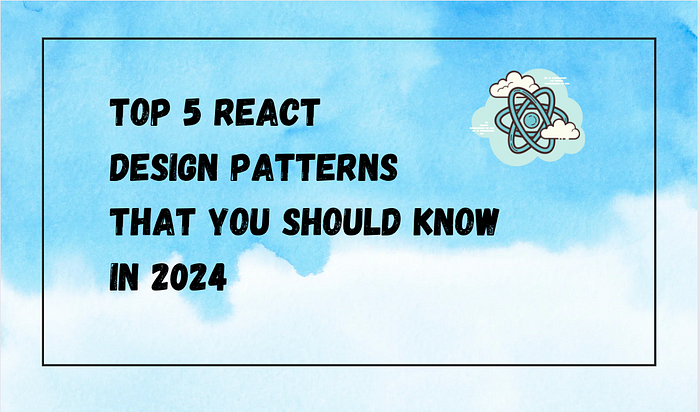
**Top 5 React Design Patterns That You Should Know in 2024**

Build Better React Components using these Design Patterns in 2024



If you’ve worked in React, one of the first things you’ve learned is to think in components. Rather than implementing one single component for an entire UI collection, you’re encouraged to break it down into smaller, reusable units. For example, consider the React component diagram:

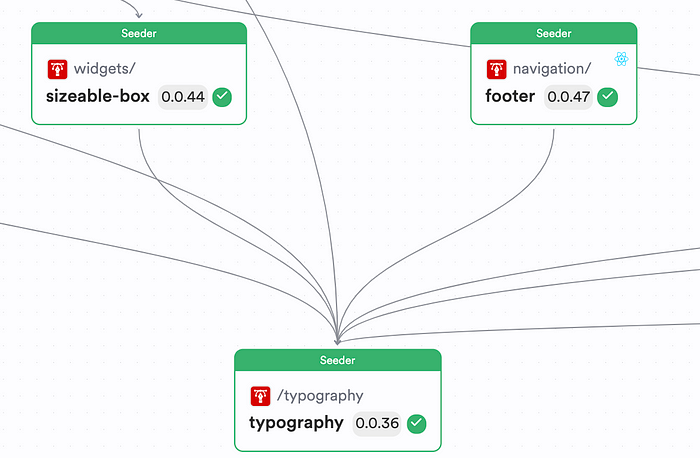


Figure: [A simple React component diagram](https://bit.cloud/showoff/design/~graph)

We have three components:

1. Typography
2. Footer
3. Sizeable Box

As you can see, the Typography component is used by both the Footer and the Sizeable Box. By doing so, we build a more straightforward application to maintain and troubleshoot errors. However, that's not all.

*If you know how to think in components, you can leverage design patterns on your React components to improve your code’s overall modularity, scalability, and maintainability.*

So, here are five design patterns you must consider when working with React.

**Pattern 01: The Base Component**

First, when working with React, try to create base components for your application.

*A*[*Base UI Component*](https://blog.bitsrc.io/how-to-build-better-react-components-in-2024-2d930b1f30b1)*is a component that has a pre-defined behavior that the consumer can customize.*

For example, in every application, you’d have a Base Theme, Base Button Design, or Base Typography that you might use to achieve a similar look and feel. But, something different here is that:

* You have a set of pre-defined configurations that your component applies. So, your consumer does not need to do any additional configuration but can quickly use the base configuration of the component instead.
* You also let consumers override the behavior of your component and provide a custom definition that defines the overall look and feel of the component.

A good implementation of the Base Component pattern can be illustrated with a simple Button component. For instance:

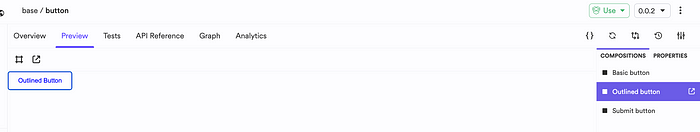
1. Your Button component might have different variations, such as an outlined or a filled variant.
2. Your Button might have an initial label on it

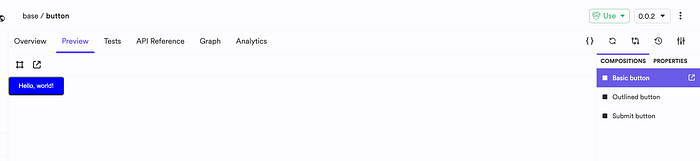
You can design your component in such a manner and let your consumers change its behavior. For example, [look at this implementation I’ve done of the Button component](https://bit.cloud/lakinduhewa/react-design-patterns/base/button) using the Base Component Pattern:

import React, { ButtonHTMLAttributes } from 'react';  
  
type ButtonVariant = 'filled' | 'outlined';  
  
export type ButtonProps = {  
 /\*\*  
 \* the variant of the button to use   
 \* @default 'outlined'  
 \*/  
 variant?: ButtonVariant;  
} & ButtonHTMLAttributes<HTMLButtonElement>;;  
  
const ButtonStyles: { [key in ButtonVariant]: React.CSSProperties } = {  
 filled: {  
 backgroundColor: 'blue', // Change this to your filled button color  
 color: 'white',  
 },  
 outlined: {  
 border: '2px solid blue', // Change this to your outlined button color  
 backgroundColor: 'transparent',  
 color: 'blue',  
 },  
};  
  
export function Button({ variant = 'outlined', children, style, ...rest }: ButtonProps) {  
 return (  
 <button  
 type='button'  
 style={{  
 ...ButtonStyles[variant],  
 padding: '10px 20px',  
 borderRadius: '5px',  
 cursor: 'pointer',  
 ...style  
 }} {...rest}>  
 {children}  
 </button>  
 );  
}

If you observe closely, the Prop type for the Button merges with the overall prop types of the HTML button. This means that apart from the default configured props, consumers can pass their own configurations, such as an onClick or even an aria-label onto this component, and it will be handled using the ...rest being spread onto the button component.

To see the Base Button component being used in different contexts, [have a look at its implemetations](https://bit.cloud/lakinduhewa/react-design-patterns/base/button/~compositions):





**Figure: Different contexts that the base component is used in**

As you can see, based on contexts, you can determine the behavior of your component. By doing so, you let your component become the foundation of something much bigger!

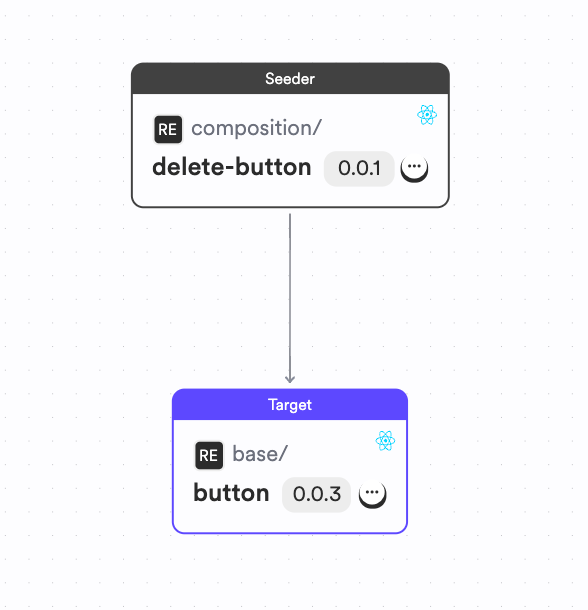
**Pattern 02: Component Composition**

Next, after you’ve successfully created your base components, there are cases where you want to create something out of your base components.

For example, you might use the Button component that we created earlier to implement a standard Delete button component. This Delete button component can be used across your application to keep all delete actions consistent and might have a fixed color, variant, and even a font type.

*Therefore, if you run into cases where you feel like you’re combining a set of components repeatedly to create the same output, wrap it in a component.*

[Let’s look at one such implementation](https://bit.cloud/lakinduhewa/react-design-patterns/composition/delete-button):

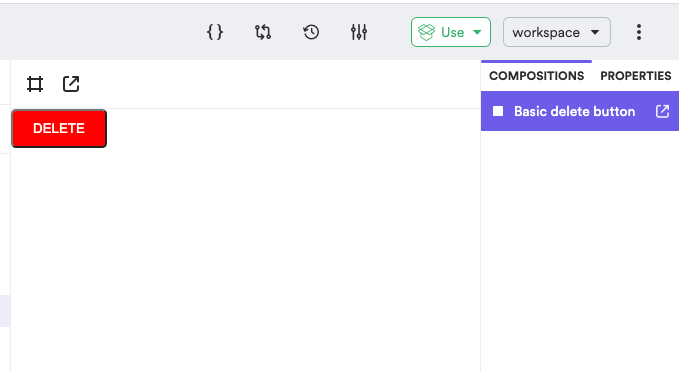


**Figure: Building a component using the composition pattern**

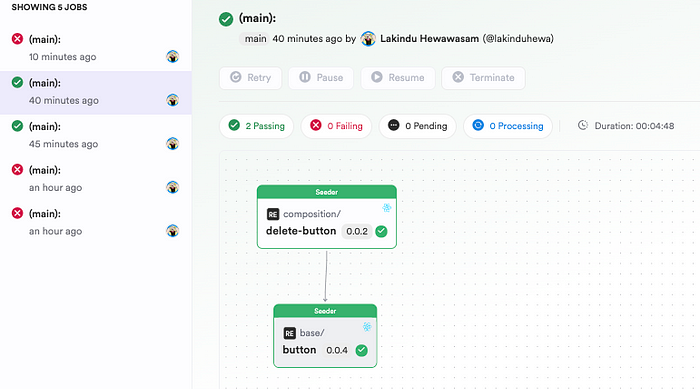
As depicted above using the [dependency graph](https://bit.cloud/lakinduhewa/react-design-patterns/composition/delete-button/~graph), the Delete button uses the Button component to provide a standard implementation for all Delete related operations. It's codebase would look something like this:

import { Button, ButtonProps } from '@lakinduhewa/react-design-patterns.base.button';  
import React from 'react';  
  
export type DeleteButtonProps = {} & ButtonProps;  
  
export function DeleteButton({ ...rest }: DeleteButtonProps) {  
 return (  
 <Button  
 variant='filled'  
 style={{  
 background: 'red',  
 color: 'white'  
 }}  
 {...rest}  
 >  
 DELETE  
 </Button>  
 );  
}

The DeleteButton would use the Button component that we implemented in Pattern 01 to create an output like this:



As you can see, we can now use a consistent Delete button across our application. Additionally, when you’re using build systems like [Bit](https://bit.de/) to design and build your components, you can leverage their CI server to [automatically propagate changes](https://bit.cloud/lakinduhewa/react-design-patterns/~ripple-ci/job/lakinduhewa-main-tlP1aG7L0) to the DeleteButton if the implementation of the Button component changes, as shown below:



**Figure: A CI Build on Bit**

**Pattern 03: Using Hooks**

Alright, you’ve probably heard of React Hooks. They’ve [been around since React v16](https://legacy.reactjs.org/docs/hooks-overview.html) and let you manage concepts like state and effects without a Class component. So, in a nutshell, you can eliminate the need for Class Components with the Hooks API. But we all know the basic state and effect hooks. So, we’re not going to talk about that. Instead, we’re going to talk about how you can leverage hooks to improve the overall maintainability of your component.

For example, consider the following scenario:

1. You have a BlogList component.
2. The BlogList component communicates with a simple API to fetch a list of blog posts and it renders the list on the component.

In such cases, you would likely write your API logic directly in a functional component as shown below:

import React, { useState, useEffect } from 'react';  
import axios from 'axios';  
  
const BlogList = () => {  
 const [blogs, setBlogs] = useState([]);  
 const [isLoading, setIsLoading] = useState(true);  
 const [error, setError] = useState(null);  
  
 useEffect(() => {  
 axios.get('https://api.example.com/blogs')  
 .then(response => {  
 setBlogs(response.data);  
 setIsLoading(false);  
 })  
 .catch(error => {  
 setError(error);  
 setIsLoading(false);  
 });  
 }, []);  
  
 if (isLoading) return <div>Loading...</div>;  
 if (error) return <div>Error: {error.message}</div>;  
  
 return (  
 <div>  
 <h2>Blog List</h2>  
 <ul>  
 {blogs.map(blog => (  
 <li key={blog.id}>{blog.title}</li>  
 ))}  
 </ul>  
 </div>  
 );  
};  
  
export default BlogList;

This component will work just fine. It will fetch the list of blog posts and it’ll render it in the UI. But, you’re coupling your UI logic to your API logic.

*In an ideal world, your React component does not need to know how it’s getting the data. All it cares is receiving an array of items and then rendering it on the DOM.*

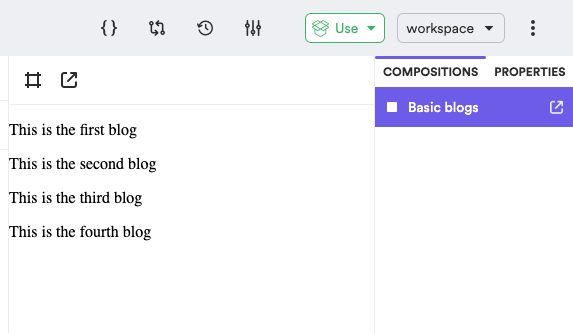
So, the best way to achieve this is to abstract your API logic onto a React Hook that you can call inside your component. Doing so breaks the coupling of the API call and the component. By doing so, you can change the underlying implementation of the invocation to get the data without impacting the component.

One such implementation would look something like this:

**1.**[**The useBlog hook**](https://bit.cloud/lakinduhewa/react-design-patterns/hooks/use-blog)

import { useEffect, useState } from 'react';  
import { Blog } from './blog.type';  
import { Blogs } from './blog.mock';  
  
export function useBlog() {  
 const [blogs, setBlogs] = useState<Blog[]>([]);  
 const [loading, setLoading] = useState<boolean>(false);  
  
 useEffect(() => {  
 setLoading(true);  
 setTimeout(() => {  
 setBlogs(Blogs);  
 setLoading(false);  
 }, 3000);  
 }, []);  
  
 return { blogs, loading }  
}

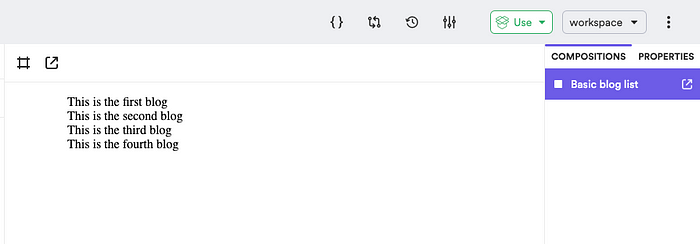
As shown above, the useBlog hook fetches a list of blogs and assigns it to a state variable and exposes the variables for the consumer (BlogList) to use:



**Figure: The Hook in Action**

**2.**[**The BlogList component**](https://bit.cloud/lakinduhewa/react-design-patterns/hooks/blog-list)

import React from 'react';  
import { useBlog } from '@lakinduhewa/react-design-patterns.hooks.use-blog';  
  
export function BlogList() {  
 const { blogs, loading } = useBlog();  
  
 if (loading) {  
 return (  
 <p>We are loading the blogs...</p>  
 )  
 }  
  
 return (  
 <ul>  
 {blogs.map((blog) => <ol  
 key={blog.id}  
 >  
 {blog.title}  
 </ol>)}  
 </ul>  
 );  
}



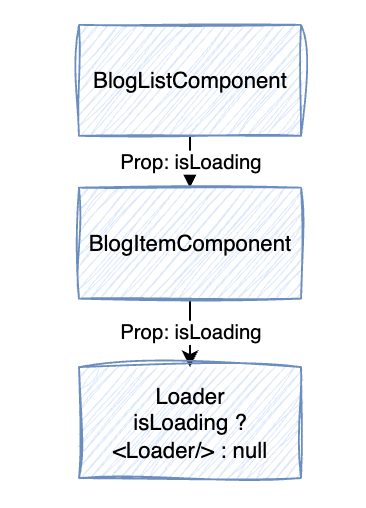
**Figure: The BlogList in action**

We’ve successfully used the hook in the BlogList component by invoking useBlog and using the state variables offered to us. By doing so, we can reduce the huge chunk of code that we initially started with, and maintain two components with minimal code and effort!

Plus, when you’re using build systems like Bit (like I do), all you have to do is import the useBlog component onto your local development environment and then make a change and push it back to Bit Cloud. Its build server can propagate the changes across the tree, so you don't even need access to the entire project to make a simple change!

**Pattern 04: React Providers**

This pattern revolves around sharing state across a set of components. We’ve all become a victim to prop drilling at some point in our careers. But, if you haven’t, prop drilling is when you pass a prop down a tree of components only for it to be used at the lowest level and not anywhere else. For example, consider this diagram:



**Figure: Prop Drilling**

As you can see, we are passing a prop isLoading from the BlogListComponent all the way to Loader down the tree. But, the isLoading is only used in the Loader component. So, in such cases, you introduce unnecessary props across your components and can take a tool in performance too, as React will re-render your tree when isLoading changes, even though it isn't being used.

So, one fix is to use the React Context Provider pattern by introducing a React Context. A React Context is a state manager for a set of components in which you provide a particular context for a set of components. By doing so, you can define and manage your state in the Context and let your components access the context at different levels and use the props that they need. By doing so, you eliminate prop-drilling.

One example of using this in practise is a Theme component. For instance, you want your Theme to be accessed globally in your app. But passing the theme down every component you use is not practical. Instead, you can create a Context that holds the theme information, and then consume the context to set the theme. Let’s [explore my Theme Context to get a better understanding](https://bit.cloud/lakinduhewa/react-design-patterns/contexts/consumer-component) of this:

import { useContext, createContext } from 'react';  
  
export type SampleContextContextType = {  
 /\*\*  
 \* primary color of theme.  
 \*/  
 color?: string;  
};  
  
export const SampleContextContext = createContext<SampleContextContextType>({  
 color: 'aqua'  
});  
  
export const useSampleContext = () => useContext(SampleContextContext);

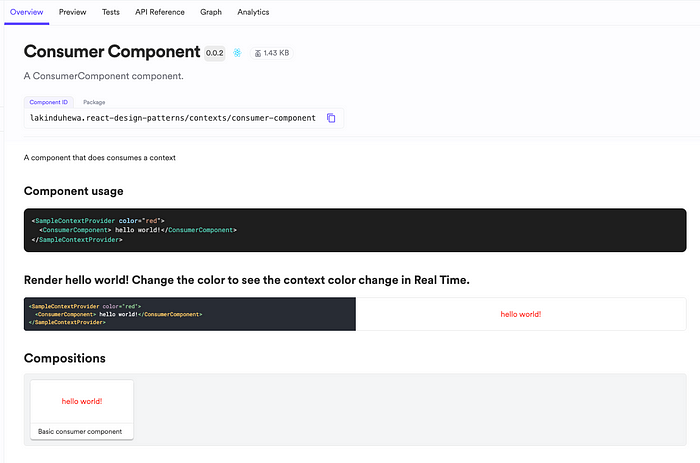
The context defines a theme color that it’ll use to configure the font color across all of its implementations. Next, I’ve also exposed a hook — useSampleContext that lets consumers directly consume the Context.

But, that’s not all. We’ll also need to define a Provider. The provider is the component that answers the questions — “To what components shall I share my state with?” You can [implement a provider like this](https://bit.cloud/lakinduhewa/react-design-patterns/contexts/sample-context/~code/sample-context-context-provider.tsx):

import React, { ReactNode } from 'react';  
import { SampleContextContext } from './sample-context-context';  
  
export type SampleContextProviderProps = {  
 /\*\*  
 \* primary color of theme.  
 \*/  
 color?: string,  
  
 /\*\*  
 \* children to be rendered within this theme.  
 \*/  
 children: ReactNode  
};  
  
export function SampleContextProvider({ color, children }: SampleContextProviderProps) {  
 return <SampleContextContext.Provider value={{ color }}>{children}</SampleContextContext.Provider>  
}

As you can see, the provider plays a crucial role in managing the initial state and setting the context of the components that can access the state.

Next, you can create a [consumer component](https://bit.cloud/lakinduhewa/react-design-patterns/contexts/consumer-component) that consumes the state:



**Figure: The consumer component**

**Pattern 05: Conditional Rendering**

The final pattern that I’d like to leave you wish is conditional rendering. Now, we all know conditional rendering in React. It’s where you render a component based on a condition.

But, we fail to implement this correctly. We usually implement conditional rendering like this:

// ComponentA.js  
const ComponentA = () => {  
 return <div>This is Component A</div>;  
};  
  
// ComponentB.js  
const ComponentB = () => {  
 return <div>This is Component B</div>;  
};  
  
// ConditionalComponent.js  
import React, { useState } from 'react';  
import ComponentA from './ComponentA';  
import ComponentB from './ComponentB';  
  
const ConditionalComponent = () => {  
 const [toggle, setToggle] = useState(true);  
  
 return (  
 <div>  
 <button onClick={() => setToggle(!toggle)}>Toggle Component</button>  
 {toggle ? <ComponentA /> : <ComponentB />}  
 </div>  
 );  
};  
  
export default ConditionalComponent;

If you noticed, we couple conditional-based logic onto our JSX fragments. Generally, you don’t want to add any compute-related logic onto your JSX and only render UI components in it.

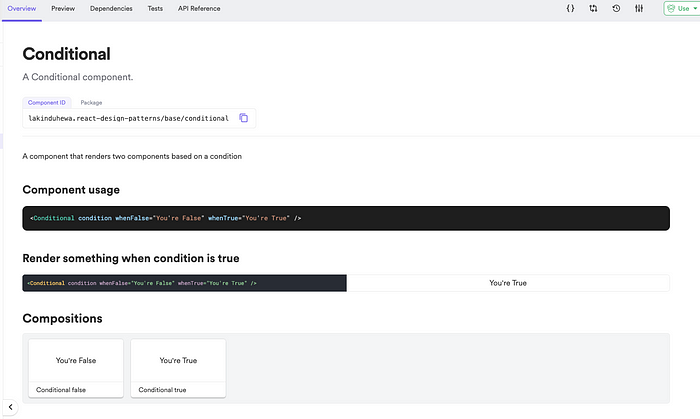
One way to solve this is using the Conditional Render Component pattern. You can create a reusable React component that would render two components based on a condition. [It’s implementation would look something like this](https://bit.cloud/lakinduhewa/react-design-patterns/base/conditional):

import React, { ReactNode } from 'react';  
  
export type ConditionalProps = {  
 /\*\*  
 \* the condition to test against  
 \*/  
 condition: boolean  
 /\*\*  
 \* the component to render when condition is true  
 \*/  
 whenTrue: ReactNode  
 /\*\*  
 \* the component to render when condition is false  
 \*/  
 whenFalse: ReactNode  
};  
  
export function Conditional({ condition, whenFalse, whenTrue }: ConditionalProps) {  
 return condition ? whenTrue : whenFalse;  
}

As you can see, we’ve created a component that conditionally renders two components. This will give us a clean code when we integrate this into our other components as we eliminate the need to have complex rendering logic inside our React components. You can use it like this:

export const ConditionalTrue = () => {  
 return (  
 <Conditional  
 condition  
 whenFalse="You're False"  
 whenTrue="You're True"  
 />  
 );  
}  
  
export const ConditionalFalse = () => {  
 return (  
 <Conditional  
 condition={false}  
 whenFalse="You're False"  
 whenTrue="You're True"  
 />  
 );  
}

This would create outputs like this:



**Wrapping Up**

And that’s it! Mastering these five design patterns will keep you well-prepared for 2024 and will let you build highly scalable and maintainable

If you’d like to explore the patterns we discussed in this article, feel free to [check out my scope on Bit Cloud](https://bit.cloud/lakinduhewa/react-design-patterns).

I hope you found this article helpful.

Thank you for reading!