Data is the New Oil

Higher Order Components

Higher order components (HOC) are functions that take a component and return a new component. This pattern is used to enhance the functionality of existing components.

1. **What is the theory behind it?**
    - Essentially, a higher order component takes a component as input, adds additional features or behavior to it, and then returns the modified component.
2. **Why do we use Higher Order Components?**
    - Higher order components are often pure functions, meaning they don't modify the input component directly.
    - For example, if we have a component called `RestroCard` and we want to enhance its functionality, we can use a higher order component to add new features without directly modifying the `RestroCard` component.

    Syntax:

    ```
    const EnhancedRestroCard = higherOrderComponent(BaseComponent);
    ```

    - This allows us to keep the original features of `RestroCard` intact while adding new functionalities on top of it. Higher order components serve the purpose of enhancing the components without altering their core functionality.
3. **How to define hoc?**

    ```
    export const withPromotedLabel = (RestroCard) => {
    return () => {
        return (
        <>
            <RestroCard />
        </>
        );
    };
    };
    ```

    - First, We define the variable name with small letter because it is a function,
    - then take a component name as a parameter(*in capital letter because it is a component*),
    - then returning a component, (*at the end component is function so writing arrow function in return*),
    - then finally in function return we pass the component.
4. **How to use hoc?**

Syntax:

```
const RestroCardWithOpenStatus = withRestroOPenStatus(RestroCard);
```

```
{restaurant.info.availability.opened === true ? (
<RestroCardWithOpenStatus resDataList={restaurant} />
) : (
<RestroCard resDataList={restaurant} />
)}
```

how to use props ?

```
export const withRestroOPenStatus = (RestroCard) => {
return (props) => {
    return (
    <>
        <label>Restaurant is Open!!</label>
        <RestroCard {...props} />
    </>
    );
};
};
```

We used spread operator, so it will pass all the props that i receive.

```
{...props}
```

Note : Also, see hoc from `D:\Code Toor SSD\FrontEnd Developer Notes\react_app\react_tutorials\random-practice\src\components\hoc` for more information.

Prop Drilling

> In React, prop drilling refers to the process of passing down props (short for properties) through multiple layers of nested components. This happens when a piece of data needs to be transferred from a higher-level component to a deeply nested child component, and it must pass through several intermediary components in between.

> 💡 Here's a simple example to illustrate prop drilling in React:

```
// Top-level component
function App() {
  const data = "Hello, prop drilling!";
  return (
    <div>
      <ParentComponent data={data} />
    </div>
  );
}

// Intermediate component
function ParentComponent({ data }) {
  return (
    <div>
      <ChildComponent data={data} />
    </div>
  );
}

// Deeply nested component that actually uses the data
function ChildComponent({ data }) {
 return <div>{data}</div>;
}
```

In this example, the `data` prop is passed from the App component through the `ParentComponent down to the ChildComponent`. The ParentComponent itself doesn't use the data prop; it merely passes it down. `This process of passing data through intermediate components that don't use the data is what is referred to as prop drilling .`

Prop drilling can make the code harder to maintain, especially as the application grows and the number of components in the hierarchy increases. To mitigate this, developers often use other state management solutions, like the `React Context API`, `Redux`, or `other state management libraries`, to avoid passing props through multiple layers of components. These alternatives provide a centralized way to manage and access state without the need for prop drilling.

Lifting The State Up

> `Lifting state up` in React refers to the practice of `moving the state from a lower-level (child) component to a higher-level (parent or common ancestor) component in the component tree .` This is done to share and manage state across multiple components.

When a child component needs access to certain data or needs to modify the data, instead of keeping that data and the corresponding state management solely within the child component, we move the state to a shared ancestor component. By doing so, the parent component becomes the source of truth for the state, and it can pass down the necessary data and functions as props to its child components.

> 💡 Here's a simple example to illustrate lifting state up :

```
// Parent component
class ParentComponent extends React.Component {
constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  incrementCount = () => {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <ChildComponent count={this.state.count} onIncrement=
{this.incrementCount} />
      </div>
    );
  }
}

// Child component
function ChildComponent({ count, onIncrement }) {
  return (
    <div>
      <p>Child Count: {count}</p>
      <button onClick={onIncrement}>Increment</button>
    </div>
  );
}
```

In this example, the ParentComponent holds the state (count), and it passes both the state value (count) and a function (onIncrement) down to the ChildComponent as props. The child component can then display the count and trigger an increment when the button is clicked.

By lifting the state up to a common ancestor, you centralize the state management, making it easier to control and share state among components. This pattern is especially useful in larger React applications where multiple components need access to the same data or where the state needs to be synchronized across different parts of the application.

Context Provider and Context Consumer

> In React, the `Context API` provides `a way to pass data through the component tree without having to pass props manually at every level.` The two main components associated with the `Context API are the Context Provider and Context Consumer .`

> Note : We can create a context as many as we want.

1. For creating a context we use `createContext()` :

```
const UserContext = createContext()

// we pass the default value here.
// Always start with the capital letter.
```

2. For using the context we use `useContext()` :

```
const data = useContext(UserContext)
```

1. Should we keep all the data inside context?

- Only the data which you are using at multiple places or you feel that it can be used in multiple places , that is where you will use your context.

`Context Provider:` The Context Provider is a `component that allows its children to subscribe to a context's changes .` It accepts a value prop, which is the data that will be shared with the components that are descendants of this provider. The Provider component is created using `React.createContext()` and then rendered as part of the component tree. It establishes the context and provides the data to its descendants.

> 💡 Here's an example:

```
// Creating a context
const MyContext = React.createContext();

// Parent component serving as the provider
class MyProvider extends React.Component {
  state = {
    data: "Hello from Context!",
  };

  render() {
    return (
      <MyContext.Provider value={this.state.data}>
        {this.props.children}
      </MyContext.Provider>
    );
  }
}
```

`Context Consumer:` The Context Consumer is a component that subscribes to the changes in the context provided by its nearest Context Provider ancestor. It allows components to access the context data without the need for prop drilling. The Consumer component is used within the JSX of a component to consume the context data. It takes a function as its child, and that function receives the current context value as an argument.

> We use `consumer` in class component.

Here's an example:

```
// Child component consuming the context
class MyConsumerComponent extends React.Component {
  render() {
    return (
      <MyContext.Consumer>
        {(contextData) => (
          <p>{contextData}</p>
        )}
      </MyContext.Consumer>
    );
  }
}
```

By using the Context Provider and Context Consumer, you can avoid prop drilling and make it easier to share global or shared state across different parts of your React application. This is particularly useful when passing data to deeply nested components without explicitly passing the data through each intermediate component.

1. **If we don't pass a value to the provider does it take the default value ?** Yes, If we don't pass a value to the Provider in React's Context API, `it does use the default value specified when creating the context using React.createContext(defaultValue)`.

   > 💡 Here's the corrected explanation:

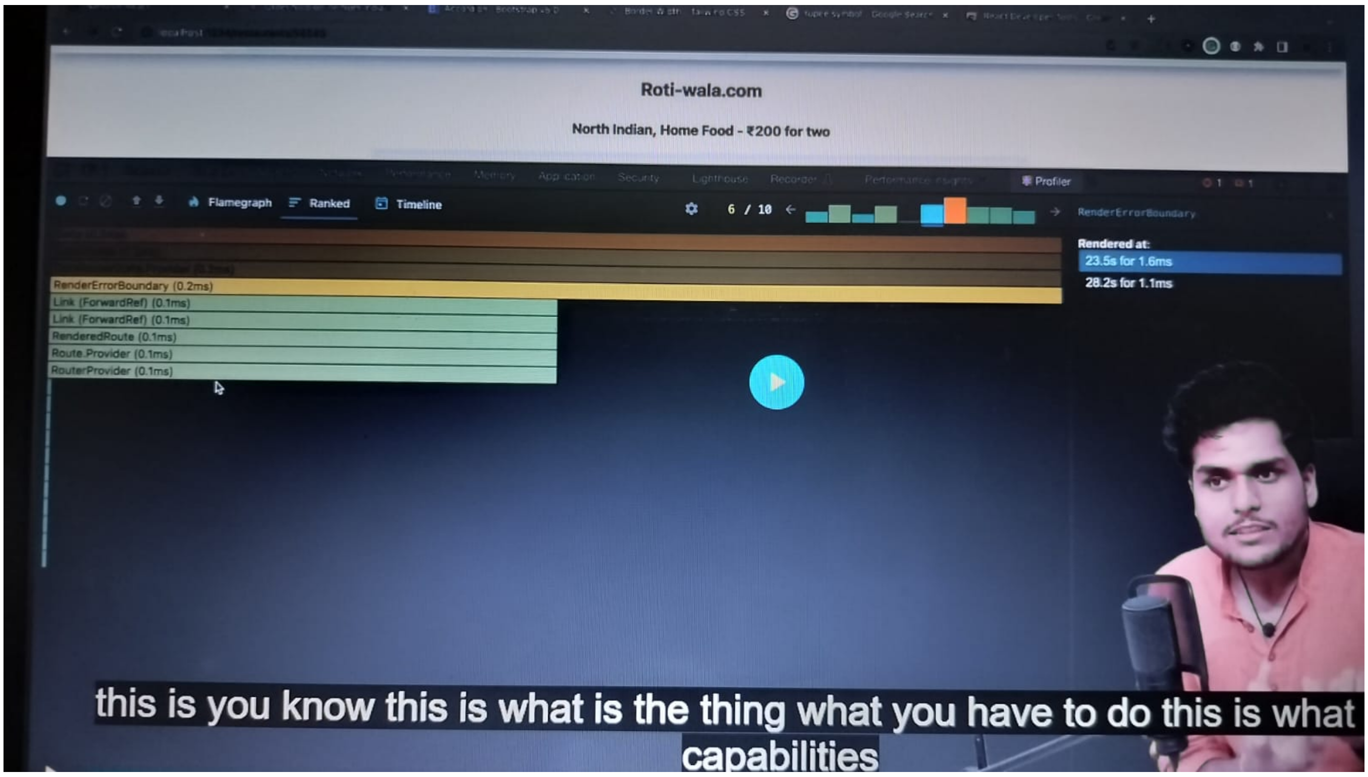
   ```
   // Creating a context with a default value
   const MyContext = React.createContext("Default Value");

   // Parent component serving as the provider without providing
   a value
   class MyProvider extends React.Component {
   render() {
       return (
       <MyContext.Provider>
           {this.props.children}
       </MyContext.Provider>
       );
   }
   }
   ```

   In this example, if we don't provide a value to the `MyContext.Provider`, it will use the default value ("Default Value" in this case) specified during the creation of the context. Any component that consumes this context using `MyContext.Consumer` will receive the default value if there is no Provider higher up the tree providing a different value.

React Dev Tools

`React DevTools` is a browser extension that allows developers to inspect and debug React applications. It's available for both Chrome and Firefox and provides a panel with useful information about the component hierarchy, props, state, and more in a React application. You can use it to inspect and manipulate the state and props of your components, which can be very helpful for debugging and understanding the behavior of your React application.


Roti-wala.com
North Indian, Home Food - ₹200 for two
Profiler
Flamegraph
Ranked
Timeline
6 / 10
RenderErrorBoundary
Rendered at:
23.5s for 1.6ms
28.2s for 1.1ms
RenderErrorBoundary (0.2ms)
Link (ForwardRef) (0.1ms)
Link (ForwardRef) (0.1ms)
RenderedRoute (0.1ms)
Route.Provider (0.1ms)
RouterProvider (0.1ms)
this is you know this is what is the thing what you have to do this is what capabilities

This profiler section can be used to find out the components which takes a lot of time to render when app grows big.

Controlled vs Uncontrolled Components in ReactJS:

Controlled Components:

- Controlled components in React are components where the state is controlled by React. The component's state is stored in the React component and is passed down to the component as props.
- Changes to the state of controlled components are handled through event handlers in the React component itself.
- Controlled components provide a predictable behavior and a single source of truth for the component's state.
- Examples include form inputs where the input value is controlled by React state.

Uncontrolled Components:

- Uncontrolled components in React are components where the state is managed by the DOM (Document Object Model) directly.
- These components are not controlled by the React state and their state and behaviors are managed independently by the DOM.
- Uncontrolled components are useful for form elements like input, textarea, and select elements where direct access to the DOM is required.

- Examples include file input fields that rely on the browser for handling the input value.

Difference between Controlled and Uncontrolled Components:

Controlled Components	Uncontrolled Components
The component's state is controlled by React.	The component's state is managed directly by the DOM.
Predictable behavior as controlled by its state.	Less predictable as not controlled by a single source.
Internal state maintained by React.	Internal state is managed by the DOM.
Changes are handled via event handlers in React components.	Changes and state are managed independently by the DOM.
Controlled by the parent component or React itself.	Controlled directly by the DOM.
Suitable for better control over form data and values.	Useful when direct DOM manipulation is required.

Refer to the provided code examples for practical implementation details of controlled and uncontrolled components in React.

Additional Example:

```
// Controlled Component
import { useState } from "react";

function ControlledComponent() {
    const [value, setValue] = useState("");

    const handleChange = (e) => {
        setValue(e.target.value);
    };

    return (
        <input
            type="text"
            value={value}
            onChange={handleChange}
        />
    );
}

// Uncontrolled Component
import React, { useRef } from "react";

function UncontrolledComponent() {
    const inputRef = useRef(null);

    const handleSubmit = () => {
        console.log("Input value: ", inputRef.current.value);
    };

    return (
        <input
            type="text"
            ref={inputRef}
        />
    );
}
```