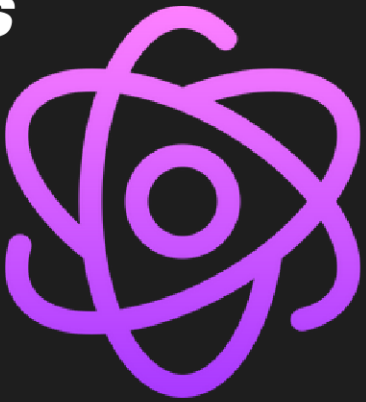
React With *Design Patterns*



React design patterns help developers solve common problems when building components by simplifying state management, logic, and element composition. Common patterns like custom hooks, higher-order components (HOCs), and prop-based rendering improve code consistency, reusability, and scalability. Using these patterns also makes the code easier to read, maintain, and collaborate on.

Why React Design Patterns Matter ?

Simple answer is YES!. It's because react design patterns save time, reduce costs, and make code easier to maintain by providing a clear, consistent structure. They help developers avoid issues and support long-term, sustainable development.

HOC Pattern

HOC (Higher Order Component) is a composition pattern for reusing any logic between components. It takes a component and returns a new component with upgraded features.

```jsx
// HOC Component
function withLogging(WrappedComponent) {
  return function WrappedWithLogging(props) {
    console.log("Component is rendering");
    return <WrappedComponent {...props} />;
  };
}

// Component using HOC
function MyComponent() {
  return <div>Hello, this is MyComponent!</div>;
}

export default withLogging(MyComponent);
```

Explanation

In this example, **“withLogging”** is the HOC that logs a message each time the component renders. And **“MyComponent”** is the component wrapped by the HOC to include the logging behavior. Basically, now **“MyComponent”** has added new functionality to log message.

When to use this pattern?

When need to share common functionality (like logging, authentication) between multiple components.
When you need to handle tasks like data fetching or handling UI permissions across various components.
When need to manage complex stateful logic outside components for better readabiity
When you need your core component logic clean by moving shared functionality outside the component.

Don’t Use When,

You don’t need shared behavior of component. Keep it simple.
It’s prefer to use hooks than HOCs. If you don’t feel good with hooks, Use HOCs.
If HOCs are gonna add unnecessary complexity to your code, especially if simpler solutions exist.

Compound Components Pattern

Compound Component Pattern allows a main component to contain multiple child components. You can choose which child components to use and control their order. This gives you flexibility and control over the component's structure and behavior.

```jsx
// Main Compound Component
const Dropdown = ({ children }) => <div className="dropdown">{children}</div>;

// Child Components
Dropdown.Toggle = ({ onClick, children }) => (
  <button onClick={onClick} className="dropdown-toggle">
    {children}
  </button>
);

Dropdown.Menu = ({ children, isOpen }) => (
  isOpen ? <div className="dropdown-menu">{children}</div> : null
);
```

```
Dropdown.Item = ({ onClick, children }) => (
  <div onClick={onClick} className="dropdown-item">
    {children}
  </div>
);

// Usage Example
function App() {
  const [isOpen, setIsOpen] = React.useState(false);

  return (
    <Dropdown>
      <Dropdown.Toggle onClick={() => setIsOpen(!isOpen)}>
        Toggle Menu
      </Dropdown.Toggle>
      <Dropdown.Menu isOpen={isOpen}>
        <Dropdown.Item onClick={() => alert('Item 1 clicked')}>Item 1</Dropdown.Item>
        <Dropdown.Item onClick={() => alert('Item 2 clicked')}>Item 2</Dropdown.Item>
      </Dropdown.Menu>
    </Dropdown>
  );
}
```

Dropdown is the main component, and **Dropdown.Toggle**, **Dropdown.Menu**, and **Dropdown.Item** are its child components. In the App component, you control the menu's visibility with **isOpen** and handle item clicks. This gives you flexible control over the dropdown's structure and behavior.

When to use this pattern?

When you need a component with flexible child elements that users can control and arrange.
When multiple child components share behavior or state.
When creating reusable, customizable UI elements like tabs, dropdowns, or forms.

Don't Use When,

Your component doesn't need flexible child components, keep it simple.
It adds unnecessary complexity to a straightforward component.
Your child components don't need to interact with shared logic or state.

Custom Hook Pattern

Hooks encapsulate reusable logic into functions. In React, hooks are JavaScript functions, with many built-in options available. You can also create custom hooks to share logic between components.

“useScrollPosition” is a custom hook here. It handles a logic to detect scroll position and we can use it anywhere in our app.

```
const ExampleComponent = () => {
  const scrollPosition = useScrollPosition();

  return (
    <div>
      <p>Current Scroll Position: {scrollPosition}</p>
      {scrollPosition > 200 && <button>Back to Top</button>}
    </div>
  );;
};

export default ExampleComponent;
```

When to use this pattern?

Use custom hooks when you need to share logic between multiple components.
To make your code more cleaner by seperating logics from UI
Manage complex stateful logic outside components for better readabiity
Encapsulate side effects, like data fetching or subscriptions, into a custom hook for reuse.

Don't Use When,

If the logic is only needed in one component.
It's a simple logic, that doesn't worth to become a custom hook.
There's no side effects or state. You can use regular functions.
If that makes your code harder to understand without real benefit.

Extensible Styles Pattern

This pattern will enable you to stylish your components in more flexible way. Here we use dynamic CSS properties, instead directly code styles in the components. So styles can be extended and modified later we want.

```
// component
function Button({ children, className = "", style = {}, ...props }) {
  const baseStyles = {
    padding: "10px 20px",
    backgroundColor: "blue",
    color: "white",
  };

  return (
    <button
      className={`btn ${className}`}
      style={{ ...baseStyles, ...style }}
      {...props}
    >
      {children}
    </button>
  );
}

// Usage
<Button className="custom-btn" style={{ backgroundColor: "green" }}>
  Click Me
</Button>;
```

Explanation

Button component has some base styles, like padding and background color, but you can customize them by passing className or style props. It merges the base styles with any new styles you provide. This lets you easily tweak the button's appearance without changing the component itself.

When to use this pattern?

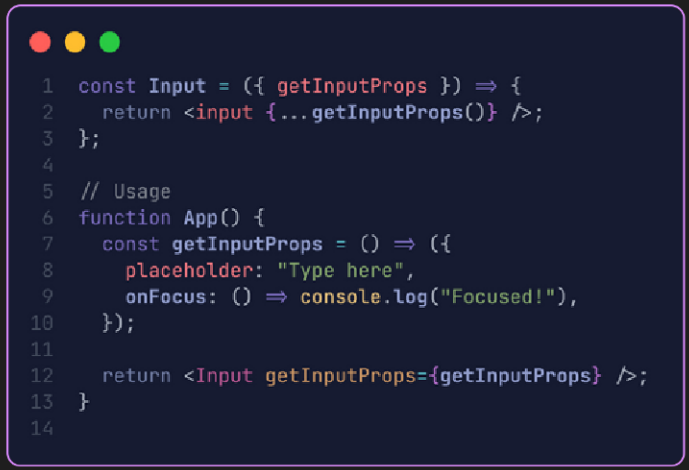
When building components that need to be styled differently across various parts of the app.
Great for reusable buttons, cards, or forms where styles may need adjustments without duplicating code.
Ideal for creating components in a design system where consumers of the system need to customize the look.

Don’t Use When,

Styles are fixed and don't need customization.
it adds unnecessary complexity to simple components, so stick with straightforward styling.

Props Getters Pattern

Props Getters Pattern lets a component provide a function to get the props needed for an element. It manages some logic while allowing the user to customize or extend the props. This makes the component flexible and reusable.



```jsx
const Input = ({ getInputProps }) => {
  return <input {...getInputProps()} />;
};

// Usage
function App() {
  const getInputProps = () => ({
    placeholder: "Type here",
    onFocus: () => console.log("Focused!"),
  });

  return <Input getInputProps={getInputProps} />;
}
```

When to use this pattern?

When you want to share logic across components while allowing flexibility in how props are applied.
When users need to modify or extend props without losing the built-in functionality.
Ideal for creating components that are customizable but still maintain default behavior.

Don’t Use When,

The component doesn’t need flexible or customized props.
A component has fixed behavior and no need for extra prop flexibility, keep it simple.

Render Props Pattern

Render Props Pattern lets a component use a function (passed as a prop) to decide what to render. This allows you to share logic between components without duplicating code. The component doesn't hardcode the output but delegates it to the passed function.

```
const DataFetcher = ({ render }) => {
  const data = "Hello from Render Props!";
  return <div>{render(data)}</div>;
};

// Usage
function App() {
  return (
    <DataFetcher render={(data) => <h1>{data}</h1>} />
  );
}
```

Explanation

In this example, **DataFetcher** is the component with a render prop. It passes the data to the function provided by the App component, which controls how that data is displayed (in this case, inside an <h1> tag).

When to use this pattern?

When you want to reuse logic between components while keeping control over the rendering.
When different components need different ways to render the same data.

Don't Use When,

If it adds unnecessary complexity to simple components.
It's preferred hooks for managing state and logic if they make the code simpler.

State Initializer Pattern

State Initializer Pattern in React is when you initialize a component's state using props or an external function during the initial render. This allows more flexible state setup, especially if the state depends on props or requires complex logic for initialization.

```
const Counter = ({ initialCount = 0 }) => {
  const [count, setCount] = React.useState(() => initialCount);
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
};

```

When to use this pattern?

When state initialization needs a function or depends on props.
When you want to avoid recalculating state on every render, using a lazy initializer function.

Don't Use When,

State can be initialized directly without complex logic, so no need for this pattern.
It adds complexity without performance or flexibility benefits.

Control Props Pattern

Control Props Pattern in React is when a component's state is controlled by its parent through props. The parent component passes down values and callbacks to control the behavior of the child component, giving the parent full control over the child's state and actions.

```jsx
const Toggle = ({ isOn, onToggle }) => {
  return <button onClick={onToggle}>{isOn ? "ON" : "OFF"}</button>;
};

// Usage
function App() {
  const [isOn, setIsOn] = React.useState(false);
  return <Toggle isOn={isOn} onToggle={() => setIsOn(!isOn)} />;
}

```

Explanation

Toggle component receives **isOn** and **onToggle** as control props from the parent. The parent (**App**) manages the state and passes it down, allowing full control over when the toggle is **ON** or **OFF**. This keeps the logic in the parent, while Toggle just displays and triggers the change.

When to use this pattern?

When the parent needs complete control over the child component's state.
When state needs to be shared across multiple components.

Don't Use When,

The component doesn't need external control.
The component can manage its own state, control props may not be needed.

State Reducer Pattern

State Reducer Pattern in React is when a component's state and its update logic are separated, often using a reducer function. This allows us for more control over how the state is managed and makes the component more flexible by providing external control over state updates.

```jsx
const Toggle = ({ stateReducer }) => {
  const [isOn, setIsOn] = React.useReducer(stateReducer, false);

  return <button onClick={() => setIsOn({ type: 'toggle' })}>{isOn ? "ON" : "OFF"}</button>;
};

// Usage with custom state reducer
function App() {
  const stateReducer = (state, action) => {
    if (action.type === 'toggle') {
      return !state;
    }
    return state;
  };

  return <Toggle stateReducer={stateReducer} />;
}
```

When to use this pattern?

When you need to allow external control over how state updates occur.
Ideal for managing components with complex state transitions.

Don't Use When,

The component's state management is straightforward and doesn't require custom logic.
It adds unnecessary complexity to otherwise simple components.