WEB322 Assignment 6

Submission Deadline:

Saturday, August 11th, 2018 @ 11:59 PM

Assessment Weight:

10% 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)

Part C: The look-and-feel will be marked (**2 pts**) in this assignment 6. Create your own custom CSS to style your website.

Specification:

For this assignment, we will be allowing users to "register" for an account on your WEB322 App. Once users are registered, they can log in and access all related employee/department views. By default, these views will be hidden from the end user and unauthenticated users will only see the "home" 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 (effectively removing any custom CSS or text added to your solution). Remember, you must successfully complete ALL assignments to pass this course.

Part A: User Accounts / Sessions

Step 1: Getting Started:

Before we get started, we must create a new account on www.mlab.com to host our new MongoDB database:

- Navigate to www.mlab.com
- Click the blue "SIGN UP" button on the top right of the page
- Fill out the form to enter your:
    - Account Name (use your My.Seneca user name - if it's already taken, choose something similar, so that it's easy to remember)
    - Email (use your full Seneca email address)
    - User Name (use your My.Seneca user name - if it's already taken, choose something similar, so that it's easy to remember - **you will need this user name to log in**)
    - Password (pick something you'll remember - **you will need this password to log in**)

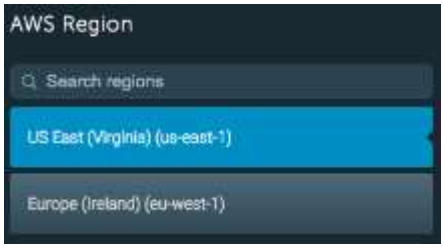
- Enter your password one more time, accept the terms and click the blue "Create Account" button
- You should then receive (in your Seneca email) an email from mLab asking you to "Verify your mLab email address". Click the link in the email to verify your account

Now that we have created and verified our account, we must create a new "MongoDB Deployment"

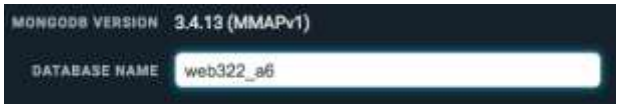
- Click the "Create new" button (indicated with a plus sign) next to the "MongoDB Deployments" section:

- On the next page, leave the "Cloud Provider" at the default ("Amazon web services")
- For "Plan Type" click the green "FREE" button next to "Sandbox" - this is the FREE option

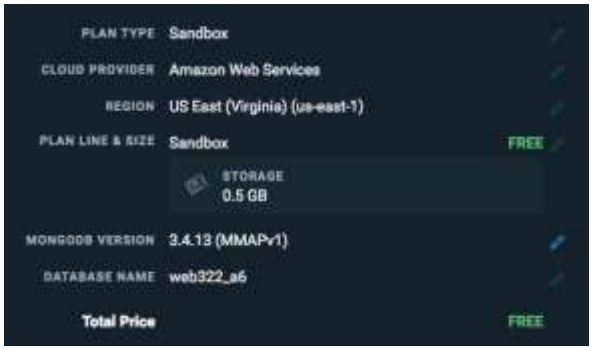
- Hit the blue "Continue" button on the bottom-right to continue to the next step
- Choose "US East" for the AWS Region and hit "Continue" again



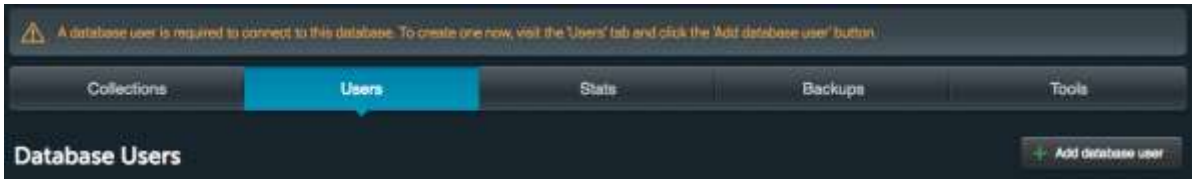
- For the "Database Name" field enter: "web322_a6"



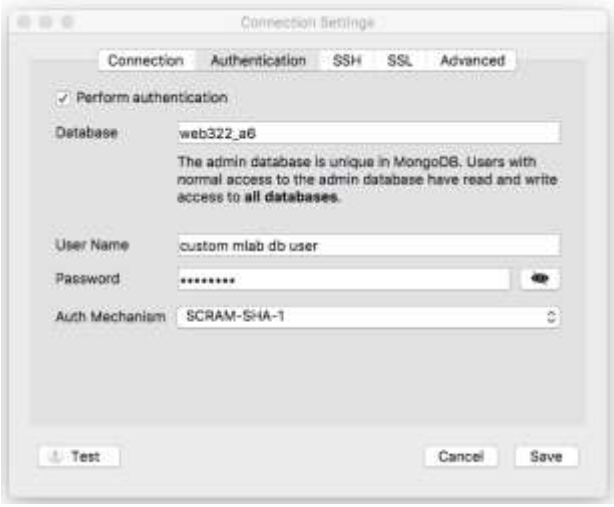
- Once again, hit "Continue" to proceed to the next step
- Finally, you should see the following view, outlining your (Free) plan. At the bottom of the page (where the "Continue" button usually is, you will see a "Submit Order" button - click this to create your deployment



- Once this is complete, you should be redirected to your "MongoDB Deployments" page and you should see a table with single deployment with a green checkmark next to it. Click this row to edit the details of the deployment
- This will take you to a page "Database: web322_a6" where you can edit the details of the database.
- Click on the "Users" tab and click the button: "+ Add database user"



- This will open a modal window prompting you to create a user (user name / password). Use something that you will remember and click the blue "CREATE" button (make sure to leave "Make read-only" unchecked)
- Take note of the section (near the top of the page): "To connect using a driver via the standard MongoDB URI" - it should look *something* like this:
"**mongodb://<dbuser>:<dbpassword>@ds000000.mlab.com:00000/web322_a6"**
- This is the connection string that we will be using to connect to our database (where **<dbuser>** is your newly created "Database User" and **<dbpassword>** is the corresponding password
- Write down your **completed connection string** using your **<dbuser>** and **<dbpassword>** - we will be using it in our application
- **NOTE**: It is recommended at this point that you confirm your connection using Robo 3T (refer to the Week 8 notes under "Tools" & "The MongoDB Connection String Format" for instructions) - once connected, leave the window open to view your data & test your solution. **NOTE**: You will have to use the "Connection" and "Authentication" tabs with your new mLab database credentials to successfully connect:



Step 2: Adding a new "data-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 "**data-service-auth**". 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 root of your web322-app folder called "**data-service-auth.js**"
3. "**Require**" your new "**data-service-auth.js**" module at the top of your **server.js** file as "**dataServiceAuth**"
4. Inside your **data-service-auth.js** file write code to **require** the **mongoose** module and create a **Schema** variable to point to **mongoose.Schema** (**Hint**: refer to the Week 8 notes)
5. 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: **Property** / **Mongoose Schema Type** dateTime / Date userAgent / String

6. Once you have defined your "**userSchema**" per the specification above, add the line:
    - **let User; // to be defined on new connection (see initialize)**

data-service-auth.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 it's "**resolve**" method (or if an error was encountered, passes an **error message** via it's "**reject**" method). When we access these methods from the server.js file, we will be assuming that they return a promise and will respond appropriately with **.then()** and .**catch()**.

initialize()

- Much like the "initialize" function in our data-service module, we must ensure that we are able to 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 ***connectionString*** is your **completed connection string** to your mLab database as identified above:

```
module.exports.initialize = function () {
  return new Promise(function (resolve, reject) {
    let db = mongoose.createConnection("connectionString");

    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 Week 8 notes)
        - If an error (**err**) occurred and it's **err.code is 11000** (duplicate key), **reject** the returned promise with the message: "**User Name already taken**".

- If an error (**err**) occurred and it's **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 an error (**err**) **did not occur** at all, **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 **user property** matches **userData.user,** 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:
    - 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({ user: userData.userName })** (**Hint**: refer to the Week 8 notes)
        - 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:
            - Using the returned user object (ie, **users[0]**), **push** the following object onto its "loginHistory" array:
                - {dateTime: (new Date()).toString(), userAgent: userData.userAgent}
            - Next, invoke the **update** method on the **User** object where userName is **users[0].userName** and **$set** the **loginHistory** value to **users[0].loginHistory**. (**Hint:** refer to the Week 8 notes for a refresher on **update**)
            - 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.user** value

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

Once the code for **dataServiceAuth** is complete, we need to add it's **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:

```
data.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 **dataServiceAuth** to be working properly, we must add it's **initialize** method (ie: **dataServiceAuth.initialize**) to the promise chain:

```
dataService.initialize()
.then(dataServiceAuth.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 --save

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 Week 10 notes under "Step 2: Create a middleware function to setup client-sessions.")

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(function(req, res, next) {
    res.locals.session = req.session;
    next();
   });
   ```

5. Define a helper middleware function (ie: **ensureLogin** from the Week 10 notes) that checks if a user is logged in (we will use this in all of our employee / department routes). If a user is not logged in, redirect the user to the "/login" route.

6. Update all routes that **begin** with one of: **"/employees"**, "**/employee**", "**/images**", "**/departments**" or "**/department**" (ie: everything that is **not "/" or "/about"** - this should be **14** 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 without any data (See **login.hbs** under Adding New Routes below)

GET /register

- This "GET" route simply renders the "**register**" view without any data (See **register.hbs** under Adding New Routes below)

POST /register

- This "POST" route will invoke the **dataServiceAuth.RegisterUser(userData)** method with the POST data (ie: **req.body**).
    - If the promise resolved successfully, **render** the **register** view with the following data: **{successMessage: "User created"}**
    - If the promise was rejected (**err**), **render** the **register** view with the following data: **{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 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 **dataServiceAuth.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 "/employees" view, ie:

      ```
      dataServiceAuth.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('/employees');
      })
      ```

    - 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 Week 10 notes) 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.hbs** under Adding New Routes below). **IMPORTANT NOTE:** This route (like the **14 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 **.hbs** files (views) within the **views** directory.

layouts/main.hbs

- 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 main.hbs.
- Update the code inside the … block in the header, just below the <ul class="nav navbar-nav">… element (this element has the "home" and "about" links) according to the following specification: (**Note**: pay **close attention** to the **formatting** when copying/pasting code from this document)
    - **If session.user** exists (ie: the user is logged in), show the following HTML:

```
<form class="navbar-form navbar-right">
 <div class="dropdown">
  <button class="btn btn-primary dropdown-toggle" type="button" id="userMenu" data-toggle="dropdown">
   <span class="glyphicon glyphicon-
user"></span>  {{session.user.userName}}  <span class="caret"></span>
  </button>
  <ul class="dropdown-menu" aria-labelledby="userMenu">
   <li><a href="/userHistory">User History</a></li>
   <li><a href="/logout">Log Out</a></li>
  </ul>
 </div>
</form>
<ul class="nav navbar-nav navbar-right">
  {{#navLink "/images"}}Images{{/navLink}}
  {{#navLink "/employees"}}Employees{{/navLink}}
  {{#navLink "/departments"}}Departments{{/navLink}}
</ul>
```

    - **If session.user** does not exist (ie: the user is not logged in), show the following HTML:

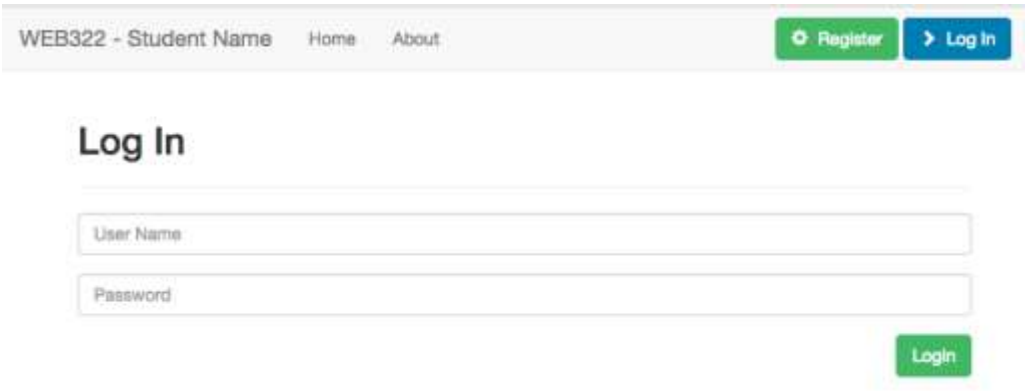
```
<form class="navbar-form navbar-right">
 <a href="/register" class="btn btn-success"><span class="glyphicon glyphicon-
cog"></span>  Register</a>
 <a href="/login" class="btn btn-primary"><span class="glyphicon glyphicon-chevron-
right"></span>  Log In</a>
</form>
```

login.hbs

- 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"	

- Above the form, we must have a space available for error output: Show the element: ** Error: {{errorMessage}} ** 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://morning-brook-58697.herokuapp.com/login. When complete, the form should look like this:



register.hbs

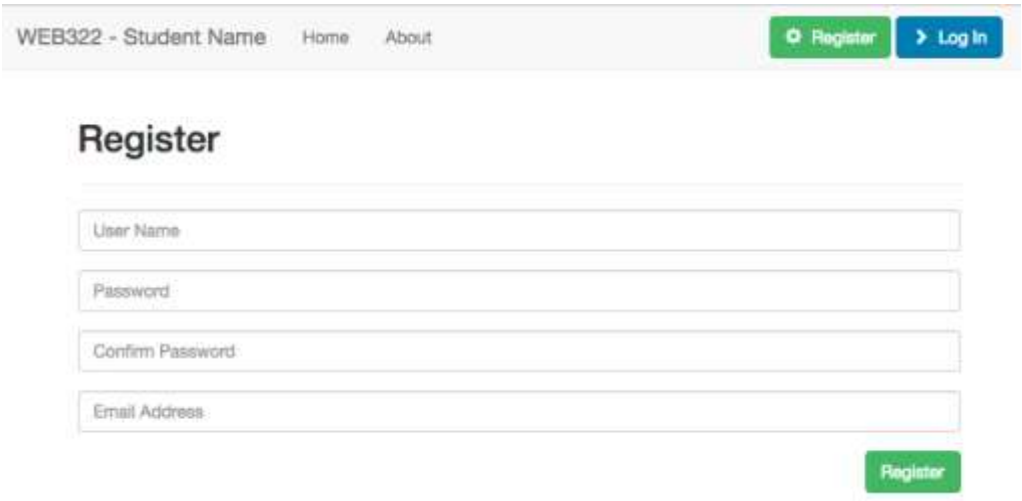
- This (new) view must consist of the "register form" which will allow the user to submit new credentials (using **POST**) to the **"/register"** POST route. **IMPORTANT NOTE:** this form is **only visible** if **successMessage** was **not** rendered with the view (refer to the "/register" POST route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

- Above the form, we must have a space available for error output: Show the element: ** Error: {{errorMessage}} ** only **if there is an errorMessage** rendered with the view.
- Additionally, we must also have a space available for success output: Show the elements: ** Success: {{successMessage}}   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 here: https://morning-brook-58697.herokuapp.com/register. When complete, the form should look like this:



userHistory.hbs

- 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
Client Information	This will be the **userAgent** value for the current **loginHistory** object

- Additionally, in the page title <h2>…</h2> block, add the code to show the **userName** and **email** properties of the logged in user (**session.user**) in the following format: **userName (email) History**
- 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://morning-brook-58697.herokuapp.com/userHistory. When complete, the form should look like this:

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 "data-service-auth.js" module).

1. Open the integrated terminal and enter the command: **npm install "bcryptjs" --save**
2. At the top of your **data-service-auth.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 in **Robo 3T** (previously known as: **Robomongo**):

- **Using Robo 3T:** double-click the "users" collection to see all of the entries. Next, select them all, **right-click** and select **"Delete Documents…"** from the pop up menu.

It is also possible to delete documents individually using **mLab**, ie:

- Click on your "**users**" collection and you will see the data. Red "trash" icons are provided next to each document in the collection that can be used to remove the document.

Step 2: Updating our data-service-auth.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 data-service-auth.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 Week 12 notes - to encrypt a value (ie: "myPassword123"), we can use the following code:

```
bcrypt.genSalt(10, function(err, salt) { // Generate a "salt" using 10 rounds
   bcrypt.hash("myPassword123", salt, function(err, hash) { // encrypt the password: "myPassword123"
      // TODO: Store the resulting "hash" value in the DB
   });
});
```

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

- If there was an error (ie, **if(err){ … }**) trying to **generate the salt** or **hash the password**, **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 Week 12 notes - to compare an encrypted (hashed) value (ie: **hash**) with a plain text value (i.e.: "**myPassword123**", we can use the following code:

```
bcrypt.compare("myPassword123", hash).then((res) => {
  // res === true if it matches and res === 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.user**) 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 match (ie: **res === true**) **resolve** the returned promise without any message

If the passwords do not match (ie: **res === false**) **reject** the returned promise with the message "Unable to find user: ***user***" where ***user*** is the **userData.user** value

Sample Solution

To see a completed version of this app running, visit: https://morning-brook-58697.herokuapp.com

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  Academic Policy.  No part of this
* assignment has been copied manually or electronically from any other source (including web sites) or
* distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
* Online (Heroku) Link: _____
*
********************************************************************************/
```

- Publish your application on Heroku & test to ensure correctness
- Compress your web322-app folder and Submit your file to My.Seneca under **Assignments** -> **Assignment** 6

Important Note:

- If the assignment will not run (using "node server.js") due to an error, the assignment will receive a **grade of zero (0).** Missing Heroku link in Comments Section will result in a -1 penalty.
- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.