DESIGN PATTERNS

IN REACT JS

THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

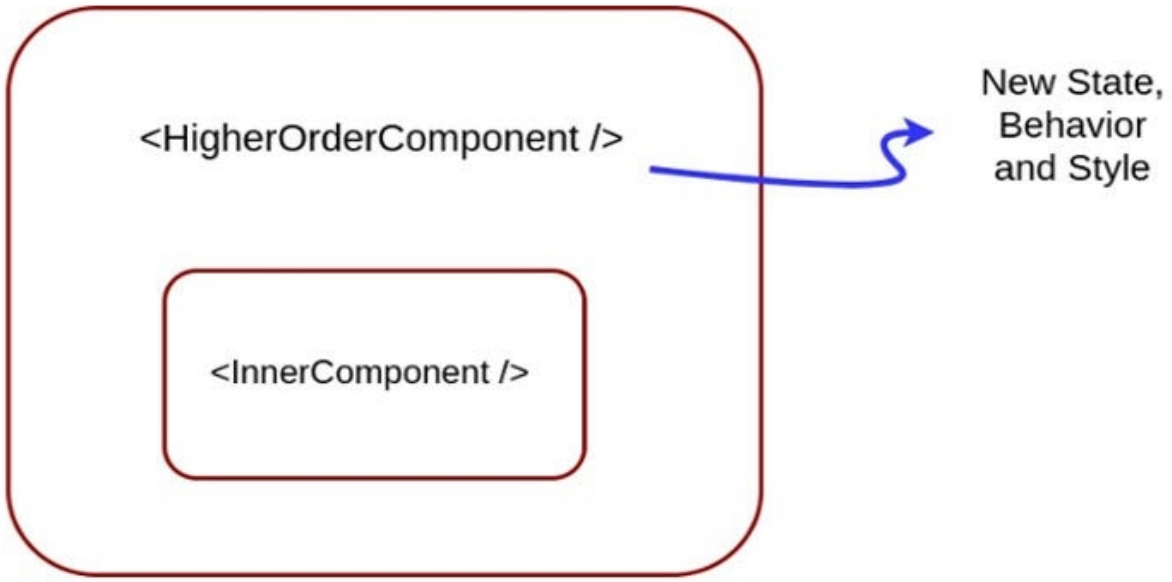
https://arihara-sudhan.github.io

DESIGN PATTERNS

A Blog says, “according to a study conducted by Stack Overflow, the cost of dealing with bad code is as high as $85 billion annually”. Besides, it also leads us to issues like technical debt in software development, less scalability, and excess use of resources. Design Patterns not only controls your expenditure but also helps you to reform your application. Let’s discuss the significance of React design patterns in this book. Design Patterns are the pre-built templates. It ensures proficient code organization and reusability. It smoothens the development, extension. It provides way for uniformity.

✧ HIGHER ORDER COMPONENT

A Higher Order Component (HOC) in React is a pattern where a function takes a component and returns a new component with any changes applied on it.



It is a technique in React for reusing component logic. HOCs are not part of the React API, itself.

An example, where we set background color using a Higher Order Component.

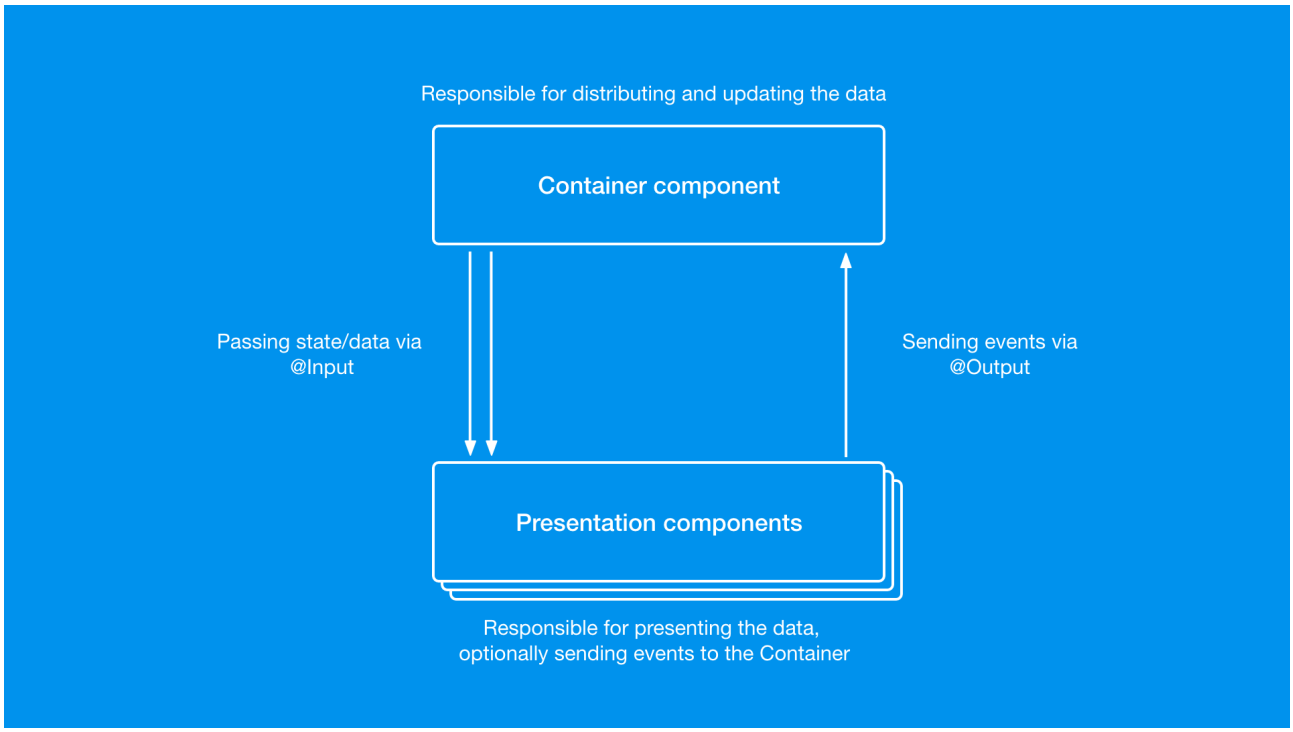
HIGHER ORDER COMPONENT

```jsx
import React from 'react';

// Higher Order Component
const withBackgroundColor = (WrappedComponent, bgColor) => {
  return (props) => {
    const style = {
      backgroundColor: bgColor,
      padding: '20px',
      borderRadius: '5px',
    };

    return (
      <div style={style}>
        <WrappedComponent {...props} />
      </div>
    );
  };
};

// Regular component
const MyComponent = () => {
  return <h1>Hello, World!</h1>;
};

// Using the HOC to wrap MyComponent with a specific background color
const MyComponentWithBackground = withBackgroundColor(MyComponent, 'lightblue');

const App = () => {
  return (
    <div>
      <MyComponentWithBackground />
    </div>
  );
};

export default App;
```

✧ PRESENTATIONAL & CONTAINER

The presentational and container components represent a design pattern approach that helps separate concerns and responsibilities in React applications. Presentational components are focused on how a user interface should look like. Presentational components receive data and callbacks through props. On the other hand, container components handle tasks like fetching data from servers, managing changes in data, and state management in React applications.

Container components ensure that things are working well in the backend. In addition, a clear separation between presentational and container components can help to identify and rectify errors, which helps to reduce the application's downtime.



A Miserable fact is that the presentation components are called Dumb Components and the container components are called Smart Components. Here is an example.

PRESENTATIONAL AND CONTAINER COMPONENTS

```
const useUsers = () => {
  const [users, setUsers] = React.useState([]);

  React.useEffect(() => {
    // Simulate fetching data
    const fetchedUsers = [
      { id: 1, name: 'Alice' },
      { id: 2, name: 'Bob' },
    ];
    setUsers(fetchedUsers);
  }, []);

  return users;
};

const UserListContainer = () => {
  const users = useUsers();
  const [selectedUserId, setSelectedUserId] = React.useState(null);
  const handleSelectUser = (userId) => {
    setSelectedUserId(userId);
  };

  return (
    <UserList
      users={users}
      onSelectUser={handleSelectUser}
    />
  );
};
```

✧ PROVIDER

The Provider Pattern in React is a way to manage and share state or services across your entire component tree without passing props down manually at every level which is called, Props Drilling. It is commonly used in conjunction with the Context API. The Provider Pattern allows components to subscribe to the context values and use those values anywhere in the component tree, making it easier to manage global state, themes, or configurations.

PROVIDER >>>> CONSUMER

Let’s understand this with a simple Theme provider example. Following is the Theme Provider.

PROVIDER

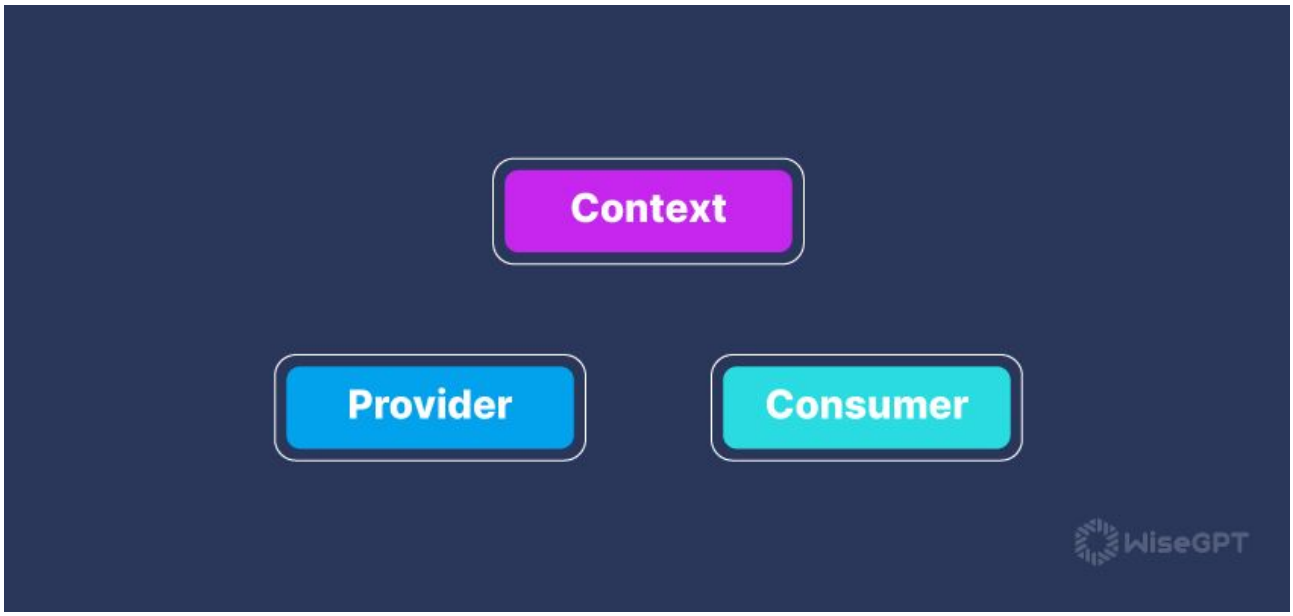
```
import React, { createContext, useState, useContext } from 'react';

// Create a Context for the theme
const ThemeContext = createContext();

// Create a provider component
const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

export { ThemeProvider, ThemeContext };
```

This should be consumed by a consumer.

Let’s consume and say yummy.

CONSUMER OF THE PROVIDER

```
import React, { useContext } from 'react';
import { ThemeContext } from './ThemeProvider'; // Import the context

const ThemedComponent = () => {
  const { theme, toggleTheme } = useContext(ThemeContext); // Consume the context

  return (
    <div style={{
        backgroundColor: theme === 'light' ? '#fff' : '#333',
        color: theme === 'light' ? '#000' : '#fff',
        padding: '20px',
        textAlign: 'center'
      }}>
      <h1>{theme === 'light' ? 'Light Mode' : 'Dark Mode'}</h1>
      <button onClick={toggleTheme}>
        Switch to {theme === 'light' ? 'Dark' : 'Light'} Mode
      </button>
    </div>
  );
};

export default ThemedComponent;
```



✧ ERROR BOUNDARY

In React, an Error Boundary is a special component that catches JavaScript errors anywhere in its child component tree, logs those errors, and displays a Fallback UI instead of crashing the whole app.

ERROR BOUNDARY

```
import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.error("Error caught in Error Boundary:", error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}

export default ErrorBoundary;
```

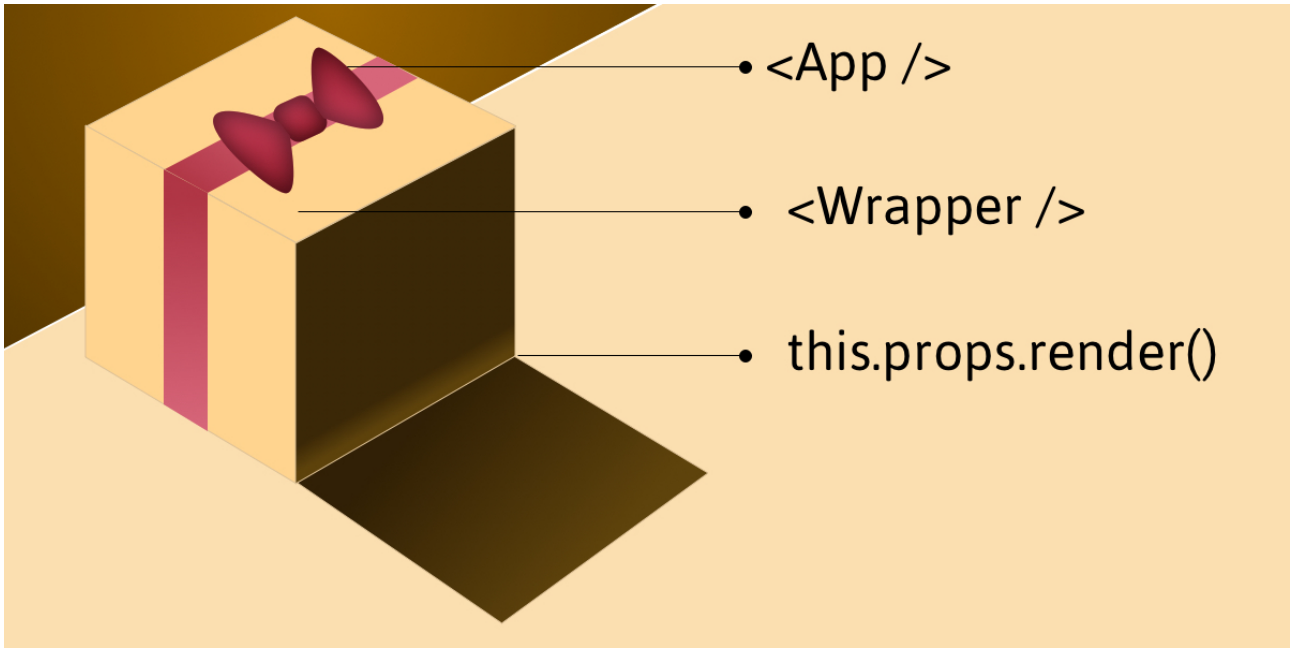
Error Boundaries are particularly useful for handling runtime errors that might occur in rendering, lifecycle methods, or within child components.

ERROR BOUNDARY : USAGE

```
import ErrorBoundary from "ErrorBoundary";

function BuggyComponent() {
  return (
    <>
      <h1>HELLO!</h1>
    </>
  );
}

function App() {
  return (
    <div>
      <ErrorBoundary>
        <BuggyComponent/>
      </ErrorBoundary>
    </div>
  );
}

export default App;
```

✧ RENDER PROPS

By employing a function prop, the so called Render-Props paradigm enables code sharing between components. In this pattern, one component gives another component a render prop, and the receiving component utilizes the prop to render its own content. By using this pattern, we can write more reused code and share logic between components.



```jsx
import React from "react";

const RenderPropsComponent = (props) => {
  return props.render();
}

export default function App(){
  return (
    <>
      <RenderPropsComponent render={()=>{
        return <h1>HELLO! HOW ARE YOU?</h1>
      }}/>
      <RenderPropsComponent render={()=>{
        return <h1>HELLO! WHAT ARE YOU DOING?</h1>
      }}/>
      <RenderPropsComponent render={()=>{
        return <h1>HELLO! WHO ARE YOU?</h1>
      }}/>
    </>
  )
}
```

This is the so-called, Render Props pattern where we share the logic from App component into RenderPropsComponent.

We can take one example to demonstrate where this pattern can be helpful.

A problem with this approach is that the components such as Kelvin, Fahrenheit doesn't have

access to the data in Input. One way tom handle this is to lift the states up! Yep! We can put the states in App Component.

```
import React, { useState } from "react";

function Input({ value, handleChange }) {
  return <input value={value} onChange={(e) => handleChange(e.target.value)} />;
}

export default function App() {
  const [value, setValue] = useState("");

  return (
    <div className="App">
      <h1>Temperature Converter</h1>
      <Input value={value} handleChange={setValue} />
      <Kelvin value={value} />
      <Fahrenheit value={value} />
    </div>
  );
}

function Kelvin({ value = 0 }) {
  return <div className="temp">{value + 273.15}K</div>;
}

function Fahrenheit({ value = 0 }) {
  return <div className="temp">{(value * 9) / 5 + 32}°F</div>;
}
```

Although this is a valid solution, it can be tricky to lift state in larger applications with components that handle many

children. Each state change could cause a re-render of all the children, even the ones that don’t handle the data, which could negatively affect the performance of your app.

So, the only solution we posed is, now.... not okay! So, what can we do instead to achieve the state globalization?

Render Props for our rescue!

RENDER PROPS

```
import React, { useState } from "react";

function Input(props) {
  const [value, setValue] = useState(0);

  return (
    <>
      <input
        type="number"
        value={value}
        onChange={e => setValue(e.target.value)}
        placeholder="Temp in °C"
      />
      {props.children(value)}
    </>
  );
}

export default function App() {
  return (
    <div className="App">
      <h1>Temperature Converter</h1>
      <Input>
        {value => (
          <>
            <Kelvin value={value} />
            <Fahrenheit value={value} />
          </>
        )}
      </Input>
    </div>
  );
}

function Kelvin({ value }) {
  return <div className="temp">{parseInt(value || 0) + 273.15}K</div>;
}

function Fahrenheit({ value }) {
  return <div className="temp">{(parseInt(value || 0) * 9) / 5 + 32}°F</div>;
}
```

Now, The intuition...

MERCI