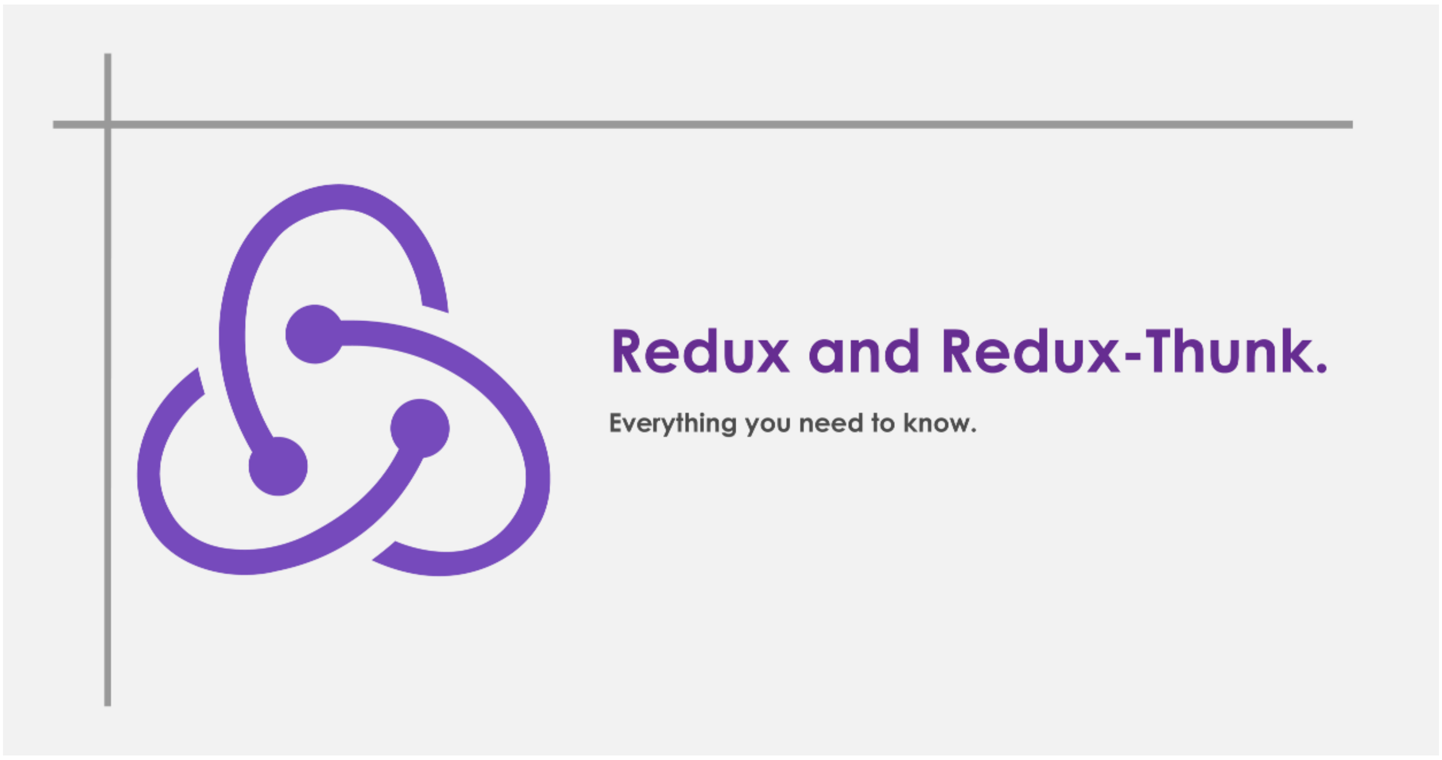
Bogdan Kulbida

Follow 133 Followers About

2019 Using Redux and Redux-Thunk with ReactJS. Explained.

Bogdan Kulbida May 19, 2019 · 9 min read



In this article, we are going to clarify what is *Redux* and *Redux-Thunk*. Here you will find all the information you need to understand where each fits into your application. Also, you will find a lot of examples with detailed explanations that will help you to understand all the concepts. Complementary source code will be provided for you as well with illustrations for a quick refresh.

find this post useful, please tap 👏 button below :)

We will so much appreciate that!

1. Redux and Redux-Thunk

First, let’s clarify what *Redux* is. In simple words, *Redux* is a tool or an addon for your *ReactJS* App that mutates your application state and makes two-way state exchange possible between your App and a Store. *Redux* changes your application state upon actions that the user of your application initiates. Such an action could be a mouse click or a button press.

A Store is a place in the memory within the browser where your application keeps its state.

What is *Redux-Thunk*? Essentially, it is an orchestration mechanism for commands for *Redux*. In other words, *Redux* changes application state upon instructions, and *Redux-Thunk* puts those instructions into a queue and makes sure each action is performed one at a time.

2. Why?

You may be wondering why we would want to orchestrate those commands? Since most of the events (or actions) within nearly all *ReactJS* applications happen asynchronously, to avoid race conditions, we need a robust mechanism, and *Redux-Thunk* is yet another synchronization tool.

then in request “**B**” by the CompanyID that came in the payload of the request “**A**”, you want to fetch details about the company. Since both requests are asynchronous, request “**B**” may start and finish before request “**A**” completes. Put simply, we have a few concurrent requests (because of the asynchronous nature of the environment in which our code executes), and so our code is prone to race condition bugs unless we explicitly control the order of each action’s execution. *Redux-Thunk* does that for us; it helps us to mitigate those issues and avoid data race conditions.

3. Data-Down, Actions-Up

Before we dive into *Redux-Thunk*, we need to understand some core concepts of the *Redux*. As we already mentioned, *Redux* is a tool that changes our application state, and to do that *Redux* needs to know where to take State object and where to put that object back once a mutation is done. We know that Store is a storage of some sort for our application State object. To achieve the desired behavior, we need to glue Store and our App together. How we do that? We create a Root component which takes our App and wraps it into a *Redux* native “Provider” component. A Root component is a gluing proxy object between our App and a Store. By wrapping App into the Producer component, we create a link for *Redux* and acknowledge it with the existence of our App.

To know what is “Data-Down, Actions-Up” design pattern is, please read this article.

Now we need to acknowledge within our App the existence of the Store and create a link for two-way communication (remember, it is a two-way binding information exchange to make *Data-Down* -> *Actions-Up* work).

```
import React, { Component } from "react";
import { Provider } from 'react-redux';
import App from './App';

class Root extends Component {

  render() {

    return (
    );

  }

};

export default Root;
```

Root.jsx hosted with ♡ by GitHub view raw

Root Component — it “simply” wraps our App Component

We do that by passing “store” property into the Provider component. So now the Provider is aware of our App’s existence and where the Store location is. Here is some code on the left that illustrates what we have just said.

Let’s step back and clarify where `this.props.store` came from. And to answer that question, let’s take a look at our entry point `index.js` file.

```
import React from "react";
import ReactDOM from "react-dom";
import Root from "./Root";

import { combineReducers, createStore, applyMiddleware } from "redux";
import thunk from "redux-thunk";

import { subscriptionsReducer as subscriptions } from "./reducers/subscription"

// you can have multiple reducers, this is why you need to merge them into one
// reducer using `combinereducers` function.
const reducer = combineReducers({state: subscriptions});

// Store object is managed by `redux-thunk`. This is now `redux-thunk` 'orchestrate' actions
```

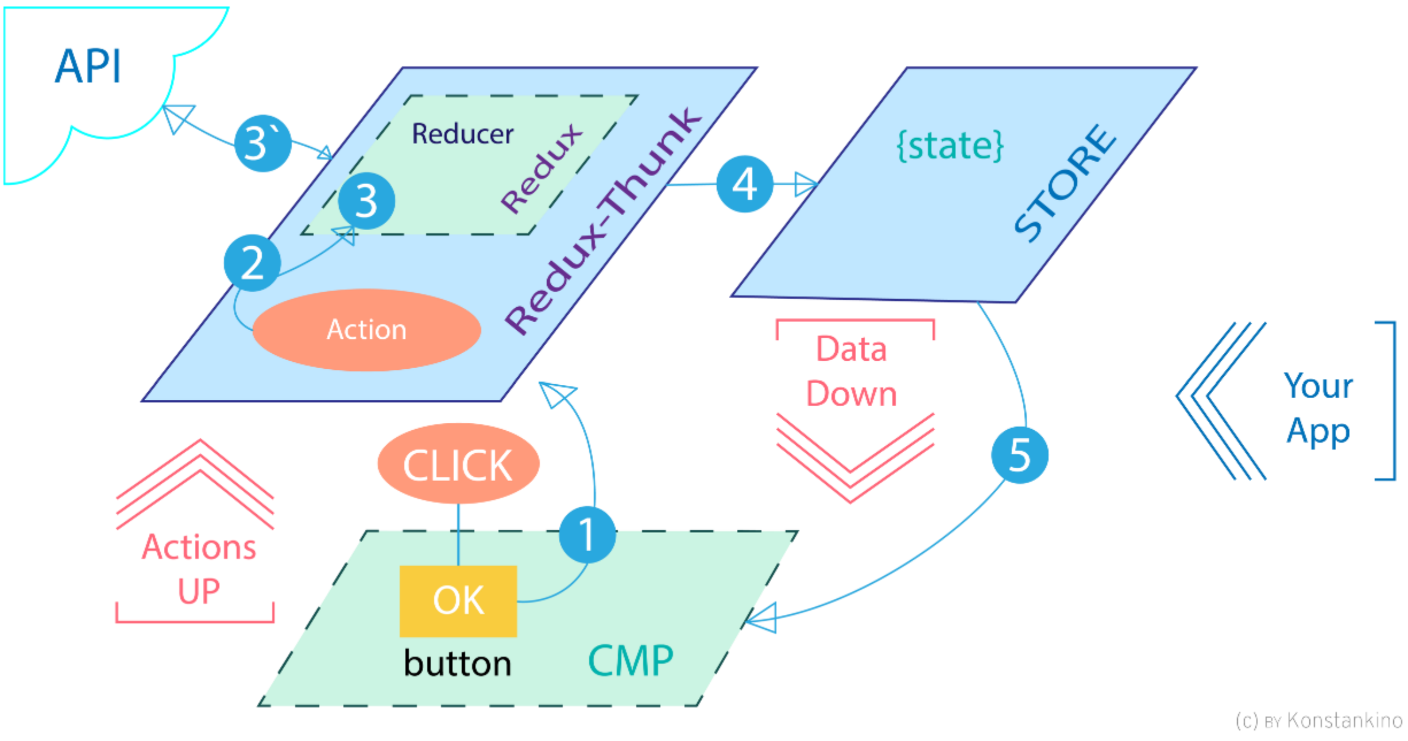
```
ReactDOM.render(<Root store={store} />, document.getElementById("root"));
```

index.js hosted with ♡ by GitHub view raw

index.js — entry point into our ReactJS App

Line #18 we mount our Root component into the HTML page — nothing exciting there. However, look at how we pass on "store" as a property (or props) into Root component. Also, **on line #16** we create our "store" object. We use the method `createStore` that ships with *Redux* and apply middleware with "**thunk**" object. About "**thunk**" we will talk later in this article. The line that has the most interest for us is **line #12**. We create a "**reducer**" object on that line, which is essentially a management layer for *ReduxJS* to mutate application `state` object.

Here is a general overview of what we have just said:



General structure of pretty much any ReactJS app

4. Reducer

`combineReducers` function accepts a hash object, and in our case that object has a single key-value pair. Please note that we named the key `state` — it is important. We will get back to that in a bit. For now, it is essential to understand that we need to create a single `reducer` object by calling `combineReducers` function and pass on all reducers our application may have. In our example, we have one reducer that is brought into the scope as a `subscriptions` object from the external file. Let's take a look at our *Reducer*.

Subscription Reducer — sets our App state by the action type

The *Reducer* has a few constants that represent action names that it needs to know how to handle. Then it has some default or better to say the initial state. **Line #13 to #46** is our *Reducer* in its glory. As we can see, it knows now to handle three commands or actions (match arm for each case operator).

OK, let's pause with our *Reducer* for a second and switch back to our App component. We have a few more concepts to clarify.

App.jsx Component

Let's take a look again at the `App` component. Previously, we mentioned two-way communication *Store* -> *Redux* -> *App* to render our application with data on the screen, and *App* -> *Redux* -> *Store* to continue the same state in the *Store* when *Reducer* updates (or shapes) our application `state`.

To make two-way communication possible, we need to apply some magic that **redux** and **react-redux** have to offer. And that is the `connect` method. We call that method with some configuration functions `mapStateToProps` and `mapActionsToProps`. Then whatever callback function is able to `connect` returns, we pass on to our `App`

5. Gluing Things Together

Now, let's take a look at `mapStateToProps`. That is a data layer of your application State. In our simple case, it returns a `state` object; however, as your application grows, you may want to slice that single `state` object into smaller, more cohesive and functional chunks of state and return a more complex data structure that conforms to your business logic and abstract or real-world objects within your application.

Another exciting portion of bits is `mapActionsToProps`. That is a behavioral or functional layer of your application. That is a scope where all your actions for all your components live. As you can see, we named our scope `actions` but again, as your application gains more code, you may want to slice and group your actions into cohesive, functional abstractions.

App.jsx Component

Now, let's look at **line #12**.

Remember when we previously named our key `state` above? That is where things get glued together. We are destructuring `this.props` and binding to `actions` and `state` variables and passing them onto our main component. Now our `MainComponent` knows where to take data (or state) from the Store and has some actions to offer for nested components.

6. Wiring Actions To the Component

Before we move on to our `MainComponent`, let's clarify how we make `actions` available for our components.

App.jsx Component

However, what interests us now is a way we plug-in those actions into our App via `mapActionsToProps` function. Again, we use some magic that *Redux* provides to us. Please note, we pass on function `dispatch` into our "actions" hash, that is how we create an interface or an API for *Redux-Thunk* to be able to manage and orchestrate operations on *Redux*'s behalf. We will get back to *Redux-Thunk* in the next section.

Now it's time to dive deeper and understand our application's actions.

Subscription Actions file — defines actions for our SubscriptionReducer

In this file, we define names for our actions by binding them to constants since they will never change and should be reused in our *Reducer*. What is interesting in this file is a function `fetchSubscription`. That is a public API for our component to trigger an action and initiate all the related actions. Note it calls "dispatch" functions quite a few times, and that is where we use an interface of the *Redux* and plug-in *Redux-Thunk* to be responsible for subsequent actions' execution.

7. Tripod: Three Sub-states

It is considered good practice to have three distinct actions (or sub-states) for one single interaction with your component. For example, when we click on a button, we expect to:

1. Show a loader => **this is sub-state one**

2. Fetch the data from the API, and

4. If the fetch fails, we may want to hide our loader and display an error message => **this is sub-state three**

Here is how that could be designed using three actions for one user interaction approach. `fetchSubscriptionBegin` is an action for subscription reducer to mutate our application state to show a loader. See our subscription actions code below again; **on line #16**, it sets `loading` to `true` via `fetchSubscriptionBegin` function. The following step, we call `fetch` function to get the data from the back end API, and once it is finished, we want *Redux-Thunk* to trigger the appropriate action: `fetchSubscriptionSuccess` or `fetchSubscriptionFailure` respectively.

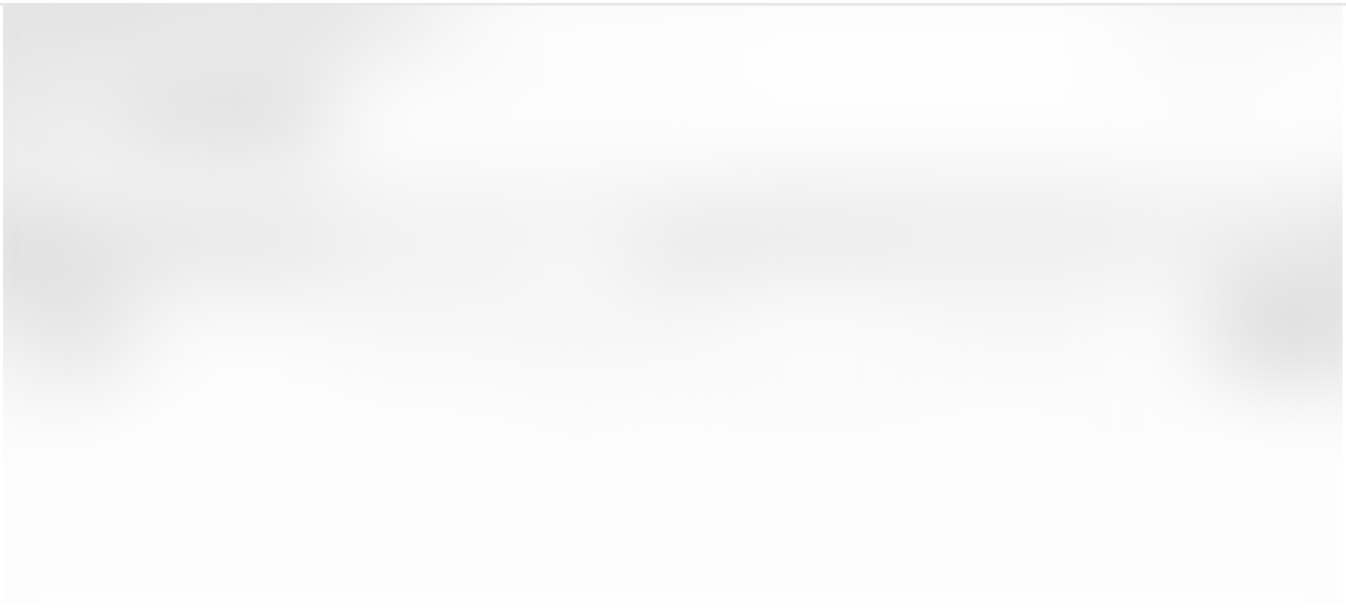
Please note you may have an explicit layer to handle errors. In our subscriptions actions example below, we have a simple `handleErrors` proxy function, but for your application, you may choose a more robust approach.

All three actions (or sub-states) `fetchSubscriptionBegin`, `fetchSubscriptionSuccess` and `fetchSubscriptionFailure` inform our *Reducer* that some part of our application state needs to change. All functions must return a simple JSON object with required `type` field that corresponds to an action type and so defines a slice of the state that *Reducer* has to mutate or change. Also, you may pass on any other data, such as we do with the `payload` key-pair, which is a JSON object that we got from our back end API.

Now, it's time to look at our `MainComponent` and see where we trigger our `fetchSubscription` action.

MainComponent.jsx — Our the only Component with user facing controls

The `MainComponent` is a straightforward component; our goal is to avoid extra complexity and make it as lean as possible. The function that interests us here is `handleLoadDataOnClick`. That is where we pass-in our back end API endpoint URL and trigger action with our `SubscriptionActions` scope object. For the sake of simplicity, we hard-coded the back end API URL. Also, note how we used the `loading` flag to give feedback to the user. Here is a quick demo:



8. Conclusion

In this article, we clarified what is *Redux* and *Redux-Thunk*. Here you have all the information you need to understand where each fits into your application. We showed some real-world examples with detailed explanations that help to understand all the concepts. If you like this article, please subscribe to our newsletters to get fresh updates — it is free.

As a thank you, here is a complete source code that we demonstrated in this article. You can use our code and pictures to refresh your knowledge at any time.

> We really need your clap! Please tap 👏 button below :) Here is our first 👏 to you. You are amazing!

.

React Redux Redux Thunk 2019 Apps Development