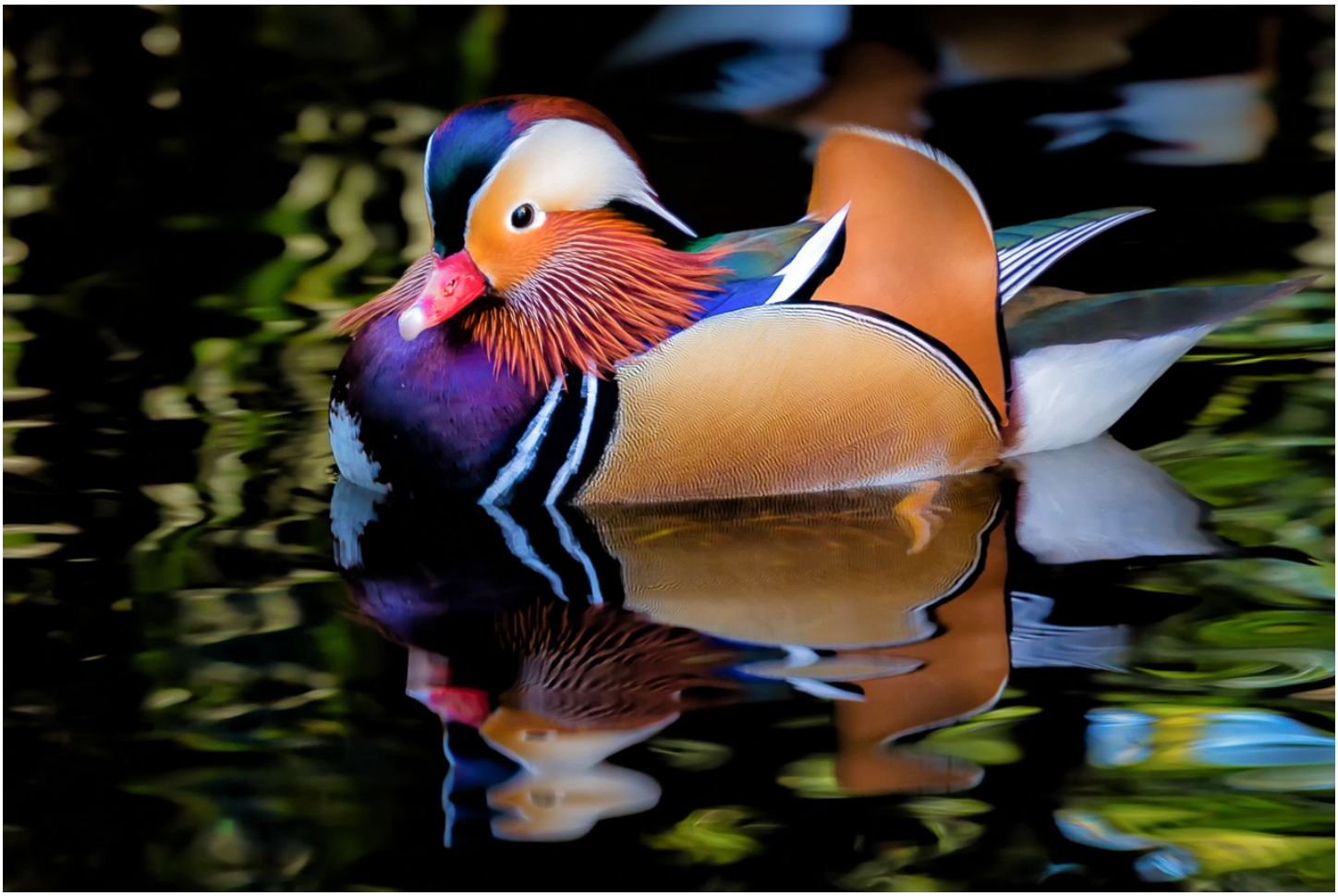
Do React Hooks Replace Redux?

TL;DR: Hooks are Great, but No.

Eric Elliott
Jul 25 · 10 min read



Mandarin Duck — Malcolm Carlaw (CC-BY-2.0)

Since the React hooks API was introduced, a lot of questions have risen about whether or not React hooks will replace Redux.

In my view, there is little overlap between hooks and Redux. Hooks didn't give us magical new state capabilities. Instead, it enhanced the API for things we could already do with React. However, the hooks API has made the native React state API a lot more

usable, and because it's easier than the `class` model it replaces, I use component state a lot more than I used to *when it's appropriate*.

To understand what I mean by that, we'll need a better understanding of why we might consider Redux in the first place.

What is Redux?

Redux is a predictable state management library *and architecture* which easily integrates with React.

The primary selling points of Redux are:

- **Deterministic state resolution** (enabling deterministic view renders when combined with pure components).
- **Transactional state.**
- **Isolate state management** from I/O and side-effects.
- **Single source of truth** for application state.
- **Easily share state** *between different components.*
- **Transaction telemetry** (auto-logging action objects).
- **Time travel debugging.**

In other words, Redux gives you code organization and debugging superpowers. It makes it easier to build more maintainable code, and much easier to track down the root cause when something goes wrong.

What are React Hooks?

React hooks let you use state and React lifecycle features without using `class` and React component lifecycle methods. They were introduced in React 16.8.

The primary selling points of React hooks are:

- **Use state and hook into the component lifecycle** without using a `class`.
- **Colocate related logic** in one place in your component, rather than splitting it between various lifecycle methods.

- **Share reusable behaviors** independent of component implementations (like the render prop pattern).

Note that these fantastic benefits don't really overlap with the benefits of Redux. You can and should use React hooks to get deterministic state updates, but that's always been a feature of React, and Redux's deterministic state model plugs nicely into it. That's how React affords deterministic view rendering, and is literally one of the driving motivations for the creation of React.

With tools like the react-redux hooks API, and React's useReducer hook, there's no need to choose one over the other. Use both. Mix and match.

What Do Hooks Replace?

Since the hooks API was introduced, I have stopped using:

- **`class` components.**
- **The render prop pattern.**

What Do Hooks Not Replace?

I still frequently use:

- **Redux** for all the reasons listed above.
- **Higher Order Components** to compose in cross-cutting concerns that are shared by all or most of my application views, such as the Redux provider, a common layout provider, a configuration provider, authentication/authorization, i18n, and so on.
- **Separation between container and display components** for better modularity, testability, and easier separation between effects and pure logic.

When to Use Hooks

You don't always need Redux for every app, or every component. If your app consists of a single view, doesn't save or load state, and has no asynchronous I/O, I can't think of a good reason to add the complexity of Redux.

Likewise, if your component:

- **Doesn't use the network.**

- **Doesn't save or load state.**
- **Doesn't share state** with other non-child components.
- **Does need some ephemeral local component state.**

You may have a good use case for React's built-in component state model. React hooks will serve you well in those cases. For example, the following form uses the local component state `useState` hook from React.

```
import React, { useState } from 'react';
import t from 'prop-types';
import TextField, { Input } from '@material/react-text-field';

const noop = () => {};

const Holder = ({
  itemPrice = 175,
  name = '',
  email = '',
  id = '',
  removeHolder = noop,
  showRemoveButton = false,
}) => {
  const [nameInput, setName] = useState(name);
  const [emailInput, setEmail] = useState(email);

const setter = set => e => {
    const { target } = e;
    const { value } = target;
    set(value);
  };

return (
    <div className="row">
      <div className="holder">
        <div className="holder-name">
          <TextField label="Name">
            <Input value={nameInput} onChange={setter(setName)}
required />
          </TextField>
        </div>
        <div className="holder-email">
          <TextField label="Email">
            <Input
              value={emailInput}
              onChange={setter(setEmail)}
              type="email"
              required
            />
          </TextField>
        </div>
```

```
          {showRemoveButton && (
            <button
              className="remove-holder"
              aria-label="Remove membership"
              onClick={e => {
                e.preventDefault();
                removeHolder(id);
              }}
            >
              &times;
            </button>
          )}
        </div>
        <div className="line-item-price">${itemPrice}</div>
        <style jsx>{cssHere}</style>
      </div>
    );
  };
  Holder.propTypes = {
    name: t.string,
    email: t.string,
    itemPrice: t.number,
    id: t.string,
    removeHolder: t.func,
    showRemoveButton: t.bool,
  };

  export default Holder;
```

This code uses `useState` to keep track of the ephemeral form input states for name and email:

```
  const [nameInput, setName] = useState(name);
  const [emailInput, setEmail] = useState(email);
```

You may notice there's also a `removeHolder` action creator coming into props from Redux. It's fine to mix and match.

It has always been fine to use local component state for things like this, but prior to React hooks, I probably would have been tempted to stuff it into Redux and pull the state from props, anyway.

Using component state would have meant using a `class` component, setting initial state with the `class` instance property syntax (or a `constructor` function), and so on — too much added complexity just to avoid Redux. It helped that there are plug-and-play

tools to manage form state with Redux, so I didn't have to worry about ephemeral form state bleeding into my business logic.

Since I was already using Redux in all my non-trivial apps, the choice was simple: Redux (almost) all the things!

Now the choice is still simple:

> Component state for component state, Redux for application state.

When to Use Redux

Another common question is *"should you put everything in Redux? Won't it break time travel debugging if you don't?"*

No, because there is a lot of state in an application which is *ephemeral* and *too granular* to provide very useful information for things like logging telemetry or time travel debugging. Unless you're building a realtime collaborative editor, you probably don't need to put every user keystroke or mouse movement into Redux state. When you add things to Redux state, you add a layer of abstraction and the complexity that goes along with it.

In other words, you should feel free to use Redux, but when you do, you should have a reason for it.

Redux is useful if your component:

- **Uses I/O like network or device APIs.**
- **Saves or loads state.**
- **Shares its state with non-child components.**
- **Deals with any business logic or data processing shared with other parts of the application.**

Here's another example from the TDDDay app:

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { compose } from 'ramda';

import page from '../../hocs/page.js';
import Purchase from './purchase-component.js';
import { addHolder, removeHolder, getHolders } from './purchase-
reducer.js';

const PurchasePage = () => {
  // You can use these instead of
  // mapStateToProps and mapDispatchToProps
  const dispatch = useDispatch();
  const holders = useSelector(getHolders);

const props = {
    // Use function composition to compose action creators
    // with dispatch. See "Composing Software" for details.
    addHolder: compose(
      dispatch,
      addHolder
    ),
    removeHolder: compose(
      dispatch,
      removeHolder
    ),
    holders,
  };

return <Purchase {...props} />;
};

// `page` is a Higher Order Component composed of many
// other higher order components using function composition.
export default page(PurchasePage);
```

This component doesn't handle any of the DOM. It's a presentation component. It connects to Redux using the React-Redux hooks API.

It uses Redux because we need the data this form cares about in other parts of the UI, and when we're done with the purchase flow, we'll need to save the data to a database.

The state it cares about is shared between components, not localized to a single component, it's persisted rather than ephemeral, and it potentially spans multiple page views or sessions. These are all things that local component state won't help with unless you build your own state container library on top of the React API — and that's a lot more complicated than just using Redux.

In the future, React's suspense API may help with saving and loading state. We'll need to wait until the suspense API lands to see if it can replace my saving/loading patterns in Redux. Redux allows us to cleanly separate side-effects from the rest of the component logic without requiring us to mock I/O services. (Isolation of effects is why I prefer redux-saga over thunks). In order to compete with this use-case for Redux, React's API's will need to afford effect isolation.

Redux is Architecture

Redux is a lot more (and often a lot less) than a state management library. It's also essentially a subset of the Flux Architecture which is more opinionated about how state changes are made. I have another blog post which details Redux architecture in more depth.

I frequently use redux-style reducers when I need complex component state but I don't need the Redux library. I also use Redux-style actions (and even Redux tools like Autodux and redux-saga) to dispatch actions in Node apps, without ever importing the Redux library.

Redux has always been more architecture and unenforced convention than library. In fact, the essential implementation of Redux can be reproduced in a couple dozen lines of code.

This is all great news if you'd like to start using more local component state with the hooks API instead of Reduxing all the things.

React supplies a `useReducer` hook that will interface with your Redux-style reducers. That's great for non-trivial state logic, dependent state, and so on. If you have a use-case where you think you can contain ephemeral state to a single component, you can use the Redux architecture, but use the `useReducer` hook instead of using Redux to manage the state.

If you later need to persist or share the state, you're 90% done already. All that's left is to connect the component and add the reducer to your Redux store.

More Q&A

> *"Does determinism break if everything isn't in Redux?"*

No. In fact, Redux doesn't enforce determinism, either. Convention does. If you want your Redux state to be deterministic, use pure functions. If you want your ephemeral component state to be deterministic, use pure functions.

> *"Don't you need Redux as a single source of truth?"*

The single source of truth principle does not state that you need all of your state to come from a single source. Instead, it means that for each piece of state, there should be a single source of truth for that state. You can have many different pieces of state, each with their own single source of truth.

That means you can choose what goes into Redux, and what goes into component state. You can also get state from other sources, like the browser APIs for the current location href.

Redux is a great way to maintain a single source of truth for your application state, but if your component state is localized to a single component and only used in one place, by definition, it already has a single source of truth for that state: React component state.

If you do put something into Redux state, you should always read it from Redux state. For all state in Redux, Redux should be your single source of truth for that state.

It's fine to put everything in Redux if you need to. There may be performance implications for state that needs to update frequently, or components with lots of dependent state. You shouldn't worry about performance unless it becomes an issue, but when you do, try both ways and see if it has an impact. Profile, and remember the RAIL performance model.

> *"Should I use react-redux connect, or the hooks API?"*

That depends. `connect` creates a reusable higher-order component, whereas the hooks API is optimized for integration with a single component. Will you need to connect the same store props to other components? Go with `connect`. Otherwise, I prefer how the hooks API reads. For example, imagine you have a component which handles permissions authorization for user actions:

```
import { connect } from 'react-redux';
import RequiresPermission from './requires-permission-component';
import { userHasPermission } from '../../features/user-profile/user-
```

```
profile-reducer';
import curry from 'lodash/fp/curry';

const requiresPermission = curry(
  (NotPermittedComponent, { permission }, PermittedComponent) => {
    const mapStateToProps = state => ({
      NotPermittedComponent,
      PermittedComponent,
      isPermitted: userHasPermission(state, permission),
    });

    return connect(mapStateToProps)(RequiresPermission);
  },
);

export default requiresPermission;
```

Now if you have a bunch of admin views that all require admin permission, you can create a higher-order component that composes in this permission requirement for all of them in with all your other cross-cutting concerns:

```
import NextError from 'next/error';
import compose from 'lodash/fp/compose';
import React from 'react';
import requiresPermission from '../requires-permission';
import withFeatures from '../with-features';
import withAuth from '../with-auth';
import withEnv from '../with-env';
import withLoader from '../with-loader';
import withLayout from '../with-layout';

export default compose(
  withEnv,
  withAuth,
  withLoader,
  withLayout(),
  withFeatures,
  requiresPermission(() => <NextError statusCode={404} />, {
    permission: 'admin',
  }),
);
```

To use that:

```
import compose from 'lodash/fp/compose';
import adminPage from '../HOCs/admin-page';
import AdminIndex from '../features/admin-index/admin-index-
component.js';
```

```
export default adminPage(AdminIndex);
```

The higher-order component API is convenient for this use-case, and it's actually more concise than the hooks API (it requires less code), but in order to read the `connect` API, you have to remember that it takes `mapStateToProps` as the first argument, and `mapDispatchToProps` as the second argument, and you should be aware that it can take functions or object literals, and you should know the differences in those behaviors. You also need to remember that it's curried, but not auto-curried.

In other words, I find the `connect` API does the job with concise code, but it's not particularly readable or ergonomic. If I don't need to reuse the connection for other components, I much prefer the infinitely more readable hooks API, even though it involves slightly more typing.

> *"If singletons are an anti-pattern, and Redux is a singleton, isn't Redux an anti-pattern?"*

No. Singletons are a code-smell that could indicate *shared mutable state,* which is the real anti-pattern. Redux prevents shared mutable state via encapsulation (you should not be mutating app state directly outside of reducers, instead, Redux handles state updates) and message passing (only dispatched action objects can trigger state updates).

Next Steps

Learn a lot more about React and Redux on EricElliottJS.com. Functional patterns such as function composition and partial application used in the code examples in this article are discussed in-depth with lots of examples and video walkthroughs.

Learn how to unit test React components, and speaking of testing, learn about Test Driven Development (TDD) on TDDDay.com.

. . .

Eric Elliott *is the author of the books, "Composing Software" and "Programming JavaScript Applications". As co-founder of EricElliottJS.com and DevAnywhere.io, he teaches developers essential software development skills. He builds and advises development teams for crypto projects, and has contributed to software experiences for* ***Adobe Systems,***

Zumba Fitness, The Wall Street Journal, ESPN, BBC, *and top recording artists including* ***Usher, Frank Ocean, Metallica,*** *and many more.*

He enjoys a remote lifestyle with the most beautiful woman in the world.

Thanks to JS_Cheerleader.

React Redux Technology JavaScript Software Development

About Help Legal