**What were the major problems with MVC framework?**

Following are some of the major problems with MVC framework:

* DOM manipulation was very expensive
* Applications were slow and inefficient
* There was huge memory wastage
* Because of circular dependencies, a complicated model was created around models and views

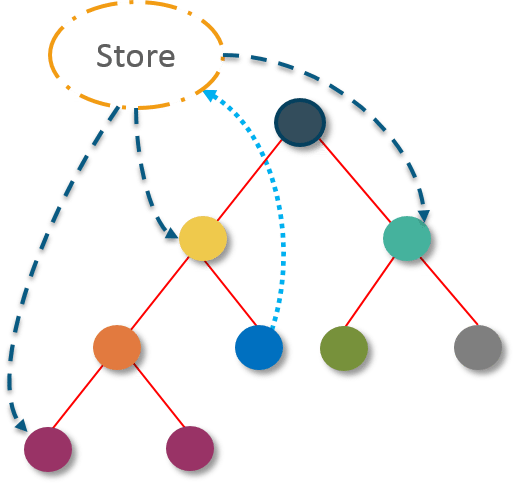
**Explain Flux.**

Flux is an architectural pattern which enforces the uni-directional data flow. It controls derived data and enables communication between multiple components using a central Store which has authority for all data. Any update in data throughout the application must occur here only. Flux provides stability to the application and reduces run-time errors.

**What is Redux?**

Redux is one of the hottest libraries for front-end development in today’s marketplace. It is a predictable state container for JavaScript applications and is used for the entire applications state management. Applications developed with Redux are easy to test and can run in different environments showing consistent behavior.

**What are the three principles that Redux follows?**

1. ***Single source of truth:***The state of the entire application is stored in an object/ state tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
2. ***State is read-only:***The only way to change the state is to trigger an action. An action is a plain JS object describing the change. Just like state is the minimal representation of data, the action is the minimal representation of the change to that data.
3. ***Changes are made with pure functions:*** In order to specify how the state tree is transformed by actions, you need pure functions. Pure functions are those whose return value depends solely on the values of their arguments.

**What do you understand by “Single source of truth”?**

Redux uses ‘Store’ for storing the application’s entire state at one place. So all the component’s state are stored in the Store and they receive updates from the Store itself. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.

**List down the components of Redux.**

Redux is composed of the following components:

1. **Action** – It’s an object that describes what happened.
2. **Reducer**–  It is a place to determine how the state will change.
3. **Store** – State/ Object tree of the entire application is saved in the Store.
4. **View** – Simply displays the data provided by the Store.

**Show how the data flows through Redux?**



## React Redux tutorial: getting to know the Redux store

The **store orchestrates all the moving parts in Redux**. Repeat with me: **the 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**.

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

createStore is the function for creating the Redux store.

createStore takes a reducer as the first argument, rootReducer in our case.

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

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

What matters now is understanding what does a reducer do.

In Redux **reducers produce the state**. The state is not something you create by hand.

Armed with that knowledge let’s move on to our first Redux reducer.

## React Redux tutorial: getting to know Redux reducers

While an initial state is useful for [SSR](https://redux.js.org/docs/recipes/ServerRendering.html), in Redux **the state must return entirely from reducers**.

Cool but what’s a reducer?



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

The third principle of Redux says that the state is immutable and cannot change in place.

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.

Creating a reducer is not that hard. It’s a plain Javascript function with two parameters.

In our example we’ll be creating a **simple reducer taking the initial state** as the first parameter. As a **second parameter** we’ll provide **action**. As of now the reducer will do nothing than returning the initial state.

Create a directory for the root reducer:

1. mkdir -p src/js/reducers

Then create a new file named index.jsin the src/js/reducers:

1. // src/js/reducers/index.js
2. const initialState = {
3. articles: []
4. };
5. const rootReducer = (state = initialState, action) => state;
6. export default rootReducer;

I promised to keep this guide as simple as possibile. That’s why our first reducer is a silly one: it returns the initial state without doing anything else.

Notice how the initial state is passed as a [default parameter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters).

In the next section we’ll add an action to the mix. That’s where things will become interesting.

## React Redux tutorial: 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.

Now, how do you change an immutable state? You won’t. The resulting state is a copy of the current state plus the new data.

That’s a lot to know.

The reassuring thing is that **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 Tutorial', id: 1 }
4. }

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

You 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 pratice to **wrap every action within a function**. Such function is an **action creator**.

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

Create a directory for the actions:

1. mkdir -p src/js/actions

Then create a new file named index.jsin src/js/actions:

1. // src/js/actions/index.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**.

Create a new directory:

1. mkdir -p src/js/constants

Then create a new file named action-types.jsinto the src/js/constants:

1. // src/js/constants/action-types.js
2. export const ADD\_ARTICLE = "ADD\_ARTICLE";

Now open up again src/js/actions/index.jsand update the action to use action types:

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

We’re one step closer to have a working Redux application. Let’s refactor our reducer!

## React Redux tutorial: refactoring the reducer

Before moving forward let’s recap the main Redux concepts:

* the **Redux store** is like a brain: it’s in charge for **orchestrating all the moving parts** in Redux
* the **state of the application lives as a single, immutable object** within the store
* as soon as **the store receives an action it triggers a reducer**
* the **reducer returns the next state**

What’s a **Redux reducer** made of?

A reducer is a Javascript function taking **two parameters**: the **state** and the **action**.

A reducer function has a **switch statement** (although unwieldy, a naive reducer could also use if/else).

The **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**. Here’s an excerpt of the code:

1. // ...
2. **switch** (action.type) {
3. **case** ADD\_ARTICLE:
4. **return** { ...state, articles: [...state.articles, action.payload] };
5. default:
6. **return** state;
7. }
8. // ...

The reducer we created in the previous section does nothing than returning the initial state. Let’s fix that.

Open up src/js/reducers/index.jsand update the reducer as follow:

1. import { ADD\_ARTICLE } from "../constants/action-types";
2. const initialState = {
3. articles: []
4. };
5. const rootReducer = (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 rootReducer;

What do you see here?

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" \t "_blank) in place of Array.prototype.push is enough to keep the initial array immutable:

1. import { ADD\_ARTICLE } from "../constants/action-types";
2. const initialState = {
3. articles: []
4. };
5. const rootReducer = (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 rootReducer;

We’re not done yet! With the [spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) we can make our reducer even better:

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

In the example above the initial state is left utterly untouched.

The initial articles array doesn’t change in place.

The initial state object doesn’t change as well. The resulting state is a copy of the initial state.

There are two key points 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

The **object spread operator** is still in stage 3. Install [Object rest spread transform](https://babeljs.io/docs/plugins/transform-object-rest-spread/) to **avoid a SyntaxError Unexpected token** when using the object spread operator in Babel:

1. npm i --save-dev babel-plugin-transform-object-rest-spread

Open up .babelrcand update the configuration:

1. {
2. **"presets"**: ["env", "react"],
3. **"plugins"**: ["transform-object-rest-spread"]
4. }

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

In the next section we’ll play with Redux from the console. Hold tight!

## React Redux tutorial: Redux store methods

This will be super quick, I promise.

I want you to play with the brower’s console for gaining a quick understanding of how Redux works.

Redux itself is a small library (2KB). The [Redux store exposes a simple API](https://redux.js.org/docs/api/Store.html" \t "_blank) for managing the state. The most important methods are:

* [getState](https://redux.js.org/docs/api/Store.html#getState) for accessing the current state of the application
* [dispatch](https://redux.js.org/docs/api/Store.html#dispatch) for dispatching an action
* [subscribe](https://redux.js.org/docs/api/Store.html#subscribe) for listening on state changes

We will play in the brower’s console with the above methods.

To do so we have to export as global variables the store and the action we created earlier.

Create src/js/index.jsand update the file with the following code:

1. import store from "../js/store/index";
2. import { addArticle } from "../js/actions/index";
3. window.store = store;
4. window.addArticle = addArticle;

Open up src/index.jsas well, clean up its content and update it as follows:

1. import index from "./js/index"

Now run webpack dev server (or Parcel) with:

1. npm start

head over http://localhost:8080/ and open up the console with F12.

Since we’ve exported the store as a global variable we can access its methods. Give it a try!

Start off by **accessing the current state**:

1. store.getState()

output:

1. {articles: Array(0)}

Zero articles. In fact we haven’t update the initial state yet.

To make things interesting we can listen for state updates with [subscribe](https://redux.js.org/docs/api/Store.html#subscribe).

The **subscribe method accepts a callback that will fire whenever an action is dispatched**. Dispatching an action means notifying the store that we want to change the state.

Register the callback with:

1. store.subscribe(() => console.log('Look ma, Redux!!'))

To **change the state in Redux we need to dispatch an action**. To dispatch an action you have to call the [dispatch](https://redux.js.org/docs/api/Store.html#dispatch) method.

We have one action at our disposal: addArticle for adding a new item to the state.

Let’s dispatch the action with:

1. store.dispatch( addArticle({ name: 'React Redux Tutorial for Beginners', id: 1 }) )

Right after running the above code you should see:

1. Look ma, Redux!!

To verify that the state changed run again:

1. store.getState()

The output should be:

1. {articles: Array(1)}

And that’s it. This is Redux in its simplest form.

Was that difficult?

Take your time to explore these three Redux methods as an exercise. Play with them from the console:

* [getState](https://redux.js.org/docs/api/Store.html#getState) for **accessing the current state** of the application
* [dispatch](https://redux.js.org/docs/api/Store.html#dispatch) for **dispatching an action**
* [subscribe](https://redux.js.org/docs/api/Store.html#subscribe) for **listening on state changes**

That’s everything you need to know for getting started with Redux.

Once you feel confident head over the next section. We’ll go straight to connecting React with Redux!

## React Redux tutorial: connecting React with Redux

After learning Redux I realized it wasn’t so complex.

I knew how to access the current state with [getState](https://redux.js.org/docs/api/Store.html" \l "getState" \t "_blank).

I knew how to dispatch an action with [dispatch](https://redux.js.org/docs/api/Store.html#dispatch)

I knew how to listen for state changes with [subscribe](https://redux.js.org/docs/api/Store.html#subscribe)

Yet I didn’t know how to couple **React and Redux** together.

I was asking myself: should I call getState within a React component? How do I dispatch an action from a React component? And so on.

Redux on its own is framework agnostic. You can use it with vanilla Javascript. Or with Angular. Or with React. There are bindings for joining together Redux with your favorite framework/library.

For React there is [react-redux](https://redux.js.org/docs/basics/UsageWithReact.html).

Before moving forward install [react-redux](https://redux.js.org/docs/basics/UsageWithReact.html) by running:

1. npm i react-redux --save-dev

To demonstrate how React and Redux work together we’ll build a super simple application. The application is made of the following components:

* an App component
* a List component for displaying articles
* a Form component for adding new articles

(The application is a toy and it does nothing serious other than displaying a list and a form for adding new items. Nonetheless it’s still a good starting point for learning Redux)

## React Redux tutorial: react-redux

[react-redux](https://redux.js.org/docs/basics/UsageWithReact.html) is a Redux binding for React. It’s a small library for connecting Redux and React in an efficient way.

The most important method you’ll work with is [connect](https://github.com/reactjs/react-redux/blob/master/docs/api.md#connectmapstatetoprops-mapdispatchtoprops-mergeprops-options)

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

You will use connect with two or three arguments depending on the use case. The 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](https://reactjs.org/docs/components-and-props.html). By doing so a connected React component will have access to the exact part of the store it needs.

**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.

Is everything clear? If not, stop and take your time to re-read the guide. I know it’s a lot to learn and it requires time. Don’t worry if you don’t get Redux right know. It will click sooner or later.

In the next section we’ll finally get our hands dirty!

## React Redux tutorial: 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**](https://github.com/reactjs/react-redux/blob/master/docs/api.md#provider-store).

[Provider](https://github.com/reactjs/react-redux/blob/master/docs/api.md#provider-store) 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.

Why so? We saw that in Redux the store manages everything. React must talk to the store for accessing the state and dispatching actions.

Enough theory.

Open up src/js/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("app")
11. );

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

Now let’s create the **App** component since we’re requiring it. It’s nothing special: App should import a List component and render itself.

Create a directory for holding the components:

1. mkdir -p src/js/components

and a new file named App.jsinside src/js/components:

1. // src/js/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;

Take moment and look at the component without the markup:

1. import React from "react";
2. import List from "./List";
3. const App = () => (
4. <List />
5. );
6. export default App;

then move on to creating **List**.

## React Redux tutorial: List component and Redux state

We have done nothing special so far.

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

A brief recap: the key for connecting a React component with Redux is [connect](https://github.com/reactjs/react-redux/blob/master/docs/api.md#connectmapstatetoprops-mapdispatchtoprops-mergeprops-options).

Connect takes at least one argument.

Since we want List to get a list of articles it’s a matter of connecting state.articleswith the component. How? With **mapStateToProps**.

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

1. // src/js/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(el => (
10. <li className="list-group-item" key={el.id}>
11. {el.title}
12. </li>
13. ))}
14. </ul>
15. );
16. const List = connect(mapStateToProps)(ConnectedList);
17. export default List;

The List component receives the prop articleswhich is a copy of the articlesarray. Such array lives inside the Redux state we created earlier. It comes from the reducer:

1. const initialState = {
2. articles: []
3. };
4. const rootReducer = (state = initialState, action) => {
5. **switch** (action.type) {
6. **case** ADD\_ARTICLE:
7. **return** { ...state, articles: [...state.articles, action.payload] };
8. default:
9. **return** state;
10. }
11. };

Then it’s a matter of using the prop inside JSX for generating a list of articles:

1. {articles.map(el => (
2. <li className="list-group-item" key={el.id}>
3. {el.title}
4. </li>
5. ))}

**React protip**: take the habit of validating props with [PropTypes](https://reactjs.org/docs/typechecking-with-proptypes.html" \t "_blank)

Finally the component gets exported as List. List is the result of connecting the stateless component ConnectedList with the Redux store.

A stateless component does not have its own local state. Data gets passed to it as props

Still confused? I was too. Understanding how **connect** works will take some time. Fear not, the road to learn Redux is paved with “ah-ha” moments.

I suggest taking a break for exploring both **connect** and **mapStateToProps.**

Once you’re confident about them head over the next section!

## React Redux tutorial: Form component and Redux actions

The Form component we’re going to create is a bit more complex than List. It’s a form for adding new items to our application.

Plus it is a **stateful component**.

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

A stateful component? “Valentino, we’re talking about Redux for managing the state! Why on earth would you give Form its own local state??”

**Even when using Redux it is totally fine to have stateful components**.

Not every piece of the application’s state should go inside Redux.

In this example I don’t want any other component to be aware of the Form local state.

And that’s perfectly fine.

What does the component do?

The component contains some logic for updating the local state upon a form submission.

Plus it receives a Redux action as prop. This way it can update the global state by dispatching the addArticle action.

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

1. // src/js/components/Form.js
2. import React, { Component } from "react";
3. import { connect } from "react-redux";
4. import uuidv1 from "uuid";
5. import { addArticle } from "../actions/index";
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;

What can I say about the component? Besides **mapDispatchToProps** and **connect** it’s standard React stuff.

**mapDispatchToProps connects Redux actions to React props**. This way a connected component is able to dispatch actions.

You can see how the action gets dispatched in the handleSubmit method:

1. // ...
2. handleSubmit(event) {
3. event.preventDefault();
4. const { title } = **this**.state;
5. const id = uuidv1();
6. **this**.props.addArticle({ title, id }); // Relevant Redux part!!
7. // ...
8. }
9. // ...

Finally the component gets exported as Form. Form is the result of connecting ConnectedForm with the Redux store.

Side note: the first argument for connect must be nullwhen mapStateToProps is absent like in the Form example. Otherwise you’ll get TypeError: dispatch **is** not a function.

Our components are all set!

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;

Install uuid with:

1. npm i uuid --save-dev

Now run webpack (or Parcel) with:

1. npm start