고용노동부 한국산업인력공단 국가인적자원개발컨소시엄

Vue.js

2021.3.15 ~ 3.16

백명숙

과정개요

■ Vuejs의 개념과 Architecture를 이해하고, Vuejs 컴포넌트의 생성 방법을 살펴본다. Vuejs에서 컴포넌트를 작성할 때 사용하는 Template과 Vuejs의 Directive, 데이터 흐름, 이벤트 핸들링 등을 살펴본다. State를 관리를 효율적으로 해주는 Vuex와 라우팅 기능을 제공하는 Vue-Router를 활용해 본다.

과정목표

■ 방대해진 웹서비스의 서버의 처리속도, 방대해진 HTML의 양에 따른 유지 및 관리를 Vuejs가 제시하고 있는 기능에 따라 고급 자바스크립트 기술을 교육함으로써 웹 개발자의 능력을 향상시키기 위한 필수 과정입니다.

러닝 맵

강사 프로필

성명	백명숙
소속 및 직함	휴클라우드 이사
주요 경력	일은시스템, Sun MicroSystems 교육서비스, 인터넷커머스코리아
강의/관심 분야	Java, Framework, Python, Data Science, Machine Learning, Deep Learning
자격/저서/대외활동	Java기반 오픈소스 프레임워크/NCS 개발위원

Learning Object(학습모듈) 및 커리큘럼

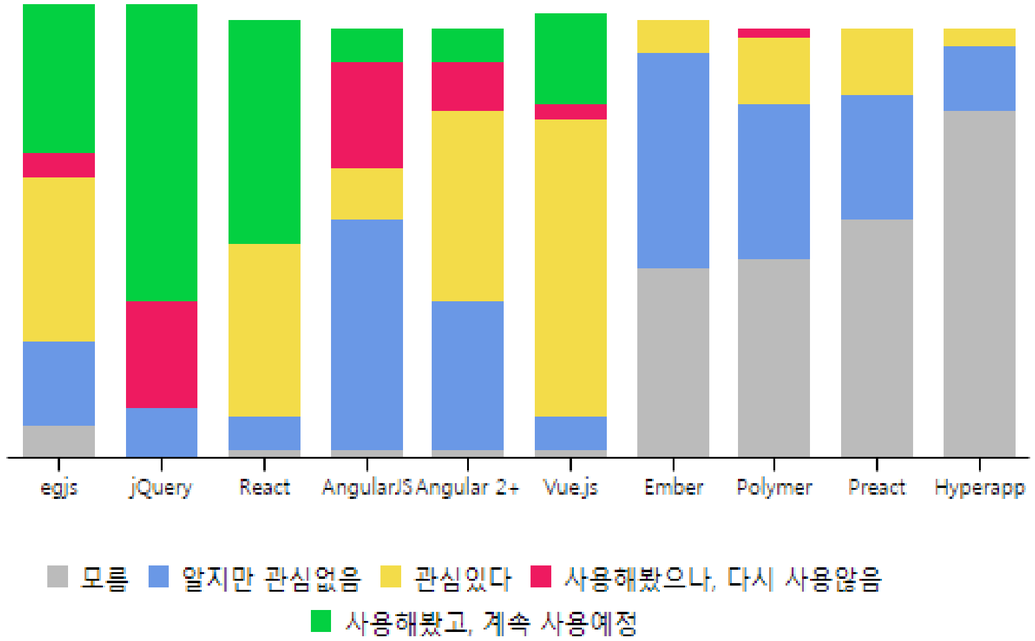
LO	커리큘럼
Vue.js 소개와 ES6	- Vue.js 소개 - 개발환경 설정 - ECMA Script 6 소개
Vue.js Directive	- Vue.js Directive - Vue.js Component 간의 통신
Vue.js 컴포넌트	- Vue-CLI 설치 및 사용 - Todo(할일 관리) App 만들기
Vue.js App 리팩토링	- Todo App 리팩토링 - Todo App UX 개선
Vuex	- Vuex 소개 - Todo App에 Vuex 적용
axios 와 vue-axios	- axios와 vue-axios 소개 - Todo App에 axios 적용
Vue Router	- Vue Router

Vue.JS

백명숙(ms.baek@hucloud.co.kr)

Javascript 라이브러리와 프레임워크

- 이제 프론트엔드 개발은 '라이브러리와 프레임워크를 사용한 개발'을 의미한다. 다양한 라이브러리와 프레임워크가 등장하고 저마다 장점을 내세우지만 현재(2018년 10월)는 React와 Vue.js, Angular가 주요 라이브러리와 프레임워크로 자리를 잡았다고 할 수 있습니다. 어떤 라이브러리와 프레임워크가 프로젝트에 가장 잘 맞을지 고민하는 모습이 오늘날 프론트엔드 개발의 모습이라 할 수 있습니다.
- The State of JavaScript의 2017년 설문조사(https://2017.stateofjs.com/2017/front-end/results/)
- 2018년에 실시한 네이버 개발자 대상 설문 조사 결과



React

React(https://reactjs.org/)

- “컴포넌트” 라는 개념에 집중이 되어 있는 라이브러리입니다. 생태계가 엄청 넓고, 사용하는 곳도 많습니다. HTTP 클라이언트, Router, 상태 관리 등의 기능들은 내장되어 있지 않습니다. 따로 공식 라이브러리가 있는 것이 아니므로, 개발자가 원하는 스택을 맘대로 골라서 사용 할 수 있습니다.
- 2017년 9월에 React 16(코드명 Fiber)이 릴리즈 되었다. React 16에서는 서버 렌더러를 다시 작성해 SSR(server-side rendering)을 개선했고, render() 메서드가 배열을 반환할 수 있게 했다. 또한 사용자 정의 DOM 속성(custom DOM attribute) 지원과 Fragment 지원이 추가되었다. Fragment는 자식 요소를 감싸는 별도의 요소를 추가하지 않고 자식 요소를 여러 개 추가할 수 있게 한다.
- React 애플리케이션 개발을 손쉽게 시작할 수 있게 도와주는 CLI 도구인 Create React App이 2018년 1월에 Facebook의 인큐베이터 프로그램을 벗어나 Facebook의 공식 저장소 프로젝트로 이관 되었다.

Angular

Angular(https://angular.io/)

- Angular(v2 이상)는 Google의 Angular 팀과 개인 및 기업 공동체에 의해 주도되는 Typescript 기반 오픈 소스 프론트엔드 웹 애플리케이션 프레임워크이다. Angular는 AngularJS를 개발한 동일 팀으로부터 완전히 다시 작성한 것이다.
- Router, HTTP 클라이언트 등 웹 프로젝트에서 필요한 대부분의 도구들이 프레임워크 안에 내장되어 있으며, 컴포넌트와 템플릿이 완벽하게 분리되어 있다.
- Angular는 6개월마다 메이저 버전을 릴리즈 한다. 현재는 Angular 7 버전입니다.
- Angular 애플리케이션 개발을 손쉽게 시작할 수 있게 도와주는 CLI 도구인 Angular CLI 가 있습니다.

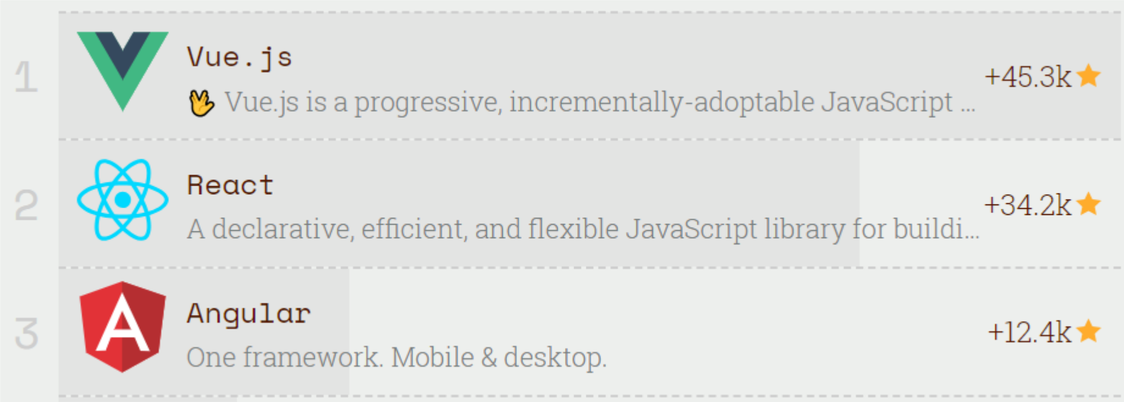
Vue.js

Vue.js(https://vuejs.org/)

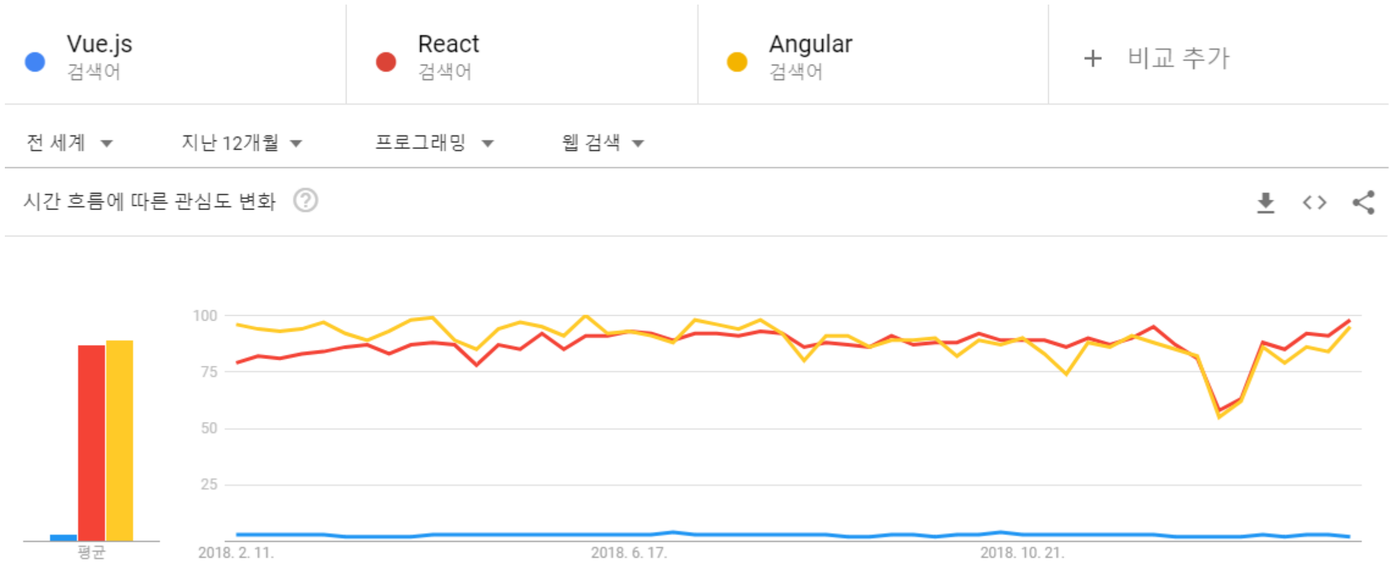
- Vue.js는 2017년에 그야말로 '엄청난' 성장을 이루었다. 이제는 React, Angular 등과 경쟁하는 동등한 위치에 올라섰다.
- Vue.js의 개발자인 Evan You의 말에 따르면 Vue.js의 개발은 AngularJS에서 가장 마음에 드는 특징을 복제하는 것에서 시작됐다(Between the Wires의 인터뷰 참고). AngularJS를 Angular로 바꾸는 마이그레이션 작업이 간단하지 않고 AngularJS의 지원 또한 언젠가 종료될 것을 감안 했을 때, AngularJS와 유사성을 가진 Vue.js는 수많은 AngularJS의 사용자에게 매력적인 대체재일 수 있습니다.
- Vue.js의 코어 라이브러리는 React와 유사하게 데이터 바인딩과 컴포넌트에만 집중한다.
- 복잡한 대규모 애플리케이션을 개발하려면 라우팅, 상태 공유, 컴포넌트 간의 통신 등을 위해 별도의 수많은 도구가 필요해진다. Ember나 Angular는 이런 도구를 프레임워크에 내재하는 형태로 접근한다.
- React는 도구를 제공하는 역할을 커뮤니티를 통한 생태계에 맡기고 있다. Vue.js는 중간적인 형태로 접근을 한다. 코어는 최소한의 기능만 제공하고, 필요한 다른 도구는 별도로 제공한다.

Vue.JS

- 2018년 GitHub에서 Star를 받은 수 랭킹(링크)

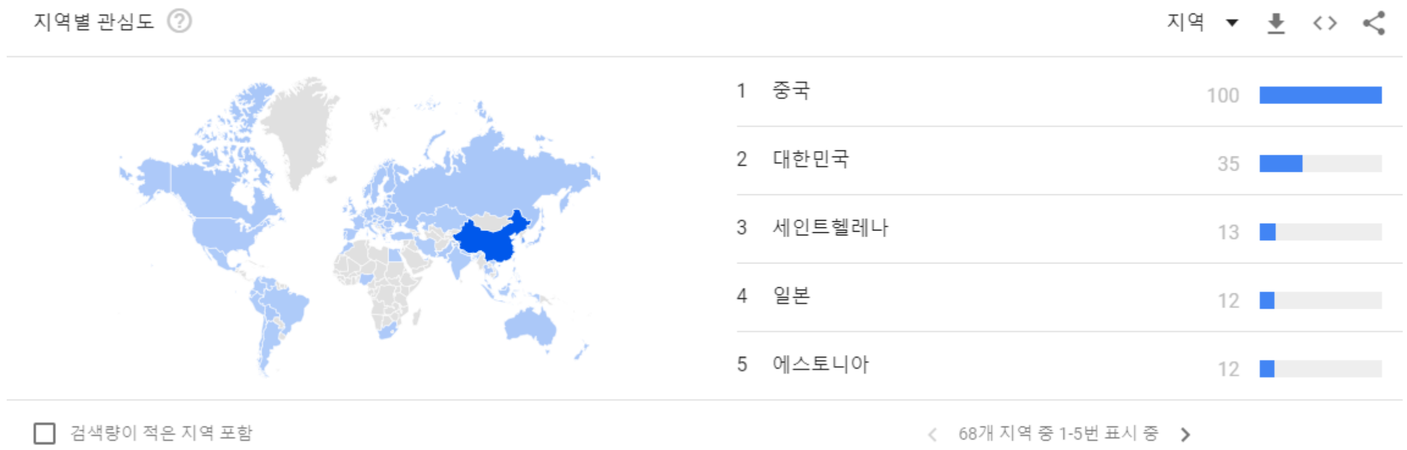


- 구글 트렌드 키워드별 관심도 (링크)



Vue.JS

- 구글 트렌드 Vue.js 관심도(링크)



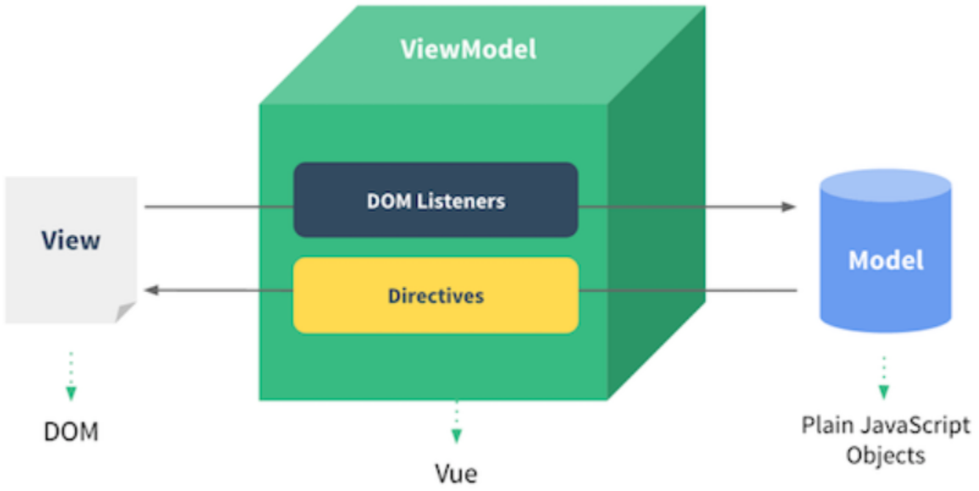
Vue.JS

- Vue.js(https://vuejs.org/)는 Evan You(https://evanyou.me/)가 만들었으며, 2014년 릴리즈를 시작으로 꾸준히 발전하고 있는 자바스크립트 프레임워크이다. Angular나 React보다 덜 알려져 있으나, Vue.js 포럼(https://kr.vuejs.org/v2/guide/join.html)에 한국어 카테고리가 추가될 정도로 인지도에서 상승 곡선을 그리는 추세입니다.
- Vue.js는 발음대로 철저히 뷰(View)에 최적화된 프레임워크이다. 컨트롤러 대신 뷰 모델을 가지는 MVVM(Model-View-ViewModel) 패턴을 기반으로 디자인되었으며, 컴포넌트(Components)를 사용하여 재사용이 가능한 UI들을 묶고 뷰 레이어를 정리하는 것을 가장 강력한 기능으로 꼽는다. 또한 템플릿(Template) 위주의 개발을 권장한다.
- Vue.JS의 특징
  1. Virtual DOM 으로 빠른 렌더링
  2. 경량 라이브러리 (작은 용량)
  3. Router, Bundler, state management 와 결합이 쉬움
  4. 간편한 Syntax 와 프로젝트 설정
  5. 훌륭한 개발자 커뮤니티와 지원

Vue.JS

MVVM 모델

- 웹페이지는 HTML DOM과 Javascript의 연합으로 만들어지게 되는데 HTML DOM이 View 역할을 하고, Javascript가 Model 역할을 한다.
- ViewModel이 없는 아키텍처에서는 getElementById 같은 DOM API를 직접 이용해서 모델과 뷰를 연결해야 한다. Javascript에서 컨트롤러 혹은 프리젠터 역할까지 하면서 코드의 분량이 증가하는 단점이 생기게 되는데 jQuery를 이용해 DOM에 데이터를 출력하는 코드들이 대부분 그랬다.
- ViewModel에 자바스크립트 객체와 DOM을 연결해 주면 ViewModel은 이 둘간의 동기화를 자동으로 처리한다. 이것이 Vuejs가 하는 역할이다. 즉, MVVM 모델의 VM을 Vue.js가 담당합니다.



Vue.JS

Vue.JS와 React 비교

1. 공통점

- 가상돔(Virtual DOM)을 사용합니다.
- 컴포넌트를 제공합니다.
- 뷰에만 집중을 하고 있고, 라우터,상태 관리를 위해서는 서드파티 라이브러리를 사용합니다.

	Vue	React
Fastest	23ms	63ms
Median	42ms	81ms
Average	51ms	94ms
95th Perc.	73ms	164ms
Slowest	343ms	453ms

2. 성능의 차이

- 10000개의 컴포넌트를 100번 렌더링 했을 때 결과
- React에서는 불필요한 업데이트를 방지할 때는 shouldComponentUpdate 라는 메소드를 통해서 최적화를 합니다. Vue 에서는 컴포넌트의 종속성이 렌더링 중 자동으로 추적되어 시스템에서 다시 렌더링 해야 하는 컴포넌트를 정확히 알고 있기 때문에 이 작업이 불필요 합니다.

Vue.JS와 React 비교

3. JSX vs Template

- React에서는 JSX를 사용하고, Vue에서는 Template을 사용합니다.
- Vue에서도 원한다면 JSX 를 사용 할 수 있다, 템플릿을 사용 할 때의 장점은 HTML 파일에서 바로 사용 할 수 있다는 점 입니다.

Vue.JS

Vue.JS의 Component

- Vue.JS는 단일 파일 컴포넌트를 추천합니다.
- 확장자가 vue인 파일로 컴포넌트를 만들고 HTML, Javascript, CSS 코드로 구성한다.
- CSS는 상위 CSS의 영향을 받지 않습니다. 빌드 시점에 고유한 셀렉터 이름으로 대체하는 방식을 사용하기 때문입니다.
- 레이아웃의 각 섹션이 트리 형태로 구성되듯이 이에 상응하는 컴포넌트를 트리 형태로 구성하여 개발할 수 있는 구조입니다.

Vue.JS 프로젝트 환경 설정

- 1.Node.js 설치하기

Node.js 를 현재 기준 LTS 버전인 v10 버전을 설치하세요. 노드 공식 홈페이지

- 2.Visual Studio Code 설치 및 Plug In 설치하기

VS Code 설치는 Visual Studio Code 에서 하실 수 있습니다.

Vue.JS 프로젝트 환경 설정

- 3.Visual Studio Code Plug In 설치하기

1. Vue.js Extension Pack

(https://marketplace.visualstudio.com/items?itemName=shekhardtu.vuejspack)

Vetur 도움말 https://flaviocopes.com/vue-vscode/

2. Vue.js Auto Import

(https://marketplace.visualstudio.com/items?itemName=ishiyama.vue-autoimport)

3. Vue 2 Snippets

https://marketplace.visualstudio.com/items?itemName=hollowtree.vue-snippets

4. className Completion in CSS

https://marketplace.visualstudio.com/items?itemName=zitup.classnametocss

Vue.JS 프로젝트 환경 설정

- 5.Vue CLI 설치하기

```
npm install -g @vue/cli
# OR
yarn global add @vue/cli
```

```
vue --version
```

- 6.Chrome에 Vue.js DevTools 설치

Webpack : 번들링 도구

- 번들링 도구
  - 번들링 도구는 Browserify, RequireJS, Webpack (https://webpack.js.org/)등이 대표적이다.
  - Vue-Cli 프로젝트에서는 내부에서는 Webpack을 사용한다.
  - 번들링 도구를 사용하면 require (또는 import)로 모듈을 불러 왔을 때 번들링 되면서 모듈들을 파일 하나로 합쳐 줍니다.
  - src/main.js를 시작으로 필요한 파일을 다 불러온 후에 파일 하나로 합쳐 주는 것이다.
  - css 파일도 Webpack의 css-loader가 불러온다.
  - File-loader는 웹폰트나 미디어 파일 등을 불러온다.
  - Babel-loader는 js 파일들을 불러 오면서 ES6로 작성된 코드를 ES5로 문법으로 변환해 줍니다.

ECMA Script 6
: ecma 2015

ECMAScript 6 소개

- ECMA(European Computer Manufacturers Association) Script는 JavaScript 프로그래밍 언어를 정의하는 국제 표준의 이름입니다.
- ECMA의 Technical Committee39(TC39)에서 논의 되었습니다.
- 현재 사용하는 대부분의 JavaScript는 2009년에 처음 제정되어 2011년에 개정된 ECMAScript 5.1 표준에 기반하고 있습니다.
- 2015년은 ES5(2009년)이래로 첫 메이저 업데이트가 승인된 해입니다.
- 이후 클래스 기반 상속, 데이터 바인딩(Object.observe), Promise 등 다양한 요구사항들이 도출 되었고 그 결과 2015년 6월에 대대적으로 업데이트된 ECMAScript 6 가 발표되었고, 매년 표준을 업데이트하는 정책에 따라 2016년 6월에 ECMAScript 7 까지 발표 되었습니다.
- ES6 = ECMAScript6 = ECMAScript 2015 = ES2015
- 최신 Front-End Framework인 React, Angular, Vue에서 권고하는 언어 형식

ES6 : const & let

1. const & let - 새로운 변수 선언 방식

- 블록단위 { } 로 변수의 범위가 제한 되었음
- const : 한번 선언한 값에 대해서 변경할 수 없음 (상수 개념)
- let : 한번 선언한 값에 대해서 다시 변경할 수 있음

const & let

```
const value = 30;
value = 40;
Uncaught TypeError: Assignment to constant variable.

var b = 30;
var b = 40;

let a = 20;
let a = 20;
Uncaught SyntaxError: Identifier 'a' has already been declared
```

ES5 : Hoisting

2. Hoisting

- Hoisting 이란 선언한 함수와 변수를 해석기가 있는 상단에 있는 것 처럼 인식한다.
- js 해석기는 코드의 라인 순서와 관계 없이 함수 선언식과 변수를 위한 메모리 공간을 먼저 확보한다.
- 따라서, function a()와 var는 코드의 최상단으로 끌어 올려진 것(hoisted) 처럼 보인다.

Hoisting

```
//function statement (함수 선언문)
function willBeOveridden() {
    return 10;
}
willBeOveridden(); //5
function willBeOveridden() {
    return 5;
}
```

```
//function expression(함수 표현식)
var sum = function() {
    return 10 + 20;
}
```

ES5 : Hoisting

2. Hoisting

- 아래와 같은 코드를 실행할 때 자바스크립트 해석기가 어떻게 코드 순서를 재조정 할까요?

Hoisting

```
var sum = 5;
sum = sum + i;

function sumAllNumbers() {
    //…
}
var i = 10;
```

Hoisting

```
//#1 - 함수 선언식과 변수 선언을 hoisting
var sum;
function sumAllNumbers() {
    //..
}
var i;

//#2 - 변수 대입 및 할당
sum = 5;
sum = sum + i;
i = 10;
```

ES6 : const & let

3. const keyword

- const로 지정한 값 변경 불가능
- 하지만, 객체나 배열의 내부는 변경할 수 있다.

const

```
const obj = {};
obj.value = 40
console.log(obj);

const arr = {};
arr.push(20);
console.log(arr);
```

ES6 : 변수의 Scope

3. let keyword : 변수의 scope

- 기존 자바스크립트(ES5)는 { } 에 상관 없이 스코프가 설정됨.
- ES6는 { } 단위로 변수의 스코프가 제한됨

ES5 : 변수의 scope

```
var sum = 0;
for(var i = 1; i <= 5; i++) {
    sum = sum + i;
}
console.log(sum);    //15
console.log(i);      //6
```

ES6 : 변수의 scope

```
let sum = 0;
for(let i = 1; i <= 5; i++) {
    sum = sum + i;
}
console.log(sum);    //15
console.log(i);      //Uncaught Reference Error: i is not defined
```

ES6 : const & let

3. const & let

const & let

```
function f() {
  {
      let x;
      {
       //새로운 블록 안에 새로운 x의 scope가 생김
       const x = "sneaky";
       x = "foo";  //위에 이미 const로 x를 선언했으므로 다시 값을 대입하면 에러 발생
      }
      //이전 블록 범위로 돌아왔기 때문에 'let x'에 해당하는 메모리에 값을 대입
      x = "bar";
      let x = "inner"; //Uncaught SyntaxError: Identifier 'x' has already been declare
  }
}
```

ES6 : Arrow Function

4. Arrow Function - 화살표 함수

- 함수를 정의할 때 function 키워드를 사용하지 않고 => 로 대체
- 흔히 사용하는 콜백함수의 문법을 간결화

ES5 : 함수 정의

```
var sum = function(a, b) {
     return a + b;
}
sum(10,20);
```

ES6 : 함수 정의

```
var sum = (a, b) => {
     return a + b;
}
sum(10,20);

var sum2 = (a,b) => a + b;
sum2(10,20);
```

ES6 : Enhanced Object Literals

5. Enhanced Object Literals – 향상된 객체 리터럴

- 객체의 속성을 메서드로 사용할 때 function 예약어를 생략하고 생성 가능

Enhanced Object Literals

```
var dictionary {
  words: 100;
  //ES5
  lookup: function() {
     console.log("find words");
  },
  //ES6
  lookup() {
     console.log("find words");
  }
}
```

ES6 : Enhanced Object Literals

5. Enhanced Object Literals – 향상된 객체 리터럴

- 객체의 속성 이름과 값의 이름이 같으면 아래와 같이 축약 가능

Enhanced Object Literals

```
var figures = 10;
var dictionary = {
   //figures: figures
   figures
}
```

ES6 축약코딩기법

- 1.삼항 조건 연산자 (The Ternary Operator)

기존

```
const x = 20;
let answer;
if (x > 10) {
    answer = 'greater than 10';
} else {
    answer = 'less than 10';
}
```

축약기법

```
const answer = x > 10 ? 'greater than 10' : 'less than 10';
```

React에서 사용

```
//특정 버튼을 state 값에 따라 보여지게 할 경우에 이렇게 사용할 수 있음
{editable ? (
    <a onClick={() => this.save(record.key)}> </a>
) : (
    <a onClick={() => this.edit(record.key)}> </a>
)}
```

ES6 축약코딩기법

- 2.간략 계산법 (Short-circuit Evaluation)
- 기존의 변수를 다른 변수에 할당하고 싶은 경우, 기존 변수가 null, undefined 또는 empty 값이 아닌 것을 확인 해야 합니다. (위 세가지 일 경우 에러가 뜹니다) 이를 해결 하기 위해서 긴 if문을 작성 하거나 축약 코딩으로 한 줄에 끝낼 수 있습니다.

기존

```
if (variable1 != null || variable1 !== undefined || variable1 !== '') {
    let variable2 = variable1;
}
```

축약기법

```
const variable2 = variable1 || 'new';
```

Console에서 확인

```
let variable1;
let variable2 = variable1 || '';
console.log(variable2 === '');  //print true

let variable3 = 'foo';
let variable4 = variable3 || 'foo';
console.log(variable4 === 'foo');  //print true
```

ES6 축약코딩기법

- 3. 변수 선언
- 함수를 시작하기 전 먼저 필요한 변수들을 선언하는 것은 현명한 코딩 방법입니다. 축약 기법을 사용하면 여러 개의 변수를 더 효과적으로 선언함으로 시간과 코딩 스페이스를 줄일 수 있습니다.

기존

```
let x;
let y;
let z = 3;
```

축약기법

```
let x,y,z = 3;
```

ES6 축약코딩기법

- 4.For 루프

기존

```
for (let i=0; i < msgs.length; i++)
```

축약기법

```
for (let value of msgs)
```

Array.forEach 축약기법

```
function logArrayElements(element,index, array) {
    console.log('a[' + index + '] = ' + element);
}
[2,5,9].forEach(logArrayElements);
//a[0] = 2
//a[1] = 5
//a[2] = 9
```

ES6 축약코딩기법

- 5.간략 계산법 (Short-circuit Evaluation)
- 기본 값을 부여하기 위해 파라미터의 null 또는 undefined 여부를 파악할 때 short-circuit evaluation 방법을 이용해서 한줄로 작성하는 방법이 있습니다.
- Short-circuit evaluation 이란?

  두가지의 변수를 비교할 때, 앞에 있는 변수가 false 일 경우 결과는 무조건 false 이기 때문에 뒤의 변수는 확인 하지 않고 return 시키는 방법.
- 아래의 예제에서는 process.env.DB_HOST 값이 있을 경우 dbHost 변수에 할당하지만, 없으면 localhost를 할당 합니다.

기존

```
let dbHost;
if (process.env.DB_HOST) {
    dbHost = process.env.DB_HOST;
} else {
    dbHost = 'localhost';
}
```

축약기법

```
Const dbHost = process.env.DB_HOT || 'localhost';
```

ES6 축약코딩기법

- 6.묵시적 반환(Implicit Return)
- return 은 함수 결과를 반환 하는데 사용되는 명령어 입니다.
- 한 줄로만 작성된 arrow 함수는 별도의 return 명령어가 없어도 자동으로 반환 하도록 되어 있습니다.
- 다만, 중괄호({})를 생략한 함수여야 return 명령어도 생략 할 수 있습니다
- 한 줄 이상의 문장(객체 리터럴)을 반환 하려면 중괄호({})대신 괄호(())를 사용해서 함수를 묶어야 합니다. 이렇게 하면 함수가 한 문장으로 작성 되었음을 나타낼 수 있습니다.

기존

```
function calcCircumference(diameter) {
  return Math.PI * diameter
}
```

축약기법

```
calcCircumference = diameter => (
  Math.PI * diameter;
)
```

ES6 축약코딩기법

- 7.파라미터 기본 값 지정하기(Default Parameter Values)
- 기존에는 if 문을 통해서 함수의 파라미터 값에 기본 값을 지정해 줘야 했습니다. ES6에서는 함수 선언문 자체에서 기본값을 지정해 줄 수 있습니다.

기존

```
function volume(l, w, h) {
  if (w === undefined)
    w = 3;
  if (h === undefined)
    h = 4;
  return l * w * h;
}
```

축약기법

```
volume = (l, w = 3, h = 4 ) => (l * w * h);
volume(2) //output: 24
```

ES6 축약코딩기법

- 8.템플릿 리터럴 (Template Literals)
- 백틱(backtick) 을 사용해서 스트링을 감싸고, ${}를 사용해서 변수를 담아 주면 됩니다.

기존

```
const welcome = 'You have logged in as ' + first + ' ' + last + '.'

const db = 'http://' + host + ':' + port + '/' + database;
```

축약기법

```
const welcome = `You have logged in as ${first} ${last}`;

const db = `http://${host}:${port}/${database}`;
```

ES6 축약코딩기법

- 9.비구조화 할당 (Destructuring Assignment)
- 데이터 객체가 컴포넌트에 들어가게 되면, unpack 이 필요합니다.

Destructuring Assignment 기본 문법

```
let a, b, rest;
[a, b] = [1, 2];
[a, b, ...rest] = [10, 20, 3, 4, 5];

let foo = ["one", "two", "three"];
let [foo1, foo2, foo3] = foo;
```

갑 교환 (Swapping)

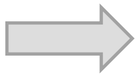
```
let a = 1;
let b = 3;

[a, b] = [b, a];
```

객체 분해

```
let obj = {p: 42, q: true};
let {p, q} = obj;
```

기존

```
const observable = require('mobx/observable');
const action = require('mobx/action');
const runInAction = require('mobx/runInAction');

const store = this.props.store;
const form = this.props.form;
const loading = this.props.loading;
const errors = this.props.errors;
const entity = this.props.entity;
```



축약기법

```
import { observable, action, runInAction } from 'mobx';

const { store, form, loading, errors, entity } = this.props;
```

ES6 축약코딩기법

- 10.전개연산자 (Spread Operator) #1
- ES6에서 소개된 전개 연산자는 자바스크립트 코드를 더 효율적으로 사용 할 수 있는 방법을 제시합니다. 간단히는 배열의 값을 변환하는데 사용할 수 있습니다. 전개 연산자를 사용하는 방법은 점 세개(...)를 붙이면 됩니다.

기존

```
// joining arrays
const odd = [1, 3, 5];
const nums = [2 ,4 , 6].concat(odd);

// cloning arrays
const arr = [1, 2, 3, 4];
const arr2 = arr.slice();
```

축약기법

```
// joining arrays
const odd = [1, 3, 5 ];
const nums = [2 ,4 , 6, ...odd];
console.log(nums); // [ 2, 4, 6, 1, 3, 5 ]

// cloning arrays
const arr = [1, 2, 3, 4];
const arr2 = [...arr];
```

ES6 축약코딩기법

- 11.전개연산자 (Spread Operator) #2
- concat()함수와는 다르게 전개 연산자를 이용하면 하나의 배열을 다른 배열의 아무 곳에나 추가할 수 있습니다.

축약기법

```
const odd = [1, 3, 5 ];
const nums = [2, ...odd, 4 , 6];
```

- 전개 연산자는 ES6의 구조화 대입법(destructuring notation)와 함께 사용할 수도 있습니다.

축약기법

```
const { a, b, ...z } = { a: 1, b: 2, c: 3, d: 4 };
console.log(a) // 1
console.log(b) // 2
console.log(z) // { c: 3, d: 4 }
```

ES6 축약코딩기법

- 12.필수(기본) 파라미터 (Mandatory Parameter)
- 기본적으로 자바스크립트는 함수의 파라미터 값을 받지 않았을 경우, undefined로 지정합니다. 다른 언어들은 경고나 에러 메시지를 나타내기도 하죠. 이런 기본 파라미터 값을 강제로 지정하는 방법은 if 문을 사용해서 undefined일 경우 에러가 나도록 하거나, ‘Mandatory parameter shorthand’을 사용하는 방법이 있습니다.

기존

```
function foo(bar) {
  if(bar === undefined) {
    throw new Error('Missing parameter!');
  }
  return bar;
}
```

축약기법

```
let mandatory = () => {
  throw new Error('Missing parameter!');
}

let foo = (bar = mandatory()) => {
  return bar;
}
```

ES6 축약코딩기법

- 13. Promise 객체
- “A promise is an object that may produce a single value some time in the future”
- Promise는 자바스크립트 비동기 처리에 사용되는 객체입니다. 여기서 자바스크립트의 비동기 처리란 ‘특정 코드의 실행이 완료될 때까지 기다리지 않고 다음 코드를 먼저 수행하는 자바스크립트의 특성’을 의미합니다.
- new Promise(function(resolve, reject) { ... });
- Promise 객체를 생성하면 resolve와 reject의 함수를 전달받는다.
- 작업이 성공하면 resolve함수를 호출하여 resolve의 인자값을 then으로 받게 되고
- 작업에 실패하면 reject 함수를 호출하여 reject의 인자값을 catch로 받게 된다.
- 성공, 실패 여부에 관계없이 항상 처리 되게 하려면 finally로 받아서 처리할 수도 있다.

Promise

```
new Promise(function (resolve, reject) {
}).then(function (resolve) {
    //resolve 값 처리
}).catch(function (reject) {
    //reject 값 처리
}).finally(function(){
    //항상 처리
});
```

ES6 축약코딩기법

- 14. import / export
- ES6(ECMA2015)부터는 import / export 라는 방식으로 모듈을 불러오고 내보낸다.
- Export는 내부 스크립트 객체를 외부 스크립트로 모듈화 하는 것이며, export 하지 않으면 외부 스크립트에서 import를 사용할 수 없다.

app.js (export)

```
export const myNumbers = [1, 2, 3, 4];
const animals = ['Panda', 'Bear', 'Eagle'];

export default function myLogger() {
  console.log(myNumbers, pets);
}

export class Alligator {
  constructor() {
    // ...
  }
}
```

import

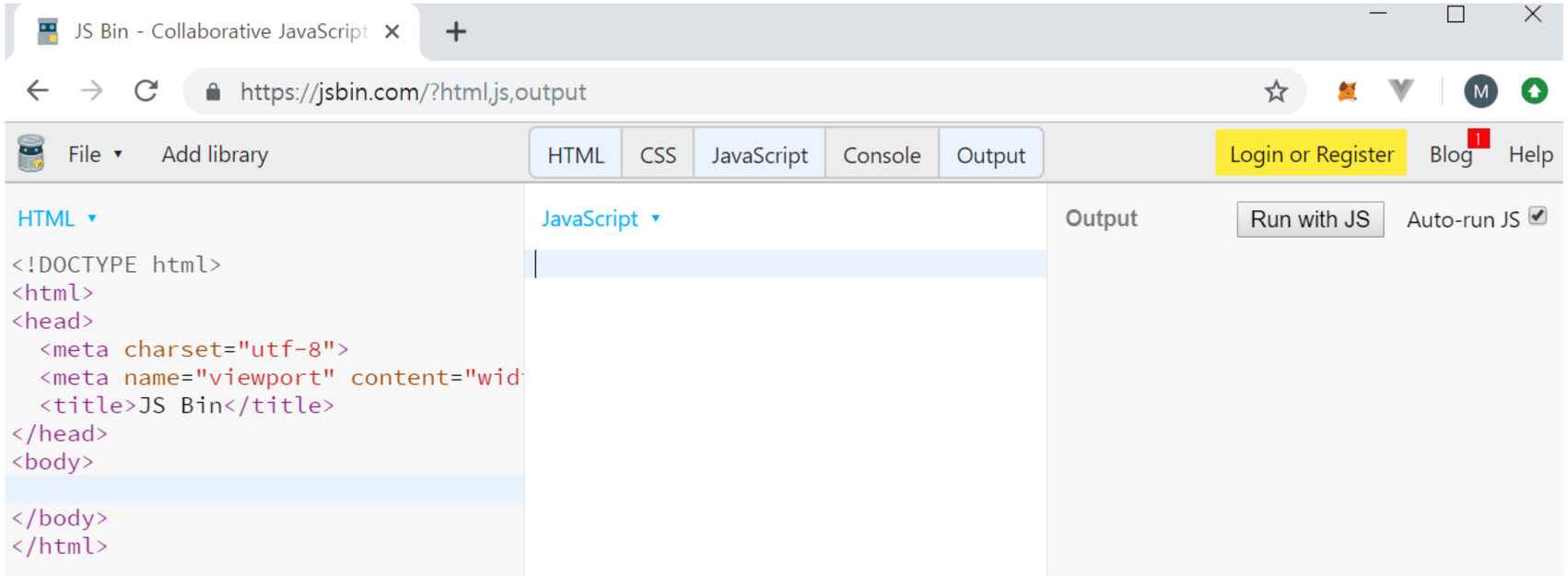
```
//Importing with alias
import myLogger as Logger from 'app.js';

//Importing all exported members
import * as utils from ‘app.js’;

utils.myLogger();

//Importing a module with a default member
import myLogger from ‘app.js’;

Import myLogger, { Alligator, myNumbers } from 'app.js';
```

- https://www.digitalocean.com/community/tutorials/js-modules-es6

Vue.js 시작하기

Vue 시작하기

- Vue를 시작할 때는 CDN(content delivery network)에서 스크립트 파일을 불러와서 하는 방법이 있고, CLI(커맨드 라인 인터페이스)를 사용 하여 프로젝트를 구성하는 방법이 있습니다.
- JSBin(https://jsbin.com/?html,output) 열기



- unpkg에서 제공하는 링크를 사용하면 됩니다. 이 링크를 사용하면 최신 버전으로 로드 합니다.
- https://unpkg.com/vue/dist/vue.js

Vue 시작하기

- Vue.js HelloWorld 예제 작성하기

html

```
<body>
  <div id="app">
    <h1>Hello, {{ name }}</h1>
  </div>
  <script src="https://unpkg.com/vue/dist/vue.js"></script>
</body>
```

javascript

```
 // 새로운 뷰를 정의합니다
var app = new Vue({
  el: '#app', // 어떤 엘리먼트에 적용을 할 지 정합니다
  // data 는 해당 뷰에서 사용할 정보를 지닙니다
  data: {
    name: 'Vue'
  }
});
```

- console에서 app.name = "뷰" 라고 입력을 하면 output 화면에 바로 값이 바뀌어서 렌더링 됩니다. one-way binding이 되어서 값을 업데이트 하면 바로 반영이 됩니다.

Vue Directive

- 디렉티브를 그대로 번역하면 "지시문" 이라는 뜻 입니다. Vue 엘리먼트에서 사용되는 특별한 속성입니다. 엘리먼트에게 이러이러하게 작동해라! 하고 지시를 해주는 지시문입니다.

1. v-text (one-way-binding)

: {{ }} 를 사용하는 대신에 v-text 라는 디렉티브를 사용합니다.

데이터 → 뷰 의 형태로 바인딩이 되어 있어서, 데이터의 값이 변하면 뷰가 업데이트가 된다.

html

```
<div id="app">
  <h1>Hello, <span v-text="name"></span></h1>
</div>
```

2. v-html

: html을 렌더링 해야 할 때도 있습니다."여기서 렌더링 할 건 html 형식이야" 라는걸 지정하기 위해서 v-html이라는 디렉티브를 사용합니다.

html

```
<div id="app">
  <h1>Hello, <span v-html="name"></span></h1>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    name: '<i>Vue</i>'
  }
});
```

Vue Directive

3. v-show

: 해당 엘리먼트가 보여질지, 말지를 true / false 값으로 지정 할 수 있습니다.

Console을 열어서 app.visible = false 라고 입력해 보세요.

html

```
<div id="app">
  <h1>Hello,
    <span v-show="visible"
        v-html="name"></span></h1>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    name: '<i>Vue</i>',
    visible: true
  }
});
```

4. v-if / v-else / v-else-if

: v-if 디렉티브를 사용할때 그 아래에 v-else 디렉티브를 사용하는 엘리먼트를 넣어주면, 조건문이 만족하지 않을 때 보여집니다.

v-else-if는 첫번째 조건문의 값이 참이 아닐 때, 다른 조건문을 체크하여 다른 결과물을 보여줄 수 있게 해줍니다.

html

```
<div id="app">
  <h1 v-if="value > 5">value가 5보다 큽니다</h1>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    value: 0
  }
});
```

Vue Directive

4-1. v-if / v-else / v-else-if

html

```
<div id="app">
  <h1 v-if="value > 5">value가 5보다 큽니다</h1>
  <h1 v-else>value가 5보다 작아요</h1>
</div>
```

html

```
<div id="app">
  <h1 v-if="value > 5">value가 5보다 큽니다</h1>
  <h1 v-else-if="value === 5">value가 5 </h1>
  <h1 v-else>value가 5보다 작아요</h1>
</div>
```

5. v-bind

: html 엘리먼트의 속성의 값을 Vue 엘리먼트의 데이터로 설정하고 싶다면 v-bind 디렉티브를 사용합니다.예를 들어 와 같은 형식으로 하면 됩니다. v-bind: 뒤에 속성의 이름을 넣어줍니다.

html

```
<div id="app">
  <h1>Hello, {{ name }}</h1>
  <img v-bind:src="vuelogo"/>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    name: 'Vue',
    vuelogo: 'https://vuejs.org/images/logo.png'
  }
});
```

Vue Directive

5-1. v-bind의 응용

: Vue인스턴스의 data 안에 flag 값에 따라 다른 이미지를 보여주기

조건 ? true 일 때의 값 : false 일 때의 값

html

```
<div id="app">
  <h1>Hello, {{ name }}</h1>
  <img :src="flag ? vuelogo : anglogo"/>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    name: 'Vue',
    flag: true,
    vuelogo: 'https://vuejs.org/images/logo.png',
    anglogo:
'https://angular.io/assets/images/logos/angular/angular.svg'
  }
});
```

6. v-for

: 반복할 태그에서 사용하면 된다.

html

```
<div id="app">
    <h2>오늘 할 일</h2>
    <ul>
      <li v-for="todo in todos">
       {{ todo.text }}
      </li>
    </ul>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    todos: [
      { text: 'Vue.js 튜토리얼 작성하기' },
      { text: 'Webpack2 알아보기' },
      { text: '사이드 프로젝트 진행하기' }
    ]
  }
});
```

Vue Directive

6-1. v-for

: index 값 받아오기

html

```
<div id="app">
    <h2>오늘 할 일</h2>
    <ul>
      <li v-for="(todo, index) in todos"> {{index}} {{ todo.text }}</li>
    </ul>
</div>
```

7. v-model(two-way-binding)

: 뷰 ⇄ 데이터 형태로 바인딩하여 데이터가 양 방향으로 흐르게 해주는 것 입니다. 즉, 데이터에 있는 값이 뷰에 나타나고, 이 뷰의 값이 바뀌면 데이터의 값도 바뀌는 것입니다.

html

```
  <div id="app">
    <h1>Hello, {{ name }}</h1>
    <input type="text" v-model="name"/>
  </div>
```

javascript

```
 var app = new Vue({
  el: '#app',
  data: {
    name: 'Vue'
  }
});
```

Vue Directive

7-1. v-model(two-way-binding)의 응용

: checkbox를 만들어서 그 checkbox의 값에 따라서 다른 image를 보여준다.

checkbox의 속성에 v-model="flag" 설정한다.

html

```
<div id="app">
  <h1>Hello, {{ name }}</h1>
  <h3>
   <input type="checkbox" v-model="flag"/>
    Logo 이미지
  </h3>
  <img :src="flag ? vuelogo : anglogo"/>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    name: 'Vue',
    flag: true,
    vuelogo: 'https://vuejs.org/images/logo.png',
    anglogo:
'https://angular.io/assets/images/logos/angular/angular.svg'
  }
});
```

Vue Directive

8. v-on (Event 핸들링)

: 데이터 모델에 number라는 변수를 만들고, 값을 증가시키는 increment, 감소시키는 decrement 메소드들을 준비합니다. v-on: 을 @ 로 대체할 수 있습니다.

html

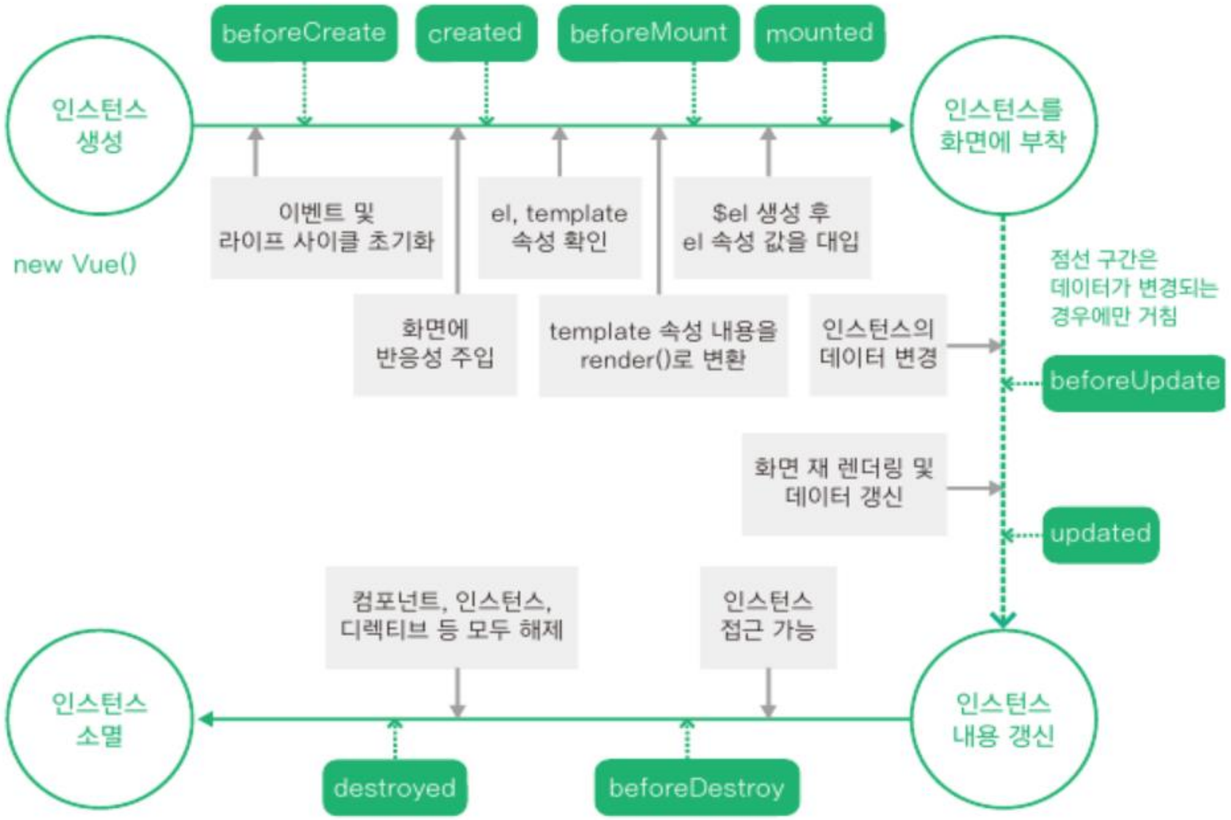
```
<div id="app">
  <h1>카운터: {{ number }}</h1>
  <button v-on:click="increment">증가</button>
  <button v-on:click="decrement">감소</button>
</div>
```

javascript

```
var app = new Vue({
  el: '#app',
  data: {
    number: 0
  },
  // app 뷰 인스턴스를 위한 메소드들
  methods: {
    increment: function() {
      // 인스턴스 내부의 데이터모델에 접근 할 땐, this 를 사용한다
      this.number++;
    },
    decrement: function() {
      this.number--;
    }
  }
});
```

Vue component life cycle

- 뷰 컴포넌트 라이프 사이클
  : 컴포넌트의 상태에 따라 호출할 수 있는 속성들을 라이프 사이클 속성이라고 합니다. 라이프 사이클 속성에는 beforeCreated, created, beforeMount, mounted 등 인스턴스의 생성,변경,소멸과 관련 되어 총 8개가 있습니다.



뷰 라이프 사이클 다이어그램

라이프 사이클 4 단계

인스턴스 생성 -> 생성된 인스턴스를 화면에 부착 -> 화면에 부착된 인스턴스 내용이 갱신 -> 인스턴스가 제거되는 소멸

Vue life cycle

- 라이프 사이클 예제

: updated 라이프 사이클 혹은 뷰 인스턴스에서 데이터 변경이 일어나 화면이 다시 그려졌을 때 호출된다. update의 앞 단계인 mounted 단계에서 기존에 정의된 data 속성의 message 값을 변경해봄

html

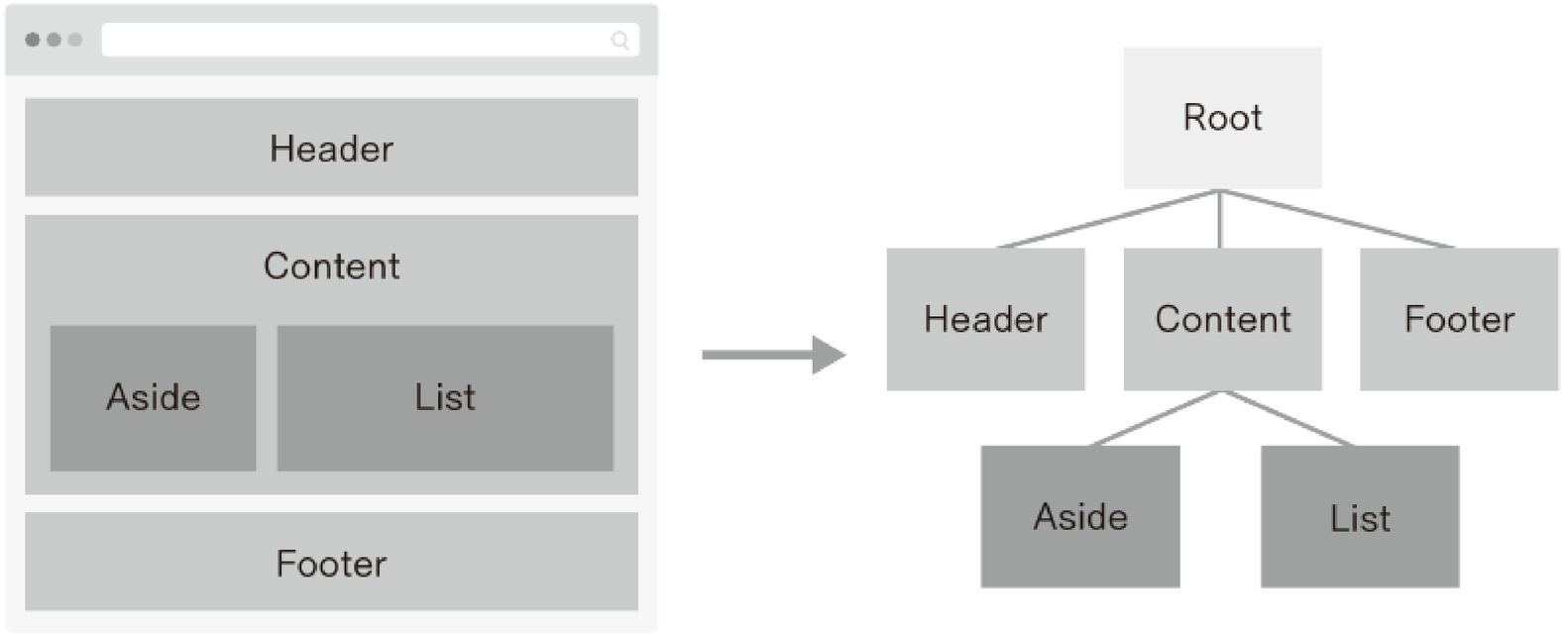
```
<div id="app">
     {{ message }}
</div>
```

javascript

```
new Vue({
        el: '#app',
        data: {
          message: 'Hello Vue.js!'
        },
        beforeCreate: function() {
          console.log("beforeCreate");
        },
        created: function() {
          console.log("created");
        },
        mounted: function() {
          console.log("mounted");
          this.message = 'Hello Vue!';
        },
        updated: function() {
          console.log("updated");
        }
});
```

Vue component

- Vue 컴포넌트
- 뷰가 가지는 큰 특징은 컴포넌트 기반 프레임워크 입니다. 뷰의 컴포넌트를 조합하여 화면을 구성할 수 있습니다.
- 컴포넌트 기반 방식으로 개발하는 이유는 코드를 재사용 하기가 쉽기 때문이다.
- 왼쪽 그림은 화면 전체는 3개의 컴포넌트로 분할한 후 분할된 1개의 컴포넌트에서 다시 2개의 하위 컴포넌트로 분할 것입니다. 오른쪽 그림은 각 컴포넌트 간의 관계를 나타냅니다. 컴포넌트 간의 관계는 뷰에서 화면을 구성하는데 중요한 역할을 하며, 웹페이지 화면을 설계할 때도 이런 골격을 유지하면서 설계를 해야 합니다.



화면을 컴포넌트로 구조화한 컴포넌트 간 관계도

Vue component

- 컴포넌트 등록하기

: 컴포넌트를 등록하는 방법은 전역과 지역 두가지가 있습니다. 지역(Local) 컴포넌트는 특정 인스턴스에서만 유효한 범위를 갖고, 전역(Global) 컴포넌트는 여러 인스턴스에서 공통으로 사용할 수 있습니다.

- 전역(Global) 컴포넌트 등록

: 전역 컴포넌트를 모든 인스턴스에 등록 하려면 Vue 생성자에서 component()를 호출하여 수행하면 됩니다.

html

```
<div id="app">
      <button>컴포넌트 등록</button>
      <my-component></my-component>
</div>
```

javascript

```
Vue.component('my-component', {
        template: '<div>전역 컴포넌트가 등록되었습니다!</div>'
});

new Vue({
   el: '#app'
});
```

Vue component

- 지역(Local) 컴포넌트 등록

: 지역 컴포넌트 등록은 인스턴스에 components 속성을 추가하고 등록할 컴포넌트 이름과 내용을 정의하면 됩니다.

: 변수 cmp에는 template 속성으로 화면에 나타낼 컴포넌트의 내용을 정의했습니다.

html

```
<div id="app">
      <button>컴포넌트 등록</button>
      <my-local-component></my-local-component>
</div>
```

javascript

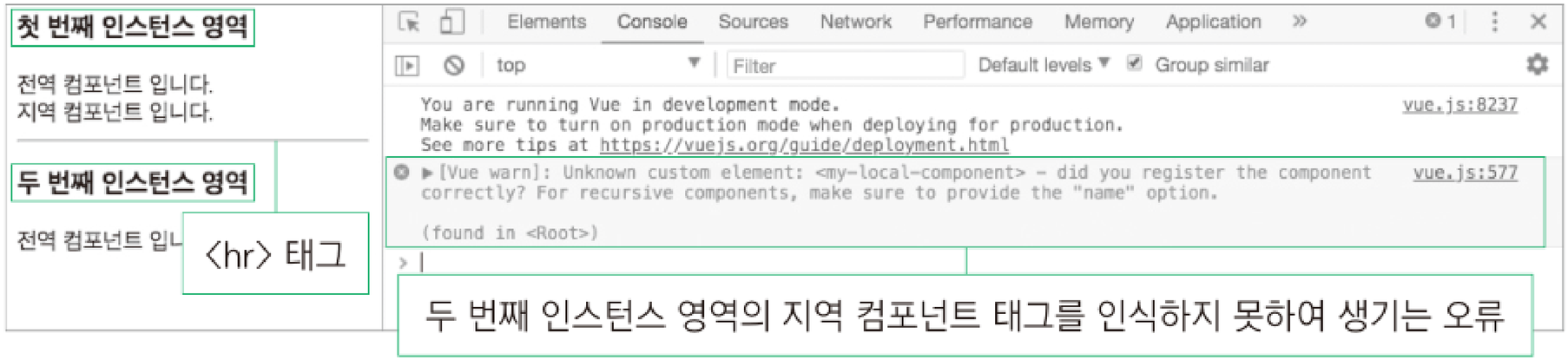
```
var cmp = {
      // 컴포넌트 내용
      template: '<div>지역 컴포넌트가 등록되었습니다!</div>'
};

new Vue({
        el: '#app',
        components: {
          'my-local-component': cmp
        }
});
```

Vue component

- 인스턴스 유효 범위와 지역 컴포넌트, 전역 컴포넌트 간 관계 확인하기

: 전역 컴포넌트는 인스턴스를 새로 생성할 때마다 인스턴스에 components 속성으로 등록할 필요없이 한 번 등록하면 모든 인스턴스에서 사용할 수 있습니다. 반대로 지역 컴포넌트는 새 인스턴스를 생성할 때마다 등록해 줘야 합니다.

: 첫번째 인스턴스 영역에는 전역,지역 컴포넌트가 모두 정상적으로 나타났습니다. 하지만 두번째 인스턴스 영역에는 전역 컴포넌트만 나타나고, 지역 컴포넌트는 나타나지 않았습니다.



Vue component

- 인스턴스 유효 범위와 지역 컴포넌트, 전역 컴포넌트 간 관계 확인하기

html

```
<div id="app">
  <h3>첫 번째 인스턴스 영역</h3>
  <my-global-component></my-global-component>
  <my-local-component></my-local-component>
</div>
<hr>
<div id="app2">
  <h3>두 번째 인스턴스 영역</h3>
  <my-global-component></my-global-component>
  <my-local-component></my-local-component>
</div>
```

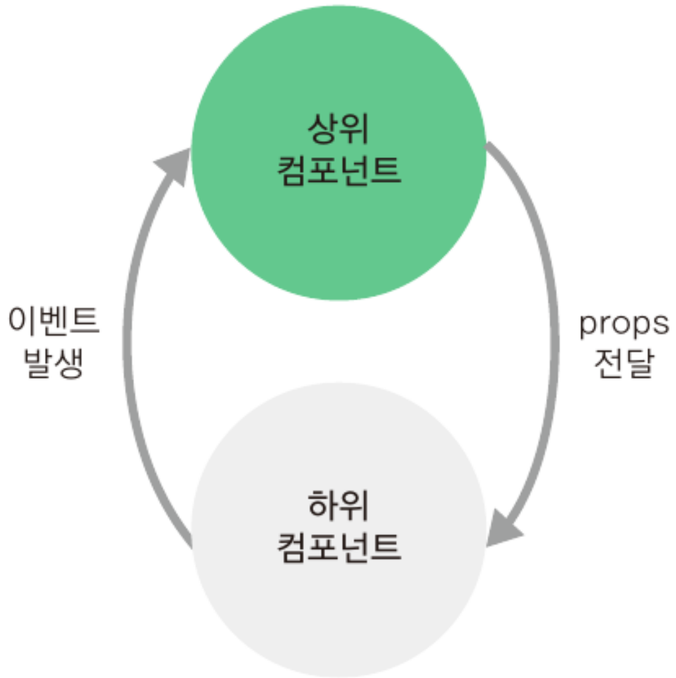
javascript

```
// 전역 컴포넌트 등록
    Vue.component('my-global-component', {
      template: '<div>전역 컴포넌트 입니다.</div>'
    });

    // 지역 컴포넌트 내용
    var cmp = {
      template: '<div>지역 컴포넌트 입니다.</div>'
    };

    new Vue({
      el: '#app',
      // 지역 컴포넌트 등록
      components: {
        'my-local-component': cmp
      }
    });

    // 두 번째 인스턴스
    new Vue({
      el: '#app2'
    });
```

Vue component

- 컴포넌트 간의 통신

: 컴포넌트는 각각 고유한 유효 범위를 가지고 있기 때문에 직접 다른 컴포넌트의 값을 참조할 수 없습니다. 가장 기본적인 데이터 전달 방법은 상위(부모)- 하위(자식) 컴포넌트 간의 데이터 전달 방법입니다.



상위 - 하위 컴포넌트 간 통신 방식

- 뷰에서 상위 - 하위 컴포넌트 간에 데이터를 전달하는 기본적인 구조를 나타냅니다.
- 상위에서 하위로는 props라는 특별한 속성을 전달합니다. 그리고 하위에서 상위로는 기본적으로 이벤트만 전달 할 수 있습니다.

Vue component

- 상위에서 하위 컴포넌트로 데이터 전달하기 : props

: props는 상위 컴포넌트에서 하위 컴포넌트로 데이터를 전달할 때 사용하는 속성입니다.

: props 속성을 사용하려면 먼저 하위 컴포넌트의 속성에 정의합니다.

```
Vue.component('child-component', {
  props: ['props 속성 이름'],
});
```

하위 컴포넌트의 props 속성 정의 방식

: 상위 컴포넌트의 HTML 코드에 등록된 child-component 컴포넌트 태그에 v-bind 속성을 추가합니다.

```
<child-component v-bind:props 속성 이름="상위 컴포넌트의 data 속성"></child-component>
```

상위 컴포넌트의 HTML 코드

: v-bind 속성의 왼쪽 값으로 하위 컴포넌트에서 정의한 props 속성을 넣고, 오른쪽 값으로 하위 컴포넌트에 전달할 상위 컴포넌트의 data 속성을 지정합니다.

Vue component

- 상위에서 하위 컴포넌트로 데이터 전달하기 : props 속성

html

```
<div id="app">
    <!-- 팁 : 오른쪽에서 왼쪽으로 속성을 읽으면 더 수월합니다. -->
    <child-component v-bind:props-data="message"></child-component>
</div>
```

javascript

```
    Vue.component('child-component', {
      props: ['propsData'],
      template: '<p>{{ propsData }}</p>',
    });

    new Vue({
      el: '#app',
      data: function() {
        return {
          message: 'Passed from Parent Component'
        }
      },
    });
```

Vue component

- 하위에서 상위 컴포넌트로 이벤트 전달하기

: 하위 컴포넌트에서 상위 컴포넌트로의 통신은 이벤트를 발생시켜 (event emit) 상위 컴포넌트에 신호를 보내면 됩니다. 상위 컴포넌트에서 하위 컴포넌트의 특정 이벤트가 발생하기를 기다리고 있다가 하위 컴포넌트에서 특정 이벤트가 발생하면 상위 컴포넌트에서 해당 이벤트를 수신하여 상위 컴포넌트의 메서드를 호출하는 것입니다.

- 이벤트 발생과 수신 형식

: 이벤트 발생과 수신은 $emit()과 v-on: 속성을 사용하여 구현합니다.

```
// 이벤트 발생
this.$emit('이벤트명');
```

$emit()을 이용한 이벤트 발생

```
// 이벤트 수신
<child-component v-on:이벤트명="상위 컴포넌트의 메서드명"></child-component>
```

v-on: 속성을 이용한 이벤트 수신

: $emit()을 호출하면 괄호 안에 정의된 이벤트가 발생합니다. 일반적으로 $emit()을 호출하는 위치는 하위 컴포넌트의 특정 메서드 내부입니다. 따라서 $emit()을 호출할 때 사용하는 this는 하위 컴포넌트를 가리킵니다.

Vue component

- 하위에서 상위 컴포넌트로 이벤트 전달하기

html

```
<div id="app">
      <child-component v-on:show-log="printText">
      </child-component>
</div>
```

javascript

```
Vue.component('child-component', {
        template: '<button v-on:click="showLog">show</button>',
        methods: {
          showLog: function() {
            this.$emit('show-log');
          }
        }
      });

      new Vue({
        el: '#app',
        data: {
          message: 'Hello Vue! passed from Parent Component'
        },
        methods: {
          printText: function() {
            console.log("received an event");
          }
        }
      });
```

할 일 관리(Todo App)
: 프로젝트 시작하기

Todo App : 프로젝트 생성

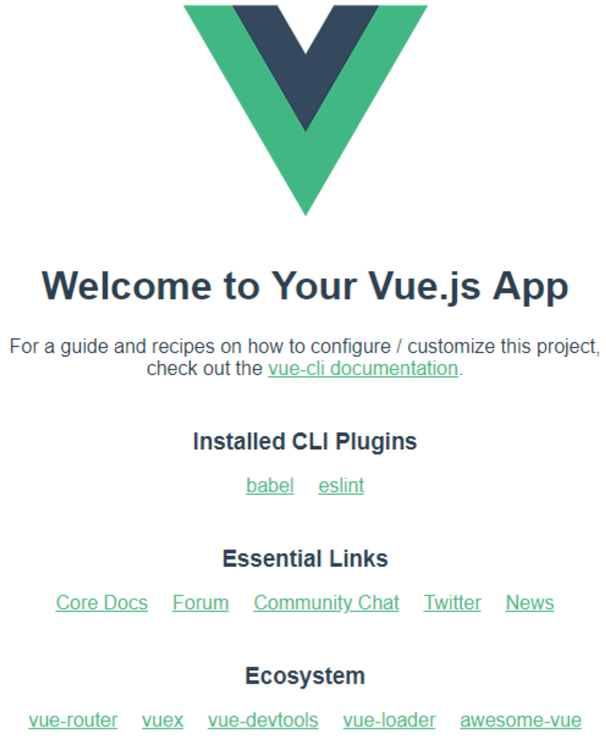
1. 새로운 프로젝트 생성 : vue-todo

```
vue create vue-todo
```

```
 Vue CLI v3.4.0
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
  Manually select features
```

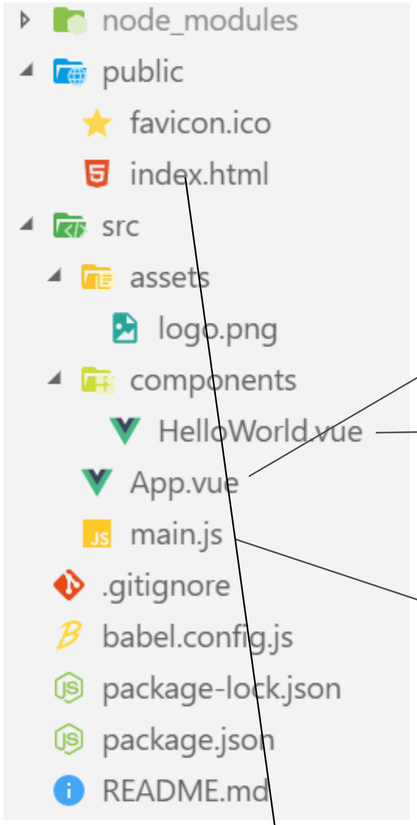
```
cd vue-todo

npm run serve
```

```
App running at:
  - Local:   http://localhost:8081/
  - Network: http://192.168.0.9:8081/
```



Vue.js 프로젝트 구조

- Vue-todo 프로젝트는 다음과 같이 파일들로 구성되어 있을 것입니다.



public/index.html

```
<div id="app"></div>
```

src/App.vue

```
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'
export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>

<style>…
```

src/components/HelloWorld.vue

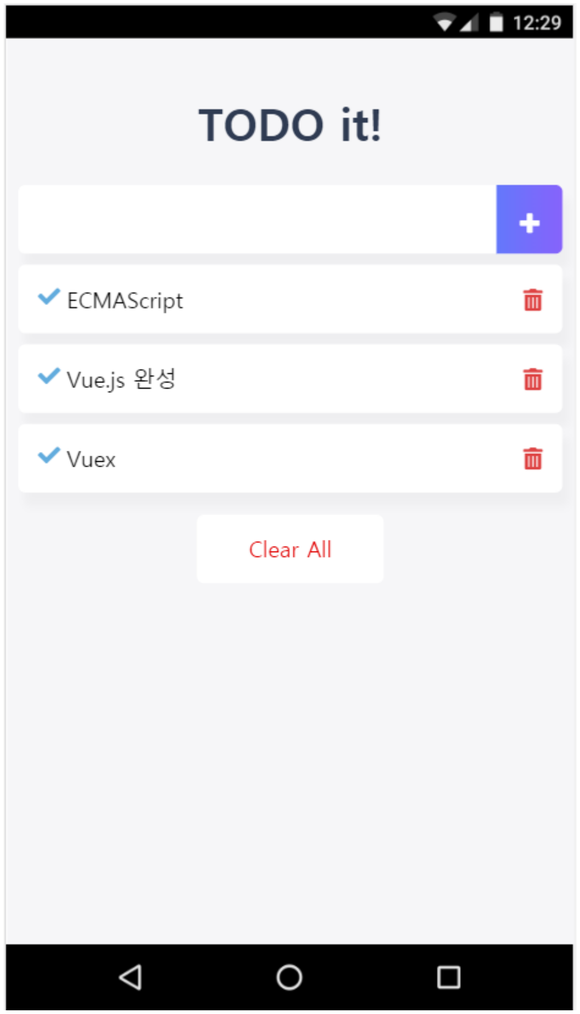
```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <p>…
    </p>
    <h3>Installed CLI Plugins</h3>
```

src/main.js

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

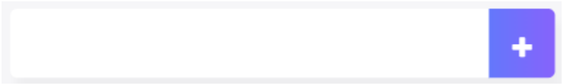
Todo App

1. App.vue : 루트 컴포넌트



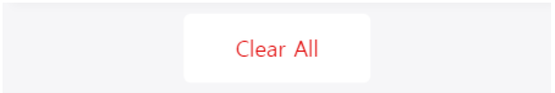
2. TodoHeader.vue : 어플리케이션의 제목 표시하는 컴포넌트

3. TodoInput.vue : 할 일 입력 및 추가하는 컴포넌트



4. TodoList.vue : 할 일 목록 표시 , 특정 할 일 삭제하는 컴포넌트

5. TodoFooter.vue : 할 일 모두 삭제하는 컴포넌트



Todo App : 프로젝트 초기화

1. App.vue 수정

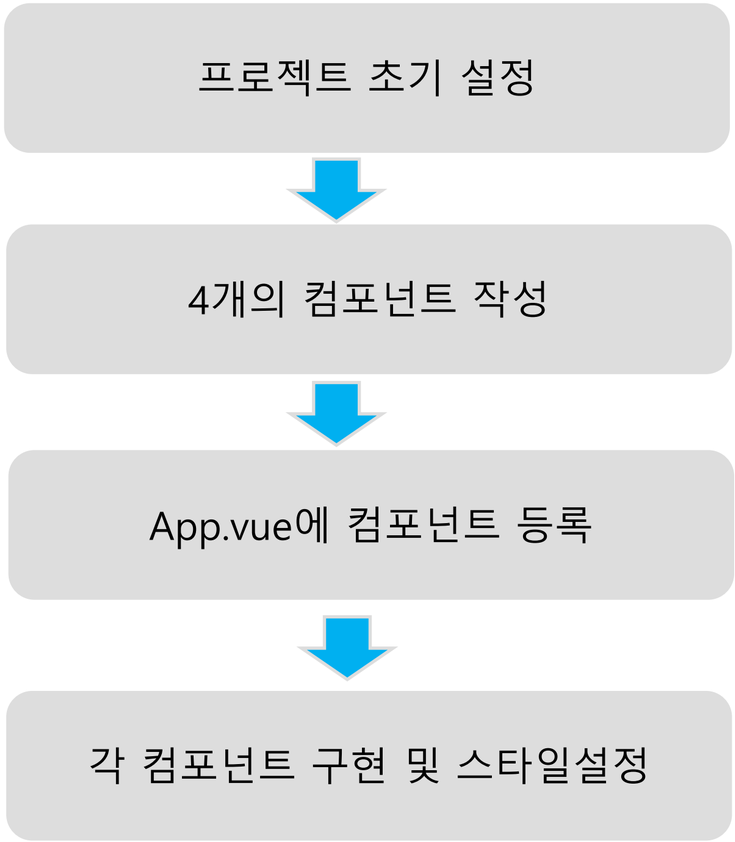
src/App.vue

```
<template>
    <div id="app"></div>
</template>

<script>

</script>

<style>

</style>
```

src/components/HelloWorld.vue 파일은 제거 하세요

Todo App : App 작성 순서

App 작성 순서



components 디렉토리에 다음 파일들을 생성하세요:

: src/components/TodoHeader.vue

: src/components/TodoInput.vue

: src/components/TodoList.vue

: src/components/TodoFooter.vue

TodoApp : 프로젝트 초기 설정

1. 프로젝트 초기 설정

1) 반응형 웹 태그 설정 : viewport meta tag 추가

2) awsome 아이콘 CSS 설정 : font awesome cdn 검색, https://fontawesome.com/start

3) favicon 설정 : favicon은 Vue에서 제공하는 기본 로고를 사용한다.

4) google Ubuntu 폰트를 사용한다. https://fonts.google.com/specimen/Ubuntu?selection.family=Ubuntu

public/index.html

```
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
  <link rel="icon" href="<%= BASE_URL %>favicon.ico">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.1/css/all.css"
integrity="sha384-fnmOCqbTlWIlj8LyTjo7mOUStjsKC4pOpQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr"  crossorigin="anonymous">
  <link href="https://fonts.googleapis.com/css?family=Ubuntu" rel="stylesheet">
  <title>Vue.js Todo</title>
</head>
```

TodoApp : 컴포넌트 생성

2. 컴포넌트 생성 및 등록

1) src/components 폴더에 TodoHeader.vue, TodoInput.vue, TodoList.vue, TodoFooter.vue 를 생성합니다.

2) src/App.vue에 생성한 컴포넌트들을 등록한다.

components/TodoHeader.vue

```
<template>
    <div>Header</div>
</template>

<script>
  export default {

  }
</script>

<style>
</style>
```

components/TodoInput.vue

```
<template>
    <div>Input</div>
</template>

<script>
  export default {

  }
</script>

<style>
</style>
```

components/TodoList.vue

```
<template>
    <div>List</div>
</template>

<script>
  export default {

  }
</script>

<style>
</style>
```

components/TodoFooter.vue

```
<template>
    <div>Footer</div>
</template>

<script>
  export default {

  }
</script>

<style>
</style>
```

TodoApp : 컴포넌트 등록

2. 컴포넌트 생성 및 등록 : src/App.vue에 생성한 컴포넌트들을 등록한다.

src/App.vue

```
<template>
  <div id="app">
    <TodoHeader></TodoHeader>
    <TodoInput></TodoInput>
    <TodoList></TodoList>
    <TodoFooter></TodoFooter>
  </div>
</template>
<script>
  import TodoHeader from './components/TodoHeader.vue'
  import TodoInput from './components/TodoInput.vue'
  import TodoList from './components/TodoList.vue'
  import TodoFooter from './components/TodoFooter.vue'
  export default {
    components: {
      'TodoHeader': TodoHeader,
      'TodoInput': TodoInput,
      'TodoList': TodoList,
      'TodoFooter': TodoFooter
   }
 }
</script>
```

TodoApp : 컴포넌트 구현

3. 컴포넌트 내용 구현하기 : TodoHeader (todo 제목)

1) TodoHeader.vue에 제목 추가하기

2) TodoHeader와 App의 style 설정하기

: <style> 태그에 사용된 scoped는 뷰에서 지원하는 속성이며, 스타일 정의를 해당 컴포넌트에만 적용한다.

src/components/TodoHeader.vue

```
<template>
    <header>
        <h1>TODO it!</h1>
    </header>
</template>

<style scoped>
    h1 {
        color:#2F3852;
        font-weight: 900;
        margin: 2.5rem 0 1.5rem;
    }
</style>
```

src/App.vue

```
<style>
  body {
    text-align: center;
    background-color: #f6f6f6;
  }
  input {
    border-style: groove;
    width:200px;
  }
  button {
    border-style: groove;
  }
  .shadow {
    box-shadow: 5px 10px 10px
rgba(0, 0, 0, 0.03);
  }
</style>
```

TodoApp : 컴포넌트 구현

4. 컴포넌트 내용 구현하기 : TodoInput (todo 입력 및 추가)

1) TodoInput.vue에 Inputbox 추가하기 : v-model (two-way binding) directive 사용

src/TodoInput.vue

```
<template>
    <div>
        <input type="text" v-model="newTodoItem">
        <button>추가</button>
    </div>
</template>

<script>
export default {
  data: function() {
        return {
            newTodoItem: ""
        }
   }
}
</script>
```

TodoApp : 컴포넌트 구현

4. 컴포넌트 내용 구현하기 : TodoInput (todo 입력 및 추가)

2) TodoInput.vue에 button 추가하기 : v-on directive 사용, v-on:click에 버튼 이벤트 핸들러 addTodo를 지정한다.

src/TodoInput.vue

```
<template>
    <div>
        <input type="text" v-model="newTodoItem">
        <button v-on:click="addTodo">추가</button>
    </div>
</template>

<script>
export default {
  methods:{
      addTodo: function() {
          console.log(this.newTodoItem);
      }
  }
}
</script>
```

```
//package.json

"rules": {
      "no-console": "off"
},
```

TodoApp : 컴포넌트 구현

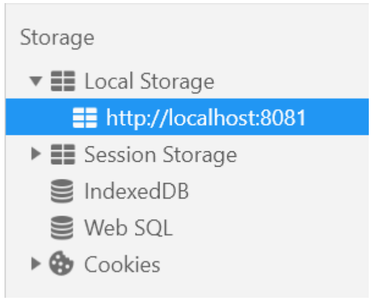
4. 컴포넌트 내용 구현하기 : TodoInput (todo 입력 및 추가)

3) TodoInput.vue에 addTodo() 구현 : 입력 받은 텍스트를 localStorage에 저장하기, localStorage의 setItem() API를 이용하여 저장하고, newTodoItem 변수 초기화 하기

src/TodoInput.vue

```
<template>
    <div>
        <input type="text" v-model="newTodoItem">
        <button v-on:click="addTodo">추가</button>
    </div>
</template>
<script>
export default {
  methods:{
      addTodo: function() {
          localStorage.setItem(this.newTodoItem, this.newTodoItem);
          this.newTodoItem = '';
      }
  }
} </script>
```

크롬 개발자 도구의 [Application -> Storage -> Local Storage -> http://localhost:8081] 에서 저장된 값을 확인합니다.



TodoApp : 컴포넌트 구현

4. 컴포넌트 내용 구현하기 : TodoInput (todo 입력 및 추가)

4) TodoInput.vue에 clearInput() 구현 : addTodo() 안에 예외처리 코드 추가하기, clearInput() 함수 추가

src/TodoInput.vue

```
<script>
export default {
  methods:{
      addTodo: function() {
          if (this.newTodoItem !== '') {
                  localStorage.setItem(this.newTodoItem, this.newTodoItem);
                  this.clearInput();
          }
      } ,
      clearInput: function() {
           this.newTodoItem = '';
      }
  }
}
</script>
```

TodoApp : 컴포넌트 구현

4. 컴포넌트 내용 구현하기 : TodoInput (todo 입력 및 추가)

5) awesome 아이콘 이용해 직관적인 버튼 만들기 : 스타일 설정

src/TodoInput.vue

```
<style scoped>
input:focus {
  outline: none;
}
.inputBox {
  background: white;
  height: 50px;
  line-height: 50px;
  border-radius: 5px;
}
.inputBox input {
  border-style: none;
  font-size: 0.9rem;
}
```

src/TodoInput.vue

```
.addContainer {
  float: right;
  background: linear-gradient(to right, #6478FB, #8763FB);
  display: block;
  width: 3rem;
  border-radius: 0 5px 5px 0;
}
.addBtn {
  color: white;
  vertical-align: middle;
}
</style>
```

TodoApp : 컴포넌트 구현

4. 컴포넌트 내용 구현하기 : TodoInput (todo 입력 및 추가)

5) awesome 아이콘 이용해 직관적인 버튼 만들기 : <button> 태그를 삭제하고 ,<i> 태그를 추가합니다. https://fontawesome.com/icons/plus?style=solid (plus icon)

inputbox에서 enter 를 입력 했을 때도 todo가 추가될 수 있도록 v-on:keyup.enter 이벤트를 처리한다.

src/TodoInput.vue

```
<template>
    <div class="inputBox shadow">
        <input type="text" v-model="newTodoItem"  v-on:keyup.enter="addTodo">
        <span class="addContainer"  v-on:click="addTodo">
            <i class="fas fa-plus addBtn"></i>
        </span>
    </div>
</template>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 목록)

1) 할 일 목록 기능 : 로컬 스토리지 데이터를 뷰에 출력하기

created() 라이프 사이클 메서드에 for 반복문과 push()로 로컬 스토리지의 모든 데이터를 todoItems에 저장하는 로직 구현

src/TodoList.vue

```
<script>
export default {
    data: function() {
        return {
            todoItems: []
        }
    },
    /* life cycle method */
    created: function() {
        if(localStorage.length > 0){
            for(var i=0; i < localStorage.length; i++) {
                this.todoItems.push(localStorage.key(i));
            }
        }
    }
}
</script>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 목록)

2) 할 일 목록 기능 : 로컬 스토리지 데이터를 뷰에 출력하기

v-for 디렉티브를 사용하여 목록을 렌더링 한다.

src/TodoList.vue

```
<template>
    <div>
        <ul>
            <li v-for="todoItem in todoItems"  v-bind:key="todoItem">
                {{todoItem}}
            </li>
        </ul>
    </div>
</template>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 목록)

3) 스타일 설정

src/TodoList.vue

```
<style scoped>
ul {
  list-style-type: none;
  padding-left: 0px;
  margin-top: 0;
  text-align: left;
}
li {
  display: flex;
  min-height: 50px;
  height: 50px;
  line-height: 50px;
  margin: 0.5rem 0;
  padding: 0 0.9rem;
  background: white;
  border-radius: 5px;
}
```

src/TodoList.vue

```
.removeBtn {
  margin-left: auto;
  color: #de4343;
}
.checkBtn {
  line-height: 45px;
  color: #62acde;
  margin-right: 5px;
}
.checkBtnCompleted {
  color: #b3adad;
}
.textCompleted {
  text-decoration: line-through;
  color: #b3adad;
}
</style>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 삭제)

4) 할 일 삭제 기능 : 할일 목록 & 삭제 버튼 마크업 작업하기

https://fontawesome.com/icons/trash-alt?style=solid (삭제 icon)

src/TodoList.vue

```
<template>
    <div>
        <ul>
            <li v-for="(todoItem, index) in todoItems" v-bind:key="index" class="shadow">
                {{todoItem}}
                <span class="removeBtn" v-on:click="removeTodo(todoItem, index)">
                    <i class="fas fa-trash-alt"></i>
                </span>
            </li>
        </ul>
    </div>
</template>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 삭제)

5) 할 일 삭제 기능 : removeTodo() 메서드 구현

localStorage의 데이터를 삭제하는 removeItem() 와 배열의 특정 인덱스를 삭제하는 splice() 함수로 todo 를 삭제합니다.

src/TodoList.vue

```
<script>
export default {
    data: function() {
        ...
    },
    methods: {
        removeTodo: function(todoItem, index) {
            localStorage.removeItem(todoItem);
            this.todoItems.splice(index, 1);
        }
    },
...
</script>
```

splice() vs slice() 함수

```
var array=[1,2,3,4,5];
console.log(array.splice(2));
```

This will return `[3,4,5]`. The **original array is affected** resulting in `array` being `[1,2]`.

```
var array=[1,2,3,4,5]
console.log(array.slice(2));
```

This will return `3,4,5`. The **original array is NOT affected** with resulting in `array` being `[1,2,3,4,5]`.

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 완료)

6) 할 일 완료 기능 : 완료 버튼 마크업 작업하기 & toggleComplete() 메서드 선언

https://fontawesome.com/icons/check?style=solid (check icon)

src/TodoList.vue

```
<template>
    <div>
        <ul>
            <li v-for="(todoItem, index) in todoItems" v-bind:key="index" class="shadow">
                <i class="fas fa-check checkBtn" v-on:click= "toggleComplete" ></i>
                {{todoItem}}
                <span class="removeBtn" v-on:click="removeTodo(todoItem, index)">
                    <i class="fas fa-trash-alt"></i>
                </span>
            </li>
        </ul>
    </div>
</template>
<script>
    methods: {  toggleComplete: function() {    }   },
</script>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoInput (todo 완료 기능 추가로 수정)

7) 할 일 완료 기능 : addTodo() 메서드 수정

JSON.stringify() 로 object 를 json string으로 변환한다. {"completed":false, "item":"Vue.js 완성"}

src/TodoInput.vue

```
<script>
export default {
  methods:{
     addTodo: function() {
        if (this.newTodoItem !== '') {
              var obj = {completed: false, item: this.newTodoItem};
              localStorage.setItem(this.newTodoItem, JSON.stringify(obj));
              this.clearInput();
        }
     } ,
     clearInput: function() {
         this.newTodoItem = '';
     }
  }
}
</script>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 완료)

8) 할 일 완료 기능 : created() 메서드 수정

JSON.parse() 로 json string 을 object 로 변환한다. 리스트에 출력하는 부분도 수정합니다.

src/TodoList.vue

```
<template>
        <ul>
            <li v-for="(todoItem, index) in todoItems" v-bind:key="index" class="shadow">
                  <span class="textCompleted">{{todoItem.item}}</span>
            </li>
        </ul>
</template>
<script>
   created: function() {
        if(localStorage.length > 0){
            for(var i=0; i < localStorage.length; i++) {
                if(localStorage.key(i) !== 'loglevel:webpack-dev-server'){
                    var itemJson = localStorage.getItem(localStorage.key(i));
                    this.todoItems.push(JSON.parse(itemJson));
                }
            }
        }
    }
</script>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 완료)

9) 할 일 완료 기능 : v-bind directive 사용

todoItem.completed 값(true/false)에 따라서 textCompleted css class 를 적용하기.

todoItem.completed 값(true/false)에 따라서 checkBtnCompleted css class 를 적용하기.

src/TodoList.vue

```
<template>
        <ul>
            <li v-for="(todoItem, index) in todoItems" v-bind:key="index" class="shadow">
 <i class="fas fa-check checkBtn" v-bind:class="{checkBtnCompleted: todoItem.completed}" v-on:click="toggleComplete"></i>
              <span v-bind:class="{textCompleted: todoItem.completed }">{{todoItem.item}}</span>
            </li>
        </ul>
</template>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 완료)

10) 할 일 완료 기능 : toggleComplete() 메서드 구현

todoItem.completed 값(true/false)의 토글링에 따라서 localStorage에 저장된 completed 값도 변경해준다.

localStorage에는 값을 수정하는 함수가 없기 때문에 removeItem()으로 먼저 삭제를 하고, setItem()으로 추가를 해줘야 함.

src/TodoList.vue

```
<template>
        <ul>
            <li v-for="(todoItem, index) in todoItems" v-bind:key="index" class="shadow">
                    <I class="fas fa-check checkBtn" v-bind:class="{checkBtnCompleted: todoItem.completed}"
                     v-on:click="toggleComplete(todoItem,index)"></i>
            </li>
        </ul>
</template>
<script>
  methods: {
       toggleComplete: function(todoItem,index) {
            todoItem.completed = !todoItem.completed;
            localStorage.removeItem(todoItem.item);
            localStorage.setItem(todoItem.item, JSON.stringify(todoItem));
        }
    },
</script>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoFooter (todo 모두 삭제)

11) 할 일 모두 삭제 기능 : clearTodo메서드 구현

스타일 설정 하고, 삭제 버튼 추가

src/TodoFooter.vue

```
<style scoped>
 .clearAllContainer {
  width: 8.5rem;
  height: 50px;
  line-height: 50px;
  background-color: white;
  border-radius: 5px;
  margin: 0 auto;
}
.clearAllBtn {
  color: #e20303;
  display: block;
}
</style>
```

src/TodoFooter.vue

```
<template>
    <div class="clearAllContainer">
        <span class="clearAllBtn" v-on:click="clearTodo">Clear All</span>
    </div>
</template>
<script>
export default {
    methods: {
        clearTodo: function() {
            localStorage.clear();
        }
    }
}
</script>
```

할 일 관리(Todo App)
: 구조 개선하기(리팩토링)

TodoApp : 문제점 & 해결방법

1. 현재 어플리케이션 구조의 문제점

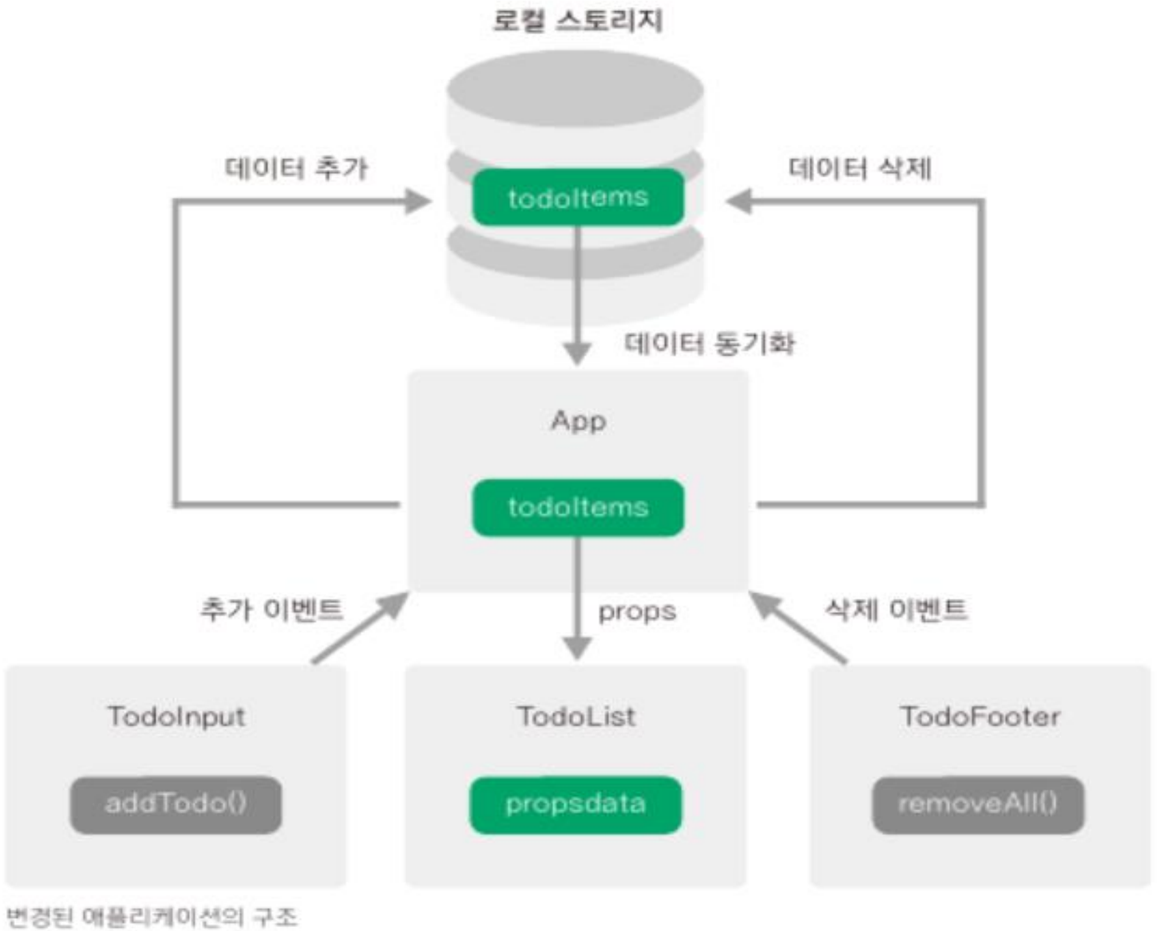
- 할 일을 입력 했을 때 할일 목록에 바로 반영되지 않는 점
- 할 일을 모두 삭제했을 때 목록에 바로 반영되지 않는 점

2. 해결 방법

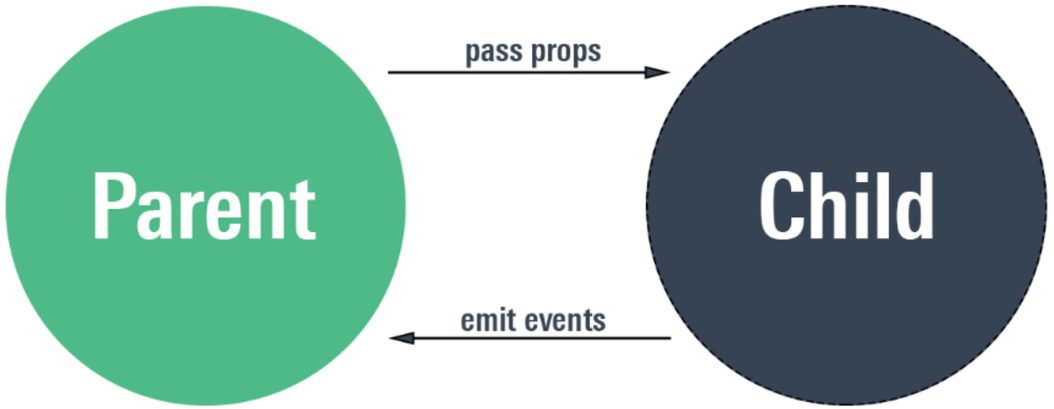
- 각각의 컴포넌트에서 각자 뷰 데이터 속성(newTodoItem, todoItems)을 가지고 있지만, localStorage의 데이터는 공유하고 있는 상황이다.
- 문제점을 해결하기 위해 데이터 속성을 최상위(루트) 컴포넌트인 App 컴포넌트에 todoItems을 정의하고, 하위 컴포넌트 TodoList에 props 로 전달합니다.
- 하위 컴포넌트에서 발생한 이벤트를 $.emit을 이용하여 상위 컴포넌트로 전달합니다.

TodoApp : 문제점 & 해결방법

2. 해결 방법



변경된 애플리케이션의 구조



TodoApp : 컴포넌트 구현 리팩토링

1. [리팩토링] 할 일 목록 표시 기능 : App, TodoList 수정

TodoList에 있던 todoItems 데이터 변수와 created life cycle hook method를 App으로 옮긴다.

App에서 TodoList에게 props로 전달합니다.

src/App.vue

```
<template>
  <div id="app">
    <TodoList v-bind:propsdata="todoItems"></TodoList>
  </div>
</template>
<script>
export default {
  data: function() {
        return {  todoItems: [] }
  },
  created: function() {
      ……
  }, ...
}
</script>
```

src/components/TodoList.vue

```
<template>
    <div>
        <ul>
            <li v-for="(todoItem, index) in propsdata" ...>
                ...
            </li>
        </ul>
    </div>
</template>
<script>
export default {
    props: ['propsdata'],
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

2. [리팩토링] 할 일 추가 기능 : App, TodoInput 수정

App에 addOneItem 메서드를 추가하고, TodoInput의 addTodo 메서드에 있던 코드를 옮긴다.

src/App.vue

```
<script>
export default {
    methods: {
      addOneItem: function() {
        var obj = {completed: false, item: this.newTodoItem};
        localStorage.setItem(this.newTodoItem,JSON.stringify(obj));
      }
    },
}
</script>
```

src/components/TodoInput.vue

```
<script>
export default {
    methods:{
        addTodo:function() {
            if (this.newTodoItem !== '') {

                this.clearInput();
            }
        },
    }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

2-1. [리팩토링] 할 일 추가 기능 : App, TodoInput 수정

TodoInput 에서 발생한 Event를 App에 전달할 때 this.$emit("이벤트이름", 인자) 를 사용한다.

<TodoInput v-on:하위컴포넌트에서 발생시킨 이벤트이름="현재 컴포넌트의 메서드명"></TodoInput>

src/App.vue

```
<template>
  <div id="app">
    <TodoInput v-on:addItemEvent="addOneItem"></TodoInput>
  </div>
</template>
<script>
export default {
   methods: {
     addOneItem: function(todoItem) {
       var obj = {completed: false, item: todoItem};
       localStorage.setItem(todoItem,JSON.stringify(obj));
       //localStorage와 화면을 동기화 시키기 위해서
       this.todoItems.push(obj);
     }
   },
}
</script>
```

src/components/TodoInput.vue

```
<script>
export default {
    methods:{
        addTodo:function() {
            if (this.newTodoItem !== '') {
                this.$emit('addItemEvent', this.newTodoItem);
                this.clearInput();
            }
        },
    }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

3. [리팩토링] 할 일 삭제 기능 : App, TodoList 수정

App에 removeOneItem 메서드를 추가하고, TodoList의 removeTodo 메서드에 있던 코드를 옮긴다.

src/App.vue

```
<script>
export default {
    methods: {
      removeOneItem: function() {
          localStorage.removeItem(todoItem);
          this.todoItems.splice(index, 1);
      }
    },
}
</script>
```

src/components/TodoList.vue

```
<script>
export default {
    methods:{
        removeTodo:function() {

        },
    }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

3-1. [리팩토링] 할 일 삭제 기능 : App, TodoList 수정

TodoList 에서 발생한 클릭 Event를 App에 전달할 때 this.$emit("이벤트이름", 인자) 를 사용한다.

<TodoList v-on:하위컴포넌트에서 발생시킨 이벤트이름="현재 컴포넌트의 메서드명"></TodoList>

src/App.vue

```
<template>
  <div id="app">
    <TodoList v-on:removeItemEvent="removeOneItem">
    </TodoList>
  </div>
</template>

<script>
export default {
   methods: {
     removeOneItem: function(todoItem, index) {
          localStorage.removeItem(todoItem.item);
          this.todoItems.splice(index, 1);
     }
   },
}
</script>
```

src/components/TodoList.vue

```
<script>
export default {
    methods:{
        removeTodo: function(todoItem, index) {
             this.$emit('removeItemEvent',todoItem,index);
         },
    }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

4. [리팩토링] 할 일 완료 기능 : App, TodoList 수정

App에 toggleOneItem 메서드를 추가하고, TodoList의 toggleComplete 메서드에 있던 코드를 옮긴다.

src/App.vue

```
<script>
export default {
    methods: {
      toggleOneItem: function() {
       todoItem.completed = !todoItem.completed;
       localStorage.removeItem(todoItem.item);
       localStorage.setItem(todoItem.item,JSON.stringify(todoItem));
      }
    },
}
</script>
```

src/components/TodoList.vue

```
<script>
export default {
    methods:{
        toggleComplete:function(todoItem, index) {

        },
    }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

4-1. [리팩토링] 할 일 완료 기능 : App, TodoList 수정

TodoList 에서 발생한 클릭 Event를 App에 전달할 때 this.$emit("이벤트이름", 인자) 를 사용한다.

<TodoList v-on:하위컴포넌트에서 발생시킨 이벤트이름="현재 컴포넌트의 메서드명"></TodoList>

src/App.vue

```
<template>
  <div id="app">
    <TodoList v-on:toggleItemEvent="toggleOneItem">
    </TodoList>
  </div>
</template>
<script>
export default {
   methods: {
     toggleOneItem: function(todoItem, index) {
         todoItem.completed = !todoItem.completed;
         localStorage.removeItem(todoItem.item);
         localStorage.setItem(todoItem.item, JSON.stringify(todoItem));
     }
   },
}
</script>
```

src/components/TodoList.vue

```
<script>
export default {
    methods:{
        toggleComplete: function(todoItem, index) {
            this.$emit('toggleItemEvent', todoItem, index);
        },
    }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

4-2. [리팩토링] 할 일 완료 기능 : App, TodoList 수정

TodoList에서 인자로 전달 받은 todoItem의 completed 값을 변경하는 것보다는, todoItems 배열 중의 1개의 todoItem의 completed 값을 변경하는 것이 더 좋은 방법입니다.

src/App.vue

```
<template>
  <div id="app">
    <TodoList v-on:toggleItemEvent="toggleOneItem">
    </TodoList>
  </div>
</template>
<script>
export default {
   methods: {
     toggleOneItem: function(todoItem, index) {
         this.todoItems[index].completed = !this.todoItems[index].completed;
         localStorage.removeItem(todoItem.item);
         localStorage.setItem(todoItem.item, JSON.stringify(todoItem));
     }
   },
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

5. [리팩토링] 할 일 모두 삭제 기능 : App, TodoFooter 수정

TodoFooter 에서 발생한 클릭 Event를 App에 전달할 때 this.$emit("이벤트이름", 인자) 를 사용한다.

<TodoFooter v-on:하위컴포넌트에서 발생시킨 이벤트이름="현재 컴포넌트의 메서드명"></TodoFooter>

src/App.vue

```
<template>
  <div id="app">
    <TodoFooter v-on:removeAllItemEvent="removeAllItems">
    </TodoFooter>
  </div>
</template>
<script>
export default {
   methods: {
     removeAllItems: function() {
          localStorage.clear();
          this.todoItems = [];
     }
   },
}
</script>
```

src/components/TodoFooter.vue

```
<script>
export default {
    methods:{
        clearTodo: function() {
            this.$emit('removeAllItemEvent');
        }
    }
}
</script>
```

할 일 관리(Todo App)
: 사용자 경험 개선

TodoApp : 사용자 경험 개선

1. 개선해야 하는 2가지 기능

- 할 일을 입력할 때 값을 입력하지 않고  버튼을 클릭하는 경우
- 할 일을 추가하거나, 삭제 할 때 좀 더 자연스럽게 화면이 보이게 하는 경우

2. 해결 방법

- Modal 과 애니메이션을 이용하여 개선합니다.
- Modal Component Example https://vuejs.org/v2/examples/modal.html
- Enter/Leave & List Transitions https://vuejs.org/v2/guide/transitions.html#Overview
  https://vuejs.org/v2/guide/transitions.html#List-Transitions

TodoApp : 컴포넌트 구현

1. 뷰 Modal : TodoInput 수정, Modal 추가

components/common/Modal.vue 컴포넌트 생성 , Modal 컴포넌트를 TodoInput 의 하위 컴포넌트로 등록한다.

src/components/TodoInput.vue

```
<template>
    <div class="inputBox shadow">
        <Modal v-if="showModal" @close="showModal = false">
          <h3 slot="header">custom header</h3>
        </Modal>
    </div>
</template>
<script>
import Modal from './common/Modal.vue'
export default {
   data:function() {
      showModal: false
   },
    components: {
      'Modal': Modal
    }
}
</script>
```

src/components/common/Modal.vue

```
<template>
  <transition name="modal">
    <div class="modal-mask">
        .....
    </div>
  </transition>
</template>

<style>
.modal-mask {
....
</style>
```

TodoApp : 컴포넌트 구현

1-1. 뷰 Modal : TodoInput, Modal 수정

Modal에 있는 slot의 header와 body 부분만 재정의 한다. Modal.vue안에 선언된 footer slot 부분은 제거한다. https://fontawesome.com/icons/times?style=solid

src/components/TodoInput.vue

```
<template>
    <div class="inputBox shadow">
        <Modal v-if="showModal" @close="showModal = false">
          <h3 slot="header">
            경고!
            <i class="closeModalBtn fas fa-times"
                @click="showModal=false"></i>
          </h3>
          <div slot="body">
            아무것도 입력하지 않으셨습니다.
          </div>
        </Modal>
</div>
</template>
```

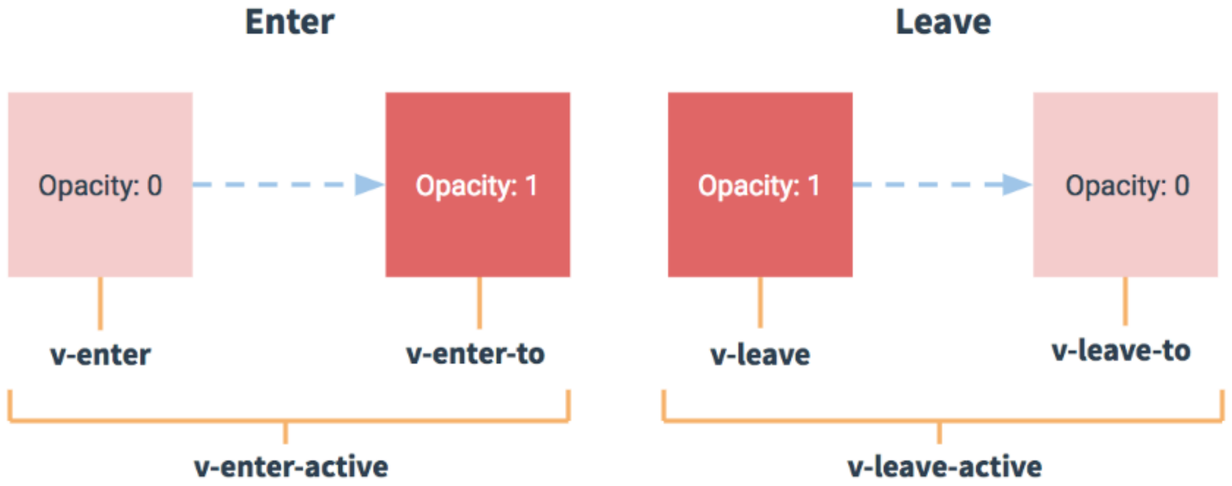
src/components/TodoInput.vue

```
<script>
export default {
methods:{
        addTodo:function() {
            if (this.newTodoItem !== '') {
            }else {
              this.showModal = !this.showModal;
            }
        },
    },
}
</script>
<style>
.closeModalBtn {
  color: #42b983;
}
</style>
```

TodoApp : 컴포넌트 구현

2. 뷰 애니메이션 : TodoList 수정

리스트 아이템의 트랜지션 효과 스타일 설정, <ul> 엘리먼트를 <transition-group> 엘리먼트로 변경한다.



src/components/TodoList.vue

```
<template>
    <div class="inputBox shadow">
        <transition-group name="list" tag="ul">
            .....
        </transition-group>
    </div>
</template>
<style>
.list-enter-active, .list-leave-active {
  transition: all 1s;
}
.list-enter, .list-leave-to  {
  opacity: 0;
  transform: translateY(30px);
}
</style>
```

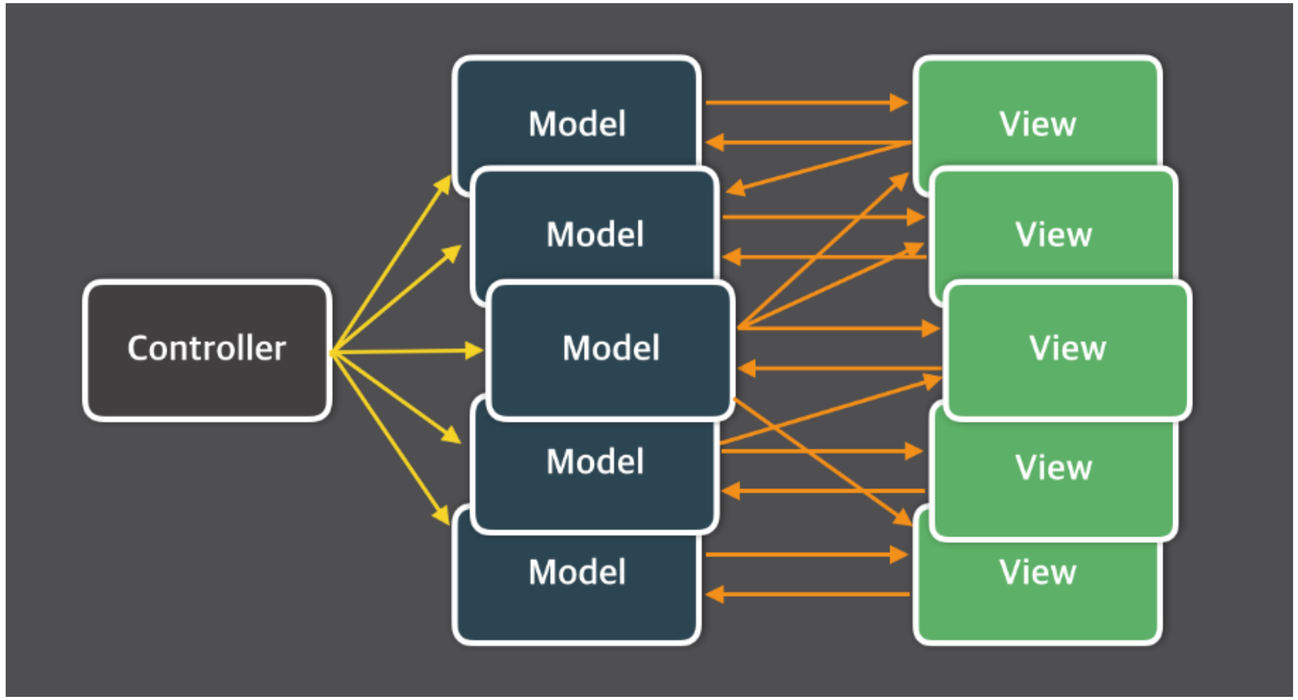
Vuex (상태 관리 라이브러리)

MVC 문제점

- MVC 패턴의 문제점

페이스북에서 이야기 하는 MVC의 가장 큰 단점은 양방향 데이터 흐름이었습니다.

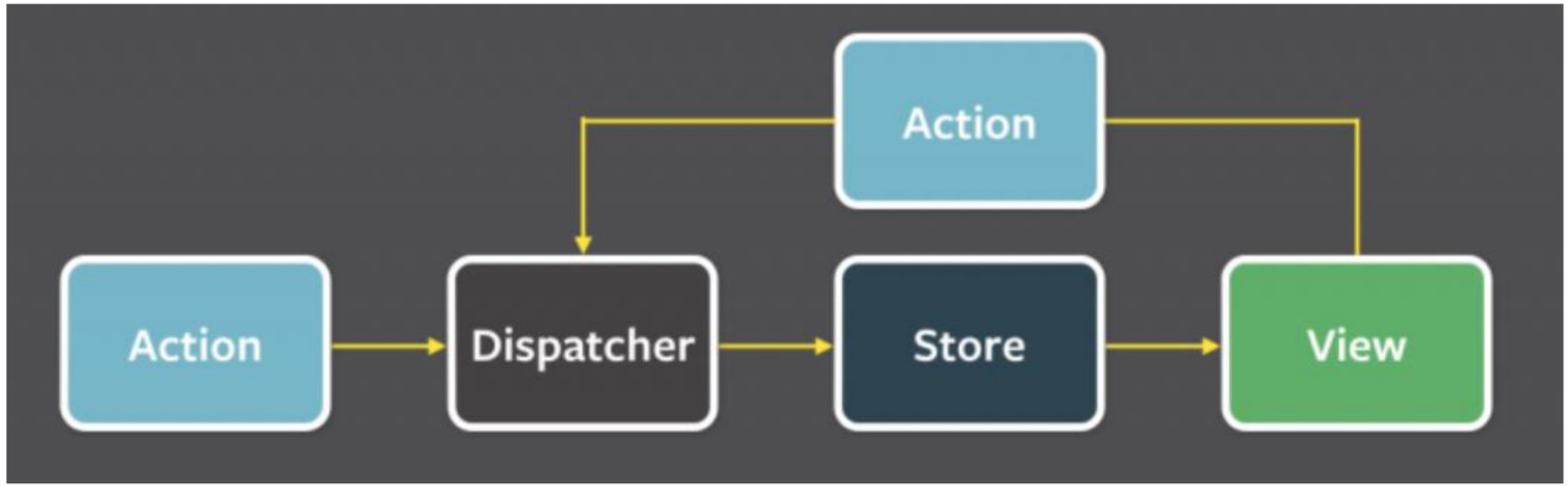
Model이 업데이트 되어 View가 따라서 업데이트 되고, 업데이트 된 View가 다시 다른 Model을 업데이트 한다면, 또 다른 View가 업데이트 될 수 있습니다. 복잡하지 않은 어플리케이션에서는 양방향 데이터 흐름이 문제가 크지 않을 수 있습니다. 하지만 어플리케이션이 복잡해 진다면 이런 양방향 데이터 흐름은 새로운 기능이 추가 될 때에 시스템의 복잡도를 증가 시키고, 예측 불가능한 코드를 만들게 됩니다.



Flux란?

- Flux로 해결

: 페이스북은 알람 버그의 원인을 양방향 데이터 흐름으로 보고, 단방향 데이터 흐름인 Flux를 도입했습니다.

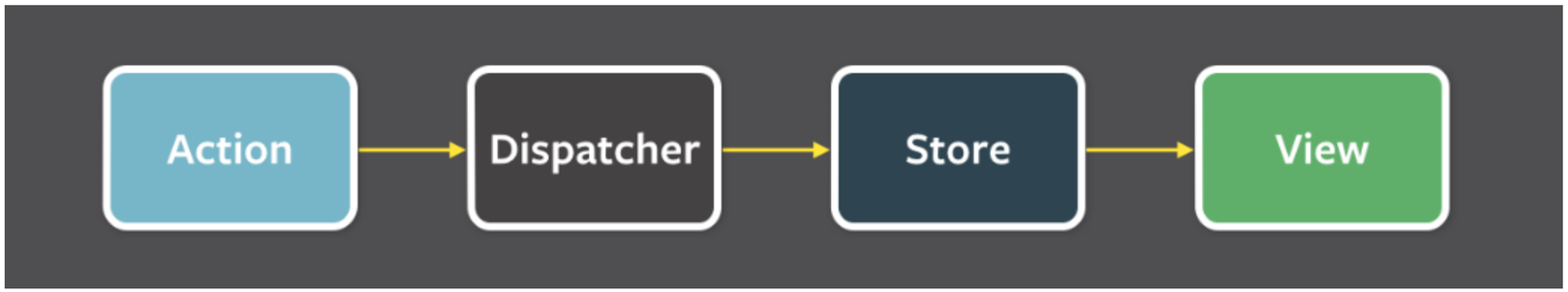


: Flux의 가장 큰 특징은 단방향 데이터 흐름입니다. 데이터 흐름은 항상 Dispatcher에서 Store로, Store에서 View로, View는 Action을 통해 다시 Dispatcher로 데이터가 흐르게 됩니다. 이런 단방향 데이터 흐름은 데이터 변화를 훨씬 예측하기 쉽게 만듭니다.

Flux란?

- Flux로 해결

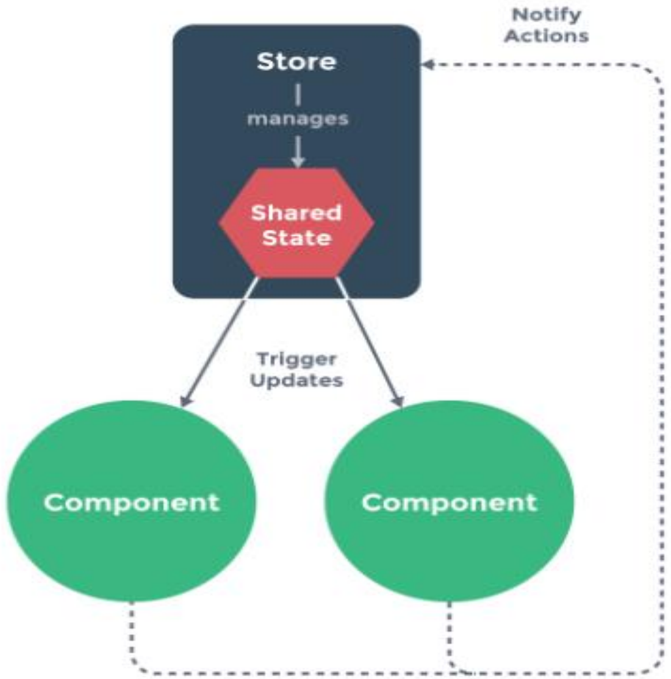
1. Dispatcher : Action이 발생되면 Dispatcher로 전달되는데, Dispatcher는 전달된 Action을 보고, 등록된 콜백 함수를 실행하여 Store에 데이터를 전달합니다.
2. Model(Store) : 어플리케이션의 모든 상태(state) 변경은 Store에 의해 결정이 됩니다.
3. View : 사용자에게 비춰지는 화면
4. Action : Dispatcher에서 콜백 함수가 실행 되면 Store가 업데이트 되게 되는데, 이 콜백 함수를 실행 할 때 데이터가 담겨 있는 객체가 인수로 전달 되어야 합니다. 이 전달 되는 객체가 Action 입니다.



Vuex

- Vuex란?

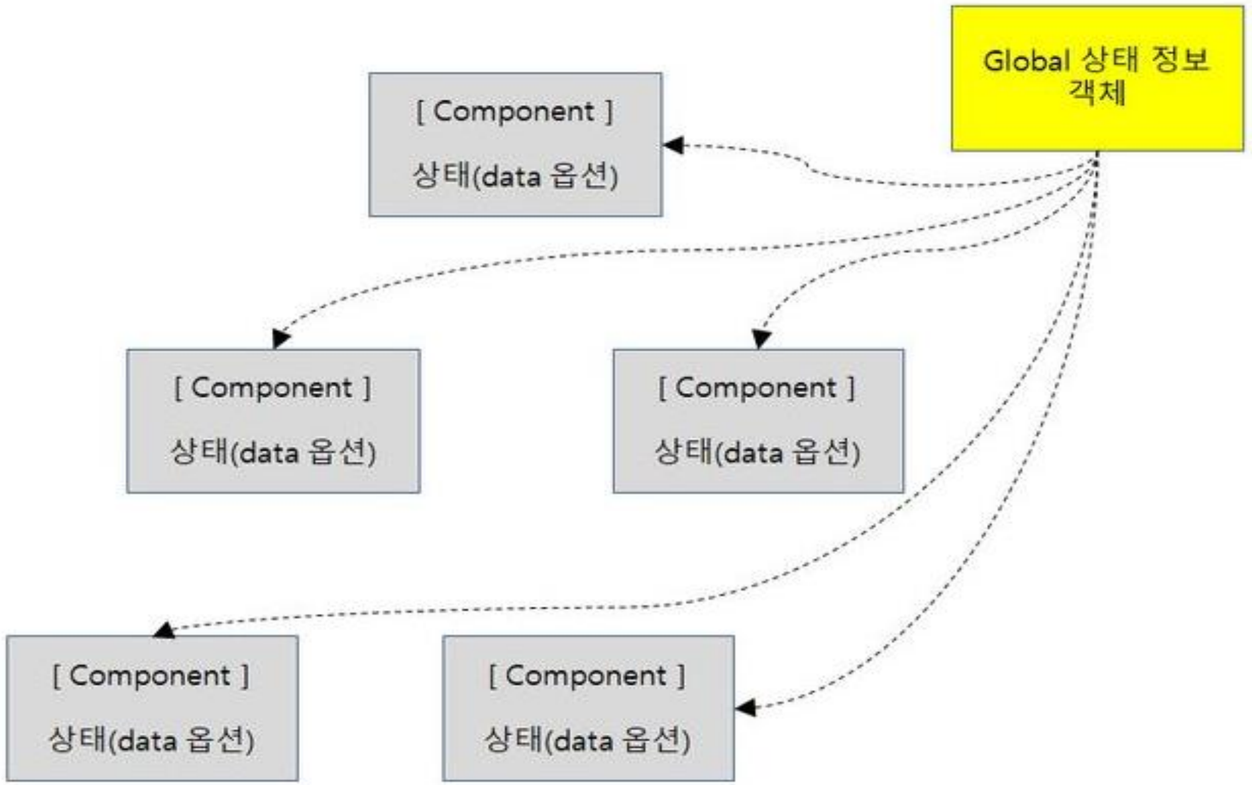
1. 무수히 많은 컴포넌트의 데이터를 더 효율적으로 관리 하는데 사용하는 상태 관리 라이브러리이다.
2. Vuex는 Vue.js 애플리케이션에 대한 상태 관리 패턴 + 라이브러리 입니다. 애플리케이션의 모든 컴포넌트에 대한 중앙 집중식 저장소 역할을 하며 예측 가능한 방식으로 상태를 변경할 수 있습니다.
3. Vuex의 기본 아이디어는 Flux, Redux, The Elm Architecture에서 영감을 받았습니다.



Vuex

Vuex가 왜 필요할까?

- 복잡한 애플리케이션에서 컴포넌트의 개수가 많아지면 컴포넌트 간에 데이터 전달이 어려워진다.
- 중앙 집중화된 상태 정보 관리가 필요하고, 상태 정보가 변경되는 상황과 시간을 추적하고 싶다.
- 컴포넌트에서 상태 정보를 안전하게 접근하고 싶다.



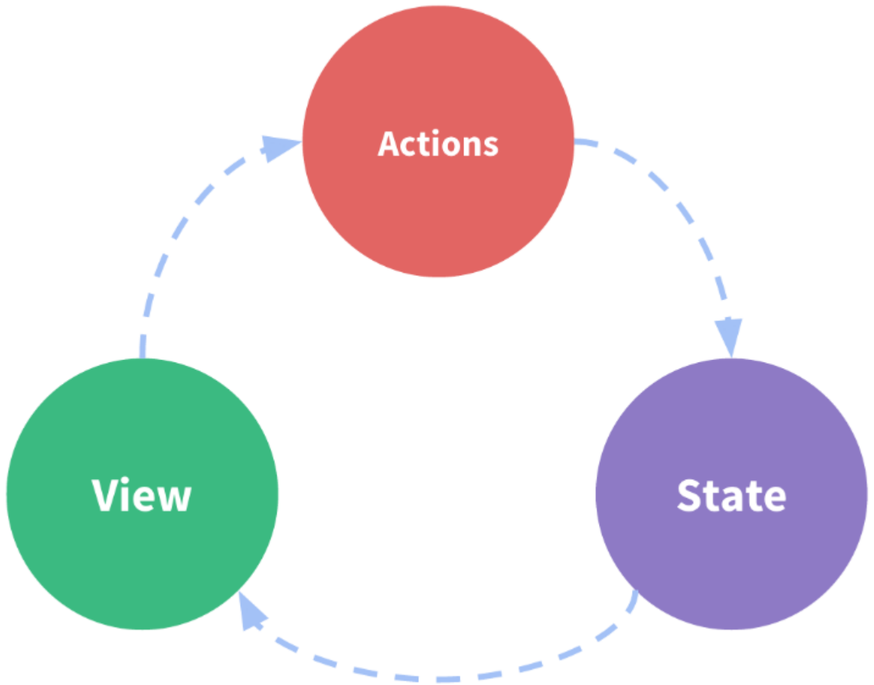
Vuex

Vuex로 해결할 수 있는 문제

- MVC 패턴에서 발생하는 구조적 오류
- 컴포넌트 간 데이터 전달 명시
- 여러 개의 컴포넌트에서 같은 데이터를 업데이트 할 때 동기화 문제

Vuex 개념

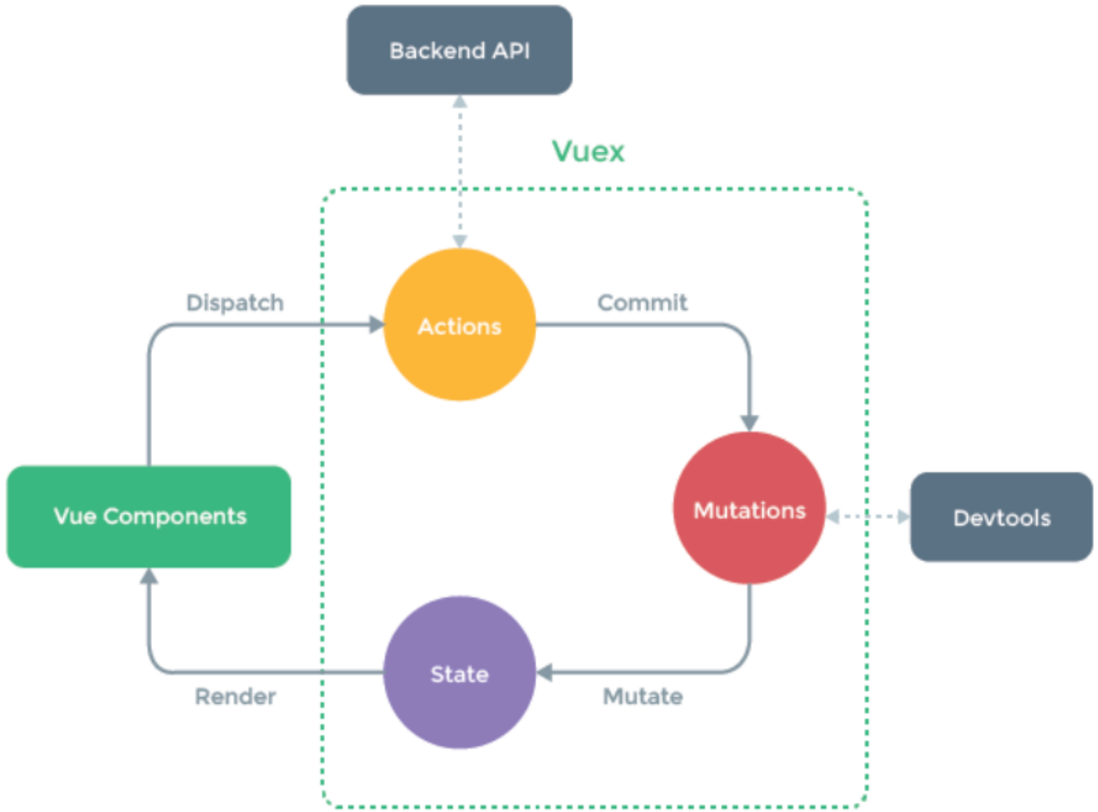
- State : 컴포넌트 간에 공유하는 데이터 data()
- View : 데이터를 표시하는 화면 template
- Action : 사용자의 입력에 따라 데이터를 변경하는 methods



Vuex

Vuex 구조

- 컴포넌트 -> Actions(비동기 로직) -> Mutations(동기 로직) -> State(상태)



Vuex 구조

- 컴포넌트가 Action을 일으키면(예:버튼 클릭)
- Action에서는 외부 API를 호출한 뒤 그 결과를 이용해 Mutations를 일으키고(만일 외부 API가 없으면 생략)
- Mutations에서는 Action의 결과를 받아 상태를 변경한다. 이 과정은 추적이 가능하므로 DevTools와 같은 도구를 이용하면 상태 변경의 내역을 모두 확인할 수 있다.
- Mutations에 의해 변경된 State는 다시 컴포넌트에 바인딩 되어 UI를 갱신한다.

Vuex : 설치

1. Vuex 설치하기

```
npm install --save vuex
```

2. Store 생성

- src/store 디렉토리와 store.js 파일을 생성한다.
- 루트 컴포넌트에 store 옵션을 제공함으로써, store는 루트의 모든 하위 컴포넌트에 주입되어진다.

src/store/store.js

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

export const store = new Vuex.Store({
    //
});
```

src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import { store } from './store/store';

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
  store,
}).$mount('#app')
```

Vuex : 구성요소

- Vuex 구성 요소

1. **state** : 여러 컴포넌트에서 공유되는 데이터 data
2. **getters** : 연산된 state 값을 접근하는 속성 computed
3. **mutations** : state 값을 변경하는 이벤트 로직과 메서드 methods (동기 메서드)
4. **actions** : 비동기 처리 로직을 선언하는 메서드 async method (비동기 메서드)

Vuex : state

- 여러 컴포넌트 간에 공유할 데이터 - 상태

데이터(상태) 선언

```
//Vue
data: {
  message: "Hello Vue.js!"
}
//Vuex
state: {
 message: "Hello Vue.js!"
}
```

데이터(상태) 사용

```
<!-- vue -->
<p>{{ message }}</p>

<!-- vuex -->
<p>{{ this.$store.state.message }}</p>
```

Vuex : getters

- getters란?

  : state 값을 접근하는 속성이며, computed() 처럼 미리 연산된 값을 접근하는 속성

메서드 선언

```
//store.js
state: {
 num: 10
},
getters: {
   getNumber(state) {
      return state.num;
   },
   doubleNumber(state) {
     return state.num * 2;
   }
}
```

메서드 사용

```
<p>{{ this.$store.getters.getNumber }}</p>
<p>{{ this.$store.getters.doubleNumber }}</p>
```

Vuex : [리팩토링] state 속성 적용

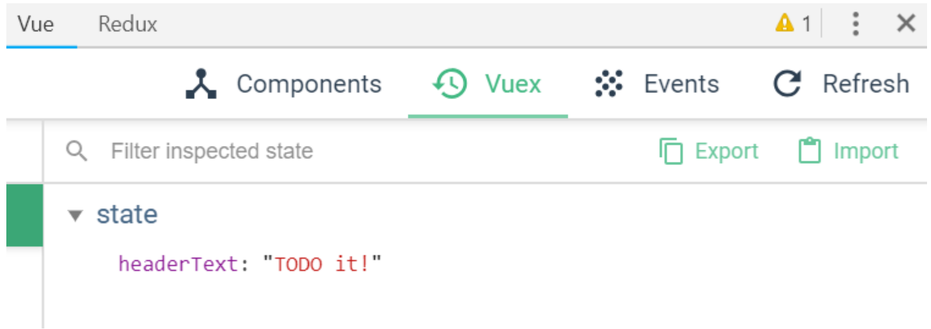
- TodoApp : 리팩토링 state 속성 적용

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        headerText: 'TODO it!'
    }
});
```

src/components/TodoHeader.vue

```
<template>
    <header>
        <h1>{{ this.$store.state.headerText }}</h1>
    </header>
</template>
```



Vuex : [리팩토링] state 속성 적용

1. TodoApp : 리팩토링 state 속성 적용

1) state에 todoItems 속성을 정의한다.

2) App.vue에 있는 created() 메서드를 store.js로 이동하고, 메서드 이름은 fetch() 로 변경한다.

src/store/store.js

```
const storage = {
    fetch() {
        const arr = [];
        if(localStorage.length > 0){
            for(let i=0; i < localStorage.length; i++) {
                if(localStorage.key(i) !== 'loglevel:webpack-dev-server'){
                    arr.push(JSON.parse(localStorage.getItem(localStorage.key(i))));
                }
            }
        }
        return arr;
    },
};
export const store = new Vuex.Store({
    state: {
        todoItems: storage.fetch()
    }
});
```

Vuex : [리팩토링] state 속성 적용

1. TodoApp : 리팩토링 state 속성 적용

3) App.vue의 <TodoList v-bind:propsdata="todoItems“>의 v-bind 속성을 제거한다.

4) TodoList.vue의 v-for 구문에서 propsdata 대신에 store에 바로 접근하기 위해 this.$store.state.todoItems 로 변경한다.

5) TodoList.vue의 props: ['propsdata'] 는 제거한다.

src/App.vue

```
<template>
 <TodoList  v-on:removeItemEvent="removeOneItem" v-on:toggleItemEvent="toggleOneItem"></TodoList>
</template>
```

src/component/TodoList.vue

```
<template>
    <div>
        <transition-group name="list" tag="ul">
            <li v-for="(todoItem, index) in this.$store.state.todoItems"
                    v-bind:key="todoItem.item" class="shadow">
              ...
            </li>
        </transition-group>
    </div>
</template>
```

Vuex : mutations와 commit()

Mutations란?

- State의 값을 변경할 수 있는 유일한 방법이며, 메서드이다.
- Mutations는 commit()으로 동작시킨다.

```
//store.js
state: { num: 10 },
mutations: {
    sumNumbers(state, anotherNum) {
      state.num += anotherNum;
    }
}
```

```
//App.vue
this.$store.commit('sumNumbers',20);
```

Vuex : mutations와 commit()

mutations의 commit() 형식

- State를 변경하기 위해 mutations를 동작시킬 때 인자(payload,객체)를 전달할 수 있음

```
//store.js
state: { storeNum: 10, message: '' },
mutations: {
   modifyState(state, payload) {
      state.message = payload.str;
      state.storeNum += payload.num;
   }
}
```

```
//App.vue
this.$store.commit('modifyState', {
    str: 'passed from payload' ,
    num: 20
});
```

Vuex : [리팩토링] mutations 적용

2. TodoApp 할 일 추가 : 리팩토링 mutations 적용

1) store.js에 mutations 속성을 정의한다.

2) store.js의 mutations 속성내에 App.vue에 있는 addOneItem() 메서드를 이동시킨다.

3) App.vue의 <TodoInput v-on:addItemEvent="addOneItem"></TodoInput> ➔ <TodoInput></TodoInput>로 변경한다.

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        todoItems: storage.fetch()
    },
    mutations: {
        addOneItem(state, todoItem) {
            const obj = { completed: false, item: todoItem };
            localStorage.setItem(todoItem,JSON.stringify(obj));
            state.todoItems.push(obj);
        },
    }
});
```

src/App.vue

```
<template>
   <TodoInput></TodoInput>
</template>
```

Vuex : [리팩토링] mutations 적용

2. TodoApp 할 일 추가 : 리팩토링 mutations 적용

3) TodoInput 에서 할 일 추가 이벤트가 발생했을 때 App.vue에게 Event를 보내는 대신에 store에 저장된 addOneItem() 메서드를 직접 호출한다.

this.$emit('addItemEvent', this.newTodoItem); ➔ this.$store.commit('addOneItem', this.newTodoItem);

src/component/TodoInput.vue

```
export default {
    methods:{
        addTodo() {
            if (this.newTodoItem !== '') {
                this.$store.commit('addOneItem', this.newTodoItem);
                this.clearInput();
```

Vuex : [리팩토링] mutations 적용

3. TodoApp 할 일 삭제 : 리팩토링 mutations 적용

1) store.js의 mutations 속성내에 App.vue에 있는 removeOneItem() 메서드를 이동시킨다.

2) App.vue의 <TodoList v-on:removeItemEvent="removeOneItem"></TodoList> ➔ <TodoList></TodoList>로 변경한다.

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        todoItems: storage.fetch()
    },
    mutations: {
        removeOneItem(state, payload) {
            localStorage.removeItem(payload.todoItem.item);
            state.todoItems.splice(payload.index, 1);
        },
    }
});
```

src/App.vue

```
<template>
    <TodoList v-on:toggleItemEvent="toggleOneItem"></TodoList>
</template>
```

Vuex : [리팩토링] mutations 적용

3. TodoApp 할 일 삭제 : 리팩토링 mutations 적용

3) TodoList 에서 할 일 삭제 이벤트가 발생했을 때 App.vue에게 Event를 보내는 대신에 store에 저장된 removeOneItem() 메서드를 직접 호출한다.

this.$emit('removeItemEvent',todoItem,index); ➔ this.$store.commit('removeOneItem', {todoItem, index});

※ 원래는 this.$store.commit('removeOneItem', {todoItem:todoItem, index:index}); 이지만 ES6 Enhanced Object Literals 를 적용해서 전달하는 객체의 key와 value값이 동일하므로 {todoItem, index} 로 전달한다.

src/component/TodoList.vue

```
export default {
    methods:{
        removeTodo(todoItem, index) {
            this.$store.commit('removeOneItem', {todoItem, index});
        },
```

Vuex : [리팩토링] mutations 적용

4. TodoApp 할 일 완료 : 리팩토링 mutations 적용

1) store.js의 mutations 속성내에 App.vue에 있는 toggleOneItem() 메서드를 이동시킨다.

2) App.vue의 <TodoList v-on:toggleItemEvent="toggleOneItem"></TodoList> ➔ <TodoList></TodoList>로 변경한다.

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        todoItems: storage.fetch()
    },
    mutations: {
        toggleOneItem(state, payload) {
            state.todoItems[payload.index].completed = !state.todoItems[payload.index].completed;
            localStorage.removeItem(payload.todoItem.item);
            localStorage.setItem(payload.todoItem.item, JSON.stringify(payload.todoItem));
        },
    }
});
```

src/App.vue

```
<template>
    <TodoList></TodoList>
</template>
```

Vuex : [리팩토링] mutations 적용

4. TodoApp 할 일 완료 : 리팩토링 mutations 적용

3) TodoList 에서 할 일 완료 이벤트가 발생했을 때 App.vue에게 Event를 보내는 대신에 store에 저장된 toggleOneItem() 메서드를 직접 호출한다.

this.$emit('toggleItemEvent',todoItem, index); ➔ this.$store.commit('toggleOneItem', {todoItem, index});

src/component/TodoList.vue

```
export default {
    methods:{
        toggleComplete(todoItem, index) {
            this.$store.commit('toggleOneItem', {todoItem, index});
        },
```

Vuex : [리팩토링] mutations 적용

5. TodoApp 할 일 모두 삭제 : 리팩토링 mutations 적용

1) store.js의 mutations 속성내에 App.vue에 있는 removeAllItems() 메서드를 이동시킨다.

2) App.vue의 <TodoFooter v-on:removeAllItemEvent="removeAllItems"></TodoFooter> ➔
<TodoFooter></TodoFooter>로 변경한다.

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        todoItems: storage.fetch()
    },
    mutations: {
        removeAllItems(state) {
            localStorage.clear();
            state.todoItems = [];
        },
    }
});
```

src/App.vue

```
<template>
    <TodoFooter></TodoFooter>
</template>
```

Vuex : [리팩토링] mutations 적용

5. TodoApp 할 일 모두 삭제 : 리팩토링 mutations 적용

3) TodoFooter 에서 할 일 모두 삭제 이벤트가 발생했을 때 App.vue에게 Event를 보내는 대신에 store에 저장된 `removeAllItems()` 메서드를 직접 호출한다.

this.$emit('removeAllItemEvent'); ➔ this.$store.commit('removeAllItems');

src/component/TodoFooter.vue

```
export default {
    methods:{
        toggleComplete(todoItem, index) {
            this.$store.commit('removeAllItems');
        },
```

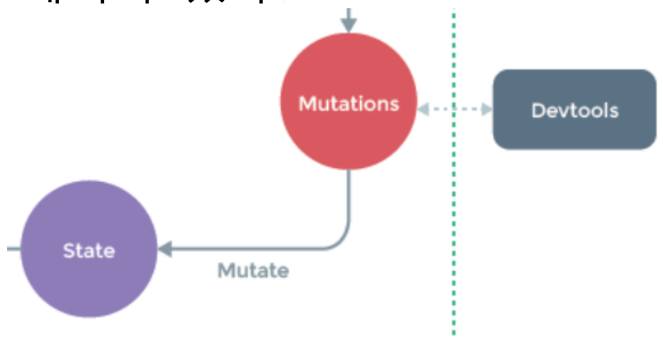
Vuex : 기술 요소

state는 왜 직접 변경하지 않고 mutations로 변경할까?

- 여러 개의 컴포넌트에서 아래와 같이 state 값을 변경하는 경우 어느 컴포넌트에서 해당 state를 변경했는지 추적하기가 어렵다.

```
methods:{
    increaseCounter() { this.$store.state.counter++ }
}
```

- 특정 시점에 어떤 컴포넌트가 state를 접근하여 변경한 것인지 확인하기 어렵기 때문이다.
- 따라서, 뷰의 반응성을 거스르지 않게 명시적으로 상태 변화를 수행한다. 반응성, 디버깅, 테스팅 혜택이 있다.



Vuex - Helper 함수

Vuex Hepler

Helper의 사용법

- Helper를 사용하고자 하는 vue 파일에서 아래와 같이 해당 Helper를 로딩한다.
- ...는 ES6의 Object Spread Operator

```
//App.vue
import { mapState } from 'vuex'
import { mapGetters } from 'vuex'
import { mapMutations } from 'vuex'
import { mapActions } from 'vuex'

export default {
  computed: { ...mapState(['num']), ...mapGetters(['countedNum']) },
  methods:
     { ...mapMutations(['clickBtn']), ...mapActions(['asyncClickBtn']) }
}
```

Vuex Hepler

mapState

- Vuex에 선언한 state 속성을 뷰 컴포넌트에 더 쉽게 연결해주는 헬퍼

```
//App.vue
import { mapState } from 'vuex'

export default {
  computed: { ...mapState(['num']) }
  //num() { return this.$store.state.num }
}
//store.js
state: {
   num: 10
}
```

```
<!-- <p>{{this.$store.state.num}}</p> -->
<p>{{num}}</p>
```

Vuex Hepler

mapGetters

- Vuex에 선언한 getters 속성을 뷰 컴포넌트에 더 쉽게 연결해주는 헬퍼

```
//App.vue
import { mapGetters } from 'vuex'

export default {
  computed : { ...mapGetters(['reverseMessage']) }
}

//store.js
getters: {
   reverseMessage(state) {
      return state.msg.split('').reverse().join('');
   }
}
```

```
<!-- <p>{{this.$store.getters.reverseMessage}}</p> -->
<p>{{reverseMessage}}</p>
```

computed : https://vuejs.org/v2/guide/computed.html

Vuex : Hepler 함수 [리팩토링] getters와 mapGetters 적용

1. TodoApp : 리팩토링 getters 적용

src/store/store.js

```
export const store = new Vuex.Store({
    getters: {
        getTodoItems(state) {
            return state.todoItems;
        }
    },
});
```

src/components/TodoList.vue

```
<template>
   <transition-group name="list" tag="ul">
            <li v-for="(todoItem, index) in this.$store.getters.getTodoItems"... >
    ...
</template>
```

Vuex : Hepler 함수 [리팩토링] getters와 mapGetters 적용

1-1. TodoApp : 리팩토링 getters 적용 #1

src/components/TodoList.vue

```
<template>
 <transition-group name="list" tag="ul">
            <li v-for="(todoItem, index) in todoItems"... >
 ...
</template>

<script>
export default {
    methods: {
        ...
    },
    computed: {
      todoItems() {
        return this.$store.getters.getTodoItems;
      }
    }
}
</script>
```

Vuex : Hepler 함수 [리팩토링] getters와 mapGetters 적용

1-1. TodoApp : 리팩토링 getters 적용 #2

1) TodoList.vue에 mapGetters를 import 한다.

2) TodoList.vue에 computed 속성을 추가한다. computed 속성내에 전개연산자(spread operator)을 사용하여 mapGetters 선언한다.

src/components/TodoList.vue

```
<template>
 <transition-group name="list" tag="ul">
            <li v-for="(todoItem, index) in getTodoItems"... >
 ...
</template>
<script>
import { mapGetters } from 'vuex'
export default {
    methods: {
      ...
    },
    computed: {
      ...mapGetters(['getTodoItems'])
    }
}
</script>
```

Vuex Hepler

mapMutations

- Vuex에 선언한 mutations 속성을 뷰 컴포넌트에 더 쉽게 연결해주는 헬퍼

```
//App.vue
import { mapMutations } from 'vuex'

export default {
   methods : {
    ...mapMutations(['setValue']),
    authLogin() {},
    displayTable() {}
   }
}
//store.js
mutations: {
   setValue(state, value) {
      this.values += value
   }
}
```

```
<button v-on:click="setValue">변경</button>
```

Vuex Hepler

mapActions

- Vuex에 선언한 actions 속성을 뷰 컴포넌트에 더 쉽게 연결해주는 헬퍼

```
//App.vue
import { mapActions } from 'vuex'

export default {
  methods : {
    ...mapActions(['delayClickBtn']),
  }
}

//store.js
actions: {
   delayClickBtn(context) {
      setTimeout(() => context.commit('clickBtn'), 2000);
   }
}
```

```
<button @click="delayClickBtn">delay popup message</button>
```

Vuex Hepler

헬퍼의 유연한 문법

- Vuex에 선언한 속성을 그대로 컴포넌트에 연결하는 문법

```
//배열 리터럴
export default {
   methods : {
    ...mapMutations(['clickBtn', 'addNumber']),
   }
}
```

- Vuex에 선언한 속성을 컴포넌트의 특정 메서드에 연결하는 문법

```
//객체 리터럴
export default {
   methods : {
    ...mapMutations({ popupMsg : 'clickBtn' }),
   }
}
```

Vuex : Hepler 함수 [리팩토링] mapMutations 적용

2. TodoApp 할 일 삭제 : 리팩토링 mapMutations 적용

1) TodoList.vue의 methods속성에 정의된 removeTodo() 메서드를 제거한다.
2) TodoList.vue에 mapMutations를 import 하고, methods 속성에 spread operator를 사용하여 mapMutations을 선언
3) v-on:click="removeTodo({todoItem, index})"의 인자의 타입을 객체로 수정한다.

src/components/TodoList.vue

```
<template>
   <transition-group name="list" tag="ul">
            <li v-for="(todoItem, index) in this.storedTodoItems"... >
                 ...
                <span class="removeBtn" v-on:click="removeTodo({todoItem, index})">
</template>
<script>
import { mapGetters, mapMutations } from 'vuex'
export default {
    methods: {
        ...mapMutations({
          removeTodo: 'removeOneItem',
        }),
    },
}
</script>
```

Vuex : Hepler 함수 [리팩토링] mapMutations 적용

3. TodoApp 할 일 완료 : 리팩토링 mapMutations 적용

1) TodoList.vue의 methods속성에 정의된 toggleComplete() 메서드를 제거한다.

2) v-on:click="toggleComplete({todoItem, index})"의 인자의 타입을 객체로 수정한다.

src/components/TodoList.vue

```
<template>
   <transition-group name="list" tag="ul">
            <li v-for="(todoItem, index) in this.storedTodoItems"... >
                <i v-on:click="toggleComplete({todoItem, index})"></i>
</template>
<script>
import { mapGetters, mapMutations } from 'vuex'
export default {
    methods: {
        ...mapMutations({
          removeTodo: 'removeOneItem',
          toggleComplete: 'toggleOneItem',
        }),
    },
}
</script>
```

Vuex : Hepler 함수 [리팩토링] mapMutations 적용

4. TodoApp 할 일 모두 삭제 : 리팩토링 mapMutations 적용

1) TodoFooter.vue의 methods속성에 정의된 clearTodo() 메서드를 제거한다.
2) TodoFooter.vue에 mapMutations를 import 하고, methods 속성에 spread operator를 사용하여 mapMutations을 선언

src/components/TodoFooter.vue

```
<script>
import { mapMutations } from 'vuex'

export default {
    methods: {
        ...mapMutations({
          clearTodo: 'removeAllItems'
        }),
    },
}
</script>
```

Vuex : Store 모듈화

프로젝트 구조화와 모듈화 방법 1

- 아래와 같은 store 구조를 어떻게 모듈화 할 수 있을까?

```
 //store.js
import Vue from 'vue'
import Vuex from 'vuex'

export const store = new Vuex.Store({
     state: {}
     getters: {},
     mutations: {},
     actions: {}
});
```

- 힌트! Vuex.Store({})의 속성을 모듈로 연결

Vuex : Store 모듈화

1. Store 모듈화 1

1) store 디렉토리 아래에 getters.js 와 mutations.js 을 생성한다.

2) store.js 내의 getters 속성의 내용을 getters.js로 이동시킨다.

const 로 선언하여 상수로 정의하고, Arrow Function 형태로 수정한다.

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        todoItems: storage.fetch()
    },
    getters: {

    },
    mutations: {

    }
});
```

src/store/getters.js

```
export const storedTodoItems = (state) => {
    return state.todoItems;
}
```

Vuex : Store 모듈화

1. Store 모듈화 #1

3) store.js 내의 mutations 속성의 내용을 mutations.js로 이동시킨다. const 로 선언하여 상수로 정의하고, Arrow Function 형태로 수정한다.

4) store.js에서 getters.js와 mutations.js 를 import 한다. mutations 속성안에 getters: getters 로 선언한다.

src/store/mutations.js

```
const addOneItem = (state, todoItem) => {   ...     }
const removeOneItem = (state, payload) => {  ...     }
const toggleOneItem = (state, payload) => {  ...     }
const removeAllItems = (state) => {  ...     }

export {addOneItem, removeOneItem, toggleOneItem, removeAllItems}
```

src/store/store.js

```
import * as getters from './getters';
import * as mutations from './mutations';

const storage = { .. }
export const store = new Vuex.Store({
    state: { .. },
    mutations: { mutations },
    getters: { getters }
```

Vuex : Store 모듈화

프로젝트 구조화와 모듈화 방법 #2

- App의 규모가 커져서 1개의 store로는 관리하기 힘들 때 modules 속성 사용

```
 //store.js
import Vue from 'vue'
import Vuex from 'vuex'
import todo from 'modules/todo.js'

export const store = new Vuex.Store({
     modules: {
         //모듈명칭 : 모듈파일명  todo:todo
         todo
     }
});

//todo.js
const state = {}
const getters = {}
const mutations = {}
const actions = {}
```

Vuex : Store 모듈화

2. Store 모듈화 #2

1) store 디렉토리 아래에 modules 디렉토리를 생성하고 todoApp.js를 생성한다.

2) store.js 내의 state, getters, mutations 속성의 내용을 todoApp.js로 이동시킨다.

todoApp 내에 state, getters, mutations를 const로 정의하고 export 한다.

src/store/store.js

```
import todoApp from './modules/todoApp';

Vue.use(Vuex);

export const store = new Vuex.Store({
    modules: {
        todoApp  //todoApp:todoApp
    }
});
```

src/store/modules/todoApp.js

```
const storage = { .. };
const state = { .. }
const getters = {
   storedTodoItems(state) { .. }
}
const mutations = {
    addOneItem(state, todoItem) {      },
    removeOneItem(state, payload) {    },
    toggleOneItem(state, payload) {    },
    removeAllItems(state) {   }
}
export default {
    state, getters, mutations
}
```

Axios

Axios

- Axios는 현재 가장 성공적인 HTTP **클라이언트 라이브러리** 중 하나입니다.
  아직 1.x 버전이 릴리즈 되지 않았지만 스타가 6만개가 넘을 정도로 인기가 좋습니다.
  특별히 언급할 만한 부분은 요청 취소와 TypeScript를 사용할 수 있는 것 입니다.

>> Features <<

- Make XMLHttpRequests from the browser
- Make http requests from node.js
- Supports the Promise API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- Client side support for protecting against XSRF

Axios 와 vue-axios

- 1.Axios (https://github.com/axios/axios)
- 2.Vue-Axios (https://github.com/imcvampire/vue-axios)

- 1. axios와 vue-axios 설치하기

```
npm install --save axios vue-axios
```

Express

- Express (https://github.com/expressjs/express)

Express 는 Node.js를 위한 빠르고 개방적이며 간결한 웹 프레임워크입니다.

Express 프레임워크를 사용한 Node 웹서버에서 간단한 REST API 를 구현하고,

Client에서는 axios 라이브러리를 사용하여 비동기적으로 통신합니다.

```
//app.js

var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

```
$ node app.js
```

CORS

- CORS (https://github.com/expressjs/cors)

Ajax에서 보안 상의 이슈 때문에 동일 출처(Single Origin Policy)를 기본적으로 웹에서는 준수하게 됩니다.

- SOP(Single Origin Policy)

: 같은 Origin에만 요청을 보낼 수 있다.

- CORS(Cross-Origin Resource Sharing)

: Single Origin Policy를 우회하기 위한 기법

: 서로 다른 Origin 간에 resource를 share 하기 위한 방법

- Origin 이란?

: URI 스키마 (http, https) + hostname (localhost) + 포트 (8080, 18080)

axios 와 vue-axios

1. Axios 와 VueAxios 의 사용

- axios 와 VueAxios 를 import 하여 사용한다.

src/store/store.js

```
import Vue from 'vue';
import Vuex from 'vuex';
import axios from 'axios';
import VueAxios from 'vue-axios';

Vue.use(Vuex);
//순서가 바뀌면 않됩니다
Vue.use(VueAxios, axios);
```

axios 와 vue-axios

2. TodoApp 할 일 목록 :

1) store.js 의 state 의 todoItems 변수를 초기화 한다.

2) getters , mutations , actions 프로퍼티를 추가한다.

: actions 프로퍼티의 loadTodoItems() 에서 axios.get() 을 호출한다.

src/store/store.js

```
export const store = new Vuex.Store({
    state: {
        todoItems: []
    },
    getters : {
        getTodoItems(state) {
            return state.todoItems;
        }
    },
    mutations: {
        setTodoItems(state, items) {
            state.todoItems = items;
        },
    },
```

src/store/store.js

```
    actions: {
        loadTodoItems (context) {
            axios
              .get('http://localhost:4500/api/todos')
              .then(r => r.data)
              .then(items => {
                context.commit('setTodoItems', items)
              })
        },
    }

});
```

axios 와 vue-axios

2. TodoApp 할 일 목록 :

1) 화면 load 할 때 store의 actions에 정의된 loadTodoItem() 를 호출한다.

2) template 영역에서 사용할 수 있도록 mapGetters 헬퍼 함수를 사용하여 getTodoItems() 를 정의한다.

src/components/TodoList.js

```
<template>
 <transition-group name="list" tag="ul">
   <li v-for="(todoItem, index) in getTodoItems" v-bind:key="index" class="shadow">
    ...
</template>
<script>
import { mapGetters } from 'vuex';

export default {
    mounted () {
      this.$store.dispatch('loadTodoItems');
    },
    computed: {
      ...mapGetters(['getTodoItems'])
    },

}
</script>
```

axios 와 vue-axios

3. TodoApp 할 일 삭제 :

1) actions 프로퍼티의 removeTodoItem() 에서 axios.delete() 을 호출한다.

src/store/store.js

```
  actions: {
    removeTodoItem(context, payload) {
       axios
       .delete(`http://localhost:4500/api/todos/${payload.id}`)
       .then(r => r.data)
       .then(items => {
          context.commit('setTodoItems', items)
       })
    },
   }
```

src/components/TodoList.js

```
<span class="removeBtn" v-on:click="removeTodoItem(todoItem)">...</span>

<script>
import { mapGetters, mapActions } from 'vuex'
export default {
    methods: {
        ...mapActions(['removeTodoItem']),
    }
}
</script>
```

axios 와 vue-axios

4. TodoApp 할 일 추가 :

1) actions 프로퍼티의 addTodoItem() 에서 axios.post() 을 호출한다.

src/store/store.js

```
  actions: {
    addTodoItem(context, payload) {
            axios
              .post(`http://localhost:4500/api/todos/`,payload)
              .then(r => r.data)
              .then(items => {
                context.commit('setTodoItems', items)
              })
    },
   }
```

src/components/TodoInput.js

```
methods:{
        addTodo() {
            if (this.newTodoItem !== '') {
                const itemObj = { completed: false, item: this.newTodoItem };
                this.$store.dispatch('addTodoItem', itemObj);
                this.clearInput();
            }else { .. }
        },
```

axios 와 vue-axios

5. TodoApp 할 일 모두 삭제 :

1) actions 프로퍼티의 removeAllTodoItems() 에서 axios.delete() 을 호출한다.

src/store/store.js

```
  actions: {
    removeAllTodoItems(context) {
            axios
              .delete(`http://localhost:4500/api/todos/`)
              .then(r => r.data)
              .then(items => {
                context.commit('setTodoItems', items)
              })
        },
   }
```

src/components/TodoFooter.js

```
<span class="clearAllBtn" v-on:click="removeAllTodoItems">Clear All</span>

<script>
import { mapActions } from 'vuex';
export default {
    methods: {
        ...mapActions(['removeAllTodoItems']),
    }
}
```

axios 와 vue-axios

6. TodoApp 할 일 완료 :

1) actions 프로퍼티의 toggleTodoItem() 에서 axios.put() 을 호출한다.

src/store/store.js

```
  actions: {
    toggleTodoItem(context, payload) {
            axios
              .put(`http://localhost:4500/api/todos/${payload.id}`,payload)
              .then(r => r.data)
              .then(items => {
                context.commit('setTodoItems', items)
              })
        },
  }
```

src/components/TodoList.js

```
<i class="fas fa-check checkBtn" v-bind:class="{checkBtnCompleted: todoItem.completed}"
                    v-on:click="toggleTodo(todoItem)"></i>
<script>
export default {
    methods: {
        toggleTodo(todoItem) {
                todoItem.completed = !todoItem.completed;
                this.$store.dispatch('toggleTodoItem', todoItem);
        }
    }
}
```

Vue - router

Vue Router

- Routing이란?

: 웹페이지 간의 이동 방법을 말합니다. Routing은 현대 웹 앱 형태 중 하나인 싱글 페이지 애플리케이션(SPA, Single Page Application)에서 주로 사용합니다.

: 일반적으로 브라우저에서 웹 페이지를 요청하면 서버에서 응답을 받아 웹 페이지를 다시 사용자에게 돌려주는 시간 동안 화면 상에 깜빡거림 현상이 나타납니다. 이런 부분들을 라우팅으로 처리하면 화면을 매끄럽게 전환할 수 있으며, 더 빠르게 화면을 조작할 수 있어 사용자 경험이 향상됩니다.

- Vue Router

: 뷰 라우터는 뷰에서 라우팅 기능을 구현할 수 있도록 지원하는 공식 라이브러리입니다.

- Github: https://github.com/vuejs/vue-router
- 문서: https://router.vuejs.org/kr/

Vue Router

- Vue Router가 제공하는 기능

✓ 중첩된 라우트/뷰 매핑

✓ 모듈화된, 컴포넌트 기반의 라우터 설정

✓ 라우터 파라미터, 쿼리, 와일드카드

✓ 세밀한 네비게이션 컨트롤

✓ active CSS 클래스를 자동으로 추가해주는 링크

✓ 사용자 정의 가능한 스크롤 동작

: 뷰 라우터를 이용하여 뷰로 만든 페이지 간에 자유롭게 이동할 수 있습니다. 뷰 라우터를 구현할 때 필요한 특수 태그의 기능은 다음과 같습니다.

태그	설명
<router-link to="URL 값">	페이지 이동 태그. 화면에서는 <a>로 표시되며 클릭하면 to에 지정한 URL로 이동합니다.
<router-view>	페이지 표시 태그. 변경되는 URL에 따라 해당 컴포넌트를 뿌려주는 영역입니다.

Vue Router : 프로젝트 생성

1. 새로운 프로젝트 생성 : vue-router-app

```
vue create vue-router-app
```

```
 Vue CLI v3.4.0
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
  Manually select features
```

2. Vue router 설치하기

```
cd vue-router-app

npm install --save vue-router
```

3. Vue router 예제 코드 자동 생성

```
vue add router
```

? Use history mode for router? (Requires proper server setup for index fallback in production) No

Vue Router

- 라우터 객체 생성

main.js : 뷰 인스턴스 생성 객체에는 router 속성이 있다. 뷰 라우터를 사용하려면 이 속성으로 뷰 라우터 객체를 넘겨줘야 한다.

router.js : 뷰 라우터는 플러그인 형태이기 때문에 Vue.use() 함수를 이용해서 등록한다. VueRouter 클래스로 라우터 객체를 생성한다.

src/main.js

```
import Vue from 'vue';
import App from './App.vue';
import router from './router';

Vue.config.productionTip = false

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

src/router/index.js

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import Home from '../views/Home.vue';

Vue.use(VueRouter)

export default new Router({
  routes: [
    { path: '/', name: 'home', component: Home },
    { path: '/about', name: 'about',
     component: () => import('../views/About.vue') }
  ]
})
```

Vue Router

- 라우터 뷰 : router-view

App.vue : App.vue 루트 컴포넌트에 라우터 뷰를 추가하면 됩니다. 라우팅이 경로에 따라 컴포넌트를 바꾸어 가면서 렌더링 하는데 렌더링 해주는 부분에 <router-view> 태그를 사용한다.

- 라우터 링크 : router-link

: 라우터에 등록된 링크는 <a> 태그 보다는 <router-link> 태그를 사용하길 권장하는데 이유는 다음과 같다.

✓ 히스토리 모드에서는 주소 체계가 달라서 <a> 태그를 사용할 경우 모드 변경시 주소값을 일일이 변경해 줘야 하지만 <router-link>는 변경할 필요가 없다.

✓ <a> 태그를 클릭하면 화면을 갱신하는데 <router-link>는 이를 차단 해준다. 갱신 없이 화면을 이동할 수 있다.

src/App.vue

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
    </div>
    <router-view/>
  </div>
</template>
```

Vue Router

- 중첩된 라우팅 : children 속성

: routes 속성에 정의하는 컬렉션에는 children 속성이 있다. 이는 특정 라우팅의 하위 경로가 변경됨에 따라 하위 컴포넌트를 변경할 수 있는 기능이다.

- 먼저 View 컴포넌트 작성

: 부모 라우터(Posts)에서는 자식 라우터들(PostNew, PostDetail)을 렌더링 하기 위한 뷰가 필요하므로 <router-view> 태그를 삽입했다. 중첩된 하위 경로가 변경될 때 이 부분에 해당 컴포넌트가 그려진다.

src/views/Posts.vue

```
<template>
  <div>
    <h1>Posts</h1>
    <router-view></router-view>
  </div>
</template>
```

src/views/PostNew.vue

```
<template>
  <div>
    <h1>Post New page</h1>
  </div>
</template>
```

src/views/PostDetail.vue

```
<template>
  <div>
    <h1>Post Detail page</h1>
  </div>
</template>
```

Vue Router

- children 속성

: 중첩된 라우터는 children 속성으로 하위 라우터를 정의할 수 있다.

/posts 경로를 포함한 하위 경로인 /posts/new와 /posts/detail을 children 옵션으로 설정한다.

- 네비게이션 메뉴에 추가

: 생성한 라우터 링크를 루트 컴포넌트 네비게이션 메뉴에 추가한다.

src/router/index.js

```
import Posts from '../views/Posts.vue';
import PostNew from '../views/PostNew.vue';
import PostDetail from '../views/PostDetail.vue';
…
export default new Router({
  routes: [

      { path: '/posts', component: Posts,
            children: [
            { path: 'new', component: PostNew},
            { path: 'detail', component: PostDetail }
            ]
      }
  ]
})
```

src/App.vue

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
      <router-link to="/posts">Posts</router-link> |
      <router-link to="/posts/new" exact>New Post
       </router-link> |
      <router-link to="/posts/detail" exact>Detail Post
       </router-link> |
    </div>
    <router-view/>
  </div>
</template>
```

Vue Router

- 동적 라우터 매핑

: /posts/detail을 Post Id에 따라 내용이 달라지도록 라우터 경로에 추가

/posts/1, /posts/2

: { path: ' detail ' } 부분을 { path: ' :id ' } 로 변경한다.

: 동적 라우트 매핑으로 그려진 컴포넌트에서 id 값에 접근할 때는 this.$route 변수로 라우터에 접근할 수 있으며, this.$route.params.id 로 id 값을 가져온다.

src/router/index.js

```
export default new Router({
  routes: [

      { path: '/posts', component: Posts,
            children: [
            { path: 'new', component: PostNew},
            { path: ':id', name: 'post',
              component: PostDetail }
            ]
      }
  ]
})
```

src/views/PostDetail.vue

```
<template>
  <div>
    <h1>This is an id: {{this.$route.params.id}}
      Post Detail page</h1>
  </div>
</template>
```

Vue Router

- 라우터 링크 스타일

: 라우터 링크는 스타일과 관련된 사항이 있다. 경로에 따라 CSS 클래스 명이 자동으로 추가 되는 것이다.

Vuejs가 알아서 CSS 클래스명을 추가 하기 때문에 개발자는 CSS 클래스 정의만 추가하면 된다.

루트 컴포넌트(App.vue)에 CSS를 추가한다.

- router-link-active: 경로 앞부분만 일치해서 추가되는 클래스
- router-link-exact-active: 모든 경로가 일치해야만 추가되는 클래스

src/App.vue

```
<template>
   <div id="app">
      <div id="nav">
         <router-link to="/">Home</router-link>
      </div>
   </div>
</template>
<style>
#nav a.router-link-exact-active {
  color: #42b983;
}
</style>
```

Vue Router

- 데이터 가져오기

: 서버로 부터 데이터를 가져오는 기능 즉, Data Fetching 이라고 부르는데 각 화면별로 라우팅이 일어나는 시점에 데이터를 불러와야 한다.

: src/api/index.js에 배열 데이터를 반환 해주는 mock server(가짜 서버) 역할을 하는 js 파일

- PostDetail.vue

: Post의 id로 상세 정보를 가져온다.

: 컴포넌트 생성시 created() lifecycle 메서드에서 데이터를 fetch한다.

: 라우터 링크를 통해 경로가 변경되는 경우 /posts/:id 경로에 매칭되는 컴포넌트(Post 컴포넌트)는 화면이 refresh 될 경우에만 created() 메서드가 동작한다. 단순히 :id 값이 변경되어도 create() 메서드에서 호출하는 fetchData() 함수가 호출되지 않는다.

: this.$route 객체는 라우팅 변경이 일어날 때마다 호출된다. 따라서 watch 에서 감시하고 있다가 변경되면 즉시 fetchData() 함수를 호출하는 로직을 추가하자.

Vue Router

- PostDetail (상세정보)

src/views/PostDetail.vue

```
<template>
  <div>
    <h2>View Post</h2>
    <div v-if="loading">Loading...</div>
    <div v-if="post">
      <h3>[ID: {{post.id}}]</h3>
      <div>{{post.text}}</div>
    </div>
  </div>
</template>
```

src/views/PostDetail.vue

```
<script>
import { Post } from '../api';
export default {
  data() {
    return {
      post: null,
      loading: true
    }
  },
  created() { this.fetchData() },
  watch: { '$route': 'fetchData' },
  methods: {
    fetchData() {
      this.post = null
      this.loading = true
      Post.get(this.$route.params.id)
      .then(data => { this.post = data
                      this.loading = false })
    }
  }
}
</script>
```

Vue Router

- Posts (리스트)

src/views/Posts.vue

```
<template>
  <div class="posts">
    <h1>Posts</h1>
    <div v-if="loading">Loading...</div>
    <div v-for="post in posts" :key="post.id">
      <router-link :to="{ name: 'post', params: { id:
post.id } }">
        [ID: {{post.id}}] {{post.text | summary}}
      </router-link>
    </div>
    <router-view></router-view>
  </div>
</template>
```

https://scotch.io/tutorials/how-to-create-filters-in-vuejs-with-examples

src/views/Posts.vue

```
<script>
import { Post } from '../api';
export default {
  data() {
    return { loading: true, posts: [] }
  },
  created() { this.fetchData() },
  filters: {
    summary(val) {
      return val.substring(0, 20) + '...';
    }
  },
  methods: {
    fetchData() {
      this.loading = true;
      Post.list()
      .then(data => {
        this.posts = data;
        this.loading = false;
      })
    }
  }
}
</script>
```

Vue Router

- PostNew (등록)

src/views/PostNew.vue

```
<template>
 <div>
    <h2>New Post</h2>
    <form @submit.prevent="onSubmit">
      <textarea cols="30" rows="10"
v-model="inputTxt" :disabled="disabled"></textarea>
      <input
type="submit" :value="btnTxt" :disabled="disabled"/>
    </form>
  </div>
</template>
```

src/views/PostNew.vue

```
<script>
import { Post } from '../api';
export default {
  data() {
    return { isSaving: false, inputTxt: '', }
  },
  computed: {
    btnTxt() {
      return this.isSaving ? 'Saving...' : 'Save';
    },
    disabled() { return this.isSaving; }
  },
  methods: {
    onSubmit() {
      this.isSaving = true;
      Post.create(this.inputTxt).then(() => {
        this.isSaving = false
        this.inputTxt = ''
        this.$router.push('/posts')
      })
    }
  }
}
</script>
```

한국소프트웨어기술진흥협회
KOSTA Korea Software Technology Association

판교 | 경기도 성남시 분당구 삼평동 대왕판교로 670길 유스페이스2 B동 8층 T. 070-5039-5805
가산 | 서울시 금천구 가산동 371-47 이노플렉스 1차 2층 T. 070-5039-5815
웹사이트 | http://edu.kosta.or.kr **팩스 |** 070-7614-3450