LAST UPDATED 24TH DECEMBER 2018 BY VALENTINO GAGLIARDI

React Redux Tutorial for Beginners: The Definitive Guide (2019)

*The simplest **React Redux tutorial** I wish I had when I started learning*

When I first started learning **Redux** I wish I could find the simplest tutorial ever.

Despite the great resources out there I couldn't wrap my head around some of the Redux concepts.

I knew what's the **state**. But **actions**, **action creators, and reducers?** They were obscure for me.

Last but not least I didn't know how to glue **React and Redux together**.

During those days I started writing my own React Redux tutorial and since then I learned a lot.

I taught myself the Redux fundamentals by writing this guide. I hope it'll be useful for all those learning React and Redux.

Enjoy the reading!

Table of Contents

1. React Redux tutorial: who this guide is for
2. React Redux tutorial: what you will learn
3. React Redux tutorial: a minimal React development environment
4. React Redux tutorial: what is the state?
5. React Redux tutorial: what problem does Redux solve?
6. React Redux tutorial: why should I learn Redux?
7. React Redux tutorial: should I use Redux?
8. React Redux tutorial: getting to know the Redux store
9. React Redux tutorial: getting to know Redux reducers
10. React Redux tutorial: getting to know Redux actions
11. React Redux tutorial: refactoring the reducer
12. React Redux tutorial: Redux store methods
13. React Redux tutorial: connecting React with Redux
14. React Redux tutorial: what is react-redux?
15. React Redux tutorial: App component and Redux store
16. React Redux tutorial: List component and Redux state
17. React Redux tutorial: Form component and Redux actions
18. React Redux tutorial: what is a Redux middleware?
19. React Redux tutorial: your first Redux middleware
20. React Redux tutorial: asynchronous actions in Redux, the naive way
21. React Redux tutorial: asynchronous actions in Redux with Redux Thunk
22. React Redux tutorial: asynchronous actions in Redux with Redux Saga
23. React Redux tutorial: best courses for learning Redux
24. React Redux tutorial: best books for learning Redux
25. React Redux tutorial: wrapping up

React Redux tutorial: who this guide is for

The following React Redux guide is exactly what you're looking for if:

- you have a good grasp of Javascript, ES6, and React
- you're looking forward to learn Redux in the most simple way

React Redux tutorial: what you will learn

In the following guide you will learn:

1. what is Redux
2. how to use Redux with React

React Redux tutorial: a minimal React development environment

Before starting off **make sure you have a React development environment ready to roll**.

To make one you can follow this tutorial of mine: How to set up React, webpack, and Babel or you can also use **create-react-app**.

React Redux tutorial: what is the state?

To **understand what is Redux** you must first understand what is the **state**.

If you have ever worked with React the term state should be no surprise to you.

I guess you already wrote some **stateful React component**:

```
import React, { Component } from "react";

class ExampleComponent extends Component {
  constructor() {
```

```
    super();

    this.state = {
      articles: [
        { title: "React Redux Tutorial for Beginners", id: 1 },
        { title: "Redux e React: cos'è Redux e come usarlo con React", id: 2 }
      ]
    };
  }

  render() {
    const { articles } = this.state;
    return <ul>{articles.map(el => <li key={el.id}>{el.title}</li>)}</ul>;
  }
}
```

A **stateful React component** is a JavaScript ES6 class (although I guess it will be no longer the case sooner or later, see React Hooks).

Every stateful React component **carries its own state**.

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

The state could also change in response to actions and events: in React you can update the local component's state with setState.

But **what is the state** anyway? The term state is not tied exclusively to React. State is all around you. Even **the simplest JavaScript application has a state**. Consider the following example:

the user clicks a button

a modal appears afterwards

Guess what, **in this trivial interaction there is a state** we must deal with.

We could describe the initial state as a plain JavaScript object:

```
var state = {
  buttonClicked: 'no',
  modalOpen: 'no'
}
```

And when the user clicks the button we have:

```
var state = {
  buttonClicked: 'yes',
  modalOpen: 'yes'
}
```

How do you **keep track of those things in JavaScript** besides storing the state in an object? Is there **a library that can help us tracking the state in a reliable way**?

React Redux tutorial: what problem does Redux solve?

A typical JavaScript application is full of state. Here are some examples:

- 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

State is everywhere in JavaScript. But can you imagine **how much state a React application has**?

Yeah, you can get by with keeping the state within a parent React component as long as the application remains **small**. Then things will become tricky especially when you add more behaviours to the app.

At some point we may want to reach for a consistent way for keeping track of state changes. Not only, I'd say that the **frontend shouldn't know about the business logic**. Ever.

So what are the alternatives for managing the state of a React component?

Redux is one of them.

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**.The benefits of this approach might be not so evident. Things will look clear as soon as you'll get your feet wet with Redux.

In the next sections we'll see why you should learn Redux and when to use Redux within your applications. But first let me help you understand why and if you should learn Redux.

React Redux tutorial: why should I learn Redux?

Are you trying to learn Redux but you're going nowhere? Redux literally scares most beginners. But that shouldn't be your case.

Redux is not that hard. The key is: don't rush learning Redux just because.

You should **start learning Redux** if you're **motivated and passionate** about it.

Take your time. I started learning Redux because:

- I was 100% interested in learning how Redux works
- I was eager to improve my React skills
- the combination React/Redux is ubiquitous
- Redux is **framework agnostic**. Learn it once, use it almost everywhere (Vue JS, Angular)

But … is Redux a good investment? State is **pervasive and hard in JavaScript application so that state management in JS is still an unsolved problem**.

Another truth is: real world JavaScript applications almost always make use of a state management library.

Will Redux disappear in the future? Maybe. But the **patterns will stick forever** and it **will be invaluable for your career as a front end developer**.

In the end, **learning Redux** or an equivalent state management library **is a must, even if it has a steep learning curve.**

React Redux tutorial: should I use Redux?

Using Redux or Flux (or Mobx) for state management is up to you.

Maybe you need none of these libraries. They have a cost: they add another layer of abstraction to your application.

But I prefer thinking about **Redux as an investment**, not as a cost.

Another common question for Redux beginners is: how do you know **when you're ready to use Redux in your application**?

If you think about it there is no rule of thumb for determining when you do need **Redux for managing the state**.

Redux also offers a lot of convenience for a JavaScript developer. Debugging, action replaying. And much more.

When I start a new React project I'm always tempted to add Redux straight away. But as developers **we overengineer our code automatically.**

So, when should you add Redux to a project?

Before picking Redux take your time to explore alternative patterns. In particular try to get the most out of React's state and props.

Dave Ceddia has a nice writeup with a lot of great insights for using children props as an alternative before reaching for Redux.

And don't forget that a React project can be easily refactored to include Redux later.

What I found is that you should **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

If that makes still no sense for you do not worry, I felt the same.

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

And in fact it worked like that for me.

Courtesy of dev.to

Before going further take your time to understand what problem does Redux solve and whether you’re motivated or not to learn it.

Be aware that Redux is not useful for smaller apps. It really shines in bigger ones. Anyway, learning Redux even if you’re not involved in something big wouldn’t harm anyone.

In the next section we’ll start building a proof of concept to introduce:

- the **Redux fundamental principles**
- Redux alongside with React

Again, make sure you have a React development environment ready to roll: you can follow How to set up React, webpack, and Babel or go with create-react-app.

React Redux tutorial: getting to know the Redux store

Actions. Reducers. I kind of knew about them. But one thing wasn't clear to me: **how were all the moving parts glued together**?

Were there some **minions** or what?

In Redux there are no minions (unfortunately).

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**.

Move into your React development environment and install Redux:

```
npm i redux --save-dev
```

Create a directory for the store:

```
mkdir -p src/js/store
```

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

```
// src/js/store/index.js

import { createStore } from "redux";
import rootReducer from "../reducers/index";

const store = createStore(rootReducer);

export default store;
```

As you can see from the code above, **store is the result of createStore** which in turn is a function from the redux library.

createStore takes a reducer as the first argument and in our case we passed in rootReducer.

You may also pass an initial state to createStore which is useful for server side rendering but for now we're not interested in that.

The most important concept here is that **the state in redux comes from reducers. Let's make it clear: reducers produce the state of your application**.

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, 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).

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

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:

```
mkdir -p src/js/reducers
```

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

```
// src/js/reducers/index.js

const initialState = {
  articles: []
};

function rootReducer(state = initialState, action) {
  return state;
};

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.

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 of stuff 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: { title: '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:

```
mkdir -p src/js/actions
```

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

```
// src/js/actions/index.js

export function addArticle(payload) {
  return { type: "ADD_ARTICLE", payload }
};
```

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:

```
mkdir -p src/js/constants
```

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

```
// src/js/constants/action-types.js

export const ADD_ARTICLE = "ADD_ARTICLE";
```

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

```
// src/js/actions/index.js

import { ADD_ARTICLE } from "../constants/action-types";

export function addArticle(payload) {
  return { type: ADD_ARTICLE, payload };
}
```

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**

But what's a **Redux reducer** made of?

A reducer is a JavaScript function taking **two parameters**: **state** and **action**. A reducer function may use a **switch statement** (but I prefer using **if**) for handling every action type.

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 valid clause the **reducer calculates the next state** and **returns a new object**.

Now 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.js and update the reducer as follow:

```
// src/js/reducers/index.js

import { ADD_ARTICLE } from "../constants/action-types";

const initialState = {
  articles: []
};

function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    state.articles.push(action.payload);
```

```
  }
  return state;
}

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 is an impure function: it alters the original array. But there's more! Do you rememeber the third principle of Redux? **The state is immutable and cannot change in place**. Instead in our reducer we're mutating the original object!

We need a fix. First we can return a new state, ie **a new JavaScript object with Object.assign**. This way we keep the **original state immutable**. Then we can use **Array.prototype.concat in place of Array.prototype.push for keeping the initial array** immutable:

```
import { ADD_ARTICLE } from "../constants/action-types";

const initialState = {
  articles: []
};

function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    return Object.assign({}, state, {
      articles: state.articles.concat(action.payload)
    });
  }
  return state;
}

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 for arrays
- Using Object.assign() and ...spread for objects

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

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 for managing the state. The most important methods are:

- getState for accessing the current state of the application
- dispatch for dispatching an action
- 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 a new file named **src/js/index.js** and update the file with the following code:

```
import store from "../js/store/index";
import { addArticle } from "../js/actions/index";

window.store = store;
window.addArticle = addArticle;
```

Now open up **src/index.js** as well, clean up its content and update it as follows:

```
import index from "./js/index"
```

Now run webpack dev server (or Parcel) with:

```
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**:

```
store.getState()
```

output:

```
{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**.

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:

```
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** method.

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

Let's dispatch the action with:

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

Right after running the above code you should see:

```
Look ma, Redux!!
```

To verify that the state changed run again:

```
store.getState()
```

The output should be:

```
{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 for **accessing the current state** of the application
- dispatch for **dispatching an action**
- 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. I knew how to dispatch an action with dispatch and how to listen for

state changes with 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**.

Before moving forward install react-redux by running:

```
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: what is react-redux?

react-redux 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.**

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. By doing so a connected React component will have access to the exact part of the store it needs.

And what about mapDispatchToProps? 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**.

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.

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 (**if you're in create-react-app modify src/index.js instead**):

```
import React from "react";
import { render } from "react-dom";
import { Provider } from "react-redux";
import store from "./store/index";
import App from "./components/App.jsx";
// if you're in create-react-app import the files as:
// import store from "./js/store/index";
// import App from "./js/components/App.jsx";

render(
  <Provider store={store}>
    <App />
  </Provider>,
  // The target element might be either root or app,
  // depending on your development environment
  // document.getElementById("app")
  document.getElementById("root")
);
```

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:

```
mkdir -p src/js/components
```

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

```
// src/js/components/App.jsx
import React from "react";
import List from "./List";

const App = () => (
  <div className="row mt-5">
```

```
    <div className="col-md-4 offset-md-1">
    <h2>Articles</h2>
      <List />
    </div>
  </div>
);

export default App;
```

Take moment and look at the component without the markup:

```
import React from "react";
import List from "./List";

const App = () => (
      <List />
);

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.

Connect takes at least one argument.

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

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

```
// src/js/components/List.jsx

import React from "react";
```

```
import { connect } from "react-redux";

const mapStateToProps = state => {
  return { articles: state.articles };
};

const ConnectedList = ({ articles }) => (
  <ul className="list-group list-group-flush">
    {articles.map(el => (
      <li className="list-group-item" key={el.id}>
        {el.title}
      </li>
    ))}
  </ul>
);

const List = connect(mapStateToProps)(ConnectedList);

export default List;
```

The List component receives the prop articles which is a copy of the articles array we saw in the Redux state. It comes from the reducer:

```
const initialState = {
  articles: []
};

function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    return Object.assign({}, state, {
      articles: state.articles.concat(action.payload)
    });
  }
  return state;
}
```

Always remeber: the state in redux comes from reducers. Now,it's a matter of using the prop inside JSX for generating a list of articles:

```
{articles.map(el => (
  <li className="list-group-item" key={el.id}>
    {el.title}
  </li>
))}
```

React protip: take the habit of validating props with PropTypes or even better, use TypeScript

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

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.

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

It receives a Redux action as well. This way it can update the global state by **dispatching the addArticle action**.

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

```
// src/js/components/Form.jsx
import React, { Component } from "react";
import { connect } from "react-redux";
import uuidv1 from "uuid";
import { addArticle } from "../actions/index";

function mapDispatchToProps(dispatch) {
  return {
    addArticle: article => dispatch(addArticle(article))
  };
}

class ConnectedForm extends Component {
  constructor() {
    super();

    this.state = {
      title: ""
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ [event.target.id]: event.target.value });
  }

  handleSubmit(event) {
    event.preventDefault();
    const { title } = this.state;
    const id = uuidv1();
    this.props.addArticle({ title, id });
    this.setState({ title: "" });
  }

  render() {
    const { title } = this.state;
    return (
      <form onSubmit={this.handleSubmit}>
        <div className="form-group">
          <label htmlFor="title">Title</label>
          <input
            type="text"
            className="form-control"
            id="title"
            value={title}
            onChange={this.handleChange}
          />
        </div>
```

```
51.          <button type="submit" className="btn btn-success btn-lg">
52.            SAVE
53.          </button>
54.        </form>
55.      );
56.    }
57.  }
58.
59.  const Form = connect(null, mapDispatchToProps)(ConnectedForm);
60.
61.  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 null when 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.jsx";
3.  import Form from "./Form.jsx";
4.
5.  const App = () => (
```

```
  <div className="row mt-5">
    <div className="col-md-4 offset-md-1">
      <h2>Articles</h2>
      <List />
    </div>
    <div className="col-md-4 offset-md-1">
      <h2>Add a new article</h2>
      <Form />
    </div>
  </div>
);

export default App;
```

Install uuid with:

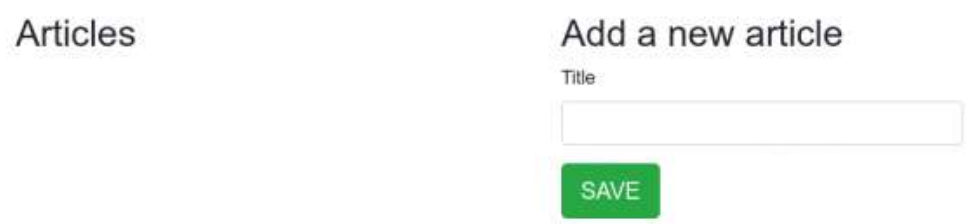
```
npm i uuid --save-dev
```

Now run webpack (or Parcel) with:

```
npm start
```

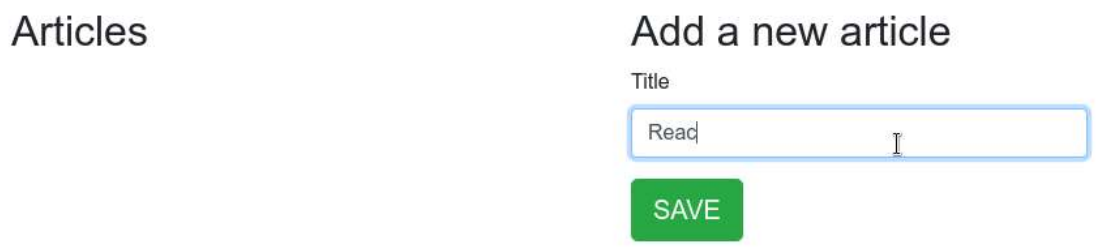
and head over to http://localhost:8080

You should see the following working POC:



Nothing fancy but still useful for showing React and Redux at work!

The **List component on the left is connected to the Redux store**. It will re-render whenever you add a new item.



Psst .. If you don't see anything on the browser make sure that document.getElementById("app") in src/js/index.js matches a real element inside the page:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.min.css" >
    <title>How to set up React, Webpack, and Babel</title>
</head>

<body>
    <div class="container">
        <div id="root">
        </div>
    </div>
</body>

</html>
```

Whoaaa! You did it! But we're not done yet! In the next section we'll look at Redux middlewares. Hold tight!

NOTE: I test and update this tutorial often. I hate to say "it works for me", but really, it works. If you have any problem with the tutorial or if your table doesn't update just send me your code to valentino@valentinog.com and I'll take a look if I can (don't send me the whole node_modules folder, thanks!)

React Redux tutorial: what is a Redux middleware?

So far we saw the building blocks of Redux: the **store**, in charge for orchestrating all the interactions in Redux. Then we saw the **reducer** which is a producer: reducers make the state in Redux.

Then there are **actions**, plain JavaScript objects with a property named type. Finally we have **action creators** which are plain JavaScript function in charge for returning Redux actions.

On top of that we saw some of the Redux's principle: the state is immutable and can change only in response to actions.

Now, imagine the following scenario: **you want to prevent the user from creating articles containing particular words inside the title**. Let's take a look at handleSubmit in Form.jsx:

```
// ...
  handleSubmit(event) {
    event.preventDefault();
    const { title } = this.state;
    const id = uuidv1();
    this.props.addArticle({ title, id });
  }
// ...
```

We can just add a check before this.props.addArticle right? It could be something like:

```
// ...
```

```
  handleSubmit(event) {
    event.preventDefault();
    const { title } = this.state;
    const id = uuidv1();
    // let's add a check for forbidden words
    const forbiddenWords = ['spam', 'money'];
    const foundWord = forbiddenWords.filter(word => title.includes(word) )
    if (foundWord) {
      return this.props.titleForbidden();
    }
    //
    this.props.addArticle({ title, id });
  }
// ...
```

But wasn't the **entire point of Redux moving the logic out of our React components**? Yes! So what? Can we check the title property inside the reducer? Maybe! And while we're there let's dispatch another action in response to a forbidden word. But how I'm supposed to access dispatch inside a reducer? Hold on ...

It's clear that we want to reach for something different. Looks like **we want to check the action payload (and the title property) before the actions is passed to the reducer**. There should be a way for tapping into the application's flow and altering its behaviour. And guess what, that's **exactly what a Redux middleware does**.

A Redux middleware is a function that is able to intercept, and act accordingly, our actions, before they reach the reducer. And while the theory is quite simple, a Redux middleware can look a bit confusing. In its basic form a Redux middleware is a function returning a function, which takes next as a parameter. Then the inner function returns another function which takes action as a parameter and finally returns next(action).

Are you still there? Hold tight because you're going to write your first Redux middleware. Here's how it looks like:

```
function forbiddenWordsMiddleware() {
  return function(next){
    return function(action){
      // do your stuff
      return next(action);
    }
  }
}
```

I know, you want to cry and change career but bear with me. **Middlewares in Redux are super important because they will hold the bulk of your application's logic**. If you think about it **there is no better place than a middleware for abstracting away business logic**. And the nice thing is that while **inside the middleware you can access getState and dispatch**, like so:

```
function forbiddenWordsMiddleware({ getState, dispatch }) {
  return function(next){
    return function(action){
      // do your stuff
      return next(action);
    }
  }
}
```

Armed with that knowledge we can create our first Redux middleware: it should check whether the action's payload has bad words into it! We'll see the actual implementation into the next section.

React Redux tutorial: your first Redux middleware

The middleware we're going to build should inspect the action's payload.

There are a lot of benefits from using a Redux middleware, even for simplest tasks:

- the logic can live outside React (or any other library/framework)
- middlewares become reusable pieces of logic, easily to reason about
- middlewares can be tested in isolation
- we keep the components clean

So, let's get our hands dirty! Create a new folder for holding middlewares:

```
mkdir -p src/js/middleware
```

Now create a new file named **index.js** in **src/js/middleware**. The structure of our first middleware should match the following:

```
function forbiddenWordsMiddleware({ dispatch }) {
  return function(next){
    return function(action){
      // do your stuff
      return next(action);
    }
  }
}
```

For now we don't need getState, we just get dispatch as the first parameter. Nice. Let's implement the logic now. We need to check the action payload, namely the title property. If the title matches one or more bad words we stop the user from adding the article.

Also, the check should fire up only when the action is of type ADD_ARTICLE. It makes sense. How about this one?

```
import { ADD_ARTICLE } from "../constants/action-types";

const forbiddenWords = ["spam", "money"];

export function forbiddenWordsMiddleware({ dispatch }) {
  return function(next) {
    return function(action) {
      // do your stuff
      if (action.type === ADD_ARTICLE) {

        const foundWord = forbiddenWords.filter(word =>
          action.payload.title.includes(word)
        );

        if (foundWord.length) {
          return dispatch({ type: "FOUND_BAD_WORD" });
        }
      }
      return next(action);
    };
  };
}
```

Here's what the middleware does: when action type is ADD_ARTICLE check if action.payload.title contains a bad word. If it does then dispatch an action of type "FOUND_BAD_WORD", otherwise **let the next action pass**.

And this last point is really important: **you should always return next(action) in your middlewares**. If you **forget to return next(action) the application will stop, and no other action will reach the reducer**.

Now, time to wire up forbiddenWordsMiddleware to the Redux store. For that we need to import our middleware, another utility from Redux (**applyMiddleware**) and then cook everything together.

Open up **src/js/store/index.js** and modify the file like so:

```
// src/js/store/index.js

import { createStore, applyMiddleware } from "redux";
import rootReducer from "../reducers/index";
import { forbiddenWordsMiddleware } from "../middleware";

const store = createStore(
  rootReducer,
  applyMiddleware(forbiddenWordsMiddleware)
);

export default store;
```

Oh, and for using **Redux Dev Tools together with other middlewares** here's what you should do (notice the use of **compose**):

```
// src/js/store/index.js

import { createStore, applyMiddleware, compose } from "redux";
import rootReducer from "../reducers/index";
import { forbiddenWordsMiddleware } from "../middleware";

const storeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(
  rootReducer,
  storeEnhancers(applyMiddleware(forbiddenWordsMiddleware))
);

export default store;
```

Save and close the file, run npm start and check if the middleware works. Try to add an article with "money" in its title:

Indeed it works! Good job! Now, here's an exercise for you: update the reducer so it recognises the new action "FOUND_BAD_WORD". Ideally it should update the state with a message of sort that can be showed to the user.

Then add a new action creator for “FOUND_BAD_WORD”. Update the middleware for using the action creator instead of the plain action. Try to add a test for your middleware. Think about it, you’ll find out it’s really simple to test middlewares in Redux.

And that’s it! In the next sections we’ll explore **asynchronous actions in Redux** with Redux Thunk and Redux Saga!

React Redux tutorial: asynchronous actions in Redux, the naive way

So far we were dealing with synchronous data. That is, the act of **dispatching an action is synchronous**. No AJAX, no promises. We return a plain object from our action creators. And when the action reaches the reducer we return the next state.

Now, **suppose we want to fetch data from an API**. In React you would put a call in componentDidMount and call it a day. But how about Redux? What’s a good place for calling asynchronous functions? Let’s think a moment about it.

Reducers? No no. Reducers should stay lean and clean. A reducer is not a good place for asynchronous logic.

Actions? How I am supposed to do that? Actions in Redux are plain objects. And **what about action creators**? An action creator is a function, and it looks like a nice spot for calling an API, doesn’t it? Let’s give it a shot.

We’ll create a new action named getData. This action calls an API with fetch and returns a Redux action.

Open up **src/js/actions/index.js** and create a new action named **getData**:

```
// src/js/actions/index.js

// ...
// our new action creator. Will it work?
export function getData() {
  return fetch("https://jsonplaceholder.typicode.com/posts")
    .then(response => response.json())
```

```
    .then(json => {
      return { type: "DATA_LOADED", payload: json };
    });
}
```

It makes perfect sense. But will it work? Now let's wire up a React component so it dispatches getData from componentDidMount. We'll use mapDispatchToProps (this time with the object shorthand form) for mapping Redux action creators to our component's props. Create a new React component in **src/js/components/Posts.jsx**:

```
import React, { Component } from "react";
import { connect } from "react-redux";
import { getData } from "../actions/index";

export class Post extends Component {
  constructor() {
    super();
  }

  componentDidMount() {
    // calling the new action creator
    this.props.getData();
  }

  render() {
    return null;
  }
}

export default connect(
  null,
  { getData }
)(Post);
```

And finally update **src/js/components/App.jsx** to use the new component:

```
import React from "react";
import List from "./List.jsx";
import Form from "./Form.jsx";
import Post from "./Posts.jsx";

const App = () => (
  <div className="row mt-5">
    <div className="col-md-4 offset-md-1">
      <h2>Articles</h2>
      <List />
    </div>
    <div className="col-md-4 offset-md-1">
      <h2>Add a new article</h2>
      <Form />
```

```
15.       </div>
16.       <div className="col-md-4 offset-md-1">
17.         <h2>API posts</h2>
18.         <Post />
19.       </div>
20.     </div>
21.   );
22.   export default App;
```

Save and close the files, and look at what we've got in the browser:

Interesting! "Error: Actions must be plain objects. Use custom middleware for async actions". It looks like **we cannot call fetch from within an action creator in Redux**. Now what?

For making things work we need a custom middleware. Luckily there's something ready for us: **redux-thunk**.

React Redux tutorial: asynchronous actions in Redux with Redux Thunk

We just learned that **calling fetch from an action creator does not work**. That's because **Redux is expecting objects as actions but we're trying to return a Promise**. With **redux-thunk we can overcome the problem** and return functions from action creators. Inside that function we can call APIs, delay the dispatch of an action, and so on.

First we need to install the middleware with:

```
npm i redux-thunk --save-dev
```

Now let's load the middleware in **src/js/store/index.js**:

```
// src/js/store/index.js

import { createStore, applyMiddleware, compose } from "redux";
import rootReducer from "../reducers/index";
import { forbiddenWordsMiddleware } from "../middleware";
import thunk from "redux-thunk";

const storeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(
  rootReducer,
  storeEnhancers(applyMiddleware(forbiddenWordsMiddleware, thunk))
);

export default store;
```

At this point we need to refactor getData to use redux-thunk. Open up **src/js/actions/index.js** and update the action creator like so:

```
// src/js/actions/index.js

// ...

export function getData() {
  return function(dispatch) {
    return fetch("https://jsonplaceholder.typicode.com/posts")
      .then(response => response.json())
      .then(json => {
        dispatch({ type: "DATA_LOADED", payload: json });
      });
  };
}
```

That's redux-thunk!

A few things worth nothing in the new version of getData: **the fetch call gets returned from an outer function and the outer function has dispatch as a parameter**. If you want to access the state inside the action creator you can add getState in the parameter's list.

Also, notice **the use of dispatch inside then**. **We need to explicitly call dispatch inside the async function for dispatching the action**, otherwise it won't work.

With that in place we're ready to update our reducer with the new action type. Open up **src/js/reducers/index.js** and add a new if statement. We can also add a new key inside initialState for saving the articles from the API:

```
// src/js/reducers/index.js
import { ADD_ARTICLE } from "../constants/action-types";

const initialState = {
  articles: [],
  remoteArticles: []
};

function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    return Object.assign({}, state, {
      articles: state.articles.concat(action.payload)
    });
  }

  if (action.type === "DATA_LOADED") {
    return Object.assign({}, state, {
      remoteArticles: state.remoteArticles.concat(action.payload)
    });
  }
  return state;
}

export default rootReducer;
```

(I know, I didn't put DATA_LOADED inside its own named costant. I'd left as an exercise for you. Hope you don't mind!)

Finally we're ready to update our Post component for displaying our "remote" posts. We will use mapStateToProps for selecting ten posts:

```
import React, { Component } from "react";
import { connect } from "react-redux";
import { getData } from "../actions/index";

export class Post extends Component {
  constructor() {
    super();
  }

  componentDidMount() {
```

```
    this.props.getData();
  }

  render() {
    return (
      <ul className="list-group list-group-flush">
        {this.props.articles.map(el => (
          <li className="list-group-item" key={el.id}>
            {el.title}
          </li>
        ))}
      </ul>
    );
  }
}

function mapStateToProps(state) {
  return {
    articles: state.remoteArticles.slice(0, 10)
  };
}

export default connect(
  mapStateToProps,
  { getData }
)(Post);
```

Save and close the files, and everything should work fine:

Look at that! Good job!

To recap: **Redux does not understand other types of action than a plain object**. If you want to move asynchronous logic from React to Redux and being able to return functions instead of plain objects you have to use a custom middleware.

redux-thunk is a middleware for Redux. With **redux-thunk you can return functions from action creators, not only objects. You can do asynchronous work inside your actions and dispatch other actions in response to AJAX calls**.

When to use redux-thunk? redux-thunk is a nice middleware that works very well for simpler use cases. But if your asynchronous logic involves more complex scenarios then redux saga might be a better fit.

And in the next section we'll finally take a look at Redux Saga. Hold tight!

An exercise for you: try to clean up your actions creators file by moving your async actions inside a custom middleware.

Another exercise for you: I didn't account for errors in our fetch call. Do it if you have time!

React Redux tutorial: asynchronous actions in Redux with Redux Saga

coming soon!

React Redux tutorial: best courses for learning Redux

Want to level up your Redux skills once you finish my tutorial? What's the best Redux course you can buy? What's the best Redux book you can get?

I use React and Redux for a lot of project and yet I can't beat the real experts out there. People like Mark Erikson or Henrik Joreteg for example. They know their stuff.

So here's my advice: **stick with the experts when looking for JavaScript (and Redux) courses**. And the **best Redux course** in my opinion is **Practical Redux by Mark Erikson**.

Mark is a Redux mantainer. **He'll show you important Redux concepts and techniques**. I don't think there is a better person than Mark for learning Redux.

In Practical Redux you'll learn how to:

- control your UI behavior with Redux
- use Redux-ORM to manage relational data in your Redux store
- build a master/detail view to display and edit data