# Hands-on Lab 6: Using Presentational and Container Components with Redux

**Estimated time: 60 minutes**

## Exercise 1:

In the last chapter, we added complexity to both the state as well as the view-layer of our application. To support threads in our app, we nested message objects inside of thread objects in our state tree. By using reducer composition, we were able to break up the management of our more complex state tree into smaller parts.

We added a new React component to support our threaded model, ThreadTabs, which lets the user switch between threads on the view. We also added some complexity to existing components.

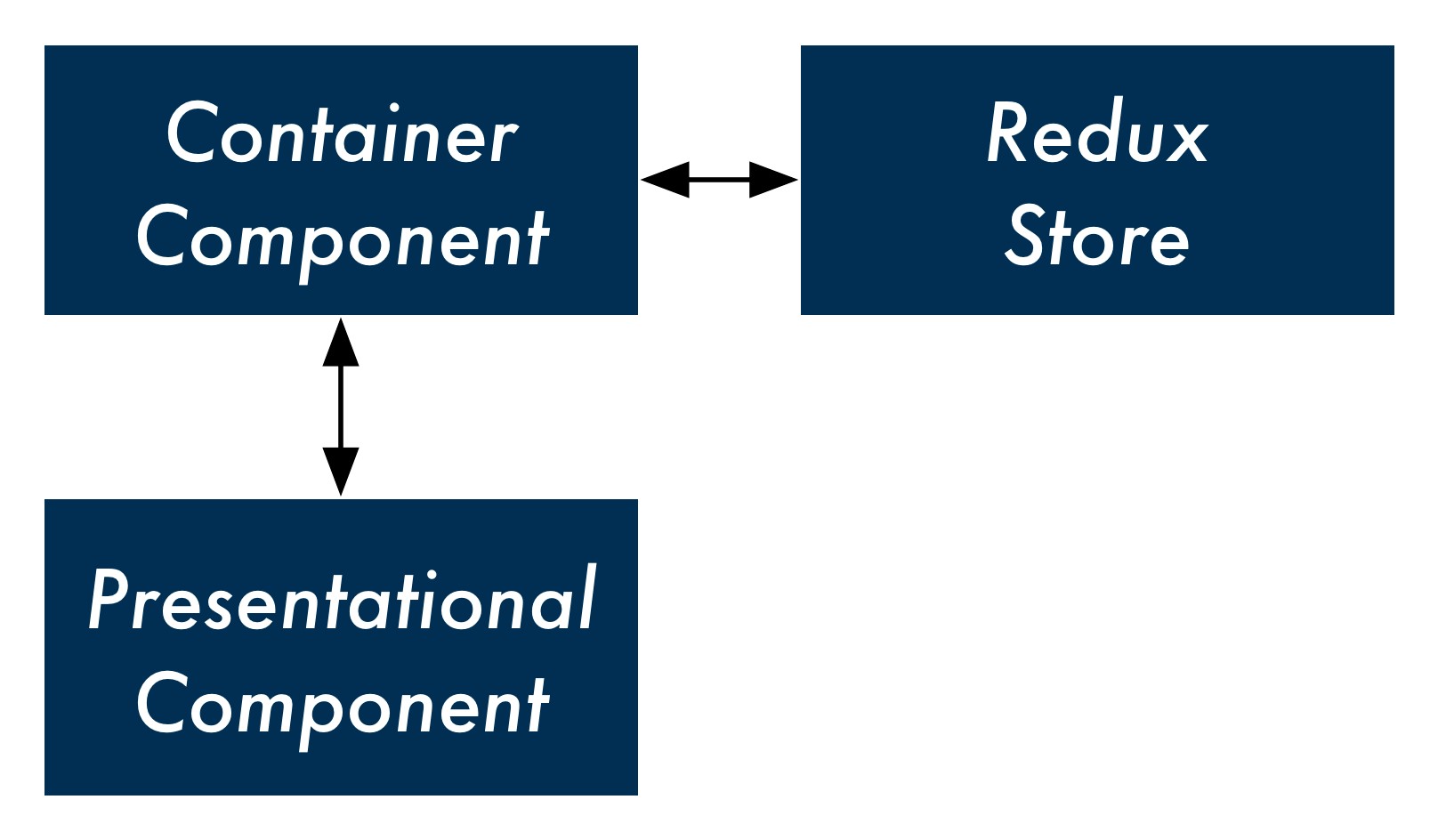
At the moment, we have four React components in our app. Every React component interacts directly with the Redux store. App subscribes to the store and uses getState() to read the state and passes down this state as props to its children. Child components dispatch actions directly to the store.

In this chapter, we’ll explore a new paradigm for organizing our React components. We can divide up our React components into two types: presentational components and container components. We’ll see how doing so limits knowledge of our Redux store to container components and provides us with flexible and re-usable presentational components.

### Presentational and container components

In React, a presentational component is a component that just renders HTML. The component’s only function is presentational markup. In a Redux-powered app, a presentational component does not interact with the Redux store.

The presentational component accepts props from a container component. The container component specifies the data a presentational component should render. The container component also specifies behavior. If the presentational component has any interactivity — like a button — it calls a prop-function given to it by the container component. The container component is the one to dispatch an action to the Redux store:



Take a look at the ThreadTabs component:

**redux/chat\_intermediate/src/complete/App-12.js**

**class** ThreadTabs **extends** React.Component {

handleClick = (id) => { store.dispatch({

type: 'OPEN\_THREAD', id: id,

}); };

render() {

**const** tabs = **this**.props.tabs.map((tab, index) => (

<div

key={index}

className={tab.active ? 'active item' : 'item'}

onClick={() => **this**.handleClick(tab.id)}

>

{tab.title}

</div> )); **return** (

<div className='ui top attached tabular menu'> {tabs}

</div>

);

}

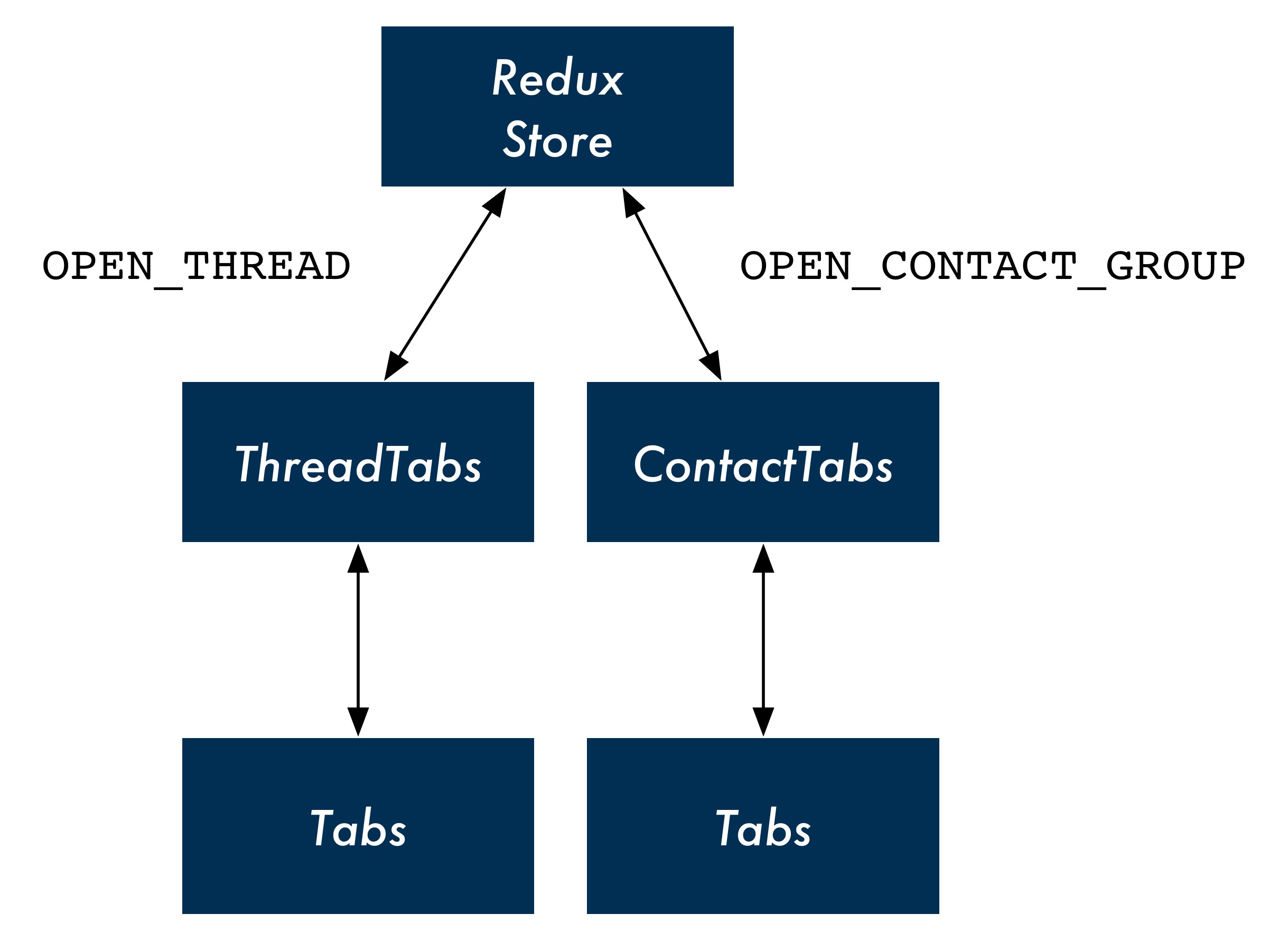
}

At the moment, this component both renders HTML (the text field input) and communicates with the store. It dispatches the OPEN\_THREAD action whenever a tab is clicked.

But what if we wanted to have another set of tabs in our app? This other set of tabs would probably have to dispatch another type of action. So we’d have to write an entirely different component even though the HTML it renders would be the same.

What if we instead made a generic tabs component, say Tabs? This presentational component would not specify what happens when the user clicks a tab. Instead, we could wrap it in a container component wherever we want this particular markup in our app. That container component could then specify what action to take by dispatching to the store.

We’ll call our container component ThreadTabs. It will do all of the communicating with the store and let Tabs handle the markup. In the future, if we wanted to use tabs elsewhere — say, in a “contacts” view that has a tab for each group of contacts — we could re-use our presentational component:



### Splitting up ThreadTabs

We’ll split up ThreadTabs by first writing the presentational component Tabs. This component will only be concerned with rendering the HTML — the array of horizontal tabs. It will also expect a prop, onClick. The presentational component will allow its container component to specify whatever behavior it wants when a tab is clicked.

Let’s add Tabs to App.js now. Write it above the current ThreadTab component. The JSX for the HTML markup is the same as before:

**redux/chat\_intermediate/src/complete/App-13.js**

**const** Tabs = (props) => (

<div className='ui top attached tabular menu'>

{ props.tabs.map((tab, index) => (

<div

key={index}

className={tab.active ? 'active item' : 'item'}

onClick={() => props.onClick(tab.id)}

>

{tab.title}

</div>

))

}

</div>

);

A unique aspect of our new presentational component is how it’s declared. So far, we’ve been using ES6 classes like this:

**class** App **extends** React.Component { *// ...*

}

React components declared in this manner are wrapped in React’s component API. This declaration gives the component all of the React-specific features that we’ve been using, like lifecycle hooks and state management.

However, as we cover in the “Advanced Components” chapter, React also allows you to declare stateless functional components. Stateless functional components, like Tabs, are just JavaScript functions that return markup. They are not special React objects.

Because Tabs does not need any of React’s component methods, it can be a stateless component.

In fact, all our presentational components can be stateless components. This reinforces their single responsibility of rendering markup. The syntax is terser. What’s more, the React core team recommends using stateless components whenever possible. Because these components are not “dressed up” with any of the capabilities of React component objects, the React team anticipates there will be many performance advantages introduced for stateless components in the near future.

As we can see, the first argument passed in to a stateless component is props:

**redux/chat\_intermediate/src/complete/App-13.js**

**const** Tabs = (props) => (

Because Tabs is not a React component object, it does not have the special property this.props. Instead, parents pass props to stateless components as an argument. So we’ll access this component’s props everywhere using props as opposed to this.props.

Our presentational component is ready. Let’s see what the container component that uses it looks like. Modify the current ThreadTabs component:

**redux/chat\_intermediate/src/complete/App-13.js**

**class** ThreadTabs **extends** React.Component {

render() { **return** (

<Tabs tabs={**this**.props.tabs} onClick={(id) => ( store.dispatch({

type: 'OPEN\_THREAD', id: id,

})

)}

/>

);

}

}

Although we don’t use any of React’s component methods, we’re still using an ES6 class component as opposed to declaring a stateless component. We’ll see why in a moment.

Our container component specifies the props and behavior for our presentational component. We set the prop tabs to this.props.tabs, specified by App. Next, we set the prop onClick to a function that calls store.dispatch(). We expect Tabs to pass the id of the clicked tab to this function.

If we were to test the app out now, we’d be happy to note that our new container/presentational component combination is working.

However, there’s one odd thing about ThreadTabs: It sends actions to the store directly with dispatch, yet at the moment it’s reading from the store indirectly through props (through this.props.tabs).

App is the one reading from the store and this data trickles down to ThreadTabs. But if ThreadTabs is dispatching directly to the store, is this indirection for reading from the store necessary?

Instead, we can have all of our container components be responsible for both sending actions to the store and reading from it.

In order to achieve this with ThreadTabs, we can subscribe directly to the store in componentDidMount, the same way that App does:

**redux/chat\_intermediate/src/complete/App-14.js**

**class** ThreadTabs **extends** React.Component {

componentDidMount() {

store.subscribe(() => **this**.forceUpdate()); }

Then, inside of render, we can read state.threads directly from the store with getState(). We’ll generate tabs here using the same logic that we used in App:

**redux/chat\_intermediate/src/complete/App-14.js**

render() {

**const** state = store.getState();

**const** tabs = state.threads.map(t => (

{

title: t.title,

active: t.id === state.activeThreadId,

id: t.id,

}

));

Now we don’t need to read from this.props at all. We pass Tabs the tabs variable that we created:

**redux/chat\_intermediate/src/complete/App-14.js**

**return** (

<Tabs tabs={tabs} onClick={(id) => (

store.dispatch({

type: 'OPEN\_THREAD', id: id,

})

)}

/>

);

Our Tabs component is purely presentational. It specifies no behavior of its own and could be dropped-in anywhere in the app.

The ThreadTabs component is a container component. It renders no markup. Instead, it interfaces with the store and specifies which presentational component to render. The container component is the connector of the store to the presentational component.

Our presentational and container component combination, **in full**:

**redux/chat\_intermediate/src/complete/App-14.js**

**const** Tabs = (props) => (

<div className='ui top attached tabular menu'>

{

props.tabs.map((tab, index) => (

<div

key={index}

className={tab.active ? 'active item' : 'item'} onClick={() => props.onClick(tab.id)}

>

{tab.title}

</div>

))

}

</div> );

**class** ThreadTabs **extends** React.Component {

componentDidMount() {

store.subscribe(() => **this**.forceUpdate());

}

render() {

**const** state = store.getState();

**const** tabs = state.threads.map(t => (

{

title: t.title,

active: t.id === state.activeThreadId, id: t.id,

} ));

**return** (

<Tabs

tabs={tabs} onClick={(id) => (

store.dispatch({

type: 'OPEN\_THREAD', id: id,

})

)}

/>

);

}

}

In addition to the ability to re-use our presentational component elsewhere in the app, this paradigm gives us another significant benefit: We’ve de-coupled our presentational view code entirely from our state and its actions. As we’ll see, this approach isolates all knowledge of Redux and our store to our app’s container components. This minimizes the switching costs in the future. If we wanted to move our app to another state management paradigm, we wouldn’t need to touch any of our app’s presentational components.

### Splitting up Thread

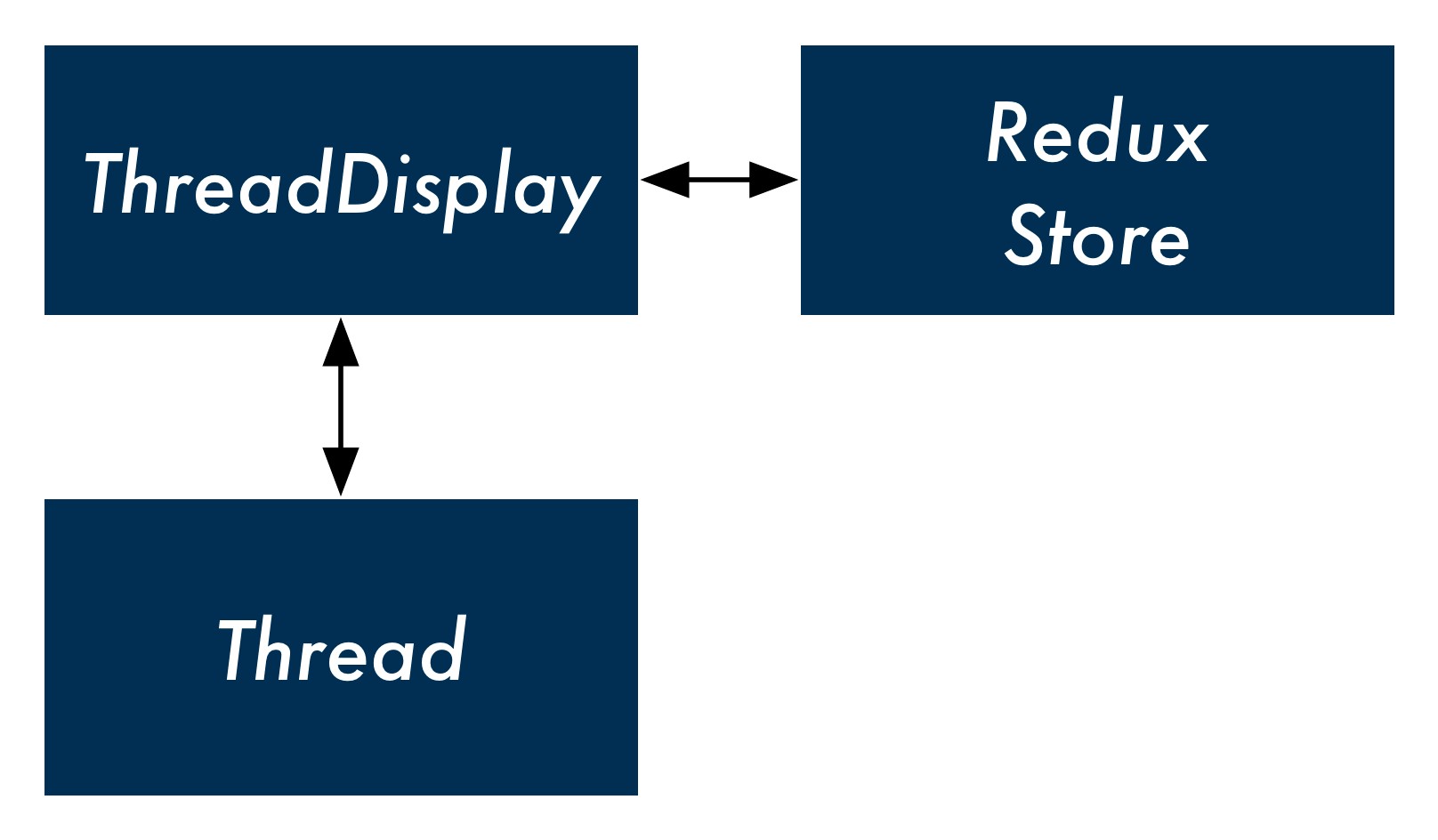
Let’s continue refactoring with our new design pattern.

Thread receives the thread as a prop and contains all the markup for rendering the messages inside of that thread as well as MessageInput. The component will dispatch to the store a DELETE\_MESSAGE action if a message is clicked.

Part of rendering the view for a thread involves rendering the view for its messages. We could have separate container and presentational components for threads and messages. In this setup, the presentational component for a thread would render the container component for a message.

But because we don’t anticipate ever rendering a list of messages outside of a thread, it’s reasonable to just have the container component for the thread also manage the presentational component for a message.

We can have one container component, ThreadDisplay. This container component will render the presentational component Thread:

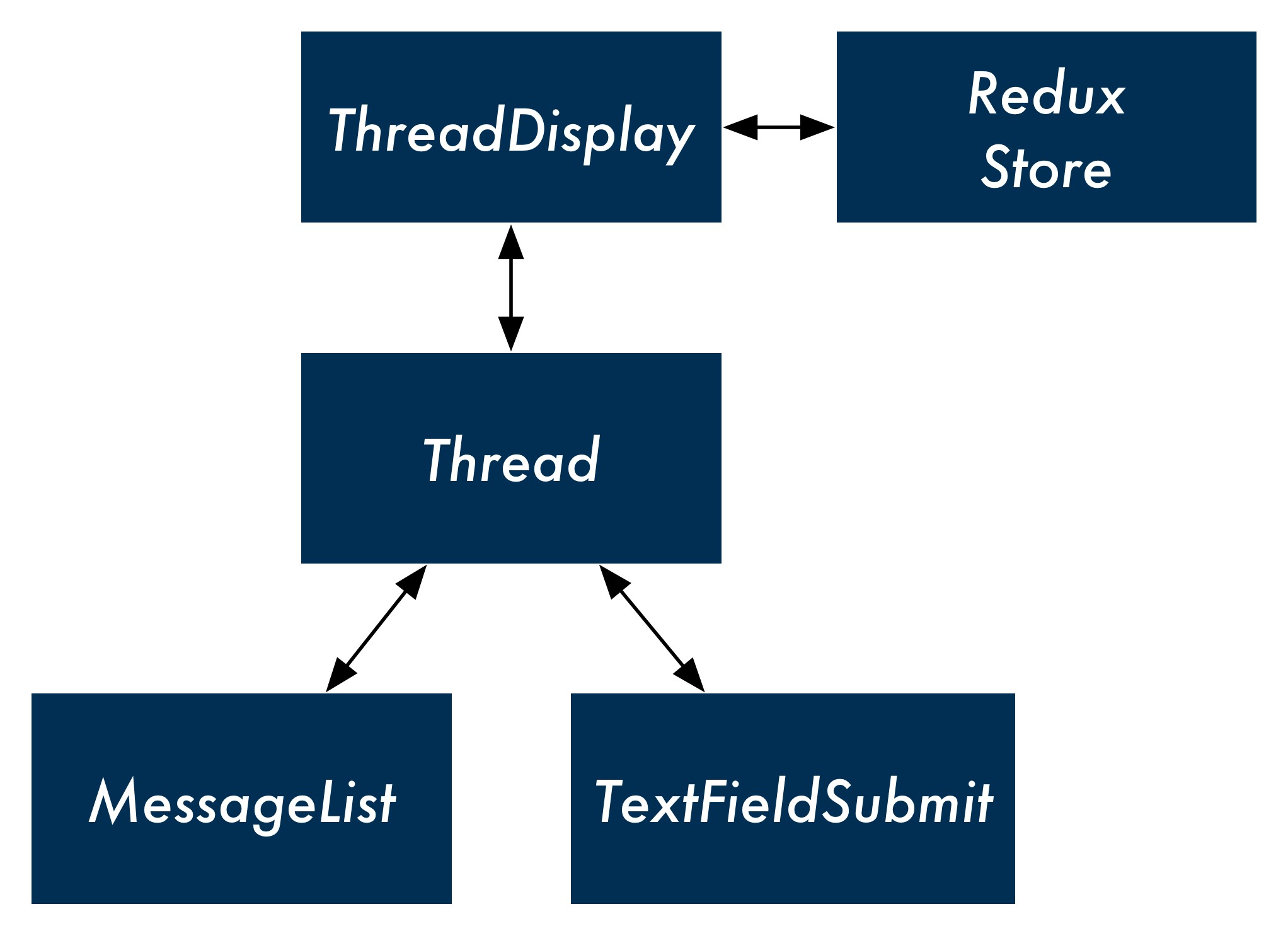


For the list of messages, we can have Thread render another presentational component, MessageList.

But what about the component MessageInput? Like our previous version of ThreadTabs, the component contains two responsibilities. The component renders markup, a single text field with a submit button. In addition, it specifies the behavior for what should happen when the form is submitted.

We could, instead, just have a generic presentational component. TextFieldSubmit only renders markup and allows its parent to specify what happens when the text field is submitted. ThreadDisplay, through Thread, could control the behavior of this text field.

With this design, we’d have one container component for a thread up top. The presentational component Thread would be a composite of two child presentational components, MessageList and TextFieldSubmit:



Let’s first rename our current Thread component to ThreadDisplay to avoid confusion:

*// Rename from `Thread`*

**class** ThreadDisplay **extends** React.Component {

*// ...*

};

We’ll begin at the bottom, writing the presentational components TextFieldSubmit and MessageList. We’ll work our way up to Thread and then ThreadDisplay.

**TextFieldSubmit**

Like with ThreadTabs, MessageInput has two distinct roles: the component both renders the HTML for an input field but also specifies what the behavior around submitting that input field should be (dispatching ADD\_MESSAGE).

If we remove the dispatch call from MessageInput, we’d be left with a generic component that just rendered markup: a text field with an adjacent submit button. The presentational component will allow its container component to specify whatever behavior it wants when the input field is submitted.

Let’s rename MessageInput to TextFieldSubmit to make it more generic. The only additional change we need to make is in handleSubmit(). We’ll have TextFieldSubmit expect a single prop, onSubmit. Instead of dispatching to the store directly, it will invoke this prop-function: **redux/chat\_intermediate/src/complete/App-14.js**

**class** TextFieldSubmit **extends** React.Component {

state = {

value: '',

};

onChange = (e) => { **this**.setState({

value: e.target.value,

})

};

handleSubmit = () => {

**this**.props.onSubmit(**this**.state.value); **this**.setState({

value: '',

});

};

**MessageList**

The MessageList component will accept two props: messages and onClick. As before, this presentational component will not specify any behavior. As a stateless component, it will only render HTML.

Write it below TextFieldSubmit and above the ThreadDisplay component in App.js:

**redux/chat\_intermediate/src/complete/App-14.js**

**const** MessageList = (props) => (

<div className='ui comments'>

{ props.messages.map((m, index) => (

<div

className='comment'

key={index}

onClick={() => props.onClick(m.id)}

>

<div className='text'>

{m.text}

<span className='metadata'>@{m.timestamp}</span> </div>

</div>

))

}

</div>

);

The map that we perform over props.messages is the same logic we had previously in Thread. We perform it in-line, nested inside of the div tag which is responsible for styling. The three changes:

* We perform the map over props.messages as opposed to this.props.threads
* The onClick attribute is now set to props.onClick
* For brevity, we’re using the variable m in place of message

**Thread**

We have two presentational components related to displaying a thread. One is MessageList, which renders all of the messages in that thread. The other is TextFieldSubmit, a generic text field entry that we’re going to have submit new messages to the thread.

We’re collecting these two presentational components under Thread, another presentational component. The container component ThreadDisplay will render Thread which in turn will render MessageList and TextFieldSubmit.

We anticipate that ThreadDisplay will pass Thread three props:

* thread: The thread itself
* onMessageClick: The message click handler
* onMessageSubmit: The text field submit handler

We’ll have Thread pass along the appropriate props to each of its child presentational components:

**redux/chat\_intermediate/src/complete/App-14.js**

**const** Thread = (props) => (

<div className='ui center aligned basic segment'>

<MessageList messages={props.thread.messages} onClick={props.onMessageClick}

/>

<TextFieldSubmit onSubmit={props.onMessageSubmit}

/>

</div>

);

**ThreadDisplay**

ThreadDisplay (previously named Thread) is our container component. Like with our two previous container components, it will subscribe to the store. The component will be responsible for both reading from the store and dispatching actions to it.

First, subscribe to the store in componentDidMount:

**redux/chat\_intermediate/src/complete/App-14.js**

**class** ThreadDisplay **extends** React.Component { componentDidMount() {

store.subscribe(() => **this**.forceUpdate()); }

ThreadDisplay will read from the store directly to get the active thread. The container component will then render Thread, passing in the props thread and onMessageClick.

Inside of render, we’ll use the same logic we had used in App to grab the active thread:

**redux/chat\_intermediate/src/complete/App-14.js**

render() {

**const** state = store.getState();

**const** activeThreadId = state.activeThreadId;

**const** activeThread = state.threads.find(

t => t.id === activeThreadId

);

onMessageSubmit:

**redux/chat\_intermediate/src/complete/App-14.js**

**return** (

<Thread thread={activeThread} onMessageClick={(id) => (

store.dispatch({

type: 'DELETE\_MESSAGE', id: id,

}) )} onMessageSubmit={(text) => (

store.dispatch({

type: 'ADD\_MESSAGE', text: text,

threadId: activeThreadId,

})

)}

/>

);

Our container component ThreadDisplay, **in full:**

**redux/chat\_intermediate/src/complete/App-14.js**

**class** ThreadDisplay **extends** React.Component { componentDidMount() {

store.subscribe(() => **this**.forceUpdate());

}

render() {

**const** state = store.getState();

**const** activeThreadId = state.activeThreadId;

**const** activeThread = state.threads.find(

t => t.id === activeThreadId

);

**return** (

<Thread thread={activeThread} onMessageClick={(id) => (

store.dispatch({

type: 'DELETE\_MESSAGE',

id: id,

}) )} onMessageSubmit={(text) => (

store.dispatch({

type: 'ADD\_MESSAGE', text: text,

threadId: activeThreadId,

})

)}

/>

);

}

}

We’ve split up all of our view components into container and presentational components. Our two container components are communicating directly with the store, performing both reads (getState()) and sending actions (dispatch()).

Because of this, we actually don’t need App to talk to the store at all. The render function for App right now reads from the store and then sends props down to its children. Now that its children are communicating with the store, it’s no longer necessary that App supply them with any props.

## Removing store from App

Because our container components are now interfacing with the store themselves, we can remove all communication with the store from App. In fact, we can just turn App into a stateless component:

**redux/chat\_intermediate/src/complete/App-15.js**

**const** App = () => (

<div className='ui segment'>

{*/\* `Thread` changed to `ThreadDisplay` below \*/*}

<ThreadTabs />

<ThreadDisplay />

</div>

);

Due to a combination of our new container and presentational component paradigm and a Redux state manager, the top-level component for this app is just specifying what container components to include on the page. All of the responsibility for reading and writing to state has been pushed down to each one of our container components.

Because we’re not dispatching actions directly from our leaf components, we’ve isolated all knowledge of the Redux store to our container components. We’re free to re-use our presentational components in other contexts within our app. What’s more, if we wanted to switch state management paradigms from Redux to something else, we would only need to modify our container components.

Our container components all look pretty similar. They subscribe to the store and then map state and actions to props on a presentational component. In the next section, we’ll explore an option for reducing some of the ceremony around writing container components. But for now, let’s pause briefly to verify that everything still works as before.

### Try it out

Save App.js. While we’ve made some big architectural changes to our React components, viewing the app at http://localhost:3000 everything is working as before.

## Generating containers with react-redux

Looking at our two container components (ThreadTabs and ThreadDisplay), they have similar behavior:

* They subscribe to the store in componentDidMount.
* They might have some logic to massage data from state into a format fit as a prop for the presentational component (like tabs in ThreadTabs).
* They map actions on the presentational component (like click events) to functions that dispatch to the store.

Because container components rely on presentational components to render markup, they just contain “glue” code between the store and presentational components.

A popular library, react-redux, gives us a couple conveniences when writing React apps that use a Redux store. The primary convenience is its connect() function.

The connect() function in react-redux generates container components. For each presentational component, we can write functions that specify how state should map to props and how events should map to dispatches.

Let’s see what this looks like in action.

### The Provider component

Before we can use connect() to generate container components, we need to make a small addition to our app.

Right now, our containers are referencing the store variable directly. This works because we declare this variable inside the same file as our components.

In order for connect() to be able to generate container components, it needs some canonical mechanism for containers to access the Redux store. The function can’t rely on the store variable being declared and available in the same file.

To solve this, the react-redux library supplies a special Provider component. You can wrap your top-level component in Provider. Provider will then make the store available to all components via React’s context feature.

When we use connect() to generate container components, those container components will assume that the store is available to them via context.

Context is a React feature that you can use to make certain data available to all React components. We talk about context in the “Advanced Component Configuration” chapter.

With props, data is explicitly passed down the component hierarchy. A parent must specify what data is available to its children.

Context allows you to make data available implicitly to all components in a tree. Any component can “opt-in” to receiving context and that component’s parent doesn’t need to do anything.

While we cover context elsewhere in the book, it is a rarely used feature in React. The React core team actively discourages its use, except in special circumstances.

### Wrapping App in Provider

Inside of package.json, we’re already including the react-redux library:

"react-redux": "5.0.4",

To use the Provider component, we first include it at the top of App.js:

**redux/chat\_intermediate/src/complete/App-15.js**

**import** { Provider } from 'react-redux';

To allow our generated container components to access the store through context, we need to wrap App in Provider.

At the bottom of App.js, we’ll declare a new component, WrappedApp. WrappedApp will return the App component wrapped in the Provider component. We’ll export WrappedApp from the file:

**redux/chat\_intermediate/src/complete/App-15.js**

**const** WrappedApp = () => (

<Provider store={store}>

<App />

</Provider>

);

**export default** WrappedApp;

Provider expects to receive the prop store. The store is now available anywhere in our component hierarchy under the context variable store.

### Using connect() to generate ThreadTabs

The ThreadTabs component connects the presentational component Tabs with our Redux store. It does so by:

* Subscribing to the store in componentDidMount
* Creating a tabs variable based on the store’s threads property and using that for the prop tabs on Tabs
* Setting the onClick prop on Tabs to a function that dispatches an OPEN\_THREAD action

We can use connect() to generate this component.

We need to pass two arguments to connect(). The first will be a function that maps the state to the props of Tabs. The second will be a function that maps dispatch calls to the component’s props.

We’ll see how this works by implementing it.

Mapping state to props

First, import the connect() function from the react-redux library:

**redux/chat\_intermediate/src/complete/App-16.js**

**import** { Provider, connect } from 'react-redux';

At the moment, we perform our “mapping” between the state and the props for Tabs inside of the render function for ThreadTabs by creating the tabs variable.

We’ll write a function that connect() will use to perform this same operation. We’ll call this function mapStateToTabsProps().

Whenever the state is changed, this function will be invoked to determine how to map the new state to the props for Tabs.

Declare the function above ThreadTabs in App.js. The function expects to receive the state as an argument:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapStateToTabsProps = (state) => {

We can copy and paste the logic that we had in ThreadTabs to produce the variable tabs, based on state.threads:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** tabs = state.threads.map(t => (

{

title: t.title,

active: t.id === state.activeThreadId, id: t.id,

}

));

Our state-to-props mapping function needs to return an object. The properties on this object are the prop names for Tabs.

Because the prop is called tabs and the variable we’re setting it to is also tabs, we can use the ES6 object shorthand:

**return** {

tabs,

};

};

Our mapStateToTabsProps() function, **in full**:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapStateToTabsProps = (state) => {

**const** tabs = state.threads.map(t => (

{ title: t.title, active: t.id === state.activeThreadId, id: t.id,

} ));

**return** {

tabs,

};

};

This function encapsulates the logic that was previously in ThreadTabs, describing how the state maps to the prop tabs for Tabs. Now we have to do the same for the prop onClick, which maps to a dispatch call.

Mapping dispatches to props

We’ll declare this function below mapStateToTabsProps(). We’ll call it mapDispatchToTabsProps():

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapDispatchToTabsProps = (dispatch) => (

We’ll pass this function second to connect(). It will be invoked on setup with dispatch passed in as an argument.

Like with mapStateToTabsProps(), we’ll return an object that maps the prop onClick to the function that will perform the dispatching. This function is identical to the one that ThreadTabs previously specified:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapDispatchToTabsProps = (dispatch) => (

{

onClick: (id) => (

dispatch({

type: 'OPEN\_THREAD', id: id,

})

),

}

);

We now have a function that maps the store’s state to the prop tabs on Tabs and another that maps the prop onClick to a function that dispatches OPEN\_THREAD.

We can now replace our ThreadTabs component by using connect(). Delete the entire ThreadTabs component currently in App.js.

The first argument to connect() is the function that maps the state to props. The second argument is the function that maps props to dispatch functions. connect() returns a function that we will immediately invoke with the presentational component we’d like to “connect” to the store with our container component:

*// Signature of `connect()`*

*// (Note this is partial, we see the full signature later)* connect(

mapStateToProps(state), mapDispatchToProps(dispatch),

)(PresentationalComponent)

Let’s use connect() to create ThreadTabs:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** ThreadTabs = connect(

mapStateToTabsProps, mapDispatchToTabsProps

)(Tabs);

On the surface it may not look like it, but ThreadTabs is a React container component, not too unlike the one we had before.

### Using connect() to generate ThreadDisplay

The next component we’ll generate with connect() is ThreadDisplay. The container component specifies three props on Thread:

* thread
* onMessageClick
* onMessageSubmit

**State to props**

We’ll call this mapping function mapStateToThreadProps(). It maps one prop:

* thread: maps to the active thread in state

**Dispatch to props**

We’ll call this mapping function mapDispatchToThreadProps(). It maps two props:

* onMessageClick: maps to a function that dispatches DELETE\_MESSAGE
* onMessageSubmit: maps to a function that dispaches ADD\_MESSAGE

**mapStateToThreadProps()**

We’ll write our state-to-props glue function above ThreadDisplay in App.js.

mapStateToThreadProps() accepts the argument state. We have it return an object that maps the thread property to the active thread in state:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapStateToThreadProps = (state) => (

{

thread: state.threads.find(

t => t.id === state.activeThreadId

),

}

);

This follows from the same logic that ThreadDisplay used to set the thread property of Thread. **mapDispatchToThreadProps()**

Below mapStateToThreadProps(), we’ll write our dispatch-to-props glue function.

The first dispatch prop we’ll write is that for onMessageClick. This function accepts an id and dispatches a DELETE\_MESSAGE action. Again, this logic matches that of ThreadDisplay:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapDispatchToThreadProps = (dispatch) => (

{

onMessageClick: (id) => (

dispatch({

type: 'DELETE\_MESSAGE',

id: id,

})

),

Next, we need to define the dispatch function for onMessageSubmit.

As you recall, inside of ThreadDisplay, this function dispatched an ADD\_MESSAGE action like this:

store.dispatch({

type: 'ADD\_MESSAGE', text: text,

threadId: activeThreadId,

})

connect() will not pass our dispatch-to-props function the state. How do we get the active thread’s id, then?

For the purposes of our app, this will work just fine. store is defined in this file, so we could just read from it directly.

But, it’s preferable that the mapping functions that you pass to connect() do not access the store directly.

Why so?

We’re about to replace our declaration of ThreadDisplay with a container component generated by connect(). Powerfully, after we’ve done so, the only reference to the store from our React components will be right here:

<Provider store={store}>

<App />

</Provider>

Isolating the reference to store to just one location has two huge benefits.

The first benefit is one we covered earlier when we discussed container components. The less references we have to store, the less work we’d have to do if we wanted to move from Redux to some other state management paradigm. Our mapping functions are just JavaScript functions and could conceivably perform mapping for some other type of store, so long as the API for the store was somewhat similar to that of our Redux store.

The more immediate benefit is for testing. When writing tests for a React app, you might want to inject a fake store into your app. With a fake store, you could specify what it should return for each spec or assert that certain methods on that store are called.

By passing the store as a prop to Provider and not referencing it directly anywhere else in the app, we could easily swap in a mock store during tests.

So, we need the id of the thread we’re displaying in order to dispatch our ADD\_MESSAGE action. But we don’t have access to this property inside our dispatch-to-props function.

connect() allows you to pass in a third function, what it calls mergeProps.

So, in full, the three functions you can pass to connect():

*// Full function signature of `connect()`* connect(

mapStateToProps(state, [ownProps]), mapDispatchToProps(dispatch, [ownProps]), mergeProps(stateProps, dispatchProps, [ownProps])

)

mergeProps is called with two arguments: stateProps and dispatchProps. These are just the objects that are returned by mapStateToProps and mapDispatchToProps.

So we can pass connect() a third function, mergeThreadProps(). This function will be invoked with two arguments:

* The object we return in mapStateToThreadProps()
* The object we return in mapDispatchToThreadProps()

connect() will use the object returned by mergeThreadProps() as the final object to determine the props for Thread.

In sequence, connect() will do the following:

1. Call mapStateToThreadProps() with state
2. Call mapDispatchToThreadProps() with dispatch
3. Call mergeThreadProps() with the results of the two previous map functions (stateProps and dispatchProps)
4. Use the object returned by mergeThreadProps() to set the props on Thread

Inside of mergeThreadProps(), we’ll need access to two items to create our ADD\_MESSAGE dispatch function:

* The id of the thread
* The dispatch function itself

We’ll get the id of the thread via stateProps, as that object has the full thread object under thread.

To get access to dispatch, we can pass it along in mapDispatchToThreadProps(). As such, our mapDispatchToThreadProps() function will define two properties:

* onMessageClick
* dispatch

The dispatch-to-props function, in full:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mapDispatchToThreadProps = (dispatch) => (

{

onMessageClick: (id) => (

dispatch({

type: 'DELETE\_MESSAGE',

id: id,

}) ),

dispatch: dispatch,

}

);

Now we’ll define our final mapping function. Again, this “merging” function will be passed the results of our state-to-props mapping function and our dispatch-to-props mapping function. The object it returns is the one that connect() will use to bind the props of Thread.

We’ll declare this function below mapDispatchToThreadProps(). Our merging function receives two arguments:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mergeThreadProps = (stateProps, dispatchProps) => (

We want to create a new object that contains:

* All the properties from stateProps
* All the properties from dispatchProps
* An additional property, onMessageSubmit

Let’s see what this looks like:

**redux/chat\_intermediate/src/complete/App-16.js**

**const** mergeThreadProps = (stateProps, dispatchProps) => (

{

...stateProps, ...dispatchProps, onMessageSubmit: (text) => (

dispatchProps.dispatch({

type: 'ADD\_MESSAGE',

text: text,

threadId: stateProps.thread.id,

})

),

}

);

We copy both stateProps and dispatchProps over to our new object using the spread operator

(...).

onMessageSubmit dispatches the same ADD\_MESSAGE action that ThreadDisplay previously dispatched. Note that we’re grabbing the dispatch function from dispatchProps:

**redux/chat\_intermediate/src/complete/App-16.js**

dispatchProps.dispatch({

And then, we’re grabbing the thread’s id from stateProps:

**redux/chat\_intermediate/src/complete/App-16.js**

threadId: stateProps.thread.id,

With our two mapping functions and one merge function prepared, we can now generate ThreadDisplay with connect().

We’ll declare ThreadDisplay below mergeThreadProps():

**redux/chat\_intermediate/src/complete/App-16.js**

**const** ThreadDisplay = connect(

mapStateToThreadProps, mapDispatchToThreadProps, mergeThreadProps

)(Thread);

Be sure to remove the old declaration of the ThreadDisplay component from App.js.

Using the mergeProps argument of connect() feels like a bit of a workaround. This is because the parameters for using connect() are quite strict. This is by design. The library enforces this usage for both performance reasons and to prevent a few possible developer mistakes.

However, with our merge function, we got connect() to generate a ThreadDisplay component as desired. We removed some of the boilerplate around our container components. And, we’ve isolated the connection between Redux and React to a single area — as a prop for Provider. Let’s verify everything is working properly.

Try it out

Make sure the server is running. Navigate to http://localhost:3000 and observe that all of the functionality of the app is working.

## Action creators

Right now, in every instance where we want to dispatch an action, we declare an action object of a certain type as well as its required properties. For example, for the DELETE\_MESSAGE action:

dispatch({ type: 'DELETE\_MESSAGE', id: id,

})

In the current iteration of the app, we only dispatch each type of action from a single location. As Redux apps grow, it’s common for the same action to be dispatched from multiple locations.

A popular pattern is to use action creators to create the action objects. An action creator is a function that returns an action object. An action creator for our DELETE\_MESSAGE action would look like this:

*// Example action creator for `DELETE\_MESSAGE`* **function** deleteMessage(id) { **return** { type: 'DELETE\_MESSAGE', id: id,

};

}

Then, anywhere in our app, if we wanted to dispatch a DELETE\_MESSAGE action, we could just use our action creator: dispatch(deleteMessage(id));

It’s a light abstraction that hides the action’s type as well as its property names from React components. More importantly, using action creators enables certain advanced patterns, like coupling an API request with an action dispatch.

Let’s swap out our action objects and use action creators instead.

First, we’ll write our action creators. Let’s declare them below the line where we initialize the store with createStore().

We already saw what the deleteMessage() action creator looks like. The action creator is a function that accepts the id of the message to be deleted. It then returns an object of type DELETE\_MESSAGE:

**redux/chat\_intermediate/src/complete/App-17.js**

**function** deleteMessage(id) { **return** {

type: 'DELETE\_MESSAGE',

id: id,

};

}

The action creator addMessage() expects both a text and threadId argument:

**redux/chat\_intermediate/src/complete/App-17.js**

**function** addMessage(text, threadId) {

**return** {

type: 'ADD\_MESSAGE', text: text, threadId: threadId,

};

}

And openThread expects the id of the thread to open:

**redux/chat\_intermediate/src/complete/App-17.js**

**function** openThread(id) { **return** {

type: 'OPEN\_THREAD',

id: id,

};

}

We can now work down App.js and replace action objects with our new action creators.

Inside mapDispatchToTabsProps():

**redux/chat\_intermediate/src/complete/App-17.js**

**const** mapDispatchToTabsProps = (dispatch) => (

{

onClick: (id) => (

dispatch(openThread(id))

),

}

);

Then mapDispatchToThreadProps():

**redux/chat\_intermediate/src/complete/App-17.js**

**const** mapDispatchToThreadProps = (dispatch) => (

{

onMessageClick: (id) => (

dispatch(deleteMessage(id))

),

dispatch: dispatch,

}

);

And finally mergeThreadProps():

**redux/chat\_intermediate/src/complete/App-17.js**

**const** mergeThreadProps = (stateProps, dispatchProps) => (

{

...stateProps, ...dispatchProps, onMessageSubmit: (text) => (

dispatchProps.dispatch(

addMessage(text, stateProps.thread.id) )

),

}

);

Another benefit of using action creators is that they list out all of the possible actions in our system in one place. We no longer have to infer the shapes of possible actions by hunting through the reducers or React components. This is exacerbated as the number of actions in a system grows.