

MERN-6: Project Part 2 - (Creating User's , Partners & Admins Route)

Add the role to the userModel.js

```
role: {  
  type: String,  
  enum: ['admin', 'user', 'partner'],  
  required: true,  
  default: 'user'  
}
```

title:Setup proxy to connect client and server

Now as we have our Pages and Form created and we have also created our Server side Routes for Login and Register Now we will connect our client and server with using proxy

In client folder in Your Package.json file at the bottom add this line mentioning the port in which you are running the server

```
},  
"proxy": "http://localhost:8082"
```

Proxy in package.json: Explanation

The proxy field in the package.json file of a Create React App project is used to redirect API requests during development to avoid CORS issues.

What is CORS?

CORS stands for Cross-Origin Resource Sharing. It's a security feature implemented by web browsers to control how resources (like HTML, JavaScript, or data) can be requested from another domain (origin) than the one that served the web page.

Why Do We Need CORS?

When you make an API request from your frontend code (like a React app running on `http://localhost:3000`) to a backend server (like `http://localhost:8081`), the browser sees this as a "cross-origin" request because the ports are different.

To protect users, browsers block these requests by default unless the server explicitly allows them. This prevents malicious websites from accessing sensitive data from other sites.

How Does CORS Work?

Preflight Request:

For certain types of requests, the browser sends an HTTP OPTIONS request to the server before the actual request. This is called a preflight request. It checks if the server permits the actual request.

Response with Headers:

If the server allows the request, it responds with specific headers:

Access-Control-Allow-Origin: Specifies which origins can access the resource.

Access-Control-Allow-Methods: Specifies which HTTP methods (GET, POST, etc.) are allowed.

Access-Control-Allow-Headers: Specifies which headers can be used in the actual request.

Example Scenario

Imagine you have:

Frontend (React app): Running on `http://localhost:3000`

Backend (API server): Running on `http://localhost:8081`

If you try to make a fetch request from the React app to the API server, the browser will block it unless the server sends back the appropriate CORS headers.

How to Avoid CORS Issues During Development

In development, dealing with CORS can be a hassle, so we use a proxy to make it easier.

Using Proxy in package.json

By adding a proxy setting in your package.json, you can instruct the development server to forward API requests to your backend server. This makes the browser think the requests are coming from the same origin.

Here's how you do it:

Add Proxy Setting:

```
"proxy": "http://localhost:8081"
```

How It Works:

When your React app running on `http://localhost:3000` makes a request to `/api/data`, the development server intercepts this request.

It then forwards the request to `http://localhost:8081/api/data`.

The server responds, and the development server sends this response back to your React app.

By doing this, you avoid CORS issues without needing to configure CORS headers on your backend server during development.

title:Setting up Axios and Axios Instance

Now in your Client Side Install axios

npm install axios

You are already familiar with axios as we have used this in our movies app with React , but there we only saw how to get data from an API , but axios is alot more it can handle post , put , delete methods as well from the client.


First we will set up axios and then will see how to use it for different requests

Inside your src directory create a directory by the name **api** , in this directory we will have all our axios calls

index.js

```
import axios from "axios";

export const axiosInstance = axios.create({
  headers: {
    "Content-Type": "application/json",
  },
});
```



This code snippet is setting up a custom instance of Axios
`axios.create({ ... })`: This method is used to create a new Axios
instance with a custom configuration.

The configuration object passed to `axios.create` includes a `headers`
property, which sets default headers for all requests made using this
instance.

`headers: { 'Content-Type': 'application/json' }`: This sets the
Content-Type header to 'application/json' for all requests made using
this Axios instance. The Content-Type header indicates that the data
being sent in the HTTP request is in JSON format.

Why Use an Axios Instance?

Creating an Axios instance with custom configuration can be useful
for several reasons:

Default Configuration: You can set default headers, base URLs,
timeouts, and other settings that will apply to all requests made with
this instance, ensuring consistency across your API calls.

Custom Interceptors: You can add request or response interceptors to handle things like authentication tokens or logging for all requests made with this instance.

Reusability: Having a pre-configured instance makes it easy to reuse the same settings across different parts of your application.

Now create a file with the name inside the same **api** directory with the name **users.js** this file will have all the user related calls for register login etc.

Inside the file users.js write your first axios instance

```
const {axiosInstance} = require('.')

//Register new User

export const RegisterUser = async (value) => {
  try{
    const response = await axiosInstance.post("api/users/register", value);
    return response.data;
  }catch(err){
    console.log(err);
  }
}
```

Now we will use this exported function RegisterUser in our Register.js page

```
import React from "react";
import { Button, Form, Input, message } from "antd";
import { Link } from "react-router-dom";
```

```

import {RegisterUser} from "../../api/user";

function Register() {
  const onFinish = async (values)=>{

    try {
      const response = await RegisterUser(values)
      if(response.success){
        message.success(response.message)
      }
      else{
        message.error(response.message)
      }
    } catch (error) {
      message.error(error.message)
    }
  }
  return (
    <>
    <main className="App-header">
      <h1>Register to BookMyShow</h1>
      <section className="main-area mw-500 text-center px-3">
        <Form layout="vertical" onFinish={onFinish}>
          <Form.Item
            label="Name"
            htmlFor="name"
            name="name"
            className="d-block"
            rules={[{ required: true, message: "Name is required" }]}
          >
            <Input
              id="name"
              type="text"
              placeholder="Enter your name"
            ></Input>
          </Form.Item>

          <Form.Item
            label="Email"
            htmlFor="email"

```



```

        name="email"
        className="d-block"
        rules={[
          { required: true, message: "Email is required" },
          { type: "email", message: "Please enter a valid email" },
        ]}
      >
      <Input
        id="email"
        type="text"
        placeholder="Enter your Email"
      ></Input>
    </Form.Item>

    <Form.Item
      label="Password"
      htmlFor="password"
      name="password"
      className="d-block"
      rules={[{ required: true, message: "Password is required" }]}
    >
      <Input
        id="password"
        type="password"
        placeholder="Enter your Password"
      ></Input>
    </Form.Item>

    <Form.Item className="d-block">
      <Button
        type="primary"
        block
        htmlType="submit"
        style={{ fontSize: "1rem", fontWeight: "600" }}
      >
        Register
      </Button>
    </Form.Item>
  </Form>
</div>
<p>
  Already a user? <Link to="/login">Login now</Link>

```

```

        </p>
      </div>
    </section>
  </main>
</>
);
}

export default Register;

```

The onFinish prop specifies the function to be called on successful submission. The onFinish function is an asynchronous function that is called when the form is successfully submitted. The values parameter contains the data entered by the user in the form fields.

If the registration is successful (response.success is true), a success message is displayed using message.success which is imported from antd.

If the registration fails (response.success is false), an error message is displayed using message.error.

title:Setting up Axios for Login

Similarily now add an Axios Instance for the Login in the same way we have added it for register , just the route will change from resgiter to login , add this in the same file

src-> api-> users.js

```
export const LoginUser = async (value) =>{
  try {
    const response = await axiosInstance.post("api/users/login", value);
    return response.data
  } catch (error) {
    console.log(error);
  }
}
```

Now Similarly add the OnFinish Function in Login and if the user will be able to successfully login we will send the user to the home page by using the useNaviagte from react-router-dom

```
import React from "react";
import { Button, Form, Input } from "antd";
import { Link, useNavigate } from "react-router-dom";
import { LoginUser } from '../api/user';
import {message} from 'antd'

function Login() {
  const navigate = useNavigate()
  const onFinish = async (values)=>{
    console.log(values)
    try {
      const response = await LoginUser(values)
      if(response.success){
        message.success(response.message)
        navigate('/')
      }
      else{
        message.error(response.message)
      }
    } catch (error) {
      message.error(error.message)
    }
  }
}
```

```

    }
  }
  return (
    <>
      <main className="App-header">
        <h1>Login to BookMyShow</h1>
        <section className="mw-500 text-center px-3">
          <Form layout="vertical" onFinish={onFinish}>
            <Form.Item
              label="Email"
              htmlFor="email"
              name="email"
              className="d-block"
              rules={[
                { required: true, message: "Email is required" },
                { type: "email", message: "Please enter a valid email" },
              ]}
            >
              <Input
                id="email"
                type="text"
                placeholder="Enter your Email"
              ></Input>
            </Form.Item>

            <Form.Item
              label="Password"
              htmlFor="password"
              name="password"
              className="d-block"
              rules={[{ required: true, message: "Password is required" }]}
            >
              <Input
                id="password"
                type="password"
                placeholder="Enter your Password"
              ></Input>
            </Form.Item>

            <Form.Item className="d-block">
              <Button
                type="primary"

```

```

        block
        htmlType="submit"
        style={{ fontSize: "1rem", fontWeight: "600" }}
      >
        Login
      </Button>
    </Form.Item>
  </Form>
</div>
<p>
  New User? <Link to="/register">Register Here</Link>
</p>
</div>
</section>
</main>
</>
);
}

export default Login;

```

title:Creating a Protected Route and JWT

Now that you've implemented the Login and Register routes, you can navigate to the Home screen. Have you noticed that once you log in to an app, even if you close the window and reopen it, you're still logged in and don't have to validate yourself repeatedly?

This is possible because of a mechanism called session persistence. When a user logs in, the server generates a token, which is then saved on the client side, either in local storage or as cookies. This token, known as a JSON Web Token (JWT), carries the user's authentication information.

When the user returns to the app, the client sends the stored token to the server with each request. The server validates the token to ensure it's still valid and has not been tampered with. If the token is valid, the server allows the user to access protected routes without requiring them to log in again.

The benefit of using JWTs is that the server doesn't need to store session information. Instead, the token itself contains all the information needed to verify the user's identity, making the system more scalable and reducing server-side complexity.

Analogy

- a. Imagine you are at a music festival. When you first enter, you show your ticket and receive a wristband. This wristband allows you to move in and out of different areas of the festival without having to show your ticket again. It's a proof that you've been authenticated at the entrance.
- b. In the digital world, a token serves a similar purpose as the festival wristband.
- c. When you log into a website or an application, instead of continuously entering your username and password for every action or transaction, the system gives you a token.

- d. This token is like a digital wristband that proves you are who you say you are and that you've already gone through the login process.
- e. A token in the context of web authentication is a string of characters that serves as a credential. Once a user is authenticated, the server generates this token and sends it to the user's device.
- f. This is often done using JSON Web Tokens (JWTs), which are encoded with a secret key.
- g. Tokens can be encrypted and signed, ensuring that the user data they carry is secure.
- h. The server doesn't need to remember the user's state between requests. The token itself contains all the necessary information, making the system more scalable and efficient.
- i. Tokens should have an expiration time to prevent long-term use in case they are compromised.

JWT

JSON Web Token (JWT) is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. JWTs are commonly used for authentication and authorization in web applications.

Components of JWT

A JWT is composed of three parts:

Header

Payload

Signature

These parts are separated by dots (.) and are encoded in Base64 URL format.

1. Header

The header typically consists of two parts: the type of token (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

2. Payload

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims:

Registered claims: Predefined claims which are not mandatory but recommended, e.g., iss (issuer), exp (expiration time), sub (subject), aud (audience).

Public claims: Custom claims created to share information, e.g., name, role. Custom claims intended to be shared openly and used across different systems, registered to avoid naming collisions.

In a microservices architecture, various microservices might need to authenticate and authorize requests based on JWTs. Public claims allow these services to understand the user information consistently.

In an SSO environment, multiple applications or services need to authenticate the same user based on a single set of credentials. Public claims ensure that the user information is consistently understood across all these systems.

Private claims: Custom claims agreed upon between parties, e.g., user_id.

Example payload:

```
{  
  "sub": "1234567890",    // Registered claim: Subject  
  "name": "John Doe",    // Public claim: User's name  
  "email": "john.doe@example.com", // Public claim: User's email
```

```
"role": "admin",          // Public claim: User's role
"user_id": "abc123",      // Private claim: Custom user ID
"department": "sales",    // Private claim: User's department
"permissions": ["read", "write"] // Private claim: User's permissions
}
```

3. Signature

To create the signature, the encoded header, encoded payload, a secret, and the algorithm specified in the header are used. For example, if using HMAC SHA256, the signature is created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

JWT Structure

A JWT looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiaWF0IjE0NjQ5OTIwMD0.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiaWF0IjE0NjQ5OTIwMD0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

How JWT Works

Client Authentication: When a user logs in, the server generates a JWT and sends it back to the client.

Client Storage: The client stores the JWT (usually in localStorage or a cookie).

Subsequent Requests: The client includes the JWT in the header of each subsequent request to the server, usually in the Authorization header as a Bearer token.

Authorization: Bearer <token>

Server Validation: The server validates the JWT using the secret key. If the JWT is valid, the server processes the request; otherwise, it rejects the request.

Benefits of Using JWT

Stateless: The server does not need to store session information, making it easier to scale.

Compact: JWTs are small in size, making them suitable for being sent via URLs, POST parameters, or inside HTTP headers.

Self-contained: JWTs contain all the necessary information about the user, eliminating the need for multiple database queries.

Use Cases

Authentication: Ensures that the client is logged in before accessing protected routes.

Information Exchange: Securely transfers information between parties, ensuring data integrity and authenticity.

Generate Tokens

Let us try to generate a unique token for in our userRoute.js file for login.

We will first require the jwt package.

In the controller, add the below import ,

```
const User = require("../models/userModel")  
const jwt = require('jsonwebtoken')
```

We will define a token variable and use the sign method that creates a token. We have to pass it an ID for it to generate a token.

In our .env file, we will define jwt-secret is a text that you create by yourself. It should not be exposed which might cause harm to the system.

In the .env file, add

```
JWT_SECRET="ScalerMovies@123"
```

In the login method,

```
const login = async (req, res) => {  
  try {  
    const user = await User.findOne({ email: req.body.email });  
    console.log("req received", req.body, user);  
    const token = jwt.sign({ userId: user._id }, process.env.jwt_secret, {  
      expiresIn: "1d",  
    });  
    console.log(token);  
  }  
}
```

The expiresIn function is used to mention for how much time we want the token to remain alive.

We will also send data as token using the following-

```
res.send({  
  success: true,  
  message: "User Logged in",  
  data: token,  
});
```

In the Postman app, let us try to login and check whether we receive tokens or not.

Go to jwt.io and decode

Save the token in localStorage

Go to index.js under Login in client folder

```
function Login() {  
  const navigate = useNavigate()  
  const onFinish = async (values)=>{  
    console.log(values)  
    try {  
      const response = await LoginUser(values)  
      if(response.success){  
        message.success(response.message)  
        localStorage.setItem('token' , response.data)  
        navigate('/')  
      }  
      else{  
        message.error(response.message)  
      }  
    } catch (error) {  
      message.error(error.message)  
    }  
  }  
}
```

Validate in the application tab in the browser

