EVERYTHING YOU NEED
TO BUILD REAL PROJECTS
WITH REDUX

# THE COMPLETE REDUX BOOK

BORIS DINKEVICH
ILYA GELMAN

# The Complete Redux Book

## Everything you need to build real projects with Redux

Ilya Gelman and Boris Dinkevich

This book is for sale at http://leanpub.com/redux-book

This version was published on 2018-04-19

ISBN 978-965-92642-0-9


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2018 Ilya Gelman and Boris Dinkevich

# Tweet This Book!

Please help Ilya Gelman and Boris Dinkevich by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Time to learn Redux!

The suggested hashtag for this book is #ReduxBook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ReduxBook

# Contents

# Chapter 5. Reducers

The word *reducer* is commonly associated in computer science with a function that takes an array or object and converts it to a simpler structure—for example, summing all the items in an array. In Redux, the role of the reducer is somewhat different: reducers create a new state out of the old one, based on an action.

In essence, a reducer is a simple JavaScript function that receives two parameters (two objects—the previous state and an action) and returns an object (a modified copy of the first argument):

**A simple reducer example**

```
const sum = (result, next) => result + next;

[1,2,3].reduce(sum); // -> 6
```

Reducers in Redux are *pure functions*, meaning they don't have any side effects such as changing `localStorage`, contacting the server, or saving any data in variables. A typical reducer looks like this:

**A basic reducer**

```
function reducer(state, action) {
  switch (action.type) {

    case 'ADD_RECIPE':
    // Create new state with a recipe

    default:
      return state;
  }
}
```

## Reducers in Practice

In Redux, reducers are the final stage in the unidirectional data flow. After an action is dispatched to the store and has passed through all the middleware, reducers receive it together with the current state of the application. Then they create a new state that has been modified according to the action and return it to the store.

The way we connect the store and the reducers is via the `createStore()` method, which can receive three parameters: a reducer, an optional initial state, and an optional store enhancer (covered in detail in the Store chapter).

As an example, we will use an application similar to the one used in the Example Redux Application chapter—a simple recipe book application.

Our state contains three substates:

- `recipes`—A list of recipes
- `ingredients`—A list of ingredients and quantities used in each recipe
- `ui`—An object containing the state of various UI elements

And we will support the following actions:

- `ADD_RECIPE`
- `FETCH_RECIPES`
- `SET_RECIPES`

## A Simple Reducer

The simplest approach to building a reducer would be to use a large `switch` statement that knows how to handle all the actions our application supports:

**Using a switch statement to build a reducer**

```
switch (action.type) {
  case 'ADD_RECIPE':
    // TODO: handle add recipe action

  case 'FETCH_RECIPES':
    // TODO: handle fetch recipes action

  ...
}
```

But it is quite clear that this approach will break down fast as our application (and the number of actions) grows.

## Reducer Separation

The obvious solution would be to find a way to split the reducer code into multiple files, or multiple reducers. Since `createStore()` receives only one reducer, it will be that reducer's job to call other reducers to help it calculate the new state.

The simplest method of determining how to split the reducer code is by examining the state we need to handle:

**Our example state**

```
const state = {
  recipes: [],
  ingredients: [],
  ui: {}
}
```

We can now create three different reducers, each responsible for part of the state:

**Three different reducers**

```
const recipesReducer = (state, action) => {};
const ingredientsReducer = (state, action) => {};
const uiReducer = (state, action) => {};

const reducer = (state, action) => {
  // TODO: combine the result of all substate reducers
}
```

Since each of the reducers calculates a new state (or returns the original if it does not recognize the action), we can build a new state by calling all the reducers one after another:

**Reducers running in sequence**

```
const recipesReducer = (state, action) => {};
const ingredientsReducer = (state, action) => {};
const uiReducer = (state, action) => {};

const reducer = (state, action) => {
  let newState = state;

  newState = recipesReducer(newState, action);
  newState = ingredientsReducer(newState, action);
  newState = uiReducer(newState, action);

  return newState;
}
```

While this approach works correctly, you might have noticed a potential problem. Why does the `recipesReducer` reducer need to access and calculate the whole state, instead of only the `recipes` substate? We can further improve our reducers by having each one act on only the substate it cares about:

**Substate handling reducers**

```
const recipesReducer = (recipes, action) => {};
const ingredientsReducer = (ingredients, action) => {};
const uiReducer = (ui, action) => {};

const reducer = (state, action) => {
  const newState = Object.assign({}, state, {
    recipes: recipesReducer(state.recipes, action),
    ingredients: ingredientsReducer(state.ingredients, action),
    ui: uiReducer(state.ui, action)
  });

  return newState;
}
```

With this new code, each reducer receives only the part of the state that is relevant to it and can't affect other parts. This separation proves very powerful in large-scale projects, as it means developers can rely on reducers being able to modify only the parts of the state they are connected to and never causing clashes.

Another side effect of this separation of concerns is that our reducers become much simpler. Since they no longer have to calculate the whole state, a large part of the code is no longer needed:

**Recipes reducer before substate**

```
const recipesReducer = (state, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return Object.assign({}, state, {
        recipes: state.recipes.concat(action.payload);
      })
    ...
  }
}
```

**Recipes reducer after substate**

```
const recipesReducer = (recipes, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return recipes.concat(action.payload);
    ...
  }
}
```

## Combining Reducers

The technique of reducer combination is so convenient and broadly used that Redux provides a very useful function named `combineReducers()` to facilitate it. This helper function does exactly what `rootReducer()` did in our earlier example, with some additions and validations:

**Root reducer using combineReducers()**

```
import { combineReducers } from 'redux';
import recipesReducer from 'reducers/recipes';
import ingredientsReducer from 'reducers/ingredients';

const rootReducer = combineReducers({
  recipes: recipesReducer,
  ingredients: ingredientsReducer
});

export const store = createStore(rootReducer);
```

We can make this code even simpler by using ES2015's property shorthand feature:

**Using ES2015 syntax in combineReducers()**

```
import { combineReducers } from 'redux';
import recipes from 'reducers/recipes';
import ingredients from 'reducers/ingredients';
import ui from 'reducers/ui';

export default combineReducers({
  recipes,
  ingredients,
  ui
});
```

In this example we provided `combineReducers()` with a configuration object holding keys named `recipes`, `ingredients`, and `ui`. The ES2015 syntax we used automatically assigned the value of each key to be the corresponding reducer.

It is important to note that `combineReducers()` is not limited only to the root reducer. As our state grows in size and depth, nested reducers will be combining other reducers for substate calculations. Using nested `combineReducers()` calls and other combination methods is a common practice in larger projects.

## Default Values

One of the requirements of `combineReducers()` is for each reducer to define a default value for its substate. Using this approach, the default structure of the state tree is dynamically built by the reducers themselves. This guarantees that changes to the tree require changes only to the applicable reducers and do not affect the rest of the tree.

This is possible because when the store is created, Redux dispatches a special action called `@@redux/INIT`. Each reducer receives that action together with the undefined initial state, which gets replaced with the default parameter defined inside the reducer. Since our `switch` statements do not process this special action type and simply return the state (previously assigned by the default parameter), the initial state of the store is automatically populated by the reducers.

To support this, each of the subreducers must define a default value for its first argument, to use if none is provided:

**Defining a default substate**

```
const initialState = [];

const recipesReducer = (recipes = initialState, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return recipes.concat(action.payload);
    ...
  }
}
```

## Tree Mirroring

This brings us to an important conclusion: that we want to structure our reducers tree to mimic the application state tree. As a rule of thumb, we will want to have a reducer for each leaf of the tree. Mimicking this structure in the reducers directory will make it self-depicting of how the state tree is structured.

As complicated manipulations might be required to add some parts of the tree, some reducers might not neatly fall into this pattern. We might find ourselves with two or more reducers processing the same subtree (sequentially), or a single reducer operating on multiple branches (if it needs to update structures in different branches). This might cause complications in the structure and composition of our application. Such issues can usually be avoided by normalizing the tree, splitting a single action into multiple ones, and other techniques.

## Alternative to switch Statements

In Redux, most reducers are just `switch` statements over `action.type`. Since the `switch` syntax can be hard to read and prone to errors, there are a few libraries that try to make writing reducers easier and cleaner.

While it is most common for a reducer to examine the `type` property of the action to determine if it should act, in some cases other parts of the action object are used. For example, you might want to show an error notification on every action that has an error in the payload.

The `redux-actions` library described in the previous chapter provides the `handleActions()` utility function for reducer generation:

**Using redux-actions for reducer creation**

```
import { handleActions } from 'redux-actions';

const initialState = [];

const recipesReducer = handleActions({
  [ADD_RECIPE]: (recipes, action) =>
    recipes.concat(action.payload),

  [REMOVE_RECIPE]: (recipes, action) =>
    recipes.filter(recipe => recipe.id === action.payload)

}, initialState);

export default recipesReducer;
```

If you are using Immutable.js, you might also want to take a look at the `redux-immutablejs`[1] library, which provides you with `createReducer()` and `combineReducers()` functions that are aware of Immutable.js features like getters and setters.

# Avoiding Mutations

The most important thing about reducers in Redux is that *they should never mutate the existing state*. There are a number of functions in JavaScript that can help when working with immutable objects. Before we look at those, however, let's consider why this is so important.

## Why Do We Need to Avoid Mutations?

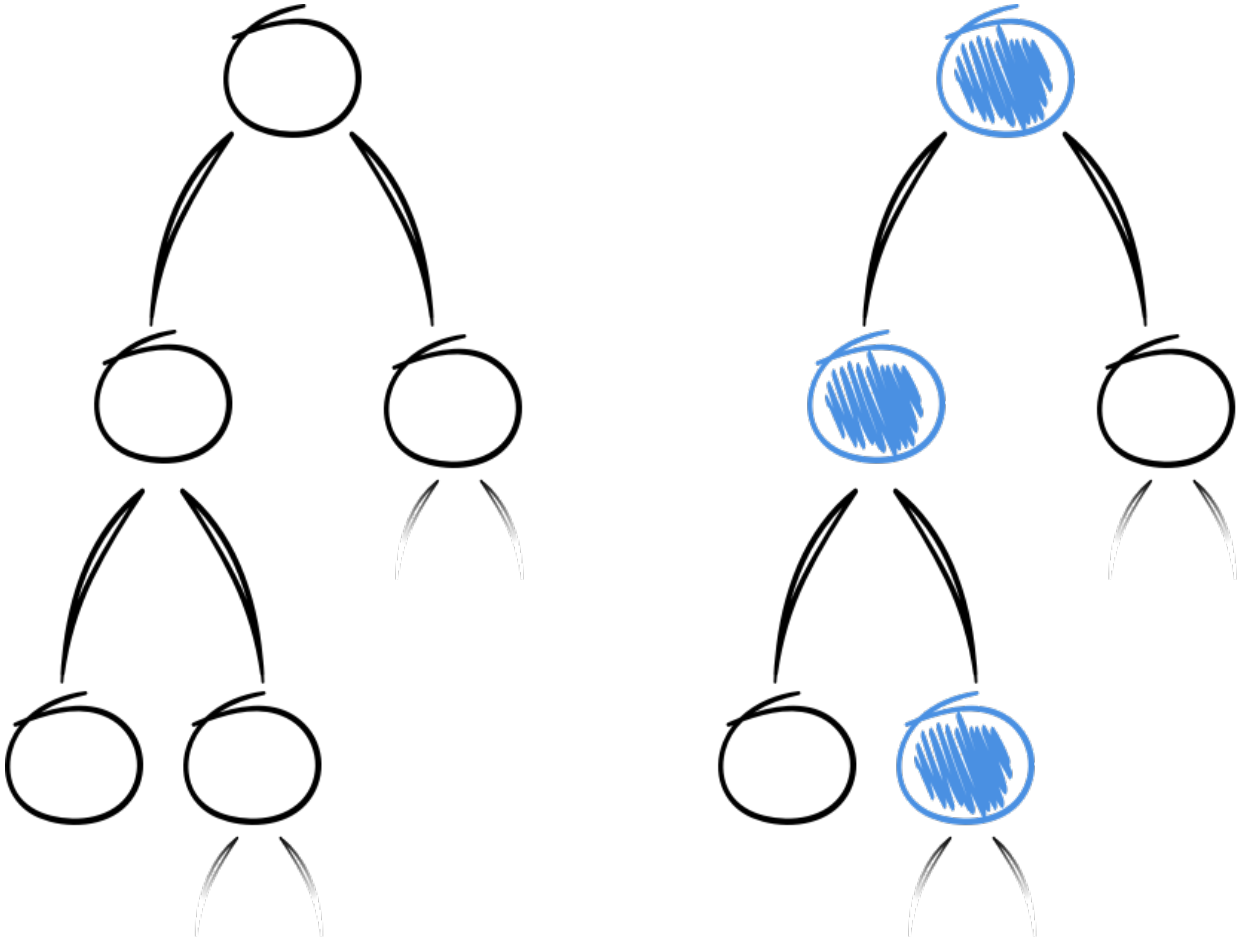
One of the reasons behind the immutability requirement for reducers is due to *change detection*. After the store passes the current state and action to the root reducer, it and various UI components of the application need a way to determine what changes, if any, have happened to the global state. For small objects, a deep compare or other similar methods might suffice. But if the state is large and only a small part may have changed due to an action, we need a faster and better method.

There are a number of ways to detect a change made to a tree, each with its pros and cons. Among the many solutions, one is to mark where changes were made in the tree. We can use simple methods like setting a `dirty` flag, use more complicated approaches like adding a version to each node, or (the preferred Redux way) use *reference comparison*.

Redux and its accompanying libraries rely on reference comparison. After the root reducer has run, we should be able to compare the state at each level of the state tree with the same level in the previous version of the tree to determine if it has changed. But instead of comparing each key and value, we can compare just the reference or *pointer* to the structure.

In Redux, each changed node or leaf is replaced by a new copy of itself that incorporates the changed data. Since the node's parent still points to the old copy of the node, we need to create a copy of it as well, with the new copy pointing to the new child. This process continues with each parent being recreated until we reach the root of the tree. This means that a change to a leaf must cause its parent, the parent's parent, etc. to be modified. In other words, it causes new objects to be created. The following illustration shows the state before and after it is run through the reducers tree and highlights the changed nodes.

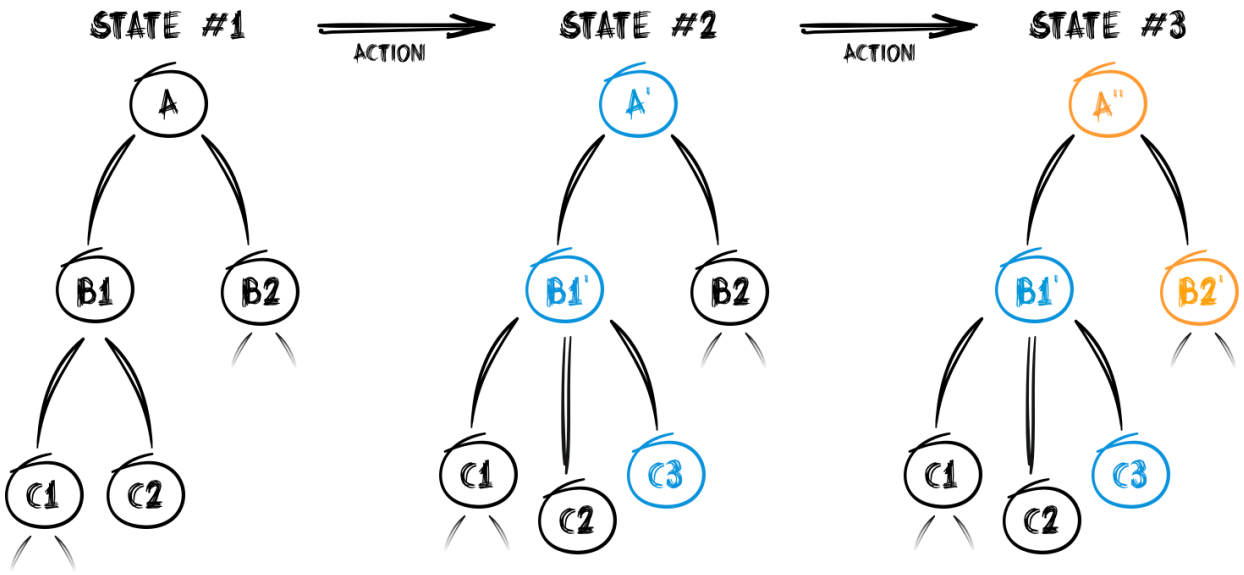
[1] https://github.com/indexiatech/redux-immutablejs



**Example of changing nodes**

The main reason for using reference comparison is that this method ensures that each reference to the previous state is kept coherent. We can examine the reference at any time and get the state exactly as it was before a change. If we create an array and push the current state into it before running actions, we will be able to pick any of the pointers to the previous state in the array and see the state tree exactly as it was before all the subsequent actions happened. And no matter how many more actions we process, our original pointers stay exactly as they were.

This might sound similar to copying the state each time before changing it, but the reference system will not require 10 times the memory for 10 states. It will smartly reuse all the unchanged nodes. Consider the next illustration, where two different actions have been run on the state, and how the three trees look afterward.




**Example of saving references**

The first action added a new node, C3, under B1. If we look closely we can see that the reducer didn't change anything in the original A tree. It only created a new A' object that holds B2 and a new B1' that holds the original C1 and C2 and the new C3'. At this point we can still use the A tree and have access to all the nodes like they were before. What's more, the new A' tree didn't copy the old one, but only created some new links that allow efficient memory reuse.

The next action modified something in the B2 subtree. Again, the only change is a new A'' root object that points to the previous B1' and the new B2'. The old states of A and A' are still intact and memory is reused between all three trees.

Since we have a coherent version of each previous state, we can implement nifty features like undo and redo (we save the previous state in an array and, in the case of "undo," make it the current one). We can also implement more advanced features like "time travel," where we can easily jump between versions of our state for debugging.

## What Is Immutability?

Let's define what the word *mutation* means. In JavaScript, there are two types of variables: ones that are copied by value, and ones that are passed by reference. Primitive values such as numbers, strings, and Booleans are copied when you assign them to other variables, and a change to the target variable will not affect the source:

**Primitive values example**

```
let string = "Hello";
let stringCopy = string;

stringCopy += " World!";

console.log(string); // => "Hello"
console.log(stringCopy); // => "Hello World!"
```

In contrast, *collections* in JavaScript aren't copied when you assign them; they only receive a pointer to the location in memory of the object pointed to by the source variable. This means that any change to the new variable will modify the same memory location, which is pointed to by both old and new variables:

**Collections example**

```
const object = {};
const referencedObject = object;

referencedObject.number = 42;

console.log(object); // => { number: 42 }
console.log(referencedObject); // => { number: 42 }
```

As you can see, the original object is changed when we change the copy. We used `const` here to emphasize that a constant in JavaScript holds only a pointer to the object, not its value, and no error will be thrown if you change the properties of the object (or the contents of an array). This is also true for collections passed as arguments to functions, as what is being passed is the reference and not the value itself.

Luckily for us, ES2015 lets us avoid mutations for collections in a much cleaner way than before, thanks to the `Object.assign()` method and the *spread operator*.

The spread operator is fully supported by the ES2015 standard. More information is available on MDN[2]. Complete `Object.assign()` documentation is also available on MDN[3]. MDN is great!

## Objects

`Object.assign()` can be used to copy all the key/value pairs of one or more source objects into one target object. The method receives the following parameters:

1. The target object to copy *to*
2. One or more source objects to copy *from*

Since our reducers need to create a new object and make some changes to it, we will pass a new empty object as the first parameter to `Object.assign()`. The second parameter will be the original subtree to copy and the third will contain any changes we want to make to the object. This will result in us always having a fresh object with a new reference, having all the key/value pairs from the original state and any overrides needed by the current action:

**Example of Object.assign()**

```
function reduce(state, action) {
  const overrides = { price: 0 };

  return Object.assign({}, state, overrides);
}

const state = { ... };
const newState = reducer(state, action);

state === newState; // false!
```

Deleting properties can be done in a similar way using ES2015 syntax. To delete the key `name` from our state we can use the following:

**Example of deleting a key from an object**

```
return Object.assign({}, state, { name: undefined } );
```

[2] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator
[3] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

## Arrays

Arrays are a bit trickier, since they have multiple methods for adding and removing values. In general, you just have to remember which methods create a new copy of the array and which change the original one. For your convenience, here is a table outlining the basic array methods:

| Safe methods | Mutating methods |
|---|---|
| `concat()` | `push()` |
| `slice()` | `splice()` |
| `map()` | `pop()` |
| `reduce()` | `shift()` |
| `reduceRight()` | `unshift()` |
| `filter()` | `fill()` |
| | `reverse()` |
| | `sort()` |
| | `copyWithin()` |

The basic array operations we will be doing in most reducers are appending, deleting, and modifying an array. To keep to the immutability principles, we can achieve these using the following methods:

**Adding items to an array**

```
const reducer = (state, action) =>
  state.concat(newValue);
```

**Removing items from an array**

```
const reducer = (state, action) =>
  state.filter(item => item.id !== action.payload);
```

**Changing items in an array**

```
const reducer = (state, action) =>
  state.map((item) => item.id !== action.payload
    ? item
    : Object.assign({}, item, { favorite: action.payload })
  );
```

# Ensuring Immutability

The bitter truth is that in teams with more than one developer, we can't always rely on everyone avoiding state mutations all the time. As humans, we make mistakes, and even with the strictest pull request, code review, and testing practices, sometimes they crawl into the code base. Fortunately, there are a number of methods and tools that strive to protect developers from these hard-to-find bugs.

One approach is to use libraries like `deep-freeze`[4] that will throw errors every time someone tries to mutate a "frozen" object. While JavaScript provides an `Object.freeze()` method, it freezes only the object it is applied to, not its children. `deep-freeze` and similar libraries perform nested freezes and method overrides to better catch such errors.

Another approach is to use libraries that manage truly immutable objects. While they add additional dependencies to the project, they provide a variety of benefits as well: they ensure true immutability, offer cleaner syntax to update collections, support nested objects, and provide performance improvements on very large data sets.

The most common library is Facebook's Immutable.js[5], which offers a number of key advantages (in addition to many more advanced features):

- Fast updates on very large objects and arrays
- Lazy sequences
- Additional data types not available in plain JavaScript
- Convenient methods for deep mutation of trees
- Batched updates

It also has a few disadvantages:

- It's an additional large dependency for the project.
- It requires the use of custom getters and setters to access the data.
- It might degrade performance where large structures are not used.

It is important to carefully consider your state tree before choosing an immutable library. The performance gains might only become perceptible for a small percentage of the project, and the library will require all of the developers to understand a new access syntax and collection of methods.

[4] https://www.npmjs.com/package/deep-freeze
[5] https://facebook.github.io/immutable-js/

Another library in this space is `seamless-immutable`[6], which is smaller, works on plain JavaScript objects, and treats immutable objects the same way as regular JavaScript objects (though it has similar convenient setters to Immutable.js). Its author has written a great post[7] where he describes some of the issues he had with Immutable.js and what his reasoning was for creating a smaller library.

`seamless-immutable` does not offer many of the advantages of Immutable.js (sequences, batching, smart underlying data structures, etc.), and you can't use advanced ES2015 data structures with it, such as `Map`, `Set`, `WeakMap`, and `WeakSet`.

The last approach is to use special helper functions that can receive a regular object and an instruction on how to change it and return a new object as a result. The `immutability-helper` library[8] provides one such function, named `update()`. Its syntax may look a bit weird, but if you don't want to work with immutable objects and clog object prototypes with new functions, it might be a good option:

**Example of using the update() function**

```
import update from 'immutability-helper';

const newData = update(myData, {
  x: { y: { z: { $set: 7 }}},
  a: { b: { $push: [9] }}
});
```

## Using Immer for Temporary Mutations

A word from our friend Michel Weststrate[9], creator of MobX[10] and Immer[11].

When writing reducers it can sometimes be beneficial to temporarily use mutable objects. This is fine as long as you only mutate new objects (and not an existing state), and as long as you don't try to mutate the objects after they have left the reducer.

[6] https://github.com/rtfeldman/seamless-immutable
[7] http://tech.noredink.com/post/107617838018/switching-from-immutablejs-to-seamless-immutable
[8] https://github.com/kolodny/immutability-helper
[9] https://twitter.com/mweststrate
[10] https://mobx.js.org
[11] https://github.com/mweststrate/immer

Immer is a tiny library that expands this idea and makes it easier to write reducers. It is comparable in functionality to the `withMutations()` method in Immutable.js, but applied to regular JavaScript structures. The advantage of this approach is that you don't have to load an additional library for data structures. Nor do you need to learn a new API to perform complex mutations in reducers. And last but not least, TypeScript and Flow are perfectly able to type-check the reducers that are created using Immer.

Let's take a quick look at Immer. The Immer package exposes a `produce()` function that takes two arguments: the *current state* and a *producer function*. The producer function is called by `produce()` with a *draft*.

The draft is a virtual state tree that reflects the entire current state. It will record any changes you make to it. The `produce()` function returns the next state by combining the current state and the changes made to the draft.

So, let's say we have the following example reducer:

**Example reducer**

```
const byId = (state, action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      return {
        ...state,
        ...action.products.reduce((obj, product) => {
          obj[product.id] = product
          return obj
        }, {})
      }

    default:
      return state
  }
};
```

This may be hard to grasp at first glance because there is quite a bit of noise, resulting from the fact that we are manually building a new state tree. With Immer, we can simplify this to:

**Example reducer with Immer**

```
const byId = (state, action) =>
  produce(state, draft => {
    switch (action.type) {
      case RECEIVE_PRODUCTS:
        action.products.forEach(product => {
          draft[product.id] = product
        });

        break;
    }
  });
```

The reducer will now return the next state produced by the producer. If the producer doesn't do anything, the next state will simply be the original state. Because of this, we don't have to handle the default case.

Immer will use structural sharing, just like if we had written the reducer by hand. Beyond that, because Immer knows which parts of the state were modified, it will also make sure that the modified parts of the tree will automatically be frozen in development environments. This prevents accidentally modifying the state after `produce()` has ended.

To further simplify reducers, the `produce()` function supports currying. It is possible to call `produce()` with just the producer function. This will create a new function that will execute the producer with the state as an argument. This new function also accepts an arbitrary amount of additional arguments and passes them on to the producer. This allows us to write the reducer solely in terms of the draft itself:

**Simplified reducer with Immer**

```
const byId = produce((draft, action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      action.products.forEach(product => {
        draft[product.id] = product
      });

      break;
  }
})
```

If you want to take full advantage of Redux, but still like to write your reducers with built-in data structures and APIs, make sure to give Immer a try.

# Higher-Order Reducers

The power of Redux is that it allows you to solve complex problems using functional programming. One approach is to use *higher-order functions*. Since reducers are nothing more than pure functions, we can wrap them in other functions and create very simple solutions for very complicated problems.

There are a few good examples of using higher-order reducers—for example, for implementing undo/redo functionality. There is a library called `redux-undo`[12] that takes your reducer and enhances it with undo functionality. It creates three substates: *past*, *present*, and *future*. Every time your reducer creates a new state, the previous one is pushed to the past states array and the new one becomes the present state. You can then use special actions to undo, redo, or reset the present state.

[12] https://github.com/omnidan/redux-undo

Using a higher-order reducer is as simple as passing your reducer into an imported function:

**Using a higher-order reducer**

```
import { combineReducers } from 'redux';
import recipesReducer from 'reducers/recipes';
import ingredientsReducer from 'reducers/ingredients';
import undoable from 'redux-undo';

const rootReducer = combineReducers({
  recipes: undoable(recipesReducer),
  ingredients: ingredientsReducer
});
```

Another example of a reducer enhancer is `redux-ignore`[13]. This library allows your reducers to immediately return the current state without handling the passed action, or to handle only a defined subset of actions. The following example will disable removing recipes from our recipe book. You might even use it to filter allowed actions based on user roles:

**Using the ignoreActions() higher-order reducer**

```
import { combineReducers } from 'redux';
import recipesReducer from 'reducers/recipes';
import ingredientsReducer from 'reducers/ingredients';
import { ignoreActions } from 'redux-ignore';
import { REMOVE_RECIPE } from 'constants/actionTypes';

const rootReducer = combineReducers({
  recipes: ignoreActions(recipesReducer, [REMOVE_RECIPE]),
  ingredients: ingredientsReducer
});
```

# Summary

In this chapter we learned about the part of Redux responsible for changing the application state. Reducers are meant to be pure functions that should never mutate the state or make any asynchronous calls. We also learned how to avoid and catch mutations in JavaScript.

In the next and final chapter in this part of the book we are going to talk about *middleware*, the most powerful entity provided by Redux. When used wisely, middleware can significantly reduce the size of our code and let us handle very complicated scenarios with ease.

[13] https://github.com/omnidan/redux-ignore