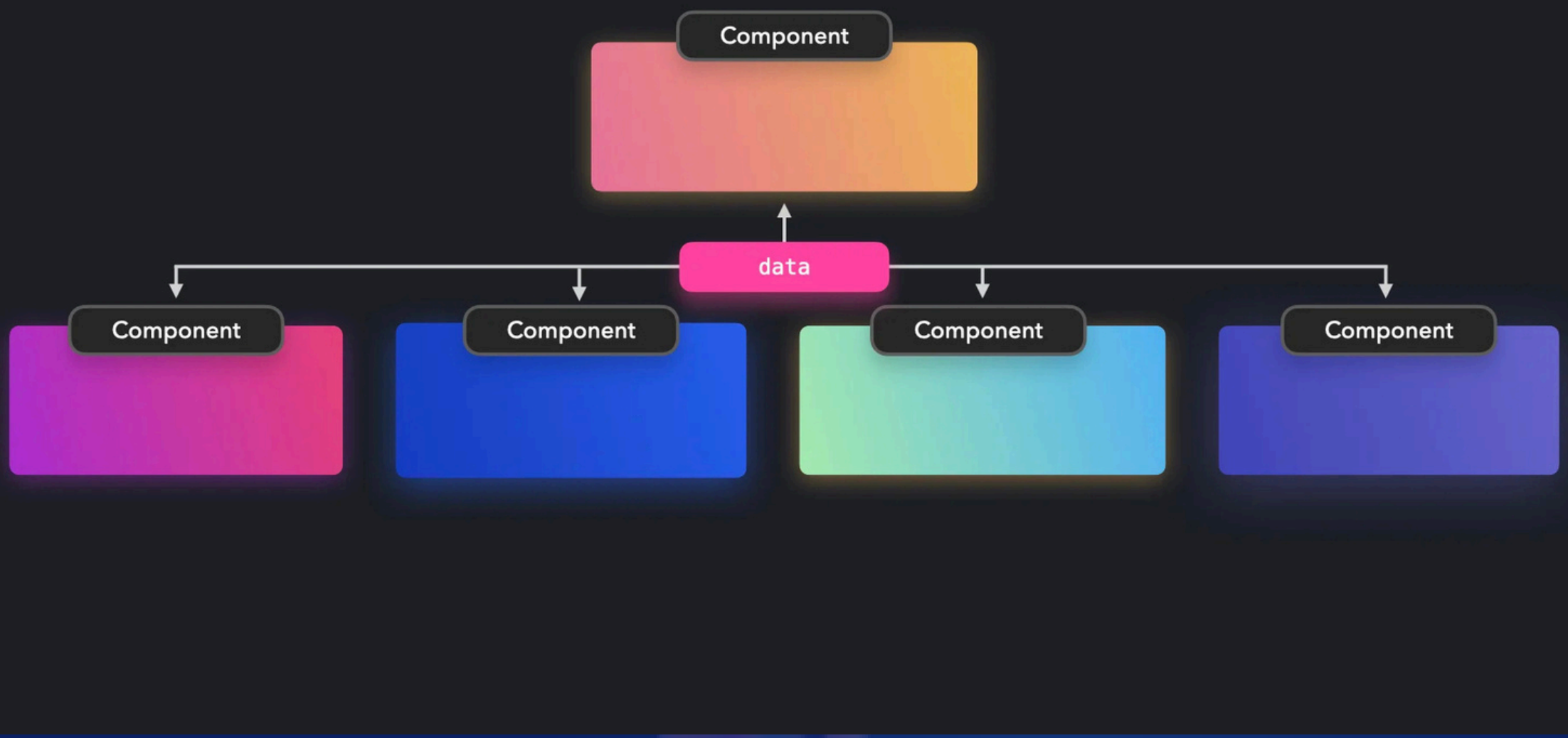
Day 30

Compound Components in React

A React Component Design Pattern



@oluwakemi Oluwadahunsi

Introduction

React offers various **patterns** for building reusable and scalable components. One such powerful pattern is the **Compound Components Pattern**.

This pattern is particularly useful when you want to create highly **flexible** and **customizable** components.

The compound components pattern allows flexibility in how components are composed. The **parent** handles state and logic, while the **child components** focus on **presentation** and **interaction**. This makes it easy to manage complex UI structures in React.

If you're looking to create more complex UI structures while maintaining simplicity, this is an essential pattern to understand.

@oluwakemi Oluwadahunsi

What Are Compound Components?

Compound components in React are components that work together to create a more flexible and cohesive UI.

The idea behind them is to **group multiple related components** that can communicate with each other, share state, and offer flexibility to the user on how to use them.

Think of compound components as a family of components that act as a **single unit**. These components allow for a more **declarative API** and make it easier to customize and configure complex behavior while maintaining simplicity.

In compound components:

- **Parent Component:** Acts as the container or the main component.
- **Child Components:** Are sub-components that share and use the parent component's state and logic.

@oluwakemi Oluwadahunsi

Why Use Compound Components?

Compound components allow for:

- **Flexibility:** You can provide a highly customizable API where developers can structure components the way they want.
- **Separation of Concerns:** You split functionality across different components, making it easier to manage and maintain each.
- **Cleaner Code:** You avoid passing a lot of props down the component tree (prop drilling) by using compound components with shared state.
- **Reusable Logic:** You can reuse and combine the components to build various UI components, thus making them more scalable.

@oluwakemi Oluwadahunsi

When to Use Compound Components?

Use compound components when:

- You need to create a reusable, customizable UI component with multiple related parts.
- You want to give the user flexibility in how they use or combine the components.
- You want to avoid prop drilling by having a parent component manage the shared state.

For example, a **Tabs** component is a classic use case. You can also use this pattern for forms, dropdown menus, accordions, modals, and more.

@oluwakemi Oluwadahunsi

How Compound Components Work

Compound components typically work by:

- **Parent-Child Relationship:** The parent component manages the shared state, and the child components consume that state.
- **Shared State:** The parent manages the state, and children access this state through a context API, props, or by accessing the parent's functions directly.
- **Context API:** The React Context API is often used to share state and logic between the components without having to pass props manually to every child component.

Example: Building Compound Components

Let’s build a **Tabs** component as an example of compound components. The component will have:

- A **Tabs** parent component to hold the state.
- **TabList**, **Tab**, and **TabPanel** as children components to display the tabs and their content.

Step 1: Creating the Parent Component (Tabs)

```jsx
import React, { useState, createContext, useContext } from "react";

// Create a Context to manage state
const TabsContext = createContext();

const Tabs = ({ children }) => {
  const [activeIndex, setActiveIndex] = useState(0);

  // Function to change active tab
  const selectTabIndex = (index) => setActiveIndex(index);

  return (
    <TabsContext.Provider value={{ activeIndex, selectTabIndex }}>
      <div className="tabs">{children}</div>
    </TabsContext.Provider>
  );
};

export default Tabs;
```

In this parent **Tabs** component:

- **State Management:** The **Tabs** component manages the state for which tab is currently active. It uses the **useState** hook to keep track of the active tab with **activeIndex**, which initially starts at **0** (the first tab).

- **Context API:** The **TabsContext** is created using React's **createContext()** function. This context is used to pass down the **activeIndex** (the current active tab) and the **selectTabIndex** function (which allows changing the active tab) to the child components without needing to pass them manually as **props**.

- **Providing Context:** Inside the **Tabs** component, a **TabsContext.Provider** is used to wrap the child components (children). The value prop in TabsContext.Provider contains both the current active index and the function to change the active tab.

@oluwakemi Oluwadahunsi

Step 2: Creating the TabList and Tab Components

Now, we will create the child components that will work inside **Tabs**.

```jsx
const TabList = ({ children }) => {
  return <div className="tab-list">{children}</div>;
};

const Tab = ({ index, children }) => {
  const { activeIndex, selectTabIndex } = useContext(TabsContext);

  const isActive = activeIndex === index;

  return (
    <button
      className={`tab ${isActive ? "active" : ""}`}
      onClick={() => selectTabIndex(index)}
    >
      {children}
    </button>
  );
};

export { TabList, Tab };
```

TabList Component: This is a simple container component that holds the **Tab** components. It accepts **children** as a prop, which can be one or more **Tab** components. It's styled as a list of tabs.

In the **Tab** Component:

- The **Tab** component uses useContext(TabsContext) to access the shared state (**activeIndex**) and the method to change the active tab (**selectTabIndex**).
- The **isActive** constant checks if the current Tab's index matches the **activeIndex**. If they match, it means this Tab is the active one.
- The **Tab** component renders a button. The **className** includes a conditional "**active**" class if the tab is currently selected. When the button is clicked, it triggers **selectTabIndex**, passing the current index, which changes the active tab.

@oluwakemi Oluwadahunsi

Step 3: Creating the TabPanel Component

```jsx
const TabPanel = ({ index, children }) => {
  const { activeIndex } = useContext(TabsContext);

  return activeIndex === index ? <div className="tab-panel">{children}</div> :
null;
};

export { TabPanel };
```

- Like the **Tab** component, the **TabPanel** also consumes the **TabsContext** using **useContext**. It needs to know the current active tab (**activeIndex**).
- The **TabPanel** checks if its index matches the **activeIndex**. If they match, it renders the panel's content ({children}). If not, it renders **null**, meaning nothing is displayed for that panel if it's not active.

@oluwakemi Oluwadahunsi

Step 4: Using the Compound Components Together

```jsx
import React from "react";
import Tabs, { TabList, Tab, TabPanel } from "./Tabs";

function App() {
  return (
    <Tabs>
      <TabList>
        <Tab index={0}>Tab 1</Tab>
        <Tab index={1}>Tab 2</Tab>
      </TabList>

      <TabPanel index={0}>Content for Tab 1</TabPanel>
      <TabPanel index={1}>Content for Tab 2</TabPanel>
    </Tabs>
  );
}

export default App;
```

@oluwakemi Oluwadahunsi

- **Tabs Component:** Acts as the parent component and manages the **shared state** for the active tab. It provides the context for its children (**TabList**, **Tab**, and **TabPanel**).

- **TabList** and **Tab**: Inside **TabList**, two **Tab** components are defined with different index values (0 and 1). These are the **clickable tabs.** When clicked, they change the active tab.

- **TabPanel:** Corresponds to the content for each tab. When a tab is selected, the corresponding **TabPanel** displays its content. If **Tab 1 is active**, the content for Tab 1 ("Content for Tab 1") is shown.

A perfect example is the Tab component from the Headless Ui library.

@oluwakemi Oluwadahunsi

Real-World Use Cases

- Accordion: Different sections expand and collapse.
- Dropdown: A menu that can expand or collapse, with different sections.
- Wizard Forms: A form that has multiple steps where each step corresponds to a different panel.
- Modals: A complex modal window that has different sections or tabs.

@oluwakemi Oluwadahunsi

Best Practices for Compound Components

- **Use Context API:** This helps you avoid prop drilling and keeps the state within the component tree without unnecessary prop passing.

- **Keep Child Components Stateless:** The parent should manage all the state and logic. The children only consume the state and call the parent's methods.

@oluwakemi Oluwadahunsi

Congratulations,

We Made It!!!

I am more than excited for this journey, how we started and how we are going (Learning is continous right).

If these 30 days has been helpful to you, kindly leave your thoughs in the comments, I would love to read them (for encouragement and improvement).

Once again, thank you for journeying with me! I love love love y'all 💯

@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