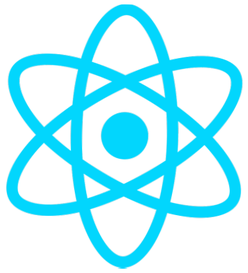


COMMON REACT DESIGN PATTERNS

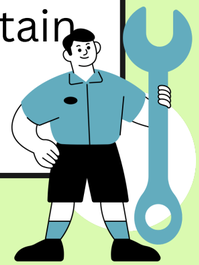
WHY TO FOLLOW A DESIGN PATTERN ?

REUSABILITY

By utilising Design patterns we can prevent reinventing the wheel and hence save time and effort by using battle tested and reusable solutions to common problems and use-cases.

MAINTABILITY

Design patterns can make code much cleaner and well-structured. Since they are based on the principles of DRY and single responsibility the resulting code is much easier to understand, test and maintain.



SCALABILITY

Most of the design patterns are built around the concept of extensibility and modularity in mind which makes adding new functionalities or modifying existing ones a piece of cake.

1. Container/Presentational Pattern

SUMMARY

Seperate out the stateful logic into **container** components and the UI rendering into dumb **presentational** components.

Container components - "**What"** data to render
Presentational components - "**How"** to render and show the data

1. Container/Presentational Pattern (cont.)

EXAMPLE

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

// Container Component
const TodoContainer = () => {
  const [todos, setTodos] = useState([]);
  const [newTodo, setNewTodo] = useState('');

  const handleInputChange = (event) => {
    setNewTodo(event.target.value);
  };

  const handleAddTodo = () => {
    if (newTodo.trim() !== '') {
      setTodos([...todos, newTodo]);
      setNewTodo('');
    }
  };

  return (
    <div>
      <input type="text" value={newTodo} onChange={handleInputChange} />
      <button onClick={handleAddTodo}>Add Todo</button>
      <TodoList todos={todos} />
    </div>
  );
};

export default TodoContainer;
```

```jsx
import React from 'react';

// Presentational Component
const TodoList = ({todos}) => {
   return (
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>
            {todo}
          </li>
        ))}
      </ul>
    );
  };

};

export default TodoList;
```

1. Container/Presentational Pattern (cont.)

EXAMPLE

In this example, the **TodoContainer** component is responsible for maintaining the todo list data and making updates to it if needed.

The **TodoList** component is a presentational component that receives the todos array as a prop and renders the list of todos and doesn't care anything about the state management or any business logic.

1. Container/Presentational Pattern (cont.)

ADVANTAGES

The main advantage of this pattern is that it helps achieve **Seperation of Concerns** which results in easier testing, modularity, and code reusability.

2. Higher Order Components (HOCs) Pattern

SUMMARY

It enables reusing logic across multiple components by wrapping them with a higher-order component.

A **HOC** is a component which takes in another component as prop and returns an enhanced version of that component.

2. Higher Order Components (HOCs) Pattern (cont.)

EXAMPLE

Let's say in our application we have a few components which are accessible only by authenticated users, instead of writing the logic to check if a user is authenticated or not inside each component, we can write an HOC ***withAuth*** *which will hold this logic and any component which is wrapped using this HOC will only be visible to authenticated users.*

Implementation on the next slide

2. Higher Order Components (HOCs) Pattern (cont.)

EXAMPLE

```
import React, { useState, useEffect } from 'react';

const withAuth = (WrappedComponent) => {
  return (props) => {
    const [isAuthenticated, setIsAuthenticated] = useState(false);

    useEffect(() => {
      // Simulating authentication check
      const isAuthenticated = checkAuthentication(); // Some authentication logic
      setIsAuthenticated(isAuthenticated);
    }, []);

    if (isAuthenticated) {
      return <WrappedComponent {...props} />;
    } else {
      return <div>Please login to access this component.</div>;
    }
  };
};
```

2. Higher Order Components(HOCs) Pattern (cont.)

ADVANTAGES

Just like the previous pattern the HOC pattern also helps achieve **Seperation of Concerns.**

In our example the **withAuth** HOC abstracted away the authentication logic from the component while the component focused solely on its primary responsibility of rendering the UI.

3. Render Props Pattern

SUMMARY

It enables passing functions as props in order to delegate the **rendering** control to the **consuming** component.

The function can take internal component data as arguments and must return a JSX element.

3. Render Props Pattern (cont.)

EXAMPLE

Let's say we have a ***<Counter/>*** *component, which keeps track of the current count and also enables us to increment the count by triggering an increment function.*
We want to use this count outside the component and render different emojis based on the current count value.
One way to solve this problem is to ***lift the state up*** *but that is not always feasible and also it might lead to unnecessary re-rendering in other child components, in order to avoid those issues we can instead use the* ***Render Props*** *pattern here*

See how on the next slide

3. Render Props Pattern (cont.)

EXAMPLE

```jsx
const Counter = ({ render }) => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount((prevCount) => prevCount + 1);
  };

  return <div>{render(count, increment)}</div>
};
```

```jsx
const App = () => {
  return (
    <div>
      <h1>Laughing Counter</h1>
      <Counter
        render={(count, increment) => (
          <div>
            <div>{new Array(count).fill(0).map((item) => '😂')}</
            <button onClick={increment}>Laugh More</button>
          </div>
        )}
      />
    </div>
  );
};
```

3. Render Props Pattern (cont.)

ADVANTAGES

This pattern helps us achieve **reusability** and **flexibility**. We can reuse the same component in multiple places and customise the rendering logic as per our requirements using the render prop.

4. Compound Components Pattern

SUMMARY

It allows creating components that work together to perform a single task, by sharing internal state and behaviour among each other.

Instead of creating a single component that takes care of all the functionality alone we break it into multiple child components which are **compounded** together to get the full functionality.

4. Compound Components Pattern (cont.)

EXAMPLE

This pattern is very commonly used while building complex components like <Select/>, <Accordion/>, <Tabs/>, <Menu/>, etc. for any reusable library.

Let's build a simple implementation for a Tabs component which will take in a array of tabs as children and also keep track of the current active tab.

Implementation on the next slide

4. Compound Components Pattern (cont.)

EXAMPLE

```jsx
const TabsContext = createContext(null);

const Tabs = ({ children }) => {
  const [activeTab, setActiveTab] = useState('prod');

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

```jsx
const Tab = ({ id, children }) => {
  const { activeTab, setActiveTab } = useContext(TabsContext);

  return (
    <button
      className={id === activeTab ? 'active' : ''}
      style={id === activeTab ? { background: 'lightblue' } : {}}
      onClick={() => setActiveTab(id)}
    >
      {children}
    </button>
  );
};

Tabs.Tab = Tab;
```

4. Compound Components Pattern (cont.)

EXAMPLE

```
const App = () => {
  return (
    <div>
      Select mode
      <Tabs>
        <Tabs.Tab id="prod">Production</Tabs.Tab>
        <Tabs.Tab id="dev">Development</Tabs.Tab>
        <Tabs.Tab id="sandbox"> Sandbox</Tabs.Tab>
      </Tabs>
    </div>
  );
};

export default App;
```

4. Compound Components Pattern (cont.)

ADVANTAGES

This pattern enables better **separation of concern**, **extensibility** and **customisation** of complex UI components.

In our Tabs example, using the Compound pattern enabled us to customise the content of each Tab without having to pass any additional props or configuration to the parent Tabs component.

Also it achieved SOC by making sure the Tabs component is responsible for managing the overall state and individual Tab component is responsible only for the rendering logic of each tab.

5. Provider Pattern

SUMMARY

It allows sharing data across multiple components without having to pass it down explicitly as props at each level.

It utilises the **Context** API to define the context to be shared across multiple components and makes it available using the **Context.Provider** component.

And a component which is a child of the provider can then consume the context using **Context.Consumer** or **useContext** hook

5. Provider Pattern (cont.)

EXAMPLE

A very common use-case of this pattern is to make the current selected theme or language accessible and modifiable across multiple child components.

*Let's create a **ThemeProvider** which will provide the ThemeContext to all children at any level of the application tree.*

Implementation on the next slide

5. Provider Pattern (cont.)

EXAMPLE

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

export const ThemeContext = createContext(null);

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 default ThemeProvider;
```

5. Provider Pattern (cont.)

EXAMPLE

```jsx
import React from 'react';
import ThemeProvider from './ThemeProvider';
import Header from './Header';
import Content from './Content';

const App = () => {
  return (
    <ThemeProvider>
      <Header />
      <Content />
    </ThemeProvider>
  );
};

export default App;
```

```jsx
const Header = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <header>
      <h1 style={{ color: theme === 'light' ? 'black' : 'white' }}>
        My app
      </h1>
    </header>
  );
};
```

5. Provider Pattern (cont.)

ADVANTAGES

This pattern results in a more **cleaner** and **maintainable** codebase.

It prevents **prop-drilling** and since the state is maintained in a centralised location it becomes very easy to **refactor** or make any changes to the state logic.

6. Hooks Pattern

SUMMARY

It enables us to write **functions** containing stateful logic which can be reused across multiple components.

Hooks allow functional components to manage state and different lifecycle methods.

This is the most recent design pattern in the React ecosystem and by far the most powerful one since it can replace most of the patterns we have discussed so far

6. Hooks Pattern (cont.)

USE CASES

1 STATE MANAGEMENT

One of the most common use case of hooks is to manage state inside functional components using **useState** hook

2 LIFECYCLE METHODS

We can perform side-effects at different lifecycle points of a comp. like mounting, unmounting and updates using the **useEffect** hook

3 REUSE LOGIC

We can encapsulate custom logic into hooks and reuse it across components just like any inbuilt hooks

4 PERFORMANCE ENHANCEMENT

Using hooks like **useMemo** and **useCallback** we can prevent unnecessary re-rendering of components and improve performance

6. Hooks Pattern (cont.)

ADVANTAGES

1 BETTER MAINTABILITY

Functional components with hooks are easier to understand and simpler as there are no intricacies of "this" unlike Class based components

2 EASIER TO TEST

Hooks make testing easier as they isolate the business logic from the rendering logic

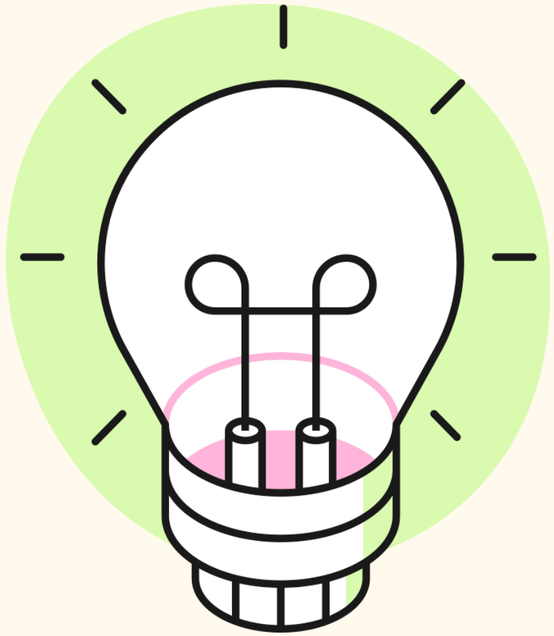
3 BETTER REUSABILITY

Custom hooks allow easy sharing of logic and reduce duplication of code

4 SMALLER BUNDLE SIZE

Functional components with hooks are much more concise when compared to their Class counterparts, resulting in smaller bundle size

REFERENCES



- https://www.patterns.dev/
- https://blog.logrocket.com/react-design-patterns/
- https://chat.openai.com/

Thank you!

learn, code & repeat.