



Gethyl George Kurian

[Follow](#)

Web developer | React | Nodejs | GraphQL | Angular 1.X

Jan 16, 2017 · 9 min read

Understanding how redux-thunk works



Redux

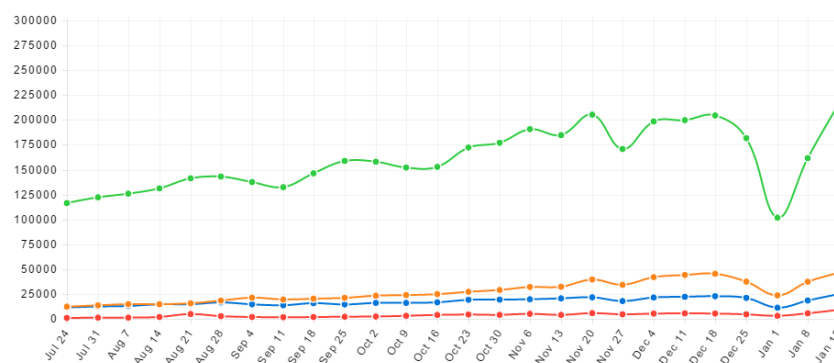
Redux-Thunk is the most popular middleware used to handle asynchronous actions in Redux.

You can read more about middleware in Redux [here](#).

You can look at the below comparison to see which middleware for Redux is downloaded the most, which in a way gives you an idea on which is the most popular on npm.

[redux-promise](#) [redux-saga](#) [redux-thunk](#) [redux-observable](#)

Downloads in past 6 Months ▾



npmtrnd.com showing number of times each packages were downloaded in last 6 months.

As you see above, redux-thunk is the most popular one. And by a huge margin :-)

I have used Redux-Saga, and see it gaining lot of popularity in the coming days/weeks/months. I also wrote a post on how to use it [here](#) in

Medium. I loved using saga, but I have to say, it is a bit complicated and would be an overkill for a simple app which needs to handle async actions.

I have not used the other two and therefore can't comment on them.

*But yes, I would definitely recommend you to have a look at **Redux-Saga**.*

Redux-Thunk is very small piece of code as you see below:-

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

    return next(action);
  };
}

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

export default thunk;
```

Oh yes, that's all there is to redux-thunk. But don't get fooled by the simplicity. This is a beautiful piece of code which does a lot.

. . .

How do I intend to make you understand redux-thunk?

Though I have used redux-thunk couple of times, I wanted to understand how exactly things were working under the hood.

In this post, I will explain what exactly happens in the functions `applyMiddleware` from **Redux** and `createThunkMiddleware` from **redux-thunk**. We need to understand both the functions to get a clear idea on how async is handled.

If you see the original code for `applyMiddleware` and `createThunkMiddleware` it may be hard to follow as you need to be familiar with the below concepts

1. ES6 arrow functions
2. Composing Functions
3. Currying functions

If you are strong with the above concepts then you might already know how redux-thunk works, and this article might not be of much help to you.

But I have written this article to help those understand who are not too familiar with the concepts, or are familiar but still find it hard to understand the code. I fall somewhere in between :-)

So I have created a small example React-Redux code, which has an async action. You can find the code [here](#).

In this example, I have created my version of `applyMiddleware` and `createThunkMiddleware` where I have

- Converted most of the ES6 arrow functions which were anonymous functions to normal named functions.
- Thrown in a lot of `console.logs` in both the functions to make it easy to understand the flow.
- And also split some steps into smaller steps so that it's easier to understand the flow.

So here is how my version of `applyMiddleware` function looks.

[Click here to check the original applyMiddleware.js](#)

```

1 //compose from redux/compose.js
2 function compose(...funcs) {
3     console.info("$$$ compose function ")
4     console.dir( funcs)
5     if (funcs.length === 0) {
6         return arg => arg
7     }
8
9     if (funcs.length === 1) {
10         return funcs[0]
11     }
12
13     return funcs.reduce((a, b) => (...args) => a(b(...ar
14 }
15
16 //applyMiddleware from redux/applyMiddleware.js
17 export const applyMiddleware = function applyMiddleware
18     //console.info("applyMiddleware")
19     return function (createStore) {
20         return function (reducer, preloadedState, enhancer
21             const store = createStore(reducer, preloadedStat
22             let dispatch = store.dispatch
23             let chain = []
24
25             const middlewareAPI = {
26                 getState: store.getState,
27                 dispatch: (action) => dispatch(action)
28             }
29
30             chain = middlewares.map(middleware => {
31                 console.info("~~~~~middleware ")
32                 console.dir( middleware)
33                 const middlewareReturn = middleware(middleware
34                 console.info("~~~~~middlewareReturn ")
35                 console.dir( middlewareReturn)
36                 return middlewareReturn
37             })

```

And here is `createThunkMiddleware` function

[click here to view the original createThunkMiddleware](#)

```
1 // createThunkMiddleware from redux-thunk.js
2 export function createThunkMiddleware(extraArgument)
3   return function thunkFunction ({ dispatch, getState}) {
4     return function nextFunction (next) {
5       console.info("In next function")
6       console.dir(next)
7       return function actionFunction (action) {
8         console.info("action RETURNED")
9         console.dir(action)
10
11         if (typeof action === 'function') {
12           console.info("++++++++++++++++++++++++++++++++++++")
13           console.info("action is function")
14           console.info("++++++++++++++++++++++++++++++++++++")
15           return action(dispatch, getState, extraArgument)
16         }
17
18         console.info("++++++++++++++++++++calling NEX")
19         console.dir(next)
```

Take a moment and try to analyze and see how this code is different from the actual source code.

. . .

Setting up

The easiest way start working with the example I have provided is to

1. Clone the project into your local folder

```
git clone https://github.com/Gethyl/UnderstandThunk
```

2. Change to that folder.

```
cd ./UnderstandThunk
```

3. Install the *dependencies* and *devDependencies* by issuing `yarn install` or `npm install`

. . .

Running The Example

If you check the `script` in `package.json` I have added

```
"dev": "webpack-dev-server --content-base --inline --hot"
```

So to run your app, all you have to do is

```
yarn run dev (or) npm run dev
```

And go to <http://localhost:8080/> and see the page.

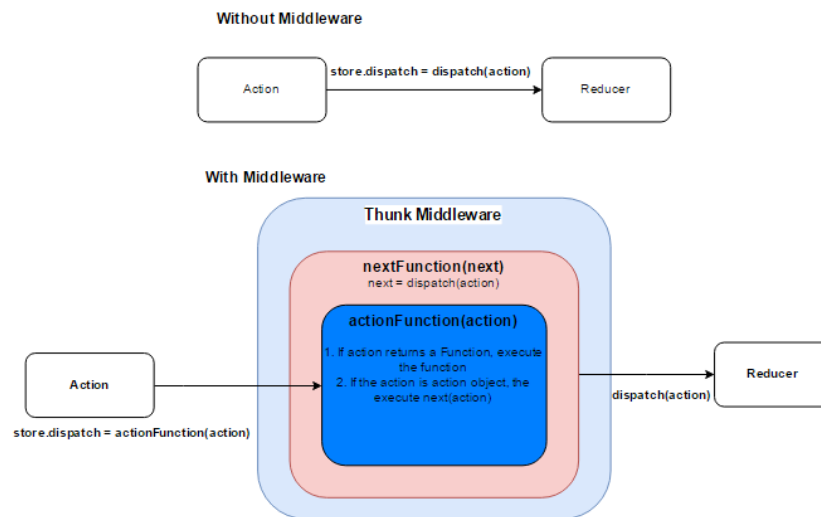
Understanding how redux-thunk works

A small example to help you understand how redux-thunk handles async actions

Load Async Action

. . .

Understanding the middleware and thunk



- As you see above, when you use **redux-thunk** to handle asynchronous actions, the action reaches the inner most function, `actionFunction`, which is responsible for executing the action. But this dispatch is something which we create in the middleware, and it is not the dispatch from the `createStore`, which has listeners and passes the values to the `reducer`.
- The dispatch to `reducer` happens from its parent function which is `nextFunction`.

Now let us try to understand how this happens.

For this you need to understand the two phases of the app.

1. When the app is starting and the store is created.
2. When the button is clicked.

When your app is starting:

Once your app is running, go to *developer tools in browser* and check your *console* to see the below messages:-

```

          ^^^^^^^^^^action.type^^^^^^^^^^^^^
1  ~~~~~middleware
    ▶ function thunkFunction(_ref)
1  ~~~~~middlewareReturn
    ▶ function nextFunction(next)
1  ~~~chain
    ▶ Array[1]
1  $$$ compose function
    ▶ Array[1]
1  ~~~composedFunc
    ▶ function nextFunction(next)
1  In next function
    ▶ function dispatch(action)
1  ~~~new dispatch AFTER compose i.e composedFunc RETURNED
    ▶ function actionFunction(action)
>

```

Perfect! Now that you see this, and before we move forward, let us try to understand what exactly happened here.

Have you ever wondered what happens when you add a middleware while creating a store?

Well let's find out

```

//client.js
import {createThunkMiddleware, applyMiddleware}
from './redux-thunk-local'
const app = document.getElementById('app')

const thunk = createThunkMiddleware()

const store = createStore(reducer, applyMiddleware(thunk))

ReactDOM.render(
  <Provider store={store}>
    <Layout/>
  </Provider>
  , app);

```

I will be using the version of `createThunkMiddleware` and `applyMiddleware` that I have created locally.

When the apps launches, and when it creates store, it will call **applymiddleware** with the middleware (**thunk**) that we have provided.

And on inspecting `redux/createStore.js`

```
//redux/createStore.js
export default function createStore(reducer, preloadedState,
  enhancer) {
  if (typeof preloadedState === 'function' && typeof
    enhancer === 'undefined') {
    enhancer = preloadedState
    preloadedState = undefined
  }

  if (typeof enhancer !== 'undefined') {
    if (typeof enhancer !== 'function') {
      throw new Error('Expected the enhancer to be a
        function.')
    }

    return enhancer(createStore)(reducer, preloadedState)
  }

  //skipping the next lines of code as it is irrelevant at
  this step
  ....
  ...
}
```

When you use a middleware, the `createStore` function calls

```
enhancer(createStore)(reducer, preloadedState)
```

In our case, it calls

```
applyMiddleware(thunk)(createStore)(reducer, preloadedState)
```

And this is how ***applyMiddleware*** function looks

```
const applyMiddleware = function
  applyMiddleware(...middlewares) {
    return function (createStore) {
      return function (reducer, preloadedState, enhancer)
```

Basically what you see above is:

1. `applyMiddleware(thunk)` returns an anonymous function `(createStore)`
2. The return is now executed with `(createStore)` and returns another anonymous function `(reducer, preloadedState, enhancer)`
3. And finally that is executed `(reducer, preloadedState)`
4. Now let us look at what goes inside the last anonymous function (step 3)

```
const store = createStore(reducer, preloadedState, enhancer)
let dispatch = store.dispatch
let chain = []
```

```
const middlewareAPI = {
  getState: store.getState,
  dispatch: (action) => dispatch(action)
}
```

```
chain = middlewares.map(middleware => {
  console.info("~~~~~middleware ")
  console.dir( middleware)
  const middlewareReturn = middleware(middlewareAPI)
  console.info("~~~~~middlewareReturn ")
  console.dir( middlewareReturn)
  return middlewareReturn
})
```

```
console.info("~~~chain ")
console.dir( chain)
```

4.1 It creates a `store` and we assign the `store.dispatch` to a local variable `dispatch`. *This step is important.*

4.2 Next it creates a local object `middlewareAPI` and assigns it the `getState` with `store.getState` and also `dispatch` which will take the action as input and dispatch it.

4.3 And finally the variable `chain` which takes all the middlewares and calls each middleware with the `middlewareAPI`. In our case, this is what the middleware looks like

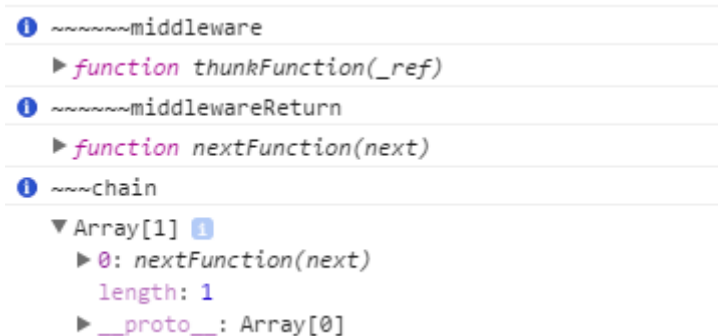
```
const middlewareReturn = thunkFunction({
  getState: store.getState,
```

```
    dispatch: (action) => dispatch(action)
  })
```

The `thunkFunction` is from `createThunkMiddleware`

```
export function createThunkMiddleware(extraArgument) {
  return function thunkFunction ({ dispatch, getState }) {
    return function nextFunction (next) {
```

which when executed will return `function nextFunction(next)` as you see below:



```

  ~~~~~middleware
    ▶ function thunkFunction(_ref)
  ~~~~~middlewareReturn
    ▶ function nextFunction(next)
  ~~~Chain
    ▼ Array[1]
      ▶ 0: nextFunction(next)
        length: 1
        __proto__: Array[0]
```

4.4 Next step is composing functions.

```
let composedFunc = compose(...chain)
console.info("~~~composedFunc ")
console.dir( composedFunc)
```

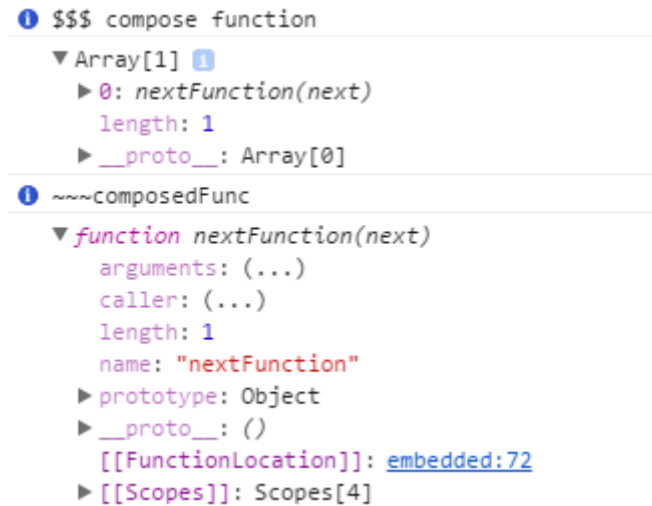
And this is how `redux/compose.js` looks.

```
function compose(...funcs) {
  console.info("$$$ compose function ")
  console.dir( funcs)
  if (funcs.length === 0) {
    return arg => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }
```

```
return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

This is where you need to be familiar with what **composing function** is. But for our example, since we use have only one middleware, as you see in the above code, the length is 1, and we return the single function as shown below, i.e we return what ever was chained earlier, and since it was just one function, we return that:-



4.5 In the next step, we call the composed function with

`store.dispatch` and pass the result to `dispatch` .i.e

```
dispatch = composedFunc(store.dispatch)
console.info("~~~new dispatch AFTER compose i.e
composedFunc RETURNED ")
console.dir( dispatch)
```

```
return {
  ...store,
  dispatch
}
```

In our case `dispatch` is:

```
dispatch = nextFunction(store.dispatch)
```

i.e we execute nextFunction with the dispatch from the store.

So keep in mind here that `(next)` that we use as parameter in `nextFunction` will be `store.dispatch` from `createStore` .

```
next = dispatch(action)
```

And this is how the next function looks:

```
return function nextFunction (next) {
  console.info("In next function")
  console.dir(next)
  return function actionFunction (action) {
    console.info("action RETURNED")
```

4.6 Therefore, local `dispatch` that we created has value

```
function actionFunction (action)
```

i.e,

```
dispatch = actionFunction(action)
```

4.7 And we create `store` with the new `dispatch` .

Below you can see the value of `next` and `dispatch`

```
❶ In next function
  ▶ function dispatch(action)
❷ ~~~new dispatch AFTER compose i.e composedFunc RETURNED
  ▶ function actionFunction(action)
> |
```

next and new value of dispatch

Now we have gone through all the logs in the console. And I hope you have a clear understanding on what has happened till now?

Let me summary what happened till now:-

1. We have the original dispatch which when we createStore in next. i.e dispatch(action).

```
next = dispatch(action)
```

2. And after we created the store and we created a new version of store with dispatch which will call actionFunction(action).

```
store.dispatch = actionFunction(action)
```

Awesome! Now let us click on **Load Async Action** button and see the rest of the magic unfold :-)

. . .

On clicking the button:

Below is the logging once you click on the button:

```

1 ++++++startLoad function Called+++++++
1 action RETURNED
  ▶ function sideEffectFunction(dispatch)
1 ++++++
1 action is function
1 ++++++
1 ~~~~~dispatching requestLoad ~~~~~
1 action RETURNED
  ▶ Object
1 ++++++calling NEXT+++++++
  ▶ function dispatch(action)
  ^^^^^^^^^^action.type^^^^^^^^^^
  === Entered LoadAPI ===
1 ~~~~~dispatching receiveLoad ~~~~~
1 action RETURNED
  ▶ Object
1 ++++++calling NEXT+++++++
  ▶ function dispatch(action)
  ~~~~~ . . . ~~~~~

```

Now let us try to understand on what is happening here.

- The code for the button:

```
<button
  children="Load Async Action"
  className="st-btn st-btn-solid st-btn-success st-btn-sm"
  onClick={() => dispatch(startLoad())}
  disabled={!isLoading}/>
```

And let us look at all our action creators:

```
export const requestLoad = function requestLoad() {
  return {
    type: 'REQUEST_LOAD',
  };
}

export const receiveLoad = function receiveLoad(timestamp) {
  return {
    type: 'RECEIVE_LOAD',
    payload: {
      lastTimestamp: timestamp,
    }
  }
}

// Note that it's a Side Effect Function
export const startLoad = function startLoad() {
  console.info("+++++++startLoad function Called+++++")
  return function sideEffectFunction (dispatch) {

    console.info("-----dispatching requestLoad .....")
    dispatch(requestLoad()); // Sub action for REQUEST_LOAD
    return loadApi()
      .then(timestamp => {
        // Sub action for RECEIVE_LOAD

        console.info("-----dispatching receiveLoad .....")
        return dispatch(receiveLoad(timestamp))
      });
  }
}
```

And the loadApi function:-

```
function loadApi() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(new Date().getTime());
    }, 2000);
  });
}
```

```
});
}
```

- So when you click on the button,

```
1 +++++startLoad function Called+++++
1 action RETURNED
  ▶ function sideEffectFunction(dispatch)
1 +++++
1 action is function
1 +++++
```

Our `store.dispatch` is now `actionFunction(action)` .

Therefore `actionFunction` 's action parameter will be `function startLoad()`

i.e `actionFunction(action) = actionFunction(startLoad())` and it returns

and it returns

```
function sideEffectFunction (dispatch).
```

as you see in the screenshot above.

i.e `action = function sideEffectFunction (dispatch)`

- And the `sideEffectFunction` function is as below

```
if (typeof action === 'function') {
  console.info("+++++")
  console.info("action is function")
  console.info("+++++")
  return action(dispatch, getState, extraArgument);
}
console.info("+++++calling NEXT+++++")
console.dir(next)
return next(action);
```

- So since, our action returns a function, we execute the action.

```
action(dispatch, getState, extraArgument)
```


i.e `sideEffectFunction (dispatch, getState, extraArgument)`

thereby executing `sideEffectFunction` with the respective parameters.

```

1 dispatching requestLoad .....
2 action RETURNED
  ▶ Object
3 ++++++calling NEXT+++++
  ▶ function dispatch(action)
    ^^^^^^^^^action.type^^^^^^^^
    === Entered LoadAPI ===
4 dispatching receiveLoad .....
5 action RETURNED
  ▶ Object
6 ++++++calling NEXT+++++
  ▶ function dispatch(action)
    ^^^^^^^^^action.type^^^^^^^^

```

- When `sideEffectsFunction` executes, it will
`dispatch(requestLoad())`
- And since this action creator returns an action object, it will execute `next(action)` where `next` is `dispatch(action)` from `createStore`
- And it will pass the value to the `reducer` .
- Next it executes `loadAPI()` and if success, then we
`dispatch(receiveLoad())` and since this is also returning action object, we will execute `next(action)`
- And the values will be passed to the `reducer` .

. . .

Conclusion

Once you break down the code and analyze both `applyMiddleware` and `createThunkMiddleware` together, it becomes much easier to follow what is happening.

I have put the flowchart earlier so that its easier for you to visualize how things are working together.

Basically we are introducing the intermediate step `(next)` which will be called to pass the values to the reducer only if the action creator returns an action object.

Hope this helps you in understanding how thunk works better :-)