Purwadhika
Digital Technology School

Full Stack Web Development

State Management

Job Connector Program

Outline

State Management:

- Global State
- useContext
- Local Storage, session storage and cookies
- Redux & Redux Toolkit

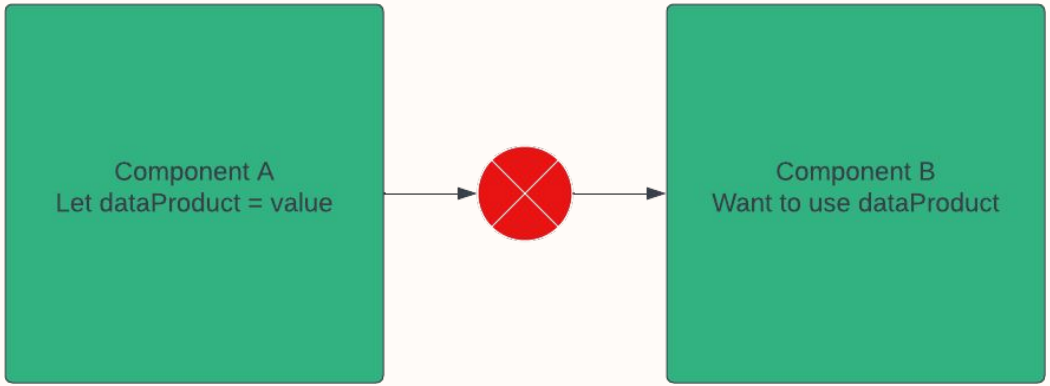
Global State

When we are dealing with a React project, of course we will not be separated from state management or how we manage to store our data.

For that we need to have a global state. What is global state? Global state is a global data repository, which means that it is not tied to a specific component scope so that all components can have the same data access.

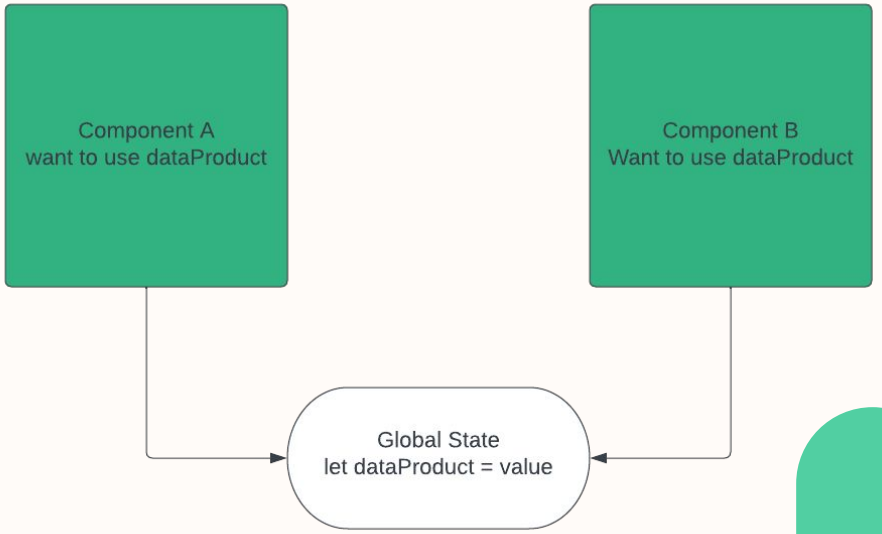
Local State

Take a look at this example, there is state in component A where is called Local State, like the name, this state only can be used in component A. When the other component, let's say component B want to use local data in component A, of course component B cannot use the state of component A.



Global State

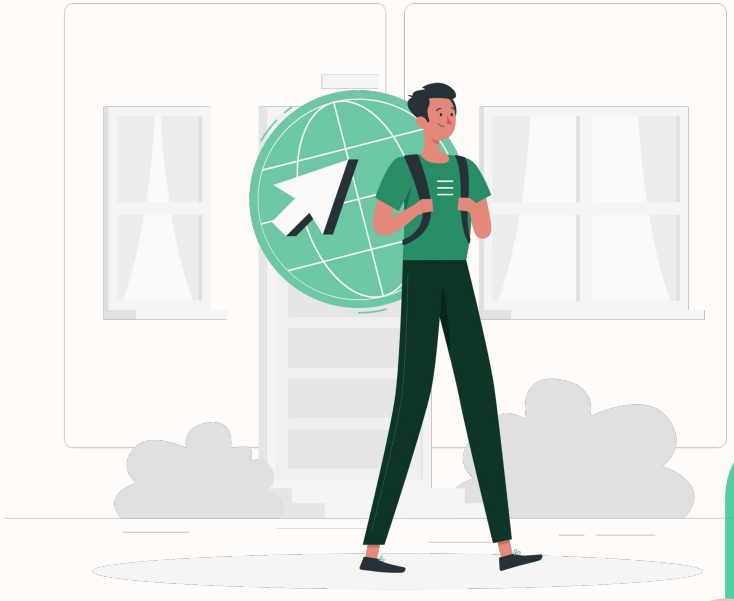
With the Global State as a global data repository, we can store dataProducts in the global state and the data is freely accessible by all components connected to the global state.



Local Storage

This read-only interface property provides access to the Document's local storage object, the stored data is stored across browser sessions.

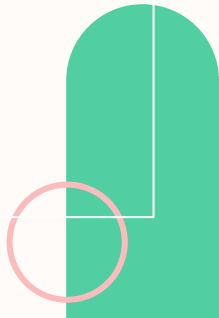
The data stored in LocalStorage is specific to a protocol in the document. If the site is loaded over HTTP (e.g., http://example.com), localStorage returns a different object than if it is loaded over HTTPS (e.g., https://abc.com).



Local Storage

Local storage has 4 methods:

- **setItem() Method –** This method takes two parameters one is key and another one is value. It is used to store the value in a particular location with the name of the key. Syntax: `localStorage.setItem(key, value)`
- **getItem() Method –** This method takes one parameter that is key which is used to get the value stored with a particular key name. Syntax: `localStorage.getItem(key)`
- **removeItem() Method –** This is method is used to remove the value stored in the memory in reference to key. Syntax: `localStorage.removeItem(key)`
- **clear() Method –** This method is used to clear all the values stored in localstorage. Syntax: `localStorage.clear()`



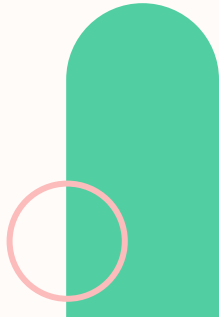
Session Storage

Session Storage objects can be accessed using the sessionStorage read-only property. The difference between sessionStorage and localStorage is that localStorage data does not expire, whereas sessionStorage data is cleared when the page session ends.

Session Storage

Session Storage has 4 methods:

- **setItem() Method –** This method takes two parameters one is key and another one is value. It is used to store the value in a particular location with the name of the key. Syntax: sessionStorage.setItem(key, value)
- **getIteam() Method –** This method takes one parameter that is key which is used to get the value stored with a particular key name. Syntax: sessionStorage.getItem(key)
- **removeItem() Method –** This is method is used to remove the value stored in the memory in reference to key. Syntax: sessionStorage.removeItem(key)
- **clear() Method –** This method is used to clear all the values stored in the session storage. Syntax: sessionStorage.clear()



Cookies

The term “cookie” refers to just the textual information about a website. In order to recognize you and show you results according to your preferences, this website saves some information in your local system when you visit a particular website. The history of the internet has long been marked by the use of cookies. A website visitor asks the server for a web page when they visit it. Every request for a server is unique.

Difference Between Local Storage, Session Storage, And Cookies

Local Storage	Session Storage	Cookies
The storage capacity of local storage is 5MB/10MB	The storage capacity of session storage is 5MB	The storage capacity of Cookies is 4KB
As it is not session-based, it must be deleted via javascript or manually	It's session-based and works per window or tab. This means that data is stored only for the duration of a session, i.e., until the browser (or tab) is closed	Cookies expire based on the setting and working per tab and window
The client can only read local storage	The client can only read local storage	Both clients and servers can read and write the cookies

useContext

In previous lesson we already told that useContext is a hooks that useful when we want to use state globally . it means we dont need props to passing value from parent component to children component. Why we mentioned useContect in state management session ? because useContext is a bone of state management like redux toolkit. If we understand useContext work , you will be understand how state management like redux toolkit works.

Take a look at this example state and props without useContext

```
import { useState } from "react"

export default function MyPage( ) {

  const [theme,setTheme] = useState("light")

  return (
     <Form theme={theme} />
     <Button onClick={() => {
        setTheme('light');
      }}>
        Switch to light theme
     </Button>
  );
}

function Form(props){
  return(<form className={props.theme}>
    <div>
      <input type="text" name="name">
      <Button theme={props.theme} />
    </div>
  </form>)
}

function Button(props) {
    return (
      <button className={props.theme}
type="submit">Submit</button>
    )
}
```

useContext

And now we want to implement useContext to our code at previous slide.

This is the first step.

We began to update MyPage Component. Import createContext and useContext. And then create ThemeContext from createContext as Component that wrapped up MyPage component

```
import { useState, createContext, useContext } from "react"

const ThemeContext = createContext(null)

export default function MyPage( ) {

  const [theme, setTheme] = useState("light")

  return (
     <ThemeContext.Provider value={theme}>
       <Form />
       <Button onClick={( ) => {
          setTheme('light');
        }}>
          Switch to light theme
       </Button>
     </ThemeContext.Provider >
  );
}
```

useContext

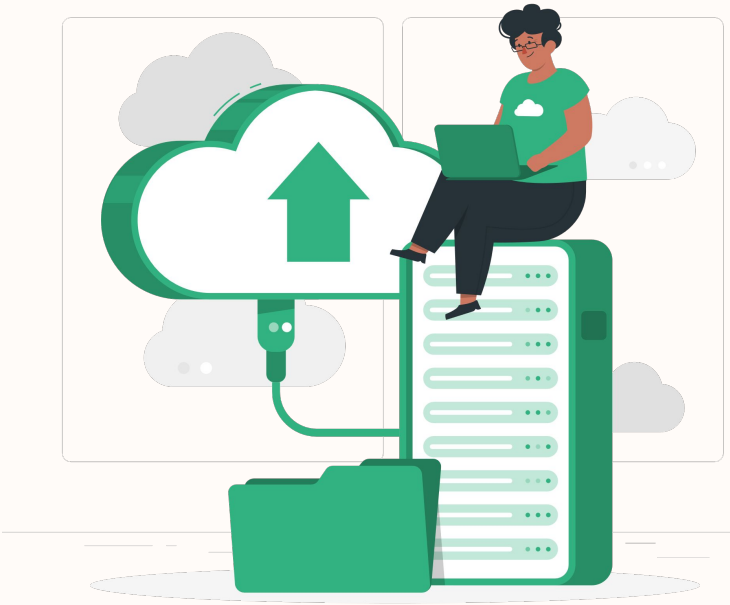
And after that, we can called useContext in children component that needs props from parent component. No matter how deep the component is, the children component can access the props using useContext

```
function Form(props){

  const theme = useContext(ThemeContext)

  return(<form className={theme}>
    <div>
      <input type="text" name="name">
      <Button  />
    </div>
  </form>)
}

function Button(props) {

    const theme = useContext(ThemeContext)

    return (
      <button className={theme} type="submit">Submit</button>
    )
}
```

Redux & Redux Toolkit

What is Redux?

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



Redux & Redux Toolkit

Redux helps you manage "global" state - state that is needed across many parts of your application.

The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur.

That’s why you need redux.

Redux & Redux Toolkit

Redux is more useful when:

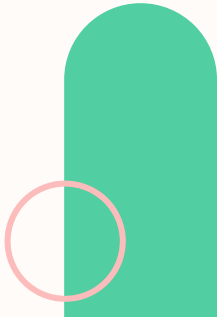
- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people

So, **not all apps need Redux.**

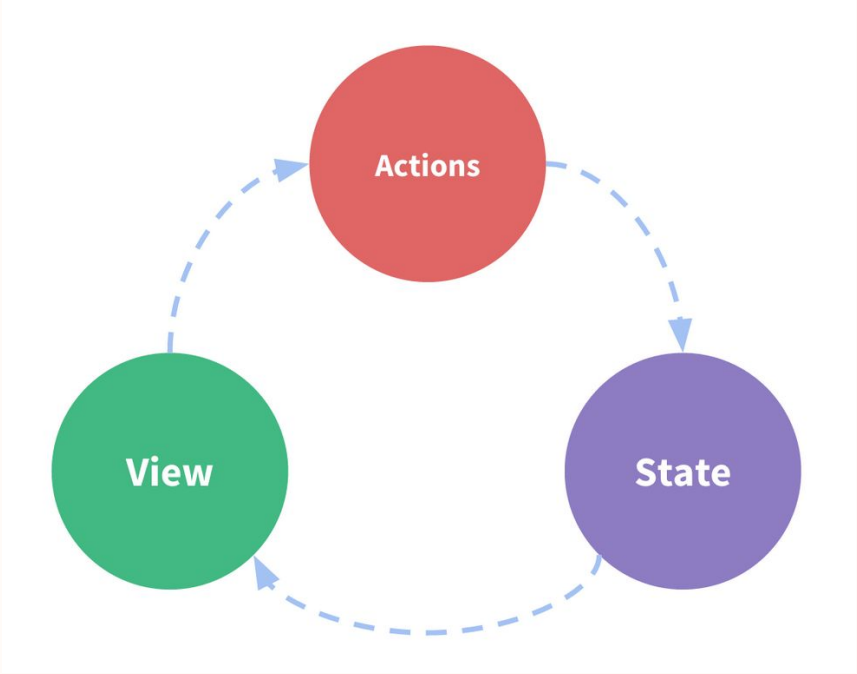
Redux Libraries and Tools

Redux is a small standalone JS library. However, it is commonly used with several other packages:

- **React-Redux :** Redux can integrate with any UI framework, and is most frequently used with React. **React-Redux** is official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.
- **Redux Toolkit :** Redux Toolkit is our recommended approach for writing Redux logic. It contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.
- **Redux DevTools Extension :** The Redux DevTools Extension shows a history of the changes to the state in your Redux store over time. This allows you to debug your applications effectively, including using powerful techniques like "time-travel debugging".



Redux Data Flow



Basic Example

The whole global state of your app is stored in an object tree inside a single *store*. The only way to change the state tree is to create an *action*, an object describing what happened, and *dispatch* it to the store. To specify how state gets updated in response to an action, you write pure *reducer* functions that calculate a new state based on the old state and the action.

Example

This is a reducer - a function that takes a current state value and an action object describing “what happened”, and returns a new state value. A reducer’s function signature is: (state, action) => newState

The Redux state should contain only plain JS objects, arrays, and primitives. The root staete value is usually an object. It’s important that you should not mutate the state object, but return a new object if the state changes.

You can use any conditional logic you want in a reducer. In this example, we use a switch statement, but it’s not required.

```
import {createStore} from 'redux';

function counterReducer(state={ value: 0 }, action) {
  switch (action.type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
      return state
  }
};
```

```
import {createStore} from 'redux';

function counterReducer(state={ value: 0 }, action) {
  switch (action.type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
      return state
  }
};

let store = createStore(counterReducer)
store.dispatch({ type: 'counter/incremented' }) //{value: 1}
store.dispatch({ type: 'counter/incremented' }) //{value: 2}
store.dispatch({ type: 'counter/decremented' }) //{value: 1}
```

Create a Redux store holding the state of your app.

You can use subscribe() to update the UI in response to state changes. Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.

The only way to mutate the internal state is to dispatch an action.

Redux Toolkit

```
import { configureStore } from '@reduxjs/toolkit'

export const store = configureStore({
  reducer: {},
})
```

Redux Toolkit simplifies the process of writing Redux logic and setting up the store.

Add the Redux Toolkit and React-Redux packages to your project:

npm install @reduxjs/toolkit react-redux

Create a file named src/app/store.js. Import the configureStore API from Redux Toolkit. We'll start by creating an empty Redux store, and exporting it

Redux Toolkit

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import { store } from './app/store'
import { Provider } from 'react-redux'

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Once the store is created, we can make it available to our React components by putting a React-Redux <Provider> around our application in src/index.js. Import the Redux store we just created, put a <Provider> around your <App>, and pass the store as a prop

Redux Toolkit

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

Add a new file named src/features/counter/counterSlice.js. In that file, import the createSlice API from Redux Toolkit.

Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated. Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.

Redux Toolkit

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```

Next, we need to import the reducer function from the counter slice and add it to our store. By defining a field inside the reducer parameter, we tell the store to use this slice reducer function to handle all updates to that state.

Redux Toolkit

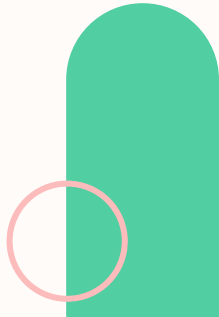
```jsx
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button onClick={() => dispatch(increment())}>
          Increment
        </button>
        <span>{count}</span>
        <button onClick={() => dispatch(decrement())}>
          Decrement
        </button>
      </div>
    </div>
  )
}
```

Now we can use the React-Redux hooks to let React components interact with the Redux store. We can read data from the store with useSelector, and dispatch actions using useDispatch. Create a src/features/counter/Counter.js file with a <Counter> component inside, then import that component into App.js and render it inside of <App>.

Redux Toolkit

Now, any time you click the "Increment" and "Decrement" buttons:

- The corresponding Redux action will be dispatched to the store
- The counter slice reducer will see the actions and update its state
- The <Counter> component will see the new state value from the store and re-render itself with the new data



Thank You!