent** value for the current **loginHistory** object

- Additionally, in the "Hero" portion of the page, add the code to show the **userName** and **email** properties of the logged in user (**session.user**)

- For layout guidelines/elements used to create the table, first **create an account to gain access** and refer to the HTML code available here: https://web322-w25-a6.vercel.app/userHistory . When complete, the view should look something like this:
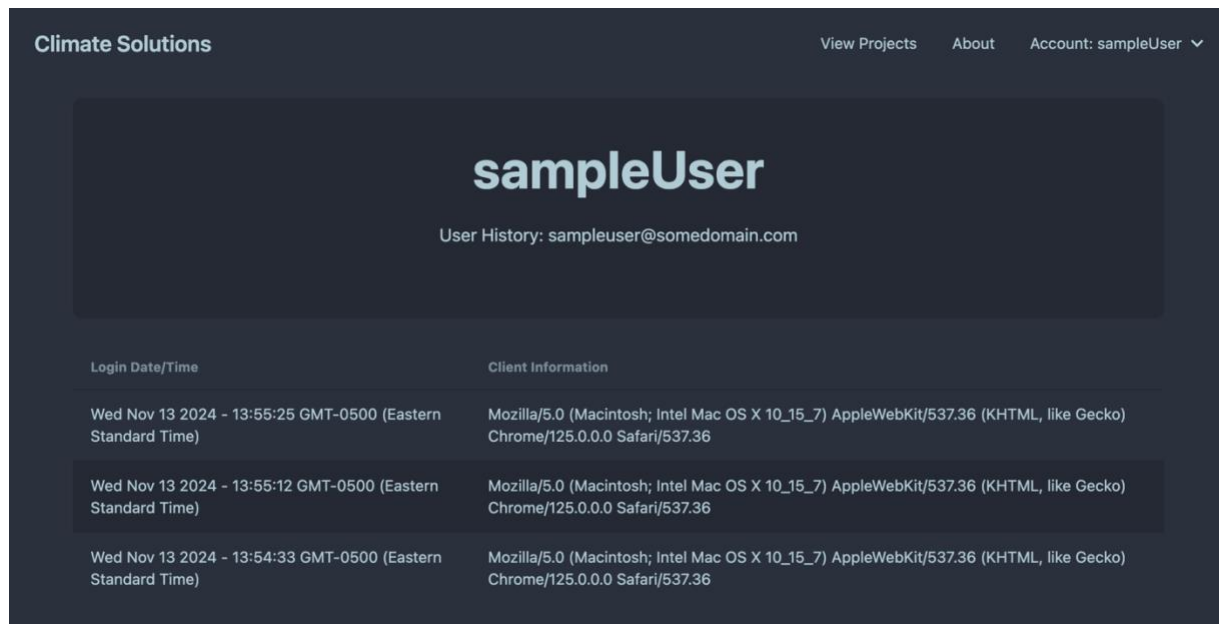
project.ejs

- Finally, we should make sure that only users with an active session should have access to the "edit" button when viewing a project:

  <% if(session.user){ %> …. (button to edit a project here) …. <% } %>

Part B - Hashing Passwords

We will be using the "bcryptjs" 3rd party module, so we must go through the usual procedure to obtain it (and include it in our "auth-service.js" module).

1. Open the integrated terminal and enter the command: **npm install "bcryptjs"**

2. At the top of your **auth-service.js** file, add the line: **const bcrypt = require('bcryptjs');**

Step 1: Clearing out the "Users" collection

Since all our new users will have encrypted (hashed) password, we will need to remove all our existing test users. This can be done easily by logging into your MongoDB Atlas account and clicking on the "collections" for your existing cluster.

- You should now see a list of databases & collections. Simply hover over the collection that you wish to remove (ie: **users**) and click the **trash can icon** that appears.

- Lastly, enter the name of the collection (ie: users) in the confirmation dialog to drop the "users" collection

Step 2: Updating our auth-service.js functions to use bcrypt:

Now that we have the bcryptjs module included and our Users collection has been cleaned out, we can focus on updating the other two functions in our auth-service.js module. We will be using bcrypt to encrypt (hash) passwords in **registerUser(userData)** and validate user passwords against the encrypted passwords in **checkUser(userData):**

Updating registerUser(userData)

- Recall from the notes on "Password Encryption" - to encrypt a value (ie: "myPassword123"), we can use the following code:

```
bcrypt.hash("myPassword123", 10).then(hash=>{ // Hash the password using a Salt that was generated using 10 rounds
    // TODO: Store the resulting "hash" value in the DB
})
.catch(err=>{
    console.log(err); // Show any errors that occurred during the process
});
```

- Use the above code to **replace** the user entered password (ie: **userData.password**) with its **hashed version** (ie: **hash**) **before** continuing to save **userData** to the database and handling errors.

- If there was an error, **reject** the **returned promise** with the message "There was an error encrypting the password" and **do not** attempt to save **userData** to the database.

Updating checkUser(userData)

- Recall from the notes on "Password Encryption" - to compare an encrypted (hashed) value (ie: **hash**) with a plain text value (ie: "**myPassword123**", we can use the following code:

```
bcrypt.compare("myPassword123", hash).then((result) => {
    // result === true if it matches and result === false if it does not match
});
```

- Use the above code to **verify** if the user entered password (ie: **userData.password**) matches the hashed version for the requested user (**userData.userName**) in the database (ie: **instead** of simply comparing users[0].password == userData.password as this will no longer work. The **compare** method must be used to compare the hashed value from the database to userData.password)

If the passwords do not match (ie: **result === false**) **reject** the returned promise with the message "Incorrect Password for user: *userName*" where *userName* is the **userData.userName** value

Assignment Submission:

- Add the following declaration at the top of your **server.js** file:

```
/*********************************************************************************
* WEB322 – Assignment 06
*
* I declare that this assignment is my own work in accordance with Seneca's
* Academic Integrity Policy:
*
* https://www.senecapolytechnic.ca/about/policies/academic-integrity-policy.html
*
* Name: _____ Student ID: _____ Date: _____
*
*********************************************************************************/
```

- Compress (.zip) your web322-app (after removing node_modules folder).
- Create a screen recording (**maximum 15 minutes**) that starts by showing your app in action (on Vercel URL). After the demo, explain the code, focusing on the flow and logic—avoid simply reading it line by line. Throughout the recording, narrate what you're doing in real time.
- **IMPORTANT**: The app demo in screen recording must be done on the hosted website (Vercel)
- **Submit the following:**
    - **Project zip file (without node_modules folder)**
    - **Published URL (host the project on Vercel and share the public link in submission comments)**
    - **Screen Recording URL (upload the video to YouTube, set it as "unlisted", and share the link in submission comments)**
  Note:
  - **All items listed above must be submitted for your submission to be considered complete/valid.**
  - **Incomplete/Invalid submissions will not be accepted and will receive a grade of zero (0).**

- Here is a video tutorial on how to upload videos to YouTube and set them as "unlisted": https://www.youtube.com/watch?v=jaftEW9WI3U
- Here is a video tutorial on how to host your web app on Vercel and get the public/hosted URL: https://www.youtube.com/watch?v=Rko7rMe4l60
- Here is a video tutorial on how to create multiple branches in Git and use them to create deployments on Vercel: https://www.youtube.com/watch?v=NyqzCuW7LMc

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.

- Submitted assignments must run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.