Contents

Introduction

Did you think Redux was complicated? The basic implementation of Redux is actually super simple and I will prove it to you in this book.

We will start by coding the createStore function which is the core of Redux. After that we will code the combineReducers function. Next step is the function that binds Redux and React together: the connect() function. Last but not least we will look into one of the most popular middlewares in Redux: the Redux Thunk Middleware.

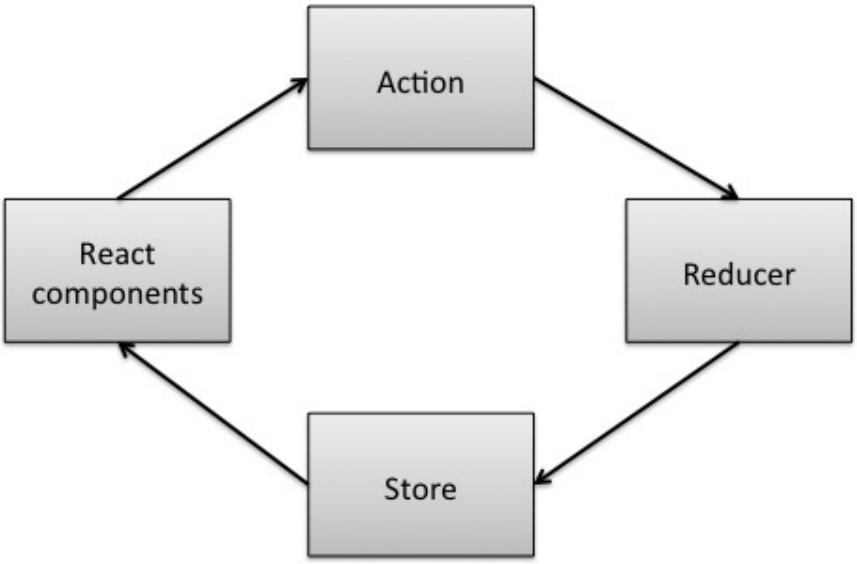
The code we are going to write will not be production ready code. We will strip away all error handling, optimizations, etc. The code we will produce is for educational purposes only.

Redux

What does Redux do?

The whole point with Redux is to have one single source of truth for your application state. The state is stored as a plain Javascript object in one place: the Redux Store. The state object is read only. If you want to change the state, you need to emit an Action, which is a plain JavaScript object.

Your application can subscribe to get notified when the store has changed. When Redux is used with React, it is the React components that get notified when state changes, and can re-render based on new content in the store.



The store needs a way to know how to update the state in the store when it gets an Action. It uses a plain JavaScript function for this that Redux calls a reducer. The reducer function is passed in when the store is created.

Let's get started coding

To summarize, we need to be able to do three things with our store:

1. Get the current state of the store
2. Dispatch an action, which is passed as an argument to the reducer to update the state in the store.
3. Listen to when the store changes

We also need to define the reducer and the initial state at startup time. Let's start with this:

```
function createStore(reducer, initialState) {
    var currentReducer = reducer;
    var currentState = initialState;
}
```

1. Get state

Ok, so we have created a function that just saves the initial state and the reducer as local variables. Now let's implement the possibility to get the state of the store.

```
function createStore(reducer, initialState) {
    var currentReducer = reducer;
    var currentState = initialState;

    return {
        getState() {
            return currentState;
        }
    };
}
```

We can now get the state object with getState() ! That was easy.

2. Dispatch an action

Next step is to implement support for dispatching an action.

```
function createStore(reducer, initialState) {
    var currentReducer = reducer;
    var currentState = initialState;

    return {
        getState() {
            return currentState;
        },
        dispatch(action) {
            currentState = currentReducer(currentState, action);
            return action;
        }
    };
}
```

The dispatch function passes the current state and the dispatched Action through the reducer that we defined at init. It then overwrites the old state with the new state.

3. Subscribe for changes

Now we can both get current state and update the state! The last step is to be able to listen to changes:

```
function createStore(reducer, initialState) {
    var currentReducer = reducer;
    var currentState = initialState;
    var listener = () => {};

    return {
        getState() {
            return currentState;
        },
        dispatch(action) {
            currentState = currentReducer(currentState, action);
            listener(); // Note that we added this line!
            return action;
        },
        subscribe(newListener) {
            listener = newListener;
        }
    };
}
```

Now we can call subscribe with a callback function as parameter that will be called whenever an action is dispatched.

We are done with our implementation of the Redux core.

Let's use it!

On the official Redux Github page, there is an example on how to use Redux. We can copy/paste that example to test our own Redux implementation:

```
function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1
  case 'DECREMENT':
    return state - 1
  default:
```

```
    return state
  }
}

let store = createStore(counter)

store.subscribe(() =>
  console.log(store.getState())
)

store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'DECREMENT' })
```

It works perfectly!

Code your own combineReducers

What problem does combineReducer solve?

In the example usage of Redux above, we use a reducer called counter. A reducer function is basically a long switch.

When the application grows with more and more actions, the reducer function grows with it. We want to avoid large functions because they are difficult to overview, extend and combine.

combineReducer lets us split the reducer up to smaller reducers and combine them so we can use them in Redux.

Let's code our own combineReducer!

We will start by copying the example usage of combineReducer from the official docs:

```
// reducers.js
export default theDefaultReducer = (state = 0, action) => state;

export const firstNamedReducer = (state = 1, action) => state;

export const secondNamedReducer = (state = 2, action) => state;

// Use ES6 object literal shorthand syntax to define the object shape
const rootReducer = combineReducers({
  theDefaultReducer,
  firstNamedReducer,
```

```
  secondNamedReducer
});

const store = createStore(rootReducer);
console.log(store.getState());
// {theDefaultReducer : 0, firstNamedReducer : 1, secondNamedReducer : 2}
```

We are going to use this as a starting point for our own implementation.

Function definition

The combineReducer function call takes an object with reducers. Let's start by implementing support for that.

```
function combineReducers(reducers) {
}
```

We now have the function combineReducers that accepts reducers as input parameters. Sweet!

Return a new empty reducer

Look at the example again. combineReducer returns a rootReducer that is used as an argument to createStore. In other words, the rootReducer works just like a normal Redux reducer.

As we learned previously, a normal Redux reducer is a function that takes two parameters: state and action.

That means our combineReducers function is going to return another function.

```
function combineReducers(reducers) {
  return function combination(state = {}, action) {
  }
}
```

A function that returns another function? If you are new to JavaScript this might look a bit weird. In Javascript, functions are "first class citizens". That means you can pass functions as parameters to other functions, assign them to variables, and even return functions from other functions. This can be a bit tricky to wrap your head around. I suggest spending a few minutes playing around with that code until you are confident in what it does.

Return a reducer that creates a new state

In the example, there is a console.log of the state of the store. We can see the expected output in the comments. The output after running getState should be an object with all the reducer names as keys and the state as the value.

Let's start by returning an object with the reducer names as keys. To simplify we will have a hardcoded string as values.

```
function combineReducers(reducers) {
  // First get an array with all the keys of the reducers (the reducer names)
  const reducerKeys = Object.keys(reducers);

  return function combination(state = {}, action) {
    // This is the object we are going to return.
    const nextState = {}

    // Loop through all the reducer keys
    for (let i = 0; i < reducerKeys.length; i++) {
      // Get the current key name
      const key = reducerKeys[i];
      nextState[key] = "Here is where the state will be";
    }
    return nextState;
  }
}
```

When we run this code we will get this output:

```
// {theDefaultReducer : "Here is where the state will be", firstNamedReducer : "Here is whe
```

We are almost there! The final thing we need to do is to print the object with the correct values.

Return a reducer that returns correct state

Let's finish this up by adding the correct state of each sub reducer as value to the printed object. To do that we will call each sub reducer.

```
function combineReducers(reducers) {
  // First get an array with all the keys of the reducers (the reducer names)
  const reducerKeys = Object.keys(reducers);

  return function combination(state = {}, action) {
    // This is the object we are going to return.
    const nextState = {}

    // Loop through all the reducer keys
    for (let i = 0; i < reducerKeys.length; i++) {
    // Get the current key name
    const key = reducerKeys[i];
    // Get the current reducer
    const reducer = reducers[key]
    // Get the the previous state
```

```
    const previousStateForKey = state[key]
    // Get the next state by running the reducer
    const nextStateForKey = reducer(previousStateForKey, action)
    // Update the new state for the current reducer
    nextState[key] = nextStateForKey;
    }
    return nextState;
  }
}
```

Now we are done! We have written the whole combineReducers function. Not that complicated, was it?

Connecting with React

Redux is a stand alone library that you don't have to use together with React. But if you do you can use the officially supported library react-redux to make it easier. This library is what we are going to base our own implementation on.

React redux

I will give a super quick step-by-step tutorial (only 2 steps!) of to use react-redux so we know how it works before we will implement it ourselves!

1. Make your store accessible from React

First, we will wrap the root component of our app in a react-redux component which takes our store as props. This wrapping component will then make the store available for all React components.

```
import { Provider } from 'react-redux';
const store = createStore(myReducer);
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

2. Make your component access Redux data

Let's say you have a simple component that you want to be able to receive data from redux.

```
let AddTodo = ({ todos }) => {
  // Some fancy implementation (not relevant for this example)
}
```

react-redux has a function called connect() that you can use to connect your Redux store to your React components.

```
const mapStateToProps = (state) => {
  return {
    todos: state.todos
  }
}
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
AddTodo = connect(mapStateToProps, mapDispatchToProps)(AddTodo)
```

The connect function automatically takes the data from the store, and passes it down as props to the connected component. When the data in the store changes, the passed down props changes, and the component is automatically re-rendered. Cool!

If you need a more detailed introduction to this library I highly suggest reading the docs about it here before moving on.

Code our own connect()

Now that we know how to use react-redux, let's implement it!

A look at the input parameters

The connect function returns a new function that is then immediately called. We now have enough knowledge to implement the skeleton for the connect function:

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
      //something happens here
  }
}
```

Component in – Component out

If we take a look at how the connect function is used, we can see that it is a function that takes our React component as input, and outputs a new React component. Let's add that to our implementation.

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
    //we return a Wrapper component:
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
          />
        )
      }
    }
  }
}
```

So now our function returns the exact component we sent as input (here named WrappedComponent) with an empty wrapper component around it.

This code is not very useful because we don't do anything with the component. Let's fix that! In the next step we will enhance the component with data from our store.

Adding props to the returned component

We want to pass down props containing data from the store to our newly created component. We will do it like this:

1. Get the state from the store with store.getState()
2. Call the function mapStateToProps that has been passed in
3. Set the returned data as props in the newly created component

It looks like this:

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
            {...mapStateToProps(store.getState(), this.props)}
          />
```

```
        )
      }
    }
  }
}
```

So we have a function which takes a React component as input and returns a new enhanced React component. This actually has a fancy name: Higher order component (HOC). It's a pattern that is commonly used in advanced React applications and React libraries because it allows you to elegantly reuse logic. And that is precisely what we have done here.

Add Action methods to the returned component

We also want our React component to be able to call actions with the dispatcher. We pass down the actions from the mapDispatchToProps function as props to our components in a similar way that we did in the previous step.

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
            {...mapStateToProps(store.getState(), this.props)}
            {...mapDispatchToProps(store.dispatch, this.props)}
          />
        )
      }
    }
  }
}
```

But wait, where did we get the store from?

Remember in the beginning of this chapter where I showed how to use react-redux? I said that you need to wrap the application root component in a Provider component which injects the store to all your components. That is how we get the store here! How this is implemented is out of scope for this book. In the meantime I will give you a hint. It is using react context.

Subscribing to the store

Now we have a way to map the data from the store to props to our components. We can also dispatch actions from our component.

We are just missing one thing right now. We need to make sure we don't miss any updates from our store by subscribing to it.

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
            {...mapStateToProps(store.getState(), this.props)}
            {...mapDispatchToProps(store.dispatch, this.props)}
          />
        )
      }
      componentDidMount() {
        this.unsubscribe = store.subscribe(this.handleChange.bind(this))
      }

      componentWillUnmount() {
        this.unsubscribe()
      }

      handleChange() {
        this.forceUpdate()
      }
    }
  }
}
```

Now we have subscribed for changes in our store with a callback function handleChange. Our callback function calls the React function forceUpdate which, not very surprisingly, forces a re-render of our component!

Redux thunk middleware

Redux is a minimalistic library that is possible to extend with middlewares. One of the most commonly used middlewares is Redux Thunk Middleware. It is used to create async actions.

The code for Redux Thunk is on Github and the whole implementation looks like this:

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}

const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;
```

Yes, that is the whole Redux Thunk middleware. 11 lines of code. Did you expect something bigger?

So what does the code do?

Main part

Most of the Redux Thunk code is actually boilerplate code for connecting to Redux as a middleware. The main part of the code is the three lines inside the if statement:

```
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }
```

When Redux Thunk is configured for your application, this code above will be run every time an action is dispatched from anywhere in the application. What the code does is that it checks if the action is a function and if it is, it calls that function with dispatch as the argument. It also passes in getState and extraArgument as arguments, but let's not care about that for now.

To summarize:

- When using Redux without middlewares, an action is always a plain Javascript object.
- When using Redux with Thunk middleware, an action can either be a plain Javascript object, OR an action can be a function.

Let's look at an example of how we can use it.

Usage of Redux Thunk

One example of an action creator that utilizes Redux Thunk could look like this:

```
function loadPostsAction() {
    return (dispatch, getState) => {
        // Do something here.
    };
}
```

A function is returned in which we have access to the Redux dispatch function. That function is the exact same dispatch function we use from other places in our Redux app to dispatch actions.

Now we can call this function how many times we want, whenever we want in our action creator. So if we work with Ajax, we might want to dispatch an action on success, and another action on failure.

We can just call dispatch in those two cases like this:

```
function loadPostsAction() {
    return (dispatch, getState) => {
        get("/api/posts").then(
            (payload) => dispatch({ type: "LOAD_POSTS_FULFILLED", payload }),
            (err) => dispatch({ type: "LOAD_POSTS_REJECTED" })
        );
    };
}
```

This time we call the dispatch function with a regular Javascript object as action like we usually do.