

# Hooks API Reference

**These docs are old and won't be updated. Go to [react.dev](https://react.dev) for the new React docs.**

These new documentation pages teach modern React:

- [react : Hooks](#)

*Hooks* are a new addition in React 16.8. They let you use state and other React features without writing a class.

This page describes the APIs for the built-in Hooks in React.

If you're new to Hooks, you might want to check out [the overview](#) first. You may also find useful information in the [frequently asked questions](#) section.

- [Basic Hooks](#)
  - [useState](#)
  - [useEffect](#)
  - [useContext](#)
- [Additional Hooks](#)
  - [useReducer](#)
  - [useCallback](#)

## INSTALLATION

## MAIN CONCEPTS

## ADVANCED GUIDES

## API REFERENCE

## HOOKS

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks

## 7. Hooks API Reference

8. Hooks FAQ

## TESTING

## CONTRIBUTING

## FAQ

- [useMemo](#)
- [useRef](#)
- [useImperativeHandle](#)
- [useLayoutEffect](#)
- [useDebugValue](#)
- [useDeferredValue](#)
- [useTransition](#)
- [useId](#)
- [Library Hooks](#)
  - [useSyncExternalStore](#)
  - [useInsertionEffect](#)

## Basic Hooks

### useState

**This content is out of date.**

Read the new React documentation for [useState](#).

```
const [state, setState] = useState(initialState);
```

Returns a stateful value, and a function to update it.

During the initial render, the returned state ( `state` ) is the same as the value passed as the first argument ( `initialState` ).

The `setState` function is used to update the state. It accepts a new state value and enqueues a re-render of the component.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

```
setState(newState);
```

During subsequent re-renders, the first value returned by `useState` will always be the most recent state after applying updates.

### Note

React guarantees that `setState` function identity is stable and won't change on re-renders. This is why it's safe to omit from the `useEffect` or `useCallback` dependency list.

## Functional updates

If the new state is computed using the previous state, you can pass a function to `setState`. The function will receive the previous value, and return an updated value. Here's an example of a counter component that uses both forms of `setState`:

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() =>
setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount =>
prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount =>
prevCount + 1)}>+</button>
    </>
  );
}
```

The "+" and "-" buttons use the functional form, because the updated value is based on the previous value. But the "Reset"

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

button uses the normal form, because it always sets the count back to the initial value.

If your update function returns the exact same value as the current state, the subsequent rerender will be skipped completely.

### Note

Unlike the `setState` method found in class components, `useState` does not automatically merge update objects. You can replicate this behavior by combining the function updater form with object spread syntax:

```
const [state, setState] = useState({});
setState(prevState => {
  // Object.assign would also work
  return {...prevState, ...updatedValues};
});
```

Another option is `useReducer`, which is more suited for managing state objects that contain multiple sub-values.

## Lazy initial state

The `initialState` argument is the state used during the initial render. In subsequent renders, it is disregarded. If the initial state is the result of an expensive computation, you may provide a function instead, which will be executed only on the initial render:

```
const [state, setState] = useState(() => {
  const initialState =
    someExpensiveComputation(props);
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

```
    return initialState;
  });
```

## Bailing out of a state update

If you update a State Hook to the same value as the current state, React will bail out without rendering the children or firing effects. (React uses the `Object.is` comparison algorithm.)

Note that React may still need to render that specific component again before bailing out. That shouldn't be a concern because React won't unnecessarily go "deeper" into the tree. If you're doing expensive calculations while rendering, you can optimize them with `useMemo`.

## Batching of state updates

React may group several state updates into a single re-render to improve performance. Normally, this improves performance and shouldn't affect your application's behavior.

Before React 18, only updates inside React event handlers were batched. Starting with React 18, batching is enabled for all updates by default. Note that React makes sure that updates from several *different* user-initiated events — for example, clicking a button twice — are always processed separately and do not get batched. This prevents logical mistakes.

In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`. However, this can hurt performance so do this only where needed.

## useEffect

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

## This content is out of date.

Read the new React documentation for [useEffect](#).

```
useEffect(didUpdate);
```

Accepts a function that contains imperative, possibly effectful code.

Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's *render phase*). Doing so will lead to confusing bugs and inconsistencies in the UI.

Instead, use `useEffect`. The function passed to `useEffect` will run after the render is committed to the screen. Think of effects as an escape hatch from React's purely functional world into the imperative world.

By default, effects run after every completed render, but you can choose to fire them only when certain values have changed.

## Cleaning up an effect

Often, effects create resources that need to be cleaned up before the component leaves the screen, such as a subscription or timer ID. To do this, the function passed to `useEffect` may return a clean-up function. For example, to create a subscription:

```
useEffect(() => {  
  const subscription = props.source.subscribe();  
  return () => {
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

```
// Clean up the subscription
subscription.unsubscribe();
};
});
```

The clean-up function runs before the component is removed from the UI to prevent memory leaks. Additionally, if a component renders multiple times (as they typically do), the **previous effect is cleaned up before executing the next effect**. In our example, this means a new subscription is created on every update. To avoid firing an effect on every update, refer to the next section.

## Timing of effects

Unlike `componentDidMount` and `componentDidUpdate`, the function passed to `useEffect` fires **after** layout and paint, during a deferred event. This makes it suitable for the many common side effects, like setting up subscriptions and event handlers, because most types of work shouldn't block the browser from updating the screen.

However, not all effects can be deferred. For example, a DOM mutation that is visible to the user must fire synchronously before the next paint so that the user does not perceive a visual inconsistency. (The distinction is conceptually similar to passive versus active event listeners.) For these types of effects, React provides one additional Hook called `useLayoutEffect`. It has the same signature as `useEffect`, and only differs in when it is fired.

Additionally, starting in React 18, the function passed to `useEffect` will fire synchronously **before** layout and paint when it's the result of a discrete user input such as a click, or when it's the result of an update wrapped in `flushSync`. This

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

behavior allows the result of the effect to be observed by the event system, or by the caller of `flushSync`.

### Note

This only affects the timing of when the function passed to `useEffect` is called - updates scheduled inside these effects are still deferred. This is different than `useLayoutEffect`, which fires the function and processes the updates inside of it immediately.

Even in cases where `useEffect` is deferred until after the browser has painted, it's guaranteed to fire before any new renders. React will always flush a previous render's effects before starting a new update.

## Conditionally firing an effect

The default behavior for effects is to fire the effect after every completed render. That way an effect is always recreated if one of its dependencies changes.

However, this may be overkill in some cases, like the subscription example from the previous section. We don't need to create a new subscription on every update, only if the `source` prop has changed.

To implement this, pass a second argument to `useEffect` that is the array of values that the effect depends on. Our updated example now looks like this:

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ



```
    };  
  },  
  [props.source],  
);
```

Now the subscription will only be recreated when `props.source` changes.

## Note

If you use this optimization, make sure the array includes **all values from the component scope (such as props and state) that change over time and that are used by the effect**. Otherwise, your code will reference stale values from previous renders. Learn more about [how to deal with functions](#) and what to do when the [array values change too often](#).

If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ( `[]` ) as a second argument. This tells React that your effect doesn't depend on *any* values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.

If you pass an empty array ( `[]` ), the props and state inside the effect will always have their initial values. While passing `[]` as the second argument is closer to the familiar `componentDidMount` and `componentWillUnmount` mental model, there are usually [better solutions](#) to avoid re-running effects too often. Also, don't forget that React defers running `useEffect` until after the browser has painted, so doing extra work is less of a problem.

We recommend using the [exhaustive-deps](#) rule as part of our [eslint-plugin-react-hooks](#) package. It warns when dependencies are specified incorrectly and suggests a fix.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

The array of dependencies is not passed as arguments to the effect function. Conceptually, though, that's what they represent: every value referenced inside the effect function should also appear in the dependencies array. In the future, a sufficiently advanced compiler could create this array automatically.

## useContext

**This content is out of date.**

Read the new React documentation for [useContext](#).

```
const value = useContext(MyContext);
```

Accepts a context object (the value returned from `React.createContext`) and returns the current context value for that context. The current context value is determined by the `value` prop of the nearest `<MyContext.Provider>` above the calling component in the tree.

When the nearest `<MyContext.Provider>` above the component updates, this Hook will trigger a rerender with the latest context `value` passed to that `MyContext` provider. Even if an ancestor uses `React.memo` or `shouldComponentUpdate`, a rerender will still happen starting at the component itself using `useContext`.

Don't forget that the argument to `useContext` must be the *context object itself*:

- **Correct:** `useContext(MyContext)`
- **Incorrect:** `useContext(MyContext.Consumer)`
- **Incorrect:** `useContext(MyContext.Provider)`

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

A component calling `useContext` will always re-render when the context value changes. If re-rendering the component is expensive, you can optimize it by using memoization.

### Tip

If you're familiar with the context API before Hooks, `useContext(MyContext)` is equivalent to `static contextType = MyContext` in a class, or to `<MyContext.Consumer>`.

`useContext(MyContext)` only lets you *read* the context and subscribe to its changes. You still need a `<MyContext.Provider>` above in the tree to *provide* the value for this context.

### Putting it together with Context.Provider

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext =
  React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

```
function Toolbar(props) {
  return (
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

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

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme.background,
color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}
```

This example is modified for hooks from a previous example in the [Context Advanced Guide](#), where you can find more information about when and how to use Context.

## Additional Hooks

The following Hooks are either variants of the basic ones from the previous section, or only needed for specific edge cases. Don't stress about learning them up front.

### useReducer

**This content is out of date.**

Read the new React documentation for [useReducer](#).

```
const [state, dispatch] = useReducer(reducer,
initialArg, init);
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

An alternative to `useState`. Accepts a reducer of type `(state, action) => newState`, and returns the current state paired with a `dispatch` method. (If you're familiar with Redux, you already know how this works.)

`useReducer` is usually preferable to `useState` when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. `useReducer` also lets you optimize performance for components that trigger deep updates because you can pass `dispatch` down instead of callbacks.

Here's the counter example from the `useState` section, rewritten to use a reducer:

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer,
initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type:
'decrement'})}></button>
      <button onClick={() => dispatch({type:
'increment'})}></button>
    </>
  );
}
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

## Note

React guarantees that `dispatch` function identity is stable and won't change on re-renders. This is why it's safe to omit from the `useEffect` or `useCallback` dependency list.

## Specifying the initial state

There are two different ways to initialize `useReducer` state. You may choose either one depending on the use case. The simplest way is to pass the initial state as a second argument:

```
const [state, dispatch] = useReducer(  
  reducer,  
  {count: initialCount}  
);
```

## Note

React doesn't use the `state = initialState` argument convention popularized by Redux. The initial value sometimes needs to depend on props and so is specified from the Hook call instead. If you feel strongly about this, you can call `useReducer(reducer, undefined, reducer)` to emulate the Redux behavior, but it's not encouraged.

## Lazy initialization

You can also create the initial state lazily. To do this, you can pass an `init` function as the third argument. The initial state

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

will be set to `init(initialArg)`.

It lets you extract the logic for calculating the initial state outside the reducer. This is also handy for resetting the state later in response to an action:

```
function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer,
initialCount, init);
  return (
    <>
      Count: {state.count}
      <button
        onClick={() => dispatch({type: 'reset',
payload: initialCount})}>
        Reset
      </button>
      <button onClick={() => dispatch({type:
'decrement'})}></button>
      <button onClick={() => dispatch({type:
'increment'})}></button>
    </>
  );
}
```

## Bailing out of a dispatch

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

If you return the same value from a Reducer Hook as the current state, React will bail out without rendering the children or firing effects. (React uses the `Object.is` comparison algorithm.)

Note that React may still need to render that specific component again before bailing out. That shouldn't be a concern because React won't unnecessarily go "deeper" into the tree. If you're doing expensive calculations while rendering, you can optimize them with `useMemo`.

## useCallback

**This content is out of date.**

Read the new React documentation for [useCallback](#).

```
const memoizedCallback = useCallback(  
  () => {  
    doSomething(a, b);  
  },  
  [a, b],  
);
```

Returns a [memoized](#) callback.

Pass an inline callback and an array of dependencies.

`useCallback` will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. `shouldComponentUpdate`).

`useCallback(fn, deps)` is equivalent to `useMemo(() => fn, deps)`.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ



## Note

The array of dependencies is not passed as arguments to the callback. Conceptually, though, that's what they represent: every value referenced inside the callback should also appear in the dependencies array. In the future, a sufficiently advanced compiler could create this array automatically.

We recommend using the `exhaustive-deps` rule as part of our `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

## useMemo

**This content is out of date.**

Read the new React documentation for `useMemo`.

```
const memoizedValue = useMemo(() =>
  computeExpensiveValue(a, b), [a, b]);
```

Returns a memoized value.

Pass a “create” function and an array of dependencies.

`useMemo` will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

Remember that the function passed to `useMemo` runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in `useEffect`, not `useMemo`.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

If no array is provided, a new value will be computed on every render.

**You may rely on `useMemo` as a performance optimization, not as a semantic guarantee.** In the future, React may choose to “forget” some previously memoized values and recalculate them on next render, e.g. to free memory for offscreen components. Write your code so that it still works without `useMemo` — and then add it to optimize performance.

### Note

The array of dependencies is not passed as arguments to the function. Conceptually, though, that’s what they represent: every value referenced inside the function should also appear in the dependencies array. In the future, a sufficiently advanced compiler could create this array automatically.

We recommend using the `exhaustive-deps` rule as part of our `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

## `useRef`

**This content is out of date.**

Read the new React documentation for `useRef`.

```
const refContainer = useRef(initialValue);
```

`useRef` returns a mutable ref object whose `.current` property is initialized to the passed argument (`initialValue`).

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

The returned object will persist for the full lifetime of the component.

A common use case is to access a child imperatively:

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input
    element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the
input</button>
    </>
  );
}
```

Essentially, `useRef` is like a “box” that can hold a mutable value in its `.current` property.

You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to React with `<div ref={myRef} />`, React will set its `.current` property to the corresponding DOM node whenever that node changes.

However, `useRef()` is useful for more than the `ref` attribute. It’s handy for keeping any mutable value around similar to how you’d use instance fields in classes.

This works because `useRef()` creates a plain JavaScript object. The only difference between `useRef()` and creating a `{current: ...}` object yourself is that `useRef` will give you the same ref object on every render.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

Keep in mind that `useRef` *doesn't* notify you when its content changes. Mutating the `.current` property doesn't cause a re-render. If you want to run some code when React attaches or detaches a ref to a DOM node, you may want to use a [callback ref](#) instead.

## useImperativeHandle

**This content is out of date.**

Read the new React documentation for [useImperativeHandle](#).

```
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` customizes the instance value that is exposed to parent components when using `ref`. As always, imperative code using refs should be avoided in most cases. `useImperativeHandle` should be used with [forwardRef](#):

```
function FancyInput(props, ref) {  
  const inputRef = useRef();  
  useImperativeHandle(ref, () => ({  
    focus: () => {  
      inputRef.current.focus();  
    }  
  }));  
  return <input ref={inputRef} ... />;  
}  
FancyInput = forwardRef(FancyInput);
```

In this example, a parent component that renders `<FancyInput ref={inputRef} />` would be able to call `inputRef.current.focus()`.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

# useLayoutEffect

**This content is out of date.**

Read the new React documentation for [useLayoutEffect](#).

The signature is identical to `useEffect`, but it fires synchronously after all DOM mutations. Use this to read layout from the DOM and synchronously re-render. Updates scheduled inside `useLayoutEffect` will be flushed synchronously, before the browser has a chance to paint.

Prefer the standard `useEffect` when possible to avoid blocking visual updates.

## Tip

If you're migrating code from a class component, note `useLayoutEffect` fires in the same phase as `componentDidMount` and `componentDidUpdate`. However, **we recommend starting with `useEffect` first** and only trying `useLayoutEffect` if that causes a problem.

If you use server rendering, keep in mind that *neither* `useLayoutEffect` nor `useEffect` can run until the JavaScript is downloaded. This is why React warns when a server-rendered component contains `useLayoutEffect`. To fix this, either move that logic to `useEffect` (if it isn't necessary for the first render), or delay showing that component until after the client renders (if the HTML looks broken until `useLayoutEffect` runs).

To exclude a component that needs layout effects from the server-rendered HTML, render it conditionally with `showChild && <Child />` and defer showing it with

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

```
useEffect(() => { setShowChild(true); }, []).
```

This way, the UI doesn't appear broken before hydration.

## useDebugValue

**This content is out of date.**

Read the new React documentation for [useDebugValue](#).

`useDebugValue(value)`

`useDebugValue` can be used to display a label for custom hooks in React DevTools.

For example, consider the `useFriendStatus` custom Hook described in ["Building Your Own Hooks"](#):

```
function useFriendStatus(friendID) {  
  const [isOnline, setIsOnline] = useState(null);  
  
  // ...  
  
  // Show a label in DevTools next to this Hook  
  // e.g. "FriendStatus: Online"  
  useDebugValue(isOnline ? 'Online' : 'Offline');  
  
  return isOnline;  
}
```

### Tip

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

We don't recommend adding debug values to every custom Hook. It's most valuable for custom Hooks that are part of shared libraries.

## Defer formatting debug values

In some cases formatting a value for display might be an expensive operation. It's also unnecessary unless a Hook is actually inspected.

For this reason `useDebugValue` accepts a formatting function as an optional second parameter. This function is only called if the Hooks are inspected. It receives the debug value as a parameter and should return a formatted display value.

For example a custom Hook that returned a `Date` value could avoid calling the `toDateString` function unnecessarily by passing the following formatter:

```
useDebugValue(date, date => date.toDateString());
```

## `useDeferredValue`

**This content is out of date.**

Read the new React documentation for [`useDeferredValue`](#).

```
const deferredValue = useDeferredValue(value);
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

`useDeferredValue` accepts a value and returns a new copy of the value that will defer to more urgent updates. If the current render is the result of an urgent update, like user input, React will return the previous value and then render the new value after the urgent render has completed.

This hook is similar to user-space hooks which use debouncing or throttling to defer updates. The benefits to using `useDeferredValue` is that React will work on the update as soon as other work finishes (instead of waiting for an arbitrary amount of time), and like `startTransition`, deferred values can suspend without triggering an unexpected fallback for existing content.

## Memoizing deferred children

`useDeferredValue` only defers the value that you pass to it. If you want to prevent a child component from re-rendering during an urgent update, you must also memoize that component with `React.memo` or `React.useMemo`:

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing tells React to only re-render when
  // deferredQuery changes,
  // not when query changes.
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ



```
);
}
```

Memoizing the children tells React that it only needs to re-render them when `deferredQuery` changes and not when `query` changes. This caveat is not unique to `useDeferredValue`, and it's the same pattern you would use with similar hooks that use debouncing or throttling.

## useTransition

**This content is out of date.**

Read the new React documentation for [useTransition](#).

```
const [isPending, startTransition] = useTransition();
```

Returns a stateful value for the pending state of the transition, and a function to start it.

`startTransition` lets you mark updates in the provided callback as transitions:

```
startTransition(() => {
  setCount(count + 1);
});
```

`isPending` indicates when a transition is active to show a pending state:

```
function App() {
  const [isPending, startTransition] =
    useTransition();
  const [count, setCount] = useState(0);
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

```
function handleClick() {
  startTransition(() => {
    setCount(c => c + 1);
  });
}

return (
  <div>
    {isPending && <Spinner />}
    <button onClick={handleClick}>{count}</button>
  </div>
);
}
```

**Note:**

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transition will not show a fallback for re-suspended content. This allows the user to continue interacting with the current content while rendering the update.

## useId

**This content is out of date.**

Read the new React documentation for [useId](#).

```
const id = useId();
```

`useId` is a hook for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

## Note

`useId` is **not** for generating keys in a list. Keys should be generated from your data.

For a basic example, pass the `id` directly to the elements that need it:

```
function Checkbox() {
  const id = useId();
  return (
    <>
      <label htmlFor={id}>Do you like React?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
};
```

For multiple IDs in the same component, append a suffix using the same `id`:

```
function NameFields() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First
Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last
Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
}
```

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

**Note:**

`useId` generates a string that includes the `:` token. This helps ensure that the token is unique, but is not supported in CSS selectors or APIs like `querySelectorAll`.

`useId` supports an `identifierPrefix` to prevent collisions in multi-root apps. To configure, see the options for [hydrateRoot](#) and [ReactDOMServer](#).

## Library Hooks

The following Hooks are provided for library authors to integrate libraries deeply into the React model, and are not typically used in application code.

### `useSyncExternalStore`

**This content is out of date.**

Read the new React documentation for [useSyncExternalStore](#).

```
const state = useSyncExternalStore(subscribe,
  getSnapshot[, getServerSnapshot]);
```

`useSyncExternalStore` is a hook recommended for reading and subscribing from external data sources in a way that's compatible with concurrent rendering features like selective hydration and time slicing.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

This method returns the value of the store and accepts three arguments:

- `subscribe` : function to register a callback that is called whenever the store changes.
- `getSnapshot` : function that returns the current value of the store.
- `getServerSnapshot` : function that returns the snapshot used during server rendering.

The most basic example simply subscribes to the entire store:

```
const state = useSyncExternalStore(store.subscribe,
store.getSnapshot);
```

However, you can also subscribe to a specific field:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);
```

When server rendering, you must serialize the store value used on the server, and provide it to `useSyncExternalStore`. React will use this snapshot during hydration to prevent server mismatches:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

### Note:

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

`getSnapshot` must return a cached value. If `getSnapshot` is called multiple times in a row, it must return the same exact value unless there was a store update in between.

A shim is provided for supporting multiple React versions published as `use-sync-external-store/shim`. This shim will prefer `useSyncExternalStore` when available, and fallback to a user-space implementation when it's not.

As a convenience, we also provide a version of the API with automatic support for memoizing the result of `getSnapshot` published as `use-sync-external-store/with-selector`.

## useInsertionEffect

**This content is out of date.**

Read the new React documentation for [useInsertionEffect](#).



```
useInsertionEffect(didUpdate);
```

The signature is identical to `useEffect`, but it fires synchronously *before* all DOM mutations. Use this to inject styles into the DOM before reading layout in [useLayoutEffect](#). Since this hook is limited in scope, this hook does not have access to refs and cannot schedule updates.

### Note:

`useInsertionEffect` should be limited to css-in-js library authors. Prefer [useEffect](#) or [useLayoutEffect](#) instead.

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ

Is this page useful?  

[Edit this page](#)

[Previous article](#)

[Building Your Own  
Hooks](#)

[Next article](#)

[Hooks FAQ](#)

1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
- 7. Hooks API Reference**
8. Hooks FAQ