On this page

Hooks

React's new "hooks" APIs give function components the ability to use local component state, execute side effects, and more. React also lets us write custom hooks, which let us extract reusable hooks to add our own behavior on top of React's built-in hooks.

React Redux includes its own custom hook APIs, which allow your React components to subscribe to the Redux store and dispatch actions.

> **TIP**
>
> **We recommend using the React-Redux hooks API as the default approach in your React components.**
>
> The existing `connect` API still works and will continue to be supported, but the hooks API is simpler and works better with TypeScript.

These hooks were first added in v7.1.0.

Using Hooks in a React Redux App

As with `connect()`, you should start by wrapping your entire application in a `<Provider>` component to make the store available throughout the component tree:

```
const store = createStore(rootReducer)

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

From there, you may import any of the listed React Redux hooks APIs and use them within your function components.

`useSelector()`

```
const result: any = useSelector(selector: Function, equalityFn?: Function)
```

Allows you to extract data from the Redux store state, using a selector function.

> **INFO**
>
> The selector function should be pure since it is potentially executed multiple times and at arbitrary points in time.

The selector is approximately equivalent to the `mapStateToProps` argument to `connect` conceptually. The selector will be called with the entire Redux store state as its only argument. The selector will be run whenever the function component renders (unless its reference hasn't changed since a previous render of the component so that a cached result can be returned by the hook without re-running the selector). `useSelector()` will also subscribe to the Redux store, and run your selector whenever an action is dispatched.

However, there are some differences between the selectors passed to `useSelector()` and a `mapState` function:

- The selector may return any value as a result, not just an object. The return value of the selector will be used as the return value of the `useSelector()` hook.
- When an action is dispatched, `useSelector()` will do a reference comparison of the previous selector result value and the current result value. If they are different, the component will be forced to re-render. If they are the same, the component will not re-render.
- The selector function does *not* receive an `ownProps` argument. However, props can be used through closure (see the examples below) or by using a curried selector.
- Extra care must be taken when using memoizing selectors (see examples below for more details).
- `useSelector()` uses strict `===` reference equality checks by default, not shallow equality (see the following section for more details).

> **INFO**
>
> There are potential edge cases with using props in selectors that may cause issues. See the Usage Warnings section of this page for further details.

You may call `useSelector()` multiple times within a single function component. Each call to `useSelector()` creates an individual subscription to the Redux store. Because of the React update batching behavior used in React Redux v7, a dispatched action that causes multiple `useSelector()`s in the same component to return new values *should* only result in a single re-render.

Equality Comparisons and Updates

When the function component renders, the provided selector function will be called and its result will be returned from the `useSelector()` hook. (A cached result may be returned by the hook without re-running the selector if it's the same function reference as on a previous render of the component.)

However, when an action is dispatched to the Redux store, `useSelector()` only forces a re-render if the selector result appears to be different than the last result. As of v7.1.0-alpha.5, the default comparison is a strict `===` reference comparison. This is different than `connect()`, which uses shallow equality checks on the results of `mapState` calls to determine if re-rendering is needed. This has several implications on how you should use `useSelector()`.

With `mapState`, all individual fields were returned in a combined object. It didn't matter if the return object was a new reference or not - `connect()` just compared the individual fields. With `useSelector()`, returning a new object every time will *always* force a re-render by default. If you want to retrieve multiple values from the store, you can:

- Call `useSelector()` multiple times, with each call returning a single field value
- Use Reselect or a similar library to create a memoized selector that returns multiple values in one object, but only returns a new object when one of the values has changed.
- Use the `shallowEqual` function from React-Redux as the `equalityFn` argument to `useSelector()`, like:

```
import { shallowEqual, useSelector } from 'react-redux'

// later
const selectedData = useSelector(selectorReturningObject, shallowEqual)
```

The optional comparison function also enables using something like Lodash's `_.isEqual()` or Immutable.js's comparison capabilities.

`useSelector` Examples

Basic usage:

```
import React from 'react'
import { useSelector } from 'react-redux'

export const CounterComponent = () => {
  const counter = useSelector((state) => state.counter)
  return <div>{counter}</div>
}
```

Using props via closure to determine what to extract:

```
import React from 'react'
import { useSelector } from 'react-redux'

export const TodoListItem = (props) => {
  const todo = useSelector((state) => state.todos[props.id])
  return <div>{todo.text}</div>
}
```

Using memoizing selectors

When using `useSelector` with an inline selector as shown above, a new instance of the selector is created whenever the component is rendered. This works as long as the selector does not maintain any state. However, memoizing selectors (e.g. created via `createSelector` from `reselect`) do have internal state, and therefore care must be taken when using them. Below you can find typical usage scenarios for memoizing selectors.

When the selector does only depend on the state, simply ensure that it is declared outside of the component so that the same selector instance is used for each render:

```
import React from 'react'
import { useSelector } from 'react-redux'
import { createSelector } from 'reselect'

const selectNumCompletedTodos = createSelector(
  (state) => state.todos,
  (todos) => todos.filter((todo) => todo.completed).length
)

export const CompletedTodosCounter = () => {
  const numCompletedTodos = useSelector(selectNumCompletedTodos)
```

```
  return <div>{numCompletedTodos}</div>
}

export const App = () => {
  return (
    <>
      <span>Number of completed todos:</span>
      <CompletedTodosCounter />
    </>
  )
}
```

The same is true if the selector depends on the component's props, but will only ever be used in a single instance of a single component:

Copy

```
import React from 'react'
import { useSelector } from 'react-redux'
import { createSelector } from 'reselect'

const selectCompletedTodosCount = createSelector(
  (state) => state.todos,
  (_, completed) => completed,
  (todos, completed) =>
    todos.filter((todo) => todo.completed === completed).length
)

export const CompletedTodosCount = ({ completed }) => {
  const matchingCount = useSelector((state) =>
    selectCompletedTodosCount(state, completed)
  )

  return <div>{matchingCount}</div>
}

export const App = () => {
  return (
    <>
      <span>Number of done todos:</span>
      <CompletedTodosCount completed={true} />
    </>
  )
}
```

However, when the selector is used in multiple component instances and depends on the component's props, you need to ensure that each component instance gets its own selector

instance (see here for a more thorough explanation of why this is necessary):

```
import React, { useMemo } from 'react'
import { useSelector } from 'react-redux'
import { createSelector } from 'reselect'

const makeSelectCompletedTodosCount = () =>
  createSelector(
    (state) => state.todos,
    (_, completed) => completed,
    (todos, completed) =>
      todos.filter((todo) => todo.completed === completed).length
  )

export const CompletedTodosCount = ({ completed }) => {
  const selectCompletedTodosCount = useMemo(makeSelectCompletedTodosCount, [])

  const matchingCount = useSelector((state) =>
    selectCompletedTodosCount(state, completed)
  )

  return <div>{matchingCount}</div>
}

export const App = () => {
  return (
    <>
      <span>Number of done todos:</span>
      <CompletedTodosCount completed={true} />
      <span>Number of unfinished todos:</span>
      <CompletedTodosCount completed={false} />
    </>
  )
}
```

useDispatch()

```
const dispatch = useDispatch()
```

This hook returns a reference to the `dispatch` function from the Redux store. You may use it to dispatch actions as needed.

Examples

```
import React from 'react'
import { useDispatch } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const dispatch = useDispatch()

  return (
    <div>
      <span>{value}</span>
      <button onClick={() => dispatch({ type: 'increment-counter' })}>
        Increment counter
      </button>
    </div>
  )
}
```

When passing a callback using `dispatch` to a child component, you may sometimes want to memoize it with `useCallback`. *If* the child component is trying to optimize render behavior using `React.memo()` or similar, this avoids unnecessary rendering of child components due to the changed callback reference.

```
import React, { useCallback } from 'react'
import { useDispatch } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const dispatch = useDispatch()
  const incrementCounter = useCallback(
    () => dispatch({ type: 'increment-counter' }),
    [dispatch]
  )

  return (
    <div>
      <span>{value}</span>
      <MyIncrementButton onIncrement={incrementCounter} />
    </div>
  )
}

export const MyIncrementButton = React.memo(({ onIncrement }) => (
  <button onClick={onIncrement}>Increment counter</button>
))
```

INFO

The `dispatch` function reference will be stable as long as the same store instance is being passed to the `<Provider>`. Normally, that store instance never changes in an application.

However, the React hooks lint rules do not know that `dispatch` should be stable, and will warn that the `dispatch` variable should be added to dependency arrays for `useEffect` and `useCallback`. The simplest solution is to do just that:

```
export const Todos() = () => {
  const dispatch = useDispatch();

  useEffect(() => {
    dispatch(fetchTodos())
  // Safe to add dispatch to the dependencies array
  }, [dispatch])
}
```

`useStore()`

```
const store = useStore()
```

This hook returns a reference to the same Redux store that was passed in to the `<Provider>` component.

This hook should probably not be used frequently. Prefer `useSelector()` as your primary choice. However, this may be useful for less common scenarios that do require access to the store, such as replacing reducers.

Examples

```
import React from 'react'
import { useStore } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const store = useStore()

  // EXAMPLE ONLY! Do not do this in a real app.
```

```
  // The component will not automatically update if the store state changes
  return <div>{store.getState()}</div>
}
```

Custom context

The `<Provider>` component allows you to specify an alternate context via the `context` prop. This is useful if you're building a complex reusable component, and you don't want your store to collide with any Redux store your consumers' applications might use.

To access an alternate context via the hooks API, use the hook creator functions:

```
import React from 'react'
import {
  Provider,
  createStoreHook,
  createDispatchHook,
  createSelectorHook
} from 'react-redux'

const MyContext = React.createContext(null)

// Export your custom hooks if you wish to use them in other files.
export const useStore = createStoreHook(MyContext)
export const useDispatch = createDispatchHook(MyContext)
export const useSelector = createSelectorHook(MyContext)

const myStore = createStore(rootReducer)

export function MyProvider({ children }) {
  return (
    <Provider context={MyContext} store={myStore}>
      {children}
    </Provider>
  )
}
```

Usage Warnings

Stale Props and "Zombie Children"

> **INFO**
>
> The React-Redux hooks API has been production-ready since we released it in v7.1.0, and **we recommend using the hooks API as the default approach in your components**. However, there are a couple of edge cases that can occur, and **we're documenting those so that you can be aware of them**.

One of the most difficult aspects of React Redux's implementation is ensuring that if your `mapStateToProps` function is defined as `(state, ownProps)`, it will be called with the "latest" props every time. Up through version 4, there were recurring bugs reported involving edge case situations, such as errors thrown from a `mapState` function for a list item whose data had just been deleted.

Starting with version 5, React Redux has attempted to guarantee that consistency with `ownProps`. In version 7, that is implemented using a custom `Subscription` class internally in `connect()`, which forms a nested hierarchy. This ensures that connected components lower in the tree will only receive store update notifications once the nearest connected ancestor has been updated. However, this relies on each `connect()` instance overriding part of the internal React context, supplying its own unique `Subscription` instance to form that nesting, and rendering the `<ReactReduxContext.Provider>` with that new context value.

With hooks, there is no way to render a context provider, which means there's also no nested hierarchy of subscriptions. Because of this, the "stale props" and "zombie child" issues may potentially re-occur in an app that relies on using hooks instead of `connect()`.

Specifically, "stale props" means any case where:

- a selector function relies on this component's props to extract data
- a parent component *would* re-render and pass down new props as a result of an action
- but this component's selector function executes before this component has had a chance to re-render with those new props

Depending on what props were used and what the current store state is, this *may* result in incorrect data being returned from the selector, or even an error being thrown.

"Zombie child" refers specifically to the case where:

- Multiple nested connected components are mounted in a first pass, causing a child component to subscribe to the store before its parent
- An action is dispatched that deletes data from the store, such as a todo item

- The parent component *would* stop rendering that child as a result
- However, because the child subscribed first, its subscription runs before the parent stops rendering it. When it reads a value from the store based on props, that data no longer exists, and if the extraction logic is not careful, this may result in an error being thrown.

`useSelector()` tries to deal with this by catching all errors that are thrown when the selector is executed due to a store update (but not when it is executed during rendering). When an error occurs, the component will be forced to render, at which point the selector is executed again. This works as long as the selector is a pure function and you do not depend on the selector throwing errors.

If you prefer to deal with this issue yourself, here are some possible options for avoiding these problems altogether with `useSelector()`:

- Don't rely on props in your selector function for extracting data
- In cases where you do rely on props in your selector function *and* those props may change over time, *or* the data you're extracting may be based on items that can be deleted, try writing the selector functions defensively. Don't just reach straight into `state.todos[props.id].name` - read `state.todos[props.id]` first, and verify that it exists before trying to read `todo.name`.
- Because `connect` adds the necessary `Subscription` to the context provider and delays evaluating child subscriptions until the connected component has re-rendered, putting a connected component in the component tree just above the component using `useSelector` will prevent these issues as long as the connected component gets re-rendered due to the same store update as the hooks component.

> **INFO**
>
> For a longer description of these scenarios, see:
>
> - "Stale props and zombie children in Redux" by Kai Hao
> - this chat log that describes the problems in more detail
> - issue #1179

Performance

As mentioned earlier, by default `useSelector()` will do a reference equality comparison of the selected value when running the selector function after an action is dispatched, and will only cause the component to re-render if the selected value changed. However, unlike `connect()`

`useSelector()` does not prevent the component from re-rendering due to its parent re-rendering, even if the component's props did not change.

If further performance optimizations are necessary, you may consider wrapping your function component in `React.memo()`:

```
const CounterComponent = ({ name }) => {
  const counter = useSelector(state => state.counter)
  return (
    <div>
      {name}: {counter}
    </div>
  )
}

export const MemoizedCounterComponent = React.memo(CounterComponent)
```

Hooks Recipes

We've pared down our hooks API from the original alpha release, focusing on a more minimal set of API primitives. However, you may still wish to use some of the approaches we tried in your own apps. These examples should be ready to copy and paste into your own codebase.

Recipe: `useActions()`

This hook was in our original alpha release, but removed in `v7.1.0-alpha.4`, based on Dan Abramov's suggestion. That suggestion was based on "binding action creators" not being as useful in a hooks-based use case, and causing too much conceptual overhead and syntactic complexity.

You should probably prefer to call the `useDispatch` hook in your components to retrieve a reference to `dispatch`, and manually call `dispatch(someActionCreator())` in callbacks and effects as needed. You may also use the Redux `bindActionCreators` function in your own code to bind action creators, or "manually" bind them like `const boundAddTodo = (text) => dispatch(addTodo(text))`.

However, if you'd like to still use this hook yourself, here's a copy-pastable version that supports passing in action creators as a single function, an array, or an object.

```
import { bindActionCreators } from 'redux'
import { useDispatch } from 'react-redux'
import { useMemo } from 'react'

export function useActions(actions, deps) {
  const dispatch = useDispatch()
  return useMemo(
    () => {
      if (Array.isArray(actions)) {
        return actions.map(a => bindActionCreators(a, dispatch))
      }
      return bindActionCreators(actions, dispatch)
    },
    deps ? [dispatch, ...deps] : [dispatch]
  )
}
```

Recipe: `useShallowEqualSelector()`

```
import { useSelector, shallowEqual } from 'react-redux'

export function useShallowEqualSelector(selector) {
  return useSelector(selector, shallowEqual)
}
```

Additional considerations when using hooks

There are some architectural trade offs to take into consideration when deciding whether to use hooks or not. Mark Erikson summarizes these nicely in his two blog posts Thoughts on React Hooks, Redux, and Separation of Concerns and Hooks, HOCs, and Tradeoffs.

Edit this page