React: Use Redux To Control Modal Visibility States

betterprogramming.pub/react-use-redux-to-control-modal-visibility-states-8953e44b71fd

Easily manage dialog opening and closing state

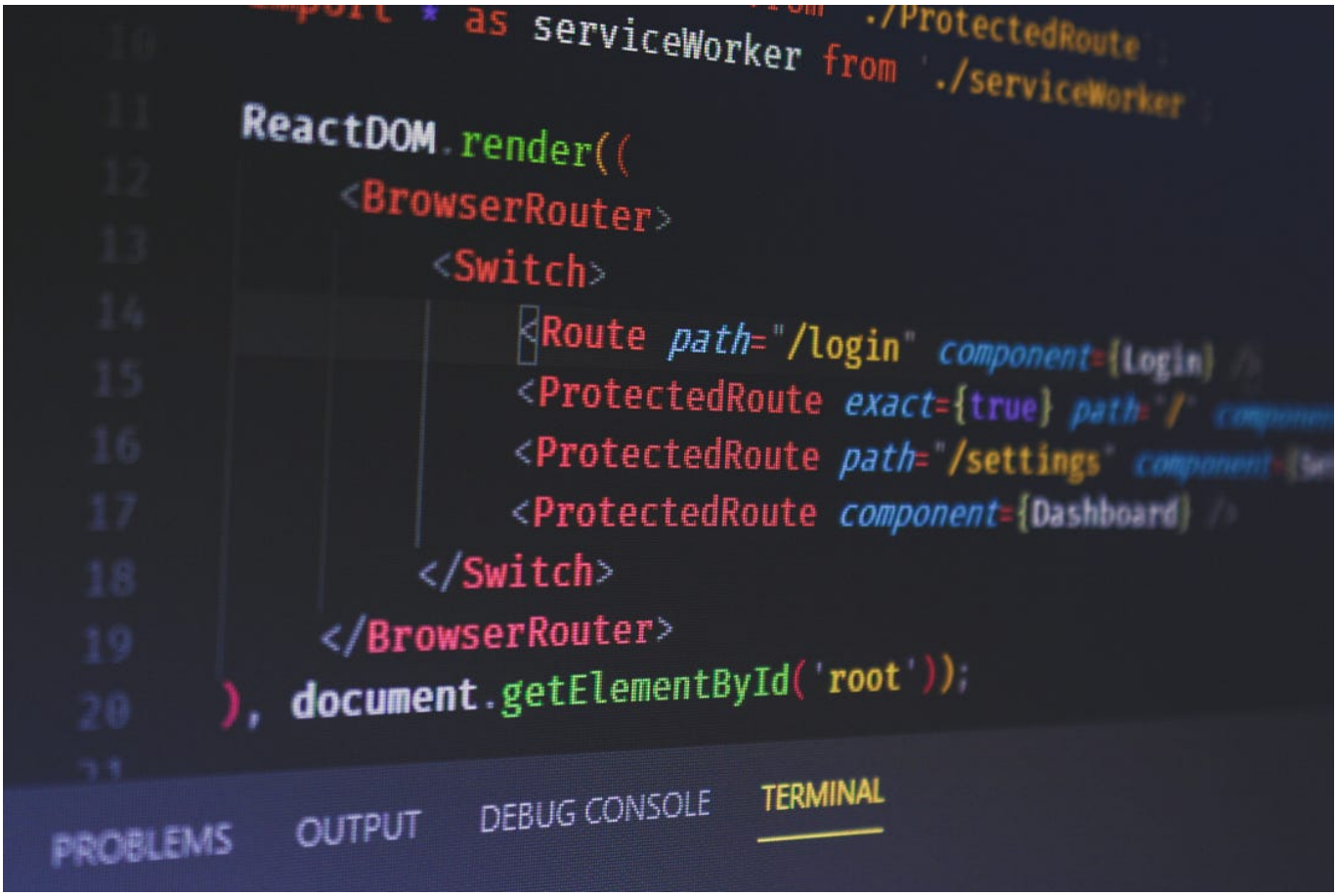


Photo by on .

Modals are UI elements that sit over the main window of the application and help users to make decisions without disrupting the current interaction flow. They also serve to provide feedback about actions triggered by the user and present some content without changing the current route/page of the app.

In this article, I will go over how we can use Redux in React to control whether the modal should or should not be visible. First, let's go through the application example and then move on to an approach using Redux in order to save us a lot of time and reduce an incredible amount of code in our components.

The Application

The application that we will use is a simple button that opens a modal asking the user to press either Yes or No:

Application example

In this approach, we are controlling the modal visibility using the `isOpen` prop that is defined directly in the `MainComponent`. If the `IsOpen` prop is `true`, the modal will be shown. Otherwise, it will be hidden.

There is no problem at all with this approach if we use this modal in only one component. If we try to use this modal in other components, which is the case of a `YesNoModal`, this approach could lead to a lot of code duplication.

Each time we use this modal, we will have to duplicate all the code that controls the open state of the modal. Each component that uses this modal will have to control the state by defining the `IsOpen` property.

What if we could invoke some action in order to show/hide this modal and centralize its state control? Let's use Redux for that.

How Redux Works

Managing state globally can be very complex. Redux helps us to better deal with state management. Briefly, Redux is a tool that allows us to centralize the state/logic of our React app. That's great news since we don't want to delegate the modal state control to every component that needs it.

Before we dive into the example using Redux, I would like to briefly explain what I consider the three main pillars of Redux: actions, reducers, and the store.

Actions are events that our components will invoke to let the rest of the application know that we want to update some state. In our case, will invoke some action that tells our state that we need to show/hide the `YesNoModal`.

Reducers are pure functions that will receive an initial state and an action, perform a logic based on this action, and return a new state.

Last but not least, the store is the mechanism that stores the whole global state of the application. We update this state invoking our actions and deal with them in the reducers.

It's hard to explain how Redux works in a few lines of text, but I promise you that it will become a lot clearer when we see some code.

Installing Redux

Let’s begin by installing Redux:

```
npm install redux -s
```

Besides Redux itself, let’s also install react-redux, an official binding for React that helps our components to know when the global state has changed and re-renders it according to that change:

```
npm install redux-react -s
```

Modals Reducer

The `modalsReducer` is where we will perform some logic based on actions triggered by the components. In this example, we have only one modal, but we can use this reducer to manage as many modals as we want.

The reducer consists of an initial state object and the function that will be invoked when we need to make some update on the state. The `modalsStore` function receives a state and an action as a parameter. The action tells what change we should make in our state.

If the type of the action is `ShowYesNoModal`, we return a new state setting the `showYesNoModal` value to `true`, indicating that we want to show the modal. Once the modal is open, we can close it by dispatching an action with the type `HideYesNoModal`, returning a new state with the prop `showYesNoModal = false`.

In order to connect this reducer to our components, we need to make some changes to the `index.js` file:

Basically, we are creating the store and connecting it to our components by wrapping our app with the `Provider` component. Our components can now dispatch actions and subscribe to state changes. Note that we can combine several reducers. In this example, we have only the `modalReducer`, but we can have `notificationReducer`, `alertReducer`, etc.

Now we can connect our `MainComponent`and start to dispatch actions in order to show the `YesNoModal`:

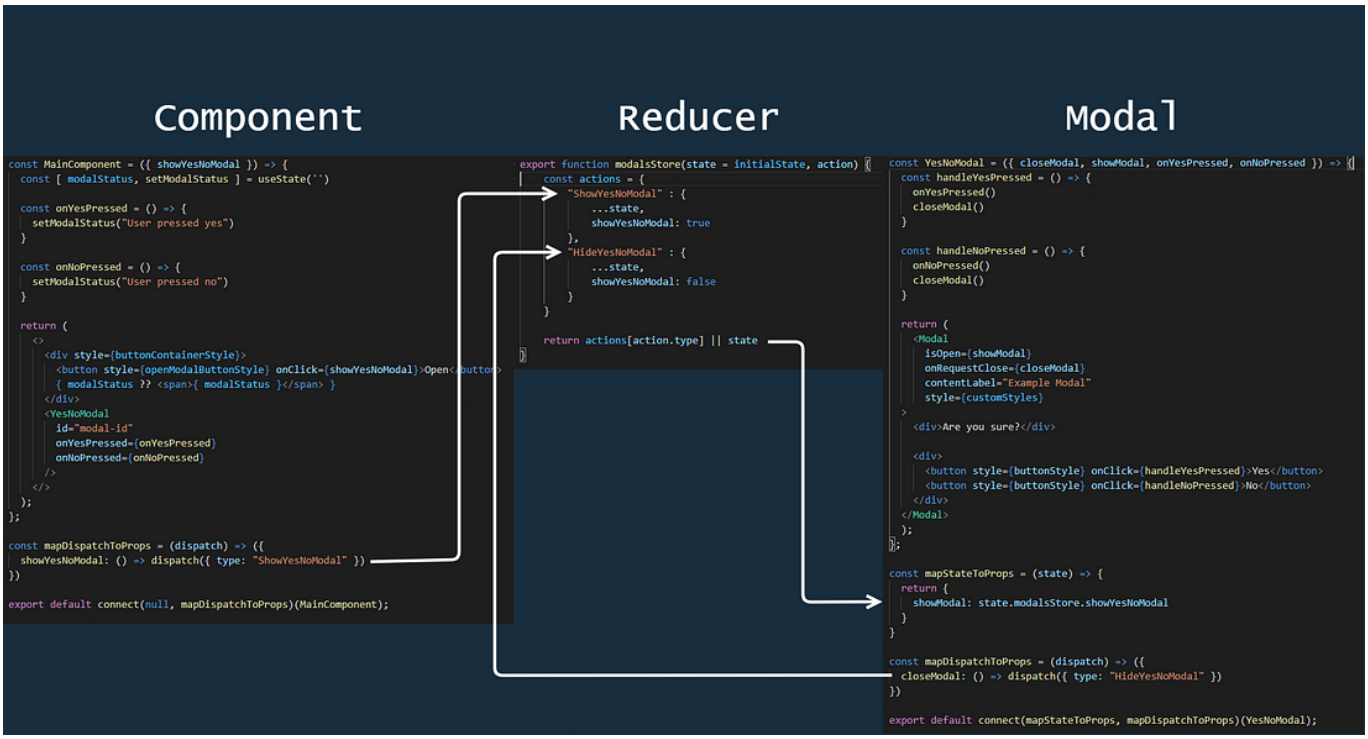
Now we don't have the modal open state in the `MainComponent` anymore. We have successfully removed the responsibility of the `MainComponent` to manage the modal state. Instead, we are receiving the function `showYesNoModal` by props, and it will be invoked when the modal should be open.

But where does this `showYesNoModal` functioncome from? This function is passed to props when we connect the `mapDispatchtoProps` to our component on line 36. Basically, we are dispatching an action of type `ShowYesNoModal` that will be captured by our reducer and trigger a state change.

Now we just need to subscribe to this state change in the `YesNoModal` itself in order to show the modal:

When we map state to props (line 33), we are just basically subscribing to any state change in the global store. The `mapStateToProps`function receives the global state object and maps it to another object containing a prop that manages the modal visibility.

Besides the `showModal` prop, the component is receiving a function called `closeModal`. Once invoked, it will dispatch an action with the type `HideYesNoModal` that will be captured by our reducer and update state, setting the prop that controls the visibility of the modal to `false`.



Dispatch/state update flow

And that's it: We are now controlling our modal by using the global state managed by Redux. At this point, you are probably asking yourself, “Why should I use Redux if I ended up writing more code to achieve basically the same result?”

In the example above, we only have one modal. What if we had several other modals? Would we want to manage the modal state each time we need it directly in the components? Probably not.

Once we write the first reducer and connect it to the rest of the app, implementing new reducers will be fairly straightforward. We just need to import them to our `index.js` file and combine them when creating the store.

To finish this off, I would like to share a real-world reducer that I implemented a long time ago in one of my projects. This is exactly what we discussed in this article:

The focus of this example was to deal with modals, but we can extend this approach to a lot of challenges that need to share this state control throughout the application, like alerts, notifications, authentication, etc.

That’s all for today, folks. I hope that this article has helped you in some way.

Take care and happy coding!

The source code is available on GitHub.