Class 11 integration , protected routes)

Agenda

1. Integrating frontend with backend
2. Adding support for login and signup
3. Implementing protected routes
4. Resolving cors error when sending the credentials

Run the frontend

1. Go inside the ecommerce_project and
2. npm i
3. Npm run dev
4. See that currently the data comes from fakestore api .com

Run the backend

1. few changes were made to the product db
    a. Added names and images
    b. Go to https://fakestoreapi.com/products and json formatter and get the names, images, etc
2. Proper products were added and images were updated in local db
3. Can use multer to add the capability to upload images
    a. Multer is a middleware for handling multipart/form-data, primarily used for uploading files in Node.js applications

b. Multer intercepts files uploaded via multipart/form-data forms and makes them accessible through the req.file

```
const multer = require('multer'); // code for reference if needed else skip

// Set up storage location and filenames
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/') // 'uploads/' is the folder where files will be saved
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname))
  }
});

// Initialize multer with the storage configuration
const upload = multer({ storage: storage });

app.post('/upload', upload.single('image'), (req, res) => {
  // Handle the uploaded file here
  // req.file contains the information about the uploaded file
  console.log(req.file);
  res.send('File uploaded successfully!');
});
```

Front end

```
<form action="/upload" method="post"
enctype="multipart/form-data">
  <input type="file" name="image" />
  <button type="submit">Upload</button>
</form>
```

4. In our project ,we will add environment variables
    a. Basically we will store the base url for backend in the env which can change when we move from dev to qa to prod
    b. Create .env file at the root front end folder
    c. https://vitejs.dev/guide/env-and-mode
    d. when using Vite as your build tool for a React app, it is mandatory to prefix environment variables with VITE_ if you want them to be exposed to your client-side code
    e. Create a variable as below

```
VITE_BASE_URL=http://localhost:3000
```

5. We will create a file urlConfig.js at root folder to store all our api endpoints
    a. Having them hardcoded everywhere can cause errors
    b. import.meta.env.VITE_BASE_URL

Integration first steps

1. Let us integrate signup, login and getAllProducts first
2. urlConfig file

```
const BASE_URL = import.meta.env.VITE_BASE_URL
const urlConfig = {
  LOGIN_URL: `${BASE_URL}/api/auth/login`,
  SIGNUP_URL: `${BASE_URL}/api/auth/signup`,
  LOGOUT_URL: `${BASE_URL}/api/auth/logout`,
  GET_PRODUCTS: `${BASE_URL}/api/products`,
  GET_CATEGORIES: `${BASE_URL}/api/products/categories`,
}

export default urlConfig;
```

FRONEND CODE WALKTHRU

1. Main.jsx is the entry point
2. App.jsx has basically four routes and components
3. In Home.jsx we have two useEffects which are fetching the categories and products
    a. We would change these to use our backend api
    b. We will use axios to make api calls - less code with axios
    c. Install axios in front end code - npm i axios
    d. Add the two imports in HOme.jsx

```
import axios from 'axios';
import URL from '../../urlConfig'
```

e. In the first useEffect to get the product data, update the code as below

```
useEffect(() => {
        (async function () {
            // const resp = await
fetch(`https://fakestoreapi.com/products`)
            // console.log(resp);
            // const anotherResp = await
fetch("/api/product");

            // const productData = await resp.json();
            // setProducts(productData);
            const productData = await
axios.get(URL.GET_PRODUCTS);
            console.log(productData);

        })()
    }, [])
```

f. If there is cors issue, in app.js in backend code

```
const cors = require("cors");

app.use(cors())
```

g. We dont have categories coming from our code. So you can choose to comment or simply create a controller to return categories in productController

```
const getProductCategories = async function (req, res)
{
 res.json({
```

```
   message: "Get categories successful",
   data: ["electronics", "men's clothing", "women's
clothing", "jewelery"],
 });
}
```

h. Update route in productRouter

```
productRouter.get('/categories',getProductCategories)
```

i. urlConfig

```
   GET_CATEGORIES:
`${BASE_URL}/api/products/categories`,
```

j. In home.jsx

```
useEffect(() => {
       (async function () {
           const categoriesData = await
axios.get(URL.GET_CATEGORIES);
           // const categoriesData = await
resp.json();
           setCategories(categoriesData.data.data);

console.log("categories",categoriesData.data.data)
       })()
   }, [])
```

Product Details

1. Look at the product details file in components which is integrated in app.jsx

2. We need title, price and image from response
3. In such cases we create a map to map the keys

```
useEffect(() => {
        (async function () {
            const productData = await
axios.get(URL.GET_PRODUCTS);
            console.log("products",productData.data.data)
            const productArr = productData.data.data;
            const productList = productArr.map((product) => {
                return{
                    id:product._id,
                    title:product.name,
                    image:product.images[0],
                    price:product.price,
                    ...product
                }
            })
            setProducts(productList);
        })()
    }, [])
```

4. To update product images and other details, can use this fake store api - https://fakestoreapi.com/products

Login and Signup

1. Create new folder Login and Signup under pages
2. Under Login create index.jsx and login.css
3. The benefit with index.jsx is that while importing we give the path only till the folder and can skip writing index.jsx ike import Login from ‘./Login’

Implementing Signup

1. Import Signup in app.jsx

```
import Signup from './pages/Signup'

<Route path="/home" element={<Navigate to="/"></Navigate>}></Route>
      <Route path="/signup" element={<Signup></Signup>}></Route>
```

2. Signup we will need all the input fields that we had created in the user model
3. In the index.jsx under signup , do rfce to get the functional component template

```
import React from 'react'

function Signup() {
 return (
   <div>Signup</div>
 )
}

export default Signup
```

4. Go to http://localhost:5173/signup
5. There are few errors and loading state , need to add state variables and handleSubmit method

```
function Signup() {
   const [name, setName] = useState('')
   const [email, setEmail] = useState('')
   const [password, setPassword] = useState('')
```

```
   const [confirmPassword, setConfirmPassword] =
useState('');
   const [errMsg, setErrMsg] = useState('')
   const [loading, setLoading] = useState(false)
```

6. Few imports are needed

```
import React,{useState} from "react";
import { Link } from "react-router-dom";
import './signup.css'
import axios from "axios";
import urlConfig from "../../../urlConfig";
```

7. Add the loading condition

```
if(loading){
   return <h1>Loading...</h1>
}
 return (
   <div className="signupscreen">
```

8. Let us update the handleSubmit handler which will create user in our DB

```
   const handleSubmit = () => {
       try{

       }catch(err){
           console.log(err)
           setLoading(false)
           setErrMsg(err.message)
```

```
            setTimeout(()=>{
                setErrMsg('')
            },2000)
        }
    }
```

9. Now the success part

```
const handleSubmit = async () => {
        try{
            setLoading(true)
            const userDetails = {name, email, password,
confirmPassword,phone}
            const res = await
axios.post(urlConfig.SIGNUP_URL, userDetails)
            console.log(res)
            setLoading(false)
            setName('')
            setEmail('')
            setPassword('')
            setConfirmPassword('')
            setPhoneNumber('')

        }catch(err){
            console.log(err)
            setLoading(false)
            setErrMsg(err.message)
            setTimeout(()=>{
                setErrMsg('')
            },2000)
        }
    }
```

10. Check in mogodb. First handshake is complete

Redirection to Login after signup

1. After the user is created, we will navigate user to Login route
2. What is the hook in react-router-dom? - useNavigate
3. In index.js under Login , add below

```
import React from 'react'


function Login() {
 return (
   <div>Login</div>
 )
}


export default Login
```

4. In app.jsx, add the route for Login

```
<Route path="/signup" element={<Signup></Signup>}></Route>
       <Route path="/login"
element={<Login></Login>}></Route>
```

5. In signup.jsx, import useNavifate

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


const [loading, setLoading] = useState(false)
   const navigate = useNavigate()
```

6. In handleSubmit, navigate to login

```
setConfirmPassword('')
          setPhoneNumber('')
          navigate('/login')
```

Login

1. User authentication using email and password
2. Passing jwt with requests for protected routes
3. Get the imports, then state variables

```
import React, { useState } from "react";
import { Link, useNavigate } from "react-router-dom";
import "./login.css";
import axios from "axios";
import urlConfig from "../../../urlConfig";

function Login() {
 const [email, setEmail] = useState("");
 const [password, setPassword] = useState("");
 const [errMsg, setErrMsg] = useState("");
 const [loading, setLoading] = useState(false);
 const navigate = useNavigate(); // take user to home page

 if (loading) {
   return <h1>Loading...</h1>;
 }
 return (
   <div className="signinscreen">
     <div className="container">
       <div className="innerContainer">
```

```
          <div
            style={{
              display: "flex",
              justifyContent: "space-between",
              alignItems: "center",
              marginBottom: "20px",
              // backgroundColor: 'red',
            }}
          >
            <div style={{ cursor: "pointer" }} onClick={()
=> {}}>
              <i class="fas fa-arrow-circle-left fa-5x"></i>
            </div>
            <p>Sign In</p>
          </div>

          <label for="email">Email</label>
          <input
            type="email"
            id="lname"
            name="email"
            placeholder="Your email.."
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          />
          <label for="password">Password</label>
          <input
            type="password"
            id="lname"
            name="password"
            placeholder="Your Password.."
            value={password}
```

```
            onChange={(e) => setPassword(e.target.value)}
          />
          <Link to="/signup" className="link">
            <span>Create a new account ?</span>
          </Link>
          <br />
          <input type="submit" value="Sign in"
onClick={handleSubmit} />
          <div className={errMsg ? "errContainer" :
""}>{errMsg}</div>
        </div>
      </div>
    </div>
  );
}

export default Login;
```

4. Copy the handle submit from signup and update the url and remove extra fields

```
const handleSubmit = async () => {
    try{
        setLoading(true)
        const userDetails = {email, password}
        const res = await axios.post(urlConfig.LOGIN_URL
, userDetails)
        console.log(res)
        setLoading(false)
        setEmail('')
        setPassword('')
```

```
        navigate('/')

    }catch(err){
        console.log(err)
        setLoading(false)
        setErrMsg(err.message)
        setTimeout(()=>{
            setErrMsg('')
        },2000)
    }
}
```

5. Test from frontend by visiting /login route

Including token received from response

1. When we login, we should see token in the response header
2. What we want is to include this token in our further requests
3. In Login.jsx

```
        const res = await axios.post(urlConfig.LOGIN_URL,
userDetails, {withCredentials:true})
```

4. Cross-Origin Cookie Handling: Normally, for security reasons, browsers do not send cookies or auth headers in cross-origin requests. Setting withCredentials: true tells Axios (and the underlying XMLHttpRequest in the browser) to include these credentials in the request to the server.
5. In app.js pn backend update cors

```
app.use(cors({ origin: "http://localhost:5173",
credentials: true }));
```

6. On the server side, the CORS configuration must explicitly allow credentials from the origin making the request.
7. This option allows the server to accept requests that include credentials like cookies, authorization headers. With credentials: true, CORS requests will also include cookies and HTTP authentication information.

Implementing Protected Routes

1. We want to make our cart, user details accessible only when user is logged in
2. Protected routes are specific areas or components in a web application that require the user to be authenticated to access them
3. We will be implementing conditional rendering using context api to control visibility of components / pages
4. Once a user logs in -> add it to the context -> use the context data to check and render conditionally
5. Create AuthProvider.jsx under contexts folder

```
import {useContext, useState} from 'react'


const AuthContext = createContext()


const AuthProvider = ({children}) => {
```

```
    const [authenticatedUser, setAuthenticatedUser] =
useState({})
    return (
        <AuthContext.Provider value={{authenticatedUser,
setAuthenticatedUser}}>
            {children}
        </AuthContext.Provider>
    )
}

export const useAuth = () => {
    return useContext(AuthContext)
}
export default AuthProvider
```

6. In App.jsx add the auth provider

```
<AuthProvider>
     <PaginationProvider>
```

7. Let us now use the setAuthenticatedUser when the login is successful
8. Check the console to see where we are getting the user from backend after login
9. Update import in Login

```
import "./login.css";
import { useAuth } from "../../contexts/AuthProvider";
```

10. In the handleSubmit handler

```
try{
        setLoading(true)
        const userDetails = {email, password}
        const res = await axios.post(urlConfig.LOGIN_URL,
userDetails, {withCredentials:true})
        console.log("user",res)
        setAuthenticatedUser(res.data.user)
        setLoading(false)
        setEmail('')
        setPassword('')
        navigate('/')

    }
```

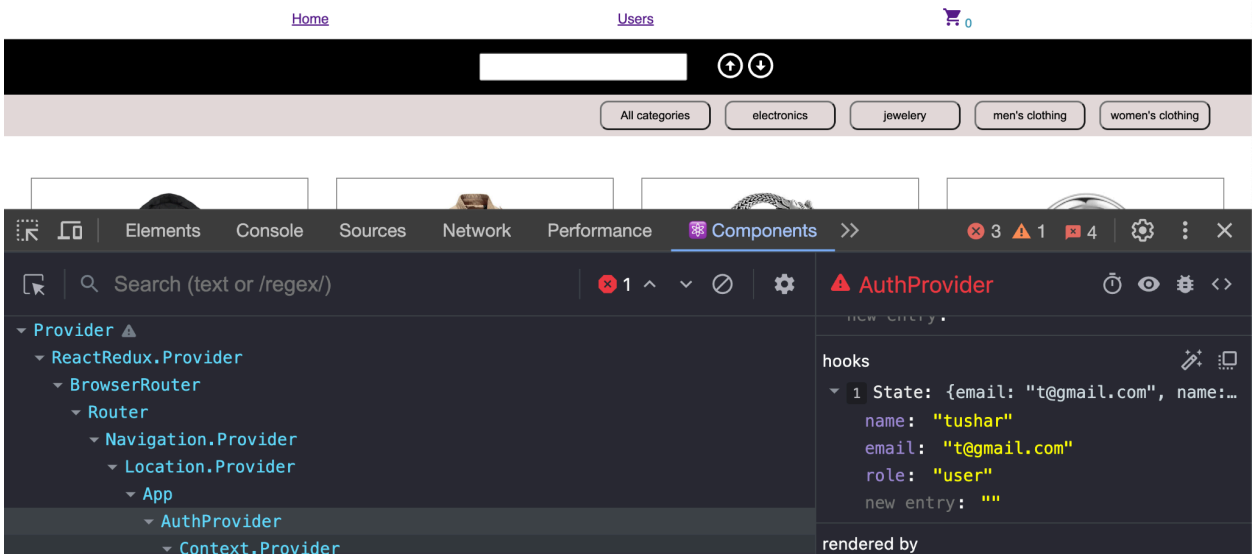
11. In the authController for login handler , we can limit the data pf the user that gets sent to the frontend

```
const loginHandler = async function (req, res) {
 try {
   let { email, password } = req.body;
   let user = await UserModel.findOne({ email });
   if (user) {
     let areEqual = password == user.password;
     if (areEqual) {
       // user is authenticated
       /* 2. Sending the token -> people remember them
        * */
       // payload : id of that user
       jwt.sign(
         { id: user["_id"] },
         SECRET_KEY,
         { expiresIn: "1h" },
```

```
          function (err, data) {
            if (err) {
              throw new Error(err.message);
            }
            res.cookie("token", data, {
              maxAge: 1000 * 60 * 60 * 24 * 7,
              httpOnly: true,
            });
            res
              .status(200)
              .json({ status: "success", message: data,
user:{
                name: user.name,
                email: user.email,
                role: user.role
              } });
          }
        );
      } else {
        // console.log("err", err);
        res.status(404).json({
          status: "failure",
          message: "email or password is incorrect",
        });
      }
    } else {
      res.status(404).json({
        status: "failure",
        message: "user not found",
      });
    }
  } catch (err) {
```

```
    console.error(err);
    res.status(500).json({
      status: "failure",
      message: err.message,
    });
  }
};
```

12. Login again and in components tab see the logged in user details



13. Now that we have the user data in our context , let us protect our routes
14. Craeting a reusable component to redirect to login when not authenticated
15. Under components folder create RequireAuth.js

```
import { useLocation, Navigate, Outlet } from
"react-router-dom";
import {useAuth} from "../contexts/AuthProvider";
```

```
const RequireAuth = () => {
 const { authenticatedUser  } = useAuth();
 const location = useLocation();
 return authenticatedUser ? (
   <Outlet />
 ) : (
   <Navigate to="/login" state={{ from: location }} replace
/>
 );
};

export default RequireAuth;
```

a. Let us understand this code
b. RequireAuth, is designed to work within our application using React Router for navigation. It acts as a guard for protected routes, ensuring that only authenticated users can access certain parts of the application.
c. useLocation: This hook from React Router is used to access the location object, which represents where the app is now,
d. Navigate: This component from React Router is used to navigate programmatically to a different URL.
e. Outlet: This component from React Router is used to render the appropriate child routes. It acts as a placeholder that is replaced by the actual content of the child routes.

f. state={{ from: location }}: This passes along the current location to the route you're navigating to (in this case, "/signin"). This is typically used to remember the URL the user tried to access before being redirected to sign in, so you can send them back to that URL after they authenticate.

g. replace: This prop on the <Navigate /> component tells the router to replace the current entry in the history stack with the new location. This means that when users sign in and hit the back button, they won't be sent back to the sign-in page but to wherever they were before they got redirected to the sign-in page

16. Adding the RequireAuth in app.jsx

```
<Routes>
          <Route path="/" element={<Home></Home>}>
            {" "}
          </Route>
          <Route element={<RequireAuth></RequireAuth>}>
            <Route path="/cart" element={<Cart></Cart>}></Route>
          </Route>
          <Route
            path="/product/:id"
            element={<ProductDetails></ProductDetails>}
          >
```

a. In React Router v6, routes can be nested, and the Route component can be used as a layout or a wrapper component to apply certain conditions or layouts to all its

child routes. This is a common pattern for implementing protected routes that require authentication.

b. The Route component that uses the RequireAuth element doesn't have a path prop because it serves as a wrapper or a layout route. It's designed to render an authentication check for any of its child routes.
c. When a user navigates to "/cart", React Router first checks the RequireAuth component because it wraps the "/cart" route.