REACTJS

A study material for the students of GLS University

Prepared by : Harshita Maheshwari

Subject : MERN Stack

React Form Handling

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

Controlled component: An input form element whose value is controlled by React in this way is called a “controlled input or Controlled Component”.

Uncontrolled component: Where form data is handled by the DOM itself. We will use ref to get the input values and Perform Operations using this data.

Controlled Component

Form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "controlled components".

In a controlled component, form data is handled by a React component.

When Use Controlled Component-

You need to write an event handler for every way your data can change and pipe all of the input state through a React component.

React Context

Context allows passing data through the component tree without passing props down manually at every level.

In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

Context solves the problem of “prop drilling”

“prop drilling” — this is when you have a top-level component and a child component way down in the hierarchy that needs the same data. You would pass that data down, parent-to-child, for each of the intermediary components but that sucks because now each layer of in component hierarchy have to know about the object being passed down even when they themselves don’t need it, just their children.

Hooks

Hooks are functions that let you “hook into” React state and lifecycle features from function components.

Hooks allow you to use React without classes. It means you can use state and other React features without writing a class.

React provides a few built-in Hooks like useState, useEffect etc

Hooks are a new addition in React 16.8.

Rules for Hooks

Only call Hooks at the top level – We should not call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function.

Only call Hooks from React functions – We should not call Hooks from regular JavaScript functions. Instead, call Hooks from React function components or call Hooks from custom Hooks

React relies on the order in which Hooks are called.

Hooks don’t work inside classes.

Hooks Types

Built- in Hooks -

- useState
- useEffect
- useRef
- useCallback
- useMemo
- useContext
- useReducer

Custom Hooks -

You can create your own custom hooks if you have stateful logic that is needed by multiple components in you application.

useState()

useState ( ) - useState is a Hook that allows you add React state to function components. We call it inside a function component to add some local state to it.

useState returns a pair - the current state value and a function that lets you update it.

React will preserve this state between re-renders.

You can call this function from an event handler or somewhere else.

```
import React, {useState} from 'react'

function HookCounter() {

  const [count, setCount] = useState(0)

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count {count}</button>
    </div>
  )
}

export default HookCounter
```

useState with previous state

```
import React, { useState } from 'react'
function HookCounterTwo() {
    const initialCount = 0
    const [count, setCount] = useState(initialCount)
    const incrementFive = () => { setCount(prevCount => prevCount + 1) }
    const decrementFive = () => { setCount(prevCount => prevCount - 5) }
    return (
        <>
            Count: {count}
            <button onClick={() => setCount(initialCount)}>Reset</button>
            <button onClick={() => setCount(prevCount => prevCount + 1)}>Increment</button>
            <button onClick={() => setCount(prevCount => prevCount - 1)}>Decrement</button>
            <button onClick={incrementFive}>Increment 5</button>
            <button onClick={decrementFive}>Decrement 5</button>
        </>
    )
}
export default HookCounterTwo
```

useState with object

```
import React, { useState } from 'react'
function HookCounterThree() {
    const [name, setName] = useState({ firstName: '', lastName: '' })
    return (
        <form>
            <input type="text" value={name.firstName}
                onChange={e => setName({ ...name, firstName: e.target.value })} />
            <input type="text" value={name.lastName}
                onChange={e => setName({ ...name, lastName: e.target.value })} />
            <h2>Your first name is - {name.firstName}</h2>
            <h2>Your last name is - {name.lastName}</h2>
            <h2>{JSON.stringify(name)}</h2>
        </form>
    )
}
export default HookCounterThree
```

useState with array

```
import React, { useState } from 'react'
function HookCounterFour() {
    const [items, setItems] = useState([])
    const addItem = () => {
        setItems([
            ...items,
            { id: items.length, value: Math.floor(Math.random() * 10) + 1 }
        ])
    }
    return (
        <div>
            <button onClick={addItem}>Add a number</button>
            <ul>
                {items.map(item => (
                    <li key={item.id}>{item.value}</li>
                ))}
            </ul>
        </div>
    )
}
export default HookCounterFour
```

useEffect Hooks

The Effect Hook lets you perform side effects in function components.

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.

It is a close replacement for componentDidMount, componentDidUpdate and componentWillUnmount

```jsx
import React, { useState, useEffect } from 'react'
function HookCounterOne() {
    const [count, setCount] = useState(0)
    const [name, setName] = useState('')
    useEffect(() => {
        console.log('useEffect - Updating document title ')
        document.title = `You clicked ${count} times`
    }, [])
    const increaseCount=() => {
        setCount(count + 1)
        console.log(`Count increased by 1`)
    }
    return (
        <div>
            <input type="text" value={name} onChange={e => setName(e.target.value)} />
            <button onClick={increaseCount}>
                useEffect - Click {count} times
            </button>
        </div>
    )
}
export default HookCounterOne
```

```jsx
import React, { useState, useEffect } from 'react'
function HookCounterOne() {
    const [count, setCount] = useState(0)
    const [name, setName] = useState('')
    useEffect(() => {
        console.log('useEffect - Updating document title ')
        document.title = `You clicked ${count} times`
    }, [name])
    return (
        <div>
            <input type="text" value={name} onChange={e => setName(e.target.value)} />
            <button onClick={() => setCount(count + 1)}>
                useEffect - Click {count} times
            </button>
        </div>
    )
}
export default HookCounterOne
```

Context

Context allows passing data through the component tree without passing props down manually at every level.

In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

There are three main steps to use the React context into the React application:

Create context

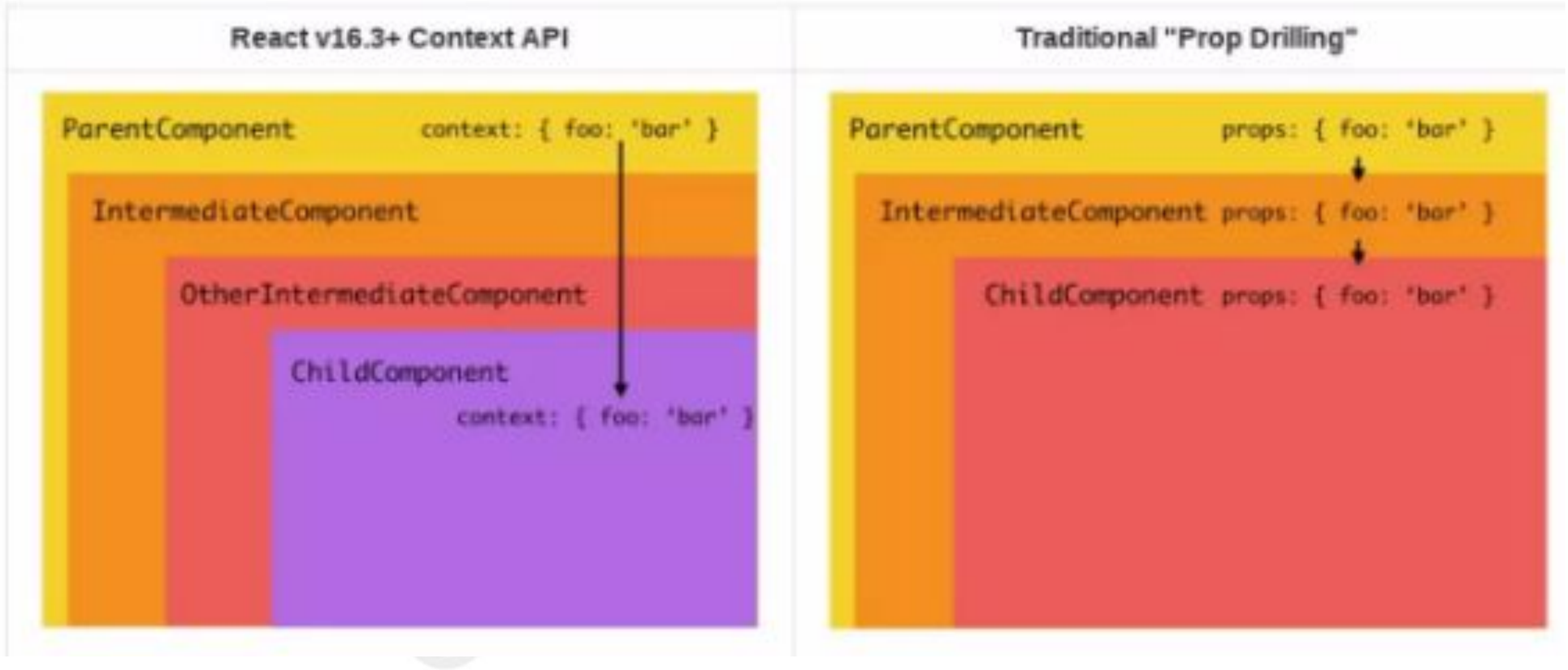
Setup a context provider and define the data which you want to store.

Use a context consumer whenever you need the data from the store

Prop Drilling

Context solves the problem of “prop drilling”

“prop drilling” — this is when you have a top-level component and a child component way down in the hierarchy that needs the same data. You would pass that data down, parent-to-child, for each of the intermediary components but that sucks because now each layer of in component hierarchy have to know about the object being passed down even when they themselves don’t need it, just their children.


React v16.3+ Context API
ParentComponent
context: { foo: 'bar' }
IntermediateComponent
OtherIntermediateComponent
ChildComponent
context: { foo: 'bar' }
Traditional "Prop Drilling"
ParentComponent
props: { foo: 'bar' }
IntermediateComponent
props: { foo: 'bar' }
ChildComponent
props: { foo: 'bar' }

React Router

What is React Router?

The go-to routing solution for React applications, enabling multi-page navigation within single-page apps (SPAs).

Bridges React 18 to React 19 with flexible usage modes: Declarative, Data, and Framework approaches.

Delivers seamless URL management without full page reloads, creating smooth user experiences.

Installing React Router: cd into your project directory and type below command in terminal - npm i react-router

Once you have this library there are three things you need to do in order to use React Router.

- Setup your router
- Define your routes
- Handle navigation

Create a Router and Render

```
import React from "react";
import ReactDOM from "react-dom/client";
import { createBrowserRouter } from "react-router";
import { RouterProvider } from "react-router/dom";

const router = createBrowserRouter([
  {
    path: "/",
    element: <div>Hello World</div>,
  },
]);

const root = document.getElementById("root");

ReactDOM.createRoot(root).render(
  <RouterProvider router={router} />,
);
```

Configuring Routes

```jsx
import { createBrowserRouter } from "react-router";

function Root() {
  return <h1>Hello world</h1>;
}

const router = createBrowserRouter([
  { path: "/", Component: Root },
]);
```

```jsx
createBrowserRouter([
  {
    path: "/",
    Component: Root,
    children: [
      { index: true, Component: Home },
      { path: "about", Component: About },
      {
        path: "auth",
        Component: AuthLayout,
        children: [
          { path: "login", Component: Login },
          { path: "register", Component: Register },
        ],
      },
    ],
  }
])
```

child (nested) routes

Nested routes are route definitions where a route has children. Those children inherit/append to the parent path and render inside the parent's component.

The parent component must render an <Outlet /> where the child UI will appear.

Outlet

<Outlet /> is a placeholder in a parent route's element where the child route's element will be rendered.

```
function App() {
  return (
    <>
      <Navbar />
      <div className="container">
        <Outlet />
      </div>
    </>
  )
}
```

index: true (index routes)

An index route is the default child of a parent. It renders when the parent path matches exactly (no extra segment).

In object routes: { index: true, element: <Home /> }. In JSX <Route> form: <Route index element={<Home/>} />.

Index route has no path and cannot have children.

Why prefer index to path: "":

index is explicit and intended for default rendering. path: "" exists but can be confusing; use index for clarity.

Link

Link is the basic navigation component.

It creates an <a> tag that changes the URL without reloading the page.

Use it when you just need navigation, no special styles for "active" links.

```
<nav>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
    <Link to="/contact">Contact</Link>
  </nav>
```

NavLink

NavLink is like Link, but smarter.

It lets you apply active styles (or classes) when the link matches the current URL.

Useful for navigation menus, sidebars, or tabs where you want to highlight the active page.

```
<nav>
  <NavLink to="/" end
    style={({ isActive }) => ({
      fontWeight: isActive ? "bold" : "normal", color: isActive ? "red" : "blue"})}>
    Home </NavLink>
  <NavLink to="/about">About</NavLink>
  <NavLink to="/contact">Contact</NavLink>
</nav>
```

Catch-all route (path: "*")

If no other route matches, the * route will render.

```
const myRoutes = createBrowserRouter([
  {path:"/", element:<App/>,
    children :[
    {path:'', element:<Home/>},
    {path:'home', element:<Home/>},
    {path:'login', element:<Login/>},
    {path:'about', element:<About/>},
    {path:'products', element:<Products/>},
    {path:'contact', element:<ConatctUs/>},
    ]
  } ,
  {path:'/register', element:<Register/>},
  {path:'*', element:<Pagenotfound/>},
])
```

Thank you