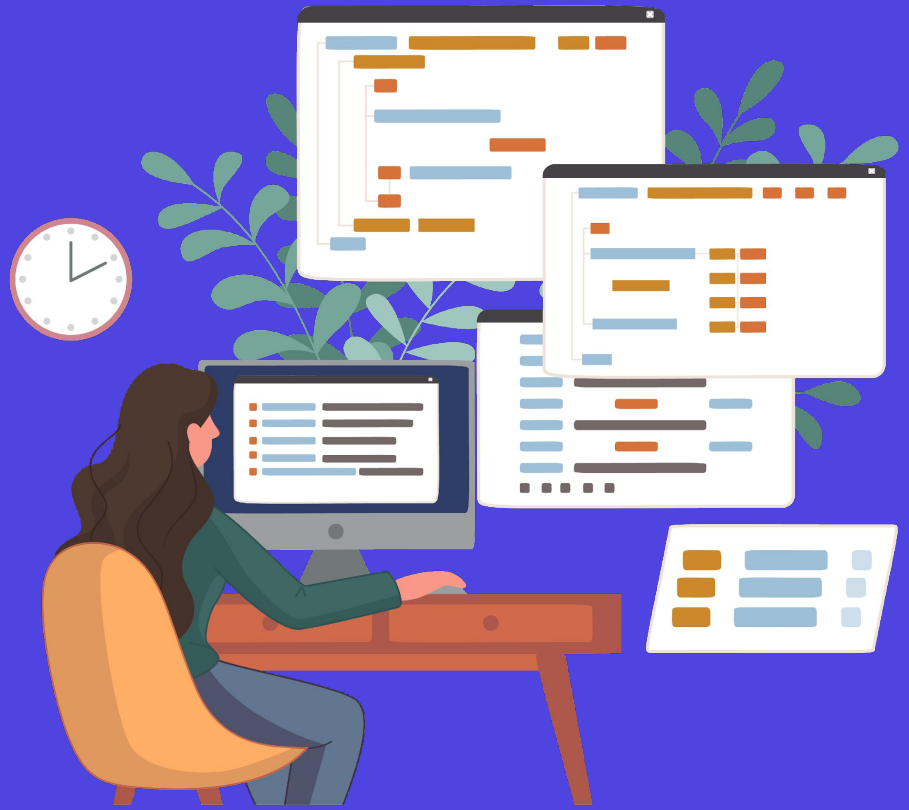
React Context API + useReducer



Relevel
by Unacademy

Topics Covered

How data can be passed among components in React application.

Redux overview and its components.

Context Api and its components.

useReducer hooks and its uses.

How we can mock the functionalities of Redux with Context api.

How data can be passed among components in React application ?

There are three approaches for this:

- Pass data from Parent to child
- Pass data from Child to parent
- Pass data among sibling Components

Pass Data From Parent Component to a Child Component

- Create two components, one parent and one child.
- Import the child component into the parent component and return it
- Now we can create a prop named data and pass it to the child component as props

Pass Data From Child Component to Parent Component

- Create a function in the parent component called childToParent and an empty state named data
- Import the child component into the parent component and return it
- Pass the childToParent function as a prop to the child component
- Accept this function call as props and assign it to an onClick event.

Passing Data Among Siblings

For this approach also we can use a callback function just like the previous approach. But it is not at all maintainable as our component tree continues to grow. so it can be converge to:-

- Combine the previous two methods
- Using Redux
- Using React's Context API

Problem With Prop Passing Approach

If we have a large application with several hundreds of components and still we want to pass/share data among the components, we can't rely on props, as it creates deep coupling between the components passing and receiving data.

Prop Drilling is the process by which you pass data from one part of the React Component tree to another by going through other parts that do not need the data but only help in passing it around.

Redux a overview and its components

Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for a state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

Components Redux

- Store
- Actions
- Reducers
- Combine Reducers
- useSelector hook
- Dispatch

Context API And Its Components

Context API is a (kind of) new feature added in version 16.3 of React that allows one to share state across the entire app (or part of it) lightly and with ease. Let's see how to use it. React.createContext() is all you need. It returns a consumer and a provider. The provider is a component that, as its name suggests, provides the state to its children. It will hold the "store" and parent all the components that might need that store. As it so happens, the consumer is a component that consumes and uses the state.

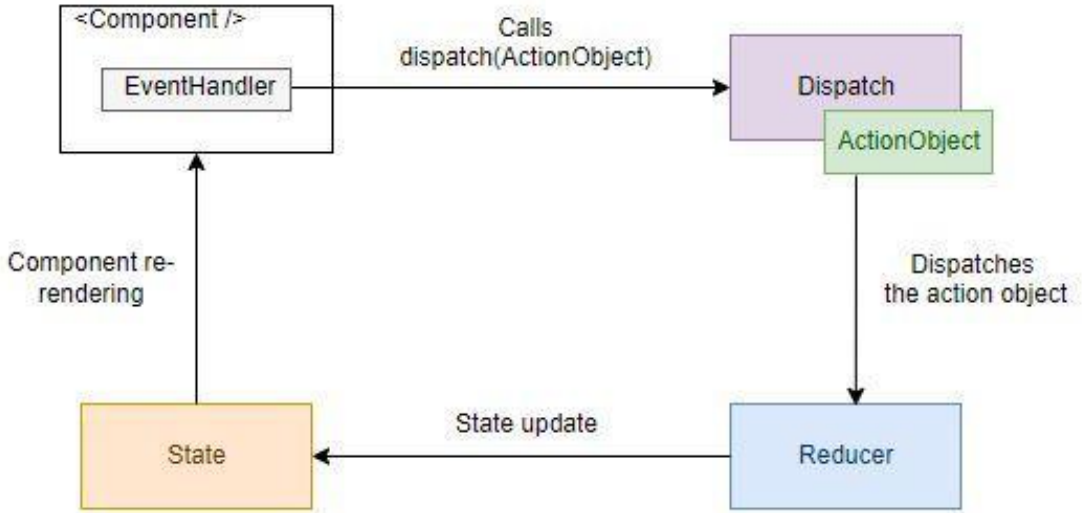
Components Of Context API

- createContext
- Provider
- Consumer
- useContext

userReducer Hook

useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. useReducer also lets you optimize performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.

useReducer()


<Component />
EventHandler
Calls
dispatch(ActionObject)
Dispatch
ActionObject
Dispatches
the action object
Reducer
State update
State
Component re-
rendering

How we can mock the functionalities of Redux with Context API and useReducer

Now that we know how the Context API and the useReducer Hook work individually, let's see what happens when we combine them in order to get the ideal global state management solution for our application.

We'll create our global state in a store.js file:

```
// store.js
import React, {createContext, useReducer} from 'react';

const initialState = {};
const store = createContext(initialState);
const { Provider } = store;

const StateProvider = ( { children } ) => {
  const [state, dispatch] = useReducer((state, action) => {
    switch(action.type) {
      case 'action description':
        const newState = // do something with the action
        return newState;
      default:
        throw new Error();
    };
  }, initialState);

  return <Provider value={{ state, dispatch }}>{children}</Provider>;
};

export { store, StateProvider }
```

Accessing State Globally:

In order to access our state globally, we'll need to wrap our root <App/> component in our StoreProvider before rendering it in our ReactDOM.render() function:

```
1  // root index.js file
2  import React from 'react';
3  import ReactDOM from 'react-dom';
4  import App from './App';
5  import { StateProvider } from './store.js';
6
7  const app = (
8    <StateProvider>
9      <App />
10   </StateProvider>
11 );
12
13 ReactDOM.render(app, document.getElementById('root'));
```

Accessing State Globally:

Now, our store context can be accessed from any component in the component tree. To do this, we'll import the useContext Hook from react and the store from our ./store.js file:

```
// exampleComponent.js
import React, { useContext } from 'react';
import { store } from './store.js';

const ExampleComponent = () => {
  const globalState = useContext(store);
  console.log(globalState); // this will return { color: red }
};
```

Adding And Removing Data From State:

We've seen how we can access our global state. We'll need the dispatch method from our store context to add and remove data from our state. We only need to call the dispatch method and pass in an object with the type (the action description as defined in our StateProvider component) as its parameter:

```
1  // exampleComponent.js
2  import React, { useContext } from 'react';
3  import { store } from './store.js';
4
5  const ExampleComponent = () => {
6    const globalState = useContext(store);
7    const { dispatch } = globalState;
8
9    dispatch({ type: 'action description' })
10 };
```

SO Yes, we did mock the redux with native available context API !!

Homework problems:

1. Create a to do list app that manage list of items with context and a reducer to add and delete to do list items
2. Basic example using Redux together with React. For simplicity, it re-renders the React component manually when the store changes
3. Create a reducer that returns the next state for each possible authentication case (LOGIN_SUCCESS, LOGIN_FAILURE, etc).

Thank You