Auth: SignUp And Login Components

Software Development Bootcamp

Topic

Auth SignUp and Login

Component Architecture Overview

- Parent Component: `Auth.jsx`
  - Imported into `App.jsx`
- Children Components:
  - `Signup.jsx`
  - `Login.jsx`
  - Both imported into `Auth.jsx`

Why Separate Components?

- Easier maintenance and debugging
- Isolated functionality
- Clear separation of concerns
- Simplified troubleshooting
- Independent feature addition

Why Use React Router?

Traditional web navigation requires full page reloads. React Router addresses this by:

- Enabling smooth transitions between views
- Maintaining application state during navigation
- Providing a more app-like user experience
- Supporting bookmarkable URLs in SPAs

Building Auth Components

- Create folders and component files
- Import `Auth` component into `App.jsx`

Topic

Building Signup Component

Initial Setup

- Create a `Signup.jsx` file
- Set up a simple form with inputs for user information
- Import `Signup.jsx` into `Auth.jsx`

Component Setup

- Functional component with `updateToken` prop
- `useRef` hooks to capture form inputs
- `useNavigate` for routing

```
function Signup({ updateToken }) {
 const firstNameRef = useRef(null);
 const lastNameRef = useRef(null);
 const emailRef = useRef(null);
 const passwordRef = useRef(null);
 const navigate = useNavigate()
```

useRef Key Characteristics

- Returns a mutable `ref` object
- Changes don't trigger re-renders
- Accessed via `.current` property

useRef And useState Key Differences

- `useState`:
  - Triggers re-renders on changes
  - Used for reactive data
  - Better for displaying values
- `useRef`:
  - No re-renders on changes
  - Updates are immediate
  - Better for form inputs

`useNavigate` Key Features

- Programmatic routing
- Post-signup redirection
- User flow management
- Route protection

`handleSubmit` Setup

- Prevents default form refresh
- Captures input values using **`useRef`**
- Stores values in variables for processing

```
async function handleSubmit(e) {
  e.preventDefault();
  const firstName = firstNameRef.current.value;
  const lastName = lastNameRef.current.value;
  const email = emailRef.current.value;
  const password = passwordRef.current.value;
}
```

Building The Request Body

- Creates user object with form data
- Uses object destructuring syntax
- Converts to JSON string for server

```
let bodyObj = JSON.stringify({
  firstName,
  lastName,
  email,
  password,
});
```

Request Setup

- URL Configuration
  - Endpoint for user signup
  - Local development server
- Headers Setup
  - Uses Headers constructor
  - Sets content type to JSON
- Request Options
  - Combines headers, body, method
  - Configures POST request

```
const url = "http://localhost:4000/user/signup";
const headers = new Headers();
headers.append("Content-Type",
"application/json");

const requestOptions = {
  headers,
  body: bodyObj,
  method: "POST",
};
```

Server Communication

- Async / Await Pattern
  - Handles asynchronous operations
  - Waits for server response
- Response Processing
  - Fetches from server
  - Parses JSON response

```
try {
    const response = await fetch(url, requestOptions);
    const data = await response.json();
```

Response Handling

- Success path
  - Validates success message
  - Updates authentication token
  - Navigates to movie page
- Error path:
  - Displays error message
  - Stays on signup page

```
try {
    const response = await fetch(url, requestOptions);
    const data = await response.json();
    if (data.message === "Success! User Created!") {
      updateToken(data.token);
      navigate("/movie");
    } else {
      alert(data.message);
    }
  }
```

Error Handling

- Catches any fetch errors
- Logs error messages
- Prevents app crashes
- Maintains user experience

```
catch (err) {
    console.error(err.message);
}
```

Topic

Updating The Token

Token Response

When we make a request for a token the response will include:

- Token
- Message
- User object

Token Purpose

- Provides route access
- Validates user authentication
- Manages user sessions

`updateToken`

- Receives new token
- Saves to localStorage
- Updates state
- Triggers UI updates
- The **`updateToken`** function is passed down the component tree to **`Auth.jsx`** and finally to the **`Signup`** and **`Login`** components

App.jsx

```
const updateToken = newToken => {
   localStorage.setItem('token', newToken)
   setToken(newToken)
 }
```

Using `updateToken`

- If a user is successfully created the **updateToken** function is called and the token is passed.

Signup.jsx

```
if (data.message === "Success! User Created!") {
        updateToken(data.token);
        navigate("/movie");
      } else {
        alert(data.message);
      }
```

Token Management Flow

1. User Action (Login/Signup)
    a. Server returns new token
2. `updateToken` Called
    a. Saves token to localStorage
    b. Updates React state
3. Application Updates
    a. UI reflects auth status
    b. Protected routes accessible

Local Storage & State

Why Both?

- Local Storage
  - Persists across refreshes
  - Survives browser restarts
  - String-based storage
  - Synchronous operations
- State
  - Triggers React updates
  - Enables reactive UI
  - In-memory storage
  - Asynchronous updates

Using the Token

- Once the `token` state has been updated the token can be passed as props to child components

```
function App() {
 const [token, setToken] = useState()
 const updateToken = newToken => {
   localStorage.setItem('token', newToken)
   setToken(newToken)
 }
 return (
   <Routes>
     <Route
       path="/"
       element={<Auth updateToken={updateToken} />}
     />
     <Route
       path="/movie"
       element={<MovieIndex token={token} />}
     />
   </Routes>
 );
}
```

MovieIndex.jsx Example

- Token used for authorization
- Validates requests
- Protects routes

```
function MovieIndex({token}) {
 const fetchMovies = async () => {
   const headers = new Headers();
   headers.append("Authorization", token);

   // Use token in fetch request
   const response = await fetch(url, {
     headers,
   });
 };
```

Best Practices

- State Location
  - Keep token in top-level component
  - Single source of truth
  - Centralized updates
- Prop Passing
  - Only pass to components that need it
  - Avoid unnecessary prop drilling
- Security
  - Validate token presence
  - Protect sensitive routes

Exercise

Build A Login Component