CMPS 356 Enterprise Application Development - Spring 2019
Lab 12 – Securing Web applications: authentication, authorization, and confidentiality

Objective

The objective of this lab is to practice how to secure web applications. You will practice:

- **JSON Web Token (JWT)**: an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- Local Storage: client-side data storage. It can be used to store JWT tokens.
- Bycrpt.js: a Library to help you hash passwords.

Overview

This lab has two parts:
• **Part A:** Secure the Banking Web API (1.5h).
• **Part B:** Secure the Book Store Web API (1.5h).

PART A – Secure the Banking Web API

A. Protecting the Web API

Download the required packages

- Open the Banking app and run npm install to get all the dependencies.
- Install the following three packages by running the following command

  npm install --save jsonwebtoken bcryptjs jwt-decode
- Install the **nodemon** package globally.

  npm install --save -g nodemon

  To Run your application type in your terminal **nodemon** instead of "node app.js"

  Nodemon

- Once you run your application using **nodemon**, the **Nodemon** will watch the files in the directory in which **nodemon** was started, and if any files change, nodemon will automatically restart your node application

Export all the credentials to config file for easy management and security

- It is a good practice to move all your credentials into one file and then use that file to access all your credentials. This way we can add that file to the **.gitignore** and avoid accidentally sharing our private credentials publicly.
- Now to achieve this you should first create a **config** folder in your Banking App(server)
- Then inside the config folder create a file named **myCredentials.js**
- In **myCredentials**.js file add the following code

```
module.exports = {
   database: 'mongodb://127.0.0.1:27017/BankingApp',
   secret: 'mySecret'
};
```

- Open the **app.js** file and replace the database link by the **myCredentials.database**. Note you need to import the myCredetial file first.

```
const myCredential = require('./config/myCredentials');

//Open connection
mongoose.connect(myCredential.database)
   .then(() => console.log('database opened successfully'))
   .catch(err => console.log(err));
```

Routes for registration and login

Now we first need to create two routes. One that handles the Login and Another one that handles the registration

- Add the following two routes to your Banking App(server) and test
  - router.route('/api/users/login').post(acctService.loginUser)
  - router.route('/api/users/') .post(acctService.addUser)
- Try to create an account using POSTMAN then check if the account is created successfully
  - You can use the Login or Mongo Terminal or any third-party mongo database viewer app (Compas , Mongo Explorer, Robo3T …)
- Encrypt the user password in the database to hide from unwanted eyes
  - Currently, during the registration, the user password is saved directly to the database table. Now that is a bad practice as it could be stolen easily. Therefore, the best way to save the user password is to first encrypt it.
    There are many ways that you can encypt your password but in this lab we will use a popular encryption function called **bcrypt** implemented in the **bcryptjs** package.
  - Require the **bcryptjs** package in your **accountRepository**
  - Then Modify the **addUser** method as follows

```
async addUser(user) {
    const newUser = new User(user);
    //get the salt
    const salt = await bcrypt.getSalt(10);
    const hash = await bcrypt.hash(newUser.password , salt);
    newUser.password = hash;
    return newUser.save() }
```

- Now check if the password in your database it encrypted or not. If it is successfully encrypted then the password should look something like this

```
"password"    $2a$10$Ov4BCEltXFGKzoRouG0.bOp8uYop4Xke.OzbyNGqCNAJX.pHtIs0a
```

- Decrypt the user password in the database to authenticate the User (Login)
  - Because the password is encrypted we will need to decrypt it back before comparing it with the user login password.
  - To do that, first, open the **account-service** and modify the **loginUser** method as follows

```
async loginUser(req, res, next) {
  const authToken = req.body;
  const user = await accountRepo.getUser(authToken.username);
  if (user) {
    //check if the password matches
    const isMatch = await bcrypt.compare(authToken.password, user.password)
    if (isMatch)
      res.status(200).json(user)
    else
      res.sendStatus(401);
  }
  else
    res.sendStatus(401);
}
```

Using JSON Web Token to authenticate and protect private routes

So far we are able to encrypt/decrypt user password and also check if the user credentials are correct or not. But we will take this one step further and use the **JWT Token** to authenticate the user and also protect the user routes. To do this,

- Use JWT Token for future communications
  - If the user logged in successfully, then we will avoid exchanging the user name and password on each request. Instead, we will use the JWT token as a means of authentication.
  - To achieve this, first, we will modify the **loginUser** method above.

```
Import
const secret = require('../config/myCredentials').secret;
```

  - In the if (isMatch) you should do the following

```
...

    if (user) {
      //check if the password matches
      const isMatch = await bcrypt.compare(authToken.password, user.password)
      if (isMatch) {
        //now we will sign the user info using jwt
        let token = jwt.sign({user}, secret);
        res.status(200).json(token)
      }
        ...
```

  - to sign the user and communicate using a JWT token. This way we can protect the user name and password from being stolen.
  - To do that, first, open the **account-service** and modify the **loginUser** method as follows
- Protecting private routes using custom middleware

In the *router.js* file after the *registration and login* routes add the following route

```
router.route('/api/*')
  .all(acctService.verifyAuthToken)
```

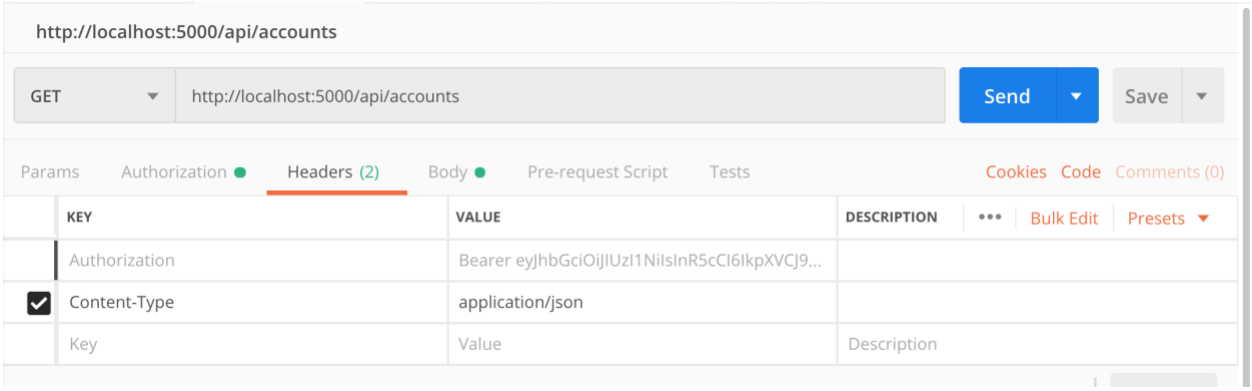
The above route will be called first before any route that starts with **/api**. The **.all** will handle all the **get/post/put/delete** methods. This way we will be protected all our routes in the /api and run the **verifyAuthToken** middleware method before letting the user access anything.

However, the **verifyAuthToken** does not exist yet. So you will need to add the following code to your the **account-service** class.

```
//Middleware to protect our routes
async verifyToken(req, res, next) {
```

```
    try {
        const bearerHeader = req.headers['authorization'];

        if (typeof bearerHeader !== 'undefined') {
            const bearerToken = bearerHeader.split(' ')[1];
            const authData = await jwt.verify(bearerToken, secret);
            req.token = bearerToken;
            req.authData = authData;
            next();
        }
        else {
            res.sendStatus(401)
        }
    }
    catch (err) {
        res.sendStatus(401)
    }
}
```

Test your code through Postman



B. Protecting the client routes

App Component

```
<Route exact path="/accts/:action" component={Accounts}/>
<Route path="/addTrans" component={TransForm}/>
```

Only allow logged In users to access this routes. Also, if a user us a clerk then you should only allow them to access the /addTrans. However, if a user is a manager then we only allow them to access the accts/:action route. To do this

1. we will track the following two states

   ```
   const [isAuthenticated, setIsAuthenticated] = useState(false);
   const [user, setUser] = useState();
   ```

2. Create the following two functions [Login and Logout]
3.

   ```
   const handleLogin = (user) => {
      setIsAuthenticated(true);
      setUser(user);
   };

   const handleLogout = () => {
      setIsAuthenticated(false);
      setUser(null);
   };
   ```

4. Create two more routes. One that loads the login and another one that loads the NavBar

```
<Router>
  <Route path="/login"
      render={(props) => {
        return <LoginForm onLogin={handleLogin} {...props} />
      }}
  />
  <Route path="/"
      render={(props) => {
        return <NavBar user={user} isAuthenticated={isAuthenticated}
                  onLogout={handleLogout} {...props}
        />
      }}
  />
</Router>
```

5. Create a component and name it **ProtectedRoute**. This component will handle the Routing to the protected routes.
   - If a user is authenticated it should check the user role and decide if they are allowed to access this page or not. If the user role allows them to access the page then it will direct them to the that page
   - otherwise they component should redirected the user to either login or home. If the issue was with "role" then redirect them to home otherwise redirect them to the login screen.

```
import React from 'react';
import {Route, Redirect} from 'react-router-dom'

  function ProtectedRoute({ component: Component,
                            isAuthenticated,
                            user,
                            authorizedRoles, ...rest }) {
    return (
      <Route
        {...rest}
        render={(props => {
          if (!isAuthenticated) {
            // Not logged in so redirect to login page with the from url to redirect to after login
            return <Redirect to={{pathname: "/login", state: {from: props.location}}}/>
```

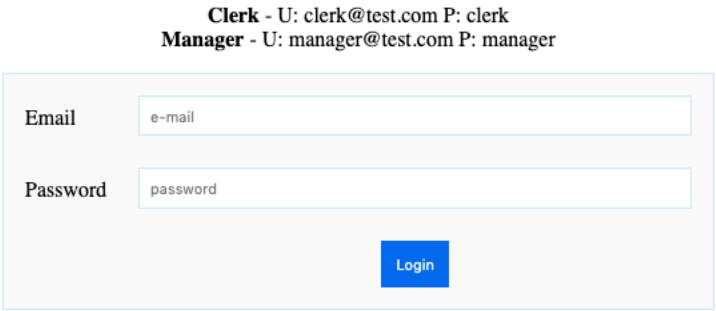
```
            }
            // check if route is restricted by role
            else if (authorizedRoles && authorizedRoles.indexOf(user.role) === -1) {
                alert(`You are NOT authorized to access ${rest.location.pathname}`);
                // role not authorised so redirect to home page
                return <Redirect to='/' />
            }
            // authorised so return component
            else
                return <Component {...props} />;
        }}
    />
  );
}
export default ProtectedRoute;
```

6. Open the App.js component and protect both routes "/addTrans" and "/accts/:action"

```
<Router>
  <Route path="/login"
      render={(props) => {
          return <LoginForm onLogin={handleLogin} {...props} />
      }}
  />
  <Route path="/"
      render={(props) => {
          return <NavBar user={user} isAuthenticated={isAuthenticated}
                    onLogout={handleLogout} {...props}
          />
      }}
  />
  <Switch>
    <ProtectedRoute path="/accts/:action" component={Accounts}
                isAuthenticated={isAuthenticated}
                user={user}
```

```
                authorizedRoles={["Manager"]}
        />
        <ProtectedRoute path="/addTrans" component={TransForm}
                isAuthenticated={isAuthenticated}
                user={user}
                authorizedRoles={["Clerk"]}
        />
    </Switch>
</Router>
```

Login Component



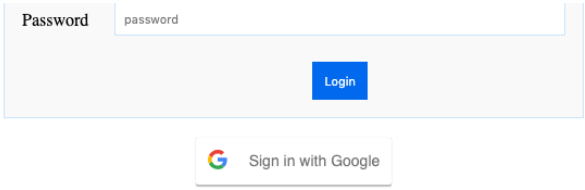
1. Create a function called handleLogin that takes a user object. This method should
    a. call the onLogin callback method to make the isAuthenticated True and to set the user
    b. depending on the user role you should redirect the user to either accts/list or /addTrans

```
const handleLogin = (user) => {
    onLogin(user);

    let redirectTo = user.role === "Manager" ? "/acct/list" : "/addTrans";
    if (location.state && location.state.from) {
        redirectTo = location.state.from;
    }

    history.push(redirectTo);
};
```

2. When the user clicks on the submit button call **handleLogin** that prevent the default behaviour of the user. Also, it shoulduser

```
const handleSubmit = e => {
    e.preventDefault();
    //alert(JSON.stringify(values));
    //ToDo: implement server-side authentication
    const user = {givenName: values.email, email: values.email};
    if (values.email === 'manager@test.com')
        user.role = 'Manager';
    else
        user.role = 'Clerk';

    handleLogin(user);
};
const handleLogin = (user) => {
  onLogin(user);

  let redirectTo = user.role === "Manager" ? "/acct/list" : "/addTrans";
  if (location.state && location.state.from) {
    redirectTo = location.state.from;
  }

  history.push(redirectTo);
};
```

Adding the Sign In Using Google Button



In this lab due to the time limitation we will not spend our time in creating a google client id. So you will be given the client ID that you can use for the lab. However, if you would like to create a google client id you can read this post that will guide you through the process https://developers.google.com/fit/android/get-api-key .

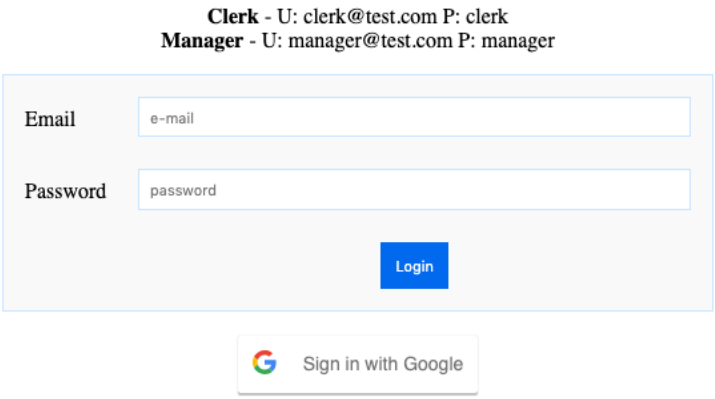
1. Install the following package **react-google-login**
2. **Import** it inside the login component
3. **Add the google button below the form**

```
<div className="align-center">
  <GoogleLogin
    clientId={googleClientId}
    onSuccess={handleGoogleResponse}
    onFailure={handleGoogleResponse}
  />
</div>
```

4. Create the callback method *handleGoogleResponse* inside the Login

```
const handleGoogleResponse = (response) => {
  const user = response.profileObj;
  user.role = 'Clerk';
  if (user) {
    handleLogin(user);
  }
};
```

- First we will create the Login Component

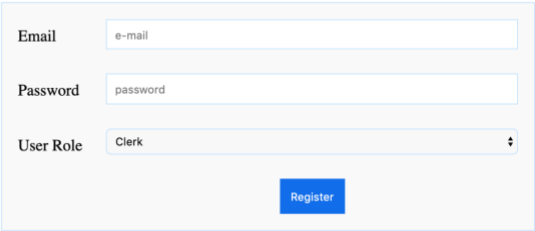


-

NavBar

In the App Component add the "/login" and "/" routes that load those components

```
<Router>
  <Route path="/login"
      render={(props) => {
        return <LoginForm onLogin={handleLogin} {...props} />
      }}
  />
  <Route path="/"
      render={(props) => {
        return <NavBar user={user} isAuthenticated={isAuthenticated}
                onLogout={handleLogout} {...props}
        />
      }}
  />
  <Switch>
    <ProtectedRoute path="/accts/:action" component={Accounts}
            isAuthenticated={isAuthenticated}
            user={user}
            authorizedRoles={["Manager"]}
    />
    <ProtectedRoute path="/addTrans" component={TransForm}
            isAuthenticated={isAuthenticated}
            user={user}
            authorizedRoles={["Clerk"]}
    />
  </Switch>
</Router>
```



Configuring the Account Service fetch

Whenever we send a request to the server we will need to attach the token. Therefore we should modify all our fetch API to have the authorization header. Here is one for the getAccounts

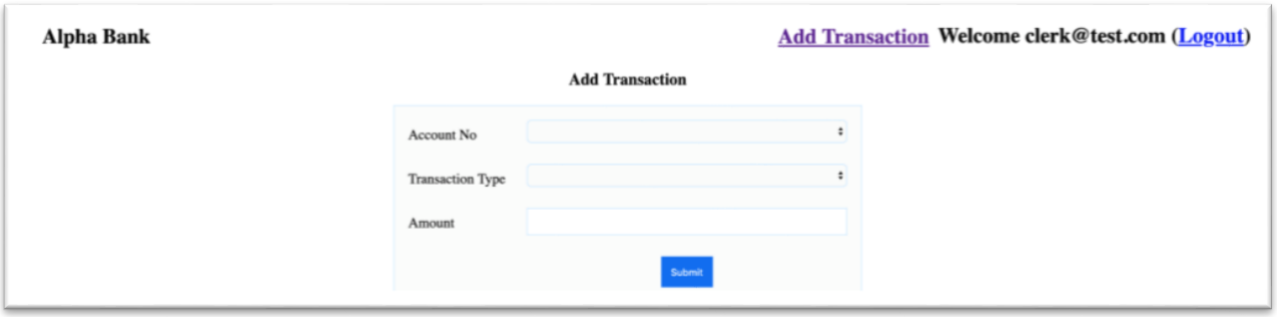
```
static async getAccounts(accountType, token) {
  const url = `${WebApiBaseUrl}/find/${accountType}`
  return await fetch(url, {
    headers: {
      'Authorization': token
    }
  });
}
```

You should add this header to all other services.

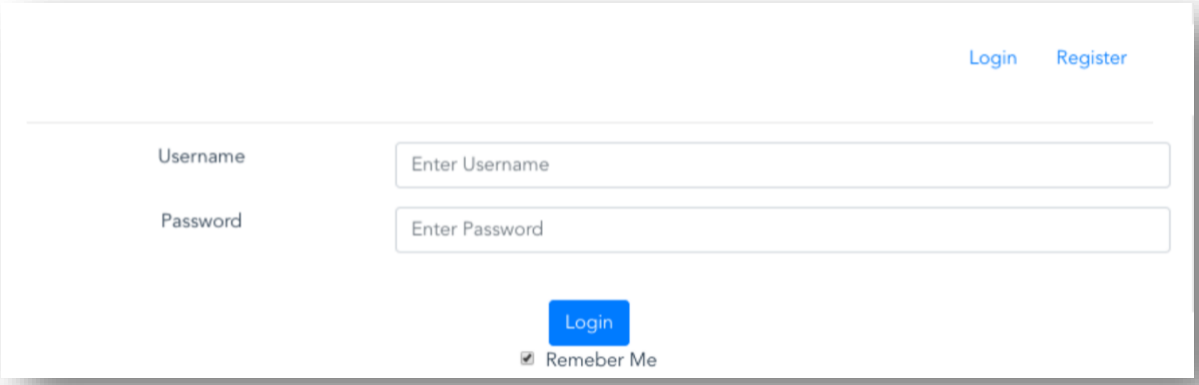
```
//helper Method to check if there is a token saved

function checkToken() {
  console.log(localStorage.getItem('token'));
  if (localStorage.getItem('token') != null
    && localStorage.getItem('token').length > 10) {
    return true;
  }
  else return false;
}
```

With this, we successfully protected our routes from being accessed without authorization. Now the continue and add the following.

1. Handle when the status is 403 then delete the local token
2. Complete the other components and services in both the client and server
3. When the user is logged in show the logout



When the user is Not LoggedIn show the Login and hide the Logout



PART B –BookStore APP

Using Similar techniques as PART A, protect the BookStore app Using JWT and bcrypt. Allow only logged in users to be able to add/remove/edit /borrow a book.

You can make use of the HTML, JavaScript and CSS provided in Lab 11 model solution.

You need to test your implementation as you progress and document your testing. After you complete the lab, fill in the ***Lab12-TestingDoc-Grading-Sheet.docx*** and save it in **Lab12-Security.js** folder. Sync your repository to push your work to Github.