On this page

Redux Fundamentals, Part 5: UI and React

WHAT YOU'LL LEARN

- How a Redux store works with a UI
- How to use Redux with React

Introduction

In Part 4: Store, we saw how to create a Redux store, dispatch actions, and read the current state. We also looked at how a store works inside, how enhancers and middleware let us customize the store with additional abilities, and how to add the Redux DevTools to let us see what's happening inside our app as actions are dispatched.

In this section, we'll add a User Interface for our todo app. We'll see how Redux works with a UI layer overall, and we'll specifically cover how Redux works together with React.

Integrating Redux with a UI

Redux is a standalone JS library. As we've already seen, you can create and use a Redux store even if you don't have a user interface set up. This also means that **you can use Redux with any UI framework** (or even without *any* UI framework), and use it on both client and server. You can write Redux apps with React, Vue, Angular, Ember, jQuery, or vanilla JavaScript.

That said, **Redux was specifically designed to work well with React**. React lets you describe your UI as a function of your state, and Redux contains state and updates it in response to actions.

Because of that, we'll use React for this tutorial as we build our todo app, and cover the basics of how to use React with Redux.

Before we get to that part, let's take a quick look at how Redux interacts with a UI layer in general.

Basic Redux and UI Integration

Using Redux with any UI layer requires a few consistent steps:

1. Create a Redux store
2. Subscribe to updates
3. Inside the subscription callback:
   i. Get the current store state
   ii. Extract the data needed by this piece of UI
   iii. Update the UI with the data
4. If necessary, render the UI with initial state
5. Respond to UI inputs by dispatching Redux actions

Let's go back to the the counter app example we saw in Part 1 and see how it follows those steps:

```js
// 1) Create a new Redux store with the `createStore` function
const store = Redux.createStore(counterReducer)

// 2) Subscribe to redraw whenever the data changes in the future
store.subscribe(render)

// Our "user interface" is some text in a single HTML element
const valueEl = document.getElementById('value')

// 3) When the subscription callback runs:
function render() {
  // 3.1) Get the current store state
  const state = store.getState()
  // 3.2) Extract the data you want
  const newValue = state.value.toString()

  // 3.3) Update the UI with the new value
  valueEl.innerHTML = newValue
}

// 4) Display the UI with the initial store state
render()

// 5) Dispatch actions based on UI inputs
```

```
document.getElementById('increment').addEventListener('click', function
() {
  store.dispatch({ type: 'counter/incremented' })
})
```

No matter what UI layer you're using, **Redux works this same way with every UI**. The actual implementations are typically a bit more complicated to help optimize performance, but it's the same steps each time.

Since Redux is a separate library, there are different "binding" libraries to help you use Redux with a given UI framework. Those UI binding libraries handle the details of subscribing to the store and efficiently updating the UI as state changes, so that you don't have to write that code yourself.

Using Redux with React

The official **React-Redux UI bindings library** is a separate package from the Redux core. You'll need to install that in addition as well:

```
npm install react-redux
```

(If you don't use npm, you may grab the latest UMD build from unpkg (either a development or a production build). The UMD build exports a global called `window.ReactRedux` if you add it to your page via a `<script>` tag.)

For this tutorial, we'll cover the most important patterns and examples you need to use React and Redux together, and see how they work in practice as part of our todo app.

INFO

See **the official React-Redux docs at https://react-redux.js.org** for a complete guide on how to use Redux and React together, and reference documentation on the React-Redux APIs.

Designing the Component Tree

Much like we designed the state structure based on requirements, we can also design the overall set of UI components and how they relate to each other in the application.

Based on the list of business requirements for the app, at a minimum we're going to need this set of components:

- **`<App>`**: the root component that renders everything else.
  - **`<Header>`**: contains the "new todo" text input and the "complete all todos" checkbox
  - **`<TodoList>`**: a list of all currently visible todo items, based on the filtered results
    - **`<TodoListItem>`**: a single todo list item, with a checkbox that can be clicked to toggle the todo's completed status, and a color category selector
  - **`<Footer>`**: Shows the number of active todos and controls for filtering the list based on completed status and color category

Beyond this basic component structure, we could potentially divide the components up in several different ways. For example, the `<Footer>` component *could* be one larger component, or it could have multiple smaller components inside like `<CompletedTodos>`, `<StatusFilter>`, and `<ColorFilters>`. There's no single right way to divide these, and you'll find that it may be better to write larger components or split things into many smaller components depending on your situation.

For now, we'll start with this small list of components to keep things easier to follow. On that note, since we assume that you already know React, **we're going to skip past the details of how to write the layout code for these components and focus on how to actually use the React-Redux library in your React components**.

Here's the initial React UI of this app looks like before we start adding any Redux-related logic:

Reading State from the Store with `useSelector`

We know that we need to be able to show a list of todo items. Let's start by creating a `<TodoList>` component that can read the list of todos from the store, loop over them, and show one `<TodoListItem>` component for each todo entry.

You should be familiar with React hooks like `useState`, which can be called in React function components to give them access to React state values. 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.

Like many other libraries, React-Redux includes its own custom hooks, which you can use in your own components. The React-Redux hooks give your React component the ability to talk to the Redux store by reading state and dispatching actions.

The first React-Redux hook that we'll look at is the **`useSelector` hook**, which **lets your React components read data from the Redux store**.

`useSelector` accepts a single function, which we call a **selector** function. **A selector is a function that takes the entire Redux store state as its argument, reads some value from the state, and returns that result**.

For example, we know that our todo app's Redux state keeps the array of todo items as `state.todos`. We can write a small selector function that returns that todos array:

```
const selectTodos = state => state.todos
```

Or, maybe we want to find out how many todos are currently marked as "completed":

```
const selectTotalCompletedTodos = state => {
  const completedTodos = state.todos.filter(todo => todo.completed)
  return completedTodos.length
}
```

So, **selectors can return values from the Redux store state, and also return *derived* values based on that state as well**.

Let's read the array of todos into our `<TodoList>` component. First, we'll import the `useSelector` hook from the `react-redux` library, then call it with a selector function as its argument:

src/features/todos/TodoList.js

```
import React from 'react'
import { useSelector } from 'react-redux'
import TodoListItem from './TodoListItem'

const selectTodos = state => state.todos

const TodoList = () => {
  const todos = useSelector(selectTodos)

  // since `todos` is an array, we can loop over it
  const renderedListItems = todos.map(todo => {
    return <TodoListItem key={todo.id} todo={todo} />
  })

  return <ul className="todo-list">{renderedListItems}</ul>
}

export default TodoList
```

The first time the `<TodoList>` component renders, the `useSelector` hook will call `selectTodos` and pass in the *entire* Redux state object. Whatever the selector returns will be returned by the hook to your component. So, the `const todos` in our component will end up holding the same `state.todos` array inside our Redux store state.

But, what happens if we dispatch an action like `{type: 'todos/todoAdded'}`? The Redux state will be updated by the reducer, but our component needs to know that something has changed so that it can re-render with the new list of todos.

We know that we can call `store.subscribe()` to listen for changes to the store, so we *could* try writing the code to subscribe to the store in every component. But, that would quickly get very repetitive and hard to handle.

Fortunately, **`useSelector` automatically subscribes to the Redux store for us!** That way, any time an action is dispatched, it will call its selector function again right away. **If the value returned by the selector changes from the last time it ran, `useSelector` will force our component to re-render with the new data**. All we have to do is call `useSelector()` once in our component, and it does the rest of the work for us.

However, there's a very important thing to remember here:

> **CAUTION**
>
> **`useSelector` compares its results using strict `===` reference comparisons, so the component will re-render any time the selector result is a new reference!** This means that if you create a new reference in your selector and return it, your component could re-render *every* time an action has been dispatched, even if the data really isn't different.

For example, passing this selector to `useSelector` will cause the component to *always* re-render, because `array.map()` always returns a new array reference:

```js
// Bad: always returning a new reference
const selectTodoDescriptions = state => {
  // This creates a new array reference!
  return state.todos.map(todo => todo.text)
}
```

> **TIP**
>
> We'll talk about one way to fix this issue later in this section. We'll also talk about how you can improve performance and avoid unnecessary re-renders using "memoized" selector function in Part 7: Standard Redux Patterns.

It's also worth noting that we don't have to write a selector function as a separate variable. You can write a selector function directly inside the call to `useSelector`, like this:

```
const todos = useSelector(state => state.todos)
```

Dispatching Actions with `useDispatch`

We now know how to read data from the Redux store into our components. But, how can we dispatch actions to the store from a component? We know that outside of React, we can call `store.dispatch(action)`. Since we don't have access to the store in a component file, we need some way to get access to the `dispatch` function by itself inside our components.

The React-Redux **`useDispatch` hook** gives us the store's `dispatch` method as its result. (In fact, the implementation of the hook really is `return store.dispatch`.)

So, we can call `const dispatch = useDispatch()` in any component that needs to dispatch actions, and then call `dispatch(someAction)` as needed.

Let's try that in our `<Header>` component. We know that we need to let the user type in some text for a new todo item, and then dispatch a `{type: 'todos/todoAdded'}` action containing that text.

We'll write a typical React form component that uses "controlled inputs" to let the user type in the form text. Then, when the user presses the Enter key specifically, we'll dispatch that action.

src/features/header/Header.js

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

const Header = () => {
  const [text, setText] = useState('')
  const dispatch = useDispatch()

  const handleChange = e => setText(e.target.value)

  const handleKeyDown = e => {
    const trimmedText = e.target.value.trim()
    // If the user pressed the Enter key:
    if (e.key === 'Enter' && trimmedText) {
      // Dispatch the "todo added" action with this text
      dispatch({ type: 'todos/todoAdded', payload: trimmedText })
      // And clear out the text input
      setText('')
    }
  }

  return (
    <input
      type="text"
      placeholder="What needs to be done?"
      autoFocus={true}
      value={text}
      onChange={handleChange}
      onKeyDown={handleKeyDown}
    />
  )
}

export default Header
```

Passing the Store with `Provider`

Our components can now read state from the store, and dispatch actions to the store. However, we're still missing something. Where and how are the React-Redux hooks finding the right Redux store? A hook is a JS function, so it can't automatically import a store from `store.js` by itself.

Instead, we have to specifically tell React-Redux what store we want to use in our components. We do this by **rendering a `<Provider>` component around our entire `<App>`, and passing the Redux store as a prop to `<Provider>`**. After we do this once, every component in the application will be able to access the Redux store if needs to.

Let's add that to our main `index.js` file:

src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'

import App from './App'
import store from './store'

ReactDOM.render(
  // Render a `<Provider>` around the entire `<App>`,
  // and pass the Redux store to as a prop
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)
```

That covers the key parts of using React-Redux with React:

- Call the `useSelector` hook to read data in React components
- Call the `useDispatch` hook to dispatch actions in React components
- Put `<Provider store={store}>` around your entire `<App>` component so that other components can talk to the store

We should now be able to actually interact with the app! Here's the working UI so far:

Now, let's look at a couple more ways we can use these together in our todo app.

React-Redux Patterns

Global State, Component State, and Forms

By now you might be wondering, "Do I always have to put all my app's state into the Redux store?"

The answer is **NO. Global state that is needed across the app should go in the Redux store. State that's only needed in one place should be kept in component state.**

A good example of this is the `<Header>` component we wrote earlier. We *could* keep the current text input string in the Redux store, by dispatching an action in the input's `onChange` handler and keeping it in our reducer. But, that doesn't give us any benefit. The only place that text string is used is here, in the `<Header>` component.

So, it makes sense to keep that value in a `useState` hook here in the `<Header>` component.

Similarly, if we had a boolean flag called `isDropdownOpen`, no other components in the app would care about that - it should really stay local to this component.

TIP

In a React + Redux app, your global state should go in the Redux store, and your local state should stay in React components.

If you're not sure where to put something, here are some common rules of thumb for determining what kind of data should be put into Redux:

- Do other parts of the application care about this data?
- Do you need to be able to create further derived data based on this original data?
- Is the same data being used to drive multiple components?
- Is there value to you in being able to restore this state to a given point in time (ie, time travel debugging)?
- Do you want to cache the data (ie, use what's in state if it's already there instead of re-requesting it)?
- Do you want to keep this data consistent while hot-reloading UI components (which may lose their internal state when swapped)?

This is also a good example of how to think about forms in Redux in general. **Most form state probably shouldn't be kept in Redux.** Instead, keep the data in your form components as you're editing it, and then dispatch Redux actions to update the store when the user is done.

Using Multiple Selectors in a Component

Right now only our `<TodoList>` component is reading data from the store. Let's see what it might look like for the `<Footer>` component to start reading some data as well.

The `<Footer>` needs to know three different pieces of information:

- How many completed todos there are
- The current "status" filter value
- The current list of selected "color" category filters

How can we read these values into the component?

We can call `useSelector` multiple times within one component. In fact, this is actually a good idea - **each call to `useSelector` should always return the smallest amount of state possible**.

We already saw how to write a selector that counts completed todos earlier. For the filters values, both the status filter value and the color filters values live in the `state.filters` slice. Since this component needs both of them, we can select the entire `state.filters` object.

As we mentioned earlier, we could put all the input handling directly into `<Footer>`, or we could split it out into separate components like `<StatusFilter>`. To keep this explanation shorter, we'll skip the exact details of writing the input handling and assume we've got smaller separate components that are given some data and change handler callbacks as props.

Given that assumption, the React-Redux parts of the component might look like this:

src/features/footer/Footer.js

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

import { availableColors, capitalize } from '../filters/colors'
import { StatusFilters } from '../filters/filtersSlice'

// Omit other footer components

```

```
const Footer = () => {
  const todosRemaining = useSelector(state => {
    const uncompletedTodos = state.todos.filter(todo =>
!todo.completed)
    return uncompletedTodos.length
  })

  const { status, colors } = useSelector(state => state.filters)

  // omit placeholder change handlers

  return (
    <footer className="footer">
      <div className="actions">
        <h5>Actions</h5>
        <button className="button">Mark All Completed</button>
        <button className="button">Clear Completed</button>
      </div>

      <RemainingTodos count={todosRemaining} />
      <StatusFilter value={status} onChange={onStatusChange} />
      <ColorFilters value={colors} onChange={onColorChange} />
    </footer>
  )
}

export default Footer
```

Selecting Data in List Items by ID

Currently, our `<TodoList>` is reading the entire `state.todos` array and passing the actual todo objects as a prop to each `<TodoListItem>` component.

This works, but there's a potential performance problem.

- Changing one todo object means creating copies of both the todo and the `state.todos` array, and each copy is a new reference in memory
- When `useSelector` sees a new reference as its result, it forces its component to re-render
- So, any time *one* todo object is updated (like clicking it to toggle its completed status), the whole `<TodoList>` parent component will re-render

- Then, because React re-renders all child components recursively by default, it also means that *all* of the `<TodoListItem>` components will re-render, even though most of them didn't actually change at all!

Re-rendering components isn't bad - that's how React knows if it needs to update the DOM. But, re-rendering lots of components when nothing has actually changed can potentially get too slow if the list is too big.

There's a couple ways we could try to fix this. One option is to wrap all the `<TodoListItem>` components in `React.memo()`, so that they only re-render when their props actually change. This is often a good choice for improving performance, but it does require that the child component always receives the same props until something really changes. Since each `<TodoListItem>` component is receiving a todo item as a prop, only one of them should actually get a changed prop and have to re-render.

Another option is to have the `<TodoList>` component only read an array of todo IDs from the store, and pass those IDs as props to the child `<TodoListItem>` components. Then, each `<TodoListItem>` can use that ID to find the right todo object it needs.

Let's give that a shot.

src/features/todos/TodoList.js

```js
import React from 'react'
import { useSelector } from 'react-redux'
import TodoListItem from './TodoListItem'

const selectTodoIds = state => state.todos.map(todo => todo.id)

const TodoList = () => {
  const todoIds = useSelector(selectTodoIds)

  const renderedListItems = todoIds.map(todoId => {
    return <TodoListItem key={todoId} id={todoId} />
  })

  return <ul className="todo-list">{renderedListItems}</ul>
}
```

This time, we only select an array of todo IDs from the store in `<TodoList>`, and we pass each `todoId` as an `id` prop to the child `<TodoListItem>`s.

Then, in `<TodoListItem>`, we can use that ID value to read our todo item. We can also update `<TodoListItem>` to dispatch the "toggled" action based on the todo's ID.

src/features/todos/TodoListItem.js

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

import { availableColors, capitalize } from '../filters/colors'

const selectTodoById = (state, todoId) => {
  return state.todos.find(todo => todo.id === todoId)
}

// Destructure `props.id`, since we only need the ID value
const TodoListItem = ({ id }) => {
  // Call our `selectTodoById` with the state _and_ the ID value
  const todo = useSelector(state => selectTodoById(state, id))
  const { text, completed, color } = todo

  const dispatch = useDispatch()

  const handleCompletedChanged = () => {
    dispatch({ type: 'todos/todoToggled', payload: todo.id })
  }

  // omit other change handlers
  // omit other list item rendering logic and contents

  return (
    <li>
      <div className="view">{/* omit other rendering output */}</div>
    </li>
  )
}

export default TodoListItem
```

There's a problem with this, though. We said earlier that **returning new array references in selectors causes components to re-render every time**, and right now we're returning a new IDs array in `<TodoList>`. In this case, the *contents* of the IDs array should be the same if we're toggling a todo, because we're still showing the same todo items - we haven't added or deleted any. But, the array *containing* those IDs is a new reference, so `<TodoList>` will re-render when it really doesn't need to.

One possible solution to this is to change how `useSelector` compares its values to see if they've changed. `useSelector` can take a comparison function as its second argument. A comparison function is called with the old and new values, and returns `true` if they're considered the same. If they're the same, `useSelector` won't make the component re-render.

React-Redux has a `shallowEqual` comparison function we can use to check if the items *inside* the array are still the same. Let's try that:

src/features/todos/TodoList.js

```
import React from 'react'
import { useSelector, shallowEqual } from 'react-redux'
import TodoListItem from './TodoListItem'

const selectTodoIds = state => state.todos.map(todo => todo.id)

const TodoList = () => {
  const todoIds = useSelector(selectTodoIds, shallowEqual)

  const renderedListItems = todoIds.map(todoId => {
    return <TodoListItem key={todoId} id={todoId} />
  })

  return <ul className="todo-list">{renderedListItems}</ul>
}
```

Now, if we toggle a todo item, the list of IDs will be considered the same, and `<TodoList>` won't have to re-render. The one `<TodoListItem>` will get an updated todo object and re-render, but all the rest of them will still have the existing todo object and not have to re-render at all.

As mentioned earlier, you can also use a specialized kind of selector function called a "memoized selector" to help improve component rendering, and we'll look at how to use those in another section.

What You've Learned

We now have a working todo app! Our app creates a store, passes the store to the React UI layer using `<Provider>`, and then calls `useSelector` and `useDispatch` to talk to the store in our React components.

INFO

Try implementing the rest of the missing UI features on your own! Here's a list of the things you'll need to add:

- In `<TodoListItem>` component, use the `useDispatch` hook to dispatch actions for changing the color category and deleting the todo
- In `<Footer>`, use the `useDispatch` hook to dispatch actions for marking all todos as completed, clearing completed todos, and changing the filter values.

We'll cover implementing the filters in Part 7: Standard Redux Patterns.

Let's see how the app looks now, including the components and sections we skipped to keep this shorter:

SUMMARY

- **Redux stores can be used with any UI layer**
  - UI code always subscribes to the store, gets the latest state, and redraws itself
- **React-Redux is the official Redux UI bindings library for React**

    - React-Redux is installed as a separate `react-redux` package
- **The `useSelector` hook lets React components read data from the store**
    - Selector functions take the entire store `state` as an argument, and return a value based on that state
    - `useSelector` calls its selector function and returns the result from the selector
    - `useSelector` subscribes to the store, and re-runs the selector each time an action is dispatched.
    - Whenever the selector result changes, `useSelector` forces the component to re-render with the new data
- **The `useDispatch` hook lets React components dispatch actions to the store**
    - `useDispatch` returns the actual `store.dispatch` function
    - You can call `dispatch(action)` as needed inside your components
- **The `<Provider>` component makes the store available to other React components**
    - Render `<Provider store={store}>` around your entire `<App>`

What's Next?

Now that our UI is working, it's time to see how to make our Redux app talk to a server. In Part 6: Async Logic, we'll talk about how asynchronous logic like timeouts and AJAX calls fit into the Redux data flow.

Edit this page

*Last updated on **11/19/2021***