# **Hands-on Lab 5: Intermediate Redux**

**Estimated time: 60 minutes**

## Exercise 1:

**Step 1:**

**Preparation**

Inside of the code download that came with the lab, navigate to redux/chat\_intermediate:

$ cd redux/chat\_intermediate

This app is setup identically to the chat app in the last chapter, powered by create-react-app.

$ ls

README.md

nightwatch.json

package.json

public

semantic

semantic.json

src

tests

yarn.lock

Checking out src/:

$ ls src/

App.js

complete

index.css

index.js

Again, App.js is where we’ll be working. It contains the app as we left it in the previous chapter. And again, complete/ contains each iteration of App.js as we build it up over the next two chapters. index.js currently includes the final version of App.js and mounts it to the DOM.

As usual, run npm install to install all of the dependencies for the project:

$ npm install

And then execute npm start to boot the server:

$ npm start

View the completed app by visiting http://localhost:3000 in your browser.

In this iteration of the chat app, our app has threads. Each message belongs to a particular thread with another user. We can switch between threads using the tabs at the top.

As in the last iteration, note that we can add messages with the text field at the bottom as well as delete messages by clicking on them.

To begin, let’s swap in App.js in src/index.js:

import React from "react";

import ReactDOM from "react-dom";

import App from "./App";

**Using createStore() from the redux library**

In the last chapter, we implemented our own version of createStore(). At the top of src/App.js, you’ll find the createStore() function just as we left it. The store object that this function creates has three methods: getState(), dispatch(), and subscribe().

As we noted in the last chapter, our createStore() function is very similar to that which ships with the redux library.

Let’s remove our implementation and use the one from redux.

In package.json, we already include the redux library:

"redux": "3.6.0",

We can import the createStore() function from the library:

**redux/chat\_intermediate/src/complete/App-1.js**

import { createStore } from 'redux';

Now we can remove our own createStore() definition from App.js.

**Try it out**

To verify everything is working properly, ensure the server is running:

$ npm start

And then check out the app on http://localhost:3000.

Behavior for the app will be the same as we left it in the previous chapter. We can add new messages and click on them to delete.

There are a couple subtle behavioral differences between our createStore() and the one that ships with the Redux library. Our app has yet to touch on them. We’ll address these differences when they come up.

**Representing messages as objects in state**

Our state up to this point has been simple. State has been an object, with a messages property. Each message has been a string:

// Example of our state object so far

{ messages: [

'Roger. Eagle is undocked',

'Looking good.',

],

}

To bring our app closer to a real-world chat app, each message will need to carry more data. For example, we might want each message to specify when it was sent or who sent it. To support this, we can use an object to represent each message as opposed to a string.

For now, we’ll add two properties to each message, timestamp and id:

// Example of our new state object

{ messages: [

// An example message

// messages are now objects

{

text: 'Roger. Eagle is undocked',

timestamp: '1461974250213',

id: '9da98285-4178',

},

// ...

]

}

The Date.now() function in JavaScript returns a number representing the number of milliseconds since January 1, 1970 00:00 U TC. This is called “Epoch” or “Unix” time. We’re using this representation for the timestamp property above.

In order to support messages that are objects, we’ll need to tweak our reducer as well as our React components.

**Updating ADD\_MESSAGE**

The reducer function we wrote in the last chapter handles two actions, ADD\_MESSAGE and DELETE\_MESSAGE.

Let’s start by updating the ADD\_MESSAGE action handler.

As you recall, the ADD\_MESSAGE action currently contains a message property:

{ type: 'ADD\_MESSAGE', message: 'Looking good.',

}

reducer() receives this action and returns a new object with a messages property. messages is set to a new array that contains the previous state.messages with the new message appended to it:

**redux/chat\_intermediate/src/complete/App-1.js**

function reducer(state, action) {

if (

action.type === 'ADD\_MESSAGE') {

return {

messages: state.messages.concat(action.message),

};

Let’s tweak our ADD\_MESSAGE action so that it uses the property name text instead of message:

// Example of what our new ADD\_MESSAGE will look like

{

type: 'ADD\_MESSAGE',

text: 'Looking good.',

}

text matches the property name that we’ll be using for the message object.

Next, let’s modify our reducer’s ADD\_MESSAGE handler so that it uses message objects as opposed to string literals.

We will give each message object a unique identifier. We include the uuid library in our package.json. Let’s import it at the top of src/App.js:

**redux/chat\_intermediate/src/complete/App-2.js**

import uuid from 'uuid';

Next, let’s modify ADD\_MESSAGE so that it creates a new object to represent the message. It will use action.text for the text property and then generate a timestamp and an id:

**redux/chat\_intermediate/src/complete/App-2.js**

if (action.type === 'ADD\_MESSAGE') {

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

We’ll use concat again. This time, we’ll use concat to return a new array that contains state.messages and our newMessage object:

return {

messages: state.messages.concat(newMessage),

};

**Our modified ADD\_MESSAGE handler in full:**

**redux/chat\_intermediate/src/complete/App-2.js**

if (action.type === 'ADD\_MESSAGE') {

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

return {

messages: state.messages.concat(newMessage),

};

**Updating DELETE\_MESSAGE**

The DELETE\_MESSAGE action up until now contained an index, the index of the message in the state.messages array to be deleted:

{ type: 'DELETE\_MESSAGE', index: 5,

}

Now that all of our messages have a unique id, we can use that:

// Example of what our new DELETE\_MESSAGE will look like

{

type: 'DELETE\_MESSAGE',

id: '9da98285-4178',

}

To remove the message from state.messages, we can use Array’s filter() method. filter() returns a new array containing all of the elements that “pass” the supplied test function:

} else if (

action.type === 'DELETE\_MESSAGE') {

return {

messages: state.messages.filter((m) => (

m.id !== action.id

))

Here, we’re building a new array containing every object that does not have an id that corresponds to the action’s id.

With these changes in place, our reducers are ready to handle our new message objects. We’ll update our React components next. We need to update both the actions they emit as well as how they render messages.

**Updating the React components**

MessageInput dispatches an ADD\_MESSAGE action whenever the user clicks its submit button. We’ll need to modify this component so that it uses the property name text as opposed to message for the action:

**redux/chat\_intermediate/src/complete/App-2.js**

handleSubmit = () => {

store.dispatch({

type: 'ADD\_MESSAGE',

text: this.state.value,

});

this.setState({ value: '', });

};

MessageView dispatches a DELETE\_MESSAGE action whenever the user clicks on a message. **We need to tweak the action it dispatches so that it uses the property id as opposed to index:**

class MessageView extends React.Component {

handleClick = (id) => {

store.dispatch({

type: 'DELETE\_MESSAGE',

id: id,

});

};

Then, we need to change the render() function for MessageView. We’ll modify the HTML for each message so that it also includes the timestamp property. To render the text of the message, we call message.text:

**redux/chat\_intermediate/src/complete/App-2.js**

render() { const messages = this.props.messages.map((message, index) => (

<div

className='comment'

key={index}

onClick={() => this.handleClick(message.id)} // Use `id`

>

<div

className='text'> {/\* Wrap message data in `div` \*/}

{message.text}

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

</div>

));

Note that we now pass in message.id as opposed to index to this.handleClick(). We wrap the display logic for each message inside a div with class text.

Our reducers and React components are now on the same page. We’re using our new representation of both the state and actions.

Save App.js. If your server isn’t already running, boot it:

$ npm start

Navigate to http://localhost:3000. When you add messages, a timestamp should appear to the right of each message. You can also delete them as before by clicking on them:

**Introducing threads**

Our state now uses message objects, which will allow us to carry information about each message in our app (like timestamp).

But in order for our app to begin reflecting a real-world chat app, we’ll need to introduce another concept: threads.

In a chat app, a “thread” is a distinct set of messages. A thread is a conversation between you and one or more other users:

Two threads in the interface

As the completed version of the app demonstrated, our app will use tabs to enable a user to switch between threads. Each message belongs to a single thread:

Messages belong to Threads

To support threads, we’ll update the shape of our state object. The top-level property will now be threads, an array of thread objects. Each thread object will have a messages property which will contain the message object we introduced to the system in the previous section:

{ threads: [

{ id: 'd7902357-4703', // UUID of the thread title: 'Buzz Aldrin', // Who the conversation is with messages: [

{ id: 'e8596e6b-97cc', text: 'Twelve minutes to ignition.', timestamp: 1462122634882,

},

// ... other messages with Buzz Aldrin ]

},

// ... other threads (with other users) ],

}

**Supporting threads in initialState**

To support threads, let’s first update our initial state.

At the moment, we’re initializing state to an object with a messages property:

**redux/chat\_intermediate/src/complete/App-2.js**

const initialState = { messages: [] };

But for now, we can take a smaller step by just initializing our state with a hard-coded set of threads.

Modify initialState now, initializing it to an object with a threads property. We’ll have two thread objects in state:

**redux/chat\_intermediate/src/complete/App-3.js**

const initialState = {

activeThreadId: '1-fca2', // New state property threads:

[ // Two threads in state

{ id: '1-fca2', // hardcoded pseudo-UUID title: 'Buzz Aldrin', messages: [

{ // This thread starts with a single message already

text: 'Twelve minutes to ignition.',

timestamp: Date.now(),

id: uuid.v4(),

},

],

},

{

id: '2-be91',

title: 'Michael Collins',

messages: [],

},

],

};

Notice our initial state object contains another top-level property, activeThreadId. Our front-end displays only one thread at a time. Our view needs to know which thread to display. In addition to threads and messages, our app should also have this additional piece of state.

Here, we initialize it to our first thread which has an id of '1-fca2'.

We now have an initial state object that our React components can use to render a threaded version of our app. We’ll update the components first to render from this new state shape.

Our app will be locked at this initial state though; we won’t be able to add or delete any messages or switch between tabs. Once we confirm the views look good, we’ll update our actions and our reducer to support our new thread-based chat app.

For now, we’re initializing the first thread object with a single message already under messages. This will enable us to verify our React components are properly rendering a thread with a message ahead of our reducers supporting our updated ADD\_MESSAGE action.

**Supporting threads in the React components**

To enable switching between threads in the app, the interface will have tabs above the messages view. We’ll need to both add new React components and modify existing ones to support this.

Looking at the completed version of this chapter’s chat app, we can identify the following components:

• App: The top-level component.

• ThreadTabs: The tabs widget for switching between threads.

• Thread: Displays all the messages in a thread. This component has been called MessageView, but we’ll update its name to reflect our new thread-based state paradigm.

– MessageInput: The input to add new messages to the open thread. We can have this nested under Thread.

Let’s first update our existing components to support our thread-based state. We’ll then add our new component, ThreadTabs.

**Modifying App**

App currently subscribes to the store. App uses getState() to read the messages property and then renders its two children.

We’ll have the component use our state’s activeThreadId property to deduce which thread is active. The component will then pass the active thread to Thread (formerly MessageView) to render its messages.

First, we read both activeThreadId and threads from the state. We then use Array’s find to find the thread object that has an id that matches activeThreadId:

**redux/chat\_intermediate/src/complete/App-3.js**

class App extends React.Component {

componentDidMount() {

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

}

render() {

const state = store.getState();

const activeThreadId = state.activeThreadId;

const threads = state.threads;

const activeThread = threads.find((t) => t.id === activeThreadId);

We pass our Thread component the activeThread for it to render. We’re removing MessageInput from App as it will now be a child of Thread:

**redux/chat\_intermediate/src/complete/App-3.js**

return (

<div className='ui segment'>

<Thread thread={activeThread} />

</div>

);

**The updated App component, in full:**

**redux/chat\_intermediate/src/complete/App-3.js**

class App extends React.Component {

componentDidMount() {

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

}

render() {

const state = store.getState();

const activeThreadId = state.activeThreadId;

const threads = state.threads;

const activeThread = threads.find((t) => t.id === activeThreadId);

return (

<div className='ui segment'>

<Thread thread={activeThread} />

</div>

);

}

}

**Turning MessageView into Thread**

Now that messages are collected under a single thread, our front-end is displaying one thread’s messages at a given time. We’ll rename MessageView to Thread to reflect this.

Thread will render both the list of messages pertaining to the thread it’s rendering as well as the MessageInput component for adding new messages to that thread. First, rename the component:

**redux/chat\_intermediate/src/complete/App-3.js**

class Thread extends React.Component {

Now, to create messages in render(), we’ll use this.props.thread.messages instead of this.props.messages:

**redux/chat\_intermediate/src/complete/App-3.js**

render() {

const messages = this.props.thread.messages.map((message, index) => (

Last, we’ll add MessageInput as a child of Thread. While we’ll eventually need to update MessageInput to work properly with our new threaded state paradigm, we’ll hold off on that update for now:

**redux/chat\_intermediate/src/complete/App-3.js**

return (

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

<div className='ui comments'> {messages}

</div>

<MessageInput />

</div>

);

Try it out

Save App.js. Navigating to http://localhost:3000, we see our app with a single message, the message we set in initialState. However, we cannot add or delete any messages:

Our actions and our reducer do not yet support our new state shape. Before we update them, though, let’s add the ThreadTabs component to our app.

Adding the ThreadTabs component

Our App component will render ThreadTabs above its other child components. ThreadTabs will need a list of thread titles to render as tabs. We’ll eventually have the component dispatch an action to update the activeThreadId piece of state whenever a tab is clicked, but for now we’ll just have it render the thread titles.

Updating App

First, we’ll prepare a tabs array. This array will contain objects that correspond to the information ThreadTabs needs to render each tab.

ThreadTabs will need two pieces of information:

• The title of each tab

• Whether or not the tab is the “active” tab

Indicating whether or not the tab is active is for style purposes. Here’s an example of two tabs. In the markup, the left tab is indicated as active:

Inside render for App, we’ll create an array of objects, tabs. Each object will contain a title and an active property. active will be a boolean:

**redux/chat\_intermediate/src/complete/App-4.js**

const tabs = threads.map(t => (

{ // a "tab" object title: t.title, active: t.id === activeThreadId,

}

));

We add ThreadTabs to the App component’s markup, passing down tabs as a prop:

**redux/chat\_intermediate/src/complete/App-4.js**

return (

<div className='ui segment'>

<ThreadTabs tabs={tabs} />

<Thread thread={activeThread} />

</div>

);

**Creating ThreadTabs**

Next, let’s add the ThreadTabs component right beneath the declaration of App. While we’ll soon dispatch actions from ThreadTabs whenever a tab is clicked, for now it will just render the HTML for our tabs.

We first map over this.props.tabs, preparing the markup for each tab. In Semantic UI, we represent each tab as a div with a class of item. The active tab has a class of active item. We’ll use index for the React-required key prop of each tab:

**redux/chat\_intermediate/src/complete/App-4.js**

class ThreadTabs extends React.Component {

render() {

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

<div

key={index}

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

>

{tab.title}

</div> ));

return (

<div

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

</div>

);

}

}

Try it out

Save App.js. Ensure the server is still running, and browse to http://localhost:3000. Everything is as before — we can’t add or delete messages — but we now have tabs in the interface. We can’t switch between the tabs yet, though:

The first thread (“Buzz Aldrin”) is our active thread. Its corresponding tab object has its active property set to true. This sets the class of the tab to active item, which gives us a nice visual indication of the active tab on the interface.

We’ve updated our state to support threads and our React components are properly rendering based on this new representation. Next, we’ll restore interactivity by updating the actions and the reducer to work with this new state model.

Supporting threads in the reducer

Because our representation of state for this app has changed, we’ll need to update the action handlers in our reducer.

Updating ADD\_MESSAGE in the reducer

Because messages now belong to a thread, our ADD\_MESSAGE action handler will need to add new messages to a specific thread. We’ll add a property to this action object, threadId:

{ type: 'ADD\_MESSAGE', text: 'Looking good.', threadId: '1-fca2', // <- Or whichever thread is appropriate

}

Before, our ADD\_MESSAGE handler in reducer() created a new message object and then used concat to append it to state.messages. Now, we need to do the following:

1. Create the new message object newMessage

2. Find the corresponding thread (action.threadId) in state.threads

3. Append newMessage to the end of thread.messages

Let’s modify reducer() in App.js now.

We keep the instantiation of the newMessage object. Next, we define threadIndex by hunting through state.threads and identifying the thread that corresponds to action.threadId:

**redux/chat\_intermediate/src/complete/App-5.js**

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

const threadIndex = state.threads.findIndex( (t) => t.id === action.threadId

);

So, adding some detail to our plan from before, we actually want to do the following:

1. Create the new message object newMessage

2. Find the corresponding thread (state.activeThreadId) in state.threads

3. Create a new thread object that contains all of the properties of the original thread object, plus an updated messages property

4. Return state with a threads property that contains our new thread object in place of the original one

We already have step 1 taken care of. We have the threadIndex defined. Let’s see how we create our new thread object:

**redux/chat\_intermediate/src/complete/App-5.js**

const oldThread = state.threads[threadIndex];

const newThread = {

...oldThread, messages: oldThread.messages.concat(newMessage), };

To create newThread, we use an experimental JavaScript feature: the spread syntax for objects. We used the spread syntax in the last chapter for arrays, creating a new array based off of chunks of an existing one. The spread syntax for arrays was introduced in ES6. Here, we’re using it to create a new object based on properties of an existing object.

This line copies all of the properties from oldThread to newThread:

**redux/chat\_intermediate/src/complete/App-5.js**

...oldThread,

And then this line sets the messages property of newThread to the new messages array that contains newMessage:

**redux/chat\_intermediate/src/complete/App-5.js**

messages: oldThread.messages.concat(newMessage),

Note that by having the property messages appear after oldThread, we’re effectively “overwriting” the messages property from oldThread.

As you may recall, the first argument for Object.assign() is the target object. You can pass in as many additional arguments which are all of the objects you would like to copy properties from.

Now, we have a newThread object that contains all of the properties of the original thread except with an updated messages property that contains the message being added.

The last step is to return the updated state. We want to return an object which has a state.threads that’s set to the original list of threads except with our new thread object “subbed-in” for our old one.

We can re-use our strategy for creating a new array that contains chunks of a previous one. We have threadIndex, the index of the thread in the array we want to swap out.

We can create a new object and again use the spread operator to copy over all of the properties of state into the new object. Then, we can overwrite the threads property with our new array:

**redux/chat\_intermediate/src/complete/App-5.js**

return {

...state, threads: [

...state.threads.slice(0, threadIndex), newThread,

...state.threads.slice( threadIndex + 1, state.threads.length

),

],

};

Updating ADD\_MESSAGE required some new concepts, but now we have an important strategy we’ll be re-using in the future: how to update state objects while avoiding mutations.

Our new logic for handling the ADD\_MESSAGE action, in full:

**redux/chat\_intermediate/src/complete/App-5.js**

if (action.type === 'ADD\_MESSAGE') {

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

const threadIndex = state.threads.findIndex( (t) => t.id === action.threadId );

const oldThread = state.threads[threadIndex];

const newThread = {

...oldThread,

messages: oldThread.messages.concat(newMessage),

};

return {

...state, threads: [

...state.threads.slice(0, threadIndex), newThread,

...state.threads.slice( threadIndex + 1, state.threads.length

),

],

};

Introducing threads to our state significantly increased the complexity of this action handler. Soon, we’ll explore ways to break this action handler apart into smaller pieces. For now, let’s update the areas where we’re dispatching the ADD\_MESSAGE action. There’s just one: the MessageInput component.

Updating the MessageInput component

Our action object for ADD\_MESSAGE should now contain the property threadId.

MessageInput is the only component that dispatches this action. The component should set threadId to the id of the active thread. We’ll have Thread pass MessageInput the active thread’s id as a prop:

**redux/chat\_intermediate/src/complete/App-5.js**

return (

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

<div className='ui comments'> {messages}

</div>

<MessageInput threadId={this.props.thread.id} /> </div>

);

}

Then, we can read from this.props in MessageInput to set threadId on the action object:

**redux/chat\_intermediate/src/complete/App-5.js**

handleSubmit = () => { store.dispatch({

type: 'ADD\_MESSAGE',

text: this.state.value,

threadId: this.props.threadId,

});

this.setState({

value: '',

});

};

With MessageInput dispatching our updated ADD\_MESSAGE action, let’s verify that the functionality for adding messages has returned.

**Try it out**

Save App.js. Refreshing http://localhost:3000, we see that we can now submit messages again:

We still can’t delete messages or switch between our threads.

Let’s focus on updating the DELETE\_MESSAGE action first.

Updating DELETE\_MESSAGE in the reducer

The DELETE\_MESSAGE action currently carries the property id which indicates which message should be deleted. While our app at the moment only permits a message to be deleted from the active thread, we’ll write our reducer so that it searches all of the threads for the matching message.

When removing a message from state, we face a similar challenge as to the one that we faced with ADD\_MESSAGE. The message is sitting in the messages array of a thread. But we can’t modify the thread object in state. We can use a similar strategy:

1. Grab the thread that holds the message we want to remove

2. Create a new thread object that contains all of the properties of the original thread object, plus an updated messages property that does not include the message we’re removing

3. Return state with a threads property that contains our new thread object in place of the original one

Let’s modify the reducer’s DELETE\_MESSAGE handler now.

First, we identify the thread that holds the message we’re deleting:

**redux/chat\_intermediate/src/complete/App-6.js**

} else if (action.type === 'DELETE\_MESSAGE') {

const threadIndex = state.threads.findIndex((t) => t.messages.find((m) => (

m.id === action.id

)) );

const oldThread = state.threads[threadIndex];

Inside of the findIndex callback, we perform a find against the messages property of that thread. If find finds the message matching the action’s id, it returns the message. This satisfies the findIndex testing function, meaning that the thread’s index is returned.

Next, we create a new thread object. We use the spread syntax to copy over all of the attributes from oldThread to this new object. We overwrite the messages property by using the filter() function just as before to generate a new array that does not include the deleted message:

**redux/chat\_intermediate/src/complete/App-6.js**

const newThread = {

...oldThread, messages: oldThread.messages.filter((m) => (

m.id !== action.id

)),

};

The return statement for DELETE\_MESSAGE is identical to the one for ADD\_MESSAGE. We’re performing the same operation: We ultimately want to “swap out” the original thread object in state.threads with our new one. So we want to:

1. Create a new object and copy over all the attributes from state

2. Overwrite the threads property with a new array of threads that has our new thread object swapped in for the original one

Again, the code is identical to that of ADD\_MESSAGE:

**redux/chat\_intermediate/src/complete/App-6.js**

return {

...state, threads: [

...state.threads.slice(0, threadIndex), newThread,

...state.threads.slice( threadIndex + 1, state.threads.length

),

],

};

In full, our DELETE\_MESSAGE action handler:

**redux/chat\_intermediate/src/complete/App-6.js**

} else if (action.type === 'DELETE\_MESSAGE') {

const threadIndex = state.threads.findIndex((t) => t.messages.find((m) => (

m.id === action.id

)) );

const oldThread = state.threads[threadIndex];

const newThread = {

...oldThread, messages:

oldThread.messages.filter((m) => (

m.id !== action.id

)),

};

return {

...state, threads: [

...state.threads.slice(0, threadIndex),

newThread,

...state.threads.slice( threadIndex + 1, state.threads.length

),

],

};

The complexity of this action handler, like that of ADD\_MESSAGE, was significantly complicated by the introduction of threads. What’s more, we’ve witnessed similar patterns emerge between the two action handlers: They both instantiate a new thread object that’s a derivative of an existing thread object. And they both swap this thread object in for the thread object being “modified” in state.

Soon, we’ll explore a new strategy to share this code and break these procedures down. But for now, let’s test out deleting messages again.

**Try it out**

Save App.js. Make sure your server is running and navigate to http://localhost:3000. Adding and deleting messages are both working now:

Clicking on the tab to switch threads still does not work, however. We initialize our state with activeThreadId set to '1-fca2' but have no actions in the system to modify this part of the state.

Adding the action OPEN\_THREAD

We’ll introduce another action, OPEN\_THREAD, which React will dispatch whenever the user clicks on a thread tab to open it. This action will ultimately modify the activeThreadId property on state.

The component ThreadTabs will be the one to dispatch this action.

The action object

The action object just needs to specify the id of the thread the user wants to open:

{ type: 'OPEN\_THREAD', id: '2-be91', // <- or whichever `id` is appropriate

}

**Modifying the reducer**

Let’s add another clause to our reducer to handle this new action.

We’ll copy over all of the attributes from state into a new object. We’ll overwrite the activeThreadId property of this new object:

**redux/chat\_intermediate/src/complete/App-7.js**

} else if (action.type === 'OPEN\_THREAD') {

return {

...state, activeThreadId: action.id,

};

Using this strategy, we do not modify state.

Now, we just need to have ThreadTabs dispatch this action whenever a tab is clicked.

Dispatching from ThreadTabs

In order for ThreadTabs to dispatch the OPEN\_THREAD action, it needs the id of the thread that was clicked on.

Right now, the tab object that we’re giving to ThreadTabs contains the properties title and active.

We’ll need to modify the instantiation of tabs in App to also include the property id.

Let’s do that first. Inside the App component, add this property to the tab object that we create:

**redux/chat\_intermediate/src/complete/App-7.js**

const tabs = threads.map(t => (

{

title: t.title,

active: t.id === activeThreadId,

id: t.id,

}

));

Next, we’ll add the component function handleClick to ThreadTabs. The function accepts id:

**redux/chat\_intermediate/src/complete/App-7.js**

class ThreadTabs extends React.Component {

handleClick = (id) => {

store.dispatch({

type: 'OPEN\_THREAD',

id: id,

});

};

Finally, we add an onClick attribute to the div tag for each tab. We set it to a function that calls handleClick with the thread’s id:

**redux/chat\_intermediate/src/complete/App-7.js**

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

<div key={index}

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

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

ThreadTabs is now emitting our new action. Let’s test everything out.

**Try it out**

Save App.js. Pull up http://localhost:3000 in your browser. At this point, adding and deleting messages works and we can switch between tabs. If we add a message to one thread, it’s added to just that thread:

This is starting to look like an actual chat application! We’ve introduced the concept of threads and now support an interface that can switch between conversations with multiple users.

However, in the process, we significantly complicated our reducer function. While it’s nice that all of our state is being managed in a single location, each of our action handlers contain a lot of logic. What’s more, our ADD\_MESSAGE and DELETE\_MESSAGE handlers duplicate a lot of the same code.

It’s also odd that the management of two distinct pieces of state — adding/deleting messages and switching between threads — are both managed in the same place. The idea of having a single reducer function manage the entirety of our state as the complexity of our app grows might feel daunting and even a bit dubious.

Indeed, Redux apps have a strategy for breaking down state management logic: reducer composition.

**Breaking up the reducer function**

With reducer composition, we can break up the state management logic of our app into smaller functions. We’ll still pass in a single reducer function to createStore(). But this top-level function will then call one or more other functions. Each reducer function will manage a different part of the state tree.

Adding and deleting messages is a distinct piece of functionality from switching between threads. Let’s break these two apart first.

A new reducer()

We’ll still call our top-level function reducer(). Except now, we’ll have two other reducer functions, each managing their part of the state’s top-level properties. We’ll name each of these sub-reducers after the property name that they manage:

To implement this, let’s first see what our new top-level reducer() function should look like. We can then create our sub-reducer functions.

To avoid confusion, let’s rename our current reducer function from reducer() to threadsReducer():

**redux/chat\_intermediate/src/complete/App-8.js**

function threadsReducer(state, action) {

We still have to tweak this function, but we’ll get back to it in a bit.

Now, above threadsReducer(), we’ll insert our new function, reducer():

**redux/chat\_intermediate/src/complete/App-8.js**

function reducer(state, action) {

return {

activeThreadId: activeThreadIdReducer(state.activeThreadId, action),

threads: threadsReducer(state.threads, action),

};

}

The function is short but there’s a lot going on.

We are returning a brand new object with the keys activeThreadId and threads. Importantly, we are delegating the responsibility of how to update activeThreadId and threads to their respective reducers.

Check out this line again:

**redux/chat\_intermediate/src/complete/App-8.js**

activeThreadId: activeThreadIdReducer(state.activeThreadId, action),

For the property activeThreadId on next state, we’re delegating responsibility to the yet-to-bedefined function activeThreadIdReducer(). Notice how the first argument is not the whole state — it’s only the part of the state that this reducer is responsible for. We pass in the action as the second argument.

Same strategy with threads:

**redux/chat\_intermediate/src/complete/App-8.js**

threads: threadsReducer(state.threads, action),

We delegate responsibility of the state’s threads property to threadsReducer(). The state that we pass this reducer — the first argument — is the part of the state that threadsReducer() is responsible for updating.

This is how sub-reducers work in tandem with the top-level reducer. The top-level reducer breaks up each part of the state tree and delegates the management of those pieces of state to the appropriate reducer.

To get a better idea, let’s write activeThreadIdReducer() below reducer():

**redux/chat\_intermediate/src/complete/App-8.js**

function activeThreadIdReducer(state, action) {

if (action.type === 'OPEN\_THREAD') {

return action.id;

} else {

return state;

}

}

This reducer only handles one type of action, OPEN\_THREAD. Remember, the state argument being passed to the reducer is actually state.activeThreadId. This is the only part of the state that this reducer needs to be concerned with. As such, activeThreadIdReducer() receives a string as state (the id of a thread) and will return a string.

Notice how this simplifies the logic for handling OPEN\_THREAD. Before, our logic had to consider the entire state tree. So we had to create a new object, copy over all of the state’s properties, and then overwrite the activeThreadId property. Now, our reducer just has to worry about this single property.

If the action is OPEN\_THREAD, activeThreadIdReducer() simply returns action.id, the id of the thread to be opened. Otherwise, it returns the same id it was passed.

Up in reducer(), the return value of activeThreadIdReducer() is set to the property activeThreadId in the new state object.

Let’s update threadsReducer() next. Before, this function received the entire state object. Now, it’s only receiving part of it: state.threads. We’ll be able to simplify the code a bit as a result.

Updating threadsReducer()

threadsReducer() is now receiving only part of the state. Its state argument is the array of threads.

As a result, we can simplify our returns like we did with activeThreadIdReducer(). We no longer need to create a new state object, copy all the old values over, then overwrite threads. Instead, we can just return the updated array of threads.

What’s more, instead of referencing state.threads everywhere we’ll reference state as this is now the array of threads.

Let’s make these modifications first for ADD\_MESSAGE.

We reference state as opposed to state.threads:

**redux/chat\_intermediate/src/complete/App-8.js**

const threadIndex = state.findIndex( (t) =>

t.id === action.threadId

);

const oldThread = state[threadIndex];

const newThread = {

...oldThread,

messages: oldThread.messages.concat(newMessage),

};

For the return statement, we no longer need to return a full state object, just the array of threads:

**redux/chat\_intermediate/src/complete/App-8.js**

return [

...state.slice(0, threadIndex),

newThread,

...state.slice( threadIndex + 1, state.length

),

];

DELETE\_MESSAGE will receive the same treatment.

We first change our references from state.threads to state:

**redux/chat\_intermediate/src/complete/App-8.js**

} else if (action.type === 'DELETE\_MESSAGE') {

const threadIndex = state.findIndex((t) => t.messages.find((m) => (

m.id === action.id

)) );

const oldThread = state[threadIndex];

Then, we return just the array as opposed to the whole state object:

**redux/chat\_intermediate/src/complete/App-8.js**

return [

...state.slice(0, threadIndex),

newThread,

...state.slice( threadIndex + 1, state.length

),

];

Finally, we can remove the OPEN\_THREAD handler from threadsReducer(). This reducer is not concerned about the OPEN\_THREAD action. That action pertains to a part of the state tree that this reducer does not work with. So, whenever that action is received, the function will reach the last else clause and will just return state unmodified.

Our updated threadsReducer() function, in full:

**redux/chat\_intermediate/src/complete/App-8.js**

function threadsReducer(state, action) {

if (action.type === 'ADD\_MESSAGE') {

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

const threadIndex = state.findIndex( (t) =>

t.id === action.threadId

);

const oldThread = state[threadIndex];

const newThread = { ...oldThread, messages: oldThread.messages.concat(newMessage),

};

return [

...state.slice(0, threadIndex), newThread, ...state.slice( threadIndex + 1, state.length

),

];

} else if (action.type === 'DELETE\_MESSAGE') {

const threadIndex = state.findIndex(

(t) => t.messages.find((m) => (

m.id === action.id

)) );

const oldThread = state[threadIndex];

const newThread = { ...oldThread, messages: oldThread.messages.filter((m) => (

m.id !== action.id

)), };

return [

...state.slice(0, threadIndex), newThread,

...state.slice( threadIndex + 1, state.length

),

];

} else {

return state;

}

}

By focusing this reducer and having it deal with only the threads property in state, we were able to simplify our code a little. Our return functions no longer have to worry about the entire state tree and we removed an action handler from this function.

We still duplicate logic between the two action handlers. Notably, their return functions are the same.

In fact, while we simplified the reducer by having it work with only state.threads, this reducer is actually dealing with two levels of the state tree: both threads and messages. We can further break our reducer logic into smaller pieces and share code by having threadsReducer() delegate the responsibility to another reducer: messagesReducer().

We’ll explore this in the next section. For now, let’s boot up the app and verify our solution so far works.

You might be asking: Why not just rename the first argument in threadsReducer() to threads as opposed to calling it state?

Indeed, doing this could improve readability of the reducer function. You’d no longer need to hold the idea in working memory that state is actually state.threads.

However, consistently calling the first argument to a Redux reducer state can be helpful for avoiding errors. It’s a clear reminder that the argument is a part of the state tree and that you should avoid accidentally mutating it. Ultimately, it’s a matter of personal preference.

**Try it out**

Save App.js. Boot the server if it isn’t running:

$ npm start

And then point your browser to http://localhost:3000. On the surface, it should appear as though nothing has changed; adding and deleting messages works and we can switch between threads.

**Adding messagesReducer()**

We used the concept of reducer composition to break up the management of our state. Our toplevel reducer() function delegates the management of the two top-level state properties to their respective reducers.

We can take this concept even further. We’ve noticed that threadsReducer() has the responsibility of managing the state of both threads and messages. What’s more, we see an opportunity to share code between the two action handlers.

We can have threadsReducer() delegate the management of each thread’s messages property to another reducer, messagesReducer(). In full, the reducer tree will look like this:

Let’s first modify threadsReducer(), anticipating this new messages reducer function. We want the threads reducer function to know as little as possible about messages. threadsReducer() will rely on messagesReducer() to determine how a given thread’s messages should be updated based on the action received.

Modifying the ADD\_MESSAGE action handler

Let’s see what happens when we delegate all message handling functionality to our anticipated messagesReducer() function.

We’ll no longer declare newMessage. We expect that messagesReducer() will take care of this.

Then, when specifying the messages attribute for newThread, we’ll delegate the creation of this array to messagesReducer():

**redux/chat\_intermediate/src/complete/App-9.js**

function threadsReducer(state, action) {

if (action.type === 'ADD\_MESSAGE') {

const threadIndex = state.findIndex( (t) => t.id === action.threadId);

const oldThread = state[threadIndex]; const newThread = {

...oldThread,

messages: messagesReducer(oldThread.messages, action),

};

We pass messagesReducer() the array oldThread.messages as the first argument and action as the second. This follows our pattern so far: reducer() passed threadsReducer()state.threads as its first argument. threadsReducer() passes thread.messages as the first argument to messagesReducer(). We now know that when we build messagesReducer(), the first argument (state) will be an array of a given thread’s messages.

The full action object is passed along in its entirety down the chain.

We do not need to make any modifications to the return value.

At the moment, this might only seem like a minor — perhaps negligible — victory. Before, we were calling concat in-line to create a new messages array with the new message inside. Now we have the added complexity of calling another function to do this.

We are breaking apart the responsibility of our functions, which is a worthwhile endeavor in its own right. Furthermore, another benefit will become apparent when we give DELETE\_MESSAGE the same treatment.

Prior to modifying our DELETE\_MESSAGE action handler, let’s start on our messagesReducer() function to get an idea of how these reducers will work together.

**Creating messagesReducer()**

We’ll write messagesReducer() below threadsReducer() in App.js. We’ll start it off with one action handler (ADD\_MESSAGE).

As we saw above, threadsReducer() passes messagesReducer() a thread’s messages array as the first argument. This will be state inside of messagesReducer().

messagesReducer() needs to:

1. Create the new message

2. Return a new array of messages that includes this new message appended to the end of it

We’ll use the same logic that we had previously in threadsReducer() to accomplish this:

**redux/chat\_intermediate/src/complete/App-9.js**

function messagesReducer(state, action) {

if (action.type === 'ADD\_MESSAGE') {

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

return state.concat(newMessage);

} else {

return state;

} }

messagesReducer() receives an array of messages as its first argument, state. The ADD\_MESSAGE handler creates a new message using action.text. We use concat just as before to return a new messages array with our new message appended to it.

Now that we’ve seen how threadsReducer() delegates ADD\_MESSAGE to messagesReducer(), let’s take a look at how DELETE\_MESSAGE can do the same thing.

Modifying the DELETE\_MESSAGE action handler

As with the ADD\_MESSAGE action handler, we want our DELETE\_MESSAGE action handler in threadsReducer() to delegate the responsibility of handling the messages property of each thread to messagesReducer().

Instead of invoking the filter() function directly, we can call messagesReducer() inside of the declaration of newThread and have it generate the messages property, like this:

**redux/chat\_intermediate/src/complete/App-9.js**

} else if (action.type === 'DELETE\_MESSAGE') {

const threadIndex = state.findIndex((t) => t.messages.find((m) => (

m.id === action.id

)) );

const oldThread = state[threadIndex];

const newThread = { ...oldThread, messages: messagesReducer(oldThread.messages, action),

};

Look familiar? With these modifications, the threads reducer’s DELETE\_MESSAGE looks almost exactly like its ADD\_MESSAGE. The only difference is how the two handlers identify threadIndex.

On reflection, this makes sense. The action that we’re dealing with is adding and removing messages. This will only ever affect the messages property on an individual thread. Because the messages property is now being managed by messagesReducer(), the differences between ADD\_MESSAGE and DELETE\_MESSAGE are expressed in that function. Aside from determining threadIndex, the rest of the ceremony — creating a new thread object with an updated messages property — is identical.

We can define a new function, findThreadIndex(), where we can store the logic to determine threadIndex. Then, we can combine our two action handlers as their code will be identical.

First, we’ll write findThreadIndex() above threadsReducer(). This function will take both threads (or state in threadsReducer()) and the action as arguments. We’ll copy the logic for finding the index of the affected thread into this function:

**redux/chat\_intermediate/src/complete/App-10.js**

function findThreadIndex(threads, action) {

switch (action.type) {

case 'ADD\_MESSAGE': {

return threads.findIndex((t) => t.id === action.threadId);

}

case 'DELETE\_MESSAGE': {

return threads.findIndex((t) => t.messages.find((m) => (

m.id === action.id

))

);

}

}

}

We decided to use a switch statement here as opposed to an if/else clause. Using a switch statement in reducers and their helper functions can be slightly easier to read and manage. You’ll see it used in many Redux apps. As the actions in a system grows, a single reducer will have to have multiple action handlers. switch is built for this use case.

Another reason for using switch in findThreadIndex() is because we’ll use one in our refactored threadsReducer():

**redux/chat\_intermediate/src/complete/App-10.js**

function threadsReducer(state, action) { switch (action.type) {

case 'ADD\_MESSAGE':

case 'DELETE\_MESSAGE': {

const threadIndex = findThreadIndex(state, action);

const oldThread = state[threadIndex]; const newThread = {

...oldThread, messages: messagesReducer(oldThread.messages, action),

};

return [

...state.slice(0, threadIndex), newThread, ...state.slice( threadIndex + 1, state.length

),

]; } default: {

return state;

}

} }

switch is nice here because this:

**redux/chat\_intermediate/src/complete/App-10.js**

switch (action.type) {

case 'ADD\_MESSAGE': case 'DELETE\_MESSAGE': {

Reads a bit clearer than the alternative:

if (action.type === 'ADD\_MESSAGE' || action.type === 'DELETE\_MESSAGE') { // ...

}

This readability concern will be exacerbated if we continue to introduce more actions to the system that just affect the messages property of a given thread (like UPDATE\_MESSAGE). With a switch statement, we can cleanly specify that all of these actions share the same code block.

Other than the call to findThreadIndex(), the body of the combined action handler matches what we had individually for each above.

Adding DELETE\_MESSAGE to messagesReducer()

The threadsReducer() function now treats ADD\_MESSAGE and DELETE\_MESSAGE actions in the exact same way. When the reducer receives a DELETE\_MESSAGE action, it will call messagesReducer() with a list of messages and the action. The list of messages corresponds to an individual thread’s messages property. The action carries the directive for which message to remove.

Let’s add the DELETE\_MESSAGE logic to messagesReducer(). You can keep the if/else clause for now or change to switch:

**redux/chat\_intermediate/src/complete/App-10.js**

function messagesReducer(state, action) {

switch (action.type) {

case 'ADD\_MESSAGE': {

const newMessage = {

text: action.text,

timestamp: Date.now(),

id: uuid.v4(),

};

return state.concat(newMessage);

}

case 'DELETE\_MESSAGE': {

return state.filter(m => m.id !== action.id);

}

default: {

return state;

}

}

}

Remember, state here is the array of messages. The logic for “filtering out” the message is the same that we had in threadsReducer().

At this point, we’ve broken out the logic to manage state across three reducer functions. Each function manages a different part of the state tree. Our tree of reducer functions is combined up at reducer(), which is the function that we pass to createStore().

The complexity of both our app and our state increased significantly in this chapter. We now see how using reducer composition allows us to manage this complexity. We have a pattern for scaling our system to handle the addition of many more actions and pieces of state.

There’s one more improvement we can make. While the logic for managing state updates is contained within our reducers, the initial state of the app is defined elsewhere, in initialState. Let’s bring the parts of this initialization closer to their respective reducers. This will scale better as our state tree grows. In addition, it means all of the logic around a particular part of the state tree is contained inside of its reducer.

Defining the initial state in the reducers

Right now, we declare our initialState object and then pass it into createStore(): const store = createStore(reducer, initialState);

This argument is not required. Let’s change this line so that we no longer pass in initialState: const store = createStore(reducer);

What will happen?

The createStore() function in the redux library contains one key difference from the one we wrote in the last chapter. After the store is initialized but right before it’s returned, createStore() actually dispatches an initialization action. That dispatch call looks like this:

// ...

// Inside of `createStore()` in `redux` dispatch({ type: '@@redux/INIT' });

return { // returns store object

dispatch, subscribe, getState,

}

}

While the initialization action has a type, chances are you’ll never need to use it.

The important part is that because we didn’t specify an initialState, state will be undefined when createStore() dispatches the initialization action. The only time state will be undefined is for this first dispatch.

Therefore, we can have our reducers specify the initial state for their part of the state tree whenever state is undefined.

Initial state in reducer()

When createStore() dispatches the initialization object, reducer() will receive a state that is undefined.

In this situation, we want to set state to a blank object ({}). This will enable reducer() to delegate the initialization of each property in the state object to its reducer. We’ll see how this works in practice shortly.

We can use ES6’s default arguments to achieve this:

**redux/chat\_intermediate/src/complete/App-11.js**

function reducer(state = {}, action) {

When reducer() receives a state of undefined, state is set to {}.

Importantly, when reducer() then calls each of its sub-reducers, each one of them will receive a state argument that is undefined.

A state that is undefined is the canonical way to initialize a reducer in Redux. While we could use this special initialization action object’s type, with ES6’s default arguments just relying on an undefined state is much easier.

Adding initial state to activeThreadIdReducer()

We can use default arguments to initialize the state in activeThreadIdReducer(). We want the initial state to be '1-fca2', the id we’ll specify for our first thread:

**redux/chat\_intermediate/src/complete/App-11.js**

function activeThreadIdReducer(state = '1-fca2', action) {

Let’s recap the initialization flow for activeThreadIdReducer().

At the end of createStore(), right before the function returns the new store object, it dispatches the initialization action. Then:

1. reducer() is called with a state of undefined and the initialization action object

2. state defaults to {}

3. reducer() calls activeThreadIdReducer() with state.activeThreadId, which is undefined

4. activeThreadIdReducer() sets state to '1-fca2' with its default argument

5. The else clause in activeThreadIdReducer() returns state ('1-fca2')

**Adding initial state to threadsReducer()**

We’ll use the same strategy for threadsReducer(). Its initial state is more complex, so we’ll split it up over multiple lines:

**redux/chat\_intermediate/src/complete/App-11.js**

function threadsReducer(state = [

{ id: '1-fca2', title: 'Buzz Aldrin', messages: messagesReducer(undefined, {}),

},

{ id: '2-be91', title: 'Michael Collins', messages: messagesReducer(undefined, {}), },

], action) {

If we weren’t initializing our app with two threads already in state, the default argument for threadsReducer() would just be []:

function threadsReducer(state = [], action) { // ...

}

Now, let’s set the default argument in messagesReducer(). While we were initializing one of the threads with a message before, we don’t need that anymore:

**redux/chat\_intermediate/src/complete/App-11.js**

function messagesReducer(state = [], action) {

Last, go ahead and remove initialState from App.js. We don’t need it anymore. Let’s kick the tires and make sure things are still working.

**Try it out**

Save App.js. Make sure the server is running, and then load http://localhost:3000 in your browser.

As expected, everything works as before: we can add and delete messages and switch between tabs. The only difference is that we no longer have a default message initialized with the state.

While on the surface the behavior has not changed, we know that under the hood our code is a lot cleaner.