# **Hands-on Lab 4: Intro to Flux and Redux**

**Estimated time: 60 minutes**

## Exercise 1:

**Building a counter**

We’ll explore Redux’s core ideas by building a simple counter. For now, we’ll focus only on Redux and state management. We’ll see later how Redux connects to React views.

**Preparation**

1. Inside of the code download that came with this lab, navigate to redux/counter:

$ cd redux/counter

All the code for the counter will go inside app.js.

Because we’re focusing on Redux and state management to start, we’ll be running our code in the terminal as opposed to the browser.

The package.json for both of the projects contains the package babel-cli. As we’ll indicate in the Try it out sections below, we’ll be using the babel-node command that comes with babel-cli to run our code examples:

# example of using `babel-node` to run code in the terminal $ ./node\_modules/.bin/babel-node app.js

1. Run npm install now inside of redux/counter to install babel-cli:

$ npm install

**Overview**

Our state will be a number. The number will start off as 0. Our actions will either be to increment or decrement the state. We know from our Redux diagram that the views would dispatch these actions to the store:

View emits “Increment” action

When the store receives an action from the views, the store uses a reducer function to process the action. The store provides the reducer function with the current state and the action. The reducer function returns the new state:

// Inside the store, receives `action` from the view state = reducer(state, action);

For example, consider a store with a current state of 5. The store receives an increment action. The store uses its reducer to derive the next state:

**Inside an example store**

We’ll start building our Redux counter by constructing its reducer. We’ll then work our way up to see what a Redux store looks like. Our store will be the maintainer of state, accepting actions and using the reducer to determine the next version of the state.

While we’re starting with a simple representation of state (a number), we’ll be working with much more complicated state in the next module.

**The counter’s actions**

We know the reducer function for our counter will accept two arguments, state and action. We know state for our counter will be an integer. But how are actions represented in Redux?

Actions in Redux are objects. Actions always have a type property. Our increment actions will look like this:

{ type: 'INCREMENT',

}

And decrement actions like this:

{ type: 'DECREMENT',

}

We can envision what a simple interface for this counter app might look like:

An example counter interface

When the user clicks the “+” icon, the view would dispatch the increment action to the store. When the user clicks the “-“ icon, the view would dispatch the decrement action to the store.

The image of the interface for the counter app is provided as just an example of what the view might look like. We will not be implementing a view layer for this app.

**Exercise 2:Incrementing the counter**

Let’s begin writing our reducer function. We’ll start by handling the increment action.

The reducer function for our counter accepts two arguments, state and action, and returns the next version of the state. When the reducer receives an INCREMENT action, it should return state +1.

1. Inside of app.js, add the code for our counter’s reducer:

**redux/counter/complete/initial-reducer.js**

function reducer(state, action) {

if (action.type === 'INCREMENT') {

return state + 1;

} else {

return state;

}

}

If the action.type is INCREMENT, we return the incremented state. Otherwise, our reducer returns the state unmodified.

You might be wondering if it would be a better idea to raise an error if our reducer receives an action.type that it does not recognize.

**Try it out**

1. At the bottom of app.js, let’s add some code to test our reducer.

We’ll call our reducer, passing in integers for state and seeing how the reducer increments the number. If we pass in an unknown action type, our reducer returns the state unchanged:

**redux/counter/complete/initial-reducer.js**

const incrementAction = { type: 'INCREMENT' };

console.log(reducer(0, incrementAction)); // -> 1

console.log(reducer(1, incrementAction)); // -> 2

console.log(reducer(5, incrementAction)); // -> 6

const unknownAction = { type: 'UNKNOWN' };

console.log(reducer(5, unknownAction)); // -> 5

console.log(reducer(8, unknownAction)); // -> 8

1. Save app.js and run it with ./node\_modules/.bin/babel-node:

$ ./node\_modules/.bin/babel-node app.js

1. And your output should look like this:

1

2

6

5

8

**Exercise 3:Decrementing the counter**

Again, decrement actions have a type of DECREMENT:

{ type: 'DECREMENT',

}

1. To support decrement actions, we add another clause to our reducer:

**redux/counter/complete/initial-reducer-w-dec.js**

function reducer(state, action) {

if (action.type === 'INCREMENT') {

return state + 1;

} **else if (action.type === 'DECREMENT') {**

**return state - 1;**

} else {

return state;

}

}

**Try it out**

1. At the bottom of app.js, below the code where we dispatched increment actions, add some code to dispatch decrement actions:

**redux/counter/complete/initial-reducer-w-dec.js**

const decrementAction = { type: 'DECREMENT' };

console.log(reducer(10, decrementAction)); // -> 9

console.log(reducer(9, decrementAction)); // -> 8

console.log(reducer(5, decrementAction)); // -> 4

1. Run app.js with ./node\_modules/.bin/babel-node:

$ ./node\_modules/.bin/babel-node app.js

1. And your output should look like this:

1

2

6

5

8

9

8

4

**Exercise 4:Supporting additional parameters on actions**

In the last example, our actions contained only a type which told our reducer either to increment or decrement the state. But often behavior in our app can’t be described by a single value. In these cases, we need additional parameters to describe the change.

For example, what if we wanted our app to allow the user to specify an amount to increment or decrement by?

An example counter interface with an amount field

We’ll have our actions carry the additional property amount. An INCREMENT action would then look like this:

{ type: 'INCREMENT', amount: 7,

}

1. We modify our reducer to increment and decrement by action.amount, expecting all actions to now carry this property:

**redux/counter/complete/reducer-w-amount.js**

function reducer(state, action) {

if (action.type === 'INCREMENT') {

**return state + action.amount;**

} else if (action.type === 'DECREMENT') {

**return state - action.amount;**

} else {

return state;

}

}

**Try it out**

1. Clear out the code we used to test out reducer() in app.js previously.

This time, we’ll test calling the reducer with our modified actions that now carry the amount property:

**redux/counter/complete/reducer-w-amount.js**

const incrementAction = {

type: 'INCREMENT',

amount: 5,

};

console.log(reducer(0, incrementAction)); // -> 5

console.log(reducer(1, incrementAction)); // -> 6

const decrementAction = {

type: 'DECREMENT',

amount: 11,

};

console.log(reducer(100, decrementAction)); // -> 89

1. Run app.js with ./node\_modules/.bin/babel-node:

$ ./node\_modules/.bin/babel-node app.js

1. And note the output:

5

6

89

**Building the store**

So far, we’ve been calling our reducer and manually supplying the last version of the state along with an action.

In Redux, the store is responsible for both maintaining the state and accepting actions from the view.

Only the store has access to the reducer:

**Inside the store**

The Redux library provides a function for creating stores, createStore(). This function returns a store object that keeps an internal variable, state. In addition, it provides a few methods for interacting with the store.

We will write our own version of createStore() so that we fully understand how Redux stores work. By the end of this module, our code for createStore() will behave almost exactly like the one provided by the Redux library.

At the moment, our store will provide two methods:

• dispatch: The method we’ll use to send the store actions

• getState: The method we’ll use to read the current value of state

1. Inside of app.js, clear out the code we used to test out reducer() previously. Below the definition of reducer(), let’s define createStore(). createStore() will accept a single argument, the desired reducer for the store.

Let’s take a look at the full createStore() function. We’ll break it down piece-by-piece below the code block:

**redux/counter/complete/reducer-w-store-v1.js**

function createStore(reducer) {

let state = 0;

const getState = () => (state);

const dispatch = (action) => {

state = reducer(state, action);

};

return {

getState,

dispatch,

};

}

The reducer argument createStore() accepts a single argument, reducer. This is how we will indicate what reducer function our store should use. state

We initialize the state to 0 at the top of createStore(). Note that we close over the state variable.

This makes state private and inaccessible outside of createStore().

**getState**

To get read access to the state from outside createStore(), we have the method getState which returns state. dispatch

The dispatch method is how we send actions to the store. We’ll call it like this:

store.dispatch({ type: 'INCREMENT', amount: 7 });

dispatch calls the reducer function passed in as an argument with the current state and the action. dispatch sets state to the reducer’s return value.

Note that dispatch does not return the state. Dispatching actions in Redux are “fire-and-forget.” When we call dispatch, we’re sending a notification to the store with no expectation on when or how that action will be processed.

Dispatching actions to the store is decoupled from reading the latest version of the state. We’ll see how this works in practice when we connect the store to React views at the end of this module.

**The return object**

At the bottom of createStore(), we return a new object. This object has getState and dispatch as methods.

**Try it out**

1. We’ll write the code to test out our store in app.js below createStore().

We’ll create our store object with createStore(). Then, instead of calling reducer() with a state and an action, we’ll dispatch actions to the store. Because our store is keeping the internal variable state, our state persists between dispatches.

We can use getState() to read state between dispatches:

**redux/counter/complete/reducer-w-store-v1.js**

const store = createStore(reducer);

const incrementAction = {

type: 'INCREMENT',

amount: 3,

};

store.dispatch(incrementAction);

console.log(store.getState()); // -> 3

store.dispatch(incrementAction);

console.log(store.getState()); // -> 6

const decrementAction = {

type: 'DECREMENT',

amount: 4,

};

store.dispatch(decrementAction);

console.log(store.getState()); // -> 2

1. Run app.js with ./node\_modules/.bin/babel-node:

$ ./node\_modules/.bin/babel-node app.js

1. And note the output:

3

6

2

The core of Redux

As it stands, our createStore() function closely resembles the createStore() function that ships with the Redux library. By the end of this module, we’ll have made just a couple of tweaks and additions to createStore() to bring it closer to that of the Redux library.

Now that we’ve seen a Redux store in action, let’s recap Redux’s key ideas:

All of your application’s data is in a single data structure called the state which is held in the store.

We saw that the store has a single private variable for the state, state.

Your app reads the state from this store.

We use getState() to access the store’s state.

The state is never mutated directly outside the store.

Because state is a private variable, it cannot be mutated outside of the store.

The views emit actions that describe what happened.

We use dispatch() to send these actions to the store.

A new state is created by combining the old state and the action by a function called the reducer.

Inside of dispatch(), our store uses reducer() to get the new state, passing in the current state and the action.

**Exercise 5: The beginnings of a chat app**

Previewing

We’ll build our chat app inside of the folder redux/chat\_simple.

From inside of the redux/counter directory, you can type:

$ cd ../chat\_simple

First, run npm install:

$ npm install

Run ls to see the contents of this folder:

$ ls

README.md

nightwatch.json

node\_modules

package.json

public

semantic

semantic.json

src

tests

yarn.lock

The structure of this app was generated with create-react-app.

Inside of src/ is App.js, the file we’ll be working with in this module:

$ ls src/

App.js

complete

index.css

index.js

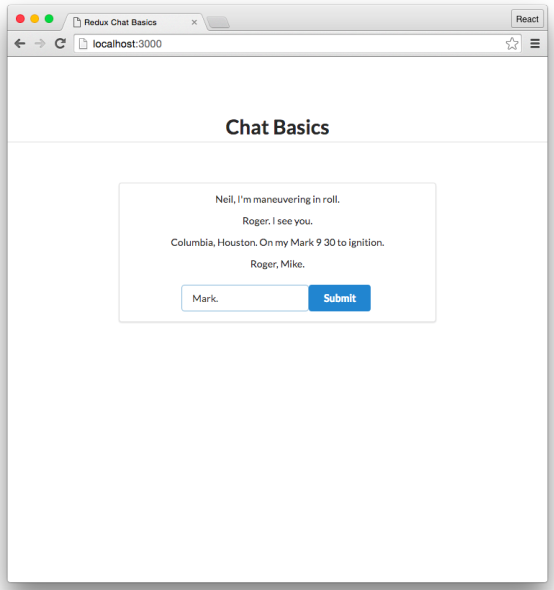
As is the case with apps generated by create-react-app, index.js is where we mount <App /> to the DOM. Inside of complete/ are the iterations of App as we build it up through the module.

At the moment, index.js is mounting complete/App-5.js, the completed version of the app that we reach at the end of this module.

We can boot our app to see it:

$ npm start

Navigate to localhost:3000:



The iteration of the chat app that we build in this module

Over the next few sections, we’ll be enhancing the feature-set (and complexity) of this app.

For now, we can add messages using the input box and delete messages by clicking on them.

While it’s on our mind, let’s modify index.js so that it includes ./App instead of ./complete/App-5:

import React from "react";

import ReactDOM from "react-dom";

**import App from "./App";**

Open up src/App.js. You’ll see that the same createStore() that we built for the counter app is already present.

As with the counter app, we’ll start by building the chat app’s reducer. Once our reducer is built, we’ll see how we connect our Redux store to React views.

Before we build our reducer, though, we should examine how the chat app will represent both its state and its actions.

**State**

The state in our counter app was a single number. In our chat app, the state is going to be an object.

This state object will have a single property, messages. messages will be an array of strings, with each string representing an individual message in the application. For example:

// an example `state` value

{ messages: [

'here is message one',

'here is message two',

],

}

**Actions**

Our app will process two actions: ADD\_MESSAGE and DELETE\_MESSAGE.

The ADD\_MESSAGE action object will always have the property message, the message to be added to the state.

The ADD\_MESSAGE action object has this shape:

{ type: 'ADD\_MESSAGE',

message: 'Whatever message is being added here',

}

The DELETE\_MESSAGE action object will delete a specified message from the state.

If each of our messages were objects, we could assign each message an id property when it is created. DELETE\_MESSAGE could then specify the message to delete with an id property.

However, for simplicity, our messages at the moment are strings. To specify the message to be removed from state, we can use the index of the message in the array.

With that in mind,

the DELETE\_MESSAGE action object has this shape:

{ type: 'DELETE\_MESSAGE',

index: 2, // <- index of whichever message is being removed here

}

**Building the reducer()**

Initializing state

Right now, we initialize state at the top of createStore() to 0:

function createStore(reducer) {

let state = 0;

// ...

}

While this works fine for our counter app, we want the initial state for our messaging app to look like this:

{ // an object

messages: [], // no messages

}

We’ll need to modify createStore() so that it will work for this and any representation of state.

We’ll have createStore() accept a second argument, initialState. The function will initialize state to this value.

Inside of App.js, edit createStore() now:

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

function createStore(reducer, initialState) {

let state = initialState; // ...

We’ll pass in initialState when we initialize the store a bit later.

Handling the ADD\_MESSAGE action

Begin writing the reducer() inside of App.js, below createStore():

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

function reducer(state, action) {

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

return

{

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

};

} else {

return state;

}

}

When our reducer receives the ADD\_MESSAGE action we want to append the new message to the end of the messages array in state. Otherwise, we return state unmodified.

Because we do not want to modify the state argument, ADD\_MESSAGE should instead create a new state object with a new messages array. The new array should have the desired message appended to it.

Look again at how we produce the next state in ADD\_MESSAGE:

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

return {

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

Crucially, Array’s concat does not modify the original array. Instead, it creates a new copy of the array that includes action.message appended to it.

In general, writing functions purely can help reduce surprises or enigmatic bugs in your code. We explore the specific motivations and benefits of Redux’s insistence on pure reducer functions in a subsequent module.

**Try it out**

We’ll write our testing code at the bottom of App.js, below the definition for reducer().

Our createStore() function now accepts initialState as an argument. Let’s first define this variable:

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

const initialState = { messages: [] };

And then initialize the store:

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

const store = createStore(reducer, initialState);

Let’s add code to dispatch add message actions to the store. This time, we’ll save each state “version” in two variables, stateV1 and stateV2. We’ll print out the two versions of our state at the end:

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

const addMessageAction1 = {

type: 'ADD\_MESSAGE',

message: 'How does it look, Neil?',

};

store.dispatch(addMessageAction1);

const stateV1 = store.getState();

const addMessageAction2 = {

type: 'ADD\_MESSAGE',

message: 'Looking good.',

};

store.dispatch(addMessageAction2);

const stateV2 = store.getState();

console.log('State v1:');

console.log(stateV1);

console.log('State v2:');

console.log(stateV2);

While we’re inside of a create-react-app project, we don’t yet have any React components. So we’ll just run App.js with babel-node:

./node\_modules/.bin/babel-node src/App.js

Which yields the following result:

State v1:

{ messages: [ 'How does it look, Neil?' ] }

State v2:

{ messages: [ 'How does it look, Neil?', 'Looking good.' ] }

Importantly, the state object was not modified between dispatches. We saved the first version of the state as the variable stateV1. Although this object was passed into reducer(), reducer() did not modify it. Instead, it created a new object with our second message appended. This new object was returned and set to the variable stateV2.

**Handling the DELETE\_MESSAGE action**

As discussed, the DELETE\_MESSAGE action object has the following shape:

{ type: 'DELETE\_MESSAGE', index: 2, // <- index of whichever message is being removed here

}

To support this action, we need to add a new else if statement to handle an action with a type of

'DELETE\_MESSAGE'. When the reducer receives this action, it should return an object with a messages array that contains every message except the one specified by the action’s index property.

We can create a new object as we did in ADD\_MESSAGE. That new object will contain a new messages array that includes all of the elements in state.messages except the one being removed. To do this in JavaScript, we can create a new array that contains:

• All of the elements from 0 to action.index

• All of the elements from action.index + 1 to the end of the array

We use Array’s slice to grab the desired “chunks” of the array:

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

function reducer(state, action) {

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

return {

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

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

return {

messages: [

...state.messages.slice(0, action.index),

...state.messages.slice( action.index + 1, state.messages.length ),

],

};

} else {

return state;

}

}

Importantly, slice does not modify the original array. Instead, it returns a new array with the elements in the range you specify. We create a new array that combines two ranges: up to and excluding action.index and every element after action.index.

**Try it out**

At the very bottom of App.js, we’ll add on to our code that tested the ADD\_MESSAGE action. Write the following below the last console.log() statement in the file:

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

const deleteMessageAction = {

type: 'DELETE\_MESSAGE',

index: 0,

};

store.dispatch(deleteMessageAction);

const stateV3 = store.getState();

console.log('State v3:');

console.log(stateV3);

By the second version of the state, we’ve added two messages to the state. We then dispatch a DELETE\_MESSAGE action, specifying the message at index 0.

Run the file with babel-node:

./node\_modules/.bin/babel-node src/App.js

As expected, in the third version of the state the first message has been removed:

State v1:

{ messages: [ 'How does it look, Neil?' ] }

State v2:

{ messages: [ 'How does it look, Neil?', 'Looking good.' ] }

State v3:

{ messages: [ 'Looking good.' ] }

**Subscribing to the store**

Our store so far provides methods for the view to dispatch actions and to read the current version of the state.

One important feature is missing before we can connect the store to React, however. While the view can read the state at any time with getState(), the view needs to know when the state has changed. Constantly polling the store with getState() is inefficient.

In our previous apps, when we wanted to modify the state we called setState(). Importantly, setState() triggers a render() call on the component.

Now, state is being modified outside of React and inside of the store. Our views are unaware of when it changes. If we’re going to keep our views up to date with the most current state in the store, then our views should receive a notification whenever the state changes.

Our store will use the observer pattern to allow the views to immediately update when the state changes. The views will register a callback function that they would like to be invoked when the state changes. The store will keep a list of all of these callback functions. When the state changes, the store will invoke each function, “notifying” the listeners of the change.

The best way to illustrate this pattern is to implement it.

Inside createStore(), we will:

1. Define an array called listeners

2. Add a subscribe() method which adds a new listener to listeners

3. Call each listener function when the state is changed

1. Define an array called listeners

We declare listeners at the top of createStore():

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

function createStore(reducer, initialState) {

let state = initialState;

const listeners = [];

// ...

Add a subscribe() method which adds a new listener to listeners Next, below the declaration of listeners, let’s add subscribe():

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

const subscribe = (listener) => (

listeners.push(listener)

);

The listener argument of subscribe() is a function, the function that the view would like invoked whenever the state changes. We add this function to the listeners array.

To make subscribe() accessible, we need to expose subscribe by adding it to the store object returned by createStore():

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

// ...

return {

**subscribe,**

getState,

dispatch,

};

Call each listener function when the state is changed

Whenever the state changes, we need to invoke all the functions kept in listeners. The state may change whenever we dispatch actions. As such, we’ll add the invocation logic to dispatch():

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

// ...

const dispatch = (action) => {

state = reducer(state, action);

listeners.forEach(l => l());

};

// ...

Note that there are no arguments passed to the listeners. This callback is solely a notification that the state changed.

createStore() in full

Here’s our createStore() function, in full:

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

function createStore(reducer, initialState) {

let state = initialState;

const listeners = [];

const subscribe = (listener) => (

listeners.push(listener)

);

const getState = () => (state);

const dispatch = (action) => {

state = reducer(state, action);

listeners.forEach(l => l());

};

return {

subscribe,

getState,

dispatch,

};

}

Stripped of comments, warnings, and sanity checks, the Redux library’s createStore() looks and behaves quite like our function.

Try it out

With subscribe() in place, our store is complete. Let’s test everything out.

In App.js, clear out all the previous testing code below this line where we initialize the store:

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

const store = createStore(reducer, initialState);

We’ll dispatch add and delete message actions like before. Except, now we’ll use subscribe() to register a function that will perform a console.log() every time the state changes. Our listener prints the current state to the console:

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

const listener = () => {

console.log('Current state: ');

console.log(store.getState());

};

Next, let’s subscribe this listener: redux/chat\_simple/src/complete/App-4.js

store.subscribe(listener);

Now, we can dispatch our actions. After every dispatch() call, the listening function we passed to subscribe() will be called, writing to the console:

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

const addMessageAction1 = {

type: 'ADD\_MESSAGE',

message: 'How do you read?',

};

store.dispatch(addMessageAction1);

// -> `listener()` is called

const addMessageAction2 = {

type: 'ADD\_MESSAGE',

message: 'I read you loud and clear, Houston.',

};

store.dispatch(addMessageAction2); // -> `listener()` is called

const deleteMessageAction = {

type: 'DELETE\_MESSAGE',

index: 0, };

store.dispatch(deleteMessageAction); // -> `listener()` is called

Save App.js and run it with babel-node:

$ ./node\_modules/.bin/babel-node src/App.js

And note the output:

Current state:

{ messages: [ 'How do you read?' ] }

Current state:

{ messages: [ 'How do you read?', 'I read you loud and clear, Houston.' ] }

Current state:

{ messages: [ 'I read you loud and clear, Houston.' ] }

With our store feature complete, we’re prepared to connect our Redux store to some React views and see a complete, working Redux pipeline.

**Connecting Redux to React**

Revisiting the Flux diagram from earlier, we can now explore the specifics behind how Redux and React work together to fulfill this design pattern:

**Using store.getState()**

React is no longer managing state. Redux is. Therefore, top-level React components will use store.getState() as opposed to this.state to drive their render() functions. The state provided by Redux will trickle down from there.

For instance, if we want to render our state’s messages, we fetch them from our Redux store:

// An example top-level component

class App extends React.Component {

// ...

render() {

const messages = store.getState().messages;

// ...

}

};

**Using store.subscribe()**

When React manages state, we call setState() to modify this.state. setState() will trigger a re-render after the state is modified.

When Redux is managing state, we use subscribe() inside the top-level React component to setup a listening function that initiates the re-render.

We can subscribe our component inside of componentDidMount. The listening function that we pass to subscribe() will call this.forceUpdate(), triggering the component (this) to re-render. As an example, subscribing a React component looks like this:

// An example top-level component

class App extends React.Component {

// ...

componentDidMount() {

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

}

// ...

};

**Using store.dispatch()**

Lower-level components will dispatch actions in response to events that should modify state. For instance, a React component might dispatch an action to the store whenever a delete button is clicked:

// An example leaf component

class Message extends React.Component {

handleDeleteClick = () => {

store.dispatch({

type: 'DELETE\_MESSAGE',

index: this.props.index,

});

};

// ...

};

This dispatch() call will modify the state. dispatch() will then invoke the listener, which we registered with subscribe(). This forces the App component to re-render. When render() is invoked, the App component reads from the store again with getState(). App then passes the latest version of the state down to its child components.

This cycle repeats every time React dispatches an action.

The app’s components

The chat app has three React components:

• App: The top-level container

• MessageView: The list of messages

• MessageInput: The input to add new messages

As we saw, the input box enables adding messages. Clicking on a message deletes it.

MessageView will render the state’s messages. It will also dispatch DELETE\_MESSAGE actions every time the user clicks an individual message.

MessageInput will not use state to render. However, it will dispatch ADD\_MESSAGE actions every time the user submits a new message.

We could have broken MessageView into MessageList and Message. This would follow the pattern from previous apps. However, each message is simple enough at the moment that this is not necessary.

**Preparing App.js**

For this project, the store logic and React components will all be inside src/App.js. We will be able to reference store directly in each of our components.

In more complex applications, the store will likely be located in a different file than React components. In a later module, we’ll explore other ways to have React components communicate with a Redux store object.

Clear out all the testing code below the declaration of store at the end of src/App.js:

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

const store = createStore(reducer, initialState);

The App component

App is the top-level React component in our app. App will be the component that reads from the store. We’ll need it to subscribe to our Redux store.

**Subscribing to changes**

Like we talked about above, we will subscribe() inside of componentDidMount. The callback function we give to subscribe() calls this.forceUpdate(). This will cause the App component to re-render every time the state changes:

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

class App extends React.Component {

componentDidMount() {

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

}

**Rendering the view**

In render, we’ll first use getState() to read messages from the store. We’ll then render the two children, MessageView and MessageInput. Only MessageView needs the list of messages:

render() {

const messages = store.getState().messages;

return (

<div className='ui segment'>

<MessageView messages={messages} />

<MessageInput />

</div>

);

}

We have our downward data pipeline, from the store through App down to MessageView. But what about the inverse direction? We want MessageView to have the ability to delete messages and MessageInput to be able to add them.

When React is managing state, we pass down functions as props from state-managing components to children. This enables children to propagate events up to the parent who modifies the state.

Now, we have a store object that we can dispatch actions to. While we could still concentrate all communication with the store inside of App, let’s explore allowing our child components to dispatch actions to the store directly.

The App component **in full**:

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

class App extends React.Component {

componentDidMount() {

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

}

render() {

const messages = store.getState().messages;

return (

<div className='ui segment'>

<MessageView messages={messages} />

<MessageInput />

</div>

);

}

}

**The MessageInput component**

MessageInput will have a single input field and a submit button. When the user clicks the submit button, the component should dispatch an ADD\_MESSAGE action.

As a controlled component, we’ll want to keep track of the value of the input in state somewhere. We could keep this state in our Redux store. But it’s usually easier to just keep form data in the form component’s state.

Let’s begin by defining the initial state as well as the onChange handler function:

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

class MessageInput extends React.Component {

state = {

value: '',

};

onChange = (e) => {

this.setState({

value: e.target.value,

})

};

Next, we’ll define handleSubmit(), the function that will call dispatch():

handleSubmit = () => {

store.dispatch({

type: 'ADD\_MESSAGE',

message: this.state.value,

});

this.setState({

value: '',

});

};

render will contain an input and a button wrapped in a div. button will have its onClick attribute set to this.handleSubmit:

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

render() {

return (

<div className='ui input'>

<input

onChange={this.onChange}

value={this.state.value}

type='text'

/>

<button

onClick={this.handleSubmit}

className='ui primary

button' type='submit'

>

Submit

</button>

</div>

);

}

**The MessageView component**

The MessageView component’s messages prop is an array of strings. MessageView will render these messages as a list. Furthermore, whenever the user clicks a message we want to dispatch a DELETE\_MESSAGE action.

Let’s begin by defining the component and its function handleClick(). handleClick() will be the function that calls dispatch(). handleClick() accepts one argument, index, which it uses in the action object it dispatches:

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

class MessageView extends React.Component {

handleClick = (index) => {

store.dispatch({

type: 'DELETE\_MESSAGE',

index: index,

});

};

The render function will use map to create the list of messages to render. We want each individual message to be wrapped inside of a div:

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

render() {

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

<div

className='comment'

key={index}

onClick={() => this.handleClick(index)}

>

{message}

</div>

));

On this div we set the onClick attribute. We want this to call a function that calls handleClick(), passing in the index of the target message.

Finally, we return messages wrapped inside of a div:

return (

<div className='ui comments'>

{messages}

</div>

);

Finally, we export App at the bottom of the file:

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

export default App;

We already include App in index.js and mount it to the DOM using ReactDOM.render(). Therefore, we’re ready to try everything out.

**Try it out**

Save App.js. In your terminal, from the root of the project folder, boot the server:

$ npm start

Add a few messages, then click on them to see them instantly disappear.