WEB322 Assignment 7

Submission Deadline:

Friday, August 4th, 2017 @ 11:59 PM

Assessment Weight:

5% of your final course Grade

Objective:

Work with Client Sessions and data persistence to add user registration and Login/Logout functionality

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.

NOTE: If you are unable to start this assignment because Assignment 6 was incomplete - email your professor for a clean version of the Assignment 6 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.

Getting Started / Configuring Client Session Middleware:

Before we get started, 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.user}} 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 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**", "**/managers**", "**/departments**" or "**/department**" (this should be **12** routes) to use your custom **ensureLogin** helper middleware.

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 persistent data store (database). For this module, we will be once again making use of **MongoDB**:

1. Create a new file at the root of your web322-app folder called "**data-service-auth.js**"

2. "**Require**" your new "**data-service-auth.js**" module at the top of your **server.js** file as "**dataServiceAuth**"

3. Log into your **mLab** account from Assignment 6 and create a new **MongoDB Deployment** with a **Database Name** of **web322_a7** and a new user to access this database (**Hint**: refer to the Week 8 notes)

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:

Field Name	Type	Properties
user	String	must be **unique**
password	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()

- This method will look almost identical to your "initialize" method within your "data-service-comments.js" module, with the following 2 changes:
  - Your **let db = mongoose.createConnection(*connectionString*)** code will need to use the **correct connectionString** for your **web322_a7** database (from above)
  - Instead of **Comment = db.model("comments", commentSchema);** we must initialize our **User** object, ie: **User = db.model("users", userSchema);**

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: **.user**, **.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.user**, ie:
      User.find({ user: userData.user }) (**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.user** 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: ***user***" where ***user*** is the **userData.user** value
        - If the **find()** promise resolved successfully and the **users[0].password matches userData.password**, **resolve** the returned promise without a message
        - 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

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:

```
dataService.initialize()
.then(dataServiceComments.initialize)
.then(()=>{
  app.listen(HTTP_PORT, onHttpStart);
})
.catch((err)=>{
  console.log("unable to start the 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(dataServiceComments.initialize)
// add dataServiceAuth.initialize to the chain here
.then(()=>{
  app.listen(HTTP_PORT, onHttpStart);
})
.catch((err)=>{
  console.log("unable to start the server: " + err);
});
```

Adding New Routes:

Now that we have a back-end to store user credentials and our app has been modified to respect client sessions, we need to create **routes** that enable the user to register for an account and login / logout of the system. Once this is complete, we will create the corresponding **views**.

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 **dataService.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, user: req.body.user} - 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

- This "POST" route will invoke the **dataService.CheckUser(userData)** method with the POST data (ie: **req.body**).
    - If the promise resolved successfully, add the user to the session - ie: **req.session.user** (**Hint:** Refer to the Week 10 notes), by providing a **username** property with the value of **req.body.user**. Next, redirect the user to the "/employees" route, **ie: res.redirect('/employees');**
    - If the promise was rejected (**err**), **render** the **login** view with the following data: **{errorMessage: err, user: req.body.user} - 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('/');**

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/layout.hbs

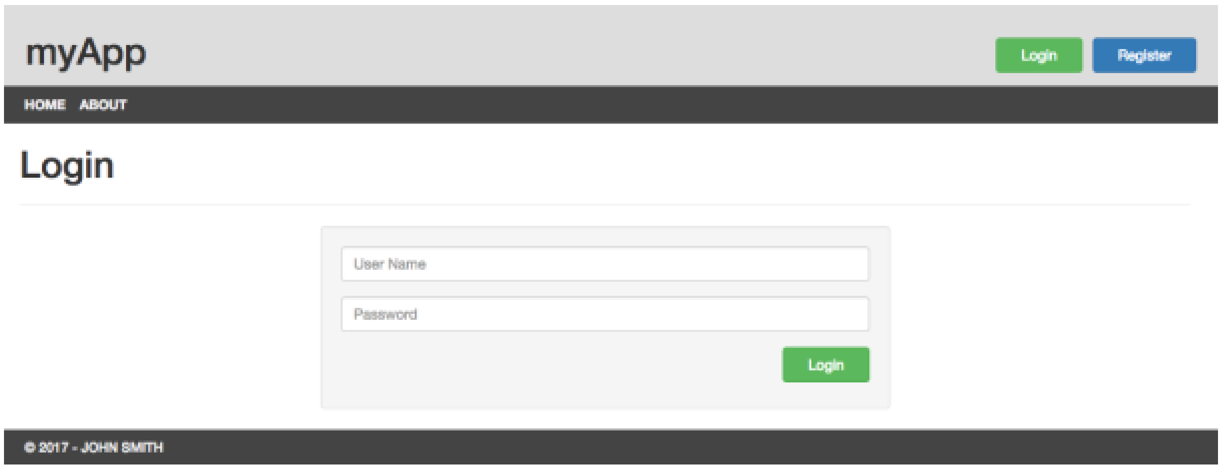
- To enable users to register for accounts, login / logout of the system, and conditionally hide / show menu items, we must make some small changes to our layout.hbs.
- In the <h1>MyApp</h1> block in the header, add the following 2 button groups:
    - **<span class="pull-right"><a href="/logout" class="btn btn-primary">&nbsp;&nbsp;&nbsp;Log Out: *user*&nbsp;&nbsp;&nbsp;</a></span>**
        - **NOTE:** this button must only be visible **if the user is currently logged in** (**Hint:** you can check #if session.user to confirm that a user is currently logged in). Also, the text ***user*** must contain the current username (**Hint:** you can get the current user name in the view with **session.user.username**)
    - **<span class="pull-right"><a class="btn btn-success" href="/login">&nbsp;&nbsp;&nbsp;Login&nbsp;&nbsp;&nbsp;</a> <a class="btn btn-primary" href="/register">&nbsp;&nbsp;&nbsp;Register&nbsp;&nbsp;&nbsp;</a></span>**
        - **NOTE:** these buttons must only be visible **if the user is not currently logged in**
- Lastly, we must **only** show the **All Employees**, **Managers**, **Departments** list of links in the **<nav> if the user is currently logged in**.

login.hbs

- This 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: "user" placeholder: "User Name" required	**user** 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 (<div class="well">…</div>), we must have a space available for error output: Show the element: **<div class="alert alert-danger"> <strong>Error:</strong> {{errorMessage}}</div>** only **if there is an errorMessage** rendered with the view.
- For layout guidelines, refer to the HTML code available here. When complete, the form should look like this:



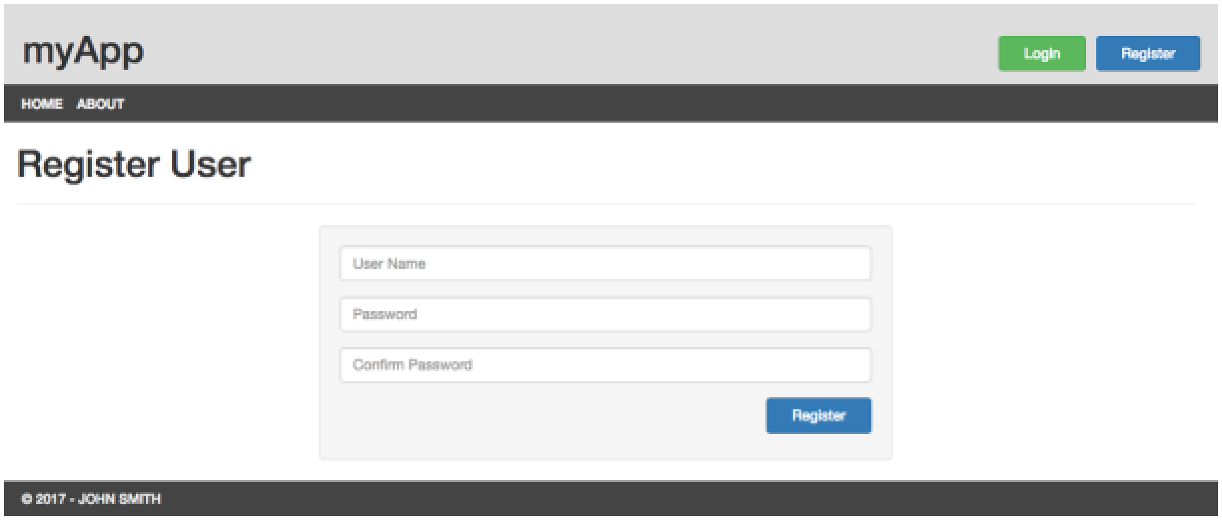
register.hbs

- This view must consist of the "register form" which will allow the user to submit new credentials (using **POST**) to the **"/register"** POST route. **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: "user" placeholder: "User Name" required	**user** 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	
submit (button)	text / value: "Register"	

- Above the form (<div class="well">…</div>), we must have a space available for error output: Show the element: **<div class="alert alert-danger"> <strong>Error:</strong> {{errorMessage}}</div>** only **if there is an errorMessage** rendered with the view.
- Additionally, above the form (<div class="well">…</div>), we must have a space available for success output: Show the elements: **<div class="alert alert-success"> <strong>Success:</strong> {{successMessage}}</div><a class="btn btn-success pull-right" href="/login">&nbsp;&nbsp;&nbsp;Proceed to Log in&nbsp;&nbsp;&nbsp;</a><br /><br /><br />** only **if there is a successMessage** rendered with the view.
- For layout guidelines, refer to the HTML code available here. When complete, the form should look like this:



Sample Solution

To see a completed version of this app running, visit: https://peaceful-kobuk-valley-83229.herokuapp.com/

Assignment Submission:

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

```
/*********************************************************************************
*  WEB322 – Assignment 07
*  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** 7

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).**
- **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.