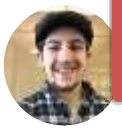


Jul 11, 2017 · 5 min read

Redux From Scratch (Chapter 8 | Dividing the State With Multiple Reducers)

Buy the Official Ebook

If you would like to support the author and receive a PDF, EPUB, and/or MOBI copy of the book, please purchase the official ebook.

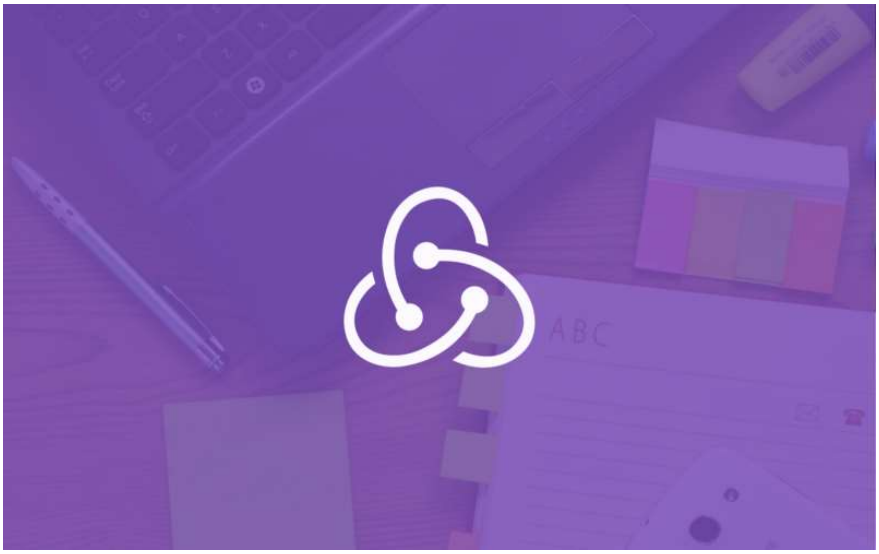
Prerequisites

Chapter 1 | Core Concepts
Chapter 2 | Practicing the Basics
Chapter 3 | Implementing With React
Chapter 4 | Async Practice With Twitch API
Chapter 5 | Implementing Middleware
Chapter 6 | React, Redux, Firebase Stack
Chapter 7 | Building an API Service With Express.js & MongoDB

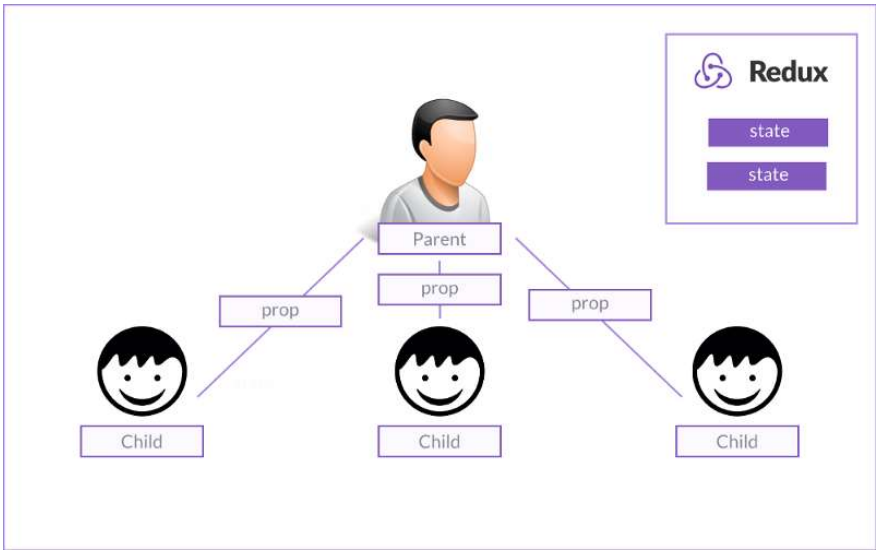
Scope of This Chapter

This will be the last chapter containing any code. Unlike other chapters, this is going to be super duper short and refreshing. We won't be doing a project. Instead, we will simply create a short demo on Codepen showing how to divide the state with multiple reducers.

Dividing the State With Multiple Reducers



In Chapter 1, I described Redux in such a fashion that alluded to the fact that it can handle multiple states within one store.



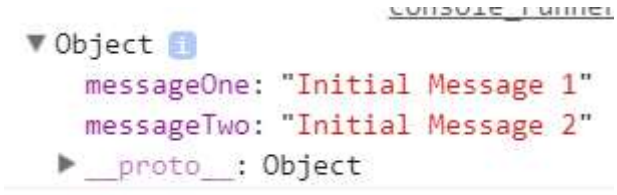
As you may have noticed, we have only been working with one state.

Theoretically, there is only one global state. However, we can divide the management of the global state into multiple reducers which all have their own state.

Let's take a look at how this can be achieved.

Before we begin, fork this template from Codepen.

We want to have the following state divided between two reducers:



Let's add the code piece by piece.

First, we can add a reducer that will just manage *messageOne*:

```
function messageOne(state = {message1: "Initial Message 1"},
action) {
  switch(action.type) {
    default:
      return state
  }
}
```

In the code above, we set the initial local state containing a property called *message1*. We have no logic to handle actions, but we do have the boilerplate.

We slightly tweak our first reducer to create our second reducer:

```
function messageTwo(state = {message2: "Initial Message 2"},
action) {
  switch(action.type) {
    default:
      return state
  }
}
```

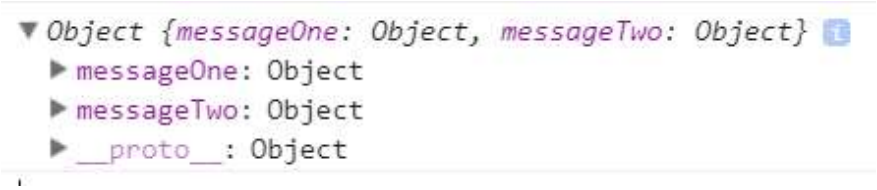
Once we have all the reducers split up, we can use `combineReducers` :

```
const displayMessages = Redux.combineReducers({
  messageOne,
  messageTwo
})
```

Note: When using npm and Webpack for your project configuration, it would just be `combineReducers` which would be imported like this:

```
import { combineReducers } from 'redux'
```

The *global* state now looks like this:



We can see that the *function names* of the reducers are the properties of the global state and are objects.

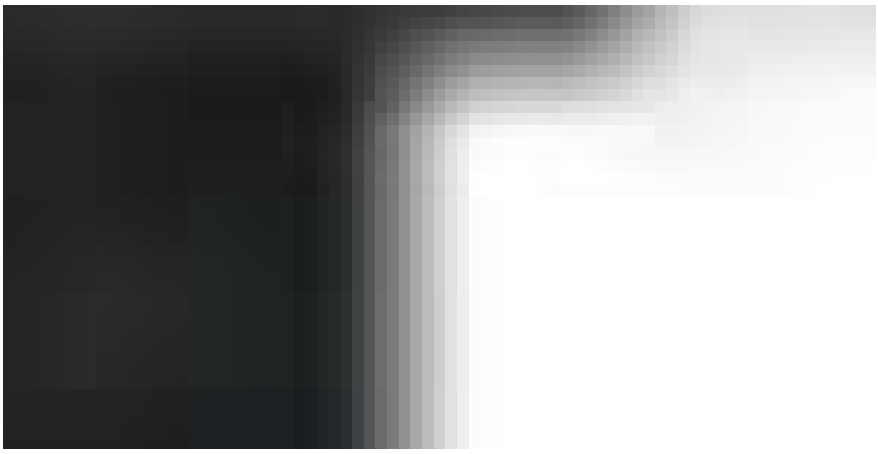
Each of these objects contains the properties of the local states:

We now have 2 local states within 1 global state containing the two messages as intended.

We can then wrap up this example by initializing the store, rendering the values from the global state and the subscription:

```
//intialize store
let store = Redux.createStore(displayMessages)

//render value of state to DOM
let valueTarget1 = document.getElementById('value1')
let valueTarget2 = document.getElementById('value2')

function render() {
  console.log(store.getState())
  valueTarget1.innerHTML =
store.getState().messageOne.message1
  valueTarget2.innerHTML =
store.getState().messageTwo.message2
}

render()

//subscribe to render
store.subscribe(render)
```

We now see the messages rendering as expected:



Cool!

The next step is going to be implementing actions to be handled by these reducers.

First, we can add the action creators:

```
//combineReducers here


//define actions creators
function updateMsg1() {
  const UPDATE_MSG_1 = 'UPDATE_MSG_1';
  return {
    type: UPDATE_MSG_1,
    newMessage: "Howdy! I'm a new property for local state
1!"
  }
}


function updateMsg2() {
  const UPDATE_MSG_2 = 'UPDATE_MSG_2';
  return {
    type: UPDATE_MSG_2,
    newMessage: "Howdy! I'm a new property for local state
2!"
  }
}


//initialize state here
```

Nothing unusual occurring here.

Next, we can add cases to each reducer to handle these actions and update their local states:

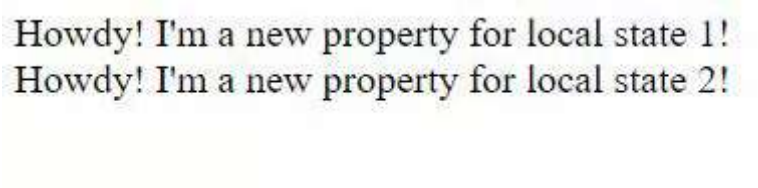
```
//define a reducer with an initialized state and logic to
handle action
function messageOne(state = {message1: "Initial Message 1"},
action) {
  switch(action.type) {
    case 'UPDATE_MSG_1':
      state.message1 = action.newMessage
      return state
    default:
      return state
  }
}
```

```
function messageTwo(state = {message2: "Initial Message 2"},
action) {
  switch(action.type) {
    case 'UPDATE_MSG_2':
      state.message2 = action.newMessage
      return state
    default:
      return state
  }
}
```

Finally, we can dispatch these actions using `setTimeout()` :

```
//dispatch actions after 2 seconds
setTimeout( () => {
  store.dispatch(updateMsg1())
  store.dispatch(updateMsg2())
}, 2000)
```

We should now see our actions updating properties from each local state:



Final Pen

. . .

I also want to quickly mention that you could store all reducers and the global state created by `combineReducers` in separate files.

To do this, you would just place each reducer in a separate file and make it exportable like this:

```
function messageTwo(state = {message2: "Initial Message 2"},
action) {
  switch(action.type) {
    case 'UPDATE_MSG_2':
      state.message2 = action.newMessage
      return state
    default:
      return state
  }
}

export default messageTwo
```

You could then import the reducers into the file containing the `combineReducers` code (perhaps *State.js*). For example:

```
import messageOne from './reducers';
import messageTwo from './reducers';


const displayMessages = combineReducers({
  messageOne,
  messageTwo
})
```

Looks pretty neat 😀

Concluding Thoughts

This chapter was originally intended to cover any remaining topics that were really critical to development with Redux. The only thing I could think of was dividing the state with multiple reducers.

I also entertained the thought of a complete chapter on having better Redux practices, however, the official documentation has plenty some recipes that should suffice if you are curious.

So far, we have completely covered Redux from the basic principles to using it within a full-stack. I'm not sure about you, but I'm ready to wrap this up.

We will do exactly that in the next chapter.

Final Chapter

The final chapter is now available.

Buy the Official Ebook

If you would like to support the author and receive a PDF, EPUB, and/or MOBI copy of the book, please purchase the official ebook.

• • •

Cheers,
Mike Mangialardi