Practice Project – Friends List Application Using Express Server with JWT

Estimated Time Needed: 1 hour

Overview:

In the CRUD lab you performed CRUD operations on transient data by creating API endpoints with an Express Server. In this lab, you will restrict these operations to authenticated users using JWT and session authentication.

- In this lab, the `friends` object will be a JSON/dictionary with `email` as the key and `friends` object as the value. The `friends` object is a dictionary with `firstName`,`lastName`, `DOB` mapped to their respective values. You will thus be using "body" from the HTTP request instead of "query" and "params".
- Only authenticated users will be able to perform all the CRUD operations.
- We will be testing the output of the endpoints on Postman.

Set-up : Create application

1. Open a terminal window by using the menu in the editor: Terminal > New Terminal.
2. Change to your project folder, if you are not in the project folder already.

   ```
   cd /home/project
   ```

3. Run the following command to clone the git repository that contains the starter code needed for this lab, if it does not already exist.

   ```
   [ ! -d 'nodejs_PracticeProject_AuthUserMgmt' ] && git clone https://github.com/ibm-developer-skills-network/nodejs_PracticeProject_A
   ```

4. Change to the directory **nodejs_PracticeProject_AuthUserMgmt** to start working on the lab.

   ```
   cd nodejs_PracticeProject_AuthUserMgmt
   ```

5. List the contents of this directory to see the artifacts for this lab.

   ```
   ls
   ```

Requisite packages for a server application

1. The packages required for this lab are defined as dependencies in `packages.json` as below. You can view the file on the file explorer.

   ```
    "dependencies": {
       "express": "^4.18.1",
       "express-session": "^1.17.3",
       "nodemon": "^2.0.19",
       "jsonwebtoken": "^8.5.1"
     }
   ```

2. In the terminal run the following command to install all the packages.

   ```
   npm install --save
   ```

This will install all the packages required for your server application to run.

Exercise 1: Understanding the User Authentication process

Let us understand the code in `index.js`.

1. Firstly, as the intent of this application is to provide access to the API endpoints only to the authenticated users, you need to provide a way to register the users. This endpoint will be a post request that accepts username and password through the **body**. The user doesn't have to be authenticated to access this endpoint.

   ```
   // Register a new user
   app.post("/register", (req, res) => {
       const username = req.body.username;
       const password = req.body.password;
       // Check if both username and password are provided
       if (username && password) {
           // Check if the user does not already exist
           if (!doesExist(username)) {
               // Add the new user to the users array
               users.push({"username": username, "password": password});
               return res.status(200).json({message: "User successfully registered. Now you can login"});
           } else {
               return res.status(404).json({message: "User already exists!"});
           }
   ```

```
    }
    // Return error if username or password is missing
    return res.status(404).json({message: "Unable to register user."});
});
```

2. You need to provide a manner in which it can be checked to see if the username exists in the list of registered users, to avoid duplications and keep the username unique. This is a utility function and not an endpoint.

```
// Check if a user with the given username already exists
const doesExist = (username) => {
    // Filter the users array for any user with the same username
    let userswithsamename = users.filter((user) => {
        return user.username === username;
    });
    // Return true if any user with the same username is found, otherwise false
    if (userswithsamename.length > 0) {
        return true;
    } else {
        return false;
    }
}
```

3. You will next check if the username and password match what you have in the list of registered users. It returns a boolean depending on whether the credentials match or not. This is also a utility function and not an endpoint.

```
// Check if the user with the given username and password exists
const authenticatedUser = (username, password) => {
    // Filter the users array for any user with the same username and password
    let validusers = users.filter((user) => {
        return (user.username === username && user.password === password);
    });
    // Return true if any valid user is found, otherwise false
    if (validusers.length > 0) {
        return true;
    } else {
        return false;
    }
}
```

4. You will now create and use a session object with user-defined secret, as a middleware to intercept the requests and ensure that the session is valid before processing the request.

```
app.use(session({secret:"fingerpint"},resave=true,saveUninitialized=true));
```

5. You will provide an endpoint for the registered users to login. This endpoint will do the following:

- Return an error if the username or password is not provided.
- Creates an access token that is valid for 1 hour (60 X 60 seconds) and logs the user in, if the credentials are correct.
- Throws an error, if the credentials are incorrect.

  Please make a note of it as you will be using this concept in the final project.

```
// Login endpoint
app.post("/login", (req, res) => {
    const username = req.body.username;
    const password = req.body.password;
    // Check if username or password is missing
    if (!username || !password) {
        return res.status(404).json({ message: "Error logging in" });
    }
    // Authenticate user
    if (authenticatedUser(username, password)) {
        // Generate JWT access token
        let accessToken = jwt.sign({
            data: password
        }, 'access', { expiresIn: 60 * 60 });
        // Store access token and username in session
        req.session.authorization = {
            accessToken, username
        }
        return res.status(200).send("User successfully logged in");
    } else {
        return res.status(208).json({ message: "Invalid Login. Check username and password" });
    }
});
```

6. You will now ensure that all operations restricted to auhtenticated users are intercepted by the middleware. The following code ensures that all the endpoints starting with **/friends** go through the middleware. It retrieves the authorization details from the session and verifies it. If the token is validated, the user is aunteticated and the control is passed on to the next endpoint handler. If the token is invalid, the user is not authenticated and an error message is returned.

```
// Middleware to authenticate requests to "/friends" endpoint
app.use("/friends", function auth(req, res, next) {
    // Check if user is logged in and has valid access token
    if (req.session.authorization) {
        let token = req.session.authorization['accessToken'];
        // Verify JWT token
        jwt.verify(token, "access", (err, user) => {
            if (!err) {
                req.user = user;
                next(); // Proceed to the next middleware
            } else {
                return res.status(403).json({ message: "User not authenticated" });
            }
```

```
        });
    } else {
        return res.status(403).json({ message: "User not logged in" });
    }
});
```

▶ Click to view the code

You have an express server that has been configured to run at port 5000. When you access the server with `/friends`, you can access the end points defined in routes/friends.js. But for doing this, you need to register as a new user in the `/register` endpoint and login with those credentials in the `/login` endpoint.

> Note: **You will now be implementing various CRUD operations for adding, editing and deleting friends by adding the necessary codes and further test the output on Postman.**

Exercise 2: Implement the GET method:

Navigate to `friends.js` file under the directory `router` and you will observe that the endpoints defined in them have skeletal,where you have to implement the get method.

1. Write the code in `router/friends.js` file inside `router.get("/",(req,res)=>{}` in the space provided to get all the user information using JSON string.

Hint: Refer to the CRUD lab from Exercise 2 in Module 3

▶ Click here to view the solution

Exercise 3: Implement the GET by specific email method:

1. Write the code inside `router.get("/:email",(req,res)=>{}` to view the user based on email but without using filter method.

Hint: Refer to the CRUD lab from Exercise 3

▼ Click here to view the solution

```
router.get('/:email', function(req, res) {
    // Retrieve the email parameter from the request URL and send the corresponding friend's details
    const email = req.params.email;
    res.send(friends[email]);
});
```

Exercise 4: Implement the POST method:

1. Paste the below code inside `router.post("/",(req,res)=>{}` to add the new user to the JSON/dictionary.And also update the codes in the places mentioned.

```
router.post("/", function(req, res) {
    // Check if email is provided in the request body
    if (req.body.email) {
        // Create or update friend's details based on provided email
        friends[req.body.email] = {
            "firstName": req.body.firstName,
            // Add similarly for lastName
            // Add similarly for DOB
        };
    }
    // Send response indicating user addition
    res.send("The user" + (' ') + (req.body.firstName) + " Has been added!");
});
```

Exercise 5: Implement the PUT method:

1. Paste the below code inside `router.put("/:email", (req, res) => {}` to modify the friend details.And also add the codes in the places mentioned.

```
router.put("/:email", function(req, res) {
    // Extract email parameter from request URL
    const email = req.params.email;
    let friend = friends[email];  // Retrieve friend object associated with email
    if (friend) {  // Check if friend exists
        let DOB = req.body.DOB;
        // Add similarly for firstName
        // Add similarly for lastName
        // Update DOB if provided in request body
        if (DOB) {
            friend["DOB"] = DOB;
        }
        // Add similarly for firstName
        // Add similarly for lastName
        friends[email] = friend;  // Update friend details in 'friends' object
        res.send(`Friend with the email ${email} updated.`);
    } else {
        // Respond if friend with specified email is not found
        res.send("Unable to find friend!");
```

```
    }
});
```

Exercise 6: Implement the DELETE method:

1. Paste the below code inside `router.delete("/:email", (req, res) => {}` to delete the friend information based on the email.

Hint:Refer to the CRUD lab from Exercise 6

▶ Click here to view the code

Run the server to view the output

1. In the terminal, ensure that you are in the `/home/projects/nodejs_PracticeProject_AuthUserMgmt`directory.
2. Install all the packages that are required for running the server

   ```
   npm install
   ```

3. Start the Express server.

   ```
   npm start
   ```

Exercise 7: User registration, login & testing the endpoints using Postman:

Go to Postman and go to a new HTTP request window (as you did in `Hands-on Lab - CRUD operations with Node.js`)

User registration

1. Submit a POST request on the endpoint – `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/register` using the below JSON paramters in the 'body' of the request.

- Select 'Body' >> 'raw' >> 'JSON' and pass the parameters.

  ```
  {"username":"user2", "password":"password2"}
  ```

  Note: "user2" & "password2" are used for reference. You can use any username & password.

2. It should return the output as `{"message": "User successfully registred. Now you can login"}`.

User login

1. Submit a POST request on the endpoint – `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/login` using the above username and password in the same JSON format in the 'body' of the request.
2. It should return the output as `User successfully logged in`.

Testing the endpoints on Postman:

- The below is similar to what you performed in `Hands-on Lab - CRUD operations with Node.js`:

1. Submit a GET request on the 'friends' endpoint – `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/friends` and ensure that you see all the friends that have been added in the code.
2. Submit a GET request on the endpoint – `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/friends/<email>` and ensure the details of that particular friend are returned.

For the user 'johnsmith@gamil.com', it will be 'johnsmith@gamil.com'" target="_blank" rel="noopener noreferrer"">https://XXXXXXXXXX-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/johnsmith@gamil.com'

3. Add a new friend using a POST request on the endpoint – `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/friends/` by using the below format in the request body:

   ```
   {"email":"andysmith@gamil.com","firstName":"Andy","lastName":"Smith","DOB":"1/1/1987"}
   ```

- The above is a reference for adding a new user with email andysmith@gamil.com.

Note: Please ensure using fictional/non-existent email domains by creating one or changing the spelling as done here (`gamil.com`). Avoid using actual email domains like gmail.com, yahoo.com.

4. Update a friend attribute(firstName, lastName, DOB) using a PUT request on the endpoint – `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/friends/<email>`

- Use the below format in the request body for updating the DOB as **1/1/1989** & likewise for updating 'firstName','lastName'.

  `{"DOB":"1/1/1989"}`

5. Delete a friend by sumbitting a DELETE request on the endpoint `https://<your-snlabs-username>-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/friends/<email>`.

Modifying the access token validity:

1. Consider the below code snippet for access token validity which we have observed earlier in index.js file.

```
    if (authenticatedUser(username, password)) {
        // Generate JWT access token
        let accessToken = jwt.sign({
            data: password
        }, 'access', { expiresIn: 60 * 60 });
        // Store access token and username in session
        req.session.authorization = {
            accessToken, username
        }
        return res.status(200).send("User successfully logged in");
    } else {
        return res.status(208).json({ message: "Invalid Login. Check username and password" });
    }
});
```

2. Now edit the 'expiresIn' attribute to 60 seconds to check the validity:

▶ Click here to view the code

3. Repeat the User Registration & User Login steps on Postman to verify the changes.

- If you submit the login request within 60 seconds of generating the access token, you will be authenticated. Else, you will get the message
  `{message: "Invalid Login. Check username and password"}`.

Congratulations! You have completed the practice project for performing CRUD operations on an Express server using Session & JWT authentication and tested them using Postman.

Summary:

In this lab, we have performed CRUD operations for the given user details on an Express server using Session & JWT authentication and tested them using Postman.

Author(s)

Lavanya T S

Sapthashree K S

K Sundararajan