**KSS - React Redux**

First install **create-react-app** <https://github.com/facebook/create-react-app>

*> npm install -g create-react-app*

> create-react-app react-redux-kss

To understand what is Redux we must first understand what is the state.

A stateful React component is a Javascript ES6 class. [<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>]

In a React component the state holds up data and the component might render such data.

The state could also change in response to actions and events: to update the local component’s state use setState.

## **what problem does Redux solve?**

A typical JavaScript application is full of states.

State is everywhere in JavaScript.

As you can see even the simplest JavaScript application has a state.

Here are some examples of state:

* what the user sees (data)
* what data are we fetching
* what URL are we showing to the user
* what items are selected inside the page
* are there errors in the applications? That’s state too

As long as the application remains small, we can get by with keeping the state within a parent React component.

Then things will become tricky, when application became complex.

Also **frontend shouldn’t know about the business logic**. Ever.

So one of the alternatives for managing the state of a React component: **Redux**

**Redux solves a problem that might not be clear in the beginning**: it helps giving **each React component** the **exact piece of state** it needs.

Redux holds up the **state** within a **single location**.

Also with Redux the **logic for fetching and managing the state** lives **outside React**.

## **should I learn Redux?**

* Redux is **framework agnostic**. Learn it once, use it everywhere (Vue JS, Angular)

Redux is just a library among the others (like flux, mobx etc), which might of-course disappear in future :) But the **patterns will stick forever**.

## **should I use Redux?**

**consider using Redux** when:

* multiple React components needs to access the same state but do not have any parent/child relationship
* you start to feel awkward passing down the state to multiple components with props

Be aware that Redux is not useful for smaller apps. It really shines in bigger ones.

Dan Abramov says “*Flux libraries are like glasses: you’ll know when you need them.*”

Dan Abramov => Working on @reactjs. Co-author of Redux and Create React App.

You Might Not Need Redux [<https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367>]

## **getting to know the Redux store**

The store in Redux is like the human brain: it’s kind of magic.

The **Redux store is fundamental**: the **state of the whole application** lives **inside the store**.

So to start playing with Redux we should **create a store for wrapping up the state**.

*> yarn add redux*

Create a directory for the store:

*> mkdir -p src/store*

Create a new file named index.js in src/store and finally initialize the store:

1. import { createStore } from "redux";
2. import reducer from "./reducer";
3. const store = createStore(reducer);
4. export default store;

createStore is the function for creating the Redux store. [<https://redux.js.org/api-reference/createstore>]

createStore takes a reducer as the first argument, [rootReducer in our case, which we will create in a while].

We may also pass an initial state to createStore. But most of the times we don’t have to. Passing an initial state is useful for server side rendering. Anyway, **the state comes from reducers**.

**Problem***: Reducer returned undefined during initialization [*[*https://stackoverflow.com/questions/36619093/why-do-i-get-reducer-returned-undefined-during-initialization-despite-pr*](https://stackoverflow.com/questions/36619093/why-do-i-get-reducer-returned-undefined-during-initialization-despite-pr) *]*

*In Redux* ***reducers produce the state****.*

## **getting to know Redux reducers**

**A reducer is just a Javascript function**. A reducer **takes two parameters**: **the current state** and an **action** (more about actions soon).

Third Principle of Redux says that the state is immutable and cannot change in place.

Note: Three principle of Redux [<https://redux.js.org/introduction/three-principles>]

This is why the reducer must be pure. [*A pure function is one that returns the exact same output for the given input.*]

In plain React the local state changes in place with setState. In Redux you cannot do that.

Lets create a file inside *src/store* directory called *reducer.js*

1. const initialState = {
2. articles: []
3. };
4. const reducer = (state = initialState, action) => state;
5. export default reducer;

In our example we created a **simple reducer taking the initial state** as the first parameter. As a **second parameter** we had provided **action**. As of now the reducer will do nothing than returning the initial state.

## **getting to know Redux actions**

Redux reducers are without doubt the most important concept in Redux. **Reducers produce the state of the application**.

But **how does a reducer know when to produce the next state**?

The second principle of Redux says the **only way to change the state is by sending a signal to the store**. This signal is an **action**. “**Dispatching an action**” is the process of sending out a signal.

**Redux actions are nothing more than Javascript objects**. This is what an action looks like:

1. {
2. type: 'ADD\_ARTICLE',
3. payload: { name: 'React Redux KSS, id: 1 }
4. }

Every action needs a type property for describing how the state should change.

We can specify a payload as well. In the above example the payload is a new article. A reducer will add the article to the current state later.

It is a best practice to **wrap every action within a function**. Such function is an **action creator**.

Let’s put everything together by creating a simple Redux action.

1. // src/store/actions.js
2. export const addArticle = article => ({ type: "ADD\_ARTICLE", payload: article });

So, the **type property** is nothing more than a string.

The reducer will use that string to determine how to calculate the next state.

Since strings are prone to typos and duplicates it’s **better to have action types declared as constants**.

This approach helps **avoiding errors that will be difficult to debug**.

1. // src/store/types.js
2. export const ADD\_ARTICLE = "ADD\_ARTICLE";

Now, update the action to use action types:

1. // src/store/actions.js
2. import { ADD\_ARTICLE } from "./types";
3. export const addArticle = article => ({ type: ADD\_ARTICLE, payload: article });

## **refactoring the reducer**

**reducer calculates the next state depending on the action type**. Moreover, **it should return at least the initial state when no action type matches**.

When the action type matches a case clause the **reducer calculates the next state** and **returns a new object**.

1. import { ADD\_ARTICLE } from "./types";
2. const initialState = {
3. articles: []
4. };
5. const reducer = (state = initialState, action) => {
6. **switch** (action.type) {
7. **case** ADD\_ARTICLE:
8. state.articles.push(action.payload);
9. **return** state;
10. default:
11. **return** state;
12. }
13. };
14. export default reducer;

Although it’s valid code the **above reducer breaks** the main Redux principle: **immutability**.

[Array.prototype.push](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push) is an impure function: it alters the original array.

Making our reducer compliant is easy. Using [Array.prototype.concat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat) in place of Array.prototype.push is enough to keep the initial array immutable:

1. import { ADD\_ARTICLE } from "./types";
2. const initialState = {
3. articles: []
4. };
5. const reducer = (state = initialState, action) => {
6. **switch** (action.type) {
7. **case** ADD\_ARTICLE:
8. **return** { ...state, articles: state.articles.concat(action.payload) };
9. default:
10. **return** state;
11. }
12. };
13. export default reducer;

With the [spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) we can make our reducer even better:

**return** { ...state, articles: [...state.articles, action.payload] };

for **avoiding mutations in Redux**:

* [Using concat(), slice(), and …spread](https://egghead.io/lessons/react-redux-avoiding-array-mutations-with-concat-slice-and-spread) for arrays
* [Using Object.assign() and …spread](https://egghead.io/lessons/react-redux-avoiding-object-mutations-with-object-assign-and-spread) for objects

## **connecting React with Redux**

Redux on its own is framework agnostic. We can use it with vanilla Javascript. Or with Angular. Or with React. Or with VueJS. There are bindings for joining together Redux with your favorite framework/library or we can use redux’s own functionalities.

For React there is **react-redux**, a bindings of reactJs and Redux. It’s a small library for connecting Redux and React in an efficient way.

*> yarn add react-redux*

## **react-redux**

The most important method we’ll work with react-redux is ***connect()***

What does react-redux’s **connect** do? Unsurprisingly it connects a React component with the Redux store.

Other fundamental things to know are:

* the ***mapStateToProps*** function
* the ***mapDispatchToProps*** function

**What does mapStateToProps do** in react-redux?

mapStateToProps does exactly what its name suggests: it **connects a part of the Redux state** to the props of a React component. By doing so a connected React component will have access to the exact part of the store it needs. (So read it as ***mapStoreToProps***)

**What does mapDispatchToProps** do in react-redux?

mapDispatchToProps does something similar, but for actions. **mapDispatchToProps connects Redux actions to React props**. This way a connected React component will be able to dispatch actions.

**Example:**

## **App component and Redux store**

We saw that mapStateToProps connects a portion of the Redux state to the props of a React component. You may wonder: is this enough for connecting Redux with React? No, it’s not.

To start off **connecting Redux with React we’re going to use Provider**.

**Provider** is an high order component coming from react-redux.

Using layman’s terms, Provider wraps up your React application and makes it aware of the entire Redux’s store.

Open up src/index.js, wipe out everything and update the file with the following code:

1. import React from "react";
2. import { render } from "react-dom";
3. import { Provider } from "react-redux";
4. import store from "./store/index";
5. import App from "./components/App";
6. render(
7. <Provider store={store}>
8. <App />
9. </Provider>,
10. document.getElementById("root")
11. );

Provider wraps up your entire React application. Moreover it gets the store as a prop.

Now, create the **App** component that import a List component and render itself.

1. // src/components/App.js
2. import React from "react";
3. import List from "./List";
4. const App = () => (
5. <div className="row mt-5">
6. <div className="col-md-4 offset-md-1">
7. <h2>Articles</h2>
8. <List />
9. </div>
10. </div>
11. );
12. export default App;

## **List component and Redux state**

our new component, List, will interact with the Redux store.

1. // src/components/List.js
2. import React from "react";
3. import { connect } from "react-redux";
4. const mapStateToProps = state => {
5. **return** { articles: state.articles };
6. };
7. const ConnectedList = ({ articles }) => (
8. <ul className="list-group list-group-flush">
9. {articles.map(article => (
10. <li className="list-group-item" key={article.id}>
11. {article.title}
12. </li>
13. ))}
14. </ul>
15. );
16. const List = connect(mapStateToProps)(ConnectedList);
17. export default List;

## **Form component and Redux actions**

The Form component we’re going to create a **stateful component**.

*A stateful component in React is a component carrying its own local state*

Create a new file named Form.js inside src/components. It should look like the following:

1. // src/components/Form.js
2. import React, { Component } from "react";
3. import { connect } from "react-redux";
4. import uuidv1 from "uuid";
5. import { addArticle } from "../store/actions";
6. const mapDispatchToProps = dispatch => {
7. **return** {
8. addArticle: article => dispatch(addArticle(article))
9. };
10. };
11. class ConnectedForm extends Component {
12. constructor() {
13. super();
14. **this**.state = {
15. title: ""
16. };
17. **this**.handleChange = **this**.handleChange.bind(**this**);
18. **this**.handleSubmit = **this**.handleSubmit.bind(**this**);
19. }
20. handleChange(event) {
21. **this**.setState({ [event.target.id]: event.target.value });
22. }
23. handleSubmit(event) {
24. event.preventDefault();
25. const { title } = **this**.state;
26. const id = uuidv1();
27. **this**.props.addArticle({ title, id });
28. **this**.setState({ title: "" });
29. }
30. render() {
31. const { title } = **this**.state;
32. **return** (
33. <form onSubmit={**this**.handleSubmit}>
34. <div className="form-group">
35. <label htmlFor="title">Title</label>
36. <input
37. type="text"
38. className="form-control"
39. id="title"
40. value={title}
41. onChange={**this**.handleChange}
42. />
43. </div>
44. <button type="submit" className="btn btn-success btn-lg">
45. SAVE
46. </button>
47. </form>
48. );
49. }
50. }
51. const Form = connect(null, mapDispatchToProps)(ConnectedForm);
52. export default Form;

Update App to include the Form component:

1. import React from "react";
2. import List from "./List";
3. import Form from "./Form";
4. const App = () => (
5. <div className="row mt-5">
6. <div className="col-md-4 offset-md-1">
7. <h2>Articles</h2>
8. <List />
9. </div>
10. <div className="col-md-4 offset-md-1">
11. <h2>Add a new article</h2>
12. <Form />
13. </div>
14. </div>
15. );
16. export default App;

## **wrapping up**

**Redux protip**: the reducer will grow as our app will become bigger. We can split a big reducer into separate functions and combine them with **combineReducers** [<https://redux.js.org/docs/api/combineReducers.html>]

Also, we are currently following “**re-ducks**” structure to manage store in larger application (<https://github.com/alexnm/re-ducks>)

Scaling your Redux App with ducks

<https://medium.freecodecamp.org/scaling-your-redux-app-with-ducks-6115955638be#.4ppptx7oq>

We have everything explained in Redux’s documentation

* <https://redux.js.org/>