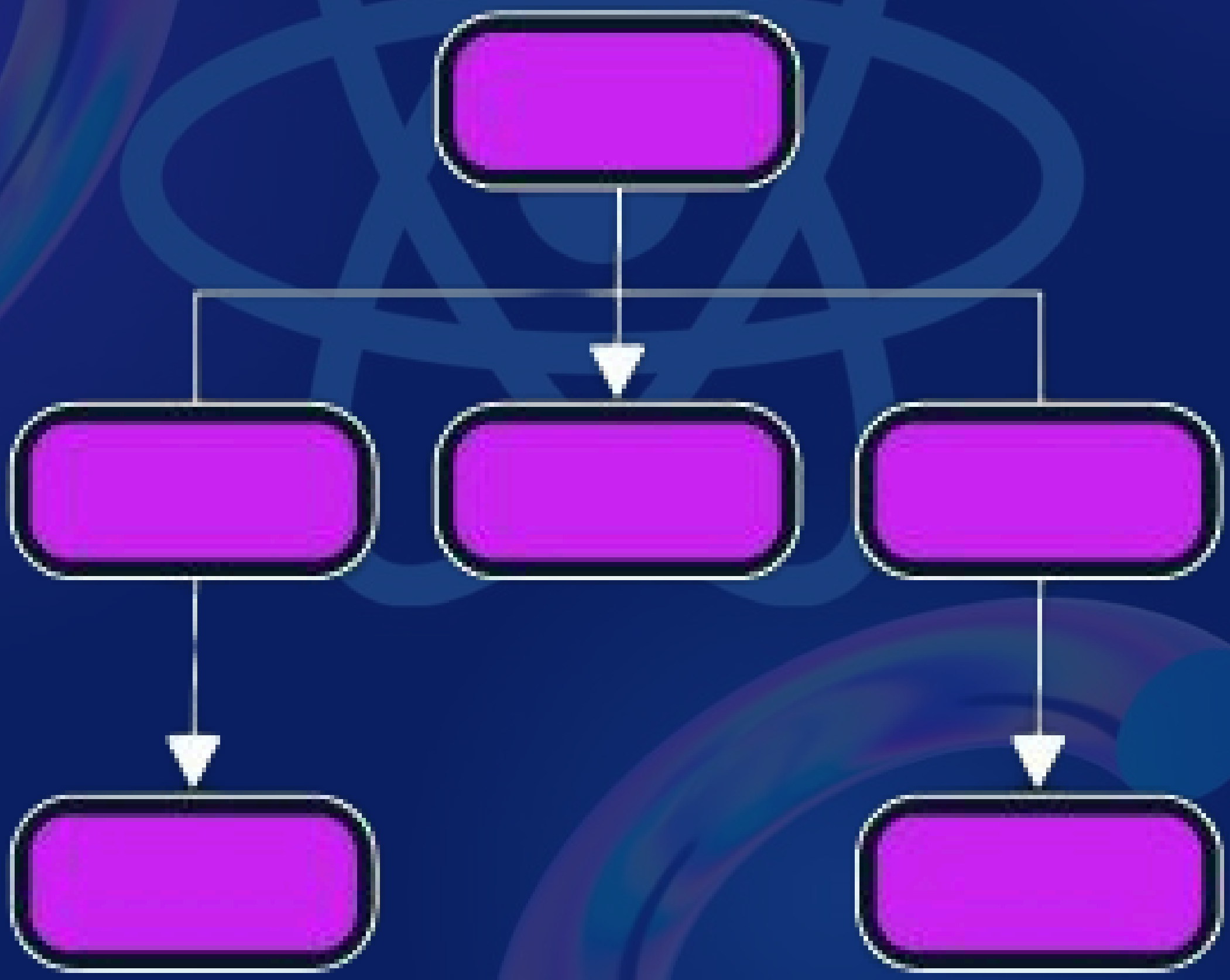
Day 20

Global State Management In React (Part 2)



@oluwakemi Oluwadahunsi

Combining Context API with Reducer

In our last lesson, we learned about using **Context API** for global state management in React. As much as that is a good approach, it can be further enhanced, especially when working with broader project base, by combining it with **Reducers**.

Combining the **Context API** with the **useReducer** hook in React is a powerful strategy for managing global state in large-scale applications.

It is an alternative to **useState**, useful when the state has multiple sub-values or when the next state depends on the previous one.

This approach allows you to handle complex state logic across multiple components without the need for a third-party library like **Redux**. Although, it is inspired by Redux, but can be used without it

@oluwakemi Oluwadahunsi

Why Combine Context API and useReducer?

Using the **Context API** alone works well for sharing state across components, but managing complex state transitions can become cumbersome.

The **useReducer** hook simplifies this by encapsulating the logic of state transitions, making your code more predictable and easier to debug.

When you combine them:

- **Context API** provides the mechanism to share state globally.
- **useReducer** manages complex state transitions in a structured way.

This combination allows you to scale state management without adding complexity.

@oluwakemi Oluwadahunsi

Benefits of This Approach

1. Centralized state management:

By combining **Context API** and **Reducers**, you can manage your application's state from a **single source**. This eliminates the need for "**prop drilling**," where props are passed through many layers of components.

2. Predictable state updates:

Using a **Reducer** ensures that state changes happen in a **predictable** and **controlled** manner. Reducers are **pure functions** that take the **current state** and **an action** to return the **new state**.

This predictable pattern makes it easier to follow and understand how the state is updated, reducing potential errors or unexpected changes.

@oluwakemi Oluwadahunsi

3. Easier debugging and testing:

With **Reducers**, **state updates** become easier to trace and test since they follow a consistent structure. Each action leads to a well-defined state transition, making it straightforward to debug state changes and test the behavior of your application in isolation.

4. No need for additional libraries:

The combination of **Context API** and **Reducers** provides a built-in, lightweight alternative to Redux. While **Redux** is a robust tool, for smaller applications, this method allows you to manage global state without the need for external libraries, keeping your project simpler and easier to maintain.

@oluwakemi Oluwadahunsi

Steps to Implement Global State Management with Context API and useReducer

- Define the Initial State and Actions
- Create a Reducer Function
- Create a Global Context
- Use the state and dispatch in your components
- Wrap your app with the provider

@oluwakemi Oluwadahunsi

Step 1. Define the Initial State and Actions

First, you need to define the **initial state** of your global data and the actions that can be performed on it. So, create a state.js file in your “src” folder like this:

```
// state.js

// Define the initial state of your application
export const initialState = {
  user: null,
  isAuthenticated: false,
  theme: 'light', // Example of additional state
};

// Define action types
export const actionTypes = {
  SET_USER: 'SET_USER',
  LOGOUT: 'LOGOUT',
  TOGGLE_THEME: 'TOGGLE_THEME',
};
```

@oluwakemi Oluwadahunsi

In the above component,

- **Initial State:** Holds the initial values of your state (user, authentication, theme).
- **Action Types:** Constants that define the types of actions that can modify the state. These actions help prevent bugs by avoiding typos in strings.

Step 2. Create a Reducer Function

The reducer is responsible for handling actions and updating the state accordingly.

So, the initial states declared in the state.js file will be updated using the **actionTypes** to get each state.

Now, create another file named reducer.js like this:

```
// reducer.js

import { actionTypes } from './state';

// Define the reducer function
export const reducer = (state, action) => {
  switch (action.type) {
    case actionTypes.SET_USER:
      return {
        ...state,
        user: action.payload,
        isAuthenticated: true,
      };

    case actionTypes.LOGOUT:
      return {
        ...state,
        user: null,
        isAuthenticated: false,
      };

    case actionTypes.TOGGLE_THEME:
      return {
        ...state,
        theme: state.theme === 'light' ? 'dark' : 'light',
      };

    default:
      return state;
  }
};
```

takes in the intial state and action for updating the state

initial state accessed

state updated

@oluwakemi Oluwadahunsi

In the component above:

Reducer Function: Takes the current state and an action, and returns a new state based on the action type.

Switch Statement: Handles different action types. For example, **SET_USER** updates the **user** and **sets the isAuthenticated flag**, **LOGOUT** clears the user, and **TOGGLE_THEME** switches between light and dark modes.

Now, our reducer setup is complete.

Step 3. Create a Global Context

Next, we create a **React context** to **provide** and **consume** the global state.

```jsx
// GlobalContext.js

import React, { createContext, useContext, useReducer } from 'react';
import { reducer } from './reducer';
import { initialState } from './state';

// Create a context for global state
const GlobalStateContext = createContext();

// Create a custom provider component
export const GlobalStateProvider = ({ children }) => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <GlobalStateContext.Provider value={{ state, dispatch }}>
      {children}
    </GlobalStateContext.Provider>
  );
};

// Custom hook to use the global state context
export const useGlobalState = () => useContext(GlobalStateContext);
```

from reducer.js

from state.js

components can now have access to these values

Here,

- **GlobalStateContext:** A context that holds the global state and the dispatch function, which are both made accessible to the component tree.

@oluwakemi Oluwadahunsi

- **GlobalStateProvider:** A wrapper component that uses useReducer to create a state and dispatch function, and passes these to the context provider.
- **useGlobalState:** A custom hook that allows any component in the app to access the global state and dispatch actions.

Step 4. Using Global State in Components

Now, we can use the **global state** and **dispatch** actions in any component.

Let’s use a User login and logout as an example. Create a login component (Login.js/jsx):

```jsx
// Login.js

import { useState } from 'react';
import { useGlobalState } from './GlobalContext';
import { actionTypes } from './state';

const Login = () => {
  const { dispatch } = useGlobalState();
  const [username, setUsername] = useState('');

  const handleLogin = () => {
    // Dispatch the action to set the user
    dispatch({
      type: actionTypes.SET_USER,
      payload: { name: username },
    });
  };

  return (
    <div>
      <input
        type="text"
        placeholder="Enter username"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      <button onClick={handleLogin}>Login</button>
    </div>
  );
};

export default Login;
```

accessing the dispatch value using thecustom hook from context

USER state is set

@oluwakemi Oluwadahunsi

In the **Login** component, the **dispatch** function is called with the **SET_USER** action to **update** the **global state** when the **login button** is clicked.

Displaying the User Info: create another component called UserInfo.js:

```
// UserInfo.js

import React from 'react';
import { useGlobalState } from './GlobalContext';

const UserInfo = () => {
  const { state } = useGlobalState();

  return (
    <div>
      {state.isAuthenticated ? (
        <p>Welcome, {state.user.name}!</p>
      ) : (
        <p>Please log in.</p>
      )}
    </div>
  );
};

export default UserInfo;
```

- **state.isAuthenticated:** The component conditionally renders content based on the **isAuthenticated flag** in the global state.

Let’s add a global logout feature:

Create a Logout Component:

```
// Logout.js

import React from 'react';
import { useGlobalState } from './GlobalContext';
import { actionTypes } from './state';

const Logout = () => {
  const { dispatch } = useGlobalState();

  const handleLogout = () => {
    // Dispatch the action to log out the user
    dispatch({ type: actionTypes.LOGOUT });
  };

  return <button onClick={handleLogout}>Logout</button>;
};

export default Logout;
```

When the **Logout button** is clicked, it dispatches the **LOGOUT** action to reset the user in the global state

Step 5: Wrap your app with the provider

Next is to wrap your app with the **GlobalStateProvider**, you simply need to import the GlobalStateProvider component (which you created earlier) and use it as a wrap around your entire app.

It could be in your **App.js** or **main.jsx** (vite)

```jsx
// App.js

import { GlobalStateProvider } from './GlobalContext'; // Import your GlobalStateProvider
import Login from './Login'; // Import the Login component
import UserInfo from './UserInfo'; // Import the UserInfo component
import Logout from './Logout'; // Import the Logout component

function App() {
  return (
    // Wrap your entire app inside the GlobalStateProvider
    <GlobalStateProvider>
      <div className="App">
        <h1>Global State with Context and useReducer</h1>
        {/* Components inside this provider will have access to the global state */}
        <Login />
        <UserInfo />
        <Logout />
      </div>
    </GlobalStateProvider>
  );
}

export default App;
```

- **GlobalStateProvider:** This provider component wraps the entire application, making the global **state** and **dispatch** function available to all nested components (Login, UserInfo, Logout).

- **Children Components:** The **Login**, **UserInfo**, and **Logout** components can now access the global state and dispatch actions because they are nested inside the **GlobalStateProvider**.

In summary, **GlobalStateProvider** encapsulates the whole application, meaning any component within it has access to the global state without needing to manually pass **props** down through multiple levels.

Best Practices for this Combination

1. Separate Concerns:

Keep your context and reducer logic in separate files. This improves maintainability and readability

2. Use Multiple Contexts:

Separate your state into logical domains, and make use of multiple contexts for different parts of your app when necessary. This can improve performance and make your code more modular, also note that too much global state can become difficult to manage.

3. Memoization

Use React.memo for components that often receive the same props and for expensive computations. This can help prevent unnecessary re-renders.

@oluwakemi Oluwadahunsi

Stay tuned for Part 3, where we'll cover the next advanced topic which is the use of Redux state management library to handle complex state logic.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app