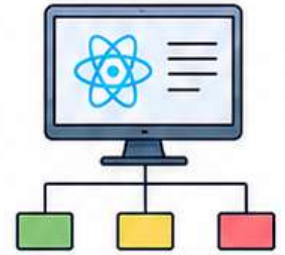


# Chapter 19: Advanced React Development

## 19.1 Advanced Component Architecture

### 19.1.1 What is Advanced Component Architecture?

- \* As applications grow, simple component structures are not enough.
- \* Advanced component architecture helps in building **scalable**, **maintainable** and **reusable** React applications.
- \* It involves organizing components in a way that improves **separation of concerns**, state management and performance.
- \* Good architecture makes the codebase **easy to understand**, **test** and **extend**.



### 19.1.2 Principles of Advanced Component Architecture

#### 1. Component Reusability



Create reusable components to avoid duplication.

#### 2. Separation of Concerns



Each component should have a single responsibility.

#### 3. Composability



Build complex UIs by composing small, independent components.

#### 4. Scalability



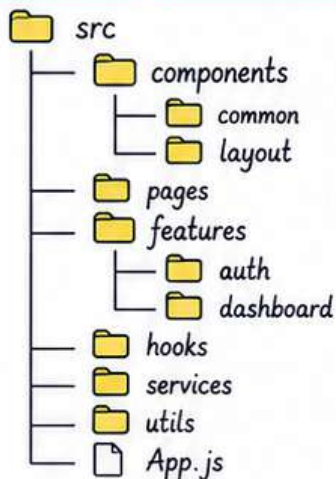
Architecture should support growth and changes easily.

#### 5. Maintainability



Well-structured code is easier to debug and maintain.

### 19.1.3 Typical Component Architecture (Folder Structure)



- \* components → Reusable UI components.
- \* common → Buttons, Inputs, Modals, etc.
- \* layout → Header, Footer, Sidebar, etc.
- \* pages → Page-level components.
- \* features → Feature-specific components, logic and related files.
- \* hooks → Custom React hooks.
- \* services → API calls and external services.
- \* utils → Helper functions and utilities.
- \* App.js → Root component.



#### Key Points:

- ✓ A well-organized architecture improves developer productivity.
- ✓ Break UI into small, reusable and independent components.
- ✓ Keep business logic separate from UI components.
- ✓ Follow consistent naming conventions and folder structure.
- ✓ Always think about scalability and maintainability.



#### Key Takeaway:



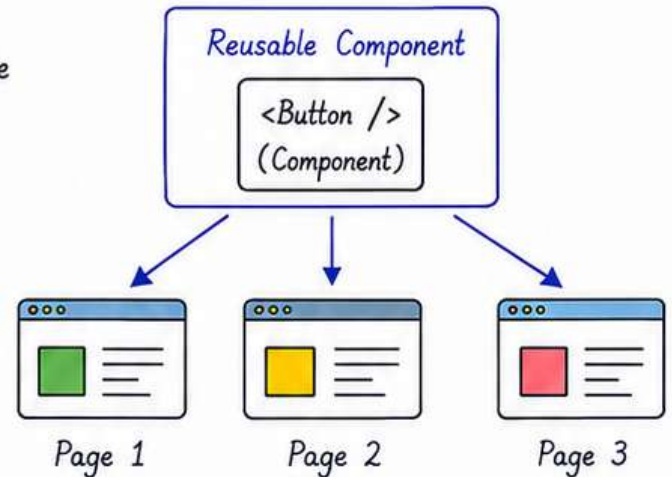
Advanced component architecture is the foundation of large-scale React applications. It helps in writing **clean**, **reusable** and **scalable** code that is easy to maintain and extend.



# 19.2 Reusable Components

## 19.2.1 What are Reusable Components?

- \* Reusable components are the **building blocks** that can be used in multiple places throughout the application.
- \* They help in **reducing code duplication** and improve **consistency** in the UI.
- \* By passing **props**, we can **customize** the component according to our needs.
- \* Reusable components **save time** and make the code **easy to maintain**.



## 19.2.2 Benefits of Reusable Components

### 1. Avoids Code Duplication



Write once, use anywhere.

### 2. Consistency



Maintain a uniform look and feel.

### 3. Easy Maintenance



Update in one place reflects everywhere.

### 4. Improved Productivity



Build faster with reusable building blocks.

### 5. Scalability



Easier to manage large applications.

## 19.2.3 Creating a Reusable Component

Example: Button Component

```
import React from 'react';
function Button({ text, onClick }) {
  return (
    <button className="btn" onClick={onClick}>
      {text}
    </button>
  );
}
export default Button;
```

Explanation:

- \* This component accepts props (text and onClick).
- \* It can be used anywhere in the application.
- \* Props make the component dynamic and reusable.

Usage:

```
import Button from './Button';
function App() {
  return (
    <div>
      <Button text="Submit" onClick={handleSubmit} />
      <Button text="Cancel" onClick={handleCancel} />
    </div>
  );
}
```



## 19.2.4 Best Practices

- \* Keep components small and focused on a **single responsibility**.
- \* Use **meaningful names** for components and props.
- \* Use props for **dynamic data**, not state (unless necessary).
- \* Avoid hardcoding values, make components **flexible**.
- \* Organize reusable components in a **separate folder**.



### Key Points:

- ✓ Reusable components reduce duplication.
- ✓ They improve consistency and maintainability.
- ✓ Use props to make components dynamic.
- ✓ Follow best practices for better scalability.
- ✓ Small components lead to clean architecture.



### Key Takeaway:



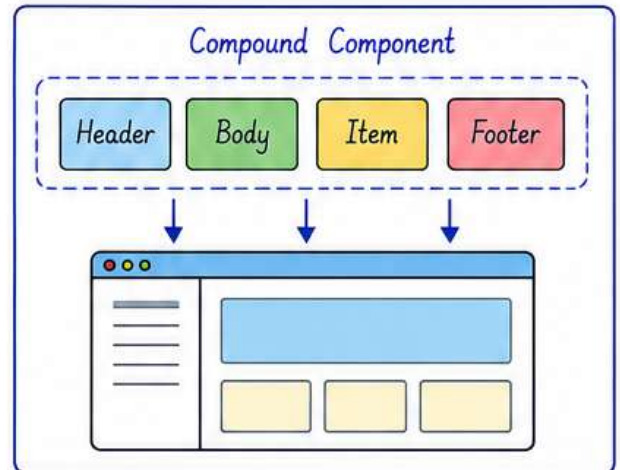
Reusable components are essential for building **scalable** and **maintainable** React applications. They save time, ensure **consistency** and make the development process **efficient** and enjoyable.



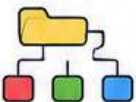
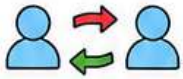
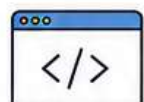
## 19.3 Compound Components

### 19.3.1 What are Compound Components?

- \* Compound components work together as a **group** to form a complete UI.
- \* They share **implicit state** and communicate with each other behind the scenes.
- \* They are designed to be used together and provide a **clean** and **simple API**.
- \* **Examples:** Tabs, Accordions, Dropdowns, Menus, etc.



### 19.3.2 Benefits of Compound Components

|                                                                                                                                                                             |                                                                                                                                                                                  |                                                                                                                                                                             |                                                                                                                                                                       |                                                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>1. Organized Structure</p>  <p>Keeps related parts together in a logical structure.</p> | <p>2. Easier State Sharing</p>  <p>Components share state implicitly without prop drilling.</p> | <p>3. Simplified API</p>  <p>Expose a simple and consistent interface for complex UIs.</p> | <p>4. Reusability</p>  <p>Use the same compound component in different places.</p> | <p>5. Maintainability</p>  <p>Easier to understand, debug and extend over time.</p> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 19.3.3 Example: Tabs using Compound Components

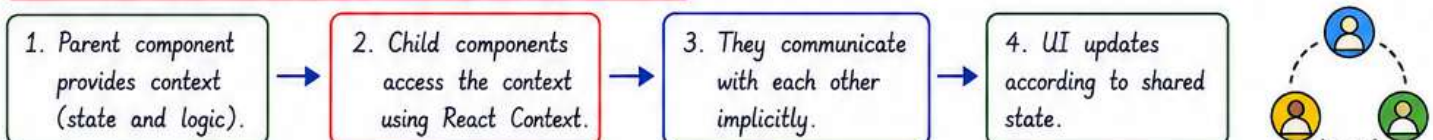
```
<Tabs>
  <Tabs.List>
    <Tabs.Tab>Profile</Tabs.Tab>
    <Tabs.Tab>Settings</Tabs.Tab>
  </Tabs.List>
  <Tabs.Panel>Profile Content</Tabs.Panel>
  <Tabs.Panel>Settings Content</Tabs.Panel>
</Tabs>
```

#### Explanation:

- \* Tabs is the root component.
- \* Tabs.List holds the list of tabs.
- \* Tabs.Tab defines individual tabs.
- \* Tabs.Panel shows content for the active tab.



### 19.3.4 How Compound Components Work?



### 19.3.5 Best Practices

- \* Use compound components for related UI parts.
- \* Keep internal implementation **hidden** from users.
- \* Use React **Context** for managing shared state.
- \* Provide a **simple** and **intuitive** API.
- \* Ensure components are **flexible** and **extensible**.



#### Key Points:

- ✓ Compound components work as a group.
- ✓ They share implicit state and logic.
- ✓ They provide a clean and simple API.
- ✓ They improve reusability and maintainability.
- ✓ Best for complex, interactive UIs.



#### Key Takeaway:



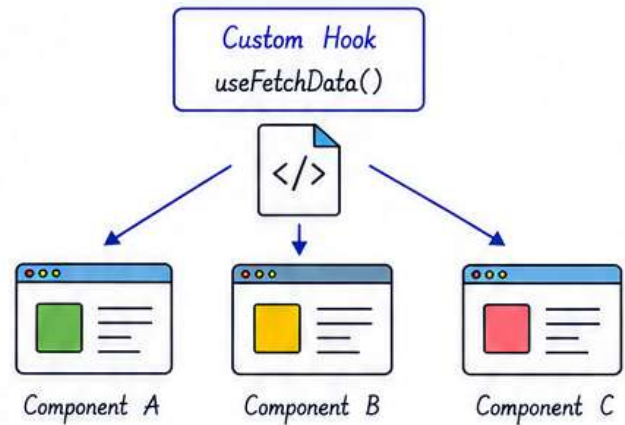
Compound components help us build **complex UIs** by combining multiple components that work together and share state implicitly. They provide a **clean API** and make our code more **maintainable** and **reusable**.



# 19.4 Custom Hooks

## 19.4.1 What are Custom Hooks?

- \* Custom Hooks are JavaScript functions whose names start with "use".
- \* They let us **extract** and **reuse** stateful logic across multiple components.
- \* They help in keeping components **clean**, **small** and **focused** on UI.
- \* React provides built-in Hooks (useState, useEffect, etc.), but we can **create our own Hooks** for specific logic.



## 19.4.2 Benefits of Custom Hooks

### 1. Code Reusability



Reuse the same logic in multiple components without duplication.

### 2. Separation of Concerns



Keep UI and logic separate for better readability.

### 3. Cleaner Components



Components become smaller and focus only on rendering UI.

### 4. Easier Maintenance



Update logic in one place and it reflects everywhere.

### 5. Better Testing



Custom Hooks can be tested independently from components.

## 19.4.3 Creating a Custom Hook

```
import { useState, useEffect } from 'react';
function useFetchData(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    fetch(url)
      .then(res => res.json())
      .then(json => {
        setData(json);
        setLoading(false);
      })
      .catch(err => {
        setError(err);
        setLoading(false);
      });
  }, [url]);
  return { data, loading, error };
}
```

### Explanation:

- \* This custom hook fetches data from the given URL.
- \* It returns an object containing **data**, **loading** and **error**.
- \* Any component can use this hook to get data.

### Usage in a Component:

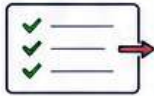
```
import useFetchData from './useFetchData';
function Users() {
  const { data, loading, error } = useFetchData('https://api.example.com/users');
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;
  return (
    <ul>
      {data.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

## 19.4.4 Rules for Custom Hooks

1. Start the name with "use".

useMyHook

2. Call Hooks only at the top level.



3. Call Hooks only from React functions (components or other Hooks).



4. Do not call Hooks inside loops, conditions or nested functions.



### Key Points:

- ✓ Custom Hooks start with "use".
- ✓ They help in sharing logic.
- ✓ They make components clean and maintainable.
- ✓ They follow the same rules as built-in Hooks.



### Key Takeaway:



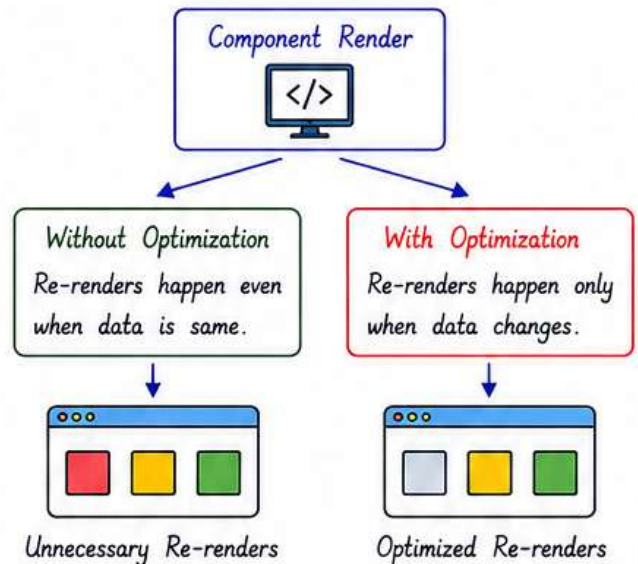
Custom Hooks allow us to extract, reuse and share stateful logic across components. They improve code **reusability**, **maintainability** and make our React applications more organized and scalable.



# 19.5 Render Optimization

## 19.5.1 What is Render Optimization?

- \* Render optimization is the process of **reducing unnecessary re-renders** in React components.
- \* It improves the **performance** of the application by updating only the parts of the UI that actually changed.
- \* Optimized rendering leads to **faster, smoother** and more **responsive** applications.
- \* It reduces the load on the browser and provides a **better user experience**.

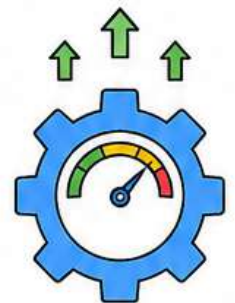


## 19.5.2 Benefits of Render Optimization

|                                                                                                |                                                                                               |                                                                                    |                                                                                                       |                                                                                                    |
|------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <b>1. Improved Performance</b><br><br>Reduces unnecessary re-renders and makes the app faster. | <b>2. Better User Experience</b><br><br>Smooth and responsive UI increases user satisfaction. | <b>3. Reduced CPU Usage</b><br><br>Less rendering means less work for the browser. | <b>4. Scalable Applications</b><br><br>Optimized components handle large data and complex UIs easily. | <b>5. Better Code Quality</b><br><br>Encourages writing clean, efficient and well-structured code. |
|------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|

## 19.5.3 Techniques for Render Optimization

- React.memo()** → Prevents re-rendering of functional components if props are same.
- useMemo()** → Memoizes expensive calculations and returns the cached result.
- useCallback()** → Memoizes functions so that they don't change on every render.
- Lazy Loading** → Loads components only when needed.
- Avoid Inline Functions** → Define functions outside JSX to prevent re-creation.
- Proper Key Usage** → Use stable and unique keys in lists.
- Avoid Unnecessary State Updates** → Update state only when value actually changes.



## 19.5.4 Best Practices

- \* Use **React.memo()** for pure functional components.
- \* Use **useMemo()** and **useCallback()** wisely.
- \* Keep component state as local as possible.
- \* Split large components into smaller components.
- \* Avoid passing new objects or functions as props unnecessarily.
- \* Test performance using React Developer Tools.



### Key Points:

- ✓ Optimization reduces unnecessary re-renders.
- ✓ It improves performance and user experience.
- ✓ Use memoization techniques correctly.
- ✓ Avoid unnecessary state updates and inline functions.
- ✓ Always measure performance before and after optimization.



### Key Takeaway:



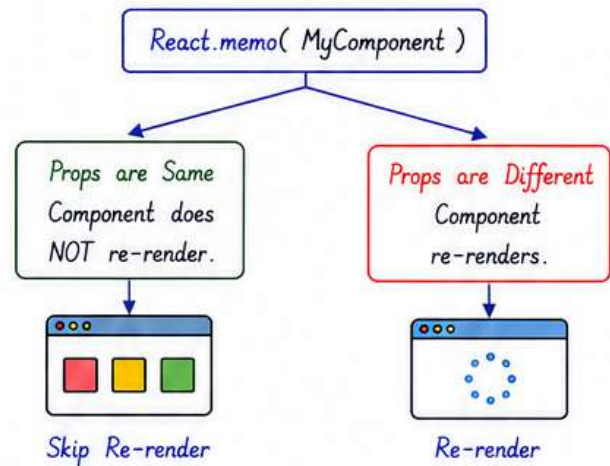
Render optimization is essential for building **high-performance** React applications. By applying the right techniques, we can reduce unnecessary renders, improve **speed**, and deliver a **smooth** and **responsive** user experience.



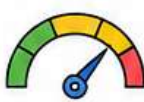
## 19.6 React.memo

### 19.6.1 What is React.memo?

- \* React.memo is a higher-order component used to **memoize** functional components.
- \* It prevents unnecessary re-renders when the props passed to a component are the same.
- \* It performs a **shallow comparison** of props by default.
- \* It is similar to **PureComponent** in class components.



### 19.6.2 Benefits of React.memo

|                                                                                                                                                                               |                                                                                                                                                                                   |                                                                                                                                                                                |                                                                                                                                                                            |                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>1. Prevents Unnecessary Re-renders</b><br><br>Re-renders only when props actually change. | <b>2. Improves Performance</b><br><br>Reduces the rendering work and improves application speed. | <b>3. Better User Experience</b><br><br>Smoother UI updates lead to better user satisfaction. | <b>4. Optimized Rendering</b><br><br>Helpful in large applications with many components. | <b>5. Easy to Use</b><br><br>Simple to implement and requires minimal changes in code. |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 19.6.3 Syntax

```
const MemoizedComponent = React.memo(MyComponent);

// or simply
export default React.memo(MyComponent);
```

Explanation:

- \* MyComponent will re-render only when the name prop changes.
- \* Clicking the button changes state in App but does not re-render MyComponent.



### 19.6.4 Example

```
// MyComponent.js
import React from 'react';

function MyComponent({ name }) {
  console.log("MyComponent Rendered");
  return <p>Hello, {name}!</p>;
}

export default React.memo(MyComponent);
```

```
// App.js
import React, { useState } from 'react';
import MyComponent from './MyComponent';

function App() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h3>Count: {count}</h3>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
      <MyComponent name="React" />
    </div>
  );
}

export default App;
```

### 19.6.5 React.memo vs Normal Component

| Feature          | Normal Component                      | Component with React.memo                      |
|------------------|---------------------------------------|------------------------------------------------|
| Re-render        | Re-renders on every parent re-render. | Re-renders only when props change.             |
| Performance      | Can be slower in large applications.  | Faster and optimized rendering.                |
| Props Comparison | Not checked.                          | Performs shallow comparison of props.          |
| Use Case         | Simple components or small apps.      | Large apps or performance-critical components. |

#### Key Points:

- ✓ React.memo is used to optimize functional components.
- ✓ It prevents unnecessary re-renders.
- ✓ It performs a shallow comparison of props.
- ✓ It improves performance and user experience.
- ✓ Use it for components that receive the same props frequently.
- ✓ Do not overuse; use only where needed.



#### Key Takeaway:



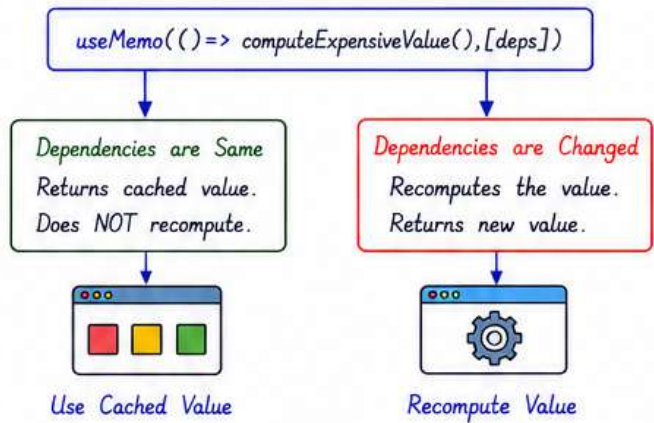
React.memo helps in **optimizing** rendering performance by ensuring that components re-render only when their **props change**. It is a simple yet powerful tool for building **faster** and more **efficient** React applications.



# 19.7 useMemo

## 19.7.1 What is useMemo?

- \* useMemo is a React Hook that returns a **memoized value**.
- \* It is used to memoize the result of **expensive calculations** and return the **cached value**.
- \* The memoized value is recomputed only when its **dependencies** change.
- \* It helps in **improving** the performance by avoiding unnecessary recalculations.



## 19.7.2 Benefits of useMemo

- 1. Improves Performance**  
Avoids expensive recalculations on every render. (Illustrated with a speedometer icon)
- 2. Optimizes Rendering**  
Reduces CPU usage by using cached computed values. (Illustrated with a bar chart icon)
- 3. Better User Experience**  
Smoother UI updates and faster response times. (Illustrated with a user icon and stars)
- 4. Useful for Expensive Operations**  
Ideal for heavy calculations, filtering, sorting, etc. (Illustrated with a gear icon)
- 5. Prevents Unnecessary Work**  
Re-computes value only when dependencies actually change. (Illustrated with a circular arrow icon)

## 19.7.3 Syntax

```
const memoizedValue = useMemo(  
  () => computeExpensiveValue,  
  [dependency1, dependency2, ...]  
);
```

### Explanation:

- \* `computeExpensiveValue` is the function that returns the value.
- \* The function is executed only when any dependency changes.
- \* Otherwise, the cached value is returned.

## 19.7.4 Example

```
// ExpensiveCalculation.js  
import React, { useMemo } from 'react';  
  
function ExpensiveCalculation({ numbers }) {  
  const sum = useMemo(() => {  
    console.log("Calculating sum...");  
    return numbers.reduce((acc, num) => acc + num, 0);  
  }, [numbers]); // Recompute only when 'numbers' change  
  
  return (  
    <div>  
      <p>Sum: {sum}</p>  
    </div>  
  );  
}  
  
export default ExpensiveCalculation;
```

```
// App.js  
import React, { useState } from 'react';  
import ExpensiveCalculation from './ExpensiveCalculation';  
  
function App() {  
  const [count, setCount] = useState(0);  
  const numbers = [1, 2, 3, 4, 5];  
  return (  
    <div>  
      <h3>useMemo Example</h3>  
      <button onClick={() => setCount(count + 1)}>  
        Increment Count  
      </button>  
      <p>Count: {count}</p>  
      <ExpensiveCalculation numbers={numbers} />  
    </div>  
  );  
}  
  
export default App;
```

## 19.7.5 When to Use useMemo?

- \* When performing **expensive calculations**.
- \* When **filtering, sorting** or transforming large data sets.
- \* When passing **computed values** to child components to prevent unnecessary re-renders.
- \* When you want to **cache results** and reuse them across renders.

### Key Points:

- ✓ `useMemo` returns a **memoized (cached) value**.
- ✓ It recomputes the value only when **dependencies** change.
- ✓ It is used to optimize **expensive calculations**.
- ✓ It improves performance and reduces unnecessary work.
- ✓ Do not use it for small or cheap calculations.



## 19.7.6 useMemo vs Normal Calculation

| Feature        | Normal Calculation                   | useMemo                            |
|----------------|--------------------------------------|------------------------------------|
| Re-computation | Runs on every render                 | Runs only when dependencies change |
| Performance    | Can be slow for expensive operations | Faster due to caching              |
| Caching        | No caching                           | Uses cached value                  |
| Use Case       | Simple calculations                  | Expensive or heavy calculations    |
| Hook/Method    | Direct calculation in component      | React Hook: <code>useMemo()</code> |



### Key Takeaway:



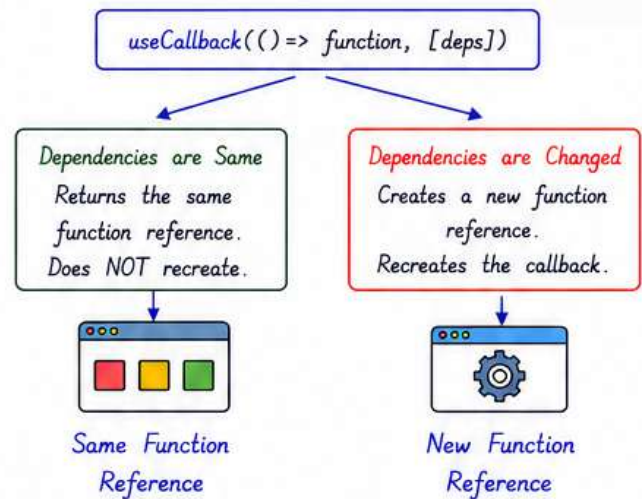
`useMemo` is a powerful Hook for **optimizing performance** by memoizing expensive calculations. It ensures that the computation is done **only when necessary**, making your React applications **faster** and more **efficient**.



# 19.8 useCallback

## 19.8.1 What is useCallback?

- \* `useCallback` is a React Hook that returns a **memoized callback function**.
- \* It is used to **prevent unnecessary re-creations** of functions on every render.
- \* The function is recreated only when its **dependencies change**.
- \* It is useful when passing callbacks to child components that rely on reference equality (like `React.memo`).



## 19.8.2 Benefits of useCallback

1. Prevents Unnecessary Re-creations



The callback function is recreated only when dependencies change.

2. Improves Performance



Reduces unnecessary re-renders in child components and improves speed.

3. Better with Memoized Components



Works great with `React.memo` or Pure Component.

4. Stable Function Reference



Provides a stable function reference, preventing unwanted updates.

5. Cleaner and Efficient Code



Helpful in large applications to maintain performance and code quality.

## 19.8.3 Syntax

```
const memoizedCallback = useCallback(  
  () => {  
    // function logic  
  },  
  [dependency1, dependency2, ...]  
);
```

### Explanation:

- \* The function is returned and cached.
- \* It is recreated only when any dependency in the array changes.
- \* Otherwise, the same function reference is returned.



## 19.8.4 Example

```
// ChildComponent.js  
import React from 'react';
```

```
const ChildComponent = React.memo(({ onClick }) => {  
  console.log('Child rendered');  
  return (  
    <button onClick={onClick}>Click Me</button>  
  );  
});  
  
export default ChildComponent;
```

```
// App.js  
import React, { useState, useCallback } from 'react';  
import ChildComponent from './ChildComponent';
```

```
function App() {  
  const [count, setCount] = useState(0);  
  
  const handleClick = useCallback(() => {  
    console.log('Button clicked');  
  }, []); // will not change on re-render  
  
  return (  
    <div>  
      <h3>useCallback Example</h3>  
      <p>Count: {count}</p>  
      <button onClick={handleClick}>Increment Count</button>  
      <ChildComponent onClick={handleClick} />  
    </div>  
  );  
}  
  
export default App;
```

## 19.8.5 When to Use useCallback?

- \* When passing callback functions to child components that are memoized using `React.memo`.
- \* When the callback function is used in dependency arrays of other Hooks (e.g., `useEffect`).
- \* When you want to avoid unnecessary function re-creations on every render.
- \* When the function is used in performance-critical components.



### Key Points:

- ✓ `useCallback` returns a memoized (cached) callback function.
- ✓ The function is recreated only when dependencies change.
- ✓ It helps prevent unnecessary re-renders in child components.
- ✓ It works well with `React.memo` and Pure Components.
- ✓ It improves performance and keeps function references stable.



## 19.8.6 useCallback vs Normal Function

| Feature     | Normal Function                  | <code>useCallback</code>                   |
|-------------|----------------------------------|--------------------------------------------|
| Re-creation | Created on every render          | Created only when dependencies change      |
| Reference   | New reference on every render    | Same reference if dependencies are same    |
| Performance | May cause unnecessary re-renders | Optimized, prevents unnecessary re-renders |
| Best For    | Simple or small components       | Large apps or memoized child components    |
| Hook/Method | Direct function                  | React Hook: <code>useCallback()</code>     |



### Key Takeaway:



`useCallback` helps in **optimizing performance** by memoizing callback functions. It ensures that functions are recreated **only when necessary**, reducing unnecessary re-renders and improving the efficiency of React applications.

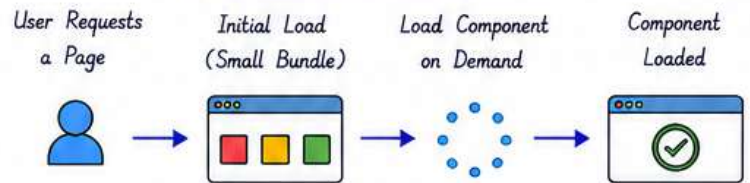


# 19.9 Lazy Loading

## 19.9.1 What is Lazy Loading?

- \* Lazy Loading is a technique to load components **only when they are needed**.
- \* It helps in **reducing the initial bundle size** of the application.
- \* It improves performance by loading code **on-demand**.
- \* React provides **React.lazy()** and **Suspense** to implement lazy loading.

### Lazy Loading Flow



Only required components are loaded, reducing load time and bandwidth.

## 19.9.2 Benefits of Lazy Loading

### 1. Faster Initial Load



Reduces the initial bundle size, so the app loads faster.

### 2. Improved Performance



Loads only the code that is needed, improving performance.

### 3. Better User Experience



Users see the app faster and experience smoother navigation.

### 4. Reduced Bandwidth



Saves bandwidth by loading code only when necessary.

### 5. Scalable Applications



Makes large applications more manageable and scalable.

## 19.9.3 Syntax

```
// 1. Import React and required hooks
import React, { lazy, Suspense } from 'react';
// 2. Lazy load a component
const MyComponent = lazy(() => import('./MyComponent'));
// 3. Use Suspense to show fallback UI while loading
function App() {
  return (
    <div>
      <h2>Lazy Loading Example</h2>
      <Suspense fallback={<p>Loading component...</p>}>
        <MyComponent />
      </Suspense>
    </div>
  );
}
export default App;
```

Explanation:

- \* **React.lazy()** is used to dynamically import a component.
- \* **Suspense** is used to show a fallback UI while the component is loading.
- \* The component is loaded only when it is rendered.



## 19.9.4 Example

```
// MyComponent.js
import React from 'react';
function MyComponent() {
  return (
    <div style={{ border: '1px solid #4CAF50', padding: '10px' }}>
      <h3>My Component</h3>
      <p>This component is loaded lazily.</p>
    </div>
  );
}
export default MyComponent;
```

```
// App.js
import React, { lazy, Suspense } from 'react';
// Lazy load MyComponent
const MyComponent = lazy(() => import('./MyComponent'));
function App() {
  return (
    <div>
      <h2>Lazy Loading Example</h2>
      <Suspense fallback={<p>Loading component...</p>}>
        <MyComponent />
      </Suspense>
    </div>
  );
}
export default App;
```

### Key Points:

- ✓ Lazy Loading loads components only when they are needed.
- ✓ It reduces initial bundle size and improves performance.
- ✓ **React.lazy()** is used to dynamically import components.
- ✓ **Suspense** is used to handle loading state with fallback UI.
- ✓ Use Lazy Loading for large components and route-based code splitting.



## 19.9.5 When to Use Lazy Loading?

- \* When you have large components that are not needed on initial load.
- \* When components are used in routes or on specific user actions.
- \* When you want to improve the performance of large applications.



## 19.9.6 Lazy Loading vs Normal Import

| Feature        | Normal Import                              | Lazy Loading                                             |
|----------------|--------------------------------------------|----------------------------------------------------------|
| Loading Time   | All components are loaded at once.         | Components are loaded on demand.                         |
| Bundle Size    | Larger initial bundle size.                | Smaller initial bundle size.                             |
| Performance    | Slower initial load.                       | Faster initial load.                                     |
| Use Case       | Small applications or critical components. | Large applications or non-critical components.           |
| Implementation | import MyComponent from './MyComponent';   | const MyComponent = lazy(() => import('./MyComponent')); |



### Key Takeaway:



Lazy Loading helps in **optimizing** React applications by loading components only when needed. It **reduces** initial load time, saves bandwidth, and provides a **better user experience**.

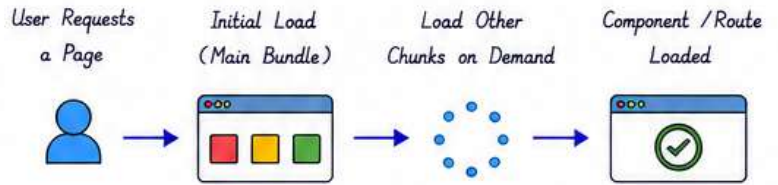


# 19.10 Code Splitting

## 19.10.1 What is Code Splitting?

- \* Code Splitting is a technique to split your code into **smaller chunks (bundles)** that can be loaded **on demand**.
- \* Instead of loading the entire application code at once, only the **required parts** are loaded.
- \* It helps in **reducing the initial bundle size** and improves the performance of the application.
- \* React supports code splitting using **React.lazy()**, **Suspense** and dynamic **import()**.

### How Code Splitting Works?



Only the necessary code is loaded initially. Other parts are loaded only when needed.

## 19.10.2 Benefits of Code Splitting

### 1. Smaller Initial Bundle



Reduces the initial bundle size, so the app loads faster.

### 2. Improved Performance



Loads only the code that is needed, when it is needed.

### 3. Better User Experience



Users get faster load times and smoother navigation.

### 4. Efficient Resource Usage



Reduces unnecessary data transfer and improves resource utilization.

### 5. Scalable Applications



Helps in building scalable applications with better maintainability.

## 19.10.3 Syntax

```
// 1. Lazy load a component using React.lazy()
const MyComponent = React.lazy(() => import('./MyComponent'));

// 2. Use Suspense to show fallback UI
import { Suspense } from 'react';
function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <MyComponent />
    </Suspense>
  );
}
```

### Explanation:

- \* **React.lazy()** is used to dynamically import a component.
- \* **Suspense** is used to display a fallback UI while the component is being loaded.
- \* The component is loaded only when it is rendered.



## 19.10.4 Example

```
// MyComponent.js
import React from 'react';

function MyComponent() {
  return (
    <div style={{ border: '1px solid #4CAF50', padding: '10px' }}>
      <h3>My Component</h3>
      <p>This component is loaded using code splitting.</p>
    </div>
  );
}

export default MyComponent;
```

```
// App.js
import React, { Suspense } from 'react';

const MyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <div>
      <h2>Code Splitting Example</h2>
      <Suspense fallback={<p>Loading component...</p>}>
        <MyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

```
// App with Route-based Code Splitting (React Router)
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = React.lazy(() => import('./pages/Home'));
const About = React.lazy(() => import('./pages/About'));

function App() {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </Suspense>
    </Router>
  );
}

export default App;
```

### Key Points:

- ✓ Code Splitting splits your code into smaller chunks.
- ✓ It loads only the required code on demand.
- ✓ **React.lazy()** and **import()** are used for code splitting.
- ✓ **Suspense** helps in showing fallback UI while loading.
- ✓ It improves performance and user experience.
- ✓ Use code splitting for large or infrequently used modules.



## 19.10.5 When to Use Code Splitting?

- \* When your application has large components or libraries.
- \* When certain features are used rarely.
- \* When you want to improve the initial load time.
- \* When using route-based or feature-based modules.
- \* When building large-scale React applications.



## 19.10.6 Code Splitting vs Normal Import

| Feature             | Normal Import                            | Code Splitting                                                 |
|---------------------|------------------------------------------|----------------------------------------------------------------|
| Loading             | All code is loaded at once.              | Code is loaded only when needed.                               |
| Initial Bundle Size | Larger initial bundle size.              | Smaller initial bundle size.                                   |
| Performance         | Slower initial load.                     | Faster initial load.                                           |
| User Experience     | May cause slower navigation.             | Smooth and fast navigation.                                    |
| Use Case            | Small apps or simple projects.           | Large apps or complex projects.                                |
| Implementation      | import MyComponent from './MyComponent'; | const MyComponent = React.lazy(() => import('./MyComponent')); |



### Key Takeaway:



Code Splitting is an important optimization technique in React applications. It **reduces the initial bundle size**, loads code on demand, and provides a **better performance and user experience**.

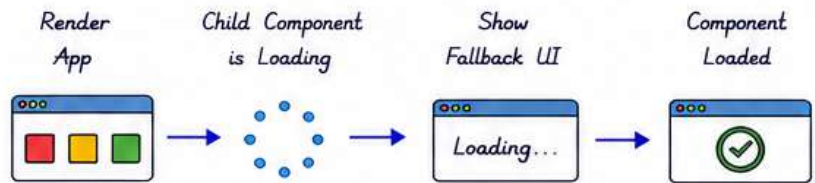


# 19.11 Suspense

## 19.11.1 What is Suspense?

- \* Suspense is a React component that lets you "wait" for something before rendering.
- \* It is used to show a **fallback UI** while the child components are loading.
- \* It works with **React.lazy()** for code splitting and with data fetching libraries.
- \* It improves user experience by avoiding blank screens and showing a meaningful fallback while loading.

### How Suspense Works?



Suspense shows the fallback until the component or data is ready, then it shows the real content.

## 19.11.2 Benefits of Suspense

### 1. Better User Experience



Users see a fallback instead of a blank screen while loading.

### 2. Cleaner Code



Simplifies loading states by handling them in one place.

### 3. Works with Code Splitting & Data Fetching



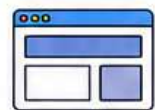
Seamlessly works with **React.lazy()** and data fetching libraries.

### 4. Improved Performance



Loads only what is needed and shows fallback until ready.

### 5. Prevents Layout Shift



Fallback UI keeps the layout stable while content loads.

## 19.11.3 Syntax

```
<Suspense fallback={<FallbackComponent />}>
  <LazyComponent />
</Suspense>
```

### Explanation:

- \* **fallback**: The UI to show while the child component is loading.
- \* **LazyComponent**: A component loaded lazily using **React.lazy()** or a component that suspends while fetching data.

## 19.11.4 Example

```
// MyComponent.js
import React from 'react';

function MyComponent() {
  return (
    <div style={{ border: '1px solid #4CAF50', padding: '10px' }}>
      <h3>My Component</h3>
      <p>This component is loaded using Suspense.</p>
    </div>
  );
}

export default MyComponent;
```

```
// App.js
import React, { Suspense } from 'react';

const LazyComponent = React.lazy(() =>
  import('./MyComponent'));

function App() {
  return (
    <div>
      <h2>Suspense Example</h2>
      <Suspense fallback={<p>Loading component...</p>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

```
// Output Flow
// 1. App renders.
// 2. LazyComponent starts loading.
// 3. Suspense shows fallback.
Loading component...
// 4. Once loaded, real component is displayed.
My Component
This component is loaded using Suspense.
```

## 19.11.5 When to Use Suspense?

- \* When using **React.lazy()** for code splitting.
- \* When fetching data with libraries that support Suspense (e.g., React Query, SWR).
- \* When you want to show a fallback UI while content is loading.
- \* When improving user experience during async operations.



### Key Points:

- ✓ Suspense shows a fallback UI while the child is loading.
- ✓ It works with **React.lazy()** and data fetching libraries.
- ✓ It improves user experience and prevents blank screens.
- ✓ It keeps the code clean and easy to manage.
- ✓ Use meaningful fallbacks for a better user experience.



## 19.11.6 Suspense vs Normal Loading

| Feature         | Normal Loading (useState, useEffect)         | Suspense                                         |
|-----------------|----------------------------------------------|--------------------------------------------------|
| Handling        | Manually manage loading state                | Automatically handles loading state              |
| Code Simplicity | More code (loading state, error state, etc.) | Less code, cleaner                               |
| User Experience | May show blank or partial UI                 | Shows fallback until ready                       |
| Works With      | Any data fetching method                     | <b>React.lazy()</b> and Suspense-enabled libs    |
| Performance     | Depends on implementation                    | Optimized for async rendering                    |
| Best Use        | Simple apps                                  | Modern apps with code splitting or data fetching |



### Key Takeaway:



Suspense makes it easy to handle loading states in React applications. It works hand-in-hand with **React.lazy()** and async data fetching to provide a **smooth and modern** user experience.

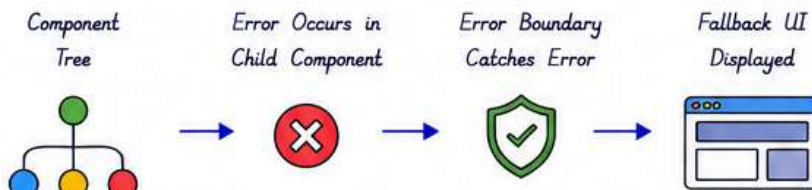


# 19.12 Error Boundaries

## 19.12.1 What is an Error Boundary?

- \* An Error Boundary is a React component that **catches JavaScript errors** anywhere in its child component tree.
- \* It logs those errors and displays a **fallback UI** instead of crashing the entire application.
- \* Error Boundaries catch errors during rendering, in lifecycle methods, and in constructors of the child components.
- \* They **do not catch errors** in event handlers, asynchronous code (e.g., `setTimeout`, `fetch`), or errors thrown in the Error Boundary itself.

### How Error Boundaries Work?



Error Boundaries prevent the whole app from crashing and show a graceful fallback UI.

## 19.12.2 Benefits of Error Boundaries

### 1. Prevents App Crash



Stops the entire React app from crashing due to errors in child components.

### 2. Better User Experience



Displays a meaningful message instead of a blank screen.

### 3. Easier Debugging



Helps in logging errors for debugging and monitoring.

### 4. Isolates Errors



Errors in one part of the UI do not affect other parts of the application.

### 5. Production Ready



Essential for building robust and reliable React applications.

## 19.12.3 Example: Error Boundary Component

```
import React from 'react';
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null };
  }
  static getDerivedStateFromError(error) {
    return { hasError: true, error };
  }
  componentDidCatch(error, errorInfo) {
    console.error("Error caught by ErrorBoundary:", error, errorInfo);
    // You can also log error to an error reporting service
  }
  render() {
    if (this.state.hasError) {
      return (
        <div style={{ padding: '20px', color: 'red', border: '1px solid red' }}>
          <h3>Something went wrong.</h3>
          <p>{this.state.error?.toString()}</p>
        </div>
      );
    }
    return this.props.children;
  }
}
export default ErrorBoundary;
```

## 19.12.4 Using Error Boundary

```
import React from 'react';
import ErrorBoundary from './ErrorBoundary';
import MyComponent from './MyComponent';

function App() {
  return (
    <div>
      <h2>Error Boundary Example</h2>
      <ErrorBoundary>
        <MyComponent />
      </ErrorBoundary>
    </div>
  );
}
export default App;
```

### Explanation:

- \* If `MyComponent` throws an error, it will be caught by `ErrorBoundary`.
- \* Instead of crashing the app, the fallback UI will be shown.
- \* The error is also logged in the console (`componentDidCatch`).



### Key Points:

- ✓ Error Boundaries catch errors in rendering, lifecycle methods, and constructors of child components.
- ✓ They **do not catch errors** in event handlers or async code.
- ✓ They help in preventing app crashes and improving user experience.
- ✓ Always wrap critical parts of your app with Error Boundaries.
- ✓ React provides error boundaries to build resilient applications.



## 19.12.5 When to Use Error Boundaries?

- \* When building large and complex React applications.
- \* When you want to prevent UI from breaking due to errors in child components.
- \* When you need to display a custom fallback UI or error message.
- \* When you want to log errors for monitoring and debugging.



## 19.12.6 Error Boundary vs try...catch.

| Feature                      | Error Boundary (React)                     | try...catch (JavaScript)          |
|------------------------------|--------------------------------------------|-----------------------------------|
| Scope                        | Catches errors in React component tree     | Catches errors in a block of code |
| Where it works               | Rendering, lifecycle methods, constructors | Anywhere in JavaScript code       |
| Catches async errors         | No                                         | Yes                               |
| Catches event handler errors | No                                         | Yes                               |
| UI Fallback                  | Yes (shows fallback UI)                    | No (you must handle it manually)  |
| Purpose                      | Prevent app crash & show fallback UI       | Handle and recover from errors    |
| Built-in (React)             | Yes (React feature)                        | No (JavaScript language feature)  |



### Key Takeaway:



Error Boundaries are an important feature in React that helps in handling errors gracefully. They **prevent the entire app from crashing** and show a fallback UI, providing a **better user experience** and making your app more reliable.

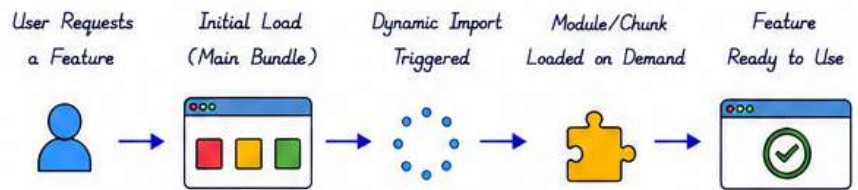


# 19.13 Dynamic Imports

## 19.13.1 What is Dynamic Import?

- \* Dynamic Import is a way to load JavaScript modules on demand using the `import()` function.
- \* It returns a `Promise` that resolves to the module.
- \* It helps in `code splitting` by loading modules only when they are needed.
- \* React uses dynamic imports with `React.lazy()`, but you can also use it directly in any component or function.

### How Dynamic Imports Work?



Only required modules are loaded when needed, reducing the initial bundle size.

## 19.13.2 Benefits of Dynamic Imports

### 1. Smaller Initial Bundle



Reduces the initial bundle size by loading modules on demand.

### 2. Improved Performance



Loads only what is needed, improving load time and performance.

### 3. Better User Experience



Users get faster initial render and smoother navigation.

### 4. Efficient Resource Usage



Reduces unnecessary network requests and saves bandwidth.

### 5. Scalable Applications



Helps in building large applications with better maintainability.

## 19.13.3 Syntax

```
// Dynamic import returns a Promise
import('./path/to/module')
  .then((module) => {
    // use the module
  })
  .catch((error) => {
    // handle error
  });

OR using async/await
const module = await import('./path/to/module');
// use module
```

### Key Points:

- \* `import()` is a function that returns a `Promise`.
- \* The module is loaded only when the `import()` is called.
- \* It can be used in any function or event.

## 19.13.4 Examples

### 1. Basic Dynamic Import

```
async function loadModule(){
  try {
    const module = await import('./MyComponent');
    console.log('Module Loaded:', module);
    // use the module
    const Comp = module.default;
    // e.g., render dynamically
  } catch (error) {
    console.error('Failed to load module:', error);
  }
}
```

```
// Trigger the function
loadModule();
```

### 2. Dynamic Import in Event Handler

```
function App(){
  const handleClick = async() => {
    const module = await import('./MyComponent');
    const Comp = module.default;
    // Do something with Comp
    console.log('Button Clicked, Module Loaded');
  };

  return (
    <div>
      <h3>Dynamic Import Example</h3>
      <button onClick={handleClick}>
        Load Component
      </button>
    </div>
  );
}
export default App;
```

### 3. Using with `React.lazy()`

```
// This is preferred for components
// with Suspense
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <React.Suspense fallback=<p>Loading...</p>>
      <LazyComponent />
    </React.Suspense>
  );
}
export default App;
```

## 19.13.5 When to Use Dynamic Imports?

- \* When a feature or component is rarely used.
- \* When a component is loaded in response to a user action (e.g., button click, route change).
- \* When you want to reduce the initial load time of your application.
- \* When building large scale React applications.



### Key Points:

- ✓ Dynamic imports load modules on demand.
- ✓ It helps in `code splitting` and reduces initial bundle size.
- ✓ `import()` returns a `Promise`.
- ✓ Can be used with `async/await` or `React.lazy()`.
- ✓ Improves performance and user experience.



## 19.13.6 Dynamic Imports vs Normal Imports

| Feature      | Normal Import                               | Dynamic Import                  |
|--------------|---------------------------------------------|---------------------------------|
| Loading Time | All modules are loaded at once.             | Modules are loaded on demand.   |
| Bundle Size  | Larger initial bundle size.                 | Smaller initial bundle size.    |
| Performance  | Slower initial load.                        | Faster initial load.            |
| Use Case     | Always needed modules.                      | Rarely used or large modules.   |
| Syntax       | <code>import Module from './Module';</code> | <code>import('./Module')</code> |
| Best Use     | Core or essential modules.                  | Optional or heavy modules.      |

### Key Takeaway:



Dynamic Imports allow you to load modules only when they are needed. They help in `code splitting`, reduce initial load time, and improve the performance of React applications.



# 19.14 Large Application Structure

## 19.14.1 What is Large Application Structure?

- \* As React applications grow, proper structure becomes very important.
- \* A good structure makes the codebase **scalable**, **maintainable** and easy to understand.
- \* It helps teams to work in parallel and reduces complexity.
- \* React does not enforce a structure, but following **best practices** leads to clean and efficient applications.

### Why Structure Matters?



A well-structured React application grows with confidence and is easy to manage.

## 19.14.2 Best Practices for Large Applications

### 1. Feature Based Structure



Organize code by features instead of type (components, utils, etc.).

### 2. Reusable Components



Create small, reusable components and compose them.

### 3. Separation of Concerns



Separate UI, business logic, API calls and state management.

### 4. State Management



Use Context API or libraries like Redux for global state.

### 5. Routing Structure



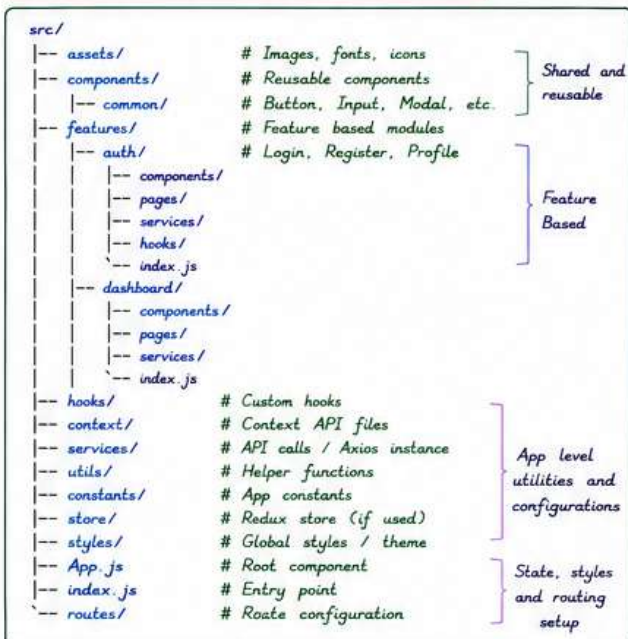
Organize routes in separate files and use nested routes.

### 6. Consistent Naming



Use clear and consistent naming for files, folders and variables.

## 19.14.3 Recommended Folder Structure



## 19.14.4 Example: Route Structure

```
// routes/AppRoutes.js
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Layout from '../components/common/Layout';
import Home from '../features/home/pages/Home';
import Login from '../features/auth/pages/Login';
import Dashboard from '../features/dashboard/pages/Dashboard';

function AppRoutes() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />} />
        <Route index element={<Home />} />
        <Route path="/dashboard" element={<Dashboard />} />
      </Routes>
    </BrowserRouter>
  );
}
export default AppRoutes;
```

### Benefits:

- \* Keeps routes organized.
- \* Supports nested routes.
- \* Improves scalability.
- \* Easy to manage as app grows.

## 19.14.5 Example: State Management Structure

```
// store/index.js
import { configureStore } from '@reduxjs/toolkit';
import authReducer from '../features/auth/store/authSlice';
import dashboardReducer from '../features/dashboard/store/dashboardSlice';

export const store = configureStore({
  reducer: {
    auth: authReducer,
    dashboard: dashboardReducer,
  },
});
```

### Benefits:

- \* Centralized state.
- \* Predictable state updates.
- \* Easy debugging.
- \* Scalable for large applications.

## 19.14.6 Key Principles

1. Organize code by features, not by type.
2. Keep components small, reusable and focused.
3. Separate concerns: UI, logic, API, state.
4. Use lazy loading and code splitting for performance.
5. Follow consistent naming and folder structure.



### Key Points:

- ✓ Structure your React app by features for scalability.
- ✓ Separate UI, logic, API and state for better maintainability.
- ✓ Use reusable components and consistent naming.
- ✓ Use proper routing and state management for large apps.
- ✓ Good structure = Easy to develop, maintain and scale.



## 19.14.7 Good Structure vs Bad Structure

| Aspect          | Good Structure              | Bad Structure                          |
|-----------------|-----------------------------|----------------------------------------|
| Organization    | Feature based               | By file type (components, utils, etc.) |
| Reusability     | High (Reusable components)  | Low (Duplicate code)                   |
| Scalability     | Easy to scale               | Hard to scale                          |
| Maintainability | Easy to maintain            | Difficult to maintain                  |
| Collaboration   | Teams can work in parallel  | Conflicts and confusion                |
| Performance     | Optimized with lazy loading | Large bundle, slow load time           |



### Key Takeaway:



A well-structured React application is the foundation of **scalability**, **performance** and **maintainability**. Follow best practices, organize by features, and keep your code clean and modular.

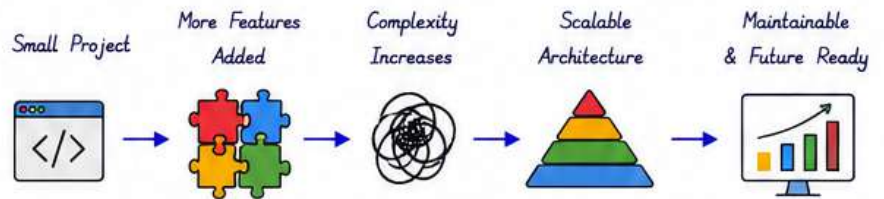


# 19.15 Scalable React Architecture

## 19.15.1 What is Scalable React Architecture?

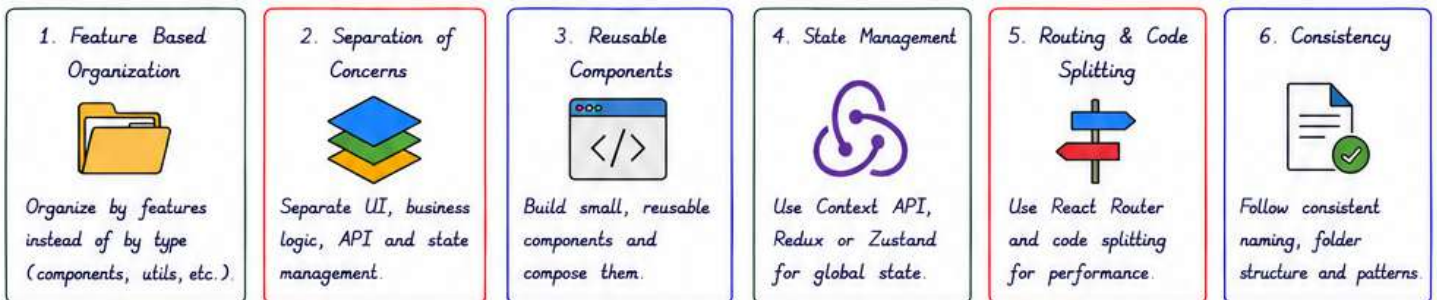
- \* Scalable React Architecture is the design of a React application in a way that it can grow with increasing features, users and complexity.
- \* It focuses on **modularity**, **separation of concerns**, **reusability** and **maintainability**.
- \* A scalable architecture reduces technical debt and supports team collaboration.
- \* It makes the application easy to test, update and extend without breaking existing features.

### Why Scalable Architecture?

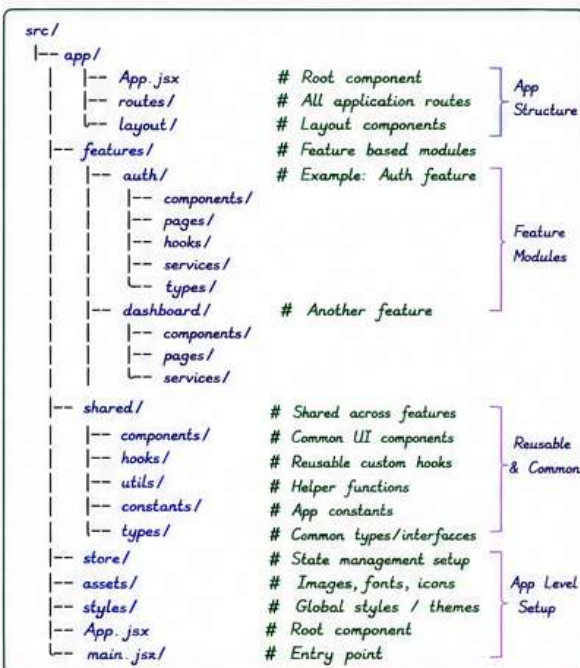


A scalable architecture helps your app grow smoothly without becoming messy and unmanageable.

## 19.15.2 Best Practices for Scalable React Architecture



## 19.15.3 Recommended Folder Structure



## 19.15.4 Example: Scalable Architecture Concepts



## 19.15.5 Key Principles

- \* Organize by features, not by file types.
- \* Keep components small, reusable and focused.
- \* Separate UI, logic, API and state.
- \* Use lazy loading and code splitting for performance.
- \* Maintain consistent naming and folder structure.
- \* Write clean, modular and testable code.



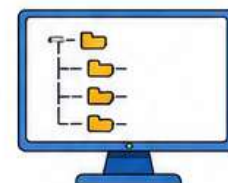
### Key Points:

- ✓ Feature based structure makes the project easy to scale.
- ✓ Modularity and separation of concerns improve maintainability.
- ✓ Reusable components and shared utilities reduce duplication.
- ✓ State management provides predictable and centralized state.
- ✓ Routing and code splitting improve performance and user experience.
- ✓ Consistent patterns and structure make onboarding easy for new developers.



## 19.15.6 Scalable Architecture vs Unstructured Architecture

| Aspect          | Scalable Architecture                   | Unstructured Architecture |
|-----------------|-----------------------------------------|---------------------------|
| Organization    | Feature based                           | File type based           |
| Maintainability | Easy to maintain                        | Hard to maintain          |
| Scalability     | Easily scalable                         | Difficult to scale        |
| Reusability     | High reusability                        | Low reusability           |
| Performance     | Optimized with lazy loading & splitting | Slower as app grows       |
| Collaboration   | Teams work efficiently                  | Conflicts and confusion   |
| Code Quality    | Clean, modular and testable             | Messy and tightly coupled |



### Key Takeaway:



Scalable React Architecture ensures that your application grows with your business. It improves code quality, performance, maintainability and team productivity.

