# BTI325 Assignment 6

## Due: Wednesday Dec 9, 2020 @ 11:59 PM

## 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**)

## Specification:

For this assignment, we will be allowing users to "register" for an account on your BTI325 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 me for a clean version of the Assignment 5 files to start from.

## Part A: User Accounts / Sessions

### Step 1: Getting Started:

Creating a Database on MongoDB Atlas referring to week 8 notes.

### 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 (i.e., user & password) using our newly created **MongoDB database**:

1. Use npm to install **mongoose**
2. Create a new file at the root of your bti325-app folder called "**data-service-auth.js**" (same file level as your server.js and data-service.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 Week 8 notes)
5. Define a new "**userSchema**" according to the following specification:

<table>
<tr><th>Property</th><th>Mongoose Schema Type</th></tr>
<tr><td>userName</td><td>String (NOTE: this value must be unique)</td></tr>
<tr><td>password</td><td>String</td></tr>
<tr><td>email</td><td>String</td></tr>
<tr><td>loginHistory</td><td>[ { Property: Type, Property: Type } ]<br><br>NOTE: this will be an array of objects. The objects has the following specification:<br><br>
<table>
<tr><th>Property</th><th>Mongoose Schema Type</th></tr>
<tr><td>dateTime</td><td>Date</td></tr>
<tr><td>userAgent</td><td>String</td></tr>
</table>
</td></tr>
</table>

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

## data-service-auth.js - Exported Functions

Each of the below functions are designed to work with the **User** Object (defined by the above **userSchema** and will be **registered** to Mongo DB). 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**" method (or if an error was encountered, passes an **error message** via its "**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, **register** the ***userSchema*** to the collection ***users***, then **resolve()** the returned promise without returning any data
- If our connection has an error, we must **reject()** the returned promise with the provided error:

### registerUser(userData)

- This function will validate user's input and return meaningful errors if the data is invalid, as well as saving **userData** to the database (if no errors occurred). 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).

- To accomplish this:
    - If the value of the **.userName**, **.password** or **.password2** property is empty or only white spaces, **reject** the returned promise with the message: "**Error:** user name or password cannot be empty or only white spaces! ".
    - If the values of the **.password** property and the **.password2** property do not match, **reject** the returned promise with the message: "**Error: Passwords do not match**"
    - Otherwise (if the passwords not empty and successfully match), create a new **User** instance 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 its **err.code is 11000** (duplicate key), **reject** the returned promise with the message: "**User Name already taken**".
        - If an error (**err**) occurred 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 an error (**err**) **did not occur** at all, **resolve** the returned promise without any message

### checkUser(userData)

- **Authentication**

  This function does the "authentication". **userData** is the user's input, which is the parameter of this function. It will **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,

    - Invoke **findOne()** or **find()** method and filter the results by only searching for users whose **userName** property matches **userData.userName**, ie:

      **User.findOne({** userName **: userData.userName }) //or**
      **User.find({** userName **: userData.userName })** (**Hint**: refer to the Week 8 notes)

      **IMPORTANT: Be very careful, findOne() returns one object (document), find() returns an array of objects (documents). The following example uses findOne().**

      **Also, call exec() to return promise after finOne()**

        - If the **findOne()** promise resolved successfully, but **no user found**, **reject** the returned promise with the message "Unable to find user: ***userName***" where ***userName*** is the **userData.userName** value
        - If the **findOne()** promise resolved successfully, but the password doesn't match user's input, **reject** the returned promise with the error "Incorrect Password for user: ***userName***" where ***userName*** is the **userData.userName** value
        - If the **findOne()** promise resolved successfully and the **foundUser.password matches userData.password ,** (**foundUser** is the result from findOne(), i.e., the parameter of **then**

**((foundUser)=>{…})** ), then we must perform the following actions to record the action in the "loginHistory" array before we can resolve the promise with the **foundUser** object:

- Using the returned user object (i.e., **foundUser), push** the following object onto its "**loginHistory**" array:
  - {dateTime: (new Date()).toString(), userAgent: userData.userAgent}

  Note: userAgent information was obtained from browser below.
- Next, invoke the **updateOne** method on the **User** object where userName is **foundUser.userName** and **$set** the **loginHistory** value to **foundUser.loginHistory**.
- Finally, if the update was successful, **resolve** the returned promise **with the foundUser 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 **findOne()** promise was rejected, **reject** the returned promise with the message "Unable to find user: ***userName***" where ***userName*** 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 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:

```
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 its **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**
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 and example code).
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"** - there should be **14** routes) to use your custom **ensureLogin** helper middleware, make sure user has to login to be able to access these routes.

## 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 (add routes before 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

- The **User-Agent** request header contains a characteristic string that allows the network protocol peers to identify the application type, operating system, software vendor or software version of the requesting software user agent (MDN). Before we do anything, we must set the value of the client's "User-Agent" to the **request body property**, 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:  …   // complete it with authenticated user's userName
        email:   …      // complete it with authenticated user's email
        loginHistory:  …      // complete it with 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 of the "**catch**": **{errorMessage: err, userName: req.body.userName} - NOTE:** we are returning the user back to the login 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** 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">…</ul> element (this element has the "home" and "about" links) according to the following specification:
    - **If session.user** exists (ie: the user is logged in), show the following:
      {{#if session.user}}

```
<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), continue and complete the above logic, show the following:

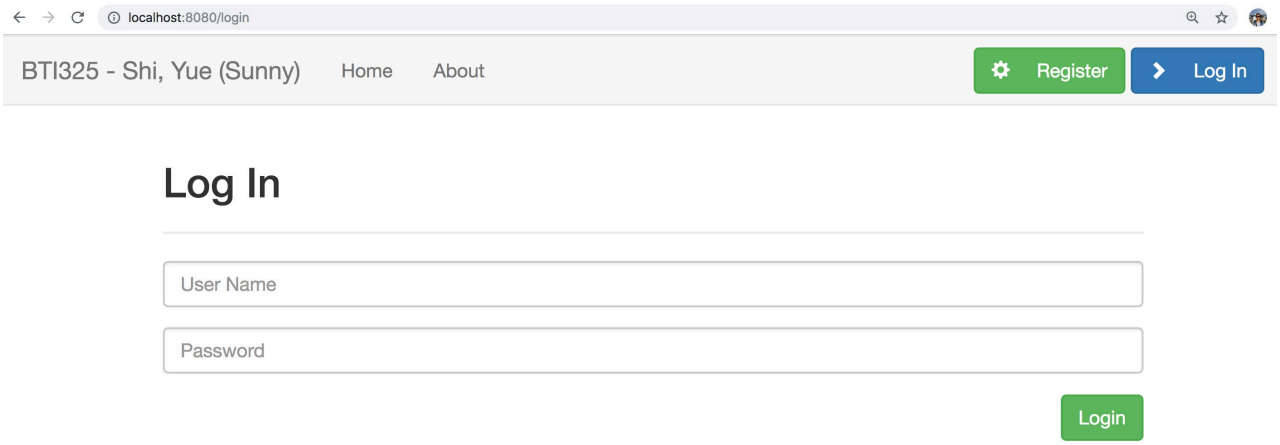
```
<form class="navbar-form navbar-right">
  <a href="/register" class="btn btn-success"><span class="glyphicon glyphicon-
cog"></span>  Register</a>
  <a href="/login" class="btn btn-primary"><span class="glyphicon glyphicon-chevron-
right"></span>  Log In</a>
</form>
```

### 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"<br>placeholder: "User Name"<br>required | **userName** if it was rendered with the view. Refer to the "/login" POST route above for more information |
| password | name: "password"<br>placeholder: "Password"<br>required | |
| submit (button) | text / value: "Login" | |

- Before 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://seneca-my.sharepoint.com/:u:/g/personal/sunny_shi_senecacollege_ca/EaA0S35qzBxLiFAlHIZa3P0Bxsv9nzOutnvsFKMjyJ-MOA?e=VK3kiv**
  When complete, the form should look like this:




register.hbs

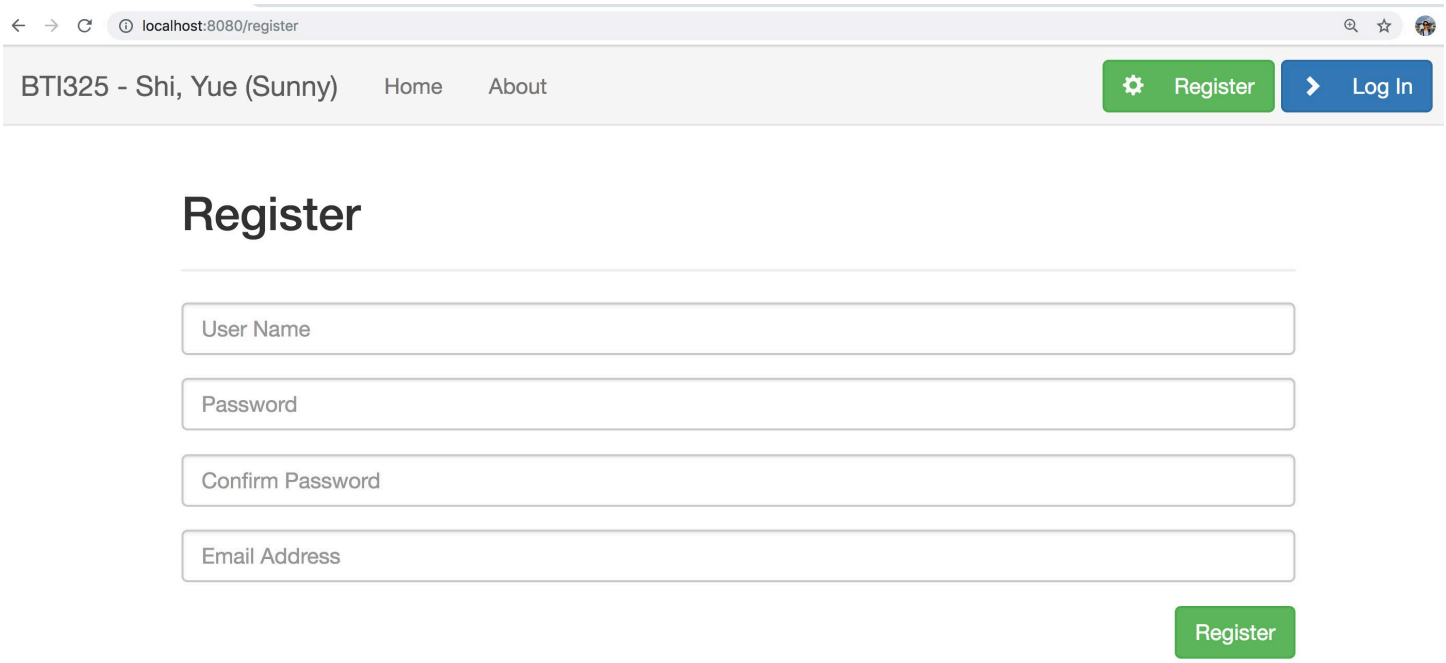
- This (new) view must consist of the "register form" which will allow the user to submit new credentials (using **POST**) to the **"/register"** POST route. **IMPORTANT NOTE:** this form is **only visible** if **successMessage** was **not** rendered with the view (refer to the "/register" POST route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

| input type | Properties |
|---|---|
| text | name: "userName"<br>placeholder: "User Name"<br>required |
| password | name: "password"<br>placeholder: "Password"<br>required |

| | |
|---|---|
| password | name: "password2"<br>placeholder: "Confirm Password"<br>required |
| email | name: "email"<br>placeholder: "Email Address"<br>required |
| submit (button) | text / value: "Register" |

- Display error output above the form: ** Error: {{errorMessage}} ,** only **if there is an errorMessage** rendered with the view.
- Additionally, for success output: Show the elements: ** Success: {{successMessage}} <a class="btn btn-success pull-right" href="/login">   Proceed to Log in    </a><br /><br /><br />** 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://seneca-my.sharepoint.com/:u:/g/personal/sunny_shi_senecacollege_ca/EUiKgpDVI0RGjPxqt0kVMfgBvG2PFz1tLKnyzj7ZcnINLA?e=X39wE9

  When complete, the form should look like this:



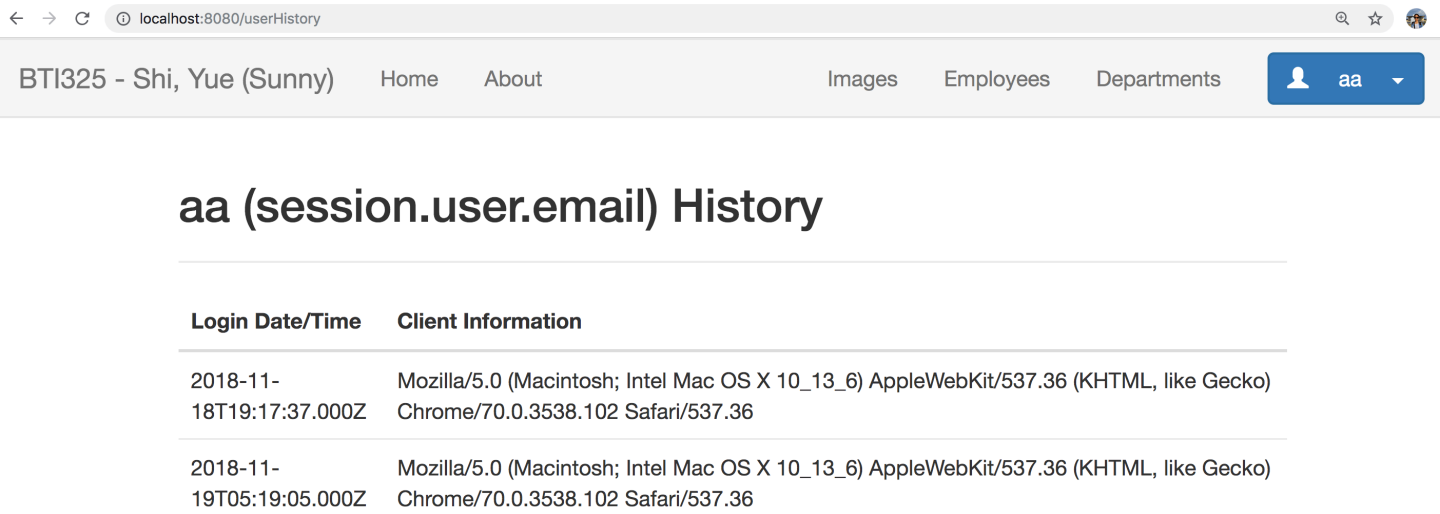

### userHistory.hbs

- 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 |

- In the page <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 form, refer to the HTML code available here: https://seneca-my.sharepoint.com/:u:/g/personal/sunny_shi_senecacollege_ca/ERD-ukZmpMFCnYM3WDLMSWEBlEMzM9L2lwQhDWb7QuA55g?e=FVYwP7
  When complete, the form should look like this:



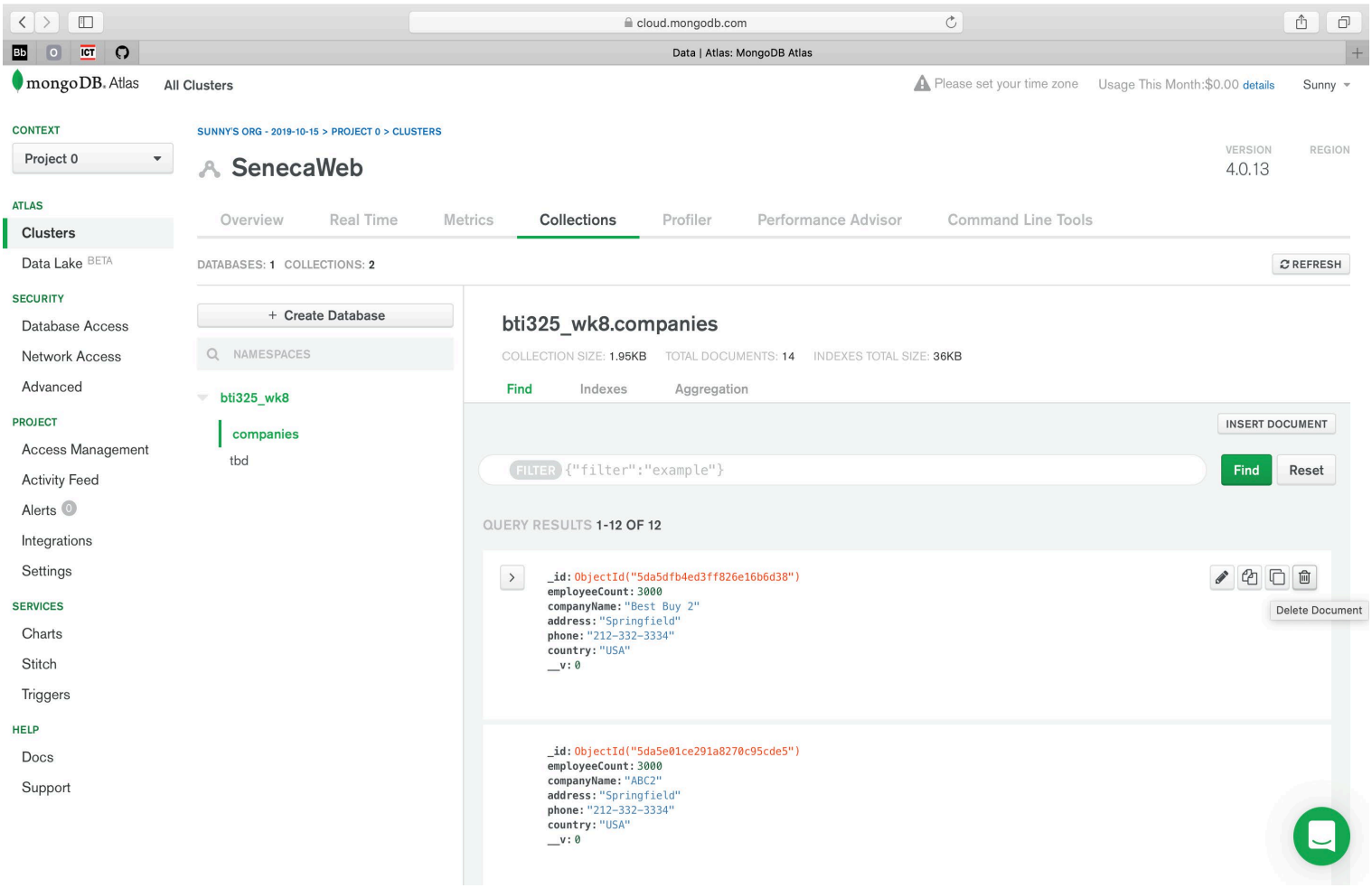

## Part B - Hashing Passwords

We will be using the "**bcrypt.js**" 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"**
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 which were saved as plain text. This can be done by writing your own code to delete the documents or in MongoDB Atlas using the right side “trash can” icon. The example screenshot:




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

Now that we have the bcrypt.js 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.js 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, hashValue) { // encrypt the password: "myPassword123"
    // TODO: Store the resulting "hashValue" value in the DB
  });
});
```

- Use the above code to **replace** the user entered password (ie: **userData.password**) with its **hashed version** (ie: **hashValue**) **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: **hashValue**) with a plain text value (ie: "**myPassword123**", we can use the following code:

```
bcrypt.compare("myPassword123", hashValue).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 foundUser.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.~~ Update loginHistory and resolve the foundUser as that in Part A.

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

### Assignment Submission:

- Before you submit, consider updating **site.css** to provide additional style to the pages in your app. (maybe something from Google Fonts)? This is your app for the semester, you should personalize it!

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

```
/*********************************************************************************
* BTI325 – Assignment 6
* 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 3rd party web sites) or distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
* Online (Heroku) Link: _____
*
********************************************************************************/
```

- **NEW: Add your Heroku Link as note/text on Blackboard submission. If the link doesn't work, please don't include it here or in the app. That's just wasting my time to open and it doesn't work. Please include a note saying the link doesn't work or any other problems your app may have. Thank you!**
- **Also,** compress (.zip) your bti325-app folder and submit the .zip file to My.Seneca under **Assignments** -> **A**6. I need to see your code.

### Important Note:

- Compress (.zip) your bti325-app folder and submit the .zip file to My.Seneca under **Assignments** -> **A**6
- Late submission will be penalized with 10% of this assignment marks for each school day. The last day for late submission is Dec 11, 2020 (Friday) @23:59.