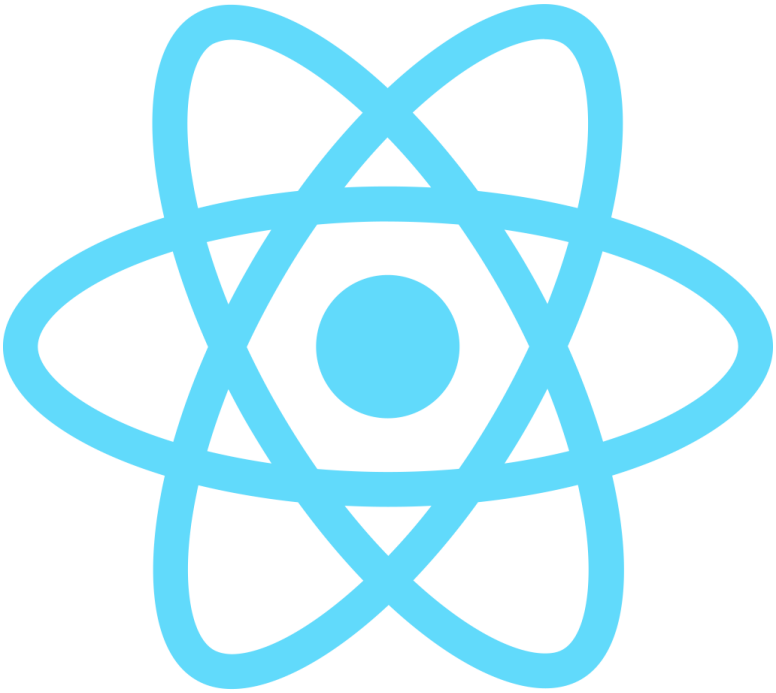
# React (Reference) Documentation



# Table of Contents

# Table of Contents

# Table of Contents

# React Reference Overview

This section provides detailed reference documentation for working with React. For an introduction to React, please visit the Learn section.

The React reference documentation is broken down into functional subsections:

## React

Programmatic React features:

- Hooks - Use different React features from your components.
- Components - Built-in components that you can use in your JSX.
- APIs - APIs that are useful for defining components.
- Directives - Provide instructions to bundlers compatible with React Server Components.

## React DOM

React-dom contains features that are only supported for web applications (which run in the browser DOM environment). This section is broken into the following:

- Hooks - Hooks for web applications which run in the browser DOM environment.
- Components - React supports all of the browser built-in HTML and SVG components.
- APIs - The `react-dom` package contains methods supported only in web applications.
- Client APIs - The `react-dom/client` APIs let you render React components on the client (in the browser).

- Server APIs - The `react-dom/server` APIs let you render React components to HTML on the server.

## Rules of React

React has idioms — or rules — for how to express patterns in a way that is easy to understand and yields high-quality applications:

- Components and Hooks must be pure – Purity makes your code easier to understand, debug, and allows React to automatically optimize your components and hooks correctly.
- React calls Components and Hooks – React is responsible for rendering components and hooks when necessary to optimize the user experience.
- Rules of Hooks – Hooks are defined using JavaScript functions, but they represent a special type of reusable UI logic with restrictions on where they can be called.

## Legacy APIs

- Legacy APIs - Exported from the `react` package, but not recommended for use in newly written code.

# Built-in React Hooks

*Hooks* let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own. This page lists all built-in Hooks in React.

---

## State Hooks

*State* lets a component "remember" information like user input. For example, a form component can use state to store the input value, while an image gallery component can use state to store the selected image index.

To add state to a component, use one of these Hooks:

- `useState` declares a state variable that you can update directly.
- `useReducer` declares a state variable with the update logic inside a reducer function.

```
function ImageGallery() {
  const [index, setIndex] = useState(0);
  // ...
```

---

## Context Hooks

*Context* lets a component receive information from distant parents without passing it as props. For example, your app's top-level component can pass the current UI theme to all components below, no matter how deep.

- `useContext` reads and subscribes to a context.

```
function Button() {
  const theme = useContext(ThemeContext);
  // ...
```

## Ref Hooks

*Refs* let a component hold some information that isn't used for rendering, like a DOM node or a timeout ID. Unlike with state, updating a ref does not re-render your component. Refs are an "escape hatch" from the React paradigm. They are useful when you need to work with non-React systems, such as the built-in browser APIs.

- `useRef` declares a ref. You can hold any value in it, but most often it's used to hold a DOM node.
- `useImperativeHandle` lets you customize the ref exposed by your component. This is rarely used.

```
function Form() {
  const inputRef = useRef(null);
  // ...
```

## Effect Hooks

*Effects* let a component connect to and synchronize with external systems. This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.

- `useEffect` connects a component to an external system.

```
function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);
  // ...
```

Effects are an “escape hatch” from the React paradigm. Don’t use Effects to orchestrate the data flow of your application. If you’re not interacting with an external system, you might not need an Effect.

There are two rarely used variations of `useEffect` with differences in timing:

- `useLayoutEffect` fires before the browser repaints the screen. You can measure layout here.
- `useInsertionEffect` fires before React makes changes to the DOM. Libraries can insert dynamic CSS here.

---

## Performance Hooks

A common way to optimize re-rendering performance is to skip unnecessary work. For example, you can tell React to reuse a cached calculation or to skip a re-render if the data has not changed since the previous render.

To skip calculations and unnecessary re-rendering, use one of these Hooks:

- `useMemo` lets you cache the result of an expensive calculation.
- `useCallback` lets you cache a function definition before passing it down to an optimized component.

```
function TodoList({ todos, tab, theme }) {
```

```
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos,
tab]);
  // ...
}
```

Sometimes, you can't skip re-rendering because the screen actually needs to update. In that case, you can improve performance by separating blocking updates that must be synchronous (like typing into an input) from non-blocking updates which don't need to block the user interface (like updating a chart).

To prioritize rendering, use one of these Hooks:

- `useTransition` lets you mark a state transition as non-blocking and allow other updates to interrupt it.
- `useDeferredValue` lets you defer updating a non-critical part of the UI and let other parts update first.

---

## Other Hooks

These Hooks are mostly useful to library authors and aren't commonly used in the application code.

- `useDebugValue` lets you customize the label React DevTools displays for your custom Hook.
- `useId` lets a component associate a unique ID with itself. Typically used with accessibility APIs.
- `useSyncExternalStore` lets a component subscribe to an external store.
- `useActionState` allows you to manage state of actions.

---

## Your own Hooks

You can also define your own custom Hooks as JavaScript functions.

# useActionState

`useActionState` is a Hook that allows you to update state based on the result of a form action.

```
const [state, formAction, isPending] = useActionState(fn, initialState, permalink?);
```

### Note

In earlier React Canary versions, this API was part of React DOM and called `useFormState`.



---

## Reference

### `useActionState(action, initialState, permalink?)`

Call `useActionState` at the top level of your component to create component state that is updated when a form action is invoked. You pass `useActionState` an existing form action function as well as an initial state, and it returns a new action that you use in your form, along with the latest form state and whether the Action is still pending. The latest form state is also passed to the function that you provided.

```
import { useActionState } from "react";

async function increment(previousState, formData) {
  return previousState + 1;
}

function StatefulForm({}) {
  const [state, formAction] = useActionState(increment, 0);
  return (
    <form>
      {state}
      <button formAction={formAction}>Increment</button>
    </form>
  )
}
```

The form state is the value returned by the action when the form was last submitted. If the form has not yet been submitted, it is the initial state that you pass.

If used with a Server Function, `useActionState` allows the server's response from submitting the form to be shown even before hydration has completed.

See more examples below.

### Parameters

- `fn` : The function to be called when the form is submitted or button pressed. When the function is called, it will receive the previous state of the form

(initially the `initialState` that you pass, subsequently its previous return value) as its initial argument, followed by the arguments that a form action normally receives.

- `initialState`: The value you want the state to be initially. It can be any serializable value. This argument is ignored after the action is first invoked.
- **optional** `permalink`: A string containing the unique page URL that this form modifies. For use on pages with dynamic content (eg: feeds) in conjunction with progressive enhancement: if `fn` is a server function and the form is submitted before the JavaScript bundle loads, the browser will navigate to the specified permalink URL, rather than the current page's URL. Ensure that the same form component is rendered on the destination page (including the same action `fn` and `permalink`) so that React knows how to pass the state through. Once the form has been hydrated, this parameter has no effect.

## Returns

`useActionState` returns an array with the following values:

1. The current state. During the first render, it will match the `initialState` you have passed. After the action is invoked, it will match the value returned by the action.
2. A new action that you can pass as the `action` prop to your `form` component or `formAction` prop to any `button` component within the form. The action can also be called manually within `startTransition`.
3. The `isPending` flag that tells you whether there is a pending Transition.

## Caveats

- When used with a framework that supports React Server Components, `useActionState` lets you make forms interactive before JavaScript has executed on the client. When used without Server Components, it is equivalent to component local state.
- The function passed to `useActionState` receives an extra argument, the previous or initial state, as its first argument. This makes its signature different than if it were used directly as a form action without using `useActionState`.

# Usage

## Using information returned by a form action

Call `useActionState` at the top level of your component to access the return value of an action from the last time a form was submitted.

```
import { useActionState } from 'react';
import { action } from './actions.js';

function MyComponent() {
  const [state, formAction] = useActionState(action, null);
  // ...
  return (
    <form action={formAction}>
      {/* ... */}
    </form>
  );
}
```

`useActionState` returns an array with the following items:

1. The current state of the form, which is initially set to the initial state you provided, and after the form is submitted is set to the return value of the action you provided.
2. A new action that you pass to `<form>` as its `action` prop or call manually within `startTransition`.
3. A pending state that you can utilise while your action is processing.

When the form is submitted, the action function that you provided will be called. Its return value will become the new current state of the form.

The action that you provide will also receive a new first argument, namely the current state of the form. The first time the form is submitted, this will be the

initial state you provided, while with subsequent submissions, it will be the return value from the last time the action was called. The rest of the arguments are the same as if `useActionState` had not been used.

```
function action(currentState, formData) {
  // ...
  return 'next state';
}
```

## Display information after submitting a form

1. Display form errors
2. Display structured information after submitting a form

### Example 1 of 2:

### Display form errors

To display messages such as an error message or toast that's returned by a Server Function, wrap the action in a call to `useActionState`.

App.js actions.js

```
import { useActionState, useState } from "react";
import { addToCart } from "./actions.js";

function AddToCartForm({itemID, itemTitle}) {
  const [message, formAction, isPending] = useActionState(addToCart, n
  return (
    <form action={formAction}>
      <h2>{itemTitle}</h2>
      <input type="hidden" name="itemID" value={itemID} />
      <button type="submit">Add to Cart</button>
      {isPending ? "Loading..." : message}
    </form>
  );
```

```
}

export default function App() {
  return (
    <>
      <AddToCartForm itemID="1" itemTitle="JavaScript: The Definitive
      <AddToCartForm itemID="2" itemTitle="JavaScript: The Good Parts"
    </>
  )
}
```

Next Example

# Troubleshooting

## My action can no longer read the submitted form data

When you wrap an action with `useActionState`, it gets an extra argument *as its first argument*. The submitted form data is therefore its *second* argument instead of its first as it would usually be. The new first argument that gets added is the current state of the form.

```
function action(currentState, formData) {
  // ...
}
```

# useCallback

`useCallback` is a React Hook that lets you cache a function definition between re-renders.

```
const cachedFn = useCallback(fn, dependencies)
```



---

## Reference

### `useCallback(fn, dependencies)`

Call `useCallback` at the top level of your component to cache a function definition between re-renders:

```
import { useCallback } from 'react';

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
```

```
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
```

See more examples below.

### Parameters

- `fn` : The function value that you want to cache. It can take any arguments and return any values. React will return (not call!) your function back to you during the initial render. On next renders, React will give you the same function again if the `dependencies` have not changed since the last render. Otherwise, it will give you the function that you have passed during the current render, and store it in case it can be reused later. React will not call your function. The function is returned to you so you can decide when and whether to call it.
- `dependencies` : The list of all reactive values referenced inside of the `fn` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]` . React will compare each dependency with its previous value using the `Object.is` comparison algorithm.

### Returns

On the initial render, `useCallback` returns the `fn` function you have passed.

During subsequent renders, it will either return an already stored `fn` function from the last render (if the dependencies haven't changed), or return the `fn` function you have passed during this render.

### Caveats

- `useCallback` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.
- React **will not throw away the cached function unless there is a specific reason to do that.** For example, in development, React throws away the cache when you edit the file of your component. Both in development and in production, React will throw away the

cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache—for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should match your expectations if you rely on `useCallback` as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

---

# Usage

## Skipping re-rendering of components

When you optimize rendering performance, you will sometimes need to cache the functions that you pass to child components. Let's first look at the syntax for how to do this, and then see in which cases it's useful.

To cache a function between re-renders of your component, wrap its definition into the `useCallback` Hook:

```
import { useCallback } from 'react';

function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
  // ...
```

You need to pass two things to `useCallback`:

1. A function definition that you want to cache between re-renders.
2. A list of dependencies including every value within your component that's used inside your function.

On the initial render, the returned function you'll get from `useCallback` will be the function you passed.

On the following renders, React will compare the dependencies with the dependencies you passed during the previous render. If none of the dependencies have changed (compared with `Object.is`), `useCallback` will return the same function as before. Otherwise, `useCallback` will return the function you passed on *this* render.

In other words, `useCallback` caches a function between re-renders until its dependencies change.

**Let's walk through an example to see when this is useful.**

Say you're passing a `handleSubmit` function down from the `ProductPage` to the `ShippingForm` component:

```
function ProductPage({ productId, referrer, theme }) {
  // ...
  return (
    <div className={theme}>
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
```

You've noticed that toggling the `theme` prop freezes the app for a moment, but if you remove `<ShippingForm />` from your JSX, it feels fast. This tells you that it's worth trying to optimize the `ShippingForm` component.

**By default, when a component re-renders, React re-renders all of its children recursively.** This is why, when `ProductPage` re-renders with a different `theme`, the `ShippingForm` component *also* re-renders. This is fine for components that don't require much calculation to re-render. But if you verified a re-render is slow, you can tell `ShippingForm` to skip re-rendering when its props are the same as on last render by wrapping it in `memo`:

```
import { memo } from 'react';

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
  // ...
});
```

**With this change, `ShippingForm` will skip re-rendering if all of its props are the *same* as on the last render.** This is when caching a function becomes important! Let's say you defined `handleSubmit` without `useCallback`:

```
function ProductPage({ productId, referrer, theme }) {
  // Every time the theme changes, this will be a different function...
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }

  return (
    <div className={theme}>
      {/* ... so ShippingForm's props will never be the same, and it will re-render
every time */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

**In JavaScript, a `function () {}` or `() => {}` always creates a *different* function,** similar to how the `{}` object literal always creates a new object. Normally, this wouldn't be a problem, but it means that `ShippingForm` props will never be the same, and your `memo` optimization won't work. This is where `useCallback` comes in handy:

```
function ProductPage({ productId, referrer, theme }) {
  // Tell React to cache your function between re-renders...
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ...so as long as these dependencies don't change...

  return (
    <div className={theme}>
      {/* ...ShippingForm will receive the same props and can skip re-rendering */}
      <ShippingForm onSubmit={handleSubmit} />
```

```
    </div>
  );
}
```

**By wrapping `handleSubmit` in `useCallback`, you ensure that it's the *same* function between the re-renders** (until dependencies change). You don't *have to* wrap a function in `useCallback` unless you do it for some specific reason. In this example, the reason is that you pass it to a component wrapped in `memo`, and this lets it skip re-rendering. There are other reasons you might need `useCallback` which are described further on this page.

### Note

**You should only rely on `useCallback` as a performance optimization.** If your code doesn't work without it, find the underlying problem and fix it first. Then you may add `useCallback` back.

DEEP DIVE

### How is useCallback related to useMemo?

You will often see `useMemo` alongside `useCallback`. They are both useful when you're trying to optimize a child component. They let you memoize (or, in other words, cache) something you're passing down:

```
import { useMemo, useCallback } from 'react';

function ProductPage({ productId, referrer }) {
  const product = useData('/product/' + productId);
```

```
  const requirements = useMemo(() => { // Calls your function and caches its
result
    return computeRequirements(product);
  }, [product]);

  const handleSubmit = useCallback((orderDetails) => { // Caches your function
itself
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);

  return (
    <div className={theme}>
      <ShippingForm requirements={requirements} onSubmit={handleSubmit} />
    </div>
  );
}
```

The difference is in *what* they're letting you cache:

- **`useMemo` caches the *result* of calling your function.** In this example, it caches the result of calling `computeRequirements(product)` so that it doesn't change unless `product` has changed. This lets you pass the `requirements` object down without unnecessarily re-rendering `ShippingForm`. When necessary, React will call the function you've passed during rendering to calculate the result.
- **`useCallback` caches *the function itself.*** Unlike `useMemo`, it does not call the function you provide. Instead, it caches the function you provided so that `handleSubmit` *itself* doesn't change unless `productId` or `referrer` has changed. This lets you pass the `handleSubmit` function down without unnecessarily re-rendering `ShippingForm`. Your code won't run until the user submits the form.

If you're already familiar with `useMemo`, you might find it helpful to think of `useCallback` as this:

```
// Simplified implementation (inside React)
function useCallback(fn, dependencies) {
  return useMemo(() => fn, dependencies);
```

```
}
```

Read more about the difference between `useMemo` and `useCallback`.

DEEP DIVE

### Should you add useCallback everywhere?

If your app is like this site, and most interactions are coarse (like replacing a page or an entire section), memoization is usually unnecessary. On the other hand, if your app is more like a drawing editor, and most interactions are granular (like moving shapes), then you might find memoization very helpful.

Caching a function with `useCallback` is only valuable in a few cases:

- You pass it as a prop to a component wrapped in `memo`. You want to skip re-rendering if the value hasn't changed. Memoization lets your component re-render only if dependencies changed.
- The function you're passing is later used as a dependency of some Hook. For example, another function wrapped in `useCallback` depends on it, or you depend on this function from `useEffect`.

There is no benefit to wrapping a function in `useCallback` in other cases. There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible. The downside is that code becomes less readable. Also, not all memoization is effective: a single value that's "always new" is enough to break memoization for an entire component.

Note that `useCallback` does not prevent *creating* the function. You're always creating a function (and that's fine!), but React ignores it and gives you back a cached function if nothing changed.

**In practice, you can make a lot of memoization unnecessary by following a few principles:**

1. When a component visually wraps other components, let it accept JSX as children. Then, if the wrapper component updates its own state, React knows that its children don't need to re-render.
2. Prefer local state and don't lift state up any further than necessary. Don't keep transient state like forms and whether an item is hovered at the top of your tree or in a global state library.
3. Keep your rendering logic pure. If re-rendering a component causes a problem or produces some noticeable visual artifact, it's a bug in your component! Fix the bug instead of adding memoization.
4. Avoid unnecessary Effects that update state. Most performance problems in React apps are caused by chains of updates originating from Effects that cause your components to render over and over.
5. Try to remove unnecessary dependencies from your Effects. For example, instead of memoization, it's often simpler to move some object or a function inside an Effect or outside the component.

If a specific interaction still feels laggy, use the React Developer Tools profiler to see which components benefit the most from memoization, and add memoization where needed. These principles make your components easier to debug and understand, so it's good to follow them in any case. In long term, we're researching doing memoization automatically to solve this once and for all.

## The difference between useCallback and declaring a function directly

1. Skipping re-rendering with `useCallback` and `memo` | 2. Always re-rendering a component

### Example 1 of 2:

### Skipping re-rendering with `useCallback` and `memo`

In this example, the `ShippingForm` component is **artificially slowed down** so that you can see what happens when a React component you're rendering is genuinely

slow. Try incrementing the counter and toggling the theme.

Incrementing the counter feels slow because it forces the slowed down `ShippingForm` to re-render. That's expected because the counter has changed, and so you need to reflect the user's new choice on the screen.

Next, try toggling the theme. **Thanks to `useCallback` together with `memo`, it's fast despite the artificial slowdown!** `ShippingForm` skipped re-rendering because the `handleSubmit` function has not changed. The `handleSubmit` function has not changed because both `productId` and `referrer` (your `useCallback` dependencies) haven't changed since last render.

App.js ProductPage.js ShippingForm.js

```
import { useCallback } from 'react';
import ShippingForm from './ShippingForm.js';

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);

  return (
    <div className={theme}>
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}

function post(url, data) {
  // Imagine this sends a request...
  console.log('POST /' + url);
  console.log(data);
}
```

Next Example

## Updating state from a memoized callback

Sometimes, you might need to update state based on previous state from a memoized callback.

This `handleAddTodo` function specifies `todos` as a dependency because it computes the next todos from it:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos([...todos, newTodo]);
  }, [todos]);
  // ...
```

You'll usually want memoized functions to have as few dependencies as possible. When you read some state only to calculate the next state, you can remove that dependency by passing an updater function instead:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos(todos => [...todos, newTodo]);
  }, []); // ✅ No need for the todos dependency
  // ...
```

Here, instead of making `todos` a dependency and reading it inside, you pass an instruction about *how* to update the state ( `todos => [...todos, newTodo]` ) to React. Read more

about updater functions.

---

## Preventing an Effect from firing too often

Sometimes, you might want to call a function from inside an Effect:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  function createOptions() {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    // ...
```

This creates a problem. Every reactive value must be declared as a dependency of your Effect. However, if you declare `createOptions` as a dependency, it will cause your Effect to constantly reconnect to the chat room:

```
  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🔴 Problem: This dependency changes on every render
  // ...
```

To solve this, you can wrap the function you need to call from an Effect into `useCallback`:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const createOptions = useCallback(() => {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }, [roomId]); // ✅ Only changes when roomId changes

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // ✅ Only changes when createOptions changes
  // ...
```

This ensures that the `createOptions` function is the same between re-renders if the `roomId` is the same. **However, it's even better to remove the need for a function dependency.** Move your function *inside* the Effect:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() { // ✅ No need for useCallback or function dependencies!
      return {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
    }

    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ Only changes when roomId changes
  // ...
```

# End of Preview

You have reached the end of the preview.

To download the complete version, please visit our website: