**Separating⛓️ Business Logic 🧠from UI Components 🔡 in React 18**

[[Israel](https://oluwadaprof.medium.com/?source=post_page-----aa1775b3caba--------------------------------)](https://oluwadaprof.medium.com/?source=post_page-----aa1775b3caba--------------------------------)

[[Bootcamp](https://bootcamp.uxdesign.cc/?source=post_page-----aa1775b3caba--------------------------------)](https://bootcamp.uxdesign.cc/?source=post_page-----aa1775b3caba--------------------------------)

[Israel](https://oluwadaprof.medium.com/?source=post_page-----aa1775b3caba--------------------------------)

·

Follow

Published in

[Bootcamp](https://bootcamp.uxdesign.cc/?source=post_page-----aa1775b3caba--------------------------------)

·

6 min read

·

Aug 7

1K

9

**Introduction**

React is a popular JavaScript library for building user interfaces, and it has undergone significant improvements in recent versions. With React 18, developers now have even more powerful tools to separate business logic from UI components effectively. This article will explore the benefits of decoupling business logic from UI components and demonstrate how to achieve it using React 18’s new features.

**Table of Content**

* Why Separate Business Logic from UI Components?
* React 18’s New Features for Business Logic Separation.
* Separating Business Logic: A Step-by-Step Guide.
* Advanced Techniques for Business Logic Separation in React 18
* Conclusion
* References

**Why Separate Business Logic from UI Components?**

*Separating business logic from UI components brings several advantages to your React applications:*

* Code Reusability: When business logic is isolated, it becomes easier to reuse components across different parts of the application or even in other projects.
* Testing: Testing becomes more straightforward as you can write unit tests for the business logic independently from the UI components. This improves test coverage and overall code quality.
* Readability and Maintainability: Decoupling business logic from UI components results in cleaner and more maintainable code. It becomes easier to understand the responsibilities of each part of the application.
* Scalability: As your application grows, separating concerns allows developers to work on different parts of the codebase simultaneously without conflicts.

**React 18’s New Features for Business Logic Separation**

*With React 18, two key features empower developers to separate business logic effectively:*

**React Hooks**

React Hooks, introduced in React 16.8, provide a way to use state and other React features without writing class components. With hooks like useState and useEffect, developers can encapsulate business logic and reuse it across components.

**Suspense with Concurrent Mode**

React 18 brings improvements in Concurrent Mode and Suspense. Concurrent Mode allows React to work on multiple tasks simultaneously, providing better performance. Suspense enables declarative data fetching and code splitting, making it easier to handle asynchronous operations in your components.

**Separating Business Logic: A Step-by-Step Guide**

*Let’s walk through a step-by-step guide on how to separate business logic from UI components in a React 18 application.*

**Step 1: Identify Business Logic**

Identify the parts of your components that deal with business logic. These are usually operations that manage data, state, or perform API calls.

**Step 2: Create Custom Hooks**

Create custom hooks to encapsulate the identified business logic. Custom hooks are functions that start with use and can use other hooks internally.

*Code Example (Custom Hook):*

// useUserData.js  
import { useState, useEffect } from 'react';  
  
export function useUserData() {  
 const [userData, setUserData] = useState([]);  
  
 useEffect(() => {  
 // Fetch user data from the API and update the state  
 fetch('https://api.example.com/users')  
 .then((response) => response.json())  
 .then((data) => setUserData(data))  
 .catch((error) => console.error('Error fetching data:', error));  
 }, []);  
  
 return userData;  
}

**Step 3: Use Custom Hooks in Components**

Utilize the custom hooks within your UI components to access the business logic and data. This keeps your components focused on rendering, while the hooks handle the underlying logic.

*Code Example (Using Custom Hook in Component):*

// UserList.js  
import React from 'react';  
import { useUserData } from './useUserData';  
  
function UserList() {  
 const userData = useUserData();  
  
 return (  
 <div>  
 <h1>User List</h1>  
 <ul>  
 {userData.map((user) => (  
 <li key={user.id}>{user.name}</li>  
 ))}  
 </ul>  
 </div>  
 );  
}

**Step 4: Code Splitting with Suspense**

With Suspense, you can code split and lazily load components that require asynchronous data fetching or expensive computations. This further separates concerns and improves the performance of your application.

*Code Example (Code Splitting with Suspense):*

// App.js  
import React, { Suspense } from 'react';  
const UserList = React.lazy(() => import('./UserList'));  
  
function App() {  
 return (  
 <div>  
 <h1>My App</h1>  
 <Suspense fallback={<div>Loading...</div>}>  
 <UserList />  
 </Suspense>  
 </div>  
 );  
}

**Advanced Techniques for Business Logic Separation in React 18**

*In the previous section, we covered the basics of separating business logic from UI components using React 18’s new features. Now, let’s explore some advanced techniques that take this separation to the next level.*

**1. Custom Hooks Composition**

One powerful aspect of custom hooks is that they can be composed to create more complex logic. By combining multiple custom hooks, you can build sophisticated business logic while keeping your UI components clean and focused.

*Code Example (Custom Hook Composition):*

// useUserData.js  
import { useState, useEffect } from 'react';  
  
export function useUserData() {  
 const [userData, setUserData] = useState([]);  
  
 useEffect(() => {  
 // Fetch user data from the API and update the state  
 fetch('https://api.example.com/users')  
 .then((response) => response.json())  
 .then((data) => setUserData(data))  
 .catch((error) => console.error('Error fetching data:', error));  
 }, []);  
  
 return userData;  
}  
  
// useFilteredUserData.js  
import { useUserData } from './useUserData';  
  
export function useFilteredUserData(searchTerm) {  
 const userData = useUserData();  
  
 // Filter user data based on the search term  
 const filteredData = userData.filter((user) =>  
 user.name.toLowerCase().includes(searchTerm.toLowerCase())  
 );  
  
 return filteredData;  
}

With the useFilteredUserData hook, you can now get filtered user data based on a search term and reuse the fetching logic from useUserData. This composition approach keeps the logic modular and reusable across components.

**2. Custom Hooks for Side Effects**

Sometimes, your business logic may involve side effects beyond data fetching, such as managing timers or interacting with external APIs. Custom hooks can encapsulate such side effects and ensure they are handled consistently across components.

*Code Example (Custom Hook for Timer):*

// useTimer.js  
import { useState, useEffect } from 'react';  
  
export function useTimer(initialTime = 0) {  
 const [time, setTime] = useState(initialTime);  
  
 useEffect(() => {  
 const interval = setInterval(() => {  
 setTime((prevTime) => prevTime + 1);  
 }, 1000);  
  
 return () => clearInterval(interval);  
 }, []);  
  
 return time;  
}

Now, you can easily incorporate the timer functionality into any component without worrying about managing the interval.

**3. Context for Global State Management**

For state management that needs to be shared across multiple components, React 18’s Context API can be leveraged. By creating a context for your business logic, you can access the state and actions from any component within the context.

*Code Example (Using Context for State Management):*

// UserDataContext.js  
import React, { createContext, useContext } from 'react';  
import { useUserData } from './useUserData';  
  
const UserDataContext = createContext();  
  
export function UserDataProvider({ children }) {  
 const userData = useUserData();  
  
 return (  
 <UserDataContext.Provider value={userData}>  
 {children}  
 </UserDataContext.Provider>  
 );  
}  
  
export function useUserDataContext() {  
 return useContext(UserDataContext);  
}

By using UserDataContext and the useUserDataContext hook, you can access the user data from any component within the UserDataProvider without prop drilling.

**4. Higher-Order Components (HOCs) for Cross-Cutting Concerns**

In cases where certain logic needs to be applied to multiple components, Higher-Order Components (HOCs) can be used. HOCs are functions that take a component and return a new component with additional props or logic.

*Code Example (HOC for Styling):*

// withStyles.js  
import React from 'react';  
  
function withStyles(WrappedComponent) {  
 return function WithStyles(props) {  
 return (  
 <div style={{ color: 'blue' }}>  
 <WrappedComponent {...props} />  
 </div>  
 );  
 };  
}  
  
// Usage  
const MyComponent = ({ name }) => <div>Hello, {name}!</div>;  
const StyledComponent = withStyles(MyComponent);

The StyledComponent now automatically applies the defined styles to the MyComponent, eliminating the need to repeat the style declarations in multiple places.

**Conclusion**

By leveraging React 18's new features, including custom hooks, Suspense, and Context API, developers can implement advanced techniques to separate business logic from UI components more effectively. Custom hook composition, side effect handling, Context API for global state management, and Higher-Order Components allow developers to create modular and maintainable React applications.

By applying these techniques, developers can build highly scalable and maintainable applications with better code organization and improved performance.

**References**

* [React Documentation](https://reactjs.org/docs/getting-started.html)
* [Introducing Hooks in React](https://reactjs.org/docs/hooks-intro.html)
* [React 18 Alpha - Concurrent Mode](https://reactjs.org/docs/concurrent-mode-intro.html)
* [Using the State Hook](https://reactjs.org/docs/hooks-state.html)
* [Suspense for Data Fetching.](https://reactjs.org/docs/concurrent-mode-suspense.html)
* [React Custom Hooks - Composition](https://reactjs.org/docs/hooks-custom.html#using-a-custom-hook)
* [React Context](https://reactjs.org/docs/context.html)
* [React Higher-Order Components](https://reactjs.org/docs/higher-order-components.html)

These references provide additional information and in-depth details on React 18’s new features and best practices for separating business logic from UI components.