Redux

2019.4

손찬욱

목차

1. Redux

Redux?

Redux

개요

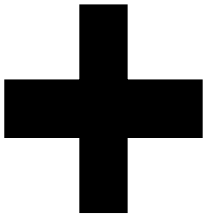
- Dan Abramov에 의해 2015년 6월경에 개발
- Facebook의 **Flux** 와 **함수형 프로그래밍 패러다임**에 영감을 받아 개발
- 상태 관리 라이브러리
- Redux is a predictable state container for JavaScript apps.

예측 가능한 상태?

- SPA의 발전으로 Rich한 클라이언트 요구사항 증가
- 많은 상태를 Javascript로 관리해야하는 상황
- 다양한 입력(사용자 입력, 서버 자원, 데이터)에 의한 **빈번한 변화**
- **비동기** 프로그래밍(Event) 방식의 Javascript

예측 가능한 상태?

빈번한 변화

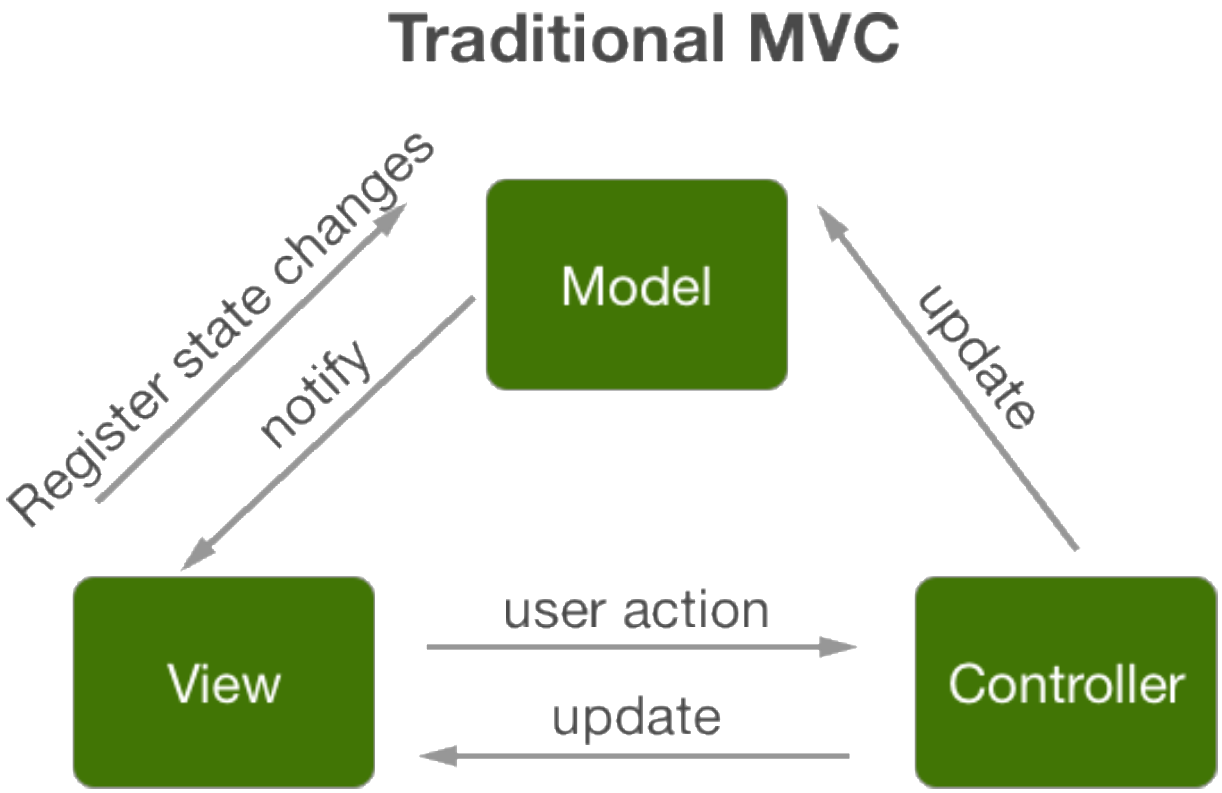


비동기

MVC?

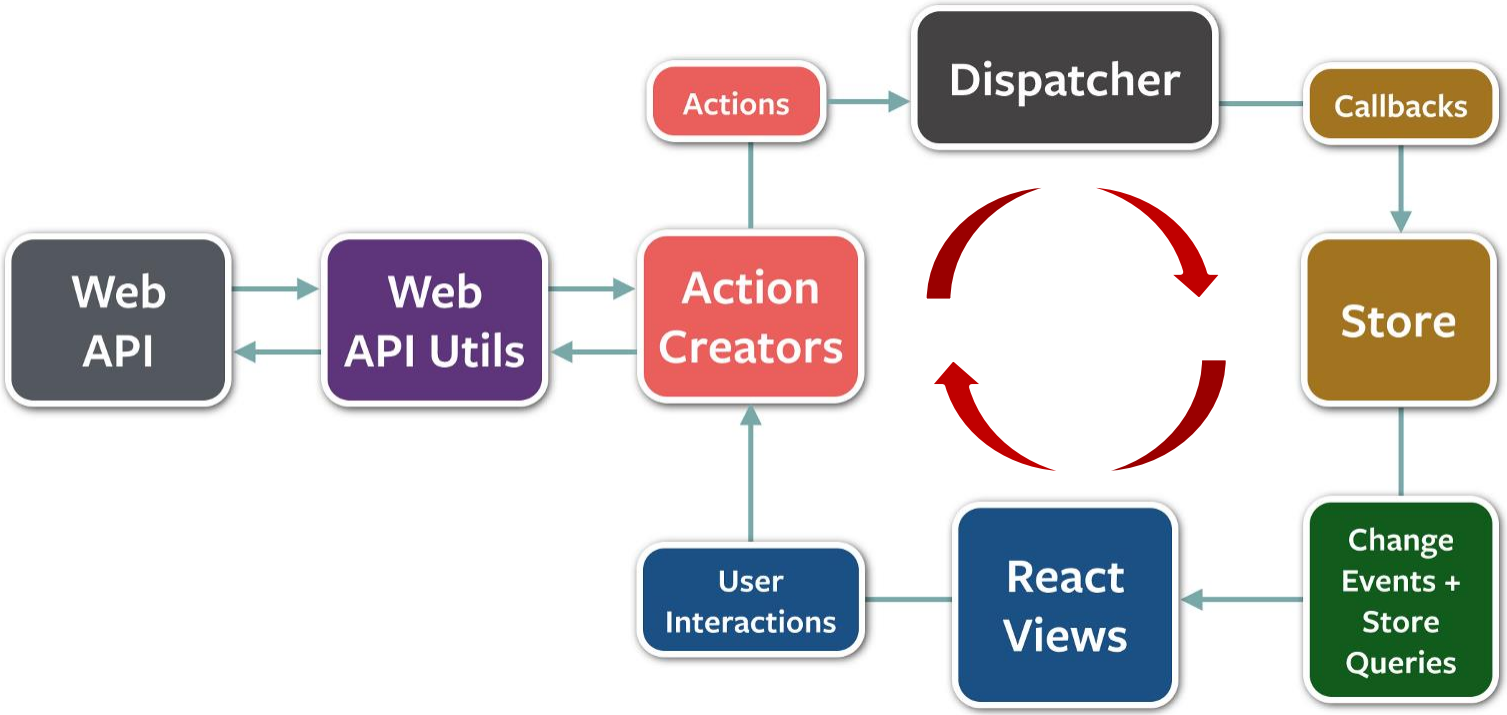
- Presentation은 View
- Data는 modal
- Logic은 controller

문제는 양방향



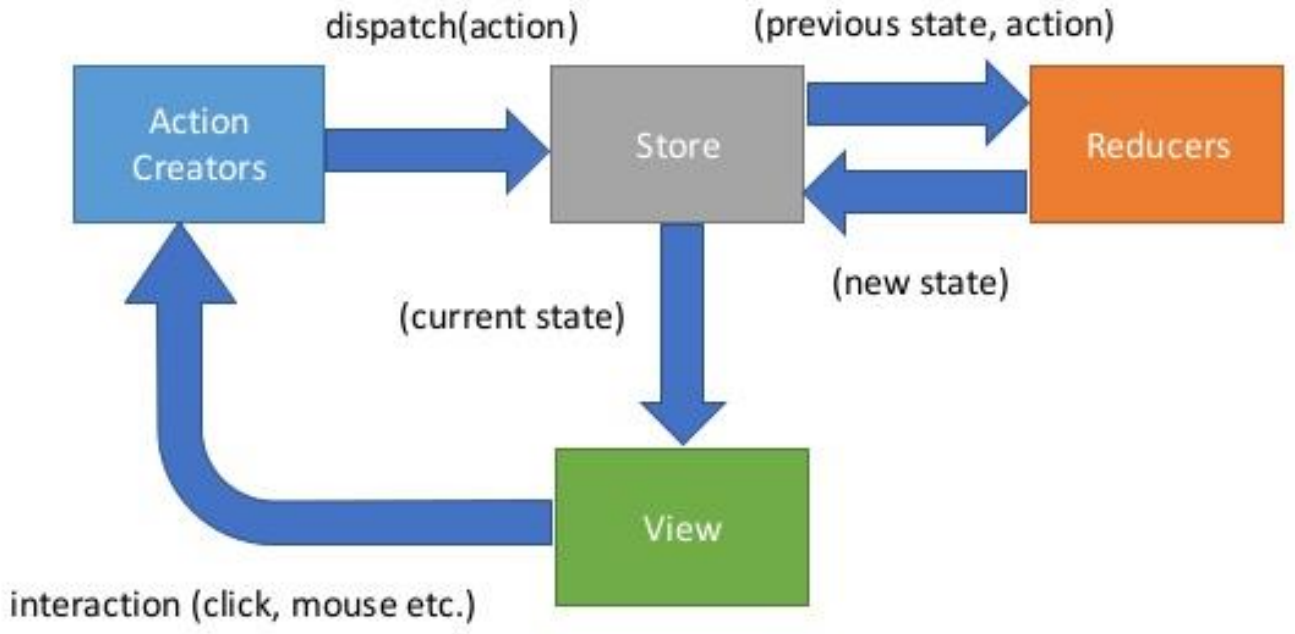
Flux Architecture

Dataflow 단방향

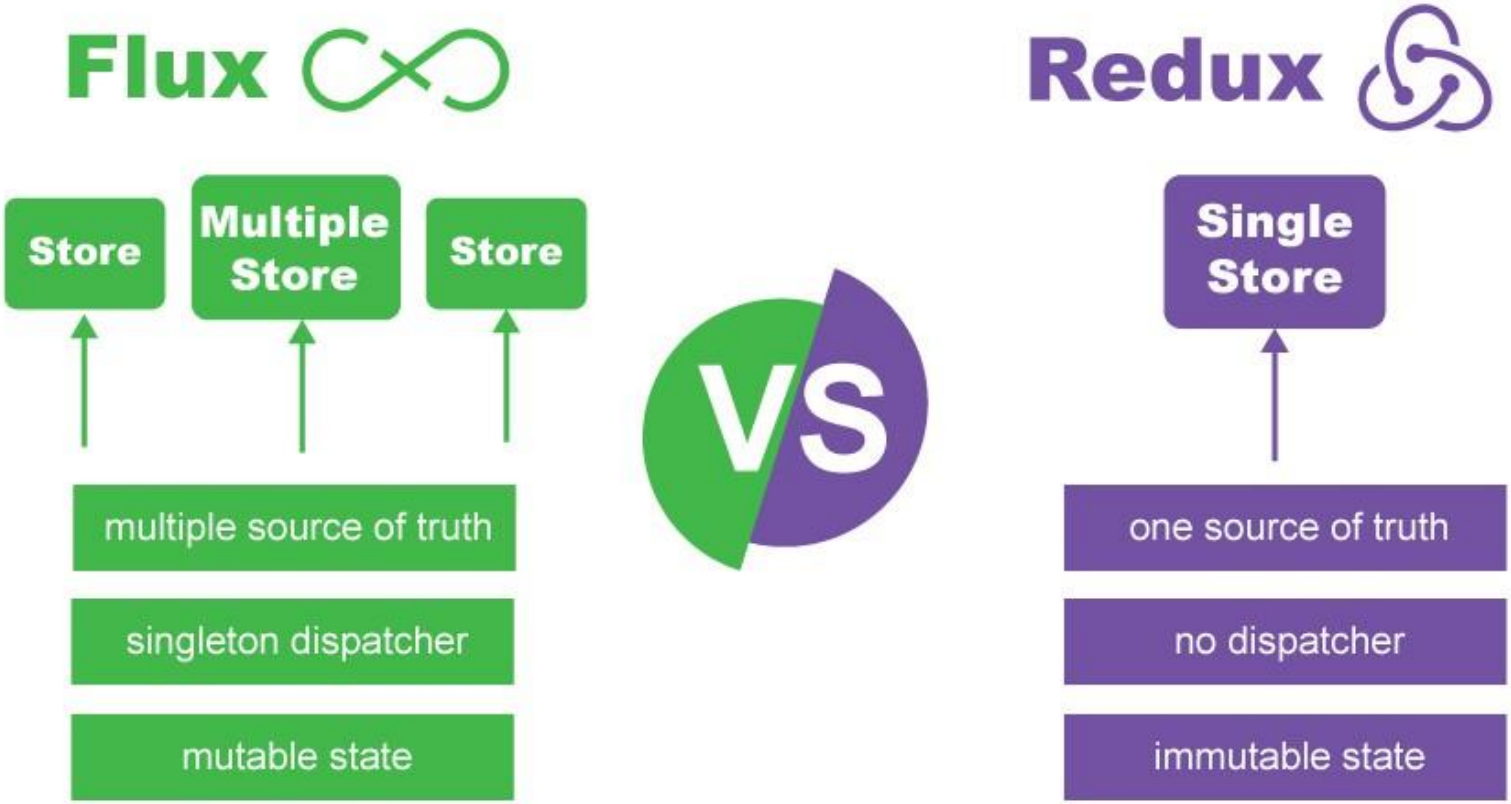


Redux Data Flow

Dataflow 단방향

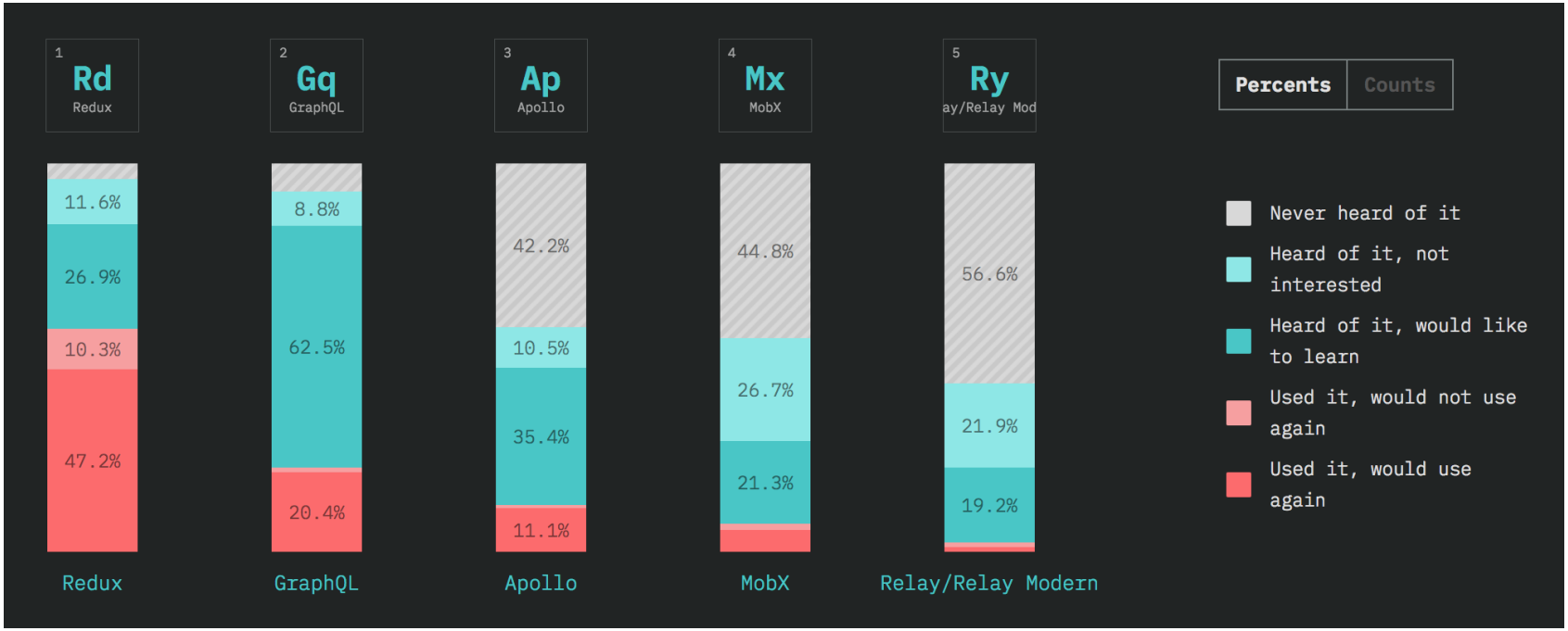


Flux VS Redux



Why Redux 인가?

StackOverflow 2018



https://2018.stateofjs.com/data-layer/overview/

Why Redux 인가?

- Predictability of outcome, Maintainability
  - 예측 가능한 구조. 엄격한 구조
- Developer tools
  - 모든 상태 변화 추적 가능. 타임 트러블
- Community and ecosystem
- Ease of testing
  - 함수형 프로그래밍 패러다임의 영향으로 테스트가 쉽다.

2. 함수형 프로그래밍 맛보기

함수형 프로그래밍

함수형 프로그래밍은 자료 처리를 수학적 함수의 계산으로 취급하고 상태 변경과 가변 데이터를 피하려는 프로그래밍 패러다임의 하나이다.

출처 : https://en.wikipedia.org/wiki/Functional_programming

1) 수학적 함수의 계산

High Order Function (고차함수)

다른 함수를 인자로 받거나 그 결과로 함수를 반환하는 함수.

고차 함수는 변경되는 주요 부분을 함수로 제공함으로서 **동일한 패턴 내에 존재하는 문제를 손쉽게 해결**할 수 있는 고급 프로그래밍 기법이다.

출처: https://en.wikipedia.org/wiki/Higher-order_function

```
if (A) {
    // 이럴 경우에는..
    for(let i = 0; i <len; i++) {
        // 실제 로직A는 여기서...
    }
} else {
    // 저럴 경우에는
    for(let i = 0; i <len; i++) {
        // 실제 로직B는 여기서...
        // 여기도 if문이...
        if (B) {
            // ...
        }
    }
    // ...
}
```

함수형 프로그래밍 책을 보면... forEach, map, filter, ...이런 함수들 이야기를 한다.

- filter => 조건문
- forEach => 반복문
- map => 반복문 + 변환(연산)
- reduce => 반복문 + 누적

실습

filter, forEach, map, reduce를 이용하여 코드의 가독성을 높여보자

2) 상태 변경과 가변 데이터를 피하려는

문제는 변수로 인해 발생한다

SideEffect와 SideCause

1. 입력값이 부정확한 경우

```
function getCurrentValue(value) {
    return (2 * value) + new Date().getTime();
}
```

2. 출력값이 부정확한 경우

```
const param = {
    value1: [10, 20, 30],
    value2: 20
};
function refFunction(value, param){
    param.value1 = [1, 2];
    param.value2 = 40;
    return value * 2;
} // 결과도 얻고, param도 바꾸고...

const result = refFunction(2, param);
```

함수형 프로그램에서는 부원인과 부작용을 지양한다

함수에 드러나지 않은 입력값 또는 결과값을 부원인(Side Cause)라고 하고 이로 인해 발생한 결과를 부작용(Side Effect)이라 한다.

가변 객체와 불변객체

가변 객체 (Mutable Object)

- 생성 후에 상태를 변경할 수 있는 객체
- Array, Object

```
const animals = ['ant', 'bison', 'camel',
'duck', 'elephant'];
console.log(animals.splice(2)); // ["camel",
"duck", "elephant"]
// animals => ["ant", "bison"]
```

불변 객체 (Immutable Object)

- 생성 후 그 상태를 바꿀수 없는 객체
- 그 외 number, boolean, string

```
const animals = ['ant', 'bison', 'camel',
'duck', 'elephant'];
console.log(animals.slice(2)); // ["camel",
"duck", "elephant"]
// animals => ['ant', 'bison', 'camel',
'duck', 'elephant'];
```

함수형 프로그래밍은 불변객체(IMMUTABLE)을 지향

= 생성 후 그 상태를 바꿀 수 없는 객체

= 만약 상태가 바뀌었다면 reference가 바뀌었다.

= reference가 바뀌었다면 상태가 바뀌었다.

```
function get(objectValue) {
    const obj = Object.assign({}, objectValue);
    obj.newProp = "바꿨으면 데이터 객체의 레퍼런스를 바꾸야지";
    return obj;
}
```

함수형 프로그래밍은

- 부작용(Side-effect)을 발생시키지 않는다.
  같은 입력이 주어지면 항상 같은 출력을 하는 **순수함수**를 지향
- 외부의 가변 데이터(Mutable)에 의존하지 않는다.

함수형 프로그래밍은

"동일 입력", "동일 출력"

- 테스트가 용이하다.
- 동시성 처리가 좋다.
- 버그 발생율이 낮다

Why Redux 인가?

다양한 입력(사용자 입력, 서버 자원, 데이터)에 의한 **빈번한 변화**

비동기 프로그래밍(Event) 방식의 Javascript

Redux는 **함수형 프로그래밍을 지향**.
Dataflow의 **단방향**

- Predictability of outcome, Maintainability
- Developer tools
- Community and ecosystem
- Ease of testing

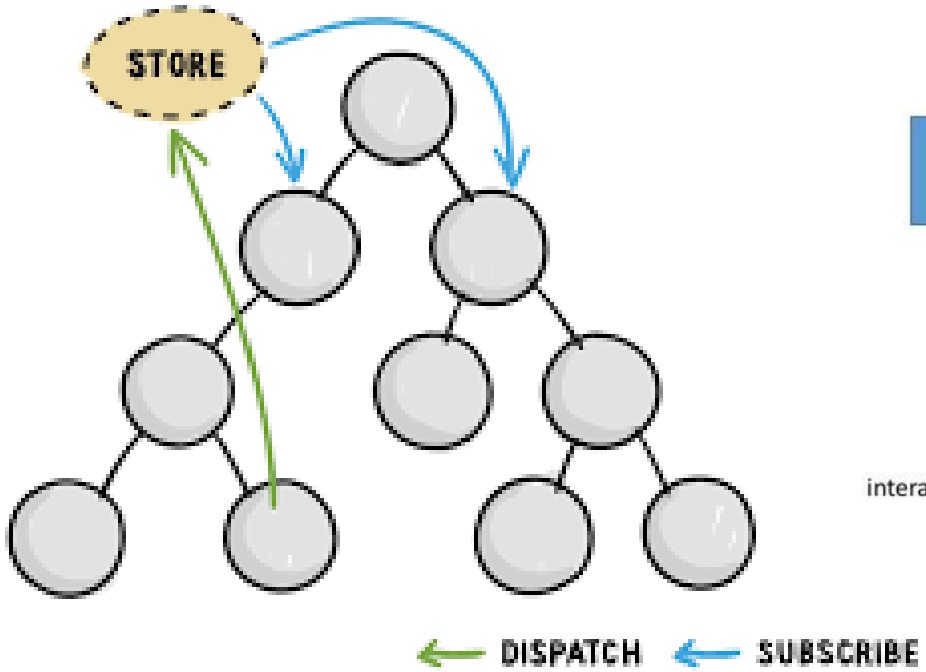
3. Redux 사용하기

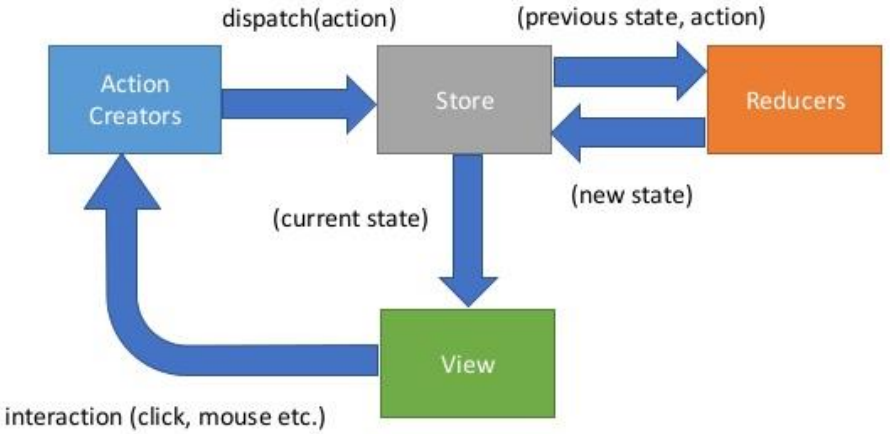
Redux 사용하기

전체 Application의 모든 상태를 **하나의 store**로 관리한다.

이 store는 **immutable**한 상태 트리이다. Store의 상태는 직접 바꿀 수 없다.

상태 변화가 일어날 때는 **항상 새로운 객체**(action, reducer)가 생성된다.





Action

액션은 type 속성을 가지는 Object이다.

액션은 **어떤 행위가 되었는지**에 대한 기술을 담당한다.

```
{
    type: “INCREASE_COUNT”
    payload: 20
}
```

Action의 포맷은 flux-standard-action에서 제시하는 디자인을 사용한다.
- type: 액션의 종류
- payload: 전달할 데이터
- error: 에러

https://github.com/redux-utilities/flux-standard-action

Action Creator

액션 생성자는 액션을 반환하는 함수이다.

액션 생성자는 호출 시 매번 새로운 액션을 생성한다.

```
function increase(step) {
    return {
        type: “INCREASE_COUNT”
        payload: step
    }
}
```

액션 생성자는 액션을 생성하여 Store의 dispatch 함수에 전달한다.

```
store.dispatch(increase(1));
```

Reducer

액션에 의해 변경된 상태를 Store로 전달하는 함수 (리듀서는 **순수 함수**로 구성)

액션은 행위에 대한 기술을 담당하는 반면,

리듀서는 액션에 의해 **어떻게 상태가 변하는지**를 기술한다.

```
function reducer(state = { count: 0 }, action) {
    switch(action.type) {
        case “INCREASE_COUNT”:
            return {
                …state,
                count: state.count + action.payload
            };
        default: return state;
    }
}
```

Store

어플리케이션의 모든 상태를 가진 단 하나의 immutable한 상태 트리

Store는 dispatch에 의해 전달된 액션을 시작으로

Store에 연결된 reducer에 의해서만 상태를 변경할 수 있다. (Read-Only)

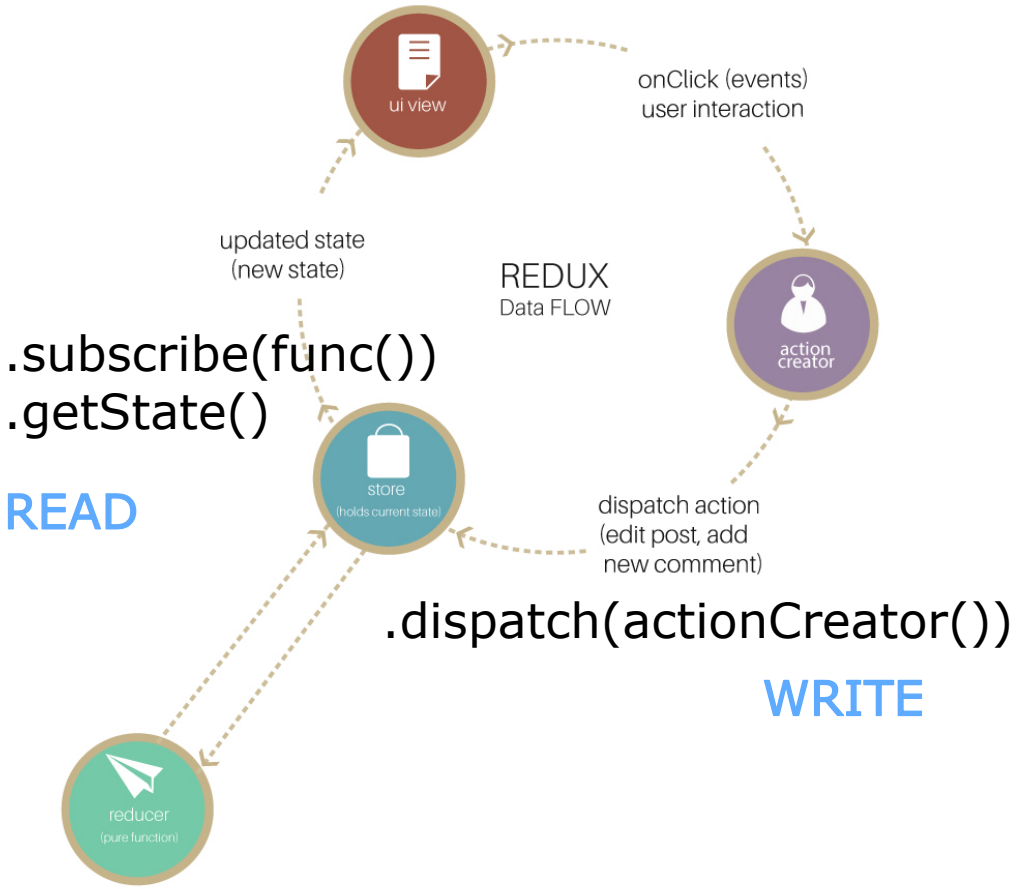
```
import { createStore } from 'redux'

// store 생성시 reducer를 인자로 받는다. (store에 reducer 연결)
const store = createStore(counter);

// 상태 변화 요청
store.dispatch(increate(1));

// 상태 변화 감지
const unsubscribe = store.subscribe(() => {
    // 상태 얻기 (read-only)
    console.log(store.getState());
});
```

Redux 정리



- 단 하나의 store로 관리
- store는 **immutable**한 상태로 **읽기 전용**이다.
- 변화를 담당하는 reducer 함수는 항상 **순수함수**

실습

“01. redux_basic.html”을 이용하여 카운터를 증가시키고 감소시키고 초기화하는 간단한 redux 예제를 만들어 보자.

- Object.assign

  (https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

- Object spread operator

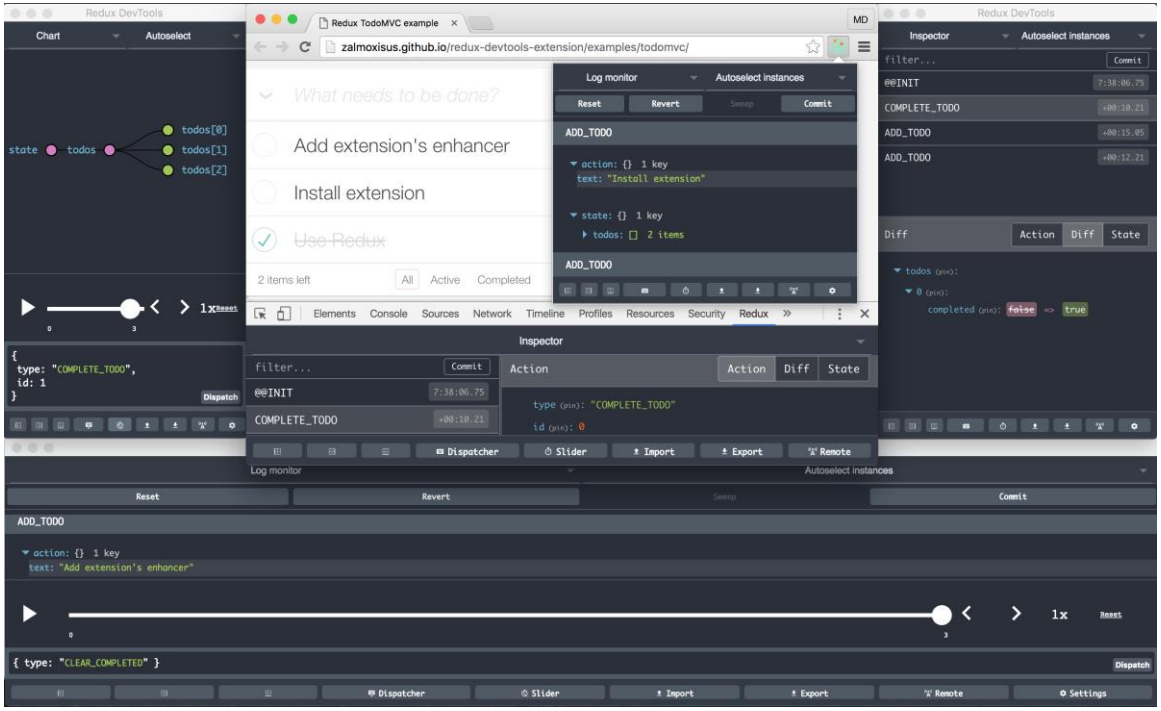
  (https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Operators/Spread_operator)

4. Redux devtool

Chrome redux devtool

https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

- 액션에 의한 상태 변화
- Store 상태
- Time travel
- …



TODO React - http://zalmoxisus.github.io/examples/todomvc/

redux devtool 설정하기

createStore의 두번째 인자에

window.__REDUX_DEVTOOLS_EXTENSION__ 함수를 호출한다.

```
const store = createStore(
    counterReducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

실습

“01. redux_basic.html”에서 store subscribe 제거하고 Redux devtool 연결하기

5. Redux 구조 설계하기

Redux Store 설계

어떻게 1depth를 나눌 것인가?

1. 페이지 단위

```
const store = createStore({
    home,
    end
});
```

2. 도메인(기능) 단위

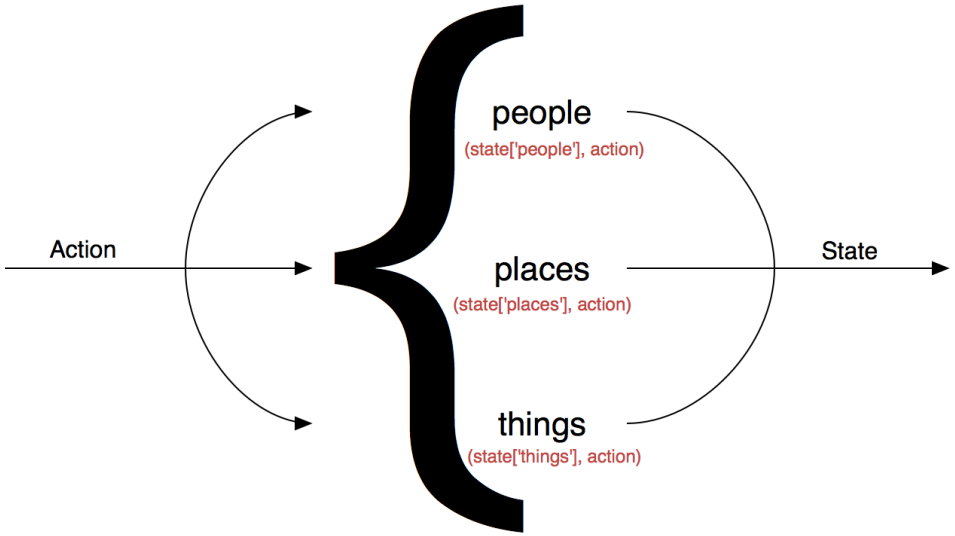
```
const store = createStore({
    counter,
    messanger
});
```

Reducer의 결합/분리

combineReducers

https://redux.js.org/api/combinereducers

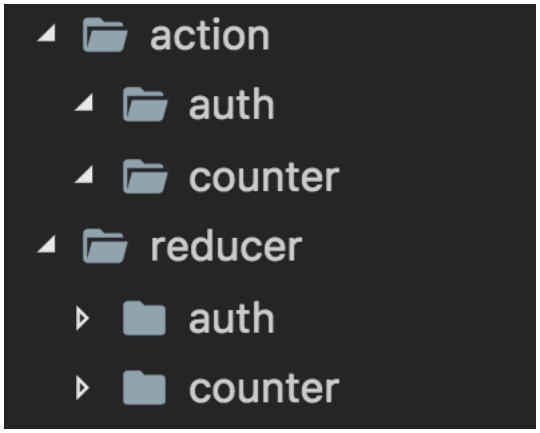
```
const store = createStore(
    combineReducers({
        people: peopleReducer,
        places: placesReducer,
        things: thingsReducer
    })
);
```



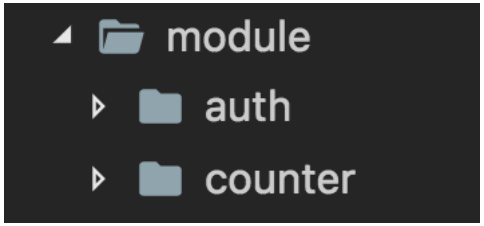
Redux 폴더 구조

Action, Action Creator, Reducer 를 어떻게 관리 할 것인가?

1. Action + Action Creator와 reducer로 구성

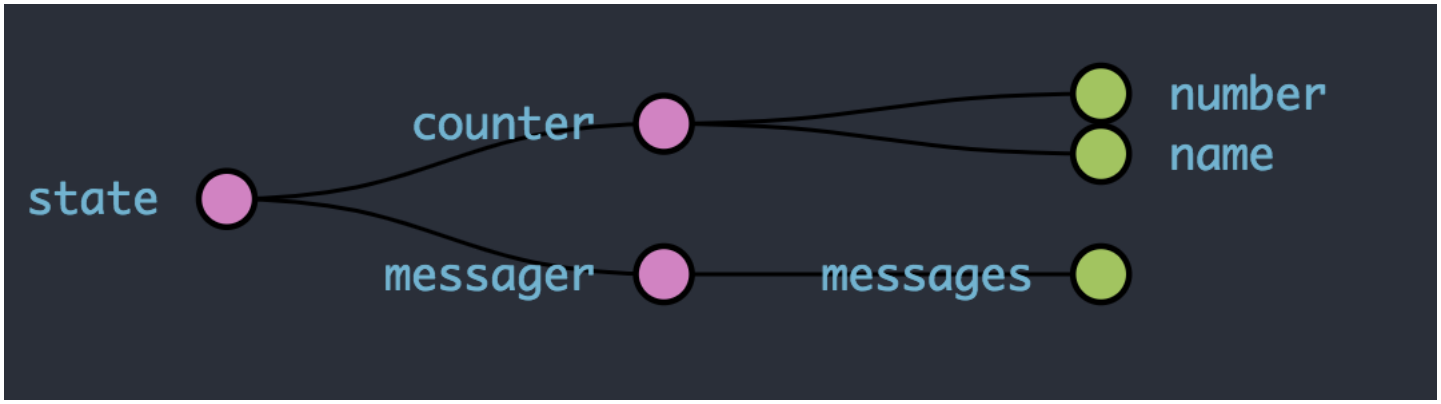


2. Action + Action Creator + reducer (**Ducks: Redux Reducer Bundles**)



실습

"02. redux_ combinereducers.html"을 이용하여 counter reducer와 messager reducer를 이용한 redux 예제를 만들어 보자.



실습

“02. redux_ combinereducers.html”로 개발한 것을 duck 구조로 변경해 보자.

6. Redux Middleware

Redux Middleware

Action와 리듀서 사이의 존재

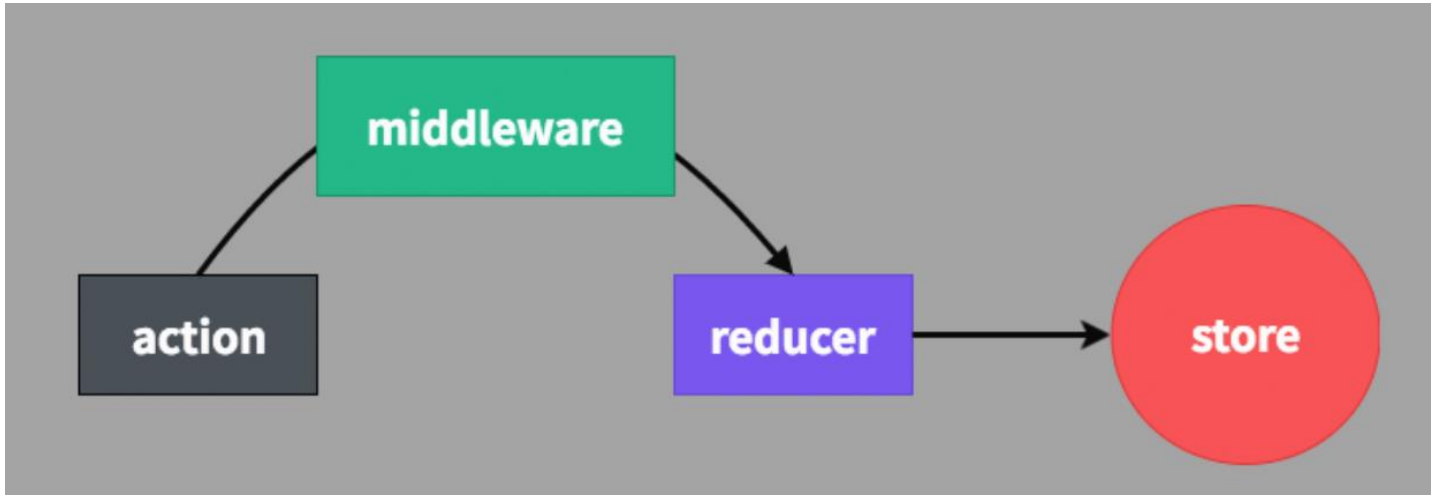
Middleware 사용 예

- Action 호출 후 일정 시간이 경과된 이후에 상태가 반영 되어야하는 **비동기 상황**

  redux-thunk, redux-promise-middleware, redux-saga, redux-observable

- 액션 호출 상태를 추적하는 경우

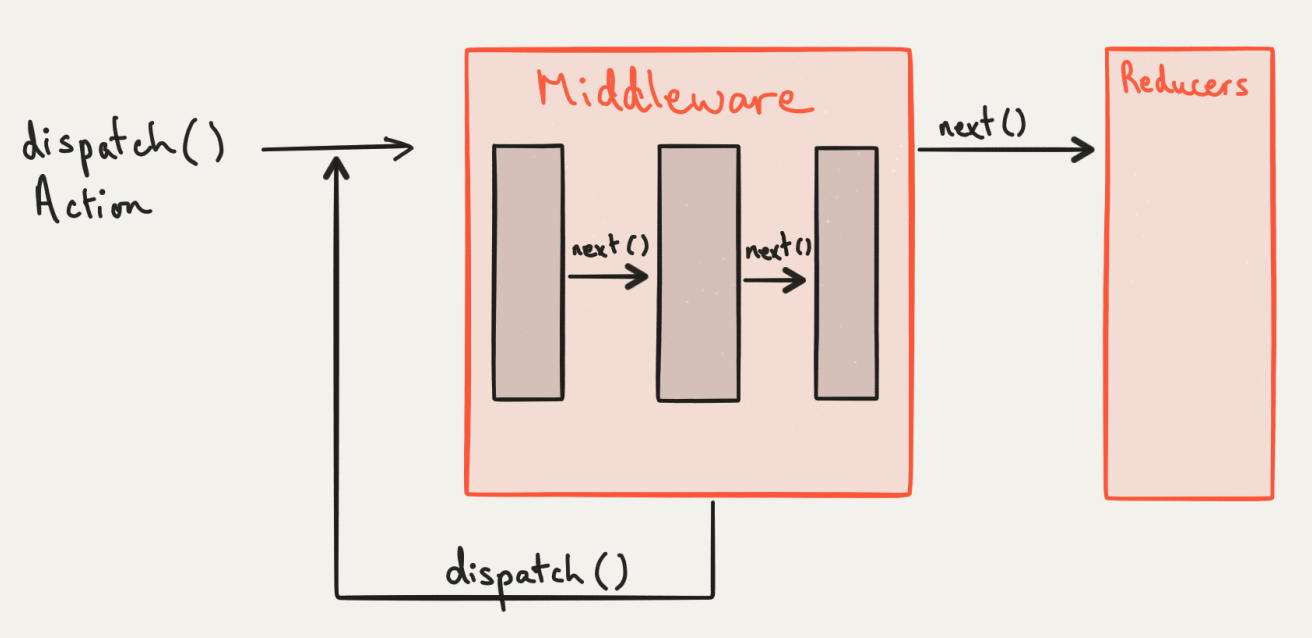
  redux-logger, …



Redux Middleware 구조

Middleware가 3개 적용되었을 경우, 다음과 같은 흐름으로 진행된다.

- 액션이 Middleware A의 입력값으로 전달
- Middleware A의 결과가 Middleware B의 입력으로 전달
- Middleware B의 결과가 Middleware C의 입력으로 전달
- Middleware C의 결과가 reducers에 전달.



Redux Middleware 적용

Store를 생성할 때 2번째 인자에 applyMiddleware를 이용하여 미들웨어를 등록

```
const store = createStore(
    reducers,
    applyMiddleware(m1, m2, m3, ...)
);
```

Redux Middleware 적용

Redux devtool 의 경우에는

__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ 함수를 이용하여

applyMiddleware 와 함께 사용할 수 있다.

```
const store = createStore(
    reducers,
    __REDUX_DEVTOOLS_EXTENSION_COMPOSE__(
        applyMiddleware(m1, m2, m3, ...)
    )
);
```

Compose 함수.

오른쪽에서 왼쪽으로 인자가 하나인 함수를 합치는 유틸.
예외적으로 가장 마지막 함수는 여러 개의 인자를 받을 수 있다.

```
compose(func1, func2, func3);   // func1(func2(func3(…)))
```

Redux Middleware 만들기

```
const loggerMiddleware = (store) => {
    return next => action => {
        // 미들웨어 작업
    }
}
```

실습

“03.middleware_logger.html”을 이용하여 Logger middleware를 작성한다.

- 기능: action 전후의 상황을 기록한다.
- console.group을 이용하여 action명 단위로 로그를 기록한다.
- action 호출 전/후 state는 console.table로 로그를 표시한다.

Redux-thunk

Redux를 개발한 Dan Abramov 가 개발

매우 직관적이고 간단한 비동기 작업이 가능함.

객체가 아닌 함수를 반환하는 Action Creator 를 dispatch할 수 있게 해준다.

왜? 함수인가? 함수는 lazy 하다.

```
const x = 1 + 2;                    // 일반적인 action

const funcX = () => 1 + 2           // 이것이 thunk 액션
```

Redux-thunk 사용

설치

```
# npm install redux-thunk
```

사용법

1. Store 생성시 middleware 추가
2. dispatch, getState를 인자로 받는 함수를 반환하는 Action Creator 생성

```
function increaseAsync(step) {
    return (dispatch, getState) => {
        // 1초후 dispatch
        setTimout(() => {
            dispatch(increase(step));
        }, 1000);
    }
}
```

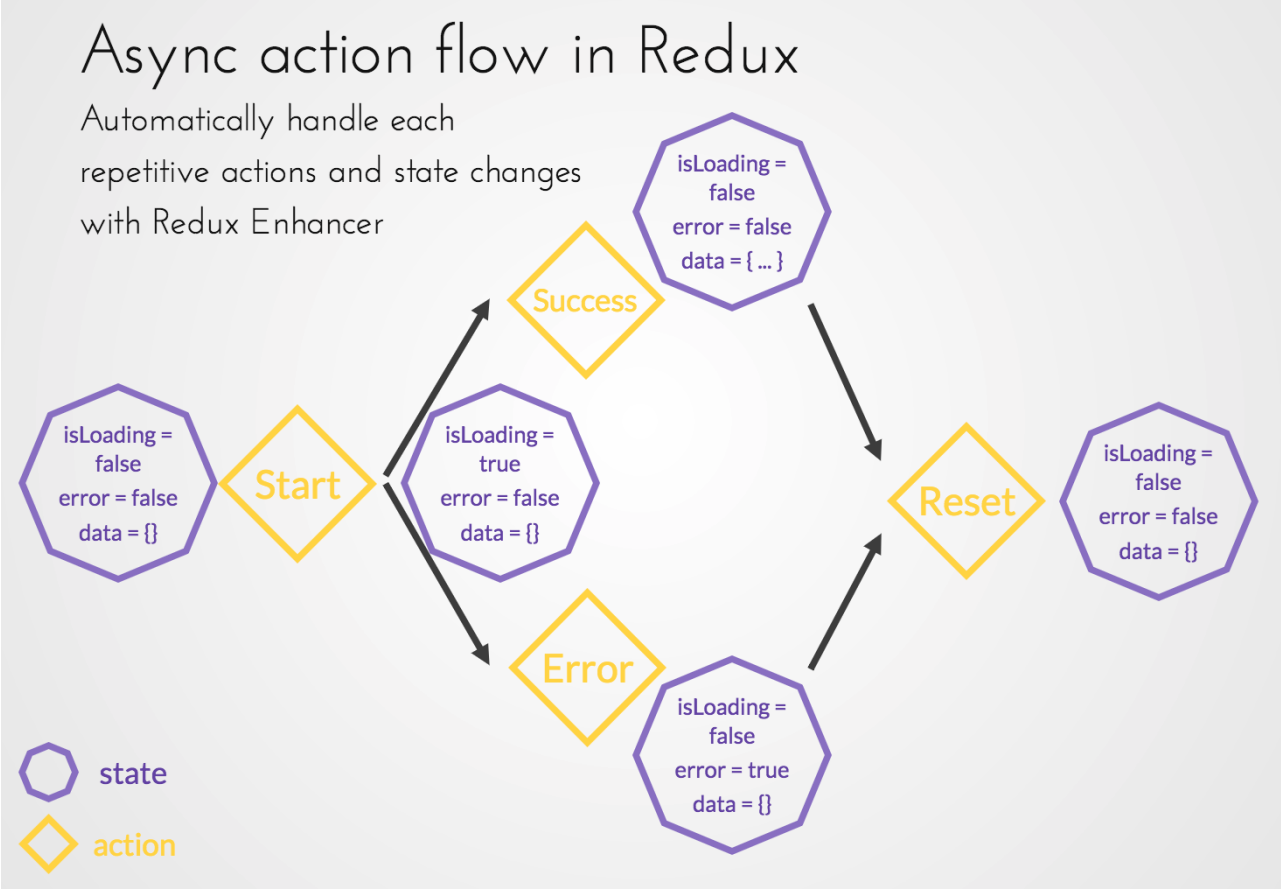
실습

“04. middleware_react-thunk.html”을 이용하여 비동기 작업이 가능하도록 redux-thunk를 설정하고 비동기 액션을 작성하라.

비동기 상태 관리

비동기 액션의 경우 다음과 같은 상태가 존재한다.

1. 액션 시작
2. 액션 진행중
3. 액션 성공시
4. 액션 실패시
5. 초기화



실습

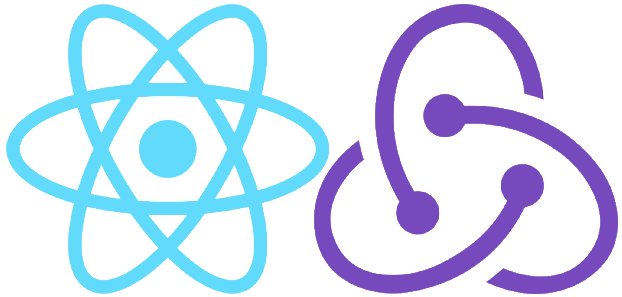
“05.react-thunk_status.html”을 이용하여 비동기 작업 상태에 따라 isLoading, hasError 값이 정상적으로 상태에 반영되도록 구성하시오.

- setTimeout을 Promise로 바꿔보세요

* status를 관리하는 미들웨어를 만들어보세요.
* react-actions로 간소화해보기.

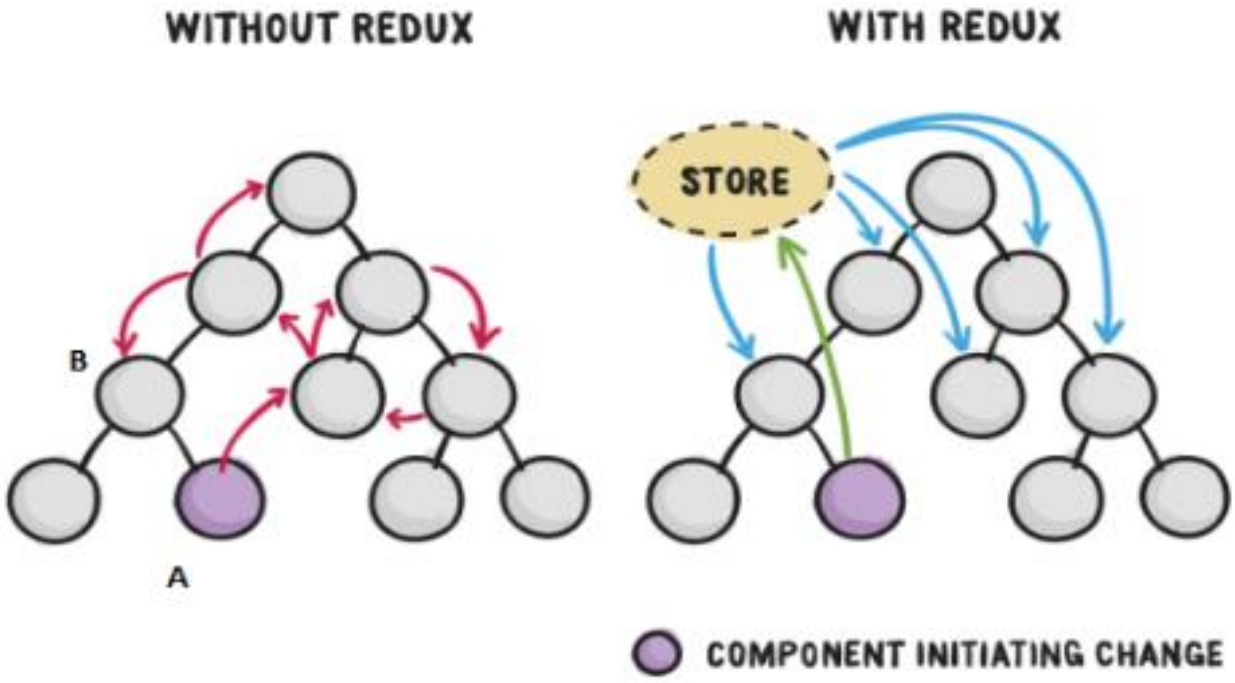
7. Redux + React

Redux + React



Redux의 장점 + React 단점을 보완

- React의 **prop drilling** 을 방지
- React의 **성능 이슈** 문제 <상위 컴포넌트가 변경되면 하위가 모두 변경된다>



React 컴포넌트 구성하기

React 컴포넌트의 **재사용성**을 높이기 위해서 다음과 같이 분리해서 컴포넌트를 사용

프리젠테이션 컴포넌트

오직! Props만 존재하는 컴포넌트

State가 존재한다면 데이터가 아닌 오직! UI에 관련된 것들로 구성된 컴포넌트

컨테이너 컴포넌트

React render가 존재하지 않는 컴포넌트 (내부에 DOM이 없음)

각각의 React 컴포넌트를 감싸는 역할만 하는 컴포넌트

Redux와의 연결을 담당

React와 Redux 연결하기 (react-redux)

React 와 Redux의 연결은 react-redux 라이브러리를 이용하여 연결

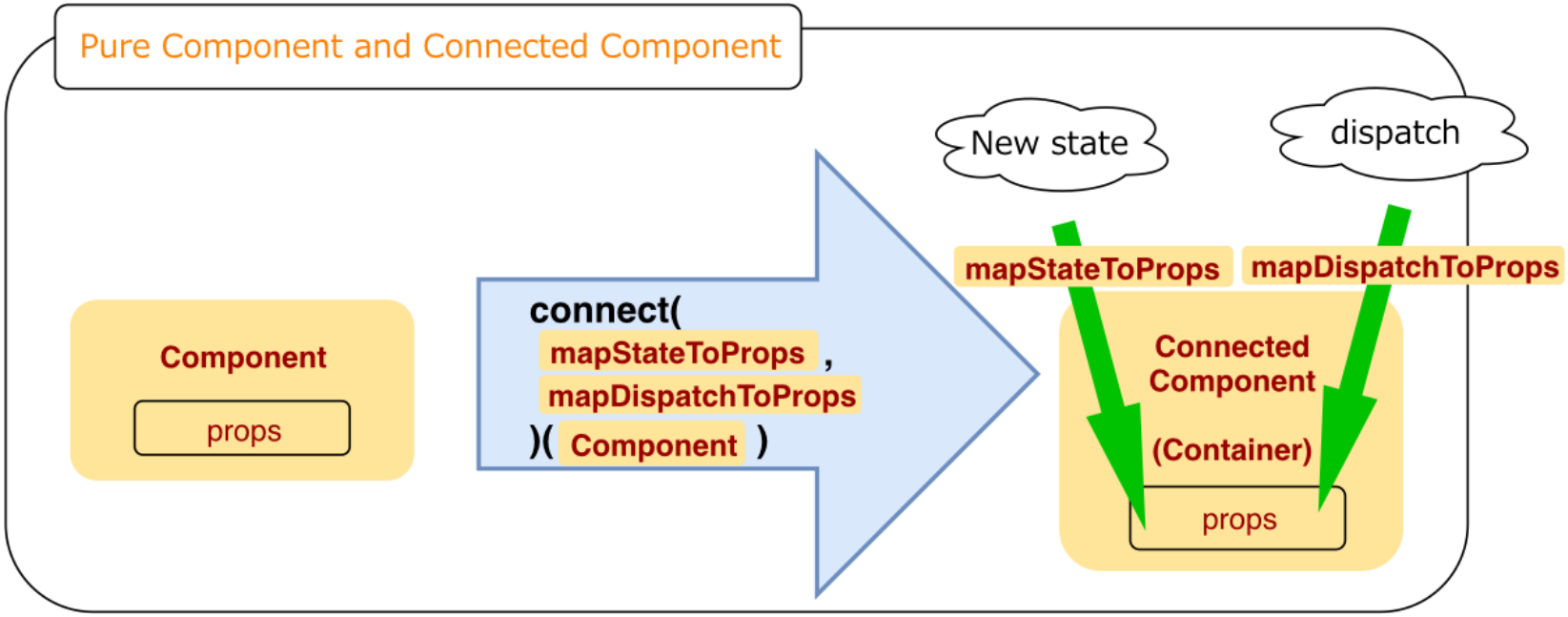
React-redux 라이브러리의 connect 함수를 이용

connect 함수는 HOC(High order component)를 반환한다.

설치

```
# npm install react-redux
```

React와 Redux 연결하기 (react-redux)




```
const connectedComponent = connect(
      mapStateToProps,
      mapDispatchToProps
)(Component)
```

React와 Redux 연결하기 (react-redux)

mapStateToProps

Store의 state 값을 연결하는 컴포넌트의 props로 전달 할때 사용되는 함수

```
const mapStateToProps = state => {
    return {
        count: state.counter.number
    };
};
```

Store의 counter.number 값을 count props로 연결된 컴포넌트에 전달한다.

React와 Redux 연결하기 (react-redux)

mapDispatchToProps

Store의 dispatch를 호출하는 함수를 props로 전달할 때 사용되는 함수

Dispatch를 인자로 받아 호출할 수 있는 action creator를 props 로 전달한다.

```
const mapDispatchToProps = dispatch => ({
    onIncrease: step => dispatch(increase(step)),
    onDecrease: step => dispatch(decrease(step)),
    onReset: () => dispatch(reset()),
    onIncreaseAsync: step => dispatch(increaseAsync(step)),
    onDecreaseAsync: step => dispatch(decreaseAsync(step)),
    onResetAsync: () => dispatch(resetAsync())
});
```

mapDispatchToProps (Helper용 인터페이스)

bindActionCreators

https://redux.js.org/api/bindactioncreators

Dispatch를 호출 할 수 있는 형태의 객체를 반환한다.

```
const mapDispatchToProps = dispatch =>
    bindActionCreators({
        onIncrease: increase,
        onDecrease: decrease,
        onReset: reset,
        onIncreaseAsync: increaseAsync,
        onDecreaseAsync: decreaseAsync,
        onResetAsync: resetAsync
    }, dispatch);
```

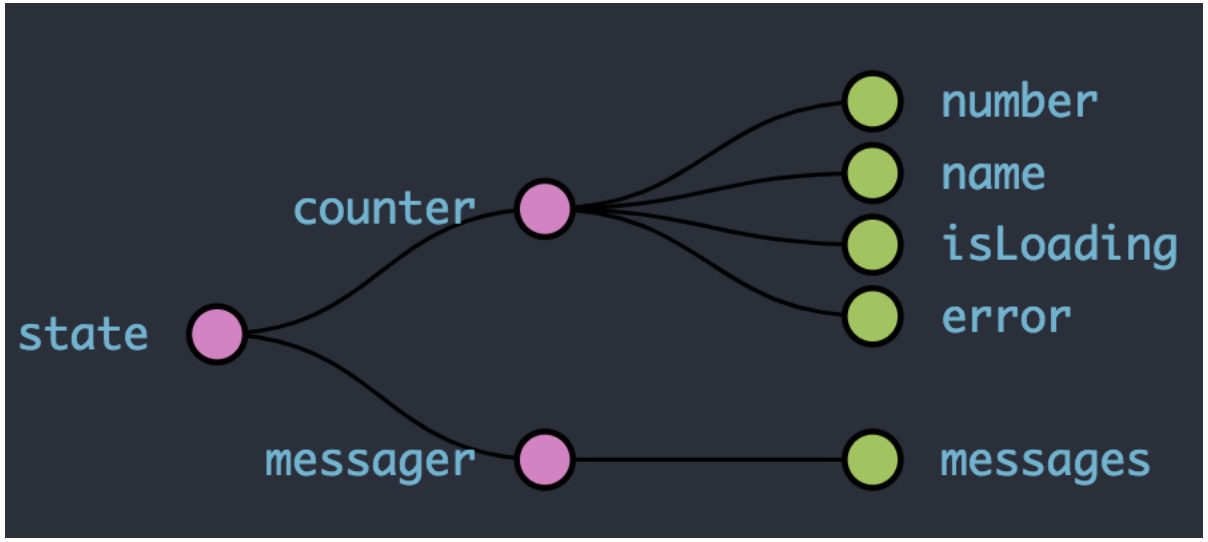
mapDispatchToProps (간편 인터페이스)

```
const mapDispatchToProps = {
    onIncrease: increase,
    onDecrease: decrease,
    onReset: reset,
    onIncreaseAsync: increaseAsync,
    onDecreaseAsync: decreaseAsync,
    onResetAsync: resetAsync
};
```

실습

앞에서 작성한 Counter, messager를 이용한 예제를 React로 만들어 보시오

- CounterContainer.jsx
- MessagerContainer.jsx



고맙습니다.