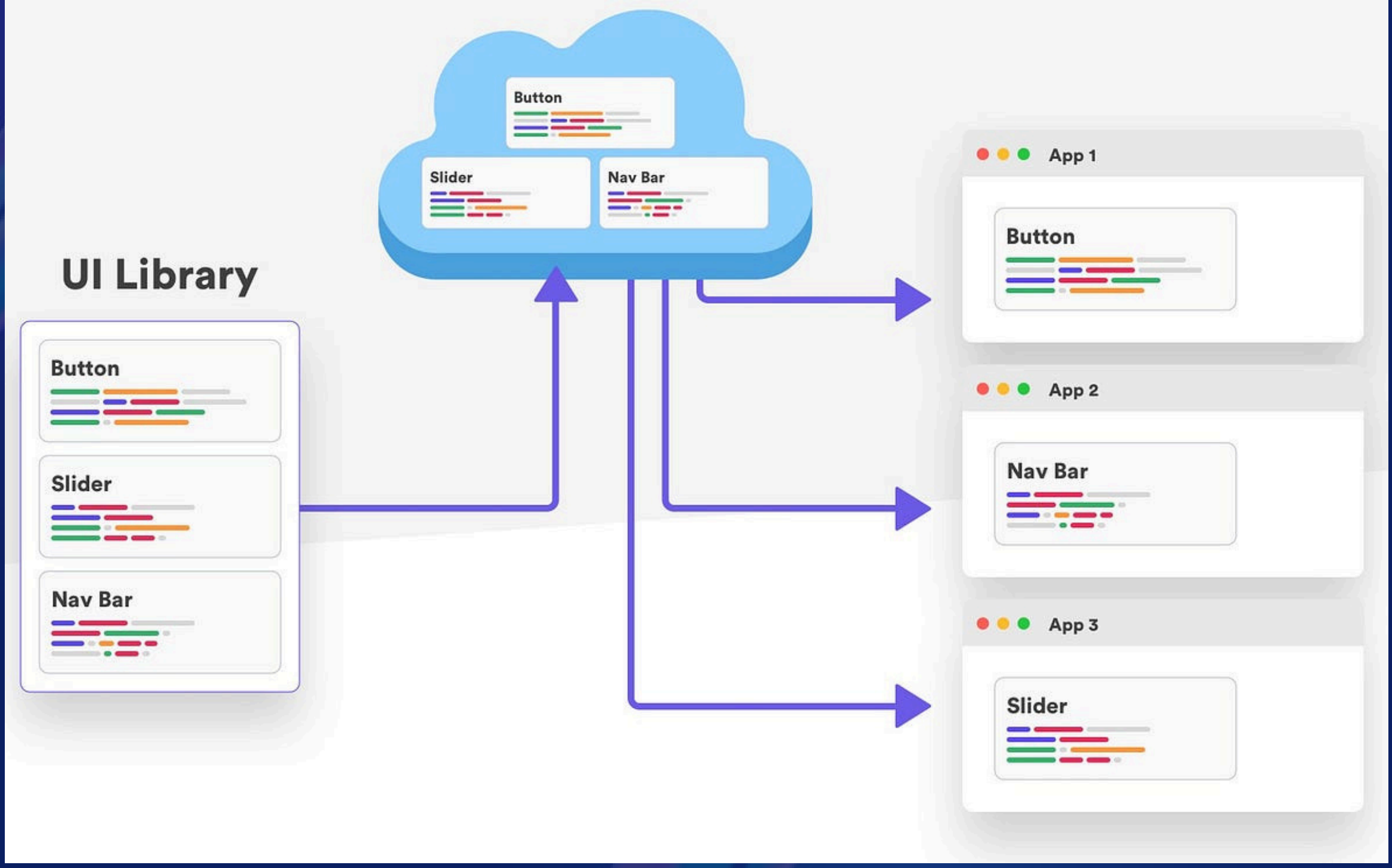
Day 14

Component Composition and Reusability in React



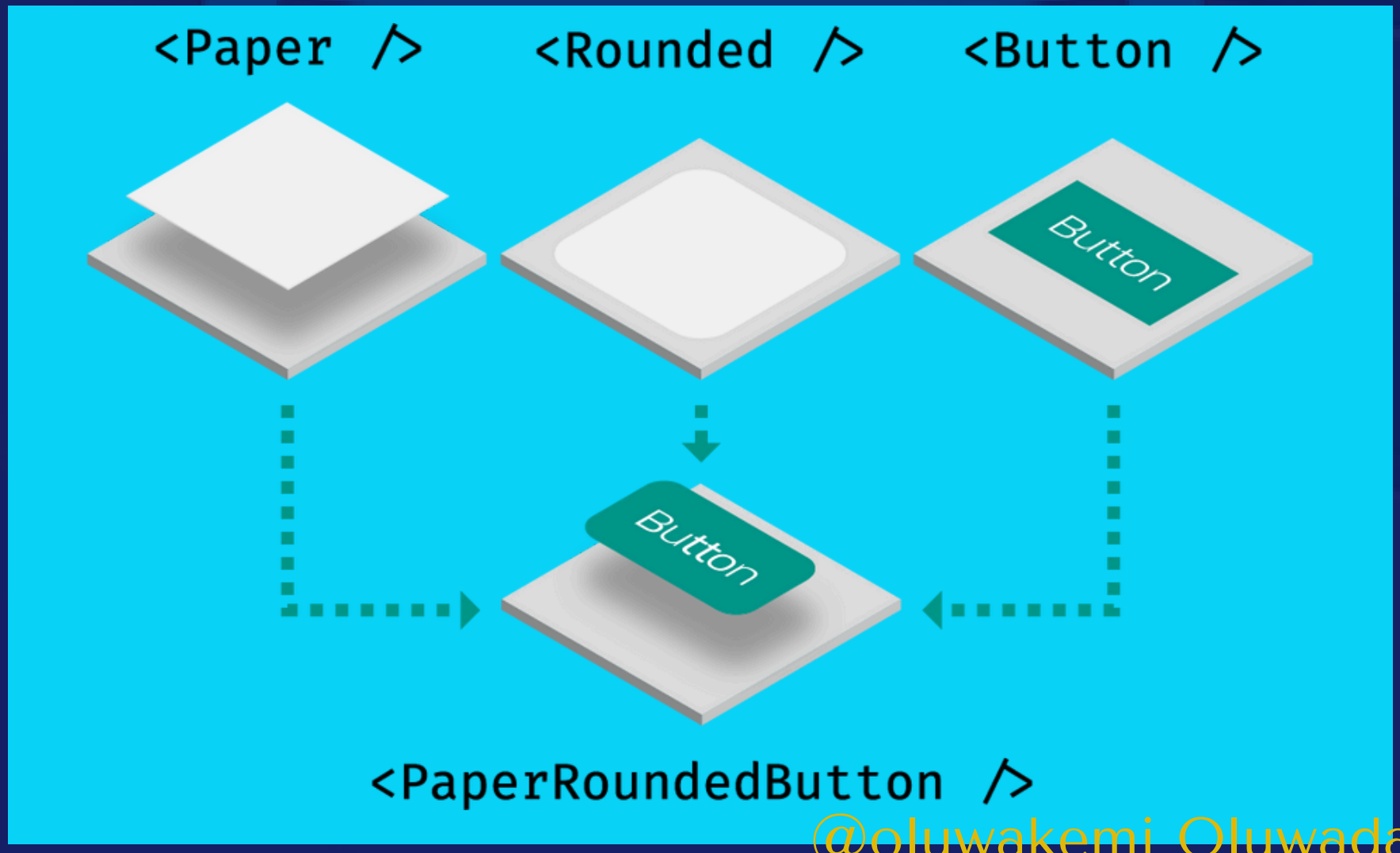
@oluwakemi Oluwadahunsi

Introduction

In React, component **composition** and **reusability** are key concepts that allow developers to build efficient, scalable, and maintainable applications.

These principles are at the core of how React applications are structured, enabling the creation of **small**, **independent** components that can be **combined** and **reused** throughout the application.

These principles help build applications that are modular, maintainable, and scalable, ultimately enhancing both development speed and the user experience.



Component Composition in React

Component composition is the process of **combining multiple components** to build a larger or more complex user interface.

Think of composition like building with **LEGO blocks**: each small block serves a specific function, and when combined, they create a more complex structure.

It follows the "**composition over inheritance**" principle, which means that instead of creating large, monolithic components or deeply inheriting features from a parent component, you build UI elements by **composing smaller, self-contained components together**.

React allows components to be composed in a tree structure. A **parent component** can include **multiple child components**, passing **props (data)** down to them. This tree structure makes it easy to create complex UIs by nesting components.

@oluwakemi Oluwadahunsi

Example of Composition:

Let’s say we are building a profile card that displays a user’s name, avatar, and description. We can break it down into smaller components:

- **Avatar Component** – Renders the user’s image.
- **Name Component** – Displays the user’s name.
- **Description Component** – Shows the user’s bio or any additional information.

Avatar Component:

```
function Avatar({ src, alt }) {
  return <img src={src} alt={alt} />;
}
```

Name Component:

```
function Name({ firstName, lastName }) {
  return <h2>{firstName} {lastName}</h2>;
}
```

Description Component:

```jsx
function Description({ text }) {
  return <p>{text}</p>;
}
```

ProfileCard component serves as the **parent component** that wraps all other components (children) to form a complete profile:

```jsx
import Avatar from './Avatar';
import Name from './Name';
import Description from './Description';

function ProfileCard({ user }) {
  return (
    <div className="profile-card">
      <Avatar src={user.avatarUrl} alt={user.name} />
      <Name firstName={user.firstName} lastName={user.lastName} />
      <Description text={user.bio} />
    </div>
  );
}
export default ProfileCard;
```

@oluwakemi Oluwadahunsi

In this example,

- The **ProfileCard** component is composed of three smaller components: **Avatar**, **Name**, and **Description**.

- Each component is focused on one responsibility, making the code more readable and maintainable.

- Composition also allows each of these components to be reused in different parts of the app, if needed.

Benefits of Composition:

- **Improved Readability:** Breaking down components into smaller, focused units makes the code more understandable.

- **Separation of Concerns:** Each component handles one specific task, which promotes better organization and easier debugging.

- **Scalability:** As applications grow in size and complexity, component composition allows developers to add or replace components without affecting the overall architecture.

- **Reusability:** Components can be reused across different parts of the application or even across projects.

@oluwakemi Oluwadahunsi

Reusability in React

Reusability refers to the practice of designing components that can be used in **multiple places** across your application with **little or no modification**.

Instead of writing the same code multiple times, you create flexible components that can adapt to different situations by accepting **props**.

By making components reusable, you eliminate **code duplication**, which reduces potential errors and makes the codebase easier to manage.

How to Make Components Reusable

For a component to be reusable, it should:

- **Be Independent:** Handle one specific task and not rely on the internal details of other components.

@oluwakemi Oluwadahunsi

- **Receive data via props:** Make the component flexible by using props to pass data and functions.
- **Avoid side effects:** A reusable component shouldn’t alter external state or perform unexpected side effects.
- **Be generic:** Design the component so that it doesn’t hardcode values, allowing you to reuse it in different contexts.

Example of Reusability: Using a “**Button**” component

```jsx
function Button({ label, onClick, type = 'button', style }) {
  return (
    <button type={type} onClick={onClick} style={style}>
      {label}
    </button>
  );
}

export default Button;
```

Reusable props

@oluwakemi Oluwadahunsi

This simple **Button** component can be used in different parts of the app by passing in different **labels**, **click handlers**, and **styles**.

Here’s how you might reuse it in different places:

```jsx
import Button from "./Button"

function ButtonUsage() {
  return (
  <Button label="Submit" onClick={() ⇒ console.log('Saved!')}>Save</>
  <Button label="Cancel" onClick={handleCancel} style={{ backgroundColor: 'red' }}>
    Cancel
  </>
  );
}

export default ButtonUsage;
```

In this example:

- The **Button** component accepts label, onClick, and stylet as props.

- It is reused multiple times with different labels and functionality, making the component dynamic and flexible.

@oluwakemi Oluwadahunsi

Benefits of Reusability:

- **Less Code Duplication:** You avoid writing the same component logic multiple times, which reduces the risk of introducing bugs.

- **Consistency:** Using the same component ensures that the look and behavior are consistent across the application.

- **Easier Maintenance:** When a reusable component needs to be updated or fixed, the changes are made in one place, and all instances are updated automatically.

- **Faster Development:** By creating reusable components, you can speed up the development process by leveraging already-built components in new features or sections of your app.

@oluwakemi Oluwadahunsi

Key Techniques for Composition and Reusability

1. Props for Dynamic Behavior

Props allow a reusable component to be customized with different data and behavior. The component's logic stays the same, but its output changes based on the props passed to it.

```
function Greeting({ name }) {
  return <p>Hello, {name}!</p>;
}

// Usage Example
<Greeting name="Alice" />
<Greeting name="Bob" />
```

By passing different name values, the Greeting component remains reusable and flexible.

@oluwakemi Oluwadahunsi

2. Children for Composability

In React, components can pass other components as children. This pattern allows more flexible and powerful composition.

```
const Card = ({ title, children }) => {
  return (
    <div className="card">
      <h2>{title}</h2>
      <div className="content">{children}</div>
    </div>
  );
};

// Using the Card component in the app:
const App = () => {
  return (
    <div>
      <Card title="Card 1">
        <p>This is the content of Card 1.</p>
      </Card>

      <Card title="Card 2">
        <ul>
          <li>Item 1</li>
          <li>Item 2</li>
        </ul>
      </Card>
    </div>
  );
};
```

In this example:

- The **Card** component can accept any children (**text, lists, other components**).
- It is reused with different content while keeping a consistent structure (title and content).

Higher Order Components (HOC)

HOC is a **function** that takes a **component** as an **argument** and **returns a new component** that enhances or modifies the behavior of the original component.

HOCs don’t modify the component directly but instead create a **wrapper** around it, injecting additional props, handling state, or connecting it to external data.

The HOC pattern works in a way that is similar to functions that accept parameters and return a new function with added capabilities.

The main purpose is to **abstract** logic away from components so that you can reuse it across different parts of your application **without duplicating code**.

@oluwakemi Oluwadahunsi

Simple Example: **Logging Component Updates**

Let’s say we want to create a higher-order component that logs whenever a component updates.

This HOC can be applied to any component to add this logging behavior without modifying the original component’s implementation.

```
import { useEffect } from 'react';

const withLogging = (WrappedComponent) => {
  return (props) => {
    // Log when the component is mounted or updated
    useEffect(() => {
      console.log(`Component ${WrappedComponent.name} rendered with props:`, props);
    });

    // Return the original wrapped component with all of its original props
    return <WrappedComponent {...props} />;
  };
};

export default withLogging;
```

In the withLogging component,

- **withLogging** is the **HOC** that takes in a **WrappedComponent** and returns a **new functional component** that logs to the console whenever the component renders.
- Inside **useEffect**, we log the component's name and the current props passed to it. This is useful for debugging or monitoring when components re-render.

@oluwakemi Oluwadahunsi

Render Props in React

The **Render Props pattern** in React allows components to **share reusable logic**. It involves passing a function (known as a "render prop") to a component, which uses it to determine what to render.

This pattern provides a more flexible way to share logic between components **without relying on Higher-Order Components (HOCs)** or tightly coupling functionality to specific components.

How Does Render Props Work?

A render prop is essentially a function prop that a component uses to know what content or JSX to display. This function gives control to the consuming component over how the data or state inside the component is rendered.

@oluwakemi Oluwadahunsi

Example: **Toggle Component Using Render Props**

Suppose you want to create a reusable **Toggle** component that provides functionality for toggling between "**on**" and "**off**." Instead of hardcoding the UI inside the Toggle component, you can use the **render props pattern** to allow the consuming component to define what should be displayed based on the **toggle state**.

1. Create the Toggle Component (Using Render Props):

```jsx
import { useState } from 'react';

const Toggle = ({ render }) => {
  // Local state to keep track of the toggle status
  const [isOn, setIsOn] = useState(false);

  // Function to toggle the state between true and false
  const toggle = () => setIsOn(prevState => !prevState);

  // Call the render prop and pass the toggle status and toggle function
  return render({ isOn, toggle });
};

export default Toggle;
```

@oluwakemi Oluwadahunsi

In the example above:

- **State:** The component maintains its own state (isOn) to track whether the toggle is on or off.

- **Toggle Function:** toggle is a function that toggles the isOn state.

- **Render Prop:** Instead of rendering UI directly, the **Toggle** component takes a **render prop**, which is a function provided by the parent component. This function is responsible for defining how the UI should look based on the **isOn** state.

2. Using the Toggle Component with Different UIs:

Now, let's use this Toggle component in different ways to see how flexible it can be.

```jsx
import Toggle from './Toggle';

const App = () => {
  return (
    <div>
      <h1>Toggle Example with Render Props</h1>

      <Toggle
        render={({ isOn, toggle }) => (
          <div>
            <button onClick={toggle}>
              {isOn ? 'Switch Off' : 'Switch On'}
            </button>
            <p>The switch is {isOn ? 'ON' : 'OFF'}</p>
          </div>
        )}
      />
    </div>
  );
};

export default App;
```

- **Render Function:** The render prop receives an object containing isOn (the toggle state) and toggle (the function to change the state).

- **UI:** Inside the render function, a button and a paragraph are rendered. The **button text** and **paragraph** change dynamically based on the toggle state (**isOn**).

@oluwakemi Oluwadahunsi

Best Practices For Using List and Keys In React

- **Keep Components Small and Focused:** Each component should do one thing and do it well. Small, focused components are easier to reuse and compose.

- **Keep Components Small and Focused:** Each component should do one thing and do it well. Small, focused components are easier to reuse and compose.

- **Leverage Composition Over Inheritance:** React favors composition over inheritance. When building reusable components, think in terms of combining components instead of extending them.

- **Abstract Logic Only When Needed:** Don’t over-abstract components too early. Start with simple, reusable components and refactor as the need arises.

@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