Middleware

One of the more interesting concepts in Redux is you can include custom middleware functions to the dispatch method of your store. However, when I first examined the applyMiddleware.js source file I really hurt my head. So I went back to the Redux middleware documentation to see if I could align my understanding of the source code with it's intent. I felt a little better after reading this at the end of the section:
Middleware sounds much more complicated than it really is. The only way to really understand middleware is to see how the existing middleware works, and try to write your own. The function nesting can be intimidating, but most of the middleware you'll find are, in fact, 10-liners, and the nesting and composability is what makes the middleware system powerful.

These 10 line middleware functions are easy to write but require some explanation if you want to understand how they fit into a middleware chain and replace the store's dispatch method. First let's briefly define what middleware is and then reverse-engineer the source code with a simple piece of middleware. The most concise definition I can give to describe middleware is this:

Middleware is created by composing functionality that wraps separate cross-cutting concerns which are not part of your main execution task.

Well that sounds very simple ;)

It is likely that you have used middleware already if you have experimented with Koa.js. My first encounters with middleware was when I was programming Java Servlet Filters and Rack with Ruby to address cross-cutting concerns like authentication,

authorization, logging, gathering performance metrics, or some other decoration before the main execution of the task is handled.

In the case of Redux middleware the main execution task is the store's dispatch function. The dispatch function is responsible for sending actions to one or many reducer functions for state changes. The composed specialized functions around the original dispatch method creates the new middleware capable dispatch method. Here is the source for applyMiddleware (from Redux 1.0.1) which we will be examining:

```
export default function applyMiddleware(...middlewares) {
  return (next) =>
    (reducer, initialState) => {
      var store = next(reducer, initialState);
      var dispatch = store.dispatch;
      var chain = [];
      var middlewareAPI = {
        getState: store.getState,
        dispatch: (action) => dispatch(action)
      };
      chain = middlewares.map(middleware =>
                    middleware(middlewareAPI));
      dispatch = compose(...chain, store.dispatch);
      return {
        ...store,
        dispatch
      };
   };
}
```

In this small amount of code there are tons of functional concepts littered in here including high-order functions, function composition, currying, and ES6 syntax. I read this about 10 times the first time I saw it and then everything just went blurry :). Let's

touch on a few functional concepts before getting back to this code so that does not happen to you.

Brief Functional Programming Concepts

Before we can reverse-engineer this Redux middleware code you might need a little more foundational functional programming knowledge. You can skip this section if you are already familiar with these concepts.

Composing Functions

Functional programming is very literal and very mathematical. In the case of composing functions with math you can express two or more functions like this:

```
given:
f(x) = x^2 + 3x + 1
g(x) = 2x
then:
(f ∘ g)(x) = f(g(x)) = f(2x) = 4x^2 + 6x + 1
```

It is no coincidence that you can compose two or more functions in a similar fashion. Here is a very simple example of a function that composes two functions to return a new specialized function:

```
var greet = function(x) { return `Hello, ${ x }` };
var emote = function(x) { return `${x} :)` };
var compose = function(f, g) {
  return function(x) {
    return f(g(x));
  }
}
var happyGreeting = compose(greet, emote);
// happyGreeting("Mark") -> Hello, Mark :)
```

Of course we will want the ability to compose more than two functions together. This was just to illustrate the basic concept. We will look at a more generic way to solve that issue by examining the Redux code.

Currying

Another powerful functional programming concept is the idea of currying or partially applying argument values to a function. By currying we can create a new specialized function that has partial information supplied to it. Here is the canonical example of currying where we have an add function that curries the first operand parameter to create a specialized add function:

```
var curriedAdd = function(a) {
    return function(b) {
        return a + b;
    };
};
var addTen = curriedAdd(10);
addTen(10); //20
```

By currying and composing your functions you can create powerful new functions that create a pipeline for data processing.

Spoiler alert: This is pretty much all there is to creating middleware in Redux.

Redux Dispatch Function

A Store in Redux have a dispatch function which is only concerned with the main execution task you are interested in. You dispatch actions to your reducer functions to update state of the application. Redux reducer functions take a state and action parameter and return a new resultant state:

```
reducer:: state -> action -> state
```

You might dispatch an action that simply sends a message to remove an item in a list which could look like this:

```
{type: types.DELETE_ITEM, id: 1}
```

The store will dispatch this action object to all of it's reducer functions which could affect state. However, the reducer functions are only concerned with executing logic around this deletion. They typically don't care who did it, how long it took, or logging the before and after effects of the state changes. This is where middleware can help us to address these non-core concerns.

Redux Middleware

Redux middleware is designed by creating functions that can be composed together before the main dispatch method is invoked. Let's start by creating a very simple logger middleware function that can echo the state of your application before and after

running your main dispatch function. Redux middleware functions have this signature:

```
middleware:: next -> action -> retVal
```

It might look something like this:

```
export default function createLogger({ getState }) {
  return (next) =>
    (action) => {
      const console = window.console;
      const prevState = getState();
      const returnValue = next(action);
      const nextState = getState();
      const actionType = String(action.type);
      const message = `action ${actionType}`;
      console.log(`%c prev state`, `color: #9E9E9E`, prevState);
      console.log(`%c action`, `color: #03A9F4`, action);
      console.log(`%c next state`, `color: #4CAF50`, nextState);
      return returnValue;
    };
}
```

Notice that *createLogger* accepts the *getState* method which is injected by applyMiddleware.js and used inside the inner closure to read the current state. This will return a new function with the *next* parameter which is used to compose the next chained middleware function or the main dispatch function. This function returns a curried function that accepts the *action* object which can be read or modified before sending it to the next middleware function in the chain. Finally, the main dispatch function is invoked with the action object.

A much more robust implementation of logger middleware for Redux can be found here. I blatantly cheated a lot and trimmed

down this implementation by making a lot of bad assumptions so I could save time and show the basics.

On a side note the logger middleware provides around advice because it does the following:

- First it captures the previous state
- The action is dispatched to the next middleware function
- All downstream middleware functions in the chain are invoked
- The reducer functions in the store are called with the action payload
- The logger middleware then gets the resulting next state

Here is an example of what around advice looks like visually when you have two middleware components that decorate an action before going through the main dispatch execution:

```
nameMiddleware before - action:  Object {type: "ADD_TODO", text: "Learn Redux!"}  state:
▼Object {todos: Array[1]}
  ▼todos: Array[1]
    ▼0: Object
       completed: false
       id: 0
       text: "Use Redux"
      ▶__proto__: Object
      length: 1
    ▶__proto__: Array[0]
  ▶__proto__: Object
timeMiddleware before - action:  Object {type: "ADD_TODO", text: "Learn Redux!", name: "mark"}  state:
▶Object {todos: Array[1]}
Adding todo in reducer:  ADD_TODO  :  Learn Redux!
timeMiddleware after - action:
▶Object {type: "ADD_TODO", text: "Learn Redux!", name: "mark", time: Wed Aug 26 2015 07:36:04 GMT-0400 (EDT)}  state:
▶Object {todos: Array[2]}
nameMiddleware after - action:
▶Object {type: "ADD_TODO", text: "Learn Redux!", name: "mark", time: Wed Aug 26 2015 07:36:04 GMT-0400 (EDT)}  state:
▼Object {todos: Array[2]}
  ▼todos: Array[2]
    ▼0: Object
       completed: false
       id: 1
       text: "Learn Redux!"
      ▶__proto__: Object
    ▼1: Object
       completed: false
       id: 0
       text: "Use Redux"
      ▶__proto__: Object
      length: 2
    ▶__proto__: Array[0]
  ▶__proto__: Object
```

Dissecting applyMiddleware.js

Now that we know what a Redux middleware function looks like and we have enough functional programming knowledge, let's put on our surgical gloves and open up the source code again to understand what is going on. Hopefully it looks a little clearer this time. Here it is again as reference:

```
export default function applyMiddleware(...middlewares) {
  return (next) =>
    (reducer, initialState) => {
      var store = next(reducer, initialState);
      var dispatch = store.dispatch;
      var chain = [];
      var middlewareAPI = {
        getState: store.getState,
        dispatch: (action) => dispatch(action)
      };
```

```
      chain = middlewares.map(middleware =>
                        middleware(middlewareAPI));
      dispatch = compose(...chain, store.dispatch);
      return {
        ...store,
        dispatch
      };
    };
}
```

The applyMiddleware function probably could have been named a little better. What are you applying middleware to exactly? I think this should be a little more explict and named something like applyMiddlewareToStore. What else would you be applying this middleware to?

We will now incise each line and knock out some ES6 syntax along the way. First we have the method signature:

```
export default function applyMiddleware(...middlewares)
```

Nothing too interesting here except we have the *middlewares* argument with a spread operator on it. This will allow us to pass in as many middleware functions that we want. Next we will return a function that takes a mysterious *next* argument:

```
return (next) => (reducer, initialState) => {...}
```

The *next* argument will be a function that is used to create a store. By default you should look at the implementation for [createStore.js](). The final returned function will be like createStore and replaces the dispatch function with it's associated middleware.

Next we assign the store implementation to the function responsible for creating the new store (again see createStore.js). Then we create a variable to the store's original dispatch function. Finally, we setup an array to hold the middleware chain we will be creating.

```
var store = next(reducer, initialState);
var dispatch = store.dispatch;
var chain = [];
```

This next bit of code injects the getState function and original dispatch function from the store into each middleware function which you can optionally use in your middleware. The resultant middleware is stored in the chain array:

```
var middlewareAPI = {
  getState: store.getState,
  dispatch: (action) => dispatch(action)
};
chain = middlewares.map(middleware =>
                     middleware(middlewareAPI));
```

Now we create our replacement dispatch function with the information about the middleware chain.

```
dispatch = compose(...chain, store.dispatch);
```

The magic to composing our middleware chain lies in this utility function supplied by Redux. Here is the implementation:

```
export default function compose(...funcs) {
 return funcs.reduceRight((composed, f) => f(composed));
}
```

Yep that is it! The compose function will literally express your functions as a composition injecting each middleware as an argument to the next middleware in the chain. Order is important here when assembling your middleware functions. Finally, the original store dispatch method is composed. This new looks something like this:

```
middlewareI(middlewareJ(middlewareK(store.dispatch)))(action)
```

See why we needed to talk about composition and currying now? The final thing to do is return the new store object with the overridden dispatch function:

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

There is that spread operator again. This spreads out the store object which includes the original *dispatch* function. Since we specify *dispatch* at the end it will be extended into the new store object which was the original intent. Here is what it looks like from Babel's perspective:

```
return _extends({}, store, { dispatch: _dispatch });
```

Let's add our logger middleware we started above into a custom store with the enhanced dispatch function. Here is how you could do this:

```
import { createStore, applyMiddleware } from 'redux';
import loggerMiddleware from 'logger';
import rootReducer from '../reducers';
```

```
const createStoreWithMiddleware =
  applyMiddleware(loggerMiddleware)(createStore);
export default function configureStore(initialState) {
  return createStoreWithMiddleware(rootReducer, initialState);
}
const store = configureStore();
```

Asynchronous Middleware

Once you get comfortable with the basics of Redux middleware you will likely want to work with asynchronous actions that involve some sort of asynchronous execution. In particular look at redux-thunk for more details. Let's say you have an action creator that has some async functionality to get stock quote information:

```
function fetchQuote(symbol) {
   requestQuote(symbol);
   return
fetch(`http://www.google.com/finance/info?q=${symbol}`)
      .then(req => req.json())
      .then(json => showCurrentQuote(symbol, json));

}
```

There is no obvious way here to dispatch an action that would be returned from the fetch which is Promise based. Plus we do not have a handle to the dispatch function. Therefore, we can use the redux-thunk middleware to defer execution of these operations. By wrapping the execution in a function you can delay this execution.

```
function fetchQuote(symbol) {
  return dispatch => {
    dispatch(requestQuote(symbol));
    return
fetch(`http://www.google.com/finance/info?q=${symbol}`)
      .then(req => req.json())
      .then(json => dispatch(showCurrentQuote(symbol, json)));
```

```
  }
}
```

Remember that the applyMiddleware function will inject the *dispatch* and the *getState* functions as parameters into the redux-thunk middleware. Now you can dispatch the resultant action objects to your store which contains reducers. Here is the middleware function for redux-thunk that does this for you:

```
export default function thunkMiddleware({ dispatch, getState })
{
  return next =>
     action =>
       typeof action === 'function' ?
         action(dispatch, getState) :
         next(action);
}
```

This should be familiar to you now that you have already seen how Redux middleware works. If the action is a function it will be called with the *dispatch*and *getState* function. Otherwise, this is a normal action that needs to be dispatched to the store. Also check out the Async example in the Redux repo for more details. Another middleware alternative for working with Promises in your actions is redux-promise. I think it is just a matter of preference around which middleware solution you choose.