Lesson 5-Building RESTful APIs with MERN Stack and Redux Toolkit

Topics

a. Redux Thunk Middleware in Redux Toolkit for Asynchronous API calls with Axios
    a. extrareducers
b. Consume REST APIs using Axios
c. MongoDb Database
d. Creating REST API Routes
e. Testing API Routes using Thunder Client
f. User Login Authentication
    a. Bcrypt
    b. UseNavigate Hook
    c. useParams Hook

Activity 5 – RestFul API & Registration-Login

The objectives of this activitiy are to:

a) Design and implement RESTful APIs using the MERN stack (MongoDB, Express, Node.js) to perform CRUD operations.
b) Integrate Redux Toolkit with a React front-end to manage asynchronous actions and handle API requests.

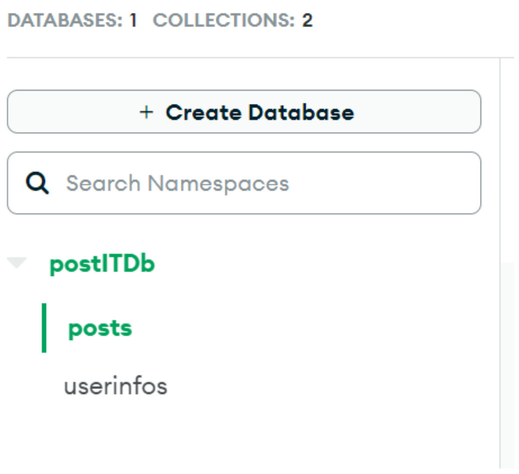
PART 1 – MongoDb Setup

Before starting the activity, ensure that you have created the MongoDb database in

https://www.mongodb.com/atlas/database.

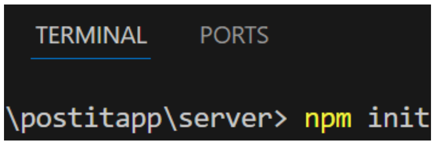
The project details are as follows:

Project Name	PostITApp
Cluster Name	PostITCluster
Database Name	postITDb
Collections	userInfos, posts
Database Username	*You provide your own*
Database Password	*You provide your own*



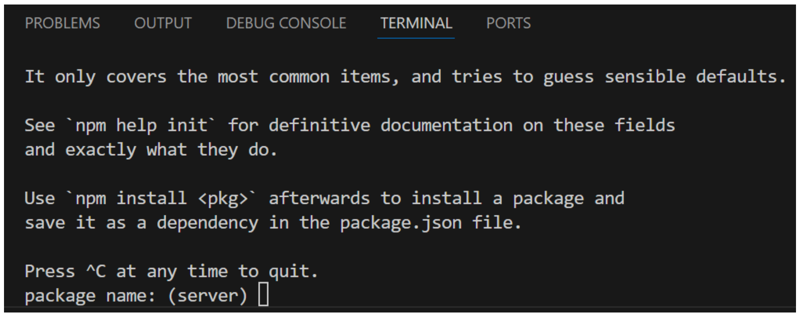
PART 2-SERVER-SIDE DEVELOPMENT USING NodeJS and Express

1. Open a new terminal in VSCode and navigate to the server folder.
    a. Execute the command to initialize the server.



```
\postitapp\server> npm init
```

    b. Accept the default server settings by pressing Enter.

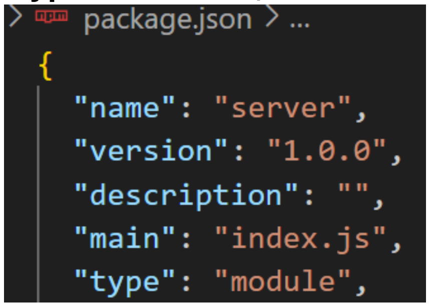


```
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (server)
```

2. Install the libraries.
    a. **npm install express mongoose cors**
3. In the server folder. Create a file index.js. This file in is the entry point for your server.

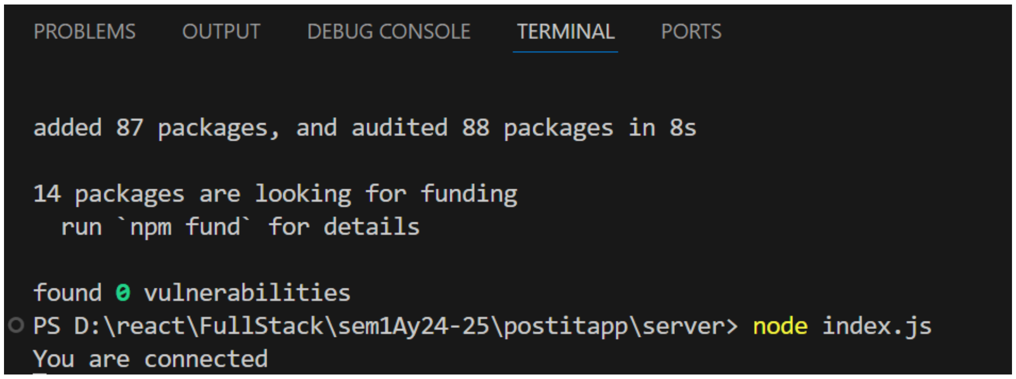
```
import mongoose from "mongoose";
import cors from "cors";

const app = express();
app.use(express.json());
app.use(cors());

//Database connection
const connectString ="";

mongoose.connect(connectString, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

app.listen(3001, () => {
  console.log("You are connected");
});
```

4. Open your package.json and add the following after the main…
   "type":"module",



```
> package.json > ...
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
```

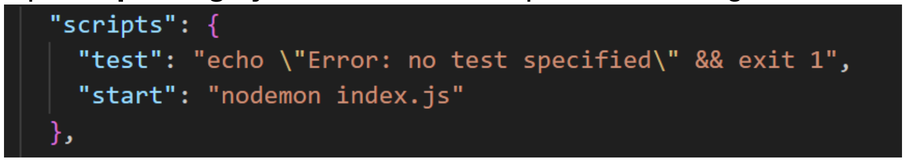
5. Run your server by executing the command: **node index.js**



PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

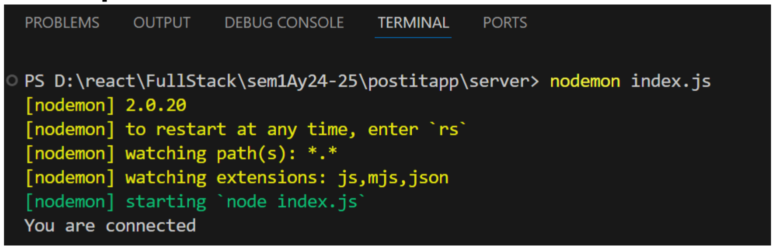
```
added 87 packages, and audited 88 packages in 8s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\react\FullStack\sem1Ay24-25\postitapp\server> node index.js
You are connected
```

6. However, every time you have changes in your index.js, you need to run again the server. This can be cumbersome to do. In order to solve this, we can use **nodemon**.
   a. Install nodemon.
      npm install nodemon
   b. Update **package.json** and write script for executing nodemon.



```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js"
  },
```

   c. Run your **nodemon** using the script. In this way, there is no need to explicitly stop and restart the server when there are changes in index.js. Use "**npm start**" instead.

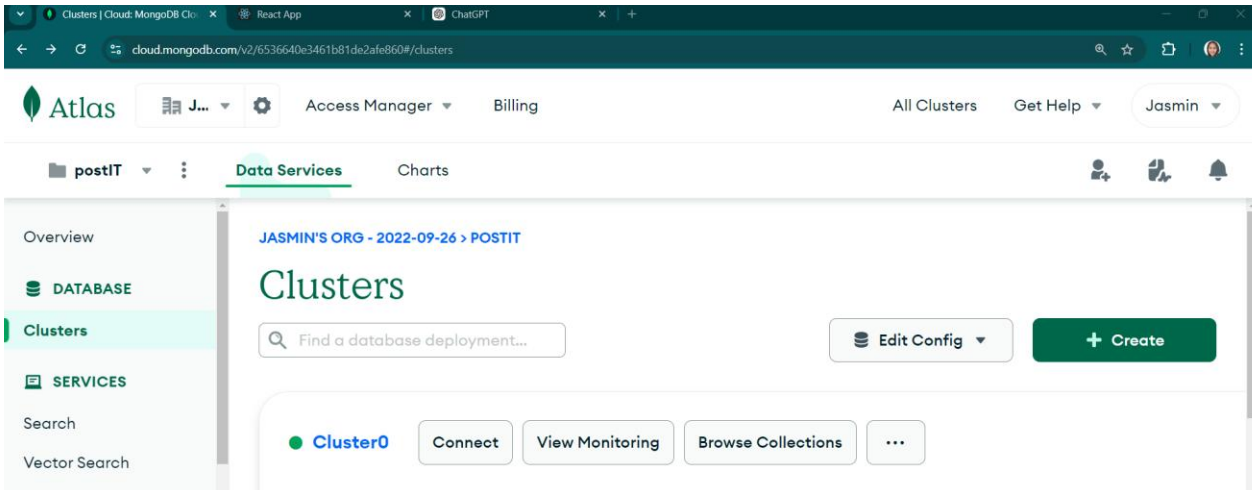


PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

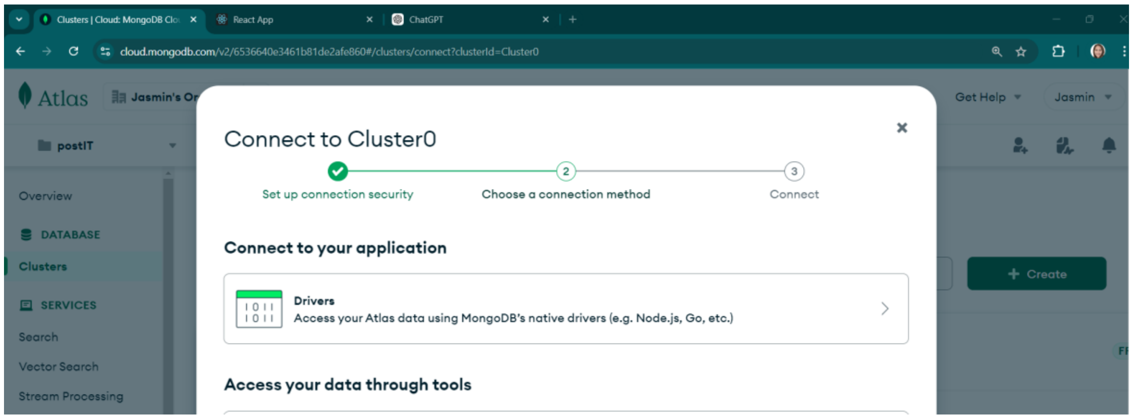
```
PS D:\react\FullStack\sem1Ay24-25\postitapp\server> nodemon index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
You are connected
```

PART 3-CONNECTING TO MONGODB

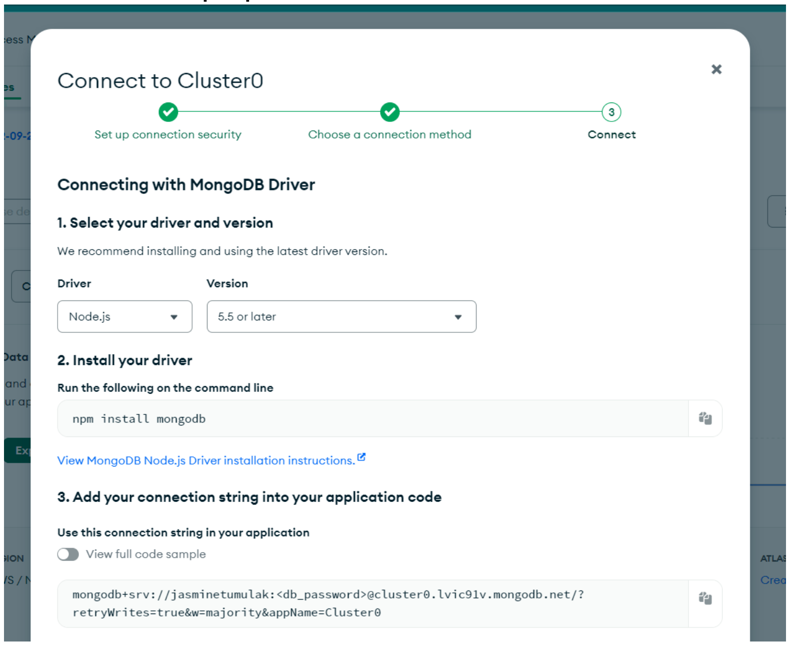
7. Login to your MongoDb Atlas account. Go to the specific project. Click the Clusters menu.



8. Select the Connect to you Connect to your Application.


Clusters | Cloud: MongoDB Clo
React App
ChatGPT
cloud.mongodb.com/v2/6536640e3461b81de2afe860#/clusters/connect?clusterId=Cluster0
Atlas
Jasmin's Or
Get Help
Jasmin
postIT
Overview
DATABASE
Clusters
SERVICES
Search
Vector Search
Stream Processing
Create
Connect to Cluster0
Set up connection security
Choose a connection method
Connect
Connect to your application
Drivers
Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)
Access your data through tools

9. Follow the steps provided.



Connect to Cluster0

Set up connection security — Choose a connection method — Connect

Connecting with MongoDB Driver

1. Select your driver and version

We recommend installing and using the latest driver version.

Driver: Node.js

Version: 5.5 or later

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

View MongoDB Node.js Driver installation instructions.

3. Add your connection string into your application code

Use this connection string in your application

View full code sample

```
mongodb+srv://jasminetumulak:<db_password>@cluster0.lvic91v.mongodb.net/?
retryWrites=true&w=majority&appName=Cluster0
```

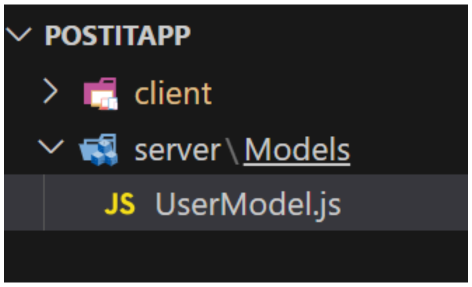
10. In **index.js**, provide the value for your database connection string. Example connection string:

    mongodb+srv://jasminetumulak:<db_password>@cluster0.lvic91v.mongodb.net/name of database?retryWrites=true&w=majority&appName=Cluster0

```
import express from "express";
import mongoose from "mongoose";

const app = express();

//Database connection
const connectString =
  "mongodb+srv://jasminetumulak:oman12345@cluster0.lvic91v.mongodb.net
/postITDb?retryWrites=true&w=majority&appName=Cluster0";

mongoose.connect(connectString, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

app.listen(3001, () => {
  console.log("You are connected");
});
```

PART 4-CREATING MODELS

11. In your **server** folder, create a folder: **Models**. In that folder, create a new file: **UserModel**.js.



12. In the file UserModel.js, write the code to create the **UserModel**.

```
import mongoose from "mongoose";

const UserSchema = mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  profilePic: {
    type: String,
  },
});

const UserModel = mongoose.model("userInfos", UserSchema);
export default UserModel;
```

13. Import the UserModel in index.js

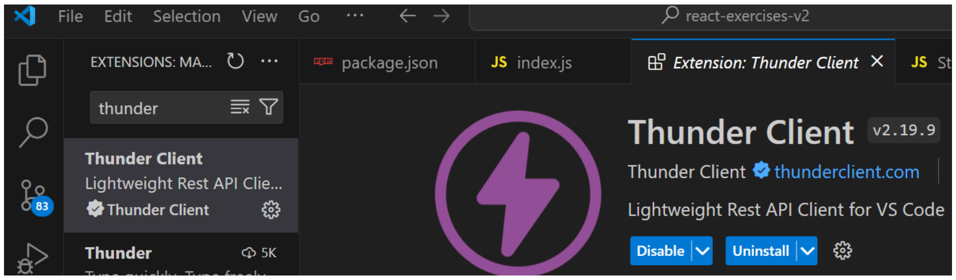
```
import UserModel from "./Models/UserModel.js";
```

PART 5-CREATING EXPRESS ROUTES

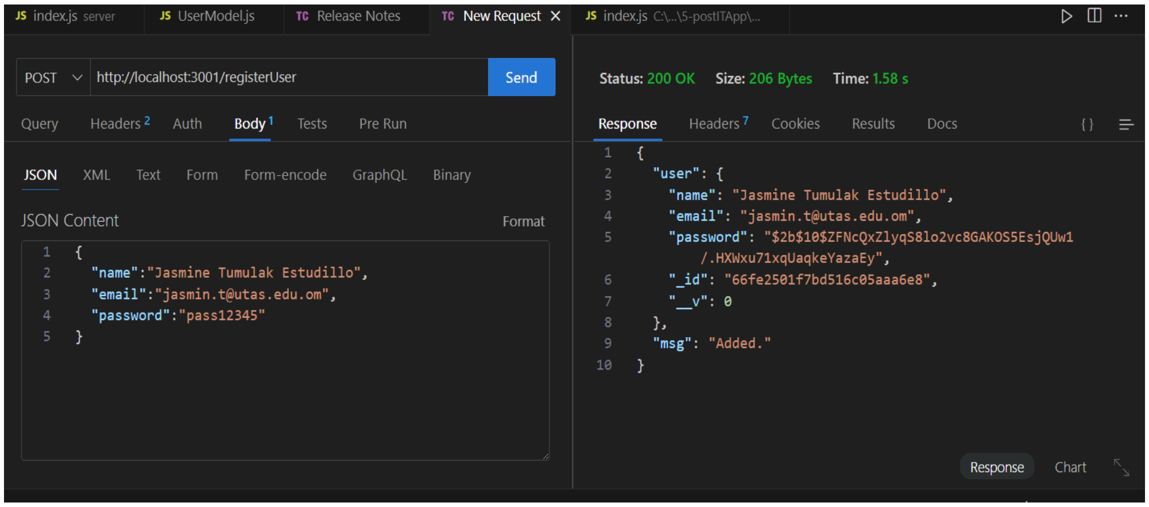
14. In **index.js**, setup an Express API route that will handle the request from the client to register the user.
    a. We will use the function **bcrypt** which is a library used to hash and securely store passwords. When a user registers on a website, instead of storing the raw password in the database (which is highly insecure), you hash it using bcrypt to protect it.
    b. Install the bcrypt library:
       npm install bcrypt
    c. Import the **bcrypt** library in index.js.

```
import bcrypt from "bcrypt";
import UserModel from "./Models/UserModel.js";
…..
app.post("/registerUser", async (req, res) => {
  try {
    const name = req.body.name;
    const email = req.body.email;
    const password = req.body.password;
    const hashedpassword = await bcrypt.hash(password, 10);

    const user = new UserModel({
      name: name,
      email: email,
      password: hashedpassword,
    });

    await user.save();
    res.send({ user: user, msg: "Added." });
  } catch (error) {
    res.status(500).json({ error: "An error occurred" });
  }
});

app.listen(3001, () => {
  console.log("You are connected");
});
```

15. Testing the Express routes using Thunder Client. In extensions, install Thunder Client.



16. After installing, the Thunder Client icon will be available. Then click on the New Request button. Click the body tab and provide example request values in JSON format.



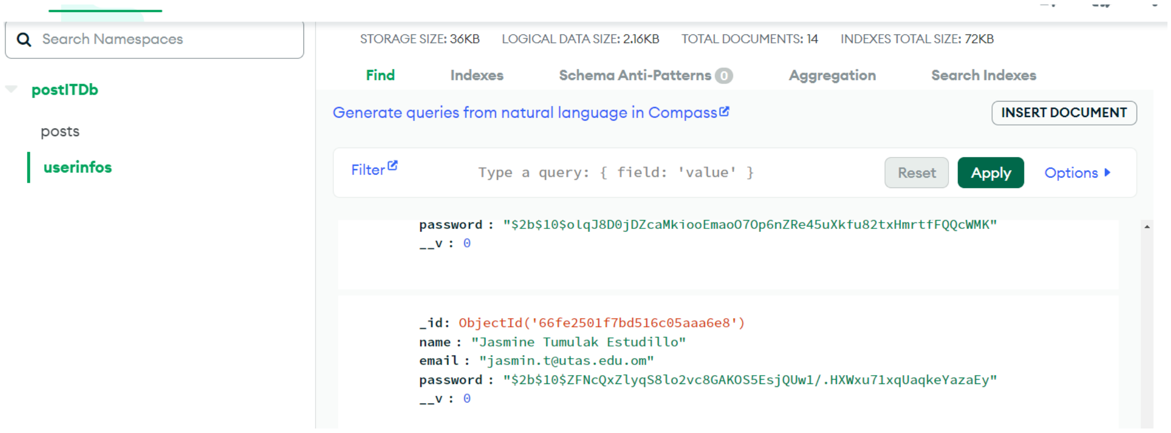
```
POST  http://localhost:3001/registerUser
```

```
{
  "name":"Jasmine Tumulak Estudillo",
  "email":"jasmin.t@utas.edu.om",
  "password":"pass12345"
}
```

Status: 200 OK   Size: 206 Bytes   Time: 1.58 s

```
{
  "user": {
    "name": "Jasmine Tumulak Estudillo",
    "email": "jasmin.t@utas.edu.om",
    "password": "$2b$10$ZFNcQxZlyqS8lo2vc8GAKOS5EsjQUw1
      /.HXWxu71xqUaqkeYazaEy",
    "_id": "66fe2501f7bd516c05aaa6e8",
    "__v": 0
  },
  "msg": "Added."
}
```

17. Login to your MongoDb account and check if the new information is saved in the collection.



STORAGE SIZE: 36KB   LOGICAL DATA SIZE: 2.16KB   TOTAL DOCUMENTS: 14   INDEXES TOTAL SIZE: 72KB

```
password : "$2b$10$olqJ8D0jDZcaMkiooEmao07Op6nZRe45uXkfu82txHmrtfFQQcWMK"
__v : 0

_id: ObjectId('66fe2501f7bd516c05aaa6e8')
name : "Jasmine Tumulak Estudillo"
email : "jasmin.t@utas.edu.om"
password : "$2b$10$ZFNcQxZlyqS8lo2vc8GAKOS5EsjQUw1/.HXWxu71xqUaqkeYazaEy"
__v : 0
```

Part 6 - Redux toolkit with async API example using createAsyncThunk

18. In your **client** folder, install the dependencies. First, install Axios which is a popular JavaScript library used for making HTTP requests from the browser or Node.js.

    npm install axios

19. Let us update our **Components/Register.js** file and comment (you may delete if you want to) the table that we used to display the value of our user state we used as test in the previous activity.

```
 {/* List of Users
         <table>
          .....
         </table> */}
```

20. In **src/Features/UserSlice.js**. Create the **thunk** using the **createAsyncThunk** and make sure to import the **createAsyncThunk** function. Also import Axios.

```
import { UsersData } from "../ExampleData";
import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";
import axios from "axios";

//const initialState = { value: [] }; //list of user is an object with empty
array as initial value
const initialState = { value: UsersData }; //Assign the data from the
exampleData

//Create the thunk
export const registerUser = createAsyncThunk();

export const userSlice = createSlice({
…..
});
```

21. Then provide the parameters to the function: **type** and **payloadcreator**. The code below is a basic format of thunk function. You need to make the callback function as an asynchronous function using the **async**. Also, we are using here the try…catch() block.

```
//Create the thunk (type and payloadCreator)
export const registerUser = createAsyncThunk("users/registerUser",
  async ()=>{
    try{

    }
    catch (error) {
      console.log(error);
    }
})
```

Thunk Name: registerUser
This is the name of the thunk. It will be used as the action type when dispatching this thunk.
Thunk Action Type: "users/registerUser"
The first argument to createAsyncThunk is a string representing the action type. It usually follows the pattern **"sliceName/actionName**".

Async Function: The second argument to **createAsyncThunk** is an asynchronous function that sends a POST request to the server using Axios. If the request is successful, it extracts the user data from the response and returns it. This returned value will be used as the payload when the thunk completes successfully.

22. Use Axios to send a request to the server and pass along with request the request body.

```
//Create the thunk (type and payloadCreator)
// Thunk Name : users/registerUser
//Thunk Action Type: "users/registerUser"
export const registerUser = createAsyncThunk(
    "users/registerUser",
    async (userData) => {
      try {
        //sends a POST request to the server along the request body
object
        const response = await
axios.post("http://localhost:3001/registerUser", {
          name: userData.name,
          email: userData.email,
          password: userData.password,
        });
        console.log(response);
        const user = response.data.user; //retrieve the response
from the server
        return user; //return the response from the server as
payload to the thunk
      } catch (error) {
        console.log(error);
      }
    }
  );
```

23. Update the initial state of the user state:

```
const initialState = {
    user: {},
    isLoading: false,
    isSuccess: false,
    isError: false,
  };
```

24. Create the **extraReducers** in the users slice of the Redux store.

```
export const userSlice = createSlice({
    name: "users", //name of the state
    initialState,
    reducers: {
      // Synchronous actions that update the state directly,
    },
    //builder.addCase(action creator(pending, fulfilled, rejected),
reducer)

    extraReducers: (builder) => {
      //Asynchronous actions that update the state directly,
      builder
        .addCase(registerUser.pending, (state) => {
          state.isLoading = true;
        })
        .addCase(registerUser.fulfilled, (state, action) => {
          state.isLoading = true;
        })
        .addCase(registerUser.rejected, (state) => {
          state.isLoading = false;
        });
    },
  }); //end of slice
```

Slice Name: "users"

This is the name of the slice. It will be used to generate action types and selectors associated with this slice.
Reducers: For synchronous actions that update the state directly.

Extra Reducers: Handle actions that are not simple synchronous actions but are produced by thunks.

Builder is an object provided as an argument to the **extraReducers** callback. It is used to define how the state should be updated in response to different actions.

25. In Register.js, import the extrareducer registerUser so we can use this in the Register component.

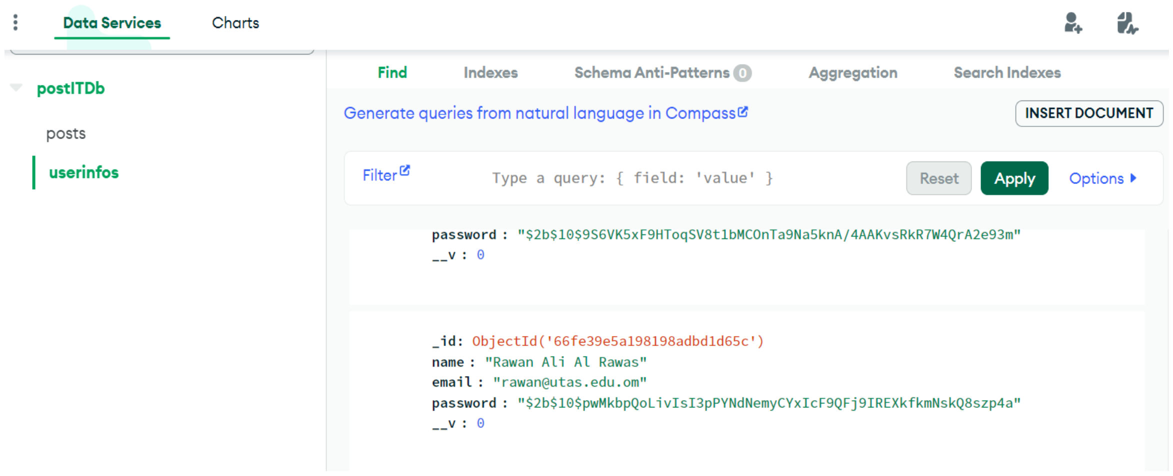
```
import { registerUser } from "../Features/UserSlice";
```

26. Then we can call in the component that needs to invoke the thunk. In this case, the **Register.js,** dispatch an action.

    *Note: This function was created in the previous activity. Simply change the function in the invoked by the dispatch hook to **registerUser** instead of the **addUser**.*

```
  // Handle form submission
  const onSubmit = (data) => {
    try {
      // You can handle the form submission here
      const userData = {
        name: data.name,
        email: data.email,
        password: data.password,
      };

      console.log("Form Data", data);
      alert("Validation all good.");
      dispatch(registerUser(userData)); // Dispatch an action to add
a new user by passing the user data to the Redux store
    } catch (error) {
      console.log("Error.");
    }
  };
```

27. You can test your code now. Input example data. Check your MongoDb database if it is saved.



```
password : "$2b$10$9S6VK5xF9HToqSV8t1bMCOnTa9Na5knA/4AAKvsRkR7W4QrA2e93m"
__v : 0

_id: ObjectId('66fe39e5a198198adbd1d65c')
name : "Rawan Ali Al Rawas"
email : "rawan@utas.edu.om"
password : "$2b$10$pwMkbpQoLivIsI3pPYNdNemyCYxIcF9QFj9IREXkfkmNskQ8szp4a"
__v : 0
```

28. After successfully saving the registration details, we will create the login functionality. So, after the user registration, the user must be redirected to the login page.
    a. In **Register.js**, import the **usenavigate**() hook from react-router-dom.

```
import { useNavigate } from "react-router-dom";
```

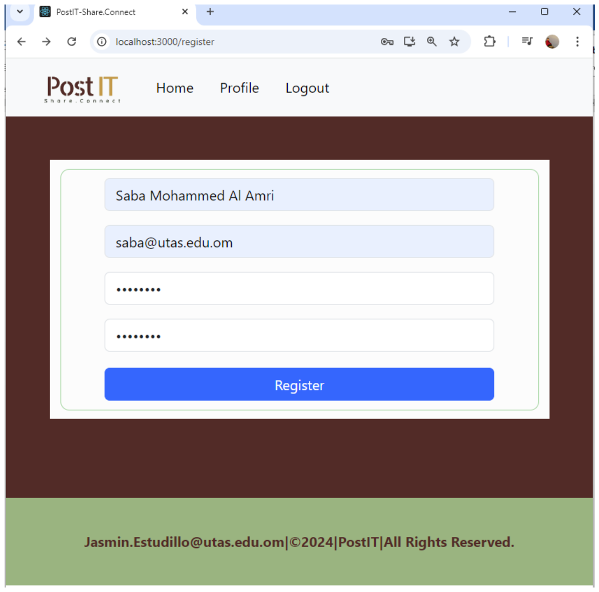
    b. Then declare a variable named navigate and assign it the value returned by the useNavigate() hook.

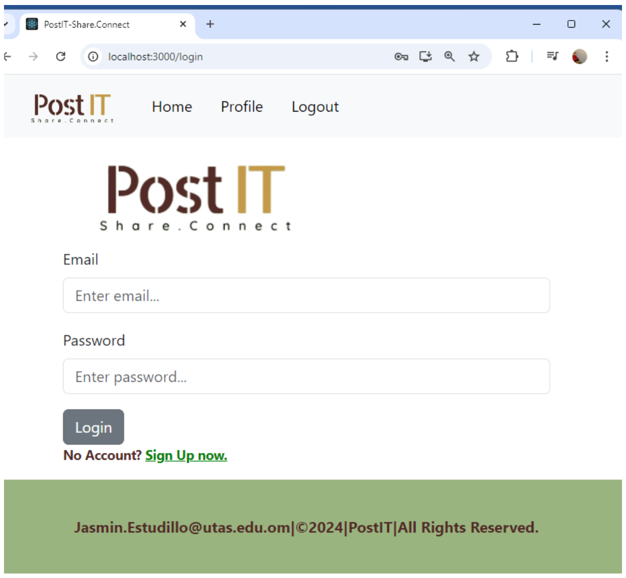
```
const Register = () => {
    …

      const dispatch = useDispatch(); //every time we want to
call an action, make an action happen
      const navigate = useNavigate() //declares a constant
variable named navigate and assigns it the value returned by
the useNavigate() hook.
    …
    }
```

c. In the **onSubmit()** function, after dispatching the action use the navigate function. The navigate is a function that you can use to control navigation within your React application when using React Router.

```
  // Handle form submission
  const onSubmit = (data) => {
    try {
      ……
      dispatch(registerUser(userData)); // Dispatch an action
to add a new user by passing the user data to the Redux store
      navigate("/login"); //redirect to login component
    } catch (error) {
      console.log("Error.");
    }
```

29. Test the navigate hook if the redirection is working. Input new data and click Register. The login component must be automatically rendered.

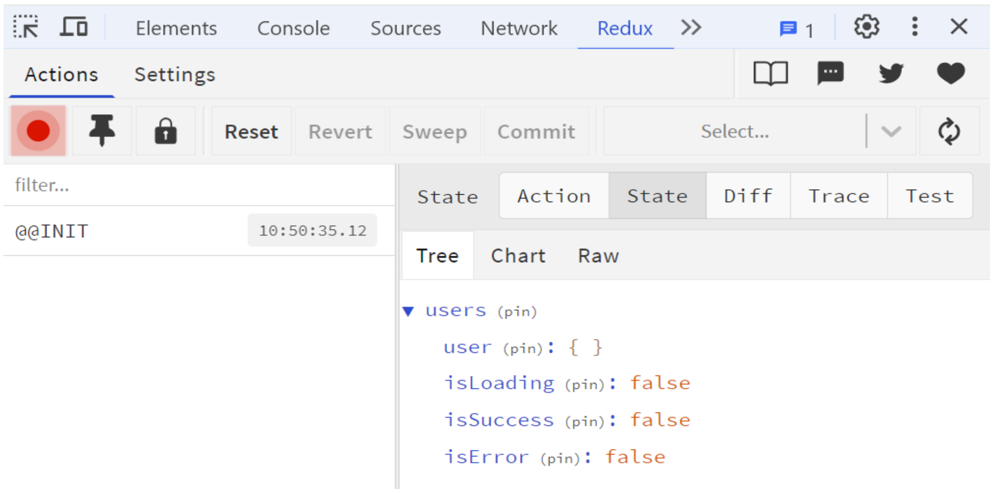




Note: We will implement in the next activities how to conditionally render the Header component.

PART 7- LOGIN IMPLEMENTATION

30. First check your store using the Chrome Dev Tools for the state of your application. At this point, the user state is empty.

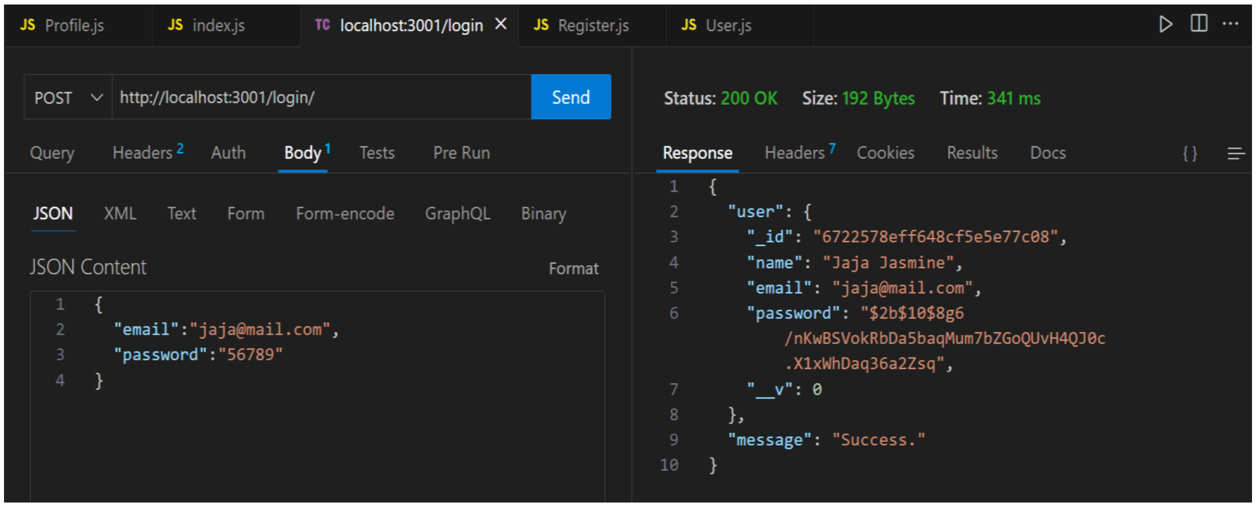


```
@@INIT    10:50:35.12

▼ users (pin)
    user (pin): { }
    isLoading (pin): false
    isSuccess (pin): false
    isError (pin): false
```

31. In **server/index.js**, let us create the API for login. The login functionality follows this logic:
    a. Retrieve the login details from the request (email and password)
    b. Query the database for the email, if email does not exist, send error response message to the client.
    c. If email exist, compare the hashed password from the database with the hashed password from the user input.
    d. If the same, send a response to the client. Include in the response user details and a successful message.

```
app.post("/login", async (req, res) => {
    try {
      const { email, password } = req.body; //using destructuring
      //search the user
      const user = await UserModel.findOne({ email: email });

      //if not found
      if (!user) {
        return res.status(500).json({ error: "User not found."
});
      }
      console.log(user);
      const passwordMatch = await bcrypt.compare(password,
user.password);

      if (!passwordMatch) {
        return res.status(401).json({ error: "Authentication
failed" });
      }

      //if everything is ok, send the user and message
      res.status(200).json({ user, message: "Success." });
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  });
```

32. Test the login API using Thunder Client.



http://localhost:3001/login/

```
{
  "email":"jaja@mail.com",
  "password":"56789"
}
```

Status: 200 OK Size: 192 Bytes Time: 341 ms

```
{
  "user": {
    "_id": "6722578eff648cf5e5e77c08",
    "name": "Jaja Jasmine",
    "email": "jaja@mail.com",
    "password": "$2b$10$8g6
        /nKwBSVokRbDa5baqMum7bZGoQUvH4QJ0c
        .X1xWhDaq36a2Zsq",
    "__v": 0
  },
  "message": "Success."
}
```

33. In src/Components/Login.js, import useState() hook.

```
import { useState } from "react";
```

34. Create the state variables for email and password. Create the event handler to assign the value of the input to the state.

```
const Login = () => {
  const [email, setemail] = useState();
  const [password, setpassword] = useState();
…..

              <Input
                id="email"
                name="email"
                placeholder="Email..."
                type="email"
                onChange={(e) => setemail(e.target.value)}
              />
….
}
```

35. In Login.js, create a function that will handle the logic of the login. Create an object that will contain the current values of the state variables. Write this function before the return statement.

```
  //function that will be invoked when the user clicks the login
button
  const handleLogin = () => {
    const userData = {
      email,
      password,
    };
  };
```

36. Call this function when the user clicks the Login button.

```
<Button
color="primary"
className="button"
onClick={() => handleLogin()}>
Sign in
</Button>
```

37. This time, we need to create a thunk for the login in our user slice in **src/Features/User.js.**

```
export const login = createAsyncThunk("users/login", async
(userData) => {
    try {
      const response = await
axios.post("http://localhost:3001/login", {
        email: userData.email,
        password: userData.password,
      });

      const user = response.data.user;
      console.log(response);
      return user;
    } catch (error) {
      //handle the error
      const errorMessage = "Invalid credentials";
      alert(errorMessage);
      throw new Error(errorMessage);
    }
  });
```

38. Then, we need to create extrareducers for the login thunk.

```
extraReducers: (builder) => {
    //Asynchronous actions that update the state directly,
    builder
…..
      .addCase(login.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(login.fulfilled, (state, action) => {
        state.user = action.payload; //assign the payload which is
the user object return from the server after authentication
        state.isLoading = false;
        state.isSuccess = true;
      })
      .addCase(login.rejected, (state) => {
        state.isLoading = false;
        state.isError = true;
      });
  },
…..
```

39. In **src/Components/Logins.js**, import the useDispatch and create a variable for which can be used to dispatch actions.

```
…..
import { useDispatch, useSelector } from "react-redux";

const Login = () => {
  const [email, setemail] = useState();
  const [password, setpassword] = useState();

  const dispatch = useDispatch();
…..
```

40. Import the login function Login.js.

```
import { login } from "../Features/UserSlice";
```

41. Dispatch the login action in the function handleLogin. Pass as argument the userData object.

```
…..
//function that will be invoked when the user clicks the login
button
  const handleLogin = () => {
    const userData = {
      email: email,
      password: password,
    };
  };
    dispatch(login(userData))  //dispatch a login action from the
user slice.
  };
…..
```

Using the useSelector() Hook

42. Now in the **src/Components/Login.js** component, we need to know what the status of the user state is so we can identify if login is successful or not. Import the useSelector() Hook.

```
import { useSelector } from "react-redux";
```

43. Use the **useSelector** Hook to retrieve the current value of the state from the store.

```
…..
const Login = () => {
  const [email, setemail] = useState();
  const [password, setpassword] = useState();

  const dispatch = useDispatch();
//Retrieve the current value of the state from the store, name of
state is users with a property user
  const user = useSelector((state) => state.users.user);
  const isSuccess = useSelector((state) => state.users.isSuccess);
  const isError = useSelector((state) => state.users.isError);

…..
```

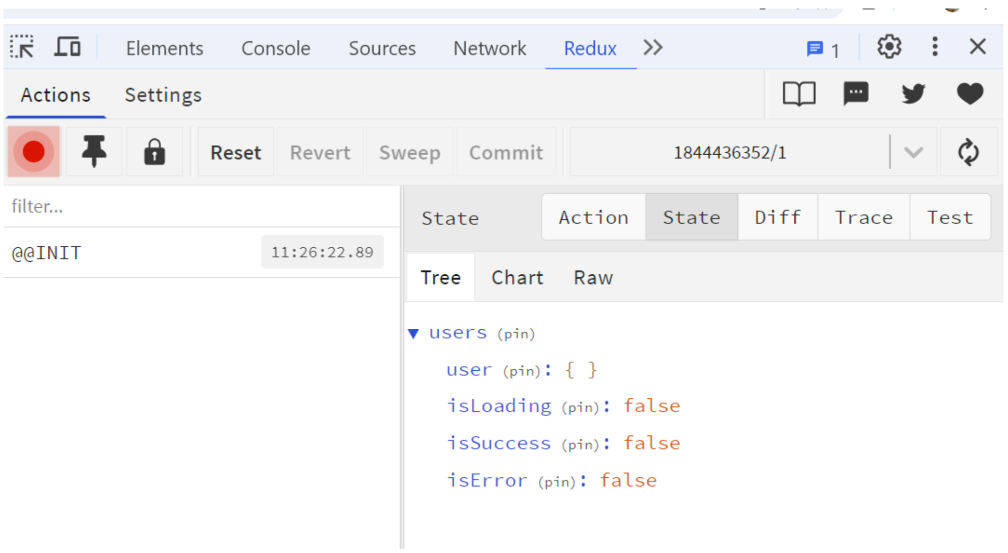
44. In **src/Components/Login.js**, import the useEffect Hook and the useNavigate Hook. Declare a variable for the

```
import { useEffect } from "react";
import { useNavigate } from "react-router-dom";

const Login = () => {
…..

  const dispatch = useDispatch();
  const navigate = useNavigate();
…..
```

45. Use the useEffect hook to determine which page to navigate based on the state of the application. Write this code before the return statement.

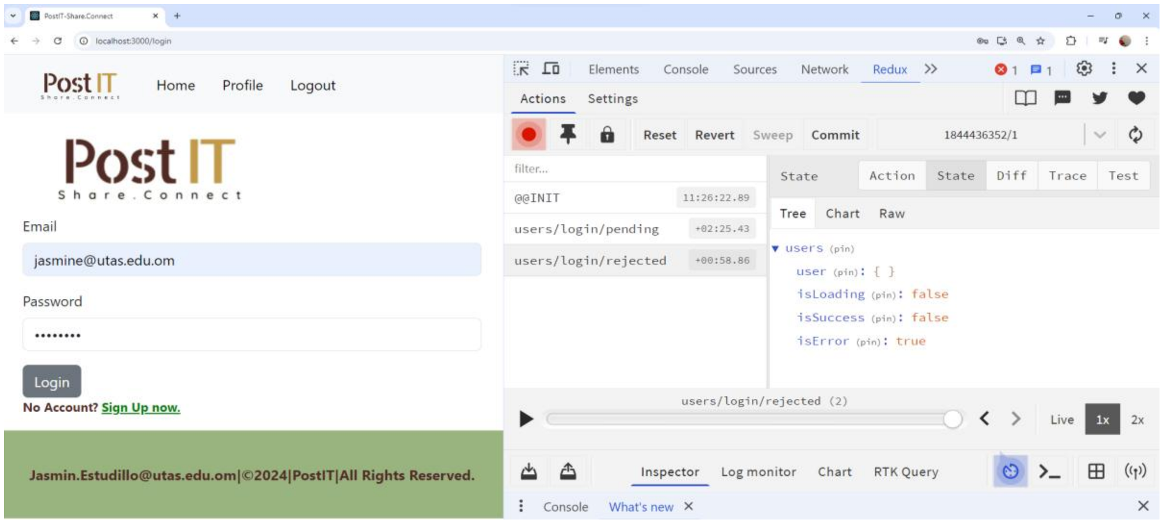
```
useEffect(() => {
    if (isError) {
      navigate("/login");
    }
    if (isSuccess) {
      navigate("/");
    } else {
      navigate("/login");
    }
  }, [user, isError, isSuccess]);
```

46. Testing the login. In your Login page open your Redux Dev tools. Notice the values of the state users.



```
@@INIT    11:26:22.89

▼ users (pin)
    user (pin): { }
    isLoading (pin): false
    isSuccess (pin): false
    isError (pin): false
```

Input incorrect credentials and observe the values of the state.



Email

jasmine@utas.edu.om

Password

••••••••

Login

No Account? Sign Up now.

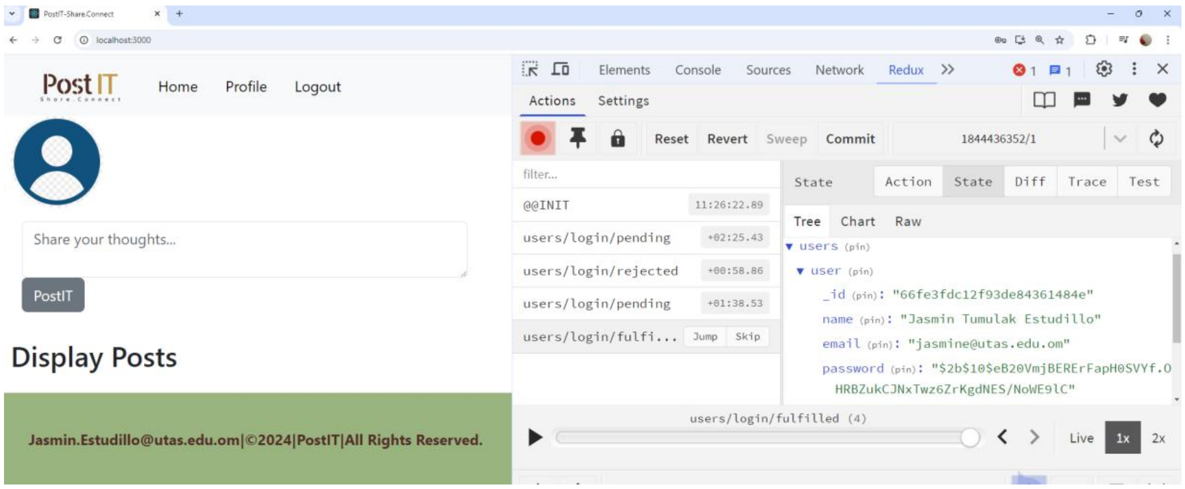
Jasmin.Estudillo@utas.edu.om|©2024|PostIT|All Rights Reserved.

```
@@INIT                 11:26:22.89
users/login/pending    +02:25.43
users/login/rejected   +00:58.86
```

```
▼ users (pin)
    user (pin): { }
    isLoading (pin): false
    isSuccess (pin): false
    isError (pin): true
```

users/login/rejected (2)

Input correct credentials for a registered user and the page should be redirected to the Home page. Also observed the value in the Redux store. Notice that the user property has been updated with the details of the user.



Share your thoughts...

PostIT

Display Posts

Jasmin.Estudillo@utas.edu.om|©2024|PostIT|All Rights Reserved.

```
@@INIT                 11:26:22.89
users/login/pending    +02:25.43
users/login/rejected   +00:58.86
users/login/pending    +01:38.53
users/login/fulfi...
```

```
▼ users (pin)
  ▼ user (pin)
      _id (pin): "66fe3fdc12f93de84361484e"
      name (pin): "Jasmin Tumulak Estudillo"
      email (pin): "jasmine@utas.edu.om"
      password (pin): "$2b$10$eB20VmjBERErFapH0SVYf.O
        HRBZukCJNxTwz6ZrKgdNES/NoWE9lC"
```

users/login/fulfilled (4)

PART 8 - LOGOUT Implementation

47. In **server/index.js**, we will make API for logout.

```
//POST API-logout
app.post("/logout", async (req, res) => {
res.status(200).json({ message: "Logged out successfully" });
});
```

48. In **client/src/Features/Users.js**, create a logout thunk.

```
export const logout = createAsyncThunk("/users/logout", async () =>
{
    try {
      // Send a request to your server to log the user out
      const response = await
axios.post("http://localhost:3001/logout");
    } catch (error) {}
  });
```

49. In the **UserSlice.js**, add the extradeducers for the logout.

```
   extraReducers: (builder) => {
    //Asynchronous actions that update the state directly,

   …..
         .addCase(logout.pending, (state) => {
           state.isLoading = true;
         })
         .addCase(logout.fulfilled, (state) => {
           // Clear user data or perform additional cleanup if
needed
           state.user = {};
           state.isLoading = false;
           state.isSuccess = false;
         })
         .addCase(logout.rejected, (state) => {
           state.isLoading = false;
           state.isError = true;
         });
   …..
```

50. **In src/Components/Header.js**, import useDispatch() and useNavigate from react-router-dom. Also import the logout function.

```
import { useDispatch } from "react-redux";
import { Link, useNavigate } from "react-router-dom";
import { logout } from "../Features/UserSlice";
```

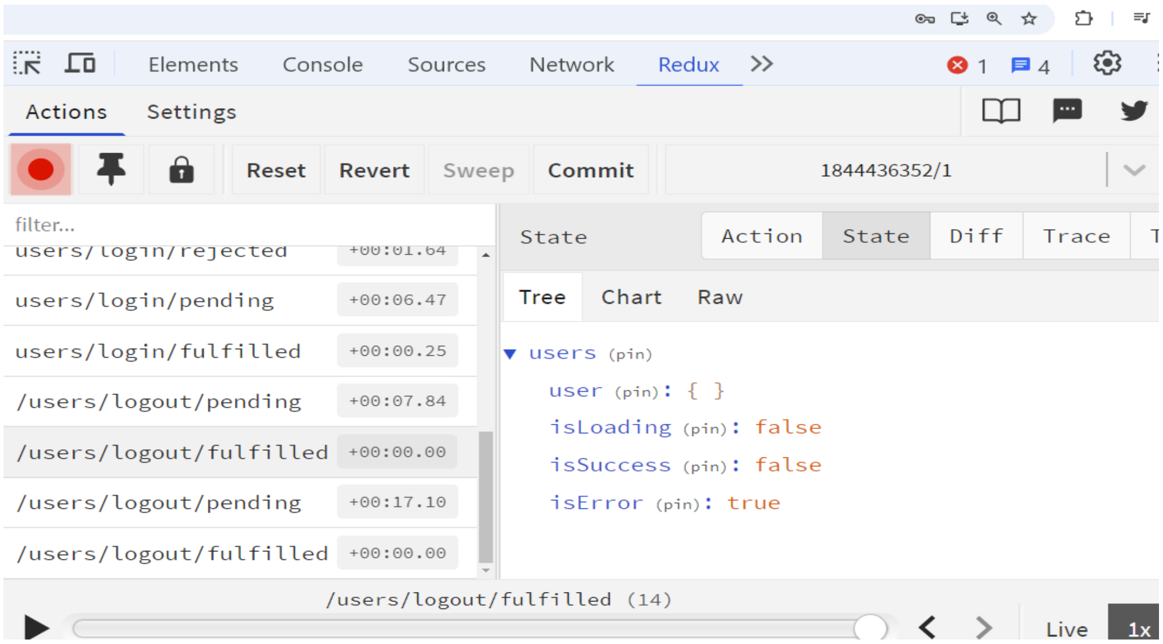
51. In src/Components/Header.js, make a new function **handlelogout().** This will be invoked when the user clicks the logout link. Declare also dispatch and navigate variables.

```
const dispatch = useDispatch();
const navigate = useNavigate();

const handlelogout = async () => {
  dispatch(logout());
  //ensure that the state update from the logout action has been
processed before proceeding to the next step.
  await new Promise((resolve) => setTimeout(resolve, 100));

  navigate("/"); //redirect to login page route.
};
```

52. In the logout link create an event handler that will call the **handlelogout**() function.

```
…
<Link onClick={handlelogout}>Logout</Link>
…
```

53. Test your logout functionality check your Redux store is the action is dispatch properly and the states are updated. Notice in the actions tab in the fulfilled action type, the user is set to empty object.



```
users/login/rejected        +00:01.64
users/login/pending         +00:06.47
users/login/fulfilled       +00:00.25
/users/logout/pending       +00:07.84
/users/logout/fulfilled     +00:00.00
/users/logout/pending       +00:17.10
/users/logout/fulfilled     +00:00.00
```

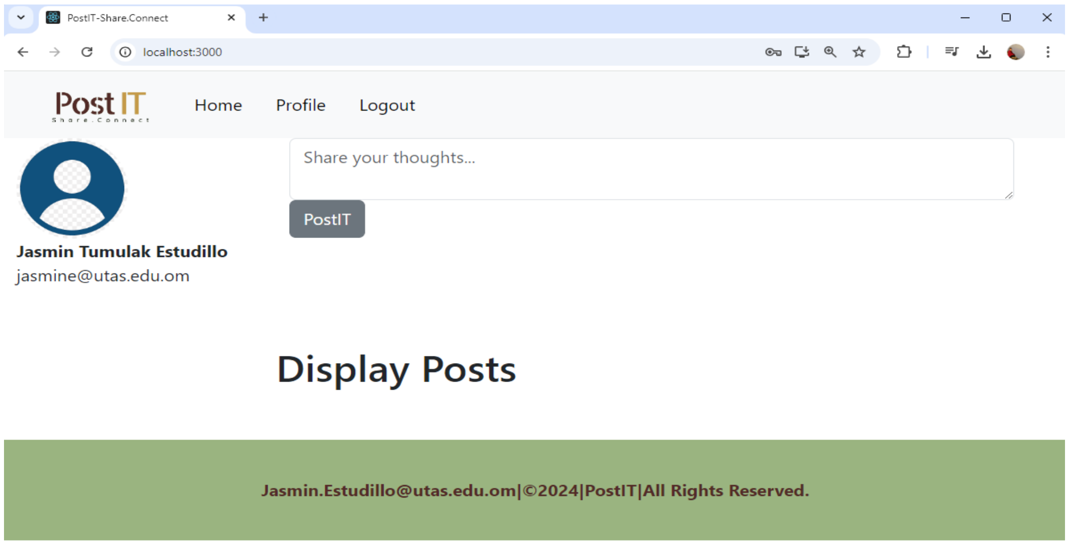
```
▼ users (pin)
    user (pin): { }
    isLoading (pin): false
    isSuccess (pin): false
    isError (pin): true
```

/users/logout/fulfilled (14)

PART 9-RETRIEVING THE USER DETAILS FROM THE REDUX STORE

54. In the component in **src/Components/User.js.** Import the useSelector hook and retrieve the current values of state and display.

```
import user from "../Images/user.png";
import { useSelector } from "react-redux";

const User = () => {
  const email = useSelector((state) => state.users.user.email);
  const name = useSelector((state) => state.users.user.name);

  return (
    <div>
      <img src={user} className="userImage" />
      <p>
        {name}
        <br />
      </p>
    </div>
  );
};

export default User;
```



Jasmin Tumulak Estudillo
jasmine@utas.edu.om

Display Posts

Jasmin.Estudillo@utas.edu.om|©2024|PostIT|All Rights Reserved.

55. Do the same in the Prolife Component.

PART 10 – Conditional Rendering of the Header and Footer Component

56. We will the code to conditionally render the Header component. This means that the component will only be rendered or displayed when the user is successfully logged in to the system and thus, the Redux state has been updated. In App.js, import the **useSelector** hook.

```
import { useSelector } from "react-redux";
```

57. Declare the variable to retrieve the email of the user from the Redux store.

```
const App = () => {
    const email = useSelector((state) => state.users.user.email);

    return (
     ….
  };
```

58. In the first row of the Container component, write the code check if email value has been set, then render the Header component, otherwise display null.

```
…..
 return (
    <Container fluid>
      <Router>
        <Row>
          {email ? (
            <>
              <Header />
            </>
          ) : null}
        </Row>
        …..
    </Container>
  );
};
```

Also, do the same logic for the Footer.

```
<Row>
{email ? (
  <>
    <Footer />
  </>
) : null}
</Row>
```

59. Test your application.