

Be part of a better internet. [Get 20% off membership for a limited time](#)

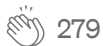
# React Design Patterns

Learn how to apply design patterns in your React applications.



Bryan Aguilar · [Follow](#)

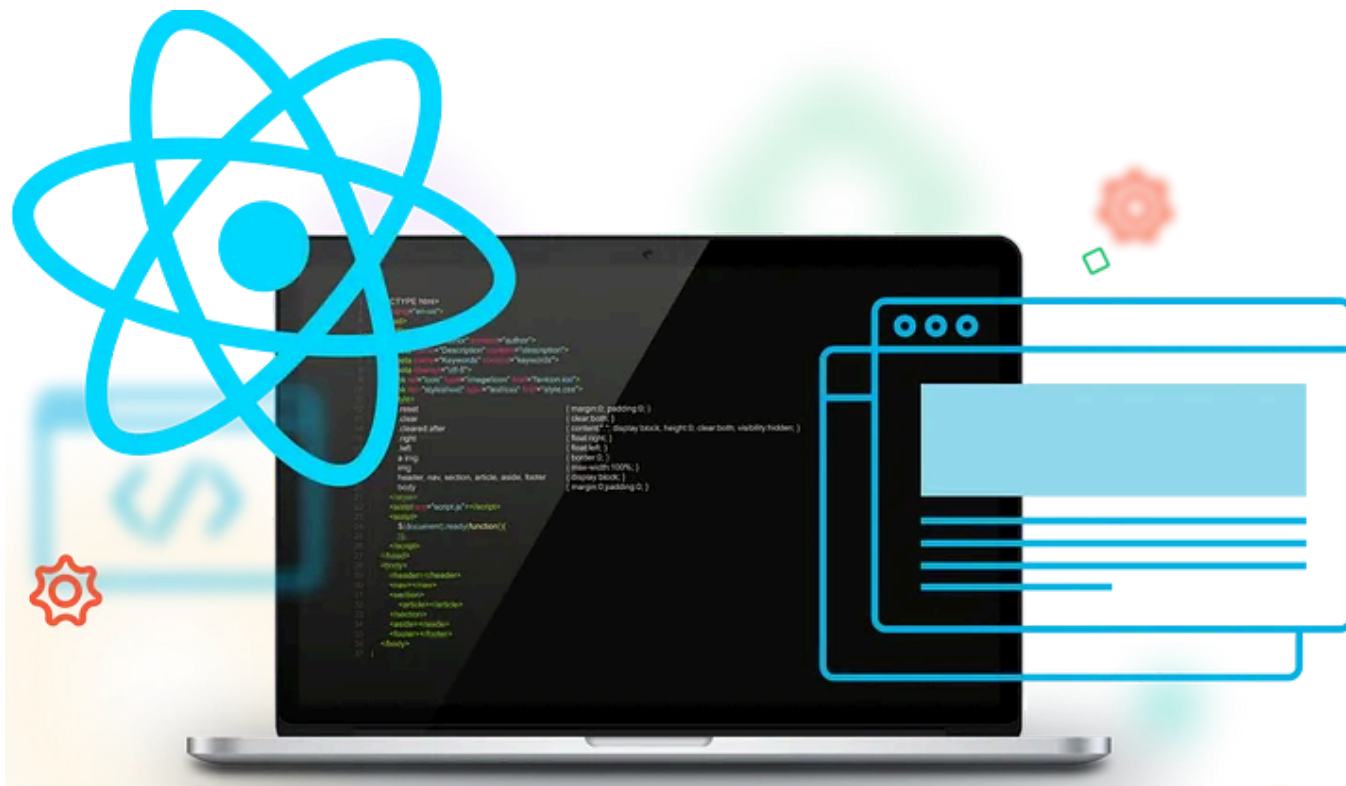
19 min read · Feb 28, 2024



279



2



In the world of frontend development with React, the application of design patterns has become an essential practice. These patterns have evolved in line with the specific needs of React, offering elegant solutions to the recurring challenges developers face when designing robust components and applications.

The fundamental purpose of these patterns is to address concrete problems in component development by simplifying the management of state, logic and element composition. By providing predefined structures and proven methodologies, design patterns in React promote consistency, modularity and scalability in the code base.

Among the most common examples of design patterns applied to React are custom hooks, higher-order components (HOCs), and prop-based rendering techniques. These elements are powerful tools that allow developers to optimize the structure and data flow in their applications, promoting code reuse and conceptual clarity.

It is important to emphasize that these patterns not only focus on solving technical problems, but also prioritize code efficiency, readability and maintainability. By adopting standardized practices and well-defined concepts, development teams can collaborate more effectively and build robust and adaptable React applications for the long term.

## **Why use patterns in React?**

The use of design patterns in React is critical to developers because of their ability to reduce development time and costs, mitigate the accumulation of technical debt, ease code maintenance, and promote continuous and sustainable development over time. These patterns provide a consistent

structure that streamlines the development process and improves overall software quality.

## Custom Hook Pattern

The Custom Hook pattern in React is a technique that allows encapsulating the logic of a component in a reusable function. Custom Hooks are JavaScript functions that use the Hooks provided by React (such as *useState*, *useEffect*, *useContext*, etc.) and can be shared between components to effectively encapsulate and reuse logic.

### When to use it

- When you need to share logic between React components without resorting to code duplication.
- To abstract the complex logic of a component and keep it more readable and easier to maintain.
- When you need to modularize the logic of a component to facilitate its unit testing.

### When not to use it

- When the logic is specific to a single component and will not be reused elsewhere.
- When the logic is simple and does not justify the creation of a Custom Hook.

## Advantages

- Promotes code reuse by encapsulating common logic in separate functions.
- Facilitates code composition and readability by separating logic from the component.
- Improves testability by enabling more specific and focused unit tests on the logic encapsulated in Custom Hooks.

## Disadvantages

- May result in additional complexity if abused and many Custom Hooks are created.
- Requires a solid understanding of React and Hooks concepts for proper implementation.

## Example

Here is an example of a Custom Hook that performs a generic HTTP request using TypeScript and React. This Hook handles the logic to make the request and handle the load status, data and errors.

```
import { useState, useEffect } from 'react';
import axios, { AxiosResponse, AxiosError } from 'axios';

type ApiResponse<T> = {
  data: T | null;
  loading: boolean;
  error: AxiosError | null;
};

function useFetch<T>(url: string): ApiResponse<T> {
  const [data, setData] = useState<T | null>(null);
  const [loading, setLoading] = useState<boolean>(true);
```

```
const [error, setError] = useState<AxiosError | null>(null);

useEffect(() => {
  const fetchData = async () => {
    try {
      const response: AxiosResponse<T> = await axios.get(url);
      setData(response.data);
    } catch (error) {
      setError(error);
    } finally {
      setLoading(false);
    }
  };

  fetchData();

  // Cleanup function
  return () => {
    // Cleanup logic, if necessary
  };
}, [url]);

return { data, loading, error };
}

// Using the Custom Hook on a component
function ExampleComponent() {
  const { data, loading, error } = useFetch<{ /* Expected data type */ >>('https

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  if (!data) {
    return <div>No data.</div>;
  }

  return (
    <div>
      { /* Rendering of the obtained data */ }
    </div>
  );
}

export default ExampleComponent;
```

In this example, the Custom Hook `useFetch` takes a URL as an argument and performs a GET request using `Axios`. It manages the load status, data and errors, returning an object with this information.

The `ExampleComponent` component uses the Custom Hook `useFetch` to fetch data from an API and render it in the user interface. Depending on the status of the request, a load indicator, an error message or the fetched data is displayed.

There are many ways to use this pattern, in this [link](#) you can find several examples of custom Hooks to solve specific problems, the uses are many.

## HOC Pattern

The Higher Order Component (HOC) pattern is a composition technique in React that is used to reuse logic between components. A HOC is a function that takes a component and returns a new component with additional or extended functionality.

### When to use it

- When you need to share logic between multiple components without duplicating code.
- To add common behaviors or features to multiple components.
- When you want to isolate presentation logic from business logic in a component.

## When not to use it

- When the logic is specific to a single component and will not be reused.
- When the logic is too complex and may make HOCs difficult to understand.

## Advantages

- Promotes code reuse by encapsulating and sharing logic between components.
- Allows clear separation of presentation logic from business logic.
- Facilitates code composition and modularity by applying functional design patterns.

## Disadvantages

- May introduce an additional layer of abstraction that makes it difficult to track data flow.
- Excessive composition of HOCs can generate complex components that are difficult to debug.
- Sometimes, it can hide the component hierarchy, making it difficult to understand how the application is structured.

## Example

Suppose we want to create a HOC that handles the state and methods for submitting data from a form. The HOC will handle the form values, validate the data and send the request to the server.

```

import React, { ComponentType, useState } from 'react';

interface FormValues {
  [key: string]: string;
}

interface WithFormProps {
  onSubmit: (values: FormValues) => void;
}

// HOC that handles form state and logic
function withForm<T extends WithFormProps>(WrappedComponent: ComponentType<T>) {
  const WithForm: React.FC<T> = (props) => {
    const [formValues, setFormValues] = useState<FormValues>({});

    const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
      const { name, value } = event.target;
      setFormValues((prevValues) => ({
        ...prevValues,
        [name]: value,
      }));
    };

    const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
      event.preventDefault();
      props.onSubmit(formValues);
    };

    return (
      <WrappedComponent
        {...props}
        formValues={formValues}
        onChange={handleInputChange}
        onSubmit={handleSubmit}
      />
    );
  };

  return WithForm;
}

// Component that uses the HOC to manage a form.
interface MyFormProps extends WithFormProps {
  formValues: FormValues;
  onChange: (event: React.ChangeEvent<HTMLInputElement>) => void;
}

const MyForm: React.FC<MyFormProps> = ({ formValues, onChange, onSubmit })

```



```

    return (
      <form onSubmit={onSubmit}>
        <input type="text" name="name" value={formValues.name || ''} onChange={onI
        <input type="text" name="email" value={formValues.email || ''} onChange={o
        <button type="submit">Enviar</button>
      </form>
    );
  };

  // Using the HOC to wrap the MyForm component
  const FormWithLogic = withForm(MyForm);

  // Main component that renders the form
  const App: React.FC = () => {
    const handleSubmit = (values: FormValues) => {
      console.log('Form values:', values);
      // Logic to send the form data to the server
    };

    return (
      <div>
        <h1>HOC Form</h1>
        <FormWithLogic onSubmit={handleSubmit} />
      </div>
    );
  };

  export default App;

```

In this example, the `withForm` HOC encapsulates the logic for handling a form. This HOC handles the state of the form values, provides a function to update the form values (`handleInputChange`), and a function to handle the form submission (`handleSubmit`). Then, the HOC is used to wrap the `MyForm` component, which is the form that will be rendered in the main application (`App`).

## Extensible Styles Pattern

The Extensible Styles pattern is a technique that allows the creation of React components with flexible and easily customizable styles. Instead of applying

styles directly to the component, this pattern uses dynamic CSS properties or classes that can be modified and extended according to the user's needs.

### **When to use it**

- When you need to create components that can adapt to different styles or themes within an application.
- To allow end users to easily customize the appearance of components.
- When you want to maintain visual consistency in the user interface while providing flexibility in the appearance of components.

### **When not to use it**

- When style customization is not a concern or styles are not expected to vary significantly.
- In applications where tight control over the styles and appearance of components is required.

### **Advantages**

- Facilitates customization and extension of styles in components without the need to modify the source code.
- Maintains visual consistency in the application while providing flexibility in styles.
- Simplifies maintenance by separating the styling logic from the component code.

### **Disadvantages**

- May result in increased complexity if extensible styles are not managed properly.

[Open in app](#)**Medium** Search Write

## Example

Suppose we want to create a button component with extensible styles that allows changing its color and size by means of props.

```
import React from 'react';
import './Button.css';

interface ButtonProps {
  color?: string;
  size?: 'small' | 'medium' | 'large';
  onClick: () => void;
}

const Button: React.FC<ButtonProps> = ({ color = 'blue', size = 'medium', onClick }) => {
  const buttonClasses = `Button ${color} ${size}`;

  return (
    <button className={buttonClasses} onClick={onClick}>
      {children}
    </button>
  );
};

export default Button;
```

```
.Button {
  border: none;
  cursor: pointer;
  padding: 8px 16px;
  border-radius: 4px;
```

```
    font-size: 14px;
    font-weight: bold;
  }

  .small {
    padding: 4px 8px;
  }

  .medium {
    padding: 8px 16px;
  }

  .large {
    padding: 12px 24px;
  }

  .blue {
    background-color: blue;
    color: white;
  }

  .red {
    background-color: red;
    color: white;
  }

  .green {
    background-color: green;
    color: white;
  }
```

In this example, the Button component accepts properties such as color and size, which can be used to customize its appearance. The CSS styles are defined in an extensible way, allowing the color and size of the button to be easily modified by props. This provides flexibility for the developer to adapt the component to different styles within the application.

## Compound Components Pattern

The Compound Components Pattern is a design technique in React that allows the creation of components that work closely and coherently together. In this pattern, a parent component can encapsulate multiple child

components, enabling seamless communication and coordinated interaction among them.

### When to use

- When you need to create components that depend on each other and perform better when grouped together.
- To build highly customizable and flexible components that can adapt to different use cases.
- When you want to maintain a clear and organized component structure in the React component tree hierarchy.

### When not to use

- In cases where the relationship between components is not close or there is no clear dependency between them.
- In situations where the added complexity of the Compound Components pattern does not justify its benefits.

### Advantages

- Facilitates encapsulation and reuse of related logic in a set of components.
- Provides a clear and consistent API for interacting with compound components.
- Allows for greater flexibility and customization by combining multiple components into one.

## Disadvantages

- Can introduce additional complexity in understanding how components interact with each other.
- Requires careful design to ensure that compound components are flexible and easy to use.

## Example

Suppose we want to create a Tabs component that encapsulates tabs (Tab) that can be shown and hidden according to the selected index.

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

interface TabProps {
  label: string;
  children: ReactNode;
}

const Tab: React.FC<TabProps> = ({ children }) => {
  return <>{children}</>;
};

interface TabsProps {
  children: ReactNode;
}

const Tabs: React.FC<TabsProps> = ({ children }) => {
  const [activeTab, setActiveTab] = useState(0);

  return (
    <div>
      <div className="tab-header">
        {React.Children.map(children, (child, index) => {
          if (React.isValidElement(child)) {
            return (
              <div
                className={`tab-item ${index === activeTab ? 'active' : ''}`}
                onClick={() => setActiveTab(index)}
              >
                {child.props.label}
              </div>
            );
          }
          return null;
        })}
      </div>
      {children}
    </div>
  );
};
```

```

        >
        {child.props.label}
      </div>
    );
  }
  })}
</div>
<div className="tab-content">
  {React.Children.map(children, (child, index) => {
    if (index === activeTab) {
      return <>{child}</>;
    }
  })}
</div>
</div>
);
};

const Example: React.FC = () => {
  return (
    <Tabs>
      <Tab label="Tab 1">
        <div>Contenido de la pestaña 1</div>
      </Tab>
      <Tab label="Tab 2">
        <div>Contenido de la pestaña 2</div>
      </Tab>
      <Tab label="Tab 3">
        <div>Contenido de la pestaña 3</div>
      </Tab>
    </Tabs>
  );
};

export default Example;

```

## Render Props Pattern

The Render Props pattern is a technique in React that allows the sharing of code between components through a special prop that indicates a function. This pattern delegates the responsibility of rendering a certain part of the component to the function provided as a prop, thus allowing for greater reuse and flexibility in component composition.

## When to use

- When you need to share rendering logic between multiple components.
- To create highly customizable components that can adapt to different use cases.
- When you want to separate presentation logic from business logic in components.

## When not to use

- In cases where rendering logic is specific to a single component and will not be reused.
- When the Render Props pattern results in unnecessary complexity and makes the code harder to understand.

## Advantages

- Facilitates the reuse of rendering logic between components more flexibly than other patterns.
- Allows for greater customization and composition of components by delegating rendering logic to external functions.
- Encourages separation of concerns by separating presentation logic from business logic in components.

## Disadvantages

- Can introduce an additional layer of abstraction that makes understanding the data flow difficult.



- Requires a solid understanding of props and functions in React for proper implementation.

## Example

The Render Props pattern can be applied in a variety of ways, including error handling with Error Boundaries in React.

```
import React, { Component, ErrorInfo, ReactNode } from 'react';

interface ErrorBoundaryProps {
  renderError: (error: Error, errorInfo: ErrorInfo) => ReactNode;
}

interface ErrorBoundaryState {
  hasError: boolean;
  error: Error | null;
  errorInfo: ErrorInfo | null;
}

class ErrorBoundary extends Component<ErrorBoundaryProps, ErrorBoundaryState> {
  constructor(props: ErrorBoundaryProps) {
    super(props);
    this.state = {
      hasError: false,
      error: null,
      errorInfo: null,
    };
  }

  componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    this.setState({
      hasError: true,
      error: error,
      errorInfo: errorInfo,
    });
  }

  render() {
    const { renderError, children } = this.props;
    const { hasError, error, errorInfo } = this.state;
```

```

    if (hasError) {
      return renderError(error!, errorInfo!);
    }

    return children;
  }
}

const App: React.FC = () => {
  const renderError = (error: Error, errorInfo: ErrorInfo) => {
    return (
      <div>
        <h2>Something went wrong.</h2>
        <details style={{ whiteSpace: 'pre-wrap' }}>
          {error && error.toString()}
          <br />
          {errorInfo.componentStack}
        </details>
      </div>
    );
  };

  return (
    <ErrorBoundary renderError={renderError}>
      {/* Componentes envueltos por el ErrorBoundary */}
      <div>
        <h1>Welcome to My App</h1>
        <p>This is a sample application.</p>
        {/* Simulando un error */}
        <button onClick={() => { throw new Error('An unexpected error occurred.')
          Trigger Error
        }}>
          </button>
        </div>
      </ErrorBoundary>
    );
  };

  export default App;

```

In this example, we create an `ErrorBoundary` component that catches any errors thrown by its child components and handles them using the `componentDidCatch` method. When an error occurs, the `ErrorBoundary`

component renders the component provided by the render function (renderError), which displays a custom UI for the error.

The App component uses ErrorBoundary by wrapping its content and providing a render function to display information about the error should one occur within its child components.

This example illustrates how the Render Props pattern can be used to provide custom behavior for handling errors within a React application.

## **Control Props Pattern**

The Control Props pattern is a technique in React that allows a component to control its internal state through props provided by the parent component. Instead of the component handling its own state internally, it delegates control of the state to the parent component via props, allowing the parent to manipulate and control the state of the child component as needed.

### **When to use**

- When a component needs to be externally controlled and its state managed by higher-level components.
- To create flexible components that can be used in different contexts and controlled dynamically.
- In situations where bidirectional communication between components is required.

### **When not to use**

- When the component's state is purely internal and does not need to be controlled from outside.
- In cases where the Control Props pattern results in prop overload and unnecessary complexity.

## Advantages

- Provides greater control over the component's state from higher-level components.
- Enables clear and bidirectional communication between components in the React hierarchy.
- Facilitates component reuse by allowing them to be used in different application contexts.

## Disadvantages

- Can introduce excessive dependency between components, making data flow understanding difficult.
- Requires careful management of props and communication between components to avoid unexpected behavior.

## Example

Suppose we want to create a `Toggle` component that can be externally toggled on and off.

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

```

interface ToggleProps {
  value: boolean;
  onChange: (value: boolean) => void;
}

const Toggle: React.FC<ToggleProps> = ({ value, onChange }) => {
  const handleClick = () => {
    onChange(!value);
  };

  return (
    <button onClick={handleClick}>
      {value ? 'On' : 'Off'}
    </button>
  );
};

// Usage of the Toggle component controlled by props
const Example: React.FC = () => {
  const [isToggled, setIsToggled] = useState(false);

  const handleToggleChange = (value: boolean) => {
    setIsToggled(value);
  };

  return (
    <div>
      <h1>Control Props Example</h1>
      <Toggle value={isToggled} onChange={handleToggleChange} />
      <p>The current state is: {isToggled ? 'On' : 'Off'}</p>
    </div>
  );
};

export default Example;

```

The example showcases the Control Props pattern in React through a `Toggle` component. This component enables controlling the toggle state from the parent component. Upon clicking the toggle button, the `onChange` function provided by the parent component is invoked to update the toggle state. The `Example` component utilizes the `Toggle` component and maintains its state via the `useState` hook. When the toggle state changes, the

`handleToggleChange` function updates the state in the `Example` component. This demonstrates how the Control Props pattern allows a component to control its internal state through props provided by the parent component, enabling clear bidirectional communication between components.

## Props Getters Pattern

The Props Getters pattern is a technique in React that allows a child component to get and modify specific props from the parent component through special functions known as “props getters”. These functions are passed as arguments to child components and allow them to access specific props from the parent and, in some cases, modify them as needed.

### When to use

- Used when a child component is required to access or modify specific props of the parent component in a controlled manner.
- For cases where clear and predictable communication between components is required, especially in highly coupled components.
- When flexibility is needed to modify certain props of the parent within the child component.

### When not to use

- In situations where communication between components does not involve accessing or modifying parent-specific props.
- When the Props Getters pattern results in unnecessary complexity and makes it difficult to understand the data flow in the application.

## Advantages

- Provides a clear and controlled mechanism for child components to access and modify specific props of the parent.
- Allows clear and predictable communication between components, facilitating code maintenance and debugging.
- Provides flexibility to adapt the behavior of the child component based on the parent's props.

## Disadvantages

- May introduce explicit dependency between components, which can increase complexity and coupling.
- Requires careful design to ensure that props getters are used consistently and do not cause unexpected side effects.

## Example

Here is an example of how to implement the Props Getters pattern for a table that allows you to sort its columns in React.

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

interface Column {
  id: string;
  label: string;
  sortable: boolean;
}

interface TableProps {
  columns: Column[];
  data: any[];
}

interface TableHeaderProps {
```

```

    column: Column;
    onSort: (columnId: string) => void;
  }

  const TableHeader: React.FC<TableHeaderProps> = ({ column, onSort }) => {
    const handleSort = () => {
      if (column.sortable) {
        onSort(column.id);
      }
    };

    return (
      <th onClick={handleSort} style={{ cursor: column.sortable ? 'pointer' : 'default' }}>
        {column.label}
      </th>
    );
  };

  const Table: React.FC<TableProps> = ({ columns, data }) => {
    const [sortColumn, setSortColumn] = useState('');

    const handleSort = (columnId: string) => {
      setSortColumn(columnId);
      // Sorting logic would go here according to the selected column
    };

    return (
      <table>
        <thead>
          <tr>
            {columns.map(column => (
              <TableHeader key={column.id} column={column} onSort={handleSort} />
            ))}
          </tr>
        </thead>
        <tbody>
          {data.map((row, index) => (
            <tr key={index}>
              {columns.map(column => (
                <td key={column.id}>{row[column.id]}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    );
  };

  // Example usage
  const Example: React.FC = () => {

```



```
const columns: Column[] = [
  { id: 'name', label: 'Name', sortable: true },
  { id: 'age', label: 'Age', sortable: true },
  { id: 'country', label: 'Country', sortable: false },
];

const data = [
  { name: 'John', age: 30, country: 'USA' },
  { name: 'Alice', age: 25, country: 'Canada' },
  { name: 'Bob', age: 35, country: 'UK' },
];

return <Table columns={columns} data={data} />;
};

export default Example;
```

This example demonstrates a table with sortable columns. The `Table` component receives `columns` and `data` as props, while the `TableHeader` component handles the click on column headers to initiate the sorting process. The sorting state is maintained in the `Table` component. This example showcases how the Props Getters pattern can be utilized to enable a child component, such as `TableHeader`, to access specific functions from the parent component, in this case `Table`, to control the behavior of the table.

## State Initializer Pattern

The State Initializer pattern is a technique used in React to define and configure the initial state of a functional component. Instead of directly initializing the state within the component, a special function called a “state initializer” is used to define the initial state. This function is passed as an argument to the React state hook (`useState`), allowing for more flexible and dynamic setup of the component's initial state.

## When to use

- It is used when the initial state of the component depends on computed values or more complex logic.
- For cases where initializing the state is based on props or other states.
- When you want to keep the component cleaner and more modular by separating the state initialization logic from the rest of the component.

## When not to use

- In situations where the initial state of the component is static and does not need additional logic.
- When the State Initializer pattern results in unnecessary complexity and makes understanding the component difficult.

## Advantages

- Provides a clear and organized way to define the initial state of the component.
- Allows for more dynamic setup of the initial state, making it easier to adapt the component to different situations.
- Promotes modularity and code reuse by separating the state initialization logic from the rest of the component.

## Disadvantages

- Can introduce an additional layer of abstraction that makes it difficult to follow the component's data flow.

- Requires a good understanding of React hooks and state management in functional components.

## Example

Here's an example of the State Initializer Pattern in forms using a custom hook

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

// Interface definition for the form state
interface FormState {
  username: string;
  password: string;
}

// Custom hook to handle form state
const useFormState = (): [FormState, (e: React.ChangeEvent<HTMLInputElement>) => void] => {
  // Initial state of the form
  const initialFormState: FormState = {
    username: '',
    password: '',
  };

  // State hook for the form
  const [formState, setFormState] = useState<FormState>(initialFormState);

  // Function to handle changes in form fields
  const handleInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target;
    setFormState(prevState => ({
      ...prevState,
      [name]: value,
    }));
  };

  return [formState, handleInputChange];
};

// Example component using the form hook
const FormExample: React.FC = () => {
  // Using the custom hook to get the form state and function to handle changes
```

```
const [formState, handleInputChange] = useFormState();

// Function to handle form submission
const handleSubmit = (e: React.FormEvent) => {
  e.preventDefault();
  console.log('Form submitted:', formState);
};

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="username">Username:</label>
      <input
        type="text"
        id="username"
        name="username"
        value={formState.username}
        onChange={handleInputChange}
      />
    </div>
    <div>
      <label htmlFor="password">Password:</label>
      <input
        type="password"
        id="password"
        name="password"
        value={formState.password}
        onChange={handleInputChange}
      />
    </div>
    <button type="submit">Submit</button>
  </form>
);
};

export default FormExample;
```

In this example, the custom hook `useFormState` manages the form state. This hook returns an array with the form state and a function to handle changes in the form fields. The `FormExample` component uses this hook to get the form state and the function to handle changes, significantly simplifying the form logic and making it easier to maintain and understand.

## State Reducer Pattern

The State Reducer Pattern is a technique used in React to manage component state by delegating control of the state to a “reducer” similar to that used in Redux. This reducer is a function that receives an action and the current state, and returns the new state. This approach allows for centralizing state update logic in one place, which improves maintainability and scalability of the application.

### When to use

- It is used when more advanced state management is needed, especially in applications with complex state logic.
- For applications that benefit from predictability and traceability of state updates.
- When you want to centralize state update logic in one place.

### When not to use

- In small or simple applications where state management is not complex and a more direct approach is sufficient.
- When the State Reducer pattern results in excessive complexity and makes it difficult to understand the data flow in the application.

### Advantages

- Improves application maintainability by centralizing state update logic.
- Facilitates debugging and tracking of state changes by having a single place where updates occur.

- Promotes a more scalable and structured design of the application, especially in applications with a large amount of state logic.

## Disadvantages

- Can introduce initial overhead when implementing reducer logic and associated infrastructure.
- Requires a deeper understanding of Redux concepts and state management patterns.

## Example

```
import React, { useReducer } from 'react';

// Definition of the action type for the reducer
type Action =
  | { type: 'ADD_TODO'; payload: string }
  | { type: 'TOGGLE_TODO'; payload: number }
  | { type: 'REMOVE_TODO'; payload: number };

// Interface definition for the todo state
interface Todo {
  id: number;
  text: string;
  completed: boolean;
}

// Interface definition for the global state
interface State {
  todos: Todo[];
}

// Reducing function to handle actions and update state
const reducer = (state: State, action: Action): State => {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        ...state,
```

```

      todos: [
        ...state.todos,
        {
          id: state.todos.length + 1,
          text: action.payload,
          completed: false,
        },
      ],
    },
  ];
  case 'TOGGLE_TODO':
    return {
      ...state,
      todos: state.todos.map(todo =>
        todo.id === action.payload ? { ...todo, completed: !todo.completed } :
      ),
    };
  case 'REMOVE_TODO':
    return {
      ...state,
      todos: state.todos.filter(todo => todo.id !== action.payload),
    };
  default:
    return state;
}
};

// Example component using the State Reducer Pattern
const TodoList: React.FC = () => {
  // Use the useReducer hook to manage state with the defined reducer
  const [state, dispatch] = useReducer(reducer, { todos: [] });

  // Functions to handle user interactions
  const addTodo = (text: string) => {
    dispatch({ type: 'ADD_TODO', payload: text });
  };

  const toggleTodo = (id: number) => {
    dispatch({ type: 'TOGGLE_TODO', payload: id });
  };

  const removeTodo = (id: number) => {
    dispatch({ type: 'REMOVE_TODO', payload: id });
  };

  return (
    <div>
      <h2>Todo List</h2>
      <ul>
        {state.todos.map(todo => (
          <li key={todo.id}>

```

```

        <span
          style={{ textDecoration: todo.completed ? 'line-through' : 'none'
          onClick={() => toggleTodo(todo.id)}
        >
          {todo.text}
        </span>
        <button onClick={() => removeTodo(todo.id)}>Remove</button>
      </li>
    )}}
  </ul>
  <input
    type="text"
    placeholder="Add todo..."
    onKeyDown={(e) => {
      if (e.key === 'Enter' && e.currentTarget.value.trim() !== '') {
        addTodo(e.currentTarget.value.trim());
        e.currentTarget.value = '';
      }
    }}
  />
</div>
);
};

export default TodoList;

```

This example presents a todo list with functionalities to add, mark as completed, and remove todos. It uses the State Reducer Pattern to manage the state of the todo list. The reducer handles different types of actions to add, toggle state, and remove todos from the list. The `TodoList` component utilizes the `useReducer` hook to manage the global state of the todo list and renders the todo list with associated functionalities. This approach makes managing state and user interaction logic in the todo list easier.



Applying design patterns in development, especially in technologies like React with TypeScript, brings a series of significant benefits that directly impact the quality, maintainability, and scalability of software. When analyzing patterns such as State Reducer, Render Props, Control Props, and others mentioned, their value for modern application development becomes evident.

Firstly, these patterns offer structure and guidance to organize code more efficiently and cohesively. They provide proven solutions to common problems that arise in user interface development, helping developers avoid reinventing the wheel and adhere to solid development practices.

Moreover, applying design patterns promotes code reuse and modularity, facilitating software extension and maintenance as it evolves over time. The ability to decouple components and separate specific functional concerns enables more agile development and reduces error-prone coding.

Code clarity and readability are also significantly improved by using these patterns. By following established conventions and recognized patterns, code becomes more understandable to other developers who may work on the project in the future. This also facilitates collaboration in development teams and reduces the learning curve for new team members.

Another important aspect is the robustness and scalability that these patterns bring. By structuring the application consistently and predictably, complex situations can be handled more effectively, and the software can be scaled to meet growing business demands and requirements.

In summary, the design patterns in React represent a fundamental set of tools and approaches for developers looking to build modern and robust user interfaces. By understanding and applying these patterns properly, teams can maximize the quality and efficiency of their frontend projects, laying a solid foundation for the continued growth and evolution of their applications.

Don't forget to visit my [website](https://bryanaguilar.com).

**Bryan Aguilar**

Bryan Aguilar's portfolio and personal website. Systems Engineer | Creative Programmer | Enthusiastic Front-End...

[www.bryan-aguilar.com](https://www.bryan-aguilar.com)

Thank you for reading this article!

If you have any questions, don't hesitate to ask me. My inbox will always be open. Whether you have a question or just want to say hello, I will do my best to answer it!

[React](#)[Design Patterns](#)[Front End Development](#)[Code Efficiency](#)[Developer Community](#)



## Written by Bryan Aguilar

[Follow](#)

125 Followers

Systems Engineer | Creative Programmer | Frontend Developer | Enthusiastic Data Analyst  
& Scientist | Constant learner | <https://www.bryan-aguilar.com>

### More from Bryan Aguilar



Bryan Aguilar

### Modern API Development with Node.js, Express, and TypeScript...

Explore API dev with Node.js, Express & TypeScript, applying Clean Architecture &...

Apr 13 152 4



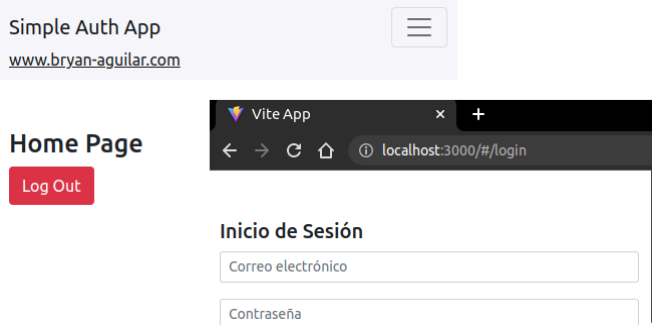
Bryan Aguilar


### Estructura base para cualquier proyecto de Angular

En este artículo te mostraré la estructura de directorios y archivos ideal para cualquier...

May 16, 2021 6






 Bryan Aguilar

**Autenticación simple con React y Redux Toolkit**

Hoy quiero compartir contigo, la manera de usar Redux Toolkit para un inicio de sesión e...

May 23, 2022  



 Bryan Aguilar

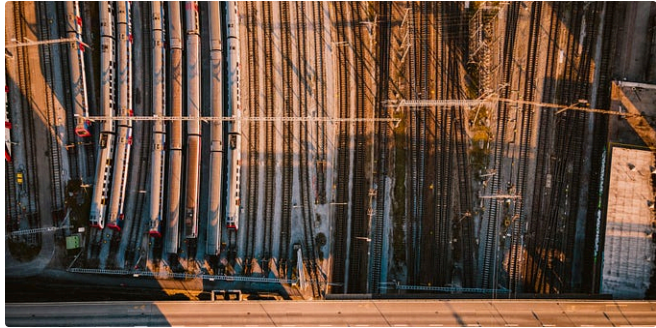
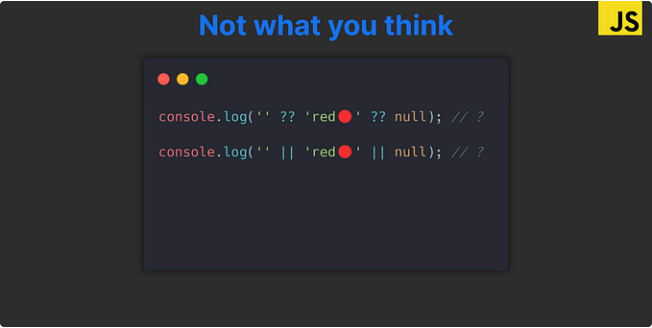
**Flutter, ¿Qué debes saber al crear, desplegar y mantener tu app?**

Este artículo te mostrará algunos aspectos que debes tener en cuenta a la hora de crear...

Jul 17, 2021  

See all from Bryan Aguilar

## Recommended from Medium





Tari Ibaba in Coding Beauty

## ?? vs || in JavaScript: The little-known difference

Learn it once and for all and avoid painful bugs down the line.



Jun 8



464



3



Andrew Zuo

## Async Await Is The Worst Thing To Happen To Programming

I recently saw this meme about async and await.



Jun 21



1.6K



110



### Lists



#### Stories to Help You Grow as a Software Developer

19 stories · 1190 saves



#### General Coding Knowledge

20 stories · 1375 saves



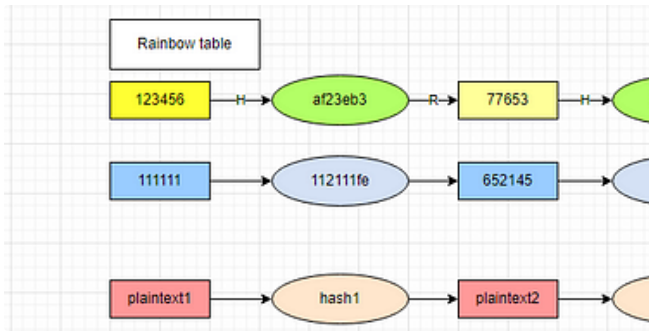
#### Medium's Huge List of Publications Accepting...

312 stories · 3078 saves



#### Natural Language Processing

1567 stories · 1117 saves

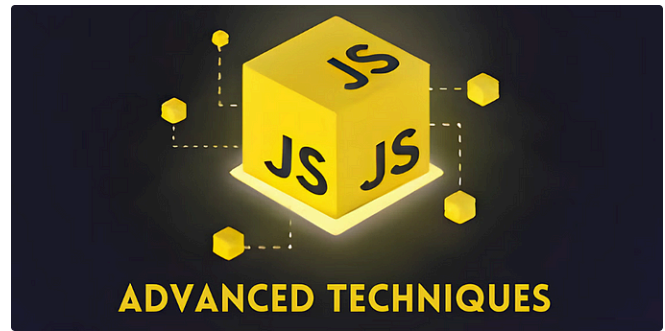



 LORY

## A basic question in security Interview: How do you store...

Explained in 3 mins.

★ May 12 🖱 4K 💬 42  ...

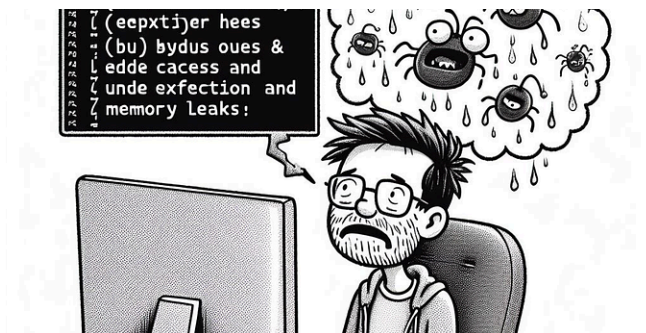



 Deepak Chaudhari

## Advanced JavaScript Concepts: 2024

Description: Uncover the intricacies of advanced JavaScript concepts, from nested...

Jan 18 🖱 1.3K 💬 6  ...

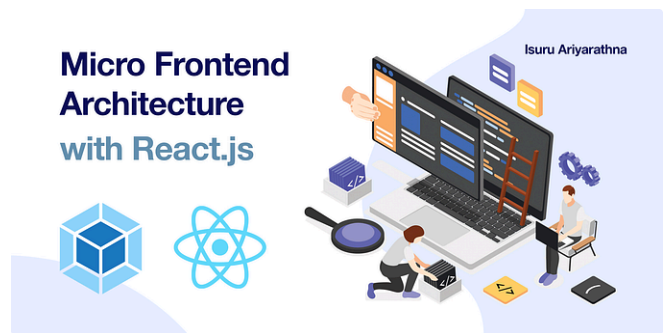



 Daniel Craciun in Level Up Coding

## The Dark Side of useEffect in React

One Mistake away from Disaster

★ Mar 3 🖱 491 💬 8  ...



 Isuru Ariyaratna

## A Deep Dive into Micro Frontend Architecture with React.js

Discover the power of Micro Frontend Architecture in React.js! Learn how to build...

Apr 11 🖱 243 💬 4  ...

[See more recommendations](#)