1. **Token Generation**: When a user logs in, the backend generates a JWT that includes the user ID and isAdmin value in its payload.
2. \*\* **Token Structure**: Ensure that the JWT payload contains the necessary claims. For example:

json

VerifyOpen In EditorEditCopy code

1{

2 "sub": "user\_id",

3 "isAdmin": true,

4 "iat": 1516239022

5}

1. **Send Token to Frontend**: After successful authentication, send the JWT to the frontend. This can be done via an HTTP response.
2. **Store the Token**: In the frontend, store the JWT securely, typically in local storage or session storage. For example:

javascript

VerifyOpen In EditorEditCopy code

1localStorage.setItem('token', jwtToken);

1. **Decode the Token**: Use a library like **jwt-decode** to decode the JWT and access its payload. Install the library using npm:

bash

VerifyOpen In EditorEditCopy code

1npm install jwt-decode

1. **Access User ID and isAdmin**: Decode the token and access the values as follows:

javascript

VerifyOpen In EditorEditCopy code

1import jwt\_decode from 'jwt-decode';

2

3const token = localStorage.getItem('token');

4if (token) {

5 const decoded = jwt\_decode(token);

6 const userId = decoded.sub; // Access user ID

7 const isAdmin = decoded.isAdmin; // Access isAdmin value

8 console.log('User ID:', userId);

9 console.log('Is Admin:', isAdmin);

10}

1. **Use the Values**: You can now use the **userId** and **isAdmin** values in your frontend application to control access to certain features or components based on the user's role.

**Security Considerations**:

* **Token Expiration**: Ensure to handle token expiration by checking the **exp** claim in the token and refreshing it if necessary.
* **Secure Storage**: Avoid storing sensitive information in local storage if possible. Consider using httpOnly cookies for better security against XSS attacks.

By following these steps, you can effectively access and utilize JWT token data in your frontend application.