Practical 10 : Laravel REST API Authentication with JWT (II)

In last class, we learnt how can create secure REST APIs in Laravel using JSON Web Token (JWT) and we used the generated JWT token to verify the user identity once the user access to the protected information such as user-profile. Postman tool is used to perform the test of all REST APIs. In this class, we will learn how can design the necessary modals for allowing user to Login using the react component instead the postman.

1. Login Modal

First, we will start designing a modal that allows the users to perform a login action by inputting their email and password as shown in the figure below.

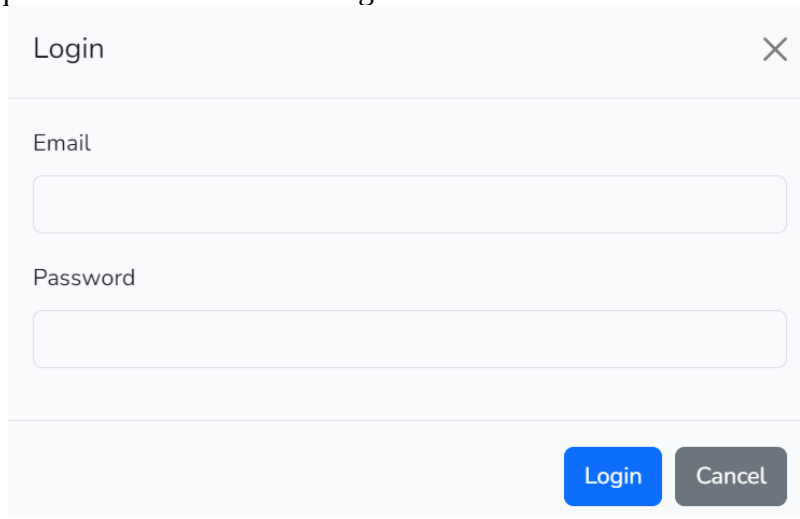

Figure 1: Login Modal.

The modal above serves as a user interface for logging into the application. It captures user credentials (email and password), provides a way to submit these credentials for authentication, and offers an option to cancel the action if the user decides not to proceed with the login. Below is the JavaScript code snippet to design the above modal.

```
<Modal isOpen={this.state.loginModal}
toggle={this.toggleloginModal.bind(this)}>
    <ModalHeader toggle={this.toggleloginModal.bind(this)}> Login
    </ModalHeader>
    <ModalBody>
        <FormGroup>
            <Label for="email">Email</Label>
            <Input
                id="email"
                value={this.state.loginData.email}
                onChange={(e) => {
                    let { loginData } = this.state
                    loginData.email = e.target.value
                    this.setState({ loginData })
                }}
```

```
            ></Input>
        </FormGroup>
        <FormGroup>
            <Label for="password">Password</Label>
            <Input
                id="password"
                type="password"
                value={this.state.loginData.password}
                onChange={(e) => {
                    let { loginData } = this.state
                    loginData.password = e.target.value
                    this.setState({ loginData })
                }}
            ></Input>
        </FormGroup>

    </ModalBody>
    <ModalFooter>
        <Button color="primary" onClick={this.login.bind(this)}> Login
</Button>{'     '}
        <Button color="secondary"
onClick={this.toggleloginModal.bind(this)}> Cancel </Button>
    </ModalFooter>
</Modal>
```

When the login button is clicked, the login() method of the component is invoked, handling the login functionality.

2. Login Button to open the modal

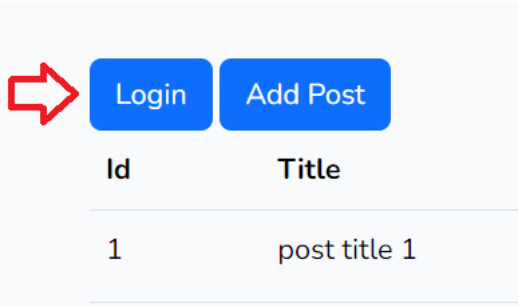

Figure 2: Login Button to open the modal.

When user clicks this button, it triggers the toggleloginModal method. This method is designed to toggle the state of loginModal, effectively showing the modal if it's currently hidden or hiding it if it's currently shown. However, its primary use here is to open the modal for the user to login. Below is the JavaScript code snippet to design the Button.

```
<Button color="primary"
onClick={this.toggleloginModal.bind(this)}>Login</Button>
```

3. toggleloginModal method

The toggleloginModal method is designed to change the state of loginModal between true and false. This state likely controls whether the login modal is currently shown or hidden to the user. Below is the JavaScript code snippet to design the The toggleloginModal method.

```
toggleloginModal() {
        this.setState({ loginModal: !this.state.loginModal })
    }
```

In the above code, this.setState(...) is a React method used to update the component's state. Updating the state in this manner also triggers a re-render of the component if the state change affects the output.
For the code { loginModal: !this.state.loginModal } is where the actual toggle happens. It sets loginModal to the opposite of whatever its current value is. If loginModal is true (indicating the modal is open), it sets it to false (closing the modal), and vice versa. However, we need to define loginModal variable in the State object as follows:

```
constructor() {
        super()
        this.state = {
            posts: [],
            newPostModal: false,
            updatePostModal: false,
            newPostData: { title: "", content: "", user_id: "" },
            updatePostData: { id:"", title: "", content:"", user_id:"" },
            loginModal: false,
            loginData: { email: "", password: ""},
            loginResponseData:[]
        }
    }
```

4. Login Method

The login method facilitates user authentication by sending a POST request to a specified endpoint (http://127.0.0.1:8000/api/auth/login) with the login credentials (e.g. email and password) stored in the component's state (this.state.loginData). Upon receiving a response from the server, the method updates the component's state to include the response data (response.data) alongside resetting the login form fields (loginData) and closing the login modal (loginModal). Additionally, it triggers a subsequent method call (loadPost()) which loads posts data and updates the UI based on the successful login operation. Upon successful login, the `loginResponseData: response.data` will include the valid access token which will be used later once the user sends a request to perform any operation on the backend. Below is the code of login method.

```
login() {
    axios.post('http://127.0.0.1:8000/api/auth/login',
        this.state.loginData).then((response) => {
            let { loginResponseData } = this.state
            this.setState({
                loginResponseData: response.data,
```

```
                loginModal: false,
                loginData: { email: "", password: "" }
            })
            this.loadPost()
        })
}
```

5. Load Posts Logic

The loading posts logic is designed to fetch data from the server exclusively when a valid access token exists. This mechanism guarantees that requests to retrieve posts are made securely and only with proper authentication. Furthermore, upon successfully receiving a response from the server, the logic efficiently manages the data by updating the component's state, ensuring seamless integration of the fetched posts into the user interface for display and interaction. Below is the modified code for loadPost method.

```
loadPost() {
    if (!this.state.loginResponseData.access_token == "") {
        axios.post('http://127.0.0.1:8000/api/auth/index', {}, {
            headers: {
                'Authorization':
                    `Bearer ${this.state.loginResponseData.access_token}`
            }
        }).then((response) => {
            this.setState({
                posts: response.data
            })
        })
    }
}
```

In the above code, the Axios POST request is directed towards the URL http://127.0.0.1:8000/api/auth/index. It sends an empty object {} as the request body and includes an authorization header in the request. The authorization header contains the access token stored in the loginResponseData state object, formatted as a bearer token. This access token serves to authenticate the user's request to the server, authorizing access to protected resources, such as retrieving posts from the specified endpoint.

6. Define REST API Authentication Endpoints

Below are routes that allow react component to perform login operations and retrieve specific data, all within the '/auth' prefix and with the 'api' middleware applied for security and other middleware-related operations.

```
Route::group([
    'middleware' => 'api', 'prefix' => 'auth'
], function ($router) {
    Route::post('/login', [AuthController::class, 'login']);
    Route::post('/index', [AuthController::class, 'index']);
});
```

7. Authenticated Post Retrieval Logic

The "Authenticated Post Retrieval Logic" introduces a systematic approach to accessing posts within a secure application environment. This logic ensures that only authenticated users are granted access to retrieve post data, protecting sensitive information from unauthorized access. This logic promotes data integrity and improves the overall security and reliability of the application.

```
public function index()
    {
        $authenticated = Auth::check();
        if ($authenticated) {
            return response()->json(Post::all());
        } else
            return response()->json(['error' => 'Unauthorized'], 401);
    }
```

The index method within the AuthController is responsible for handling requests made to the '/auth/index' endpoint. The method starts by checking if a user is authenticated using Laravel's Auth::check() method. This determines whether the current request is made by an authenticated user.

Depending on the authentication status, the method returns different responses:

1. If the user is authenticated ($authenticated is true), it returns a JSON response containing all posts retrieved from the database using Post::all(). This typically means the user has access to view posts since they are authenticated.
2. If the user is not authenticated, it returns a JSON response with an 'Unauthorized' error message and a status code of 401. This indicates that the user does not have permission to access the resource (in this case, posts) without proper authentication.

8. Exercise

Following the previous discussion, your task is to design and implement authentication logic for the following actions:

1. Adding a new post.
2. Updating an existing post.
3. Deleting an existing post.

This involves ensuring that only authenticated users can perform these actions, thereby enhancing the security and integrity of the application. Each action should be carefully validated against the user's authentication status to prevent unauthorized access and maintain data confidentiality.