FREE eBook

LEARNING redux

Free unaffiliated eBook created from
Stack Overflow contributors.

#redux

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: redux

It is an unofficial and free redux ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official redux.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with redux

Remarks

Redux is a JavaScript library that implements state container of the Flux-based architecture.

Redux can be described in three fundamental principles:

1. Single source of truth (Single store)
2. State is read only (need actions to signal change)
3. Changes are made with pure functions (This creates new state according to the actions)

Main parts:

- store constructor
- store.dispatch(action)
- middleware
- reducers

Redux is astonishingly simple. It uses a function called a reducer (a name derived from the JavaScript reduce method) that takes two parameters: An action, and a next state.

The reducer has access to the current (soon to be previous) state, applies the given action to that state, and returns the desired next state.

Reducers are designed to be pure functions; meaning, they produce no side effects. If you pass the same input values to a reducer 100 times, you will get the exact same output value 100 times. Nothing weird happens. They are completely predictable. As someone with a prominent "NO SURPRISES" sticky note on my monitor, this is a wonderful idea to contemplate.

Reducers do not store state, and they do NOT mutate state. They are passed state, and they return state. This is what reducers look like in action. http://redux.js.org/docs/basics/Reducers.html

Reference: http://redux.js.org/docs/introduction/Motivation.html

Versions

Versions	Release Date
v3.6.0	2016-09-04
v3.5.0	2016-04-20
v3.4.0	2016-04-09
v3.3.0	2016-02-06

Versions	Release Date
v3.2.0	2016-02-01
v3.1.0	2016-01-28
v3.0.0	2015-09-13
v2.0.0	2015-09-01
v1.0.0	2015-08-14

Examples

Installation or Setup

Basic installation:

You can download redux javascript file, using this link.

Also you can install redux, using bower :

```
bower install https://npmcdn.com/redux@latest/dist/redux.min.js
```

Next, you need to include redux to your page:

```
<html>
  <head>
    <script type="text/javascript" src="/path/to/redux.min.js"></script>
  </head>
  <body>
    <div id="app"></div>
    <script>
      // Redux is available as `window.Redux` variable.
    </script>
  </body>
</html>
```

Npm installation:

If you are using npm, to install redux, you need to run:

```
npm install redux --save
```

Next, to use redux, you need to require it (assuming you are using module bundler, like webpack):

```
var redux = require('redux');
```

Or if you are using es6 transpiler, like babel:

```
import redux from 'redux';
```

Vanilla Redux Example (without React or others)

You can see the running demo by clicking here.

HTML:

```
<p>
  <span>Counter State</span><br />
  (<em>Will increase each minute</em>):
  <p>
    <span id="counter-state" style="font-weight: bolder"></span>
  </p>
</p>

<p>
  <button id="increment-action">+ Increase +</button>
  <button id="decrement-action">- Decrease -</button>
</p>
```

REDUX LOGIC:

```
// ------------------------ reducer helpers ------------------------
let reducers = {}

let addReducer = (reducers, actionType, reducer) =>
  reducers[actionType] = (state, action) => {
    if (action.type == actionType) {
      return reducer(state, action)
    }
  }

let reducer = (state, action) => {
  if (reducers[action.type]) {
    return reducers[action.type](state, action)
  }
  return state
}


// ------------------------ redux setup ------------------------

const {
  createStore,
  applyMiddleware
  } = Redux


// apply logging middleware (not necessary)
// open the console to see the action logger output on each time new action dispatched
const actionLogger = ({ dispatch, getState }) => next => action => {
  console.log("action logger: action.type=%s state=%d", action.type, getState())
  return next(action)
}
```

```
// ------------------------ reducers ------------------------
// those will be creating new states and returning it,
// depending on the dispatched actions
addReducer(reducers, 'INCREMENT', (state, action) => ++state)
addReducer(reducers, 'DECREMENT', (state, action) => --state)


const DEFAULT_STATE = 0

const store = createStore(
  reducer,
  DEFAULT_STATE,
  applyMiddleware(actionLogger)
);


console.log(createStore)


// ------------------------ rendering ------------------------
let render = () => document.getElementById('counter-state').innerHTML = store.getState()

//
// IMPORTANT BINDING:
//
// store will be dispatching the new state to the render method each time the state changes
//
store.subscribe(render)

//
// render with the state for the first time
//
render()



// ------------------------ redux actions ------------------------

// continously increment the counter (the state) each second
setInterval(() => store.dispatch({type: 'INCREMENT'}), 1000)

// only increment the counter on click to the increase button
document
  .getElementById('increment-action')
  .addEventListener('click', () => store.dispatch({type: 'INCREMENT'}))

// only decrement the counter on click to the decrease button
document
  .getElementById('decrement-action')
  .addEventListener('click', () => store.dispatch({type: 'DECREMENT'}))
```

Read Getting started with redux online: https://riptutorial.com/redux/topic/1101/getting-started-with-redux

Chapter 2: Asynchronous Data Flow

Examples

Redux-thunk: basics

While redux itself is entirely synchronous, you can use a middleware such as `redux-thunk` to handle asynchronous actions.

> A "thunk" is another name for a callback. It is a function that is usually passed as an argument to be called at a later time.

To use, apply the middleware to your redux store:

```
import ReduxThunk from 'redux-thunk';

const store = createStore(
    reducer,
    applyMiddleware(ReduxThunk)
);
```

This allows you to pass a thunk to `dispatch` instead of a plain object. The middleware will recognize the thunk and call it. The thunk takes the store's `dispatch` method as a parameter:

```
// an asynchronous action - "thunk"
// This will wait 1 second, and then dispatch the 'INCREMENT' action
const delayedIncrement = dispatch => setTimeout(() => {
    dispatch({
        type: 'INCREMENT'
    });
}, 1000);

// dispatch the thunk.
// note: no () as we're passing the function itself
store.dispatch(delayedIncrement);
```

Using redux-thunk with jQuery.ajax

```
const loadUser = userId => dispatch => {
    dispatch({ type: 'USER_LOADING' });
    $.ajax('/users/' + userId, {
        type: 'GET',
        dataType : 'json'
    }).done(response => {
        dispatch({ type: 'USER_LOADED', user: response });
    }).fail((xhr, status, error) => {
        dispatch({ type: 'USER_LOAD_ERROR', status, error });
    });
};
```

To use, dispatch like any other action creator:

```
store.dispatch(loadUser(123));
```

This will result in an initial `USER_LOADING` action being dispatched, which can be used to display a loading indicator (if so desired), and after the response has been received either a `USER_LOADED` action or `USER_LOAD_ERROR` action will be dispatched, depending on the result of the `$.ajax` request.

Middleware

When you call `store.dispatch(actionObject)` it is handled synchronously. I.e. reducers would be called and your store listeners would be notified, your react views would be re-rendered on each dispatched action.

Middleware is what enables you to delay dispatching or even dispatch different actions in the middle. I.e. middleware makes your asynchronous actions look synchronous.

```
const myAsyncMiddleware = (store) => {
    return (next) => {
        return (action) => {
            if(action.type === "ASYNC_ACTION") {
              setTimeout(() => {
                store.dispatch({ type: "ASYNC_ACTION_RESPONSE" });
              }, 1000);
            } else {
              return next(action);
            }
        }
    }
}

const store = createStore(
  reducer,
  applyMiddleware(myAsyncMiddleware)
);
```

Action creators

Another approach to handling asynchrony in Redux is to use action creators. In Flux, action creators are special functions that construct action objects and dispatch them.

```
myActionCreator(dispatch) {
  dispatch({ type: "ASYNC_ACTION_START" });
  setTimeout(() => {
    dispatch({ type: "ASYNC_ACTION_END" });
  }, 1000)
}
```

Use custom middleware + Superagent

This is example has extracted from this boilerplate.

Custom middleware:

```
export default function clientMiddleware() {
  return ({dispatch, getState}) => {
    return next => action => {
      if (typeof action === 'function') {
        return action(dispatch, getState);
      }

      const { promise, types, ...rest } = action; // eslint-disable-line no-redeclare
      if (!promise) {
        return next(action);
      }

      const [REQUEST, SUCCESS, FAILURE] = types;
      next({...rest, type: REQUEST});

      const client = new ApiClient();
      const actionPromise = promise(client);
      actionPromise.then(
        (result) => next({...rest, result, type: SUCCESS}),
        (error) => next({...rest, error, type: FAILURE})
      ).catch((error)=> {
        console.error('MIDDLEWARE ERROR:', error);
        next({...rest, error, type: FAILURE});
      });

      return actionPromise;
    };
  };
}
```

Wrapping `Superagent` library for API call:

```
import superagent from 'superagent';
import config from '../config';

const methods = ['get', 'post', 'put', 'patch', 'del'];

function formatUrl(path) {
  const adjustedPath = path[0] !== '/' ? '/' + path : path;
  return adjustedPath;
}

export default class ApiClient {
  constructor(req) {
    methods.forEach((method) =>
      this[method] = (path, { params, data } = {}) => new Promise((resolve, reject) => {
        const request = superagent[method](formatUrl(path));

        if (params) {
          request.query(params);
        }

        if (data) {
          request.send(data);
        }

        request.end((err, { body } = {}) => err ? reject(body || err) : resolve(body));
      }));
  }
  empty() {}
```

```
}
```

Using redux-thunk with Promises

```
import 'whatwg-fetch';

function checkStatus(response) {
  if (response.status >= 200 && response.status < 300) {
    return response;
  }
  const error = new Error(response.statusText);
  error.response = response;
  throw error;
}

function parseJSON(response) {
  return response.json();
}

function getJSON(endpoint, params) {
  return fetch(endpoint, params)
    .then(checkStatus)
    .then(parseJSON);
}

export function action() {
  return dispatch => getJSON('/example-endpoint')
    .then((result) => {
      dispatch({
        type: GET_SUCCESS,
        result,
      });
    })
    .catch((error) => {
      dispatch({ type: GET_FAILURE, error });
    });
}
```

Read Asynchronous Data Flow online: https://riptutorial.com/redux/topic/3474/asynchronous-data-flow

Chapter 3: How to link redux and react

Syntax

- `<Provider store>`
- `connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`

Parameters

Argument	Description
store	Redux store
mapStateToProps	User provided mapping: `(state, ownProps) => resultProps`

Examples

Provider

To easily link your Redux store to your React components you can use an additional library: react-redux.

First, you need to wrap your app in a `Provider`, which is a component that passes your store to be used by child components:

```
import { Provider } from 'react-redux';

/// ... store = createStore()

const App = () => (
  <Provider store={store}>
    <MyComponent>
  </Provider>
)
```

Map state to properties

After you wrapped your app into provider you can use `connect` function to subscribe your component to store changes and provide mapping between Redux state properties and React components' properties:

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

const MyComponent = ({data})  => (
  <div>{data}</div>
);
```

```
const mapStateToProps = (state, ownProps) => ({
  data: state.myComponentData
});

connect(mapStateToProps)(MyComponent);
```

Memoizing derived data

In your redux store you hold the raw data. Some times the raw data is all you need, but other times you need to derive new data from the raw data, often by combining parts of the raw data.

A common use case for deriving data is filtering a list of data based on a criteria, where both the list and the criteria may be changed.

The following example implements mapStateToProps and filters a list of strings to keep those who match a search string, producing a new prop `filteredStringList` that can be rendered by a React Component.

```
// Example without memoized selectors
const mapStateToProps(state) => {
    const {stringList, searchString} = state;

    return {
        filteredStringList: stringList
            .filter(string => string.indexOf(searchString) > -1)
    };
}
```

To keep the reducers simple you should only hold the list of data and the filtering criteria in the store, which means you need to derive the data on read time (like we did in the example above).

Deriving data on read time poses two problems:

1. If the same data is derived many times it could compromize performance.
2. If the same data is needed in different components, you may find that you are duplicating the code to derive data.

The solution is to use memoized selectors that are defined only once. The React community suggests using the npm library reselect to create memoized selectors. In the example below we achieve the same result as in the first example, only with memoized selectors.

```
// Create memoized selector for filteredStringList
import {createSelector} from 'reselect';

const getStringList = state => state.stringList;

const getSearchString = state => state.searchString;

const getFilteredStringList = createSelector(
    getStringList,
    getSearchString,
    (stringList, searchString) => stringList
        .filter(string => string.indexOf(searchString) > -1)
```

```
);

// Use the memoized selector in mapStateToProps
const mapStateToProps(state) => {
    return {
        filteredStringList: getStringList(state)
    };
}
```

Note that the two first selectors `getStringList` and `getSearchString` are *not* memoized, because they are so simple that it would provide no performance gain. They still need to be created because we need to pass them as dependencies to `createSelector`, so that it knows when to reuse the memoized result and when to compute a new result.

The memoized selector will use the function passed as the last argument passed to `createSelector` to compute the derived data (in our example the function that returns the filtered string list). Each time the memoized selector is called, if the dependencies are not changed since the last time it was called, (in our example `stringList` and `searchString`), the memoized selector will return the previous result, saving the time it would take recompute it.

You can think of selectors (memoized or not memoized) as getters for the store state, just like action-creators are setters.

You can find more examples on computing derived data in the Redux documentation's recipes section.

Read How to link redux and react online: https://riptutorial.com/redux/topic/6621/how-to-link-redux-and-react

Chapter 4: Pure Redux - Redux without any framework

Parameters

Paramenter	Description
action	It must be an object with at least the `type` property. Any other property can be passed and will be accessible within the reducer function.

Remarks

If you're not using bundlers like Webpack and Browserify, change the first line to:

```
const { createStore } = Redux;
```

Or just call it directly from the Redux global when creating the store:

```
const store = Redux.createStore(counter);
```

Examples

Full example

index.html

```
<button id="increment">Increment</button>
<button id="decrement">Decrement</button>
<p id="app"></p>
```

---

index.js

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

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;

    case 'DECREMENT':
      return state - 1;

    default:
      return state;
```

```
  }
}

const store = createStore(counter);

function render() {
  const state = store.getState();
  document.querySelector('#app').innerHTML = `Counter: ${state}`;
}

const incrementButton = document.querySelector('#increment');
const decrementButton = document.querySelector('#decrement');

incrementButton.addEventListener('click', () => {
  store.dispatch({ type: 'INCREMENT' });
});

decrementButton.addEventListener('click', () => {
  store.dispatch({ type: 'DECREMENT' });
});

store.subscribe(() => {
  render();
});

render();
```

Read Pure Redux - Redux without any framework online:
https://riptutorial.com/redux/topic/4079/pure-redux---redux-without-any-framework

Chapter 5: Reducer

Remarks

Reducers change the application state based on the actions fired.

The state is immutable, that means reducers should be pure: for the same input, you should **always** get the same output. Because of this, mutability is forbidden in reducers.

Examples

Basic reducer

A basic reducer would look like this:

```
// Import the action types to recognise them
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';


// Set up a default state
const initialState = {
    error: undefined,
    entities: []
};

// If no state is provided, we take the default state
export default (state = initilState, action) => {

    // Based on the type of action received, we calculate the new state
    switch(action.type) {

        // Set the error
        case ACTION_ERROR:
            // Note that we create a new object, copy the current state into it,
            // and then we make the relevant changes, so we don't mutate the state
            return Object.assign({}, state, { error: action.error });

        // Unset any error, and load the entitites
        case ACTION_ENTITIES_LOADED:
            return Object.assign({}, state, {
                entities: action.entities,
                error: undefined
            };

        // Add only one entity. Again, note how we make a new entities array
        // combining the previous one with the new entity
        // instead of directly modifying it, so we don't mutate the state.
        case ACTION_ENTITY_CREATED:
            return Object.assign({}, state, {
                entities: [action.entity].concat(state.entities)
            }):

        // If the action is not relevant to this reducer, just return the state
        // as it was before.
```

```
        default:
            return state;
    }
};
```

Using Immutable

Immutable is a great library that provides us with immutable versions of widely used types of collections, such as Lists, Stacks, Maps, and more.

It simplifies the manipulation of the state and makes it easier to make pure calculations and avoid mutation.

Let's see how the Basic reducer can be rewritten using Immutable's Map and List structures:

```
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';

// Import Immutable
import Immutable from 'immutable';


// Set up a default state using a Map, a structure very similar to objects
// Note that states in Redux can be anything, not just objects
const initialState = Immutable.Map({
    error: undefined,
    entities: Immutable.List()
});

export default (state = initialState, action) => {

    switch(action.type) {

        case ACTION_ERROR:
            return state.set('error', action.error);

        case ACTION_ENTITIES_LOADED:
            return state.merge({
                entities: Immutable.List(action.entities)
                error: undefined
            });

        case ACTION_ENTITY_CREATED:
            return state.set('entities', state.entities.push(action.entity));

        default:
            return state;
    }
};
```

As you may have seen, handling the immutable state gets easier using Immutable.

Basic example using ES6 spread

```
// Import the action types to recognize them
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';
```

```
// Set up a default state
const initialState = {
    error: undefined,
    entities: [],
    loading: true
};

// If no state is provided, we take the default state
export default (state = initialState, action) => {

    // Based on the type of action received, we calculate the new state
    switch(action.type) {

        // Set the error
        case ACTION_ERROR:
            // We will create new object with state,
            // which should be produced by error action
            return {
                entities: [],
                loading: false,
                error: action.error
            };

        // Unset any error, and load the entities
        case ACTION_ENTITIES_LOADED:
            return {
                entities: action.entities,
                error: undefined,
                loading: false
            };

        // Add only one entity. We will use spread operator (...) to merge
        // objects properties and to create new entity
        case ACTION_ENTITY_CREATED:
            return {
                ...state,
                entities: [action.entity].concat(state.entities)
            };

        // Every action is processed by each reducer,
        // so we need to return same state if we do not want to mutate it
        default:
            return state;
    }
};
```

Read Reducer online: https://riptutorial.com/redux/topic/6615/reducer

Chapter 6: Testing Redux apps

Examples

Redux + Mocha

Redux is very functional, so unit testing is very straightforward.

Action creator:

```
export function showSidebar () {
  return {
    type: 'SHOW_SIDEBAR'
  }
}
```

Action creators unit test:

```
import expect from 'expect'
import actions from './actions'
import * as type from './constants'

describe('actions', () => {
  it('should show sidebar', () => {
    const expectedAction = {
      type: type.SHOW_SIDEBAR
    }
    expect(actions.showSidebar()).toEqual(expectedAction)
  })
})
```

Testing a Redux store with Mocha and Chai

```
import { expect } from 'chai';
import { createStore } from 'redux';

describe('redux store test demonstration', () => {
  describe('testReducer', () => {
    it('should increment value on TEST_ACTION', () => {
      // define a test reducer with initial state: test: 0
      const testReducer = (state = { test: 0 }, action) => {
        switch (action.type) {
          case 'TEST_ACTION':
            return { test: state.test + 1 };
          default:
            return state;
        }
      });

      // create a redux store from reducer
      const store = createStore(testReducer);

      // establish baseline values (should return initial state)
```

```
      expect(store.getState().test).to.equal(0);

      // dispatch TEST_ACTION and expect test to be incremented
      store.dispatch({ type: 'TEST_ACTION' });
      expect(store.getState().test).to.equal(1);

      // dispatch an unknown action and expect state to remain unchanged
      store.dispatch({ type: 'UNKNOWN_ACTION' });
      expect(store.getState().test).to.equal(1);
    });
  });
});
```

Read Testing Redux apps online: https://riptutorial.com/redux/topic/4059/testing-redux-apps

Credits

S. No	Chapters	Contributors
1	Getting started with redux	1ven, Arijit Bhattacharya, Community, Inanc Gumus, jpdelatorre , Random User, uddhab, Vanuan
2	Asynchronous Data Flow	Ali Sepehri.Kh, Arijit Bhattacharya, Franco Risso, Ming Soon, rossipedia, Vanuan
3	How to link redux and react	ArneHugo, Vanuan
4	Pure Redux - Redux without any framework	Guilherme Nagatomo, Vanuan, Vishnu Y S
5	Reducer	Jurosh, Marco Scabbiolo
6	Testing Redux apps	Mario Tacke, Shuvo Habib, Vanuan