Top React Design Patterns

Every Front End Developer Should Know

Find out more at → greatfrontend.com

Container-Presenter Pattern

This pattern splits components into two categories:

- **Containers**: These handle the application's logic, such as fetching data, managing state, and handling user interactions. They focus on the "how"
- **Presenters**: These are responsible for the UI and rendering logic. They focus on the "what"

Why use it?

- **Separation of concerns**: Logic and UI are decoupled, making components reusable and easier to test
- **Cleaner codebase**: Presenters are stateless and focused solely on rendering, while containers manage state and side effects

```
//Container
const UserContainer = () => {
  const [users, setUsers] = useState ([]);
  useEffect(() => {
    fetch("/api/users").then(res => res.json()).then(setUsers);
  }, []);
  return <UserPresenter users={users}/>;
};

//Presenter
const UserPresenter = ({users}) => (
  <ul>
    {users.map(user => (
      <li key={user.id}>{user.name}</li>
    ))}
  </ul>
);
```

Higher-Order Components (HOCs)

HOCs are functions that enhance components by adding shared functionality without code duplication.

Use cases

- Authentication (e.g., wrapping a component to check if a user is logged in)
- Logging, analytics, or tracking user interactions
- Enhancing components with additional props or behaviors

Why use it?

- Reusable logic that can be shared across multiple components

```
function withLoading(Component){
  return function EnhancedComponent({isLoading, ...props}){
    if (isLoading) return <p>Loading...</p>;
    return <Component {...props}/>;
  };
}

const UserList = withLoading(UserPresenter);
```

Render Props

This pattern uses a function prop to share state or behavior between components, offering a flexible alternative to HOCs.

Why use it?

- It gives components more control over what gets rendered

```
const DataFetcher = ({url, children}) => {
  const [data, setData] = useState(null);
  useEffect(() => {
    fetch(url).then(res => res.json()).then(setData);
  }, [url]);
  return children(data);
};

//Usage
<DataFetcher url="/api/users">
  {data => (data ? <UserPresenter users={data}/> : <p>Loading...</p>)}
</DataFetcher>;
```

Compound Components

Compound components allow you to create flexible and cohesive UI elements by grouping related components. This is often seen in libraries like React Router or Downshift.

Why use it?

- It gives users of the component full control over how the child components are used while maintaining a clear structure

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

Dropdown.Toggle = ({children}) => (
  <button className="dropdown-toggle">{children}</button>
);
Dropdown.Menu = ({children}) => (
  <div className="dropdown-menu">{children}</div>
);

//Usage
<Dropdown>
  <Dropdown.Toggle>Options</Dropdown.Toggle>
  <Dropdown.Menu>
    <p>Option 1</p>
    <p>Option 2</p>
  </Dropdown.Menu>
</Dropdown>;
```

Custom Hooks

Custom hooks encapsulate and reuse stateful logic across components, following React's hook rules to simplify code.

Why use it?

- Improves code readability and reusability
- Eliminates repetitive logic in functional components

```
function useFetch(url){
  const [data, setData] = useState(null);
  useEffect(() => {
    fetch(url).then(res => res.json()).then(setData);
  }, [url]);
  return data;
}

//Usage
const UserLIst = () => {
  const users = useFetch("/api/users");
  return users ? <UserPresenter users={users}/> : <p>Loading...</p>;
};
```