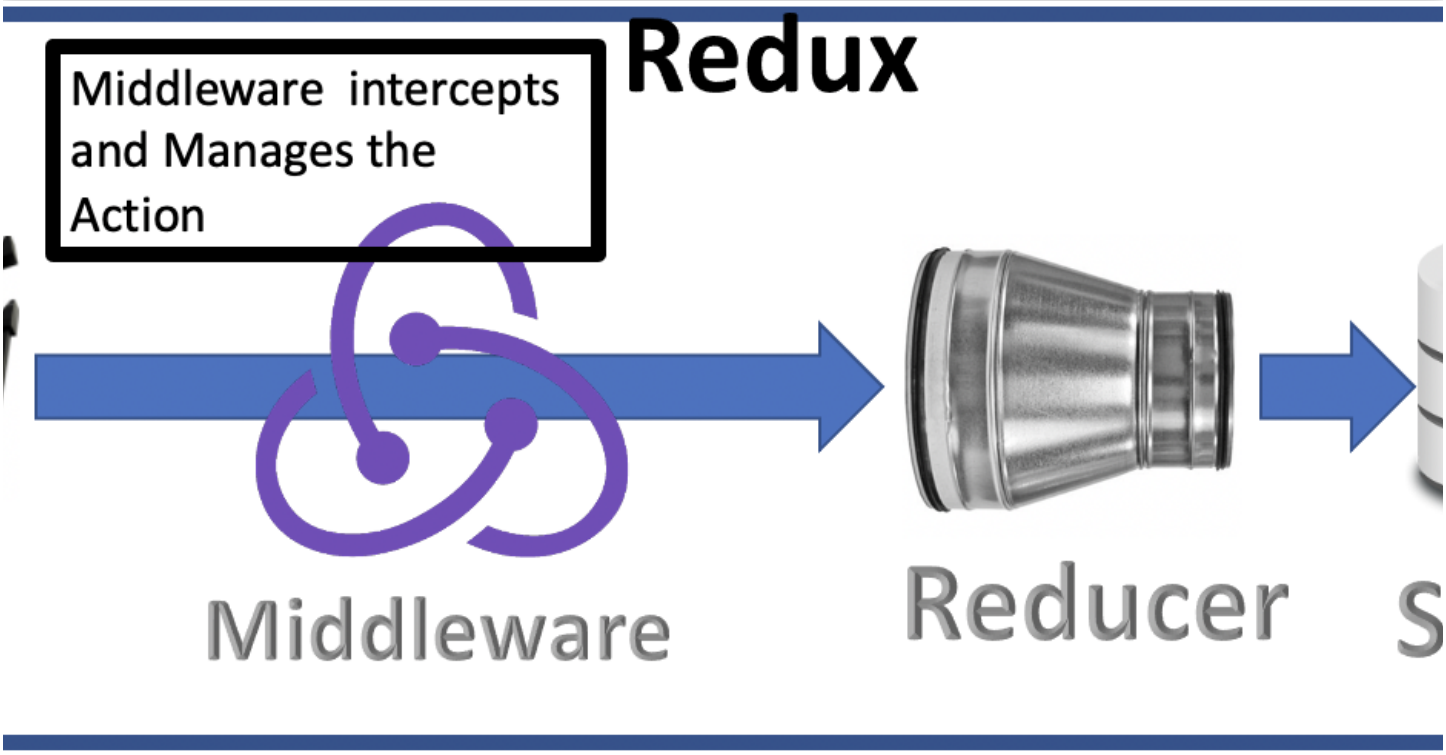


Understanding Middleware in Redux

Rany ElHousieny , PhDABD
SENIOR MANAGER OF SOFTWARE ENGINEERING, AWS SOLUTIONS ARCHITECT CERTIFIED®, PSM®, ACSA® , MIS® , UPE®, MSCA®
Published Jan 12, 2021

In this article, I will explain the concept of "Middleware" #middleware in Redux #redux

This Article is a part of a series

1. ***Create a Redux (Hello World)***: https://lnkd.in/guBuBVP

2. Create React Native App here https://lnkd.in/gDWG7AC

3. **Redux with React Native:** 1-Reading State: https://lnkd.in/gUbBRs8

5. This Article: Understanding Middleware in Redux: https://lnkd.in/g5tVFn8

and also published in Medium https://ranyel.medium.com/understanding-middleware-in-redux-9d664cc6f5a5

Final Code for the App can be found in Github https://github.com/ranyelhousieny/ReactNativeFoodApplication

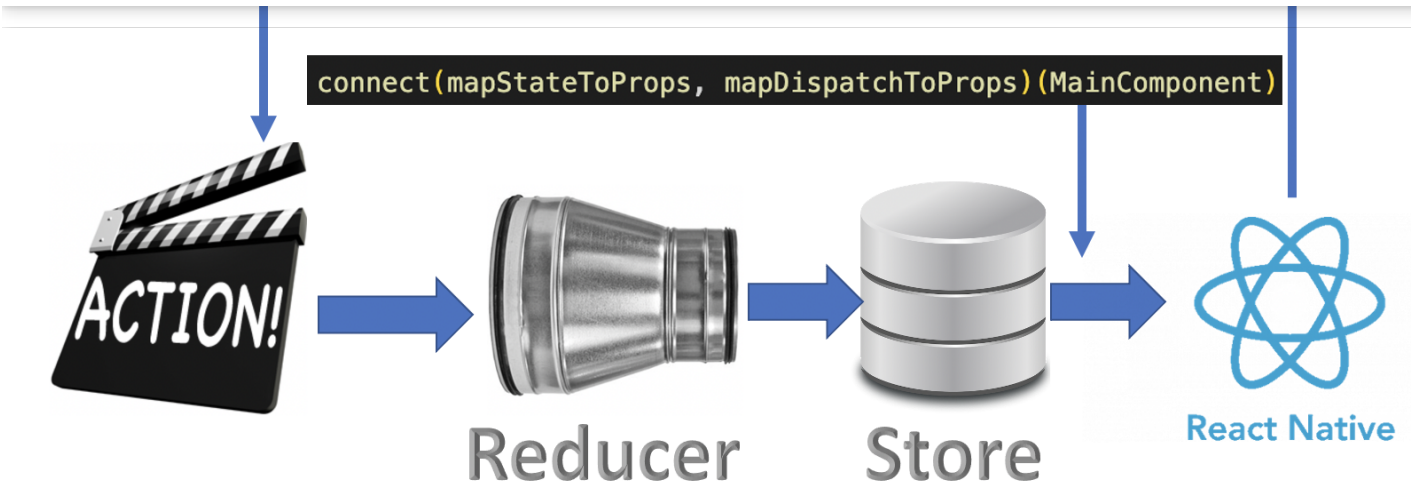
you can clone and follow along using

```
git clone https://github.com/ranyelhousieny/ReactNativeFoodApplication.
```

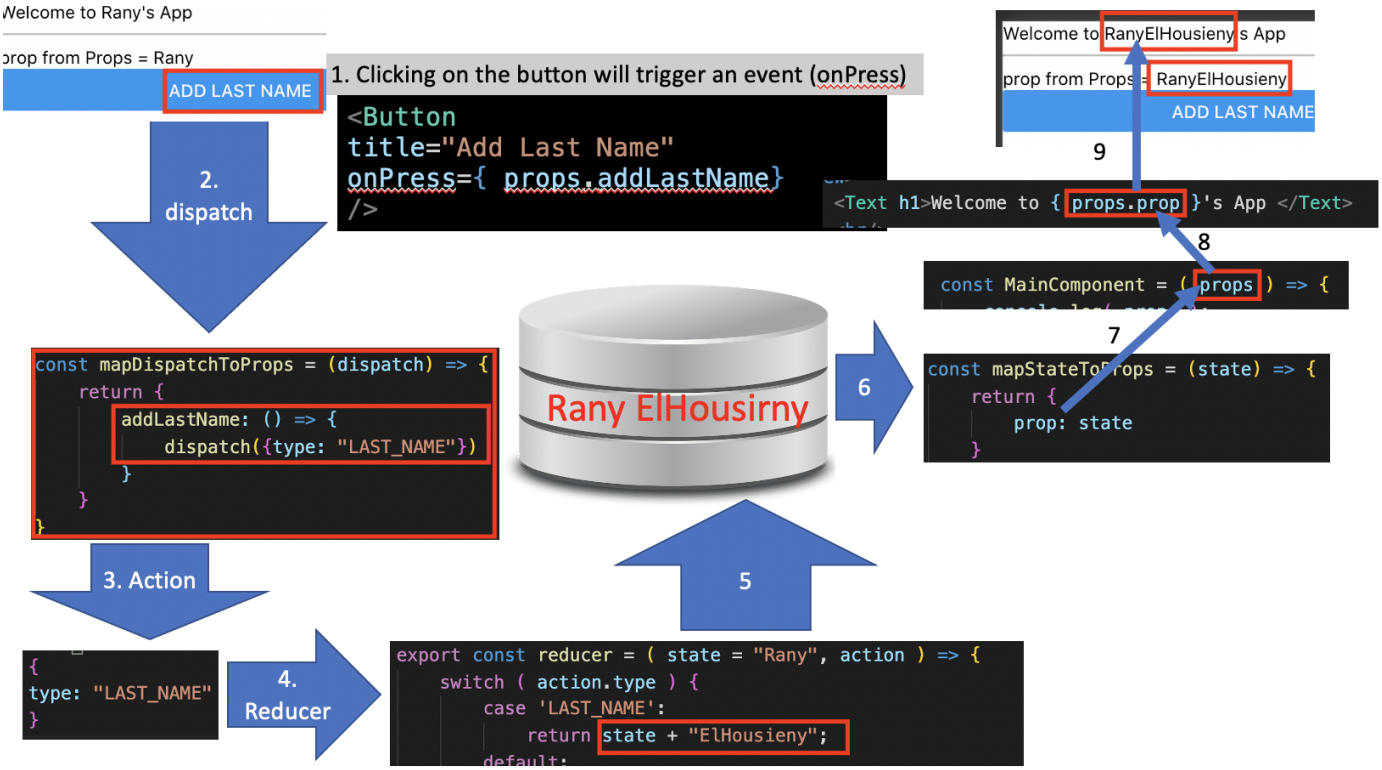
==========

I will continue from what we left in previous article. We now have the following diagram:

Rany ElHousieny , PhD[ABD]



And here are the steps for dispatch an action:

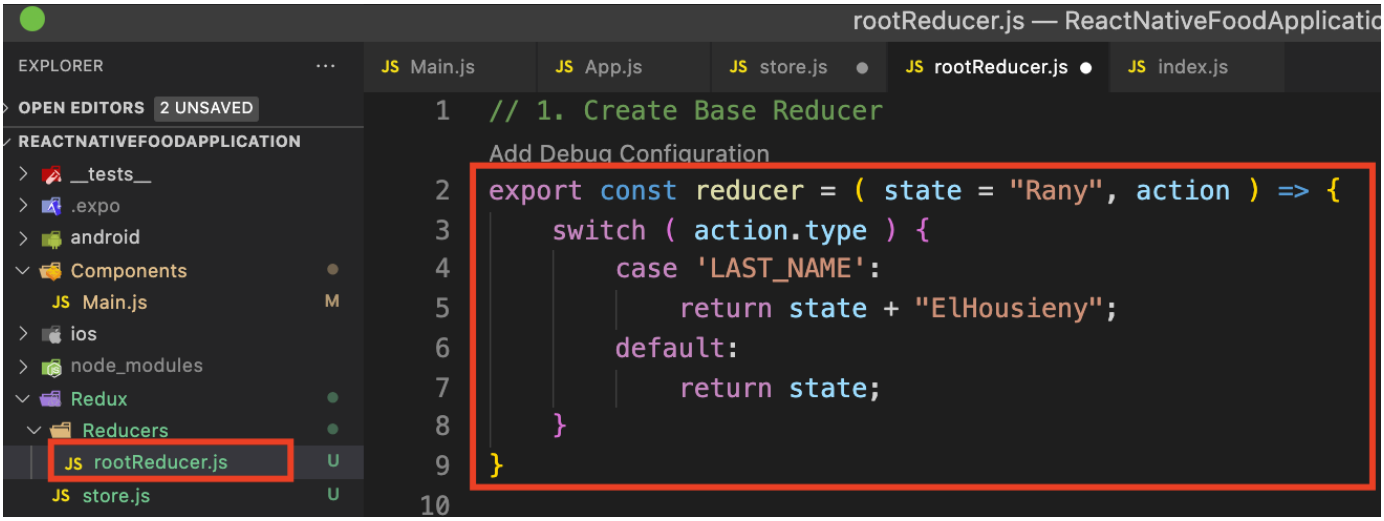


1. An event when click the button will dispatch an action through the props as setup by mapDispatchToProps

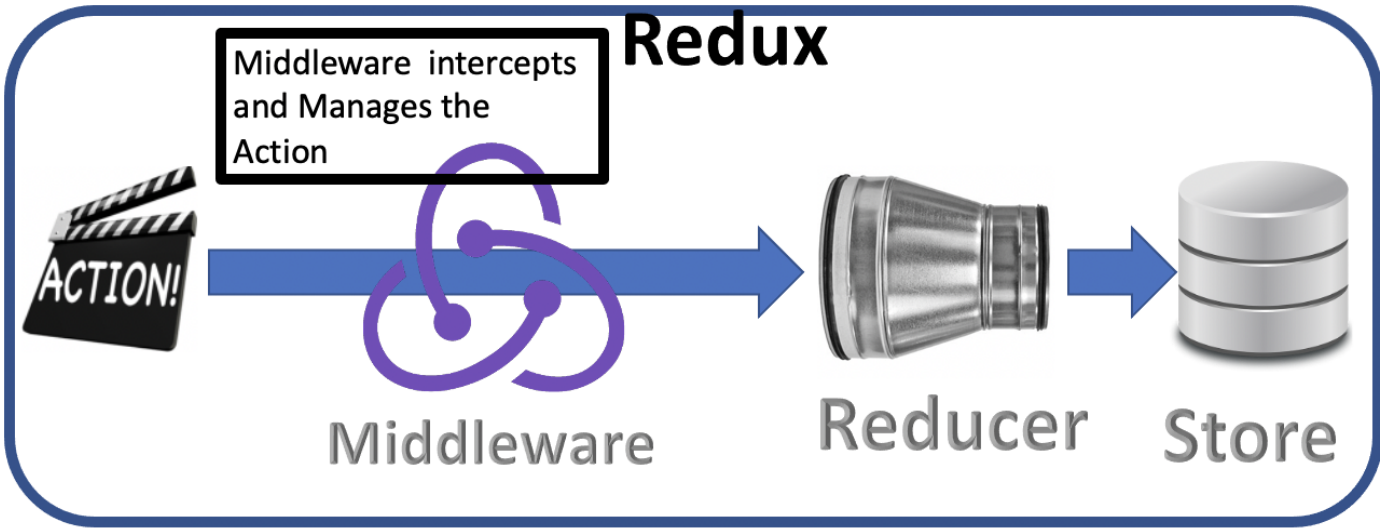
2. Building an action with type only in this case

```
    type: "LAST_NAME"
}
```

3. This action will be sent to the reducer that will go through a switch statement to decide what will be the state related to this action.type. In this case add last name (Rany ElHouieny)



Here is where the Middleware comes before step 3 to intercept the action and manage it before sending it to the reducer. as seen below:



in this scenario it might make much sense. However, in real life, the action will mostly try to fetch data from a server. fetching data from a server in an async call, which we do not want to wait for the response. Waiting for the response will cause big delay on the display. Instead, we fire the action and release the caller until we receive the response back.

To be able to explain the concept clearly, I will try to build a small middle ware with some console logging to show when the middleware get injected. All of this magic is done by Redux when we call applyMiddleware.

1. import applyMiddleware from redux

in Redux/store.js import the applyMiddleware from redux

```
import { createStore, applyMiddleware } from 'redux';
```

Create a logging middleware

In Redux/store.js file and before creating the store, let's create a simple logging middleware that will print the state along the way. a valid middleware consists of 3 nested functions as follows

The first function, takes the store as a parameter since it will work with the store.

```
};
```

This first function will return a second function that takes "next" parameter, because it will be the next step to the reducer.

```
// Logging Middleware
const logging = ( store ) => {
    return ( next ) => {

    };
};
```

Just to make it more confusing :) , the second function will return a third function that will receive an action as a parameter.

```
// Logging Middleware
const logging = ( store ) => {
    return ( next ) => {
        return ( action ) => {

        };
    };
};
```

This is a valid middleware, but very simple middleware. Let's add some code to it. All the code will be in the inner function since it has access to the store, next, and action.

Let's first log the value of the action

```
    return ( next ) => {
        return ( action ) => {
            console.log( "[inside logging],  action = ", action );
            console.log( "[inside logging],  State before next = ", st
        };
    };
};
```

Now, let's send the action to the reducer using the middle function next and log the store value after it returns as follows:

```
// Logging Middleware
const logging = ( store ) => {
    return ( next ) => {
        return ( action ) => {
            console.log( "[inside logging],  action = ", action );
            console.log( "[inside logging],  State before next = ", st
            const result = next( action );
            console.log( "[logging], store value after next", store.ge
            return result;
        };
    };
};
```

Now, let's apply this middleware to the store to allow it to intercept the dispatched actions as follows

```
export const store = createStore( reducer, applyMiddleware(logging) );
```

```
const logging = ( store ) => {
    return ( next ) => {
        return ( action ) => {
            console.log( "[inside logging],  action = ", action );
            console.log( "[inside logging],  State before next = ", store.getState() );
            const result = next( action );
            console.log( "[logging], store value after next", store.getState() );
            return result;
        };
    };
};

// 2. Create the store
export const store = createStore( reducer, applyMiddleware(logging) );
```

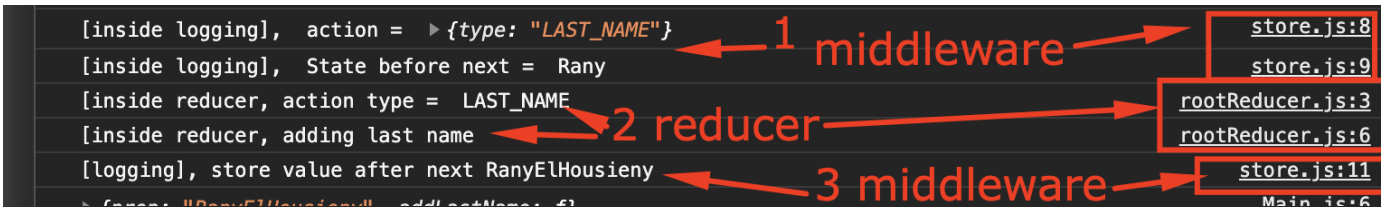
Let's run the app and watch logging (npm run web)

```
[inside logging],  action =  ▼{type: "LAST_NAME"}
                               type: "LAST_NAME"
                             ▶__proto__: Object
[inside logging],  State before next =  Rany
[logging], store value after next RanyElHousieny
```

As you can see the state was Rany before next and then Rany ElHousieny after next, which sent it to the reducer to add last name. Let's add some logging in the reducer at Redux/reducers/rootReducer.js to validate this.

```
        case 'LAST_NAME':
            console.log( '[inside reducer], adding last name ');
            return state + "ElHousieny";
        default:
            return state;
    }
}
```

And here is the logging



1. Middleware
2. Middleware calls next to call the reducer and passed the action that can be modified before sending
3. Reducer modifies the store according to action type and return it to middleware

This is simply how middleware works. in the next article I will add another known middleware (Thunk) but before that, let's arrange our code and introduce "Action Creators"

Action Creator

So far, actions have been inside mapDispatchToProps function.

As you can see, we are returning the action in the dispatch. Let's take the action out and call it from the dispatch to be able to see how middleware intercepts Actions

```
const addLastNameActionCreator = () => {
    console.log("[inside addLastNameActionCreator]")
    return {
        type: "LAST_NAME"
    }
}

const mapDispatchToProps = (dispatch) => {
    return {
        addLastName: () => {
            dispatch( addLastNameActionCreator() );
        }
    }
}
```

I added numbers and logging and you can see the full sequence below

Rany ElHousieny , PhD[ABD]

```
4. [inside reducer], action type =  LAST_NAME                    rootReducer.js:3
5. [inside reducer], adding last name                            rootReducer.js:6
6. [logging], store value after next RanyElHousieny              store.js:11
```

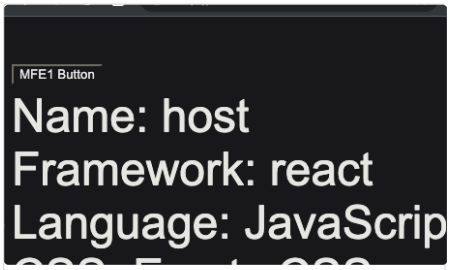

As you can see that we pass by the mapDispatchToProps first and then action creator second

In the next article we will build on that to understand Thunk

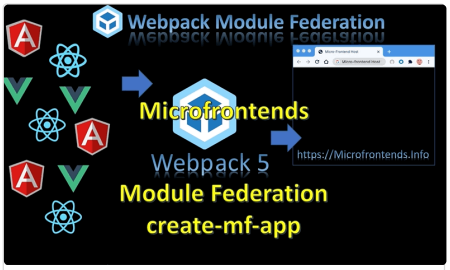
@facebook

More articles by this author

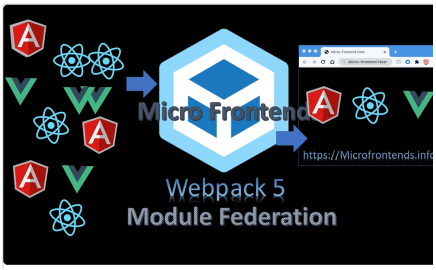
See all



Microfrontends With Module Federation:...
Oct 16, 2021



Microfrontends with Module Federation...
Oct 16, 2021



Microntends with Module Federation:...
Oct 13, 2021