USING REACT CONTEXT

FULL STACK SKILLS BOOTCAMP

SIMPLIFYING STATE MANAGEMENT IN REACT

- **Lesson Overview:**
- In this lesson, we will be introduced to:

1. What is React Context
2. Real World examples
3. Context Providers
4. Context Consumers

INTRODUCTION TO REACT CONTEXT

- **What is React Context?**
  A mechanism for sharing values across the component tree without passing props manually at every level.
- **Core Features:**
- Provides a global state that can be accessed by any component.
- Reduces prop drilling.
- **Introduced in React 16.3 and improved over time.**

INSTALLATION

- **React Context is built into React**
  no additional library is required.

  1: Import createContext

```
import React, { createContext } from 'react';
```

  2: Create a Context object

```
const MyContext = createContext();
```

WHAT PROBLEM DOES REACT CONTEXT SOLVE?

- **The Problem:**
- Prop drilling makes passing data cumbersome and error-prone, especially with deeply nested components.
- **The Solution:**
- React Context allows data to be available anywhere in the component tree without manual prop passing.

REAL-WORLD USE CASES

- **Theme Management:**
  Dynamically switch between light and dark themes.
- **User Authentication:**
  Share user details and authentication status across the app.
- **Language Translation:**
  Provide localized strings globally.
- **State Sharing:**
  Share app-wide settings or data, like cart items in an e-commerce app.

CREATING PROVIDERS

- **What is a Provider?**
  A component that supplies the Context value to its children.
- **Steps to Create a Provider:**

1. Create Context:

```
const MyContext = createContext();
```

Wrap components with MyContext.Provider

```
<MyContext.Provider value={/* your value here */}>
  <ChildComponent />
</MyContext.Provider>
```

USING PROVIDERS

- **Access the provided value using hooks:**
- **Benefits:**
- Cleaner syntax with useContext.
- Avoids manually subscribing to updates.

```
import { useContext } from 'react';
import { MyContext } from './context';

function ChildComponent() {
  const value = useContext(MyContext);
  return <div>{value}</div>;
}
```

USE OF MULTIPLE CONTEXTS/PROVIDERS

- **Why Use Multiple Contexts?**
- Separate concerns, e.g., UserContext for user info and ThemeContext for themes.
- **Implementation Example:**

```
<UserContext.Provider value={user}>
  <ThemeContext.Provider value={theme}>
    <App />
  </ThemeContext.Provider>
</UserContext.Provider>
```

Access values separately in components:

```
const user = useContext(UserContext);
const theme = useContext(ThemeContext);
```

CREATING AND USING CONSUMERS

- **What is a Consumer?**
  A component that subscribes to a Context and receives its value.
- Example with a Consumer:
- **When to Use Consumers Over useContext:**
  - When working with class components.

```
<MyContext.Consumer>
  {value => <div>{value}</div>}
</MyContext.Consumer>
```

CONCLUSION

- React Context simplifies state management.
- Avoids prop drilling for better readability.
- Scales well with multiple contexts for modular state handling.

NEXT STEPS

- Experiment with custom hooks for context.
- Combine with state libraries like Redux or Zustand for larger apps.

Step8Up Academy

QUESTIONS?