Vue3 with Typescript

2023.6.28 ~ 6.30

백명숙

Learning Object(학습모듈) 및 커리큘럼

LO	커리큘럼
Vue3 소개	- Vue3.0 소개 및 특징 - Vue-CLI 설치 및 사용
ECMAScript와 TypeScript	- ECMAScript - TypeScript 기본
Vue.js Directive와 Component	- Vue.js Directive - Vue.js Component 간의 통신
Vue3 App 작성 및 리팩토링	- Todo(할일 관리) App 프로젝트 생성과 구현 - Todo App 리팩토링 - Todo App UX 개선
Vuex4	- Todo App에 Vuex 적용 - Vuex Store 모듈화 - Type 선언하기
Nodejs + Express / Axios	- Typescript 기반 NodeJS + Express 서버 구현 - Todo App에 axios 사용하여 서버와 http 통신 - Mode와 Environment
Vue3 Router	- Vue3 Router

Vue.JS with Typescript

백명숙

Vue.JS 프로젝트 환경 설정

- Node.js 설치하기

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

- Visual Studio Code 설치 및 Plug In 설치하기

VS Code 설치는 Visual Studio Code 에서 하실 수 있습니다.

Vue.JS 프로젝트 환경 설정

- Visual Studio Code Plug In 설치하기

1. Vue Language Features (Volar)

https://marketplace.visualstudio.com/items?itemName=Vue.volar

2. TypeScript Vue Plugin (Volar)

https://marketplace.visualstudio.com/items?itemName=Vue.vscode-typescript-vue-plugin

3. Vue Snippets

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

4. ESLint

https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

5. VSCode Material Icon Theme

https://marketplace.visualstudio.com/items?itemName=PKief.material-icon-theme

6. Auto Close Tag

https://marketplace.visualstudio.com/items?itemName=formulahendry.auto-close-tag

Vue.JS 프로젝트 환경 설정

- Vue CLI 설치 (https://cli.vuejs.org/)

```
npm i -g @vue/cli
```

```
vue --version
```

- Chrome에 Vue.js DevTools 설치

Typescript와 Frontend 라이브러리

- Stack Overflow에서 2020년 전세계 6만 5천 명의 개발자를 대상으로 실시한 설문조사에 따르면 타입스크립트는 개발자가 가장 좋아하고 관심을 가지는 프로그래밍 언어 3위에 올랐습니다. 타입스크립트를 사용하는 대부분의 개발자들은 이 언어를 계속 사용하고 싶다고 응답 했습니다.
- Stackoverflow 2022 선호하는 Language

(https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-language-want)

Vue

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

- Vue.js는 Google Creative Laps에서 근무하던 Evan You가 개발하였으며, “AngularJS 에서 내가 좋아하는 특성만 담은 가벼운 라이브러리를 만들 수 있지 않을까?” (Between the Wires의 인터뷰 참고) 라는 점과 좀 더 쉽게 접근 할 수 있는 웹 프레임워크를 만들고자 탄생하게 되었습니다.

- Vue의 코어 라이브러리는 화면 단 데이터 표현에 관한 기능들을 중점적으로 지원하며, 프레임워크의 기능인 라우터, 상태 관리, 테스팅 등을 쉽게 결합 할 수 있는 형태로 제공 됩니다. 즉, 라이브러리 역할 뿐 만 아니라 프레임워크 역할도 할 수 있습니다. 단일 프레임워크 와는 달리 Vue는 이를 점진적으로 채택 할 수 있도록 설계되어 Progressive 프레임워크 라고 표현 합니다.

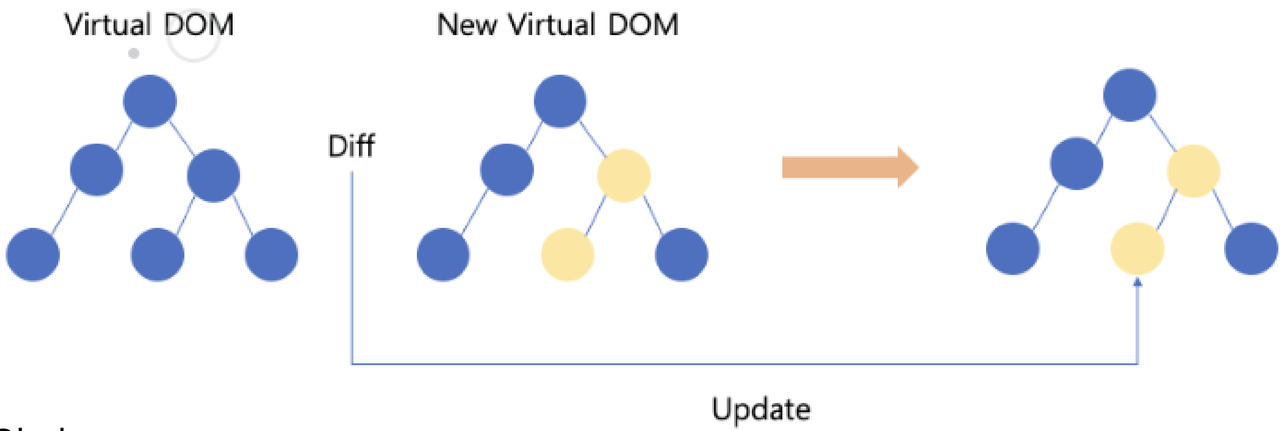
Vue 특징

- Virtual DOM의 사용
- Tree Shaking (트리 쉐이킹)
- TypeScript 지원
- SFC (Single-File Component)
- 양방향 데이터 바인딩
- component 기반
- Vue-CLI
- Template 문법

Vue3 특징

1. Virtual DOM의 사용

Virtual DOM은 Real DOM을 너무 많이 업데이트 하는 비효율적인 연산을 줄이고자 만들어진 가상의 DOM입니다. 또한 DOM 구조를 체크하고 변경하는 방식이 복잡하여 단순화/자동화 하기 위해 만들어 지기도 하였습니다. Virtual DOM은 메모리에 저장되어 사용 됩니다.



Virtual DOM 동작 원리

- UI가 변경되면 전체 UI를 Virtual DOM으로 렌더링 합니다.
- 현재 Virtual DOM과 이전 Virtual DOM을 비교해 차이를 계산하여 변경된 부분만 실제 DOM에 반영 합니다.
- 가상 돔(Virtual Dom) 렌더링은 변경된 부분만 DOM에 적용하는 방식으로 DOM 처리 횟수를 최소화 하고, 상태 변화를 자동으로 감지하며, 빠른 화면 렌더링(Rendering)이 가능 하도록 해줍니다.

Vue3 특징

1.1 Vue3의 Virtual DOM 새로운 렌더링 전략

첫 번째, 템플릿 구문에서 정적 요소와 동적 요소를 구분하여 트리를 순환 할 때 동적 요소만 순환할 수 있도록 하였습니다. 구문 내의 정적인 영역을 미리 블록으로 구분하여 렌더링 시 정적 블록에는 접근하지 않고 동적인 요소가 있는 코드에만 접근해 렌더링 트리의 탐색을 최적화 하는 방식입니다.

두 번째, 렌더링 시 객체가 여러 번 생성 되는 것을 방지하기 위하여 컴파일러가 미리 템플릿 구문 내 정적 요소, 서브트리, 데이터 객체 등을 탐지해 렌더러(Renderer) 함수 밖으로 호이스팅(Hoisting) 합니다. 이를 통해 렌더링 시 Renderer 마다 객체를 다시 생성하는 것을 방지하여 메모리 사용량을 낮추었습니다.

세 번째로 컴파일러가 미리 템플릿 구문 내에서 동적 바인딩 요소에 대해 플래그를 생성합니다. 컴파일러가 미리 생성해 둔 플래그로 필요한 부분만 처리하여 렌더링 속도를 향상 시킬 수 있습니다.

Vue3 특징

2. 트리 쉐이킹 (Tree-shaking) 강화

트리쉐이킹은 나무를 흔들어 잎을 떨어 트리는 것처럼 모듈을 번들링 하는 과정에서 사용하지 않는 코드를 제거하여 파일 크기를 줄이고, 로딩 성능을 향상 시키는 최적화 방안을 의미합니다. Vue3에서 이를 강화하여 Bundle의 크기를 절반 이상으로 대폭 줄일 수 있었습니다.

	Chrome mount	Chrome update avg	Chrome update best	Chrome memory (mb)
Vue 2 template + with	93.46	23.17	16.44	11.9
Vue 3 template + with	77.43	9.18	7.69	4.5
Improvement over v2 (in-browser compiled)	20.70%	152.40%	113.78%	-62.18%
Vue 2 render fn (manual h, no this access)	90.9	15.75	8.18	10.2
Vue 3 render fn (manual h, no this access)	76.46	7.56	5.33	7.4
Improvement over v2 (manual render function)	18.89%	108.33%	53.47%	-27.45%
Vue 2 template no with	97.44	13.18	8.29	9.9
Vue 3 template no with	62.82	5.64	3.78	4.5
Improvement over v2 (pre-compiled)	55.11%	133.69%	119.31%	-54.55%
Vue 2 raw (no reactive state)	88.67	12.27	7.99	8.6
Vue 3 raw (no reactive state)	47.6	3.37	2.38	3.7
Improvement over v2 (raw, no reactive state)	86.28%	264.09%	235.71%	-56.98%

출처: Vue 공식문서

Vue.js 3.0 개발팀은 릴리즈 노트에서 가상 DOM 최적화, 트리 쉐이킹 강화 등을 통해 이전 버전에 비해 번들 크기가 최대 41% 감소하였고 초기 렌더링은 최대 55% 빠르며 업데이트와 메모리 사용량은 최대 133% 빨라 졌다고 밝혔다. 위의 표는 실제로 2.0 버전과 3.0 버전을 비교하여 크롬(Chrome) 환경에서 성능 테스트를 해본 사용자들의 리포트이다. Vue.js 개발팀이 밝힌 바와 같이 이전 버전에 비해 눈에 띄는 성능 향상을 보이고 있다.

Vue3 특징

3. Typescript 지원

Vue2에서 TypeScript를 사용할 때 타입 감지가 안 된다는 이슈가 있었습니다. 타입스크립트의 타입 추론 방식은 타입을 명시적으로 선언하지 않아도 추론이 가능해야 하는데 객체 구조 특성상 개발자가 일일이 타입을 정의해 줘야 하는 상황이 많았기 때문입니다.

Vue3에서 TypeScript 기반으로 완전히 다시 작성 되어 TypeScript 지원이 간편 해지고, Composition API 내부의 setup() 함수에서 자동으로 타입을 추론하기 때문에 사용하기가 훨씬 수월 해졌다.
Vuex, Vue-Router 에서도 TypeScript 사용이 훨씬 더 편리 해졌습니다.

Vue3 특징

4. SFC (Single-File Component)

SFC는 Vue가 권장 하는 Vue 컴포넌트 전용 파일 포맷입니다. 단일 파일 컴포넌트를 말하며 하나의 파일이 하나의 컴포넌트가 된다 라는 의미입니다.

HTML, CSS, JavaScript를 <template>, <script>, <style> 블럭으로 캡슐화 하여 하나의 .vue 파일 내에 정의됩니다. 이러한 패턴 덕분에 웹 표준이 잘 지켜지고 있으며, 코드의 가시성과 생산성이 좋다고 평가됩니다.

5. Template 문법

Vue는 브라우저 화면에 렌더링 하는 과정에서 템플릿이란 문법을 사용합니다. Vue 템플릿은 HTML, CSS 등의 마크업 속성과 데이터 및 로직 들을 연결해 브라우저에서 볼 수 있는 HTML 형태로 변환해 줍니다.

6. Vue-CLI 사용

Vue-CLI는 기본 Vue 개발 환경을 설정해 주는 도구로, 자동으로 Vue 프로젝트를 생성합니다. Babel과 Webpack 그리고 라이브러리까지 관리하는 프로젝트를 쉽게 구축 할 수 있습니다. 기본적인 프로젝트 세팅을 해준다. (폴더 구조, Lint와 build를 위한 라이브러리로 구성, Webpack 설정)

Vue3 특징

7. Vue.JS의 Component

- Vue.JS는 단일 파일 컴포넌트를 추천 합니다.
- 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 개발 할 수 있는 구조입니다.

Vue3 특징

Vue 3 (https://github.com/vuejs/core)

: Vue2 에서 Vue3 버전으로 마이그레이션 가이드 (https://v3-migration.vuejs.org/)

Vue 3의 간략한 특징 (참고 : https://increment.com/frontend/making-vue-3/)

- 코드베이스를 TypeScript로 전환 및 재개발
- 새로운 렌더링 전략 (Virtual DOM 로직 변경)
- 더 나은 mount, update, memory 성능
- 트리쉐이킹 기법으로 사용하지 않는 코드를 제거함으로서 최종 번들 용량을 줄임
- 컴포넌트의 로직을 유연하게 구성할 수 있도록 하는 함수 기반의 Composition API 지원
- Vue2 에서도 Plugin을 설치하여 Composition API을 사용할 수 있다. 21.04.12일 기준으로 v1.0.0-rc.6버전이 pre-release 되어 있다. (https://github.com/vuejs/composition-api)

Vue3 특징

- Vue 인스턴스 생성 방법 변경

  Vue2 : new 생성자 사용

  Vue3 : createApp() 함수 사용

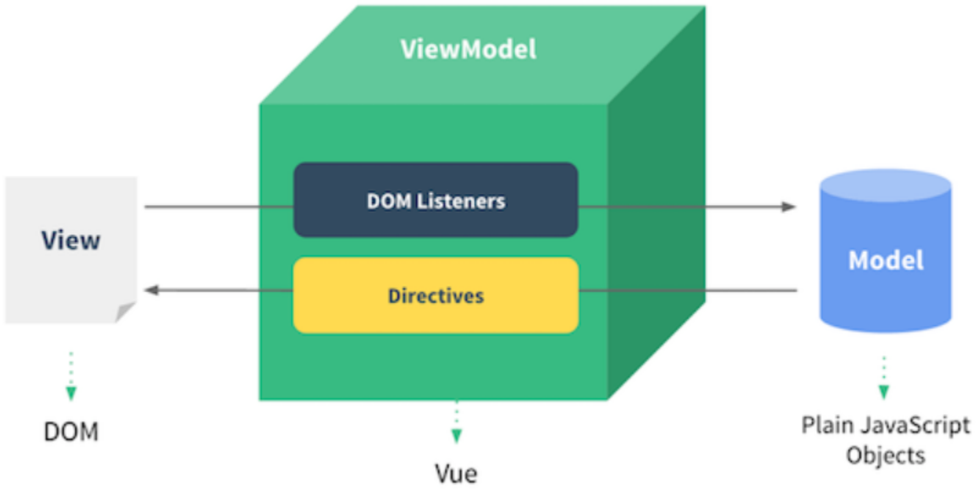
  Vue 인스턴스의 설정을 글로벌 범위 일괄 적용이 아닌 애플리케이션의 컨텍스트를 특정 인스턴스 각각에 구성 할 수 있도록 변경 되었다.

- LifeCycle Hook 이름 변경 (예) onBeforeMounted
- Event Bus 제거
- Filter 속성 제거
- emits 속성 추가

Vue 특징

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 특징

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 특징

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.

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 도출되었고 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 최상단 으로 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: Arrow Function

- 4.1 Arrow 함수와 기존 함수의 차이점
- Arrow 함수와 기존 함수는 참조하고 있는 this의 값이 다릅니다.

기존 함수

```
function BlackDog() {
   this.name = '흰둥이';
   return {
       name:'검둥이',
       bark: function() {
         console.log(this.name + ' 멍멍!');
       }
   }
}

const blackDog = new BlackDog();
blackDog.bark(); // 검둥이 멍멍!
```

arrow 함수

```
function WhiteDog() {
   this.name = '흰둥이';
   return {
       name:'검둥이',
       bark: () => {
         console.log(this.name + ' 멍멍!');
       }
   }
}

const whiteDog = new WhiteDog();
whiteDog.bark(); // 흰둥이 멍멍!
```

[mdn: Arrow function expression]

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;
```

값 교환 (Swapping)

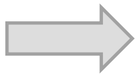
```
let a = 1;
let b = 3;

[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

- 9.Object의 비구조화 할당 (Destructuring Assignment)

Object Destructuring Assignment

```
let person = {
  name:'React',
  addr:{home:'Seoul', office:'Gyeonggi'}
};

let {name,addr} = person;
console.log(name);
console.log(addr);
let {home,office} = addr;
console.log(home);
console.log(office);
```

축약

```
let {name,addr:{home,office}} = person;

console.log(name);
console.log(home);
console.log(office);
```

Object Destructuring Assignment

```
let person = {
  name:'React',
  addr:{home:'Seoul', office:'Gyeonggi'},
  phone:{mobile:{
    phone1:'010-1234',
    phone2:'010-5678'
  }}
};
```

축약

```
let {name,
     addr:{home,office},
     phone:{mobile:{phone1,phone2}}} = person;

console.log(name);
console.log(home);
console.log(office);
console.log(phone1);
console.log(phone2);
```

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

TypeScript

http://www.typescriptlang.org

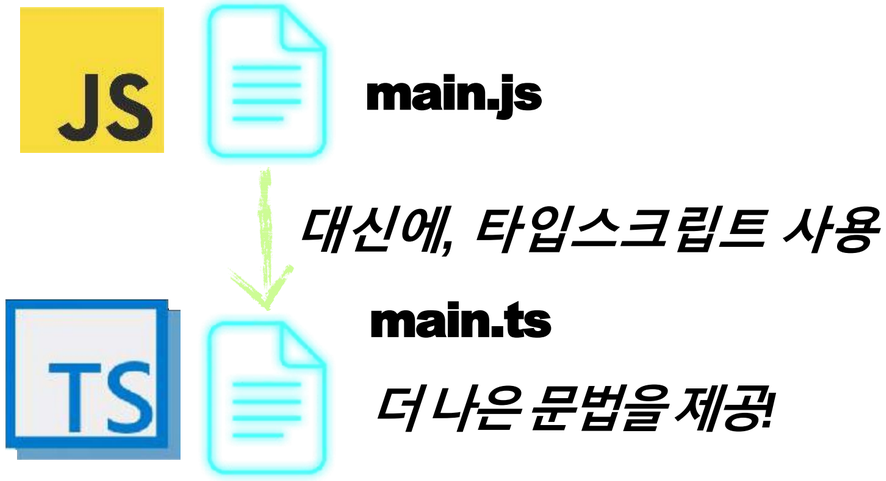
백명숙

TypeScript

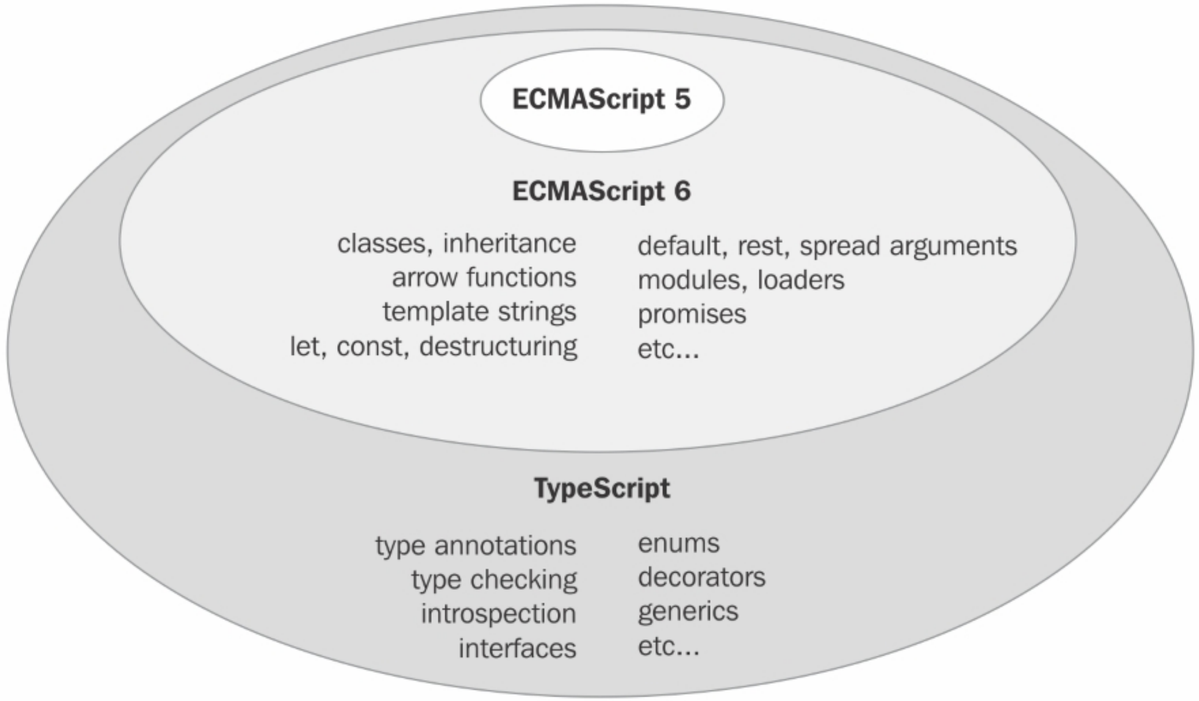
- https://www.typescriptlang.org/
- 타입스크립트 핸드북(https://www.typescriptlang.org/docs/handbook/intro.html)
- Typescript Github (https://github.com/microsoft/TypeScript)
- Stackoverflow 2022 선호하는 Language
- 2012년 10월 첫 타입스크립트 버전 0.8 발표

TypeScript의 개요

- TypeScript 또한 자바스크립트 대체 언어의 하나로써 자바스크립트(ES5)의 Superset(상위 확장)이다.
- C#의 창시자인 덴마크 출신 소프트웨어 엔지니어 Anders Hejlsberg가 개발을 주도한 TypeScript는 Microsoft에서 2012년 발표한 오픈소스로, 정적 타이핑을 지원하며 ES6(ECMAScript 2015)의 클래스, 모듈 등과 ES7의 Decorator 등을 지원한다.



http://www.typescriptlang.org



TypeScript의 특징

- 컴파일 언어, 정적 타입 언어이다. JS는 인터프리터 언어지만, TypeScript는 컴파일 언어로 코드 수준에서 미리 타입을 체크하여 오류를 체크한다. 단 전통적인 컴파일 언어와는 다르게, 링킹 (Linking) 과정이 생략되어 있다.

- 타입 스크립트의 기능
  - ✓ 타입 스크립트 = JavaScript + Type
  - ✓ 크로스 플랫폼 지원 : 자바스크립트가 실행되는 모든 플랫폼에서 사용 할 수 있습니다.
  - ✓ 객체 지향 언어: 클래스, 인터페이스, 모듈 등의 강력한 기능을 제공하며, 순수한 객체 지향 코드를 작성 할 수 있습니다.
  - ✓ 정적 타입: 정적 타입을 사용하기 때문에 코드를 입력하는 동안에 오류를 체크 할 수 있습니다.
  - ✓ DOM 제어: 자바스크립트와 같이 DOM을 제어하여 요소를 추가하거나 삭제 할 수 있습니다.
  - ✓ 최신 ECMAScript 기능 지원 : ES6 이상의 최신 자바스크립트 문법을 손쉽게 지원 할 수 있습니다.

TypeScript의 차별점

- 명시적인 자료형 선언가능

TS

```
function add(){
    let a: number = 10;
    let b: number = 10;
    let sum: number = a + b;
    console.log(sum);
}

add();
```

결과 : 20

- 명시적인 자료형 선언으로 가독성이 향상됨
- 자료형을 명시적으로 정의함으로써 오류를 사전에 감지함

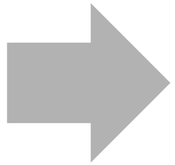
예 : 자료형이 다르면 비교나 할당이 불가능함

TypeScript의 차별점

- 객체지향 프로그래밍 지원

TS

```
class car {
   numTier: number;
   constructor(){}
   getNumTier(){}
   ….
}
```

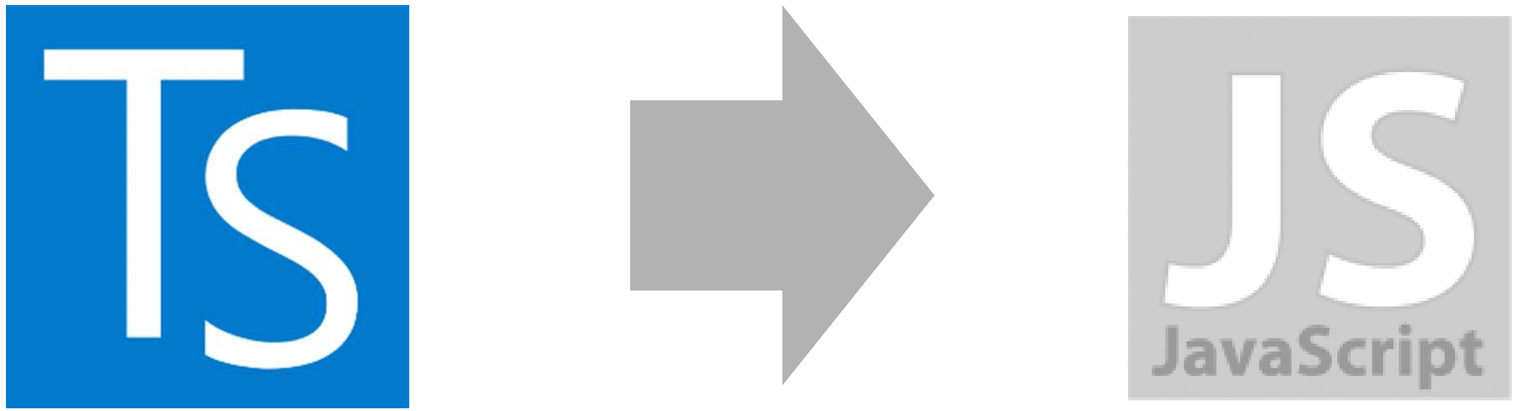


JS

```
var car = (function(){({
  function car(){};
car.prototype.getNumTier=function(){};
…
}());
```

트랜스파일러 : tsc

- TSC는 타입스크립트를 자바스크립트로 변환(transpiring) 해주는 도구이다.

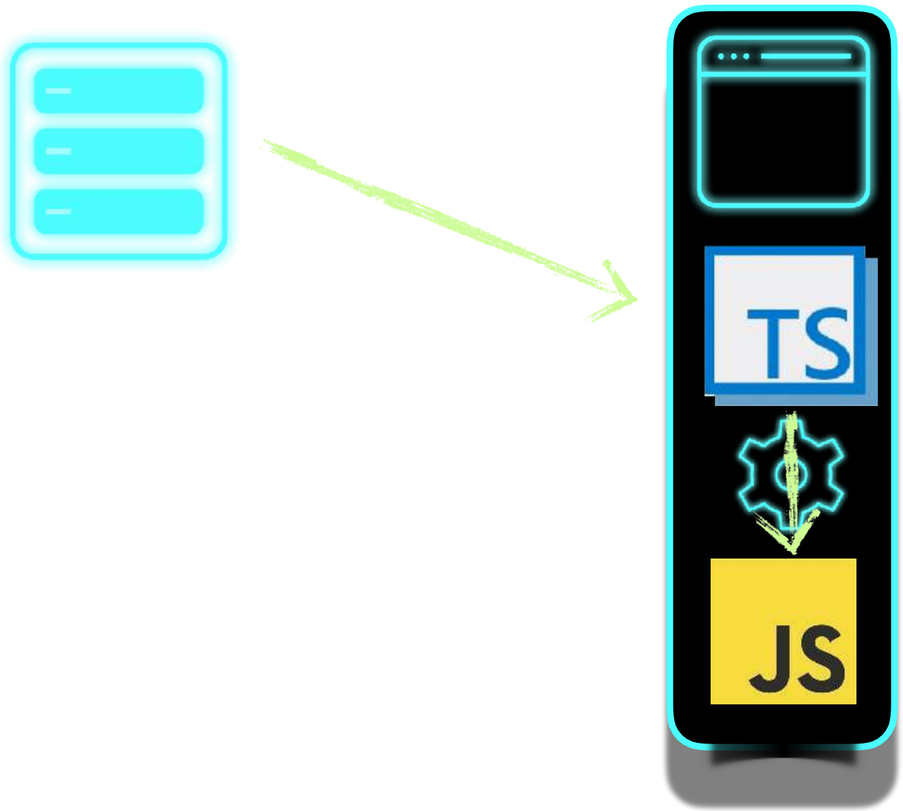


트랜스파일링

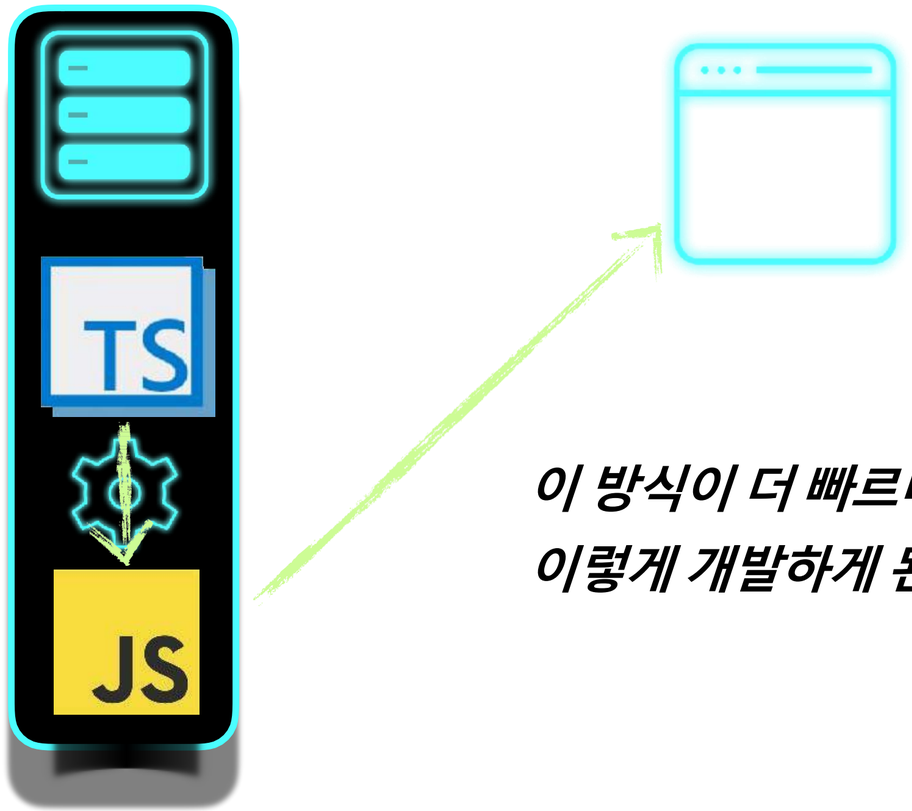
TypeScript : 트랜스파일 위치

브라우저는 타입스크립트를 해석할 수 없으며, 자바스크립트로 변환하여 브라우저에서 처리되어야 한다. 다음 두 가지 방식이 사용되고 있다.

브라우저에서 자바스크립트로 변환



자바스크립트로 변환 후 브라우저로 로딩



이 방식이 더 빠르며, 주로 이렇게 개발하게 된다.

트랜스파일러 : ts vs js

- TypeScript vs JavaScript

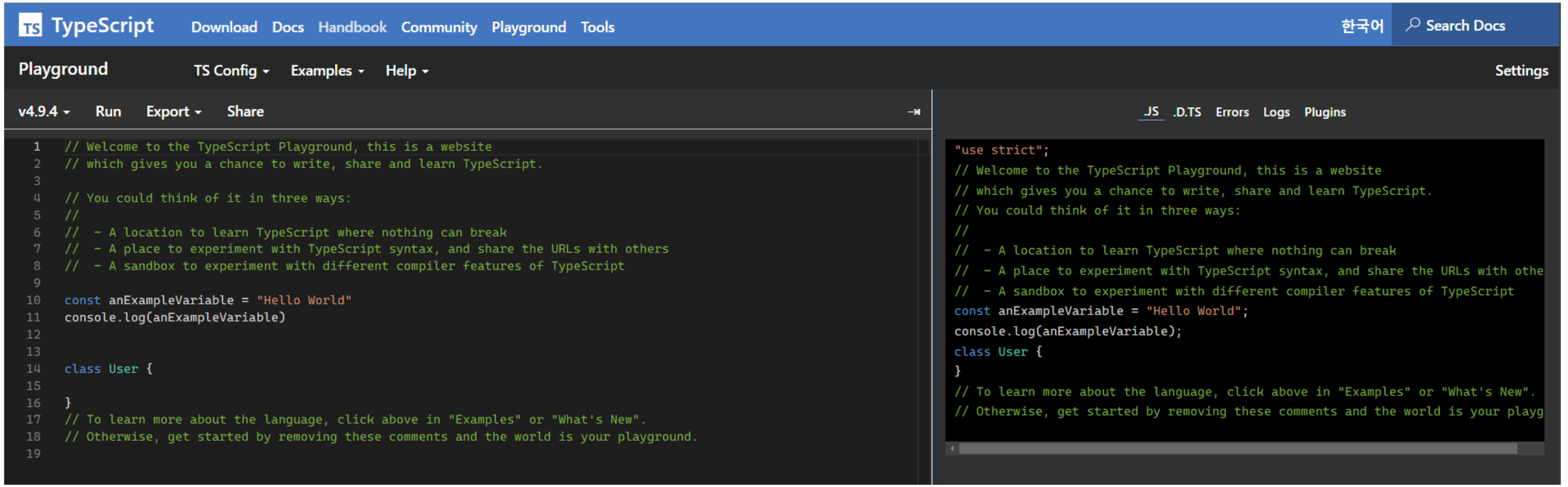
Typescript is a statically typed language
: types are checked at compile time

Javascript is a dynamically typed language
: types are checked at run time

Benefits include
: catch errors in dev!
: Great for teams

TypeScript : Playground

- https://www.typescriptlang.org/play



TypeScript : Download

Get TypeScript

Node.js

The command-line TypeScript compiler can be installed as a Node.js package.

INSTALL

```
npm install -g typescript
```

COMPILE

```
tsc helloworld.ts
```

Visual Studio


Visual Studio 2017


Visual Studio 2015


Visual Studio Code

And More...


Sublime Text


Emacs


Atom

WebStorm


Eclipse

Vim

TypeScript : Download

- 타입스크립트 설치

```
$ npm install -g typescript
$ tsc -v
$ tsc ./src/index.ts
```

- 단일 프로젝트에서 만 사용하길 희망하는 경우 현재 디렉토리에만 설치 후 npx tsc 명령으로 실행할 수도 있습니다.

```
$ npm install -D typescript
$ npx tsc --version
$ npx tsc ./src/index.ts
```

TypeScript : tsconfig.json

- 컴파일러 옵션

타입스크립트 컴파일을 위한 다양한 옵션을 지정할 수 있습니다.

공식 문서: https://www.typescriptlang.org/docs/handbook/compiler-options.html

```
$ tsc ./src/index.ts --watch --strict true --target ES6 --lib ES2015,DOM --module CommonJS
```

아래와 같이 tsconfig.json 파일로 옵션을 설정 할 수 있습니다.

"include"와 "exclude" 옵션을 같이 추가하여, 컴파일에 포함할 경로와 제외할 경로를 설정할 수 있습니다.

tsconfig.json

```
{
  "compilerOptions": {
    "strict": true,
    "target": "ES6",
    "lib": ["ES2015", "DOM"],
    "module": "CommonJS"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

```
$ tsc --watch
```

TypeScript : tsconfig.json

- tsconfig.json 자동생성

```
$ tsc --init  또는  $ npx tsc --init
```

옵션명	설명
target	타입 스크립트를 컴파일하여 변환되는 자바스크립트의 버전을 설정한다. **(기본값 ES5)**
module	코드가 동작할 대상 플랫폼에 맞는 모듈 시스템을 설정한다. 웹브라우저라면 amd, NodeJS라면 commonjs로 설정한다.
downlevelIteration	target 키에 설정된 자바스크립트 버전이 낮더라도 ES6의 문법을 사용하고자 할 때 설정한다. 예를 들어, 반복기나 생성기를 사용하려면 반드시 true로 설정해야 한다.
strict	엄격한 타입 검사를 수행한다.
noImplicitAny	Any 타입을 사용할 수 없도록 한다.
moduleResolution	모듈의 해석 방식을 설정한다. module 키가 commonjs라면 반드시 'node'로 설정해야 한다.
baseUrl	타입 스크립트를 컴파일하여 변환된 자바스크립트 파일을 저장하는 디렉터리를 설정한다. baseUrl은 그 기준이 되는 디렉터리로 보통 tsconfig.json 파일이 있는 현재 디렉터리로 설정한다.
outDir	baseUrl 키에 설정된 경로의 하위 디렉터리를 설정한다.
paths	소스 파일에서 import로 모듈을 포함할 때의 경로를 설정한다. import로 포함하는 모듈이 node_modules일 경우 node_modules/*를 넣어준다.
esModuleInterop	웹브라우저에서 동작을 목표로 만들어진 모듈을 NodeJS 환경에서 사용할 경우 활성화한다.

TypeScript : tsconfig.json

- tsc watch 기능

```
$ tsc -w
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "downlevelIteration": true,
    "strict": true,
    "noImplicitAny": true,
    "moduleResolution": "node",
    "baseUrl": ".",
    "outDir": "dist",
    "paths": {"*": ["node_modules/*"]},
    "esModuleInterop": true
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

TS Node

- TS Node (https://github.com/TypeStrong/ts-node)

- Node.js에서 Typescript를 실행 시키는 도구
- ts 파일을 미리 컴파일 하지 않고 바로 실행 할 수 있는 엔진
- JIT(Just In Time) 으로 Typescript를 Javascript로 변환 하여 실행 가능

```
# 전역
$ npm install -g ts-node
# 지역
$ npm install ts-node –D
$ ts-node –v
```

지역(Local) 설치 하면 npm scripts를 통해 사용 가능 (or npx 사용)

package.json

```
{
  "scripts": {
    “start": "ts-node server.ts",
  },
}
```

@types : DefinitelyTyped

- @types (https://github.com/DefinitelyTyped/DefinitelyTyped)

- TypeScript로 작성되지 않은 라이브러리를 import 해야 할 때도 있다.
- 만약에 styled-component를 import 하려면, 아래 명령어를 사용한다.

  $ npm i --save-dev @types/styled-components
- @types란 DefinitelyTyped라는 github Repository이며 대부분의 npm 라이브러리들을 가지고 있고 TypeScript로 작업 할 때 필요한 대부분의 라이브러리나 패키지의 type definition을 얻을 수 있다.

- @types/node는 Node.js의 type 정의를 포함하고 있는 패키지이다.

  @types/node 설치하기

  $ npm i --save-dev @types/node

Nodejs + Express

- Typescript로 Nodejs와 Express 설정하기

package.json 파일을 생성하고 의존성 설치하기

$ npm init

$ npm i express cors

$ npm i -D typescript @types/node @types/express @types/cors @types/body-parser nodemon ts-node

- express : NodeJS를 사용하여 서버를 개발 할 때 서버를 쉽게 구성할 수 있게 만든 프레임워크
- cors : cross origin resource sharing 지원
- @types/node : type definitions for Node.js
- @types/express : express는 원래 자바스크립트로 만들어졌다. TS 환경에서 사용하기 위해 express 내부의 변수들, 함수들의 타입을 정의한 ~d.ts 파일이 포함된 형태
- @types/cors : type definitions for cors
- @types/body-parser : type definitions for body-parser
- nodemon: 노드 서버를 실행하고 소스코드를 수정하면 다시 서버 재시작을 자동으로 해주는 모듈
- ts-node: Node.js용 TS 실행 엔진 및 REPL(Node.js 상에서 Typescript Compile 하지 않고, ts를 실행하는 역할을 한다.)

TypeScript 구성요소

Strongly Typed

Classes

Interfaces

Generics

Modules

Type Definitions

Compiles to JavaScript

EcmaScript 6 Features

TypeScript : 타입 시스템

- **Boolean**
- **Number**
- **String**
- **Array**
- **Tuple**
- **Enum**
- **Any**
- **Unknown**
- **Void**
- **Union**

타입 선언

- Boolean

: 참(true)/거짓(false) 값을 나타냅니다.

```
let isBoolean: boolean;
let isDone: boolean = false;
```

- Number

모든 부동 소수점 값을 사용할 수 있으며, ES6에 도입된 2진수 및 8진수 리터럴도 지원합니다.

```
let num: number;
let integer: number = 6;
let float: number = 3.14;
let hex: number = 0xf00d; // 61453
let binary: number = 0b1010; // 10
let octal: number = 0o744; // 484
let nan: number = NaN;
```

- String

문자열을 나타내며 작은따옴표('), 큰따옴표(") 뿐만 아니라 ES6의 템플릿 문자열도 지원합니다.

```
let str: string;
let red: string = 'Red';
let green: string = "Green";
let myColor: string = `My color is ${red}.`;
let yourColor: string = 'Your color is' + green;
```

타입 선언

- Array

배열은 두 가지 방법으로 타입을 선언할 수 있습니다.

```
// 문자열 배열
let fruits: string[] = ['Apple', 'Banana', 'Mango'];
let fruits: Array<string> = ['Apple', 'Banana', 'Mango'];

// 숫자 배열
let oneToSeven: number[] = [1, 2, 3, 4, 5, 6, 7];
let oneToSeven: Array<number> = [1, 2, 3, 4, 5, 6, 7];
```

유니언 타입(다중 타입)의 '문자열과 숫자를 동시에 가지는 배열' 도 선언할 수 있습니다.

```
let array: (string | number)[] = ['Apple', 1, 2, 'Banana', 'Mango', 3];
let array: Array<string | number> = ['Apple', 1, 2, 'Banana', 'Mango', 3];
```

읽기 전용 배열을 생성할 수도 있습니다.
readonly 키워드나 ReadonlyArray 타입을 사용합니다.

```
let arrA: readonly number[] = [1, 2, 3, 4];
let arrB: ReadonlyArray<number> = [0, 9, 8, 7];
arrA[0] = 123; // Error - TS2542: Index signature in type 'readonly number[]' only permits reading.
arrA.push(123); // Error - TS2339: Property 'push' does not exist on type 'readonly number[]'.
```

타입 선언

- Tuple

Tuple 타입은 배열과 매우 유사하며, 차이점은 정해진 타입의 고정된 길이(length) 배열을 표현합니다.

```
let tuple: [string, number];
tuple = ['a', 1];
tuple = ['a', 1, 2]; // Error - TS2322
tuple = [1, 'a']; // Error - TS2322
```

개별 변수로 지정하지 않고, 단일 Tuple 타입으로 지정해서 사용할 수 있습니다.

```
let userId: number = 1234;
let userName: string = '타입스크립트';
let isValid: boolean = true;

let user: [number, string, boolean] = [1234, '타입스크립트', true];
console.log(user[0]); // 1234
console.log(user[1]); // 'HEROPY'
console.log(user[2]); // true
```

readonly 키워드를 사용해서 읽기 전용 Tuple을 생성 할 수도 있습니다.

```
let a: readonly [string, number] = ['Hello', 123];
a[0] = 'World'; // Error - TS2540: Cannot assign to '0' because it is a read-only property.
```

타입 선언

- Enum

Enum은 숫자 혹은 문자열 값 집합에 이름(Member)을 부여할 수 있는 타입으로, 값의 종류가 일정한 범위로 정해져 있는 경우 유용 합니다. 기본적으로 0 부터 시작하며 값은 1 씩 증가합니다.

```
enum Week {
  Sun, Mon, Tue,  Wed,  Thu,  Fri,  Sat
}
console.log(Week.Sun); // 0
console.log(Week['Sun']); // 0
console.log(Week[0]); // 'Sun'
```

- Any

Any는 모든 타입을 의미하며 일반적인 자바스크립트 변수와 동일하게 어떤 타입의 값도 할당 할 수 있습니다.
외부 자원을 활용하여 개발 할 때 불가피 하게 타입을 단언할 수 없는 경우, 유용할 수 있습니다.

```
let any: any = 123;
any = 'Hello world';
any = {};
any = null;
```

타입 선언

- Unknown

Any와 같이 최상위 타입인 Unknown은 알 수 없는 타입을 의미합니다.

Any와 같이 Unknown에는 어떤 타입의 값도 할당할 수 있지만, Unknown을 다른 타입에는 할당할 수 없습니다.

```
let a: any = 123;
let u: unknown = 123;

let v1: boolean = a; // 모든 타입(any)은 어디든 할당할 수 있습니다.
let v2: number = u; // 알 수 없는 타입(unknown)은 모든 타입(any)을 제외한 다른 타입에 할당할 수 없습니다.
let v3: any = u; // OK!
let v4: number = u as number; // 타입을 단언하면 할당할 수 있습니다.
```

- Void

Void는 일반적으로 값을 반환하지 않는 함수에서 사용합니다.

: void 위치는 함수가 반환 타입을 명시하는 곳입니다.

```
function hello(msg: string): void {
  console.log(`Hello ${msg}`);
}
const hi: void = hello('world'); // Hello world
console.log(hi); // undefined
```

타입 선언

- Union

2개 이상의 타입을 허용하는 경우, 이를 유니온(Union)이라고 합니다.

|(vertical bar)를 통해 타입을 구분하며, ()는 선택 사항입니다.

```
let union: (string | number);
union = 'Hello type!';
union = 123;
union = false; // Error - TS2322: Type 'false' is not assignable to type 'string | number'.
```

- Function

화살표 함수를 이용해 타입을 지정할 수 있습니다.

인수의 타입과 반환 값의 타입을 입력 합니다.

```
// myFunc는 2개의 숫자 타입 인수를 가지고, 숫자 타입을 반환하는 함수.
let myFunc: (arg1: number, arg2: number) => number;
myFunc = function (x, y) {
  return x + y;
};

let assignClass: (n:string) => void;
assignClass = function(name) {
  document.documentElement.classList.add(name);
};
```

타입 선언

- 사용자 정의 타입 (Type Aliases)

타입 재사용을 위한 기능이 타입 별칭(Type Aliase) 이다.

```
type operation = {
  data: number[],
  output: (num:number) => number[]
};

// 사용자 정의 타입 operation 적용 예시
let sum: operation = {
  data: [10, 30, 60],
  output(num){
    return this.data.map(n => n+num);
  }
};

let multiply: operation = {
  data: [110, 230, 870, 231],
  output(num){
    return this.data.map(n => n*num);
  }
};
```

타입 선언

- Never 타입

never 타입은 모든 타입에 할당 가능한 하위 타입이나, never 타입에는 다른 타입을 할당 할 수 없다.

never 타입은 절대 발생할 수 없는 타입을 나타낸다.

```
let foo: never = 123; // Error: Type number is not assignable to never

let foo: never;
let goo: any = 123;
foo = goo; // Error never타입에는 any타입 조차 할당될 수 없음
```

```
function foo2(x: string | number): boolean {
  if (typeof x === "string") {
    return true;
  } else if (typeof x === "number") {
    return false;
  }
  return fail("Unexhaustive!");
}
function fail(message: string): void { throw new Error(message); }
//function fail(message: string): never { throw new Error(message); }

let tmp: any = {id:"123"}
console.log(foo2(tmp))
```

foo2()함수는 boolean타입을 반환해야 하지만 fail()함수는 void형이기 때문에 foo2()함수에서 사용 될 수 없다.

fail()함수를 never타입으로 선언하면 사용이 가능하다.

타입 추론(Inference)

- 타입 추론

명시적으로 타입 선언이 되어 있지 않은 경우, 타입스크립트는 타입을 추론하여 제공합니다.

```
let num = 12;
num = 'Hello type!'; // TS2322: Type '"Hello type!"' is not assignable to type 'number'.
```

변수 num을 초기화 하면서 숫자 12를 할당해 Number 타입으로 추론 되었기 때문에 'Hello type!'이라는 String 타입의 값은 할당 할 수 없다는 에러가 발생합니다.

- 타입을 추론하는 경우

1) 초기화 된 변수

2) 기본값이 설정된 매개 변수

3) 반환 값이 있는 함수

```
// 초기화된 변수 `num`
let num = 12;

// 기본값이 설정된 매개 변수 `b`
function add(a: number, b: number = 2): number {
  // 반환 값(`a + b`)이 있는 함수
  return a + b;
}
```

타입 단언(Assertions)

- 타입 단언

타입스크립트가 타입 추론을 통해 판단 할 수 있는 타입의 범주를 넘는 경우, 더 이상 추론하지 않도록 지시 할 수 있음

```
let num = 12;
function someFunc(val: string | number, isNumber: boolean) {
  // some logics
  if (isNumber) {
    // 1. 변수 as 타입
    (val as number).toFixed(2);
    // 또는
    // 2. <타입>변수
    (<number>val).toFixed(2);
  }
}
```

특히 컴파일 환경에서 체크하기 어려운 DOM을 사용 할 때 유용합니다.

```
// Error - TS2531: Object is possibly 'null'.
document.querySelector('.menu-item').innerHTML;

// Type assertion
(document.querySelector('.menu-item') as HTMLDivElement).innerHTML;
(<HTMLDivElement>document.querySelector('.menu-item')).innerHTML;

// Non-null assertion operator
document.querySelector('.menu-item')!.innerHTML;
```

인터페이스

인터페이스(Interface)는 타입스크립트 여러 객체를 위한 타입을 지정 할 때 사용하는 구조입니다.

```
interface IUser {
  name: string,
  age: number,
  isAdult?: boolean // Optional property
}

// `isAdult`를 초기화 하지 않아도 에러가 발생 하지 않습니다.
let user: IUser = {
  name: 'Neo',
  age: 123
};
```

StudentInfo 인터페이스를 작성하여 getStudentMoreInfo() 함수 반환 값의 타입으로 적용한다.
인터페이스는 해당 타입으로 반환되는 것을 강제한다.

```
interface StudentInfo {
  readonly studentId: number,
  studentName: string,
  age: number,
  subject?: string,
  graduated: boolean,
}
```

```
function getStudentMoreInfo(studentId: number): StudentInfo
{
  return {
    studentId: studentId,
    studentName: '타입스크립트',
    age: 29,
    graduated: false,
  };
}
```

인터페이스

함수 타입 인터페이스(Interface)

: 중괄호 안에 (매개변수 타입): 반환 값 타입; 형태로 만든다.

```
// Before: 함수 타입 설정
const factorial = (n:number): number => {
  if (n === 0) { return 0; }
  if (n === 1) { return 1; }
  return n * factorial(n - 1);
}

// After: 함수 타입 인터페이스
interface FactorialInterface {
  (n: number): number;
}

const facto: FactorialInterface = (n) => {
  if (n === 0) { return 0; }
  if (n === 1) { return 1; }
  return n * facto(n - 1);
};
```

인터페이스와 클래스

클래스의 생성자 메소드(constructor)와 일반 메소드(Methods) 멤버(Class member)와는 다르게,
속성(Properties)은 name: string와 같이 클래스 바디(Class body)에 별도로 타입을 선언합니다.

```
interface Shape {
  getArea(): number;
}

class Circle implements Shape {
  constructor(public radius: number) {
    this.radius = radius;
  }
  getArea() {
    return this.radius * this.radius * Math.PI;
  }
}

class Rectangle implements Shape {
  constructor(private width: number, private height: number) {
    this.width = width;
    this.height = height;
  }
  getArea() {
    return this.width * this.height;
  }
}
```

```
const circle = new Circle(5);
const rectangle = new Rectangle(10, 5);

console.log(circle.radius); // 에러 없이 작동
console.log(rectangle.width); // width가 private이므로 에러발생!

const shapes: Shape[] = [new Circle(5), new Rectangle(10, 5)];
shapes.forEach(shape => {
  console.log(shape.getArea());
});
```

제네릭(Generic)

- Generic

재사용을 목적으로 함수나 클래스의 선언 시점이 아닌, 사용 시점에 타입을 선언 할 수 있는 방법을 제공합니다.

```
function toArray(a: number | string, b: number | string): (number | string)[] {
  return [a, b];
}
toArray(1, 2); // Only Number
toArray('1', '2'); // Only String
toArray(1, '2'); // Number & String
```

```
function toArray<T>(a: T, b: T): T[] {
  return [a, b];
}

toArray<number>(1, 2);
toArray<string>('1', '2');
toArray<string | number>(1, '2');
toArray<number>(1, '2'); // Error
```

함수 이름 우측에 <T>를 작성해 선언 합니다.
T는 타입 변수(Type variable)로 사용자가 제공한 타입으로 변환하게 될 식별자 입니다.
세 번째 호출은 의도적으로 Number와 String 타입을 동시에 받을 수 있도록 했습니다.

인덱싱 가능 타입(Indexable types)

- Indexable Types

arr[2]와 같이 '숫자'로 인덱싱 하거나 obj['name']과 같이 '문자'로 인덱싱 하는, 인덱싱 가능 타입(Indexable types)들이 있습니다. 이런 인덱싱 가능 타입들을 정의하는 인터페이스는 인덱스 시그니처(Index signature)를 가질 수 있습니다. 인덱스 시그니처는 인덱싱에 사용 할 인덱서(Indexer)의 이름과 타입 그리고 인덱싱 결과의 반환 값을 지정합니다. 인덱서의 타입은 string과 number만 지정할 수 있습니다.

```
interface INAME {
  [INDEXER_NAME: INDEXER_TYPE]: RETURN_TYPE // Index signature
}
```

```
interface IUser {
  [userProp: string]: string | boolean
}
let user: IUser = {
  name: 'Neo',
  email: 'thesecon@gmail.com',
  isValid: true,
  0: false
};
console.log(user['name']); // 'Neo' is string.
console.log(user['email']); // 'thesecon@gmail.com' is string.
console.log(user['isValid']); // true is boolean.
console.log(user[0]); // false is boolean
console.log(user[1]); // undefined
console.log(user['0']); // false is boolean.
```

인덱싱 가능 타입(Indexable types)

- Indexable Type의 keyof

인덱싱 가능 타입에서 keyof를 사용하면 속성 이름을 타입으로 사용할 수 있습니다.

인덱싱 가능 타입의 속성 이름들은 유니온 타입으로 적용 됩니다.

```
interface ICountries {
  KR: '대한민국',
  US: '미국',
  CP: '중국'
}
let country: keyof ICountries; // 'KR' | 'US' | 'CP'
country = 'KR'; // ok
country = 'RU'; // Error - TS2322: Type '"RU"' is not assignable to type '"KR" | "US" | "CP"'.
```

keyof를 통한 인덱싱으로 타입의 개별 값에도 접근할 수 있습니다.

```
interface ICountries {
  KR: '대한민국',
  US: '미국',
  CP: '중국'
}
let country: ICountries[keyof ICountries]; // ICountries['KR' | 'US' | 'CP']
country = '대한민국';
country = '러시아'; // Error - TS2322: Type '"러시아"' is not assignable to type '"대한민국" | "미국" | "중국"'.
```

keyof 연산자

- keyof : 객체 형태의 타입을, 따로 속성들만 추출하여 Union 타입으로 만들어 주는 연산자

: keyof는 Object의 key들의 literal 값들을 가져온다.

: 만일 obj 객체의 key값인 red, yellow, green을 상수 타입으로 사용하고 싶을 때는 typeof obj 자체에 keyof 키워드를 붙여주면 된다.

```
type Type = {
   name: string;
   age: number;
   married: boolean;
}

type Union = keyof Type;
// type Union = name | age | married

const a:Union = 'name';
const b:Union = 'age';
const c:Union = 'married';
```

```
// 상수 타입을 구성하기 위해서는 타입 단언을 해준다.
const obj = { red: 'apple', yellow: 'banana', green: 'cucumber' } as
const;

// 위의 객체에서 red, yellow, green 부분만 꺼내와 타입으로서 사용하고 싶을 때
// 객체의 key들만 가져와 상수 타입으로
type Color = keyof typeof obj;

let ob2: Color = 'red';
let ob3: Color = 'yellow';
let ob4: Color = 'green';
```

typeof 연산자

- typeof : 객체 데이터를 객체 타입으로 변환 해주는 연산자

아래의 코드의 obj는 객체이기 때문에, 객체 자체를 타입으로 사용 할 수 없다.

객체에 쓰인 타입 구조를 그대로 가져와 독립된 타입으로 만들어 사용하고 싶다면, typeof 키워드를 명시 해주면 해당 객체의 타입 구조를 가져와서 사용 할 수 있다. 함수도 타입으로 변환 하여 재사용이 가능하다.

```
const obj = {
   red: 'apple',
   yellow: 'banana',
   green: 'cucumber',
};
// 위의 객체를 타입으로 변환하여 사용하고 싶을 때
type Fruit = typeof obj;
/*
type Fruit = {
    red: string;
    yellow: string;
    green: string;
}
*/
let obj2: Fruit = {
   red: 'pepper',
   yellow: 'orange',
   green: 'pinnut',
};
```

```
function fn(num: number, str: string): string {
   return num.toString();
}

type Fn_Type = typeof fn;
// type Fn_Type = (num: number, str: string) => string

const ff: Fn_Type = (num: number, str: string): string => {
   return str;
};
```

Intersection Type

- 인터섹션 타입(Intersection Type)은 여러 타입을 모두 만족하는 하나의 타입을 의미합니다.
- & 연산자를 이용해 여러 개의 타입 정의를 하나로 합치는 방식을 Intersection Type 정의 방식이라고 합니다.

```
interface Person {
  name: string;
  age: number;
}

interface Developer {
  name: string;
  skill: number;
}

type John = Person & Developer;
```

Person 인터페이스의 타입 정의와 Developer 인터페이스의 타입 정의를 & 연산자를 이용하여 합친 후 John 이라는 타입에 할당한 코드 입니다.

결과적으로 John의 타입은 아래와 같이 정의 됩니다.

```
{
  name: string;
  age: number;
  skill: string;
}
```

유틸리티 타입 **Partial<T>**

- Partial은 TypeScript에서 제공하는 타입 유틸리티 함수로, 주어진 타입의 모든 프로퍼티를 optional하게 만들어 주는 기능을 제공 합니다. 즉, 주어진 타입의 각 프로퍼티에 ?를 붙여서 각 프로퍼티를 optional 하게 만든 새로운 타입을 만들어 줍니다.

```
interface Person {
    name: string;
    age: number;
    address: string;
}

type PartialPerson = Partial<Person>;
```

```
let john: PartialPerson = {
    name: 'John Doe'
};
```

john 변수는 name 프로퍼티만 가지고 있어도 올바른 타입 입니다.

PartialPerson은 Person 타입의 name, age, address 프로퍼티를 각각 optional하게 만든 타입이 됩니다.

유틸리티 타입 **Required<T>**

- Required는 위의 Partial과 반대되는 개념이다. 제네릭 타입 T의 모든 프로퍼티에 대해 Required 속성으로 만들어준다. 즉, 모든 프로퍼티에서 Optional을 제거하는 역할을 합니다.

```
interface Comment {
  title: string;
  description?: string;
  like?: number;
}

type RequiredComment = Required<Comment>;
```

title은 필수 속성이고, description과 like는 선택 속성으로 정의되어 있지만, `Required<Comment>` 은 `title, description, like` 속성이 모두 필요한 타입이 됩니다.

```
const getData = () => {
  const commentData = db.get('comment') as Comment;
  return commentDatas;
}

const saveData = (comment: Required<Comment>) => {
  return db.save(comment)
}

// ts(2345) error : Type '{ title: string; }' is missing the following properties from
type 'Required<Comment>': description, like
const result = saveData({title: 'John'});
```

DB에 저장할 때는 모든 property를 저장해야 하는 상황이 생길 수 있습니다.

값을 저장할 때 모든 property 를 받았는지를 Required를 활용하면 손쉽게 검증 할 수 있습니다.

유틸리티 타입 **Readonly<T>**

- T 타입의 모든 프로퍼티를 readonly(읽기 전용)로 변환한 타입을 반환한다
- readonly 타입을 가진 값은 수정이 불가능 하므로, 해당 타입의 값을 변경하게 되면 에러가 발생한다.

```
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = {
  title: 'Delete inactive users',
};

todo.title = 'Hello';  // readonly 타입이므로 Error!
```

유틸리티 타입 **Record<T>**

- K를 key로, T를 type으로 하는 새로운 타입을 반환한다.
- 특정 타입만 키 또는 값으로 갖는 타입을 선언하고자 할 때 사용할 수 있다.

```
type CatName = "miffy" | "boris" | "mordred";

interface CatInfo {
  age: number;
  breed: string;
}

const cats: Record<CatName, CatInfo> = {
  miffy: { age: 10, breed: "Persian" },
  boris: { age: 5, breed: "Maine Coon" },
  mordred: { age: 16, breed: "British Shorthair" },
};
```

- K 타입으로 string, number, symbol을 사용할 경우에는 다음과 같이 선언해도 Record<K, T>로 선언한 타입과 동일하게 사용 할 수 있다.

```
const nameAgeMap: Record<string, number> = {
  'Alice': 21,
  'Bob': 25
};
// 위의 예제와 동일한 결과 타입을 가짐
const nameAgeMap: { [key: string]: number } = {
  'Alice': 21,
  'Bob': 25
};
```

유틸리티 타입 **Exclude<T,U> / Extract<T,U> / NonNullable<Type>**

- **Exclude<T, U>**

- T 타입에서 U 타입과 공통되는 프로퍼티를 제외한 나머지 프로퍼티를 타입으로 추출하여 반환한다.

```
type T0 = Exclude<"a" | "b" | "c", "a">;  // type T0 = "b" | "c“

type T1 = Exclude<"a" | "b" | "c", "a" | "b">;  // type T1 = "c“

type T2 = Exclude<string | number | (() => void), Function>;  // type T2 = string | number
```

- **Extract<T, U>**

- T 타입에서 U 타입과 공통되는 모든 프로퍼티를 타입으로 추출하여 반환한다.

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">;  // type T0 = "a“

type T1 = Extract<string | number | (() => void), Function>;  // type T1 = () => void
```

- **NonNullable<Type>**

- T 타입에서 Nullable 한 타입인 null, undefined를 제외한 타입을 반환한다.

```
type T0 = NonNullable<string | number | undefined>;  // type T0 = string | number

type T1 = NonNullable<string[] | null | undefined>;  // type T1 = string[]
```

유틸리티 타입 **Pick**과 **Omit**

- Pick : Pick 타입은 특정 타입에서 몇 개의 속성을 선택하여 타입을 정의 합니다.
- Omit : Omit 타입은 특정 속성만 제거한 타입을 정의 합니다.

```
interface Product {
  id: number;
  name: string;
  price: number;
  brand: string;
  stock: number;
}
```

```
type Product1 = Pick<Product, "id" | "name" | "price">;
const apple: Product1 = {
  id: 1,
  name: "MyApple",
  price: 1000
};
console.log(apple);
```

```
type Product2 = Omit<Product, "stock">;
const computer: Product2 = {
  id: 1,
  name: "MyCom",
  price: 100000,
  brand: "del"
};
console.log(computer);
```

유틸리티 타입 **Parameters**

- Parameters는 하나의 Function 타입을 인자로 받아서, 해당 Function 타입의 매개변수들의 Tuple 타입을 반환한다.
- 함수의 파라미터 타입과 인자로 활용 할 *튜플 *변수의 타입을 동기화 하는데 사용합니다.

```
const myFunction = (a: string, b: number) => {
    return a + b;
}

type myType = Parameters<typeof myFunction>
// 리턴타입은 튜플입니다.

// 함수의 파라미터 타입과 변수의 타입이 동기화 됩니다.
let myArray:myType = [ 'hello ', 100 ];

console.log(myFunction(...myArray));

type aType = Parameters<typeof myFunction>[0]
type bType = Parameters<typeof myFunction>[1]

let a:aType = 'hello2 '
let b:bType = 200

console.log(myFunction(a, b));
```

유틸리티 타입 **ReturnType**

- 함수의 리턴값의 타입을 활용 합니다.
- ReturnType 유틸리티 타입은 특정 함수의 출력을 다른 함수에서 가져 와야 하는 상황에서 매우 유용합니다.

```
function sendData(a: number, b: number) {
    return {
        a: `${a}`,
        b: `${b}`
    }
}

type Data = {
    a: string,
    b: string
}

function consoleData(data:Data) {
    console.log(JSON.stringify(data));
}

let stringifyNumbers = sendData(1, 2);
consoleData(stringifyNumbers);
```

만약 sendData의 출력 타입이 바뀌면, Data 타입도 변경해야 할 것입니다.

이런 경우를 막기 위해, 두 개의 타입을 동기화 해줍니다.

```
function sendData(a: number, b: number) {
    return {
        a: `${a}`,
        b: `${b}`
    }
}

type Data = ReturnType<typeof sendData>
// The same as writing:
// type Data = {
//     a: string,
//     b: string
// }
```

유틸리티 타입 **Awaited<T>**

- **Awaited<T>**
- Promise<?> 형태의 T 타입을 전달 받아, 해당 Promise가 반환하는 리턴값의 타입을 반환 한다.
- async ~ await의 await 키워드와 유사한 기능을 담당한다.

```
type A = Awaited<Promise<string>>;  // type A = string

type B = Awaited<Promise<Promise<number>>>;  // type B = number

type C = Awaited<boolean | Promise<number>>;  // type C = number | boolean
```

Mapped Type

- 타입스크립트의 고급 타입인 맵드 타입(mapped type)이란 기존에 정의되어 있는 타입을 새로운 타입으로 변환해 주는 문법을 의미 한다.
- Obj 라는 인터페이스의 객체 속성 타입 string을 ChangeType<Obj> 을 통해 Obj 타입들을 number로 모두 바꿔 주고 Result 타입 별칭에게 반환 하였다. 타입 Result는 { prop1: number; prop2; number } 와 같은 객체 타입을 가지게 되었다.

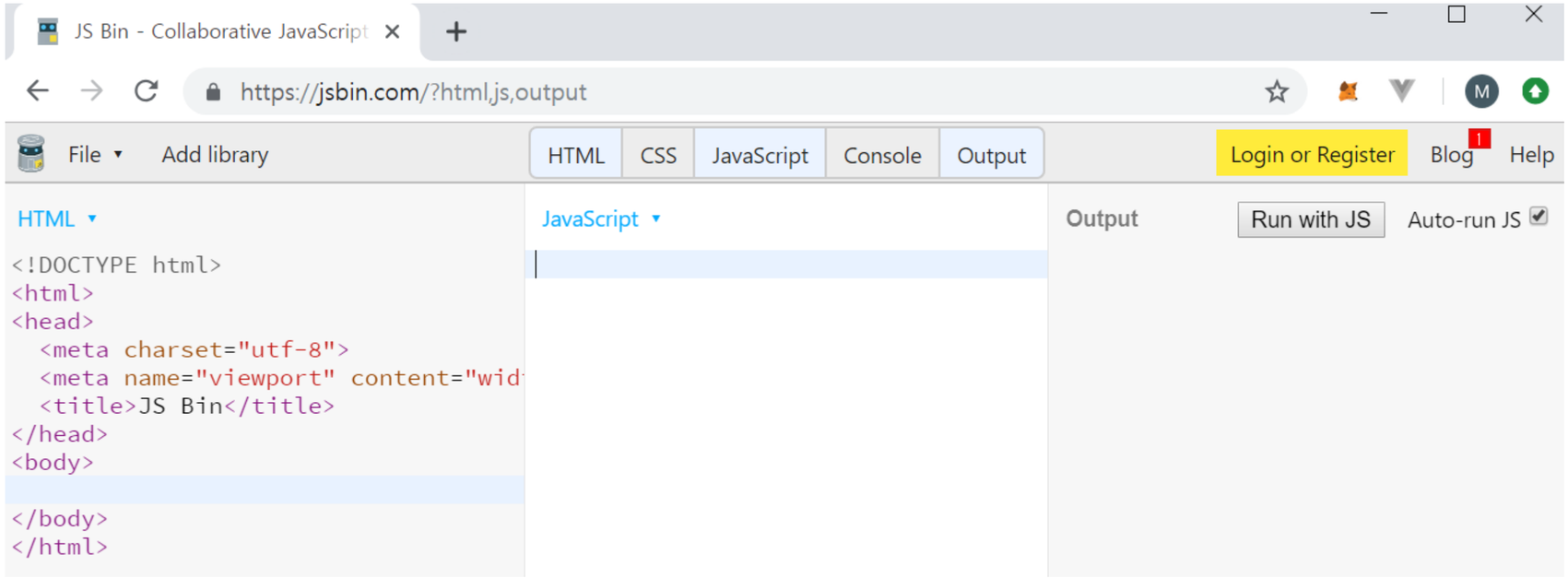
```
interface Obj {
    prop1: string;
    prop2: string;
}

type ChangeType<T> = {
    [K in keyof T]: number;
};

type Result = ChangeType<Obj>;
/*
{
    prop1: number;
    prop2: number;
}
*/
```

```
// index.ts
type Students = 'Dooly' | 'Gildong' | 'John'

type StudentAges = {[K in Students] : number}

const ages: StudentAges = {
  Dooly: 10,
  Gildong: 20,
  John: 15
}
console.log(ages);
```

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 https://unpkg.com/vue/dist/vue.js 링크를 사용하면 됩니다.

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 이렇게 작동해라! 하고 지시를 해주는 지시문 입니다.

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

v-else-if는 첫번째 조건문의 값이 true가 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>
```

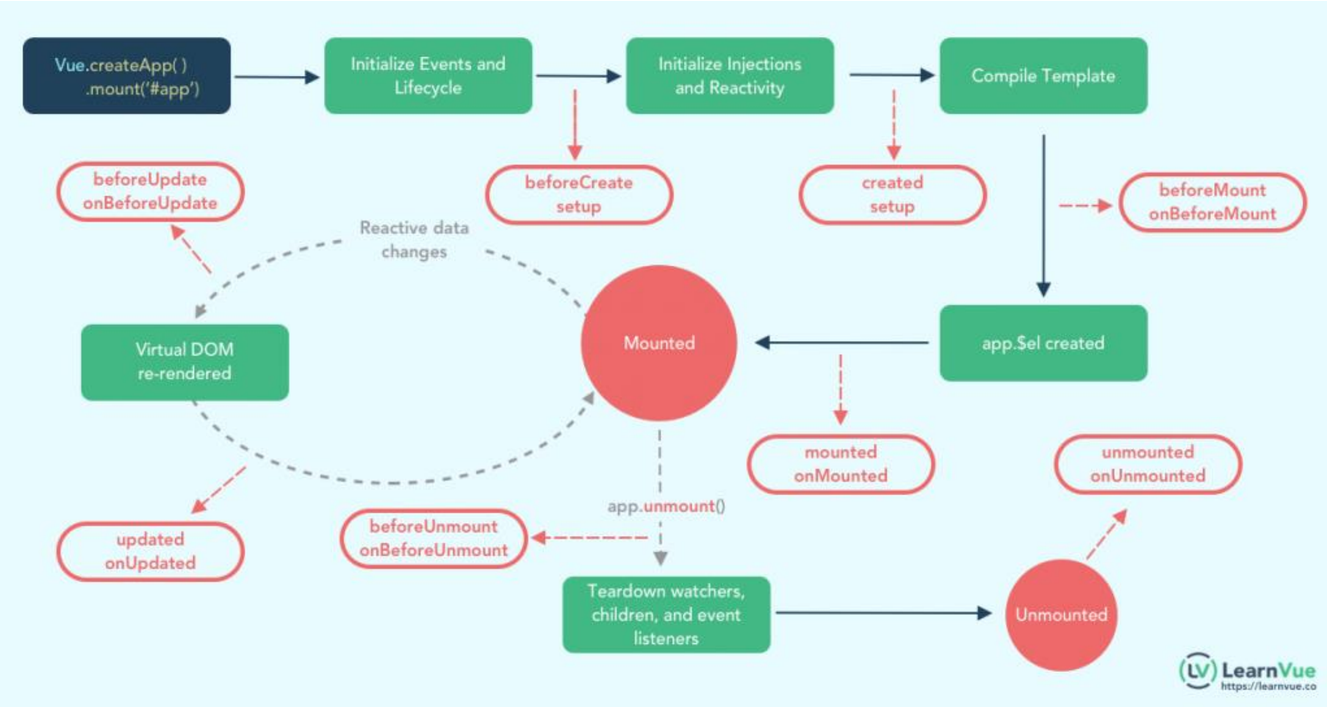
javascript

```
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 호출 할 수 있는 속성들을 라이프 사이클 속성이라고 합니다.
: 라이프 사이클 속성에는 beforeCreate 및 created(setup() 메서드 자체로 대체됨)를 제외하고 설정 메소드에서 액세스할 수 있는 옵션 API lifecycle hooks가 있다.



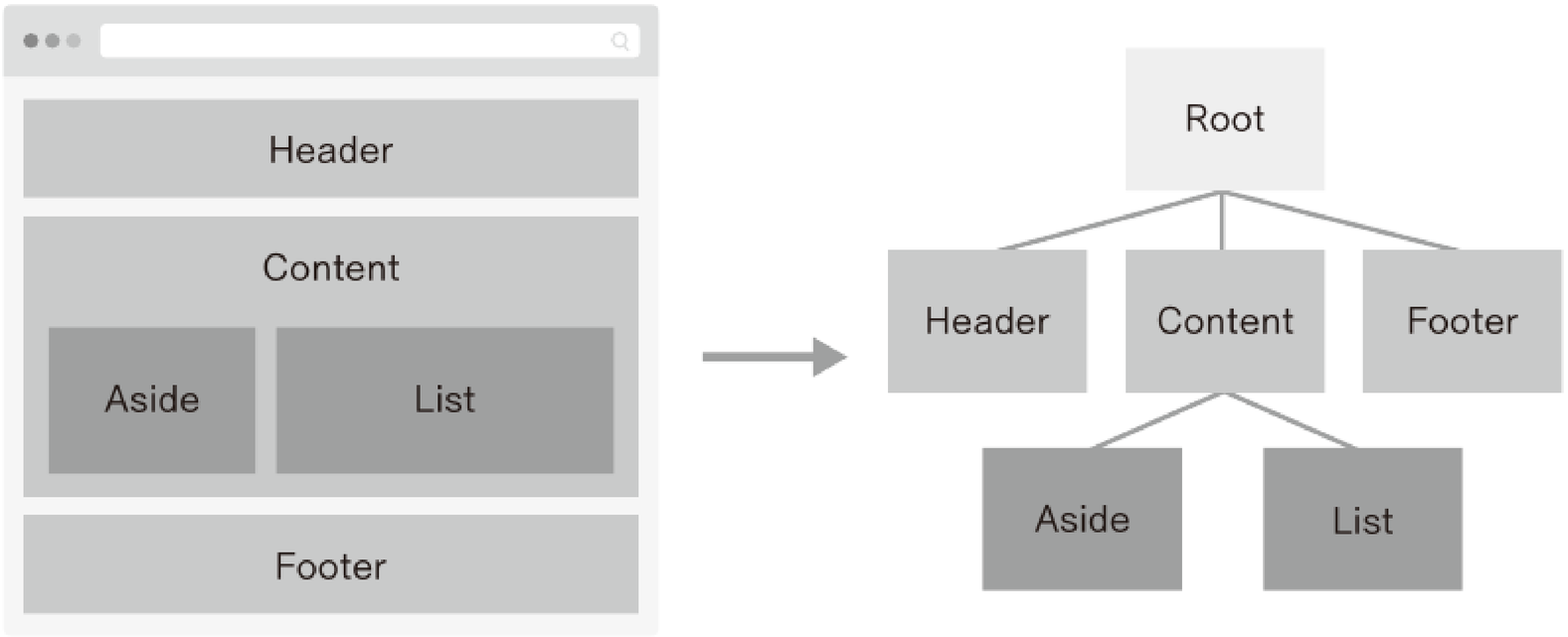
- Creation
  : runs on your component's creation
- Mounting
  : runs when the DOM is mounted
- Updates
  : runs when reactive data is modified
- Destruction
  : runs right before your element is destroyed.

https://vuejs.org/guide/essentials/lifecycle.html#lifecycle-diagram

https://vuejs.org/api/options-lifecycle.html

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 설계 할 때도 이런 골격을 유지 하면서 설계를 해야 합니다.

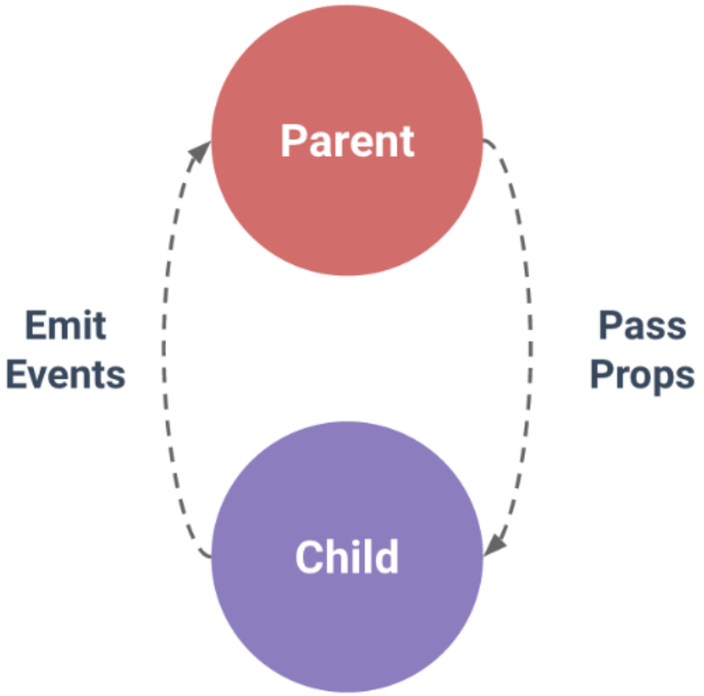


화면을 컴포넌트로 구조화한 컴포넌트 간 관계도

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.



- 뷰에서 상위 - 하위 컴포넌트 간에 데이터를 전달하는 기본적인 구조를 나타냅니다.
- 상위에서 하위로는 `props`라는 특별한 속성을 전달합니다.
- 하위에서 상위로는 `event`를 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

- 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.

Vue3 - 할 일 관리(Todo App)
: 프로젝트 시작하기

Todo App : 프로젝트 생성

1. 새로운 프로젝트 생성 : vue-todo-ts

```
vue create vue-todo-ts
```

```
 Vue CLI v5.x.x
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
> Manually select features
? Check the features needed for your project:
  (*) Babel
> (*) TypeScript
  ( ) Progressive Web App (PWA) Support
  ( ) Router
  ( ) Vuex
  ( ) CSS Pre-processors
  (*) Linter / Formatter
  ( ) Unit Testing
  ( ) E2E Testing
? Choose a version of Vue.js that
    you want to start the project with (Use arrow keys)
> 3.x
  2.x
? Use class-style component syntax? No
? Use Babel alongside TypeScript
   (required for modern mode,
      auto-detected polyfills, transpiling JSX)? (Y/n) y
```

```
? Pick a linter / formatter config: Basic
? Pick a linter / formatter config: (Use arrow keys)
> ESLint with error prevention only
  ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier

? Pick additional lint features:
>(*) Lint on save

? Where do you prefer placing config for Babel, ESLint, etc.?
> In dedicated config files
  In package.json

? Save this as a preset for future projects? No
```

Todo App : 프로젝트 생성

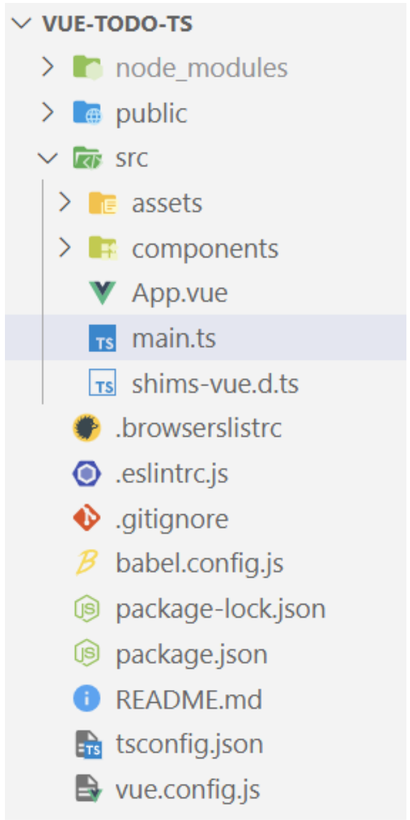
1-1. 프로젝트 실행

```
cd vue-todo-ts

npm run serve
```

```
App running at:
  - Local:   http://localhost:8080/
  - Network: unavailable
```

Vue.js 프로젝트 구조

- vue-todo-ts 프로젝트는 다음과 같이 파일들로 구성되어 있습니다.

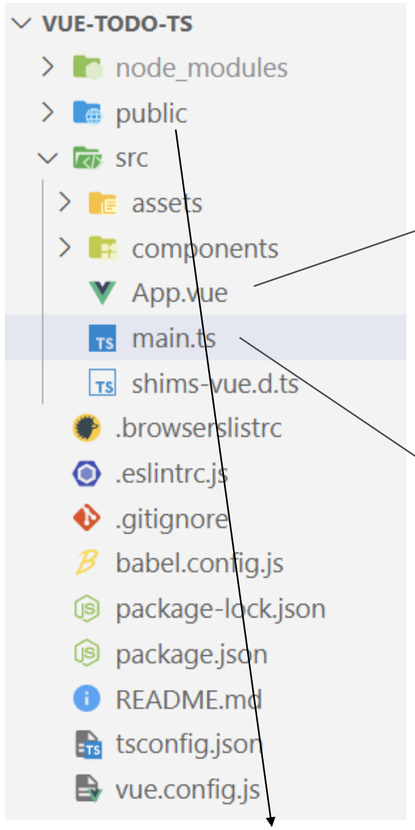


파란색은 vuex와 vue-router를 적용하면 생성될 폴더입니다.

- src/components/HelloWorld.vue
- node_modules : package.json에 종속 되어 있는 라이브러리 폴더
- public : 퍼블리싱 되는 정적 자산을 포함하는 폴더 (static assets)
- src : 어플리케이션 source 폴더
- src/assets : css나 img 등의 정적 자산을 저장하는 폴더, Webpack의 처리를 받을 수 있어 css-pre processor 사용 가능
- src/components : 컴포넌트를 담는 폴더
- src/router : Vue Router 관련 폴더
- src/store : Vuex 관련 폴더
- src/views : Router 페이지 관련 폴더
- src/App.vue : 프로젝트 최상위 컴포넌트 파일
- src/main.ts : 프로젝트의 Entry ts 파일
- src/shims-vue.d.ts : IDE가 .vue 확장자 파일이 어떤 파일인지 이해하기 쉽도록 해주는 파일
- .eslintrc.js : ESLint 설정 파일
- .gitignore : git ignore 설정 파일
- babel.config.js : babel 설정 파일
- package.json : npm(node pakage manager) 설정 파일
- package-lock.json : 동일한 node_module 트리를 생성해서 같은 dependency을 설치 할 수 있도록 해주는 설정 파일
- tsconfig.json : Typescript tsc 컴파일러 설정 파일
- vue.config.js : Vue CLI로 생성한 vue 프로젝트의 Webpack 설정을 변경할 수 있는 설정 파일

Vue.js 프로젝트 구조

- vue-todo-ts 프로젝트는 다음과 같이 파일들로 구성되어 있습니다.



src/App.vue

```vue
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld msg="Welcome to Your Vue.js + TypeScript App"/>
</template>

<script lang="ts">
import { defineComponent } from 'vue';
import HelloWorld from './components/HelloWorld.vue';

export default defineComponent({
  name: 'App',
  components: {
    HelloWorld
  }
});
</script>
```

src/components/HelloWorld.vue

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

src/main.ts

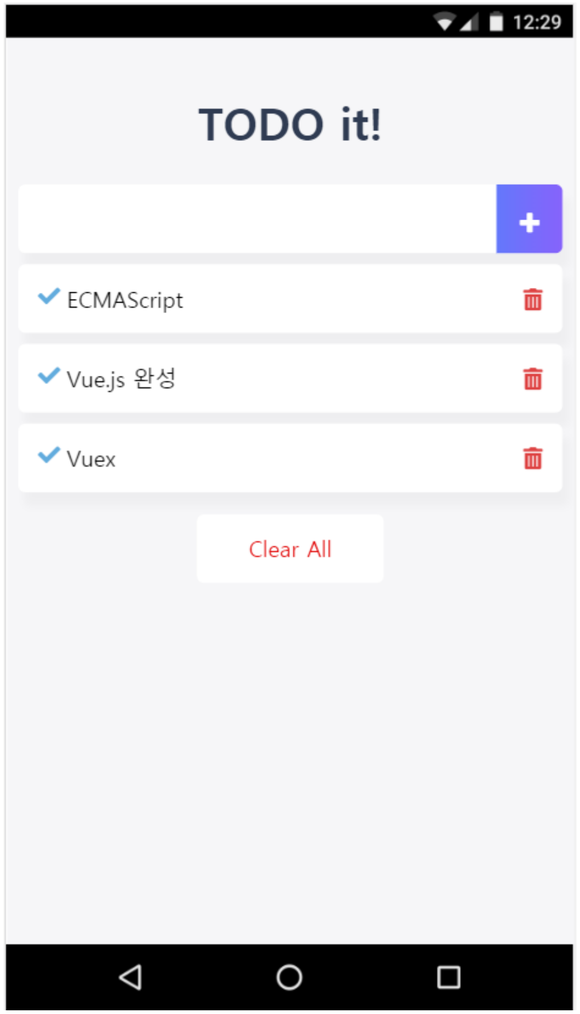
```ts
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

public/index.html

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

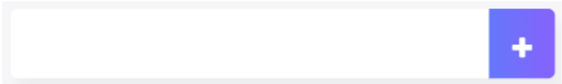
Todo App

1. App.vue : 루트 컴포넌트



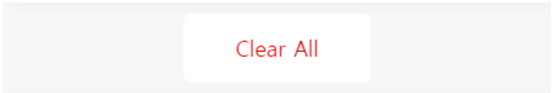
2. TodoHeader.vue : 어플리케이션의 제목 표시하는 컴포넌트

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

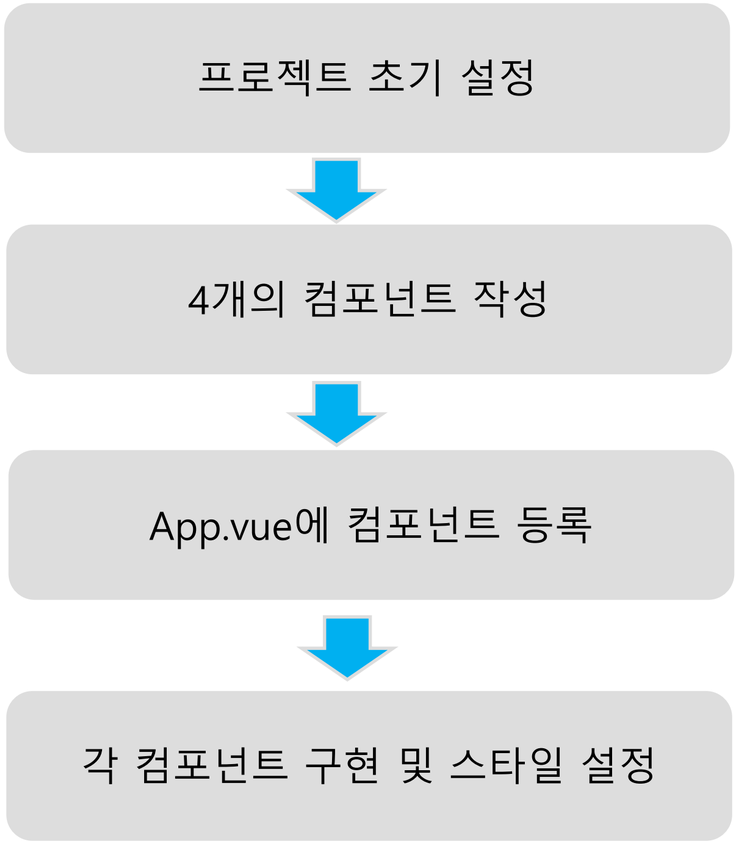
1. App.vue 수정

src/App.vue

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

<script>

</script>

<style>

</style>
```

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) awesome 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 TS 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.

: vbase-3-ts 키워드를 사용하여 템플릿 코드를 자동으로 작성한다.

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 lang="ts">
  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 defineComponent({
    components: {
      TodoHeader, TodoInput, TodoList, 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 scoped>
  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

src/TodoInput.vue

```
<template>
    <div>
        <input type="text" :value="newTodoItem">
        <button>추가</button>
    </div>
</template>

<script lang="ts" setup>
import { ref } from 'vue'

const newTodoItem = ref("")

</script>
```

- **defineComponent vs <script setup>**

https://dev.to/matijanovosel/definecomponent-vs-5104

- **ref() vs reactive()**

ref() and reactive() are used to track changes of its argument.

ref() can take as arguments primitives

(most common: Boolean, String and Number) as well as Objects,

while reactive() can only take Objects as arguments.

TodoApp : 컴포넌트 구현

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

2) 입력 받은 텍스트를 localStorage의 setItem() API를 이용하여 저장하고, newTodoItem 변수 초기화 하기

src/TodoInput.vue

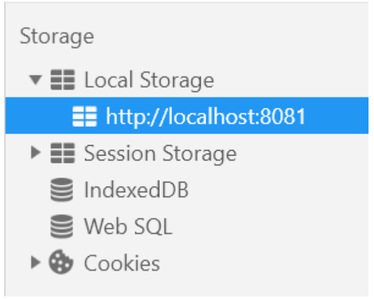
```
<template>
    <div>
        <input type="text" autofocus :value="newTodoItem" @input="handleInput">
        <button @click="addTodo">추가</button>
    </div>
</template>
```

크롬 개발자 도구의

[Application -> Storage -> Local

Storage -> http://localhost:8080] 에서

저장된 값을 확인합니다.



src/TodoInput.vue

```
<template>
<script lang="ts" setup>
import { ref, defineEmits } from "vue"

const newTodoItem = ref("")

const emit = defineEmits(["input:todo"])

const handleInput = (event: Event) => {
    const todoText = (event.target as HTMLInputElement).value
    if (!todoText) return
    emit("input:todo", todoText)
    newTodoItem.value = todoText
}
const addTodo = () => {
    const todoItem = newTodoItem.value
    localStorage.setItem(todoItem, todoItem)
    newTodoItem.value = ""
}
</script>
```

TodoApp : 컴포넌트 구현

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

3) TodoInput.vue에 clearInput() 구현 : addTodo() 안에 예외 처리 코드 추가하기, clearInput() 함수 추가

src/TodoInput.vue

```
<script>
    const addTodo = () => {
      if (newTodoItem.value !== "") {
          const todoItem = newTodoItem.value
          localStorage.setItem(todoItem, todoItem)
          clearInput()
      }
    };
    const clearInput = () => {
       newTodoItem.value = ""
    }
</script>
```

TodoApp : 컴포넌트 구현

4. 컴포넌트 내용 구현하기 : TodoInput Style

4) 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;
  width: 80%;
}
```

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>
```

https://gist.github.com/mysoyul/a59478bf90dd0a56daac4b87bf03c055

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 추가 될 수 있도록 v-on:keyup.enter 이벤트를 처리한다.

src/TodoInput.vue

```
<template>
    <div class="inputBox shadow">
        <input type="text" :value="newTodoItem" @input="handleInput" @keyup.enter="addTodo">
        <span class="addContainer"  @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

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

src/TodoList.vue

```
<script lang="ts" setup>
import { ref, onBeforeMount } from 'vue'

const todoItems = ref<string[]>([])

onBeforeMount(() => {
    console.log('mounted in the composition api!')
    if(localStorage.length > 0){
        for(var i=0; i < localStorage.length; i++) {
            const storageValue = localStorage.key(i) as string
            todoItems.value.push(storageValue)
        }
    }
    console.log(todoItems.value)
})
</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="(item,idx) in todoItems"  v-bind:key="idx">
                {{item}}
            </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>
```

https://gist.github.com/mysoyul/2bb66fa5d8f151cb35a67022d144f1e1

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 삭제)

4) 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 lang="ts" setup>

const removeTodo = (todoItem: string, index: number) => {
    localStorage.removeItem(todoItem)
    todoItems.value.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 삭제)

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

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="(item, index) in todoItems" v-bind:key="index" class="shadow">
                <i class="fas fa-check checkBtn" v-on:click= "toggleComplete" ></i>
                {{item}}
                <span class="removeBtn" v-on:click="removeTodo(item, 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 lang="ts" setup>
const addTodo = () => {
  if(newTodoItem.value !== ""){
    const todoItem = newTodoItem.value
    const todoItemObj = {completed: false, item:todoItem}
    localStorage.setItem(todoItem, JSON.stringify(todoItemObj))
    clearInput()
  }
}
</script>
```

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : Interface 선언

8) 할 일 완료 기능 : onBeforeMount() 메서드 수정

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

src/types/TodoItem.vue

```
interface TodoItem {
    completed: boolean,
    item: string
}
export default TodoItem
```

TodoItem 인터페이스를 정의한다.

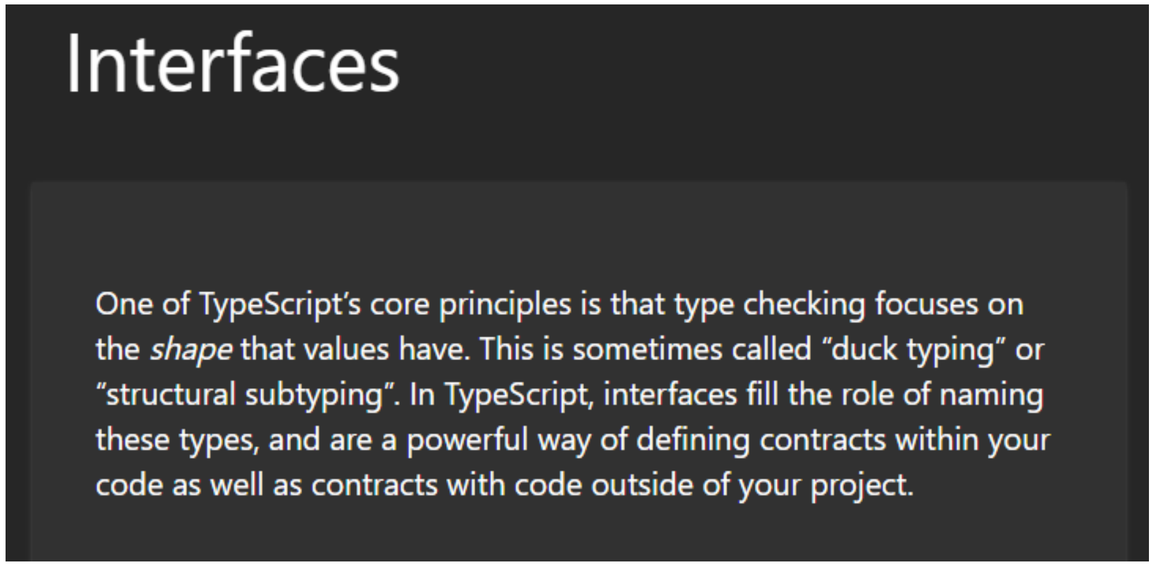
src/TodoList.vue

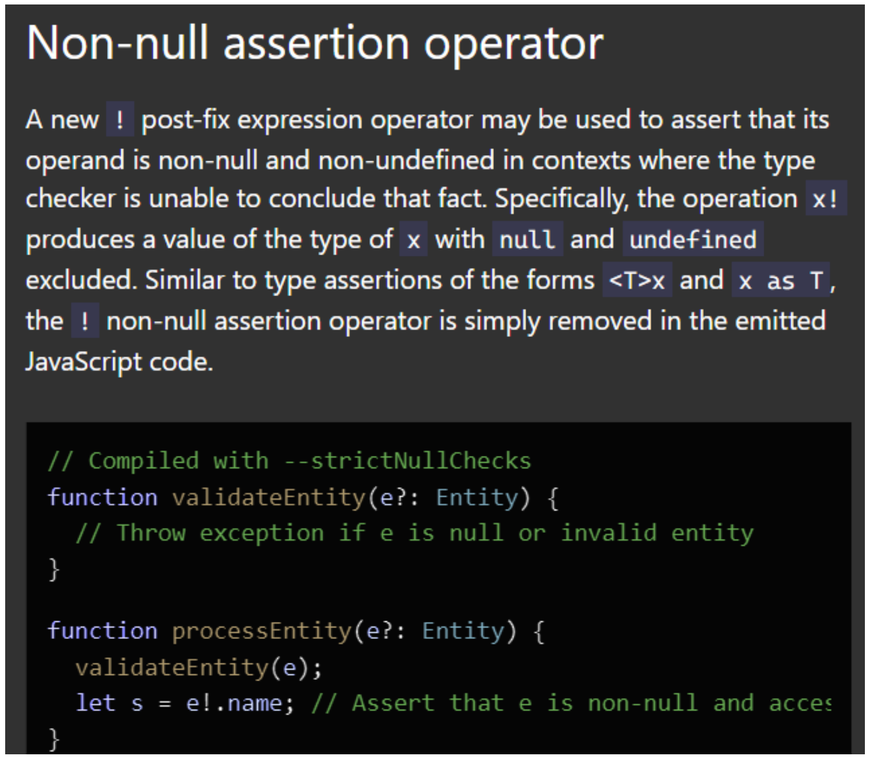
```
<template>
        <ul>
            <li v-for="(todo, index) in todoItems" :key="index" class="shadow">
                <span class="textCompleted">{{todo.item}}</span>
            </li>
        </ul>
</template>
<script lang="ts" setup>
  import TodoItem from '@/types/TodoItem';
  const todoItems = ref<TodoItem[]>([])
  onBeforeMount(() => {
   if(localStorage.length > 0){
       for(var i=0; i < localStorage.length; i++) {
           const storageValue = localStorage.key(i) as string;
           const itemJson = localStorage.getItem(storageValue);
           todoItems.value.push(JSON.parse(itemJson!));
       }
   }
})
</script>
```

TodoApp : 컴포넌트 구현

5. Typescript

TypeScript 의 Interface / Not-null assertion operator





https://www.typescriptlang.org/docs/handbook/interfaces.html

https://learntypescript.dev/07/l2-non-null-assertion-operator

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 완료)

9) 할 일 완료 기능 : v-bind directive 사용

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

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

src/TodoList.vue

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

TodoApp : 컴포넌트 구현

5. 컴포넌트 내용 구현하기 : TodoList (todo 완료)

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

todo.completed 값(true/false)의 토글링에 따라서 localStorage에 저장된 completed 값도 변경 해준다.

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

src/TodoList.vue

```
<template>
        <ul>
            <li v-for="(todo, index) in todoItems" :key= "index" class="shadow">
                    <i class="fas fa-check checkBtn" :class="{checkBtnCompleted: todo.completed}"
                     @click="toggleComplete(todo)"></i>
            </li>
        </ul>
</template>
<script lang="ts" setup>
 const toggleComplete = (todoItem: TodoItem) => {
    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, 전체 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" @click="clearTodo">Clear All</span>
    </div>
</template>
<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
    setup () {
        const clearTodo = () => {
            localStorage.clear()
        }
        return {clearTodo}
    }
})
</script>
```

할 일 관리(Todo App)
: 구조 개선하기(리팩토링)

TodoApp : 문제점 & 해결방법

1. 현재 어플리케이션 구조의 문제점

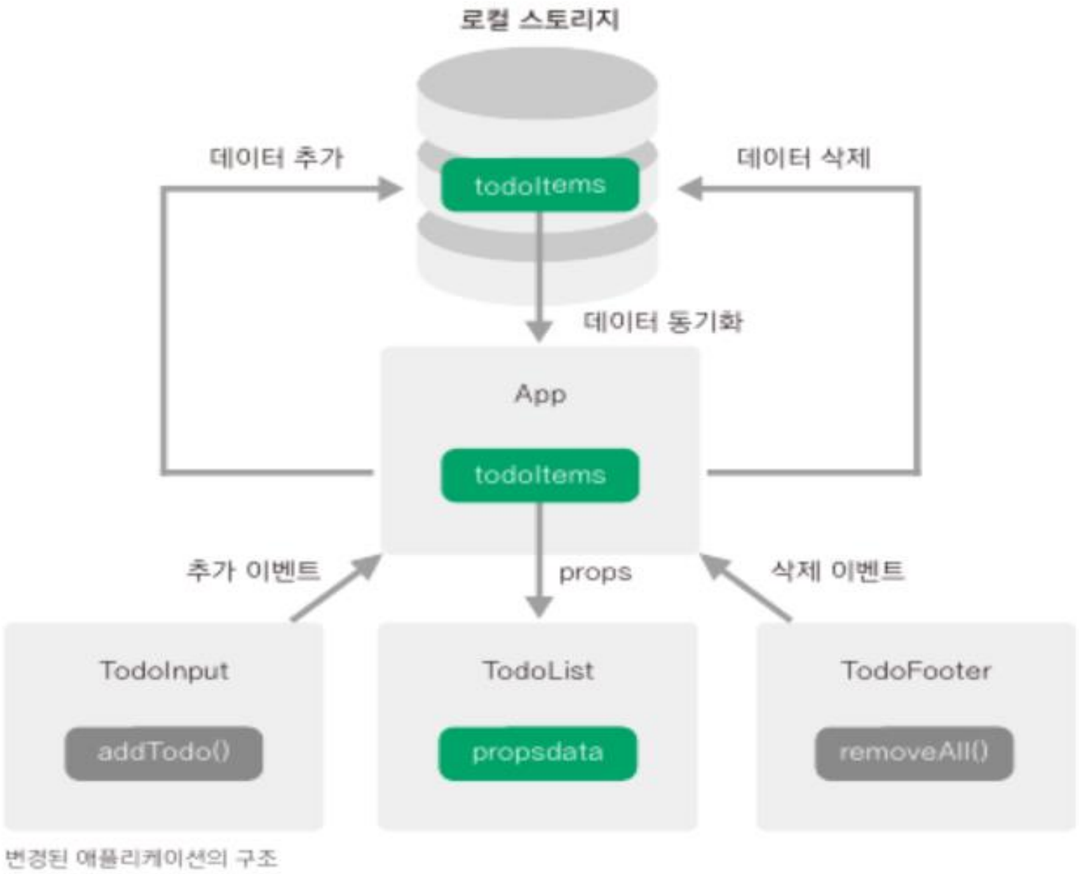
- 할 일을 입력 했을 때 할일 목록에 바로 반영되지 않는 점
- 할 일을 모두 삭제했을 때 목록에 바로 반영되지 않는 점

2. 해결 방법

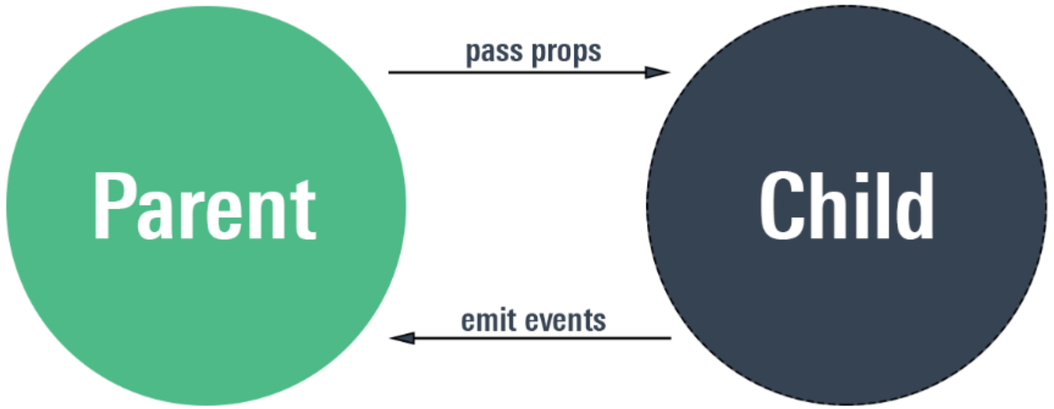
- 각각의 컴포넌트에서 각자 뷰 데이터 속성(newTodoItem, todoItems)을 가지고 있지만, localStorage의 데이터는 공유하고 있는 상황이다.
- 문제점을 해결하기 위해 데이터 속성을 최상위(루트) 컴포넌트인 App 컴포넌트에 todoItems을 정의하고, 하위 컴포넌트 TodoList에 props 로 전달합니다.
- 하위 컴포넌트에서 발생한 이벤트를 $.emit을 이용하여 상위 컴포넌트로 전달합니다.

TodoApp : 문제점 & 해결방법

2. 해결 방법



변경된 애플리케이션의 구조



TodoApp : 컴포넌트 구현 리팩토링

1. [리팩토링] 할 일 목록 표시 기능 : App 수정

TodoList에 있던 todoItems 데이터 변수와 onBeforeMount() life cycle hook method를 App으로 옮긴다.

App에서 TodoList에게 props로 전달합니다.

src/App.vue

```
<template>
<div>
<TodoList :propsdata="todoItems"></TodoList>
</div>
</template>

<script lang="ts">
import { reactive, onBeforeMount } from "vue";
import TodoItem from "@/types/TodoItem";
.....
```

src/App.vue

```
export default defineComponent({
 setup() {
    const todoItems = reactive<TodoItem[]>([]);

    onBeforeMount(() => {
      if (localStorage.length > 0) {
        for (var i = 0; i < localStorage.length; i++) {
          const storageKey = localStorage.key(i) as string;
          const itemJson = localStorage.getItem(storageKey) as string | null;
          if (itemJson) {
            todoItems.push(JSON.parse(itemJson));
          } //if
        } //for
      } //if
    });
    return { todoItems };
  }, //setup
})
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

1. [리팩토링] 할 일 목록 표시 기능 : TodoList 수정

definePropS() 함수로 App에서 전달 받은 props의 type과 required 속성을 선언한다.

src/components/TodoList.vue

```
<template>
  <div>
    <ul>
      <li v-for="(todo, index) in [...props.propsdata]" :key="index"
          class="shadow">
      </li>
  </div>
</template>
```

src/components/TodoList.vue

```
<script lang="ts" setup>
import { defineProps, PropType } from "vue";
import TodoItem from "@/types/TodoItem";

const props = defineProps({
   propsdata: { type: Array as PropType<TodoItem[]>, required: true }
})

</script>
```

TodoApp : 컴포넌트 구현 리팩토링

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

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

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

src/App.vue

```
<template>
  <div id="app">
    <TodoInput @add:todo="addTodo"></TodoInput>
  </div>
</template>
<script lang="ts">
export default defineComponent({
      setup() {
        ...
        const addTodo = (todoItemStr: string) => {
            const todoItemObj = { completed: false, item: todoItemStr };
            localStorage.setItem(todoItemStr, JSON.stringify(todoItemObj));
            todoItems.push(todoItemObj);
        };//addTodo

    return { todoItems, addTodo };
  }, //setup
});
</script>
```

src/components/TodoInput.vue

```
<script lang="ts" setup>
import { ref, defineEmits  } from "vue";

const newTodoItem = ref("")

const emit = defineEmits(["input:todo","add:todo"])
...
const addTodo = () => {
  if(newTodoItem.value !== ""){
    const todoItemStr = newTodoItem.value
    emit("add:todo", todoItemStr)
    clearInput()
  }
}
</script>
```

TodoApp : 컴포넌트 구현 리팩토링

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

TodoList 에서 발생한 클릭 Event를 App에 전달할 때 emit(“이벤트이름”, 인자) 를 사용한다.

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

src/App.vue

```
<template>
  <div id="app">
    <TodoList :propsdata="todoItems" @remove:todo="removeTodo">
    </TodoList>
  </div>
</template>
<script lang="ts">
export default defineComponent({
     setup() {
        ...
    const removeTodo = (todoItemStr: string, index: number) => {
      localStorage.removeItem(todoItemStr);
      todoItems.splice(index, 1);
    };

    return { todoItems, addTodo, removeTodo };
  }, //setup
});
</script>
```

src/components/TodoList.vue

```
<script lang="ts" setup>
import { defineProps, PropType, defineEmits } from "vue";
import TodoItem from "@/types/TodoItem";

const emit = defineEmits(["remove:todo"])

const removeTodo = (todoItemStr: string, index: number) => {
  emit("remove:todo", todoItemStr, index)
};

</script>
```

TodoApp : 컴포넌트 구현 리팩토링

4. [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 :propsdata="todoItems"
           @toggle:todo="toggleComplete">
    </TodoList>
  </div>
</template>
<script lang="ts">
export default defineComponent({
     setup() {
       ...
    const toggleComplete = (todoItem: TodoItem) => {
      todoItem.completed = !todoItem.completed;
      localStorage.removeItem(todoItem.item);
      localStorage.setItem(todoItem.item, JSON.stringify(todoItem));
    };
    return { todoItems, addTodo, removeTodo, toggleComplete };
  }, //setup
});
</script>
```

src/components/TodoList.vue

```
<script lang="ts" setup>
import { defineProps, PropType, defineEmits } from "vue";
import TodoItem from "@/types/TodoItem";

const emit = defineEmits(["remove:todo","toggle:todo"])

const props = defineProps({
  propsdata: { type: Array as PropType<TodoItem[]>, required: true }
})

const toggleComplete = (todoItem: TodoItem) => {
  emit("toggle:todo", todoItem)
};

</script>
```

TodoApp : 컴포넌트 구현 리팩토링

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

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

src/App.vue

```
<script lang="ts">
export default defineComponent({
     setup() {
       ...
    const toggleComplete = (todoItem: TodoItem, index: number) => {
      todoItems[index].completed = !todoItem.completed;
      localStorage.removeItem(todoItem.item);
      localStorage.setItem(todoItem.item,
            JSON.stringify(todoItems[index]));
    };
    return { todoItems, addTodo, removeTodo, toggleComplete };
  }, //setup
});
</script>
```

src/components/TodoList.vue

```
<template>
        <i class="fas fa-check checkBtn"
          :class="{ checkBtnCompleted: todo.completed }"
          @click="toggleComplete(todo, index)"
        ></i>
</template>

<script lang="ts" setup>
const toggleComplete = (todoItem: TodoItem, index: number) => {
  emit("toggle:todo", todoItem, index)
};

</script>
```

TodoApp : 컴포넌트 구현 리팩토링

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

TodoFooter 에서 발생한 클릭 Event를 App에 전달할 때 emit(“이벤트이름”, 인자) 를 사용한다.

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

src/App.vue

```
<template>
  <div id="app">
    <TodoFooter @clear:todo="clearTodo"></TodoFooter>
  </div>
</template>
<script lang="ts">
export default defineComponent({
     setup() {
       ...
    const clearTodo = () => {
      localStorage.clear()
      todoItems.splice(0)
    }
    return { todoItems, addTodo, removeTodo, toggleComplete, clearTodo };
  }, //setup
});
</script>
```

src/components/TodoFooter.vue

```
<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
    emits: ['clear:todo'],

    setup (props, { emit }) {
        const clearTodo = () => {
            emit('clear:todo')
        }
        return {clearTodo}
    }
})
</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/examples/#modal
- Enter/Leave & List Transitions https://vuejs.org/guide/built-ins/transition.html
  https://vuejs.org/guide/built-ins/transition-group.html

TodoApp : 컴포넌트 구현

1. 뷰 Modal : TodoInput 수정, MyModal 추가

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

src/components/TodoInput.vue

```
<template>
    <div class="inputBox shadow">
        <span class="addContainer" @click="addTodo">
            <i class="fas fa-plus addBtn"></i>
        </span>
        <MyModal v-if="showModal" @close="showModal = false">
          <template v-slot:header>
              <h3>custom header</h3>
          </template>
        </MyModal>
    </div>
</template>

<script lang="ts" setup>
import MyModal from './common/MyModal.vue'

const showModal = ref(false);
</script>
```

src/components/common/MyModal.vue

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

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

https://gist.github.com/mysoyul/b1b126771ecf6d63366ba4388e870e49

TodoApp : 컴포넌트 구현

1-1. 뷰 Modal : TodoInput 수정

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

src/components/TodoInput.vue

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

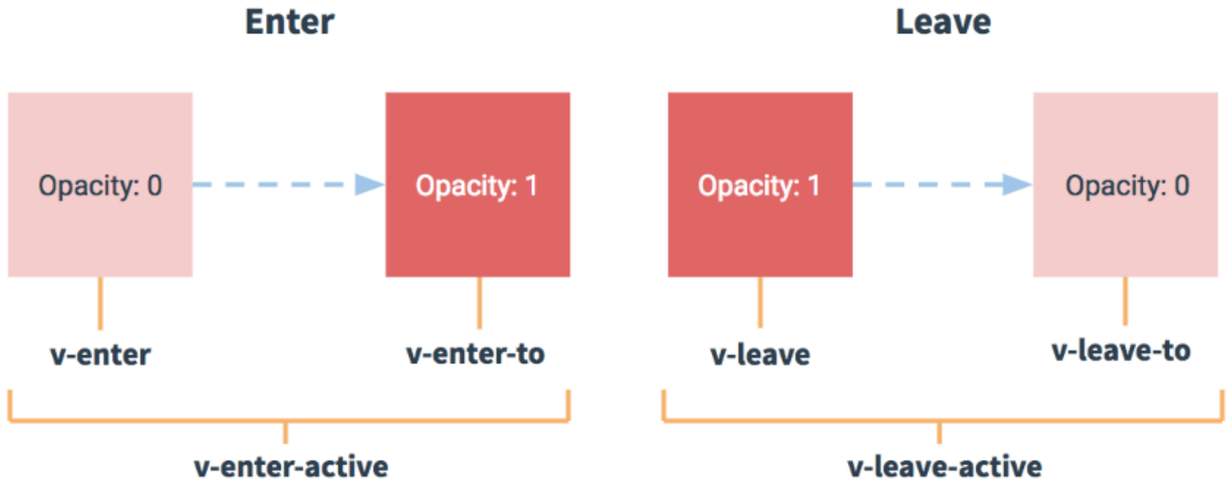
src/components/TodoInput.vue

```
<script lang="ts" setup>
import MyModal from './common/MyModal.vue'

const showModal = ref(false)

const addTodo = () => {
  if (newTodoItem.value !== "") {
    const todoItem = newTodoItem.value
    emit("add:todo", todoItem)
    clearInput();
  } else {
    showModal.value = !showModal.value
  }
}
</script>
<style>
.closeModalBtn {
  color: #42b983;
}
</style>
```

TodoApp : 컴포넌트 구현

2. 뷰 애니메이션 : TodoList 수정

리스트 아이템의 트랜지션 효과 스타일 설정", <ul> 엘리먼트를 <TransitionGroup> 엘리먼트로 변경한다.



src/components/TodoList.vue

```
<template>
    <div class="inputBox shadow">
        <TransitionGroup name="list" tag="ul">
            .....
        </TransitionGroup>
    </div>
</template>
<style>
.list-enter-active, .list-leave-active {
    transition: all 0.5s ease;
}
.list-enter-from, .list-leave-to {
    opacity: 0;
    transform: translateX(30px);
}
</style>
```

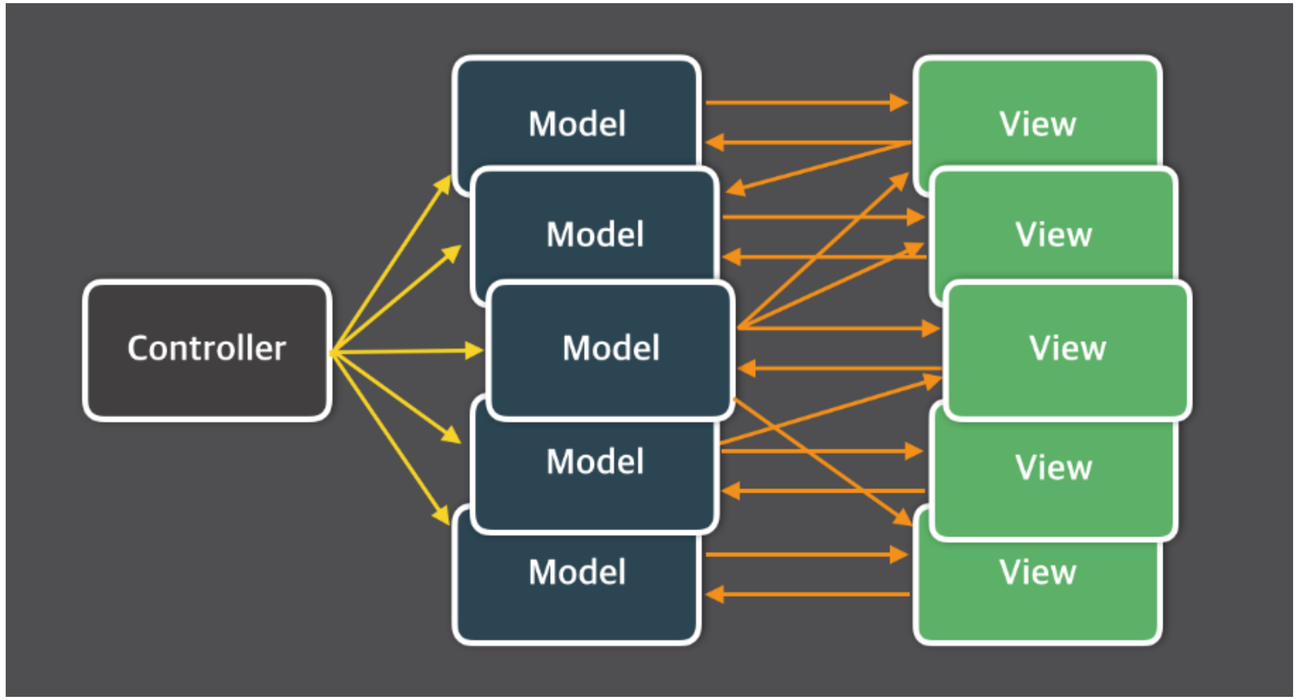
Vuex (상태 관리 라이브러리)

MVC 문제점

- MVC 패턴의 문제점

페이스북에서 이야기 하는 MVC의 가장 큰 단점은 양방향 데이터 흐름이었습니다.

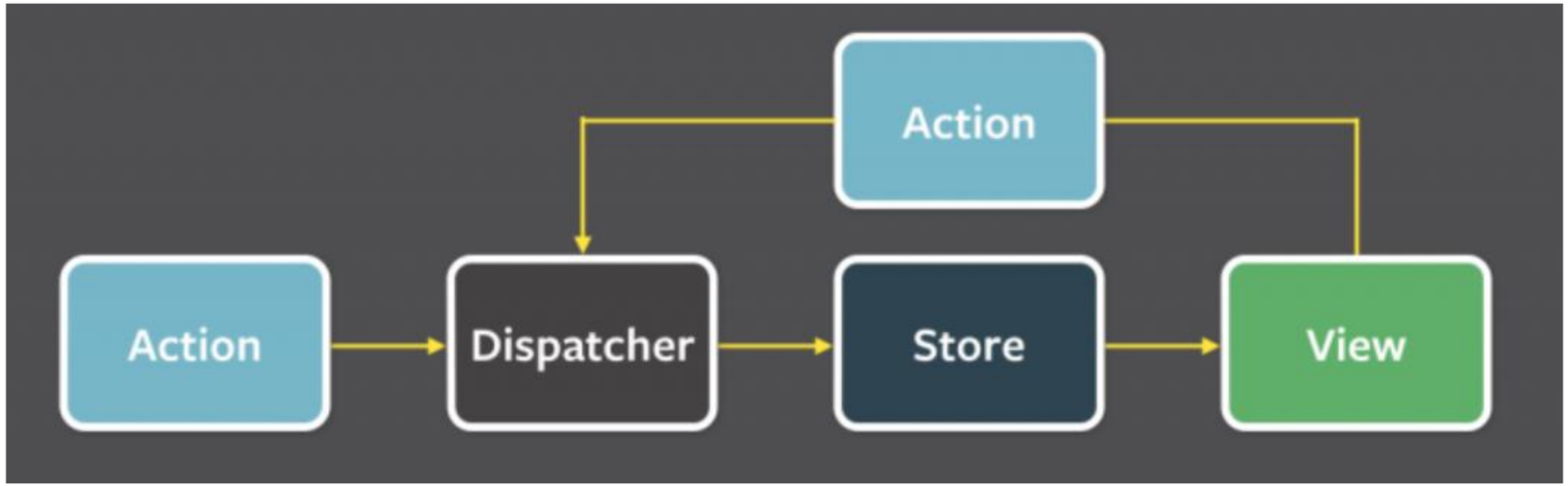
Model이 업데이트 되어 View가 따라서 업데이트 되고, 업데이트 된 View가 다시 다른 Model을 업데이트 한다면, 또 다른 View가 업데이트 될 수 있습니다. 복잡하지 않은 어플리케이션에서는 양방향 데이터 흐름이 문제가 크지 않을 수 있습니다. 하지만 어플리케이션이 복잡해 진다면 이런 양방향 데이터 흐름은 새로운 기능이 추가 될 때에 시스템의 복잡도를 증가 시키고, 예측 불가능한 코드를 만들게 됩니다.



Flux란?

- Flux로 해결

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

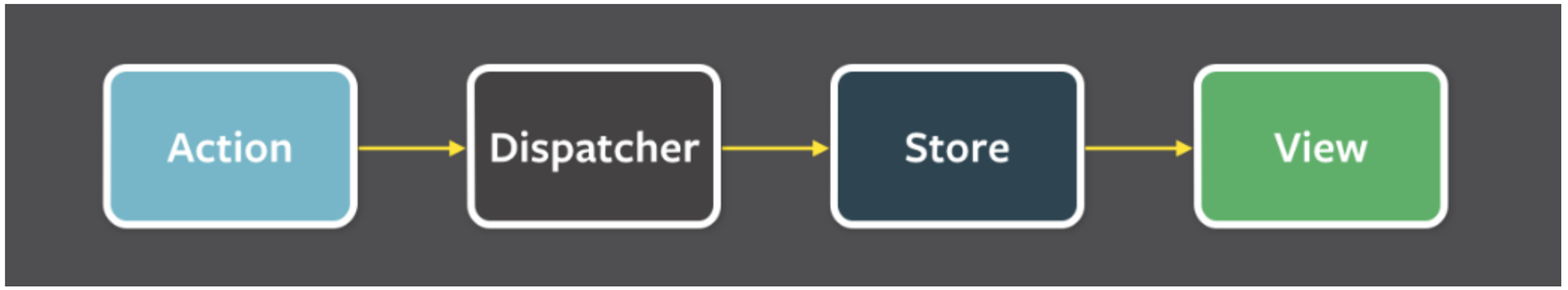


: Flux의 가장 큰 특징은 단방향 데이터 흐름입니다. 데이터 흐름은 항상 Dispatcher에서 Store로, Store에서 View로, View는 Action을 통해 다시 Dispatcher로 데이터가 흐르게 됩니다. 이런 단방향 데이터 흐름은 데이터 변화를 훨씬 예측하기 쉽게 만듭니다.

Flux란?

- Flux로 해결

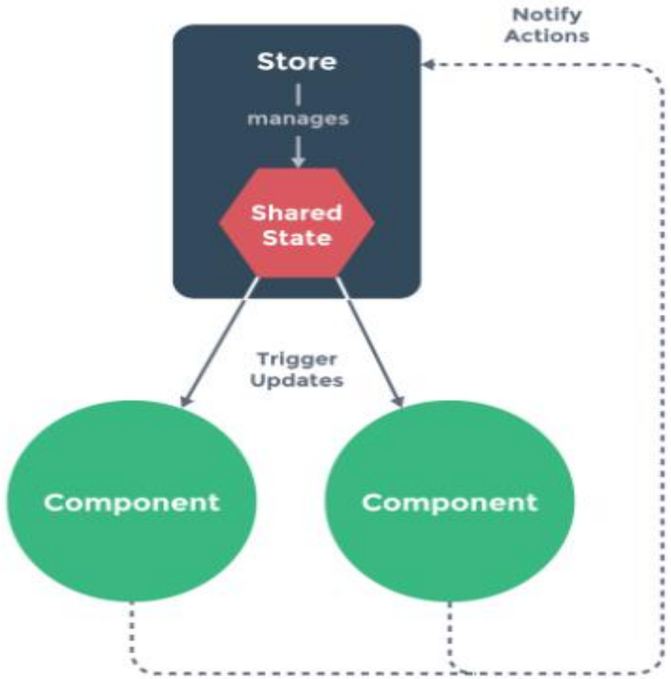
1. Dispatcher : Action이 발생되면 Dispatcher로 전달되는데, Dispatcher는 전달된 Action을 보고, 등록된 콜백 함수를 실행하여 Store에 데이터를 전달합니다.
2. Model(Store) : 어플리케이션의 모든 상태(state) 변경은 Store에 의해 결정이 됩니다.
3. View : 사용자에게 비춰지는 화면
4. Action : Dispatcher에서 콜백 함수가 실행 되면 Store가 업데이트 되게 되는데, 이 콜백 함수를 실행 할 때 데이터가 담겨 있는 객체가 인수로 전달 되어야 합니다.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



Vuex

- Vuex란?

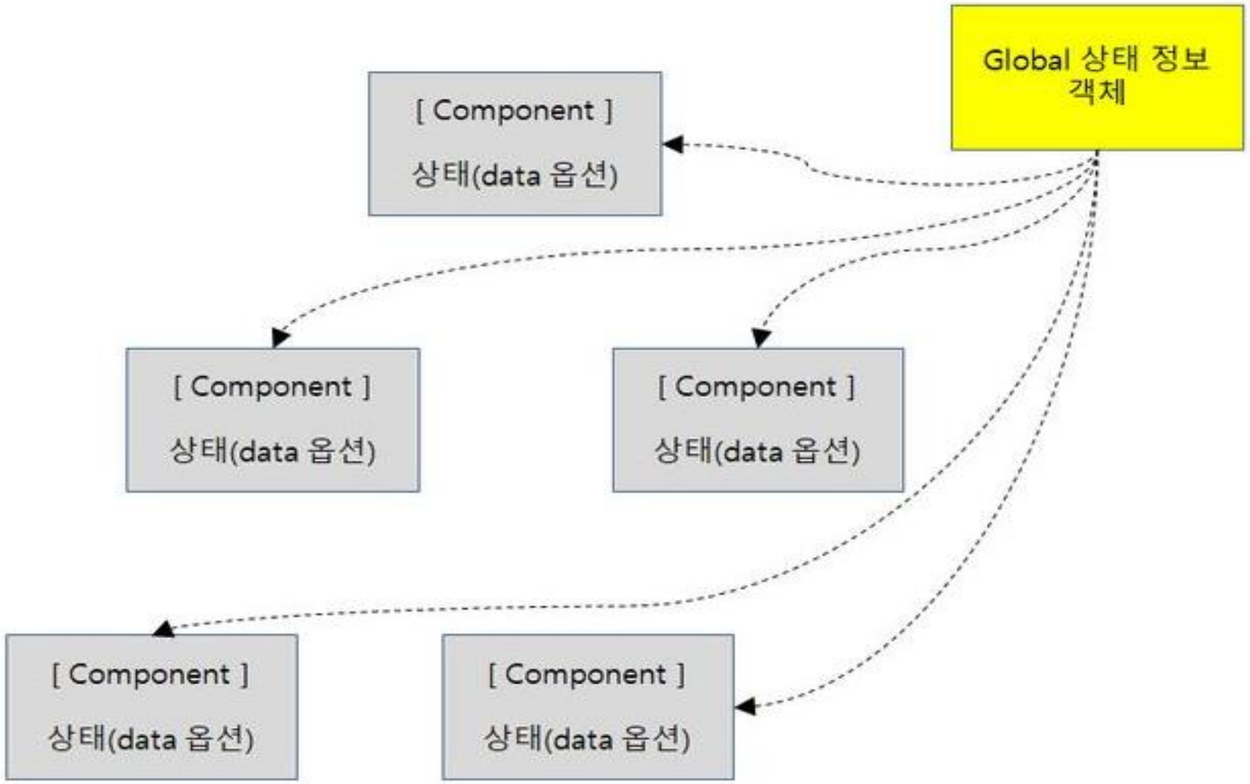
1. 무수히 많은 컴포넌트의 데이터를 더 효율적으로 관리 하는데 사용하는 상태 관리 라이브러리이다.
2. Vuex는 Vue.js 애플리케이션에 대한 상태 관리 패턴 + 라이브러리 입니다. 애플리케이션의 모든 컴포넌트에 대한 중앙 집중식 저장소 역할을 하며 예측 가능한 방식으로 상태를 변경할 수 있습니다.
3. Vuex의 기본 아이디어는 Flux, Redux, The Elm Architecture에서 영감을 받았습니다.



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.



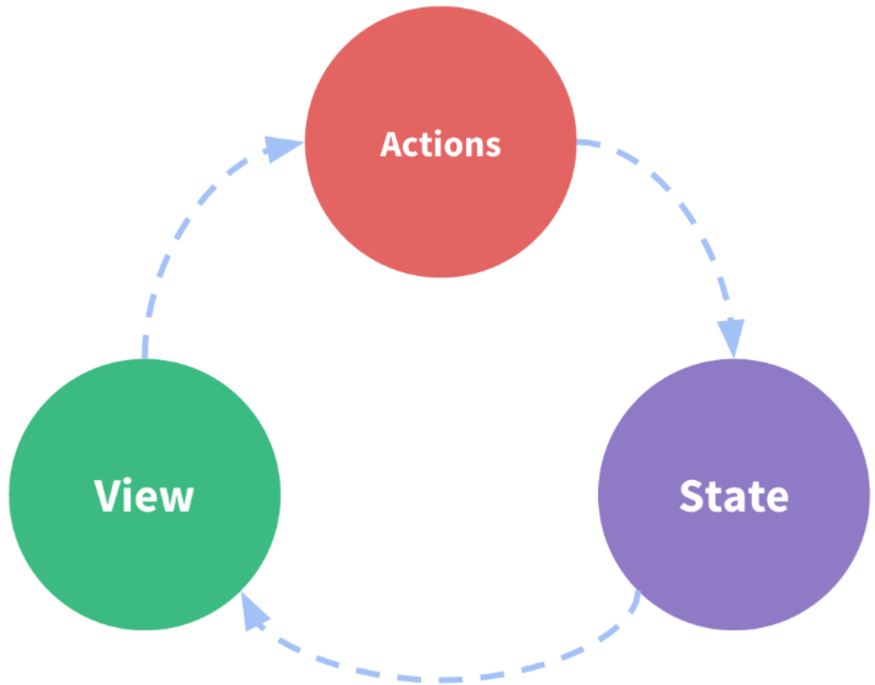
Vuex

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

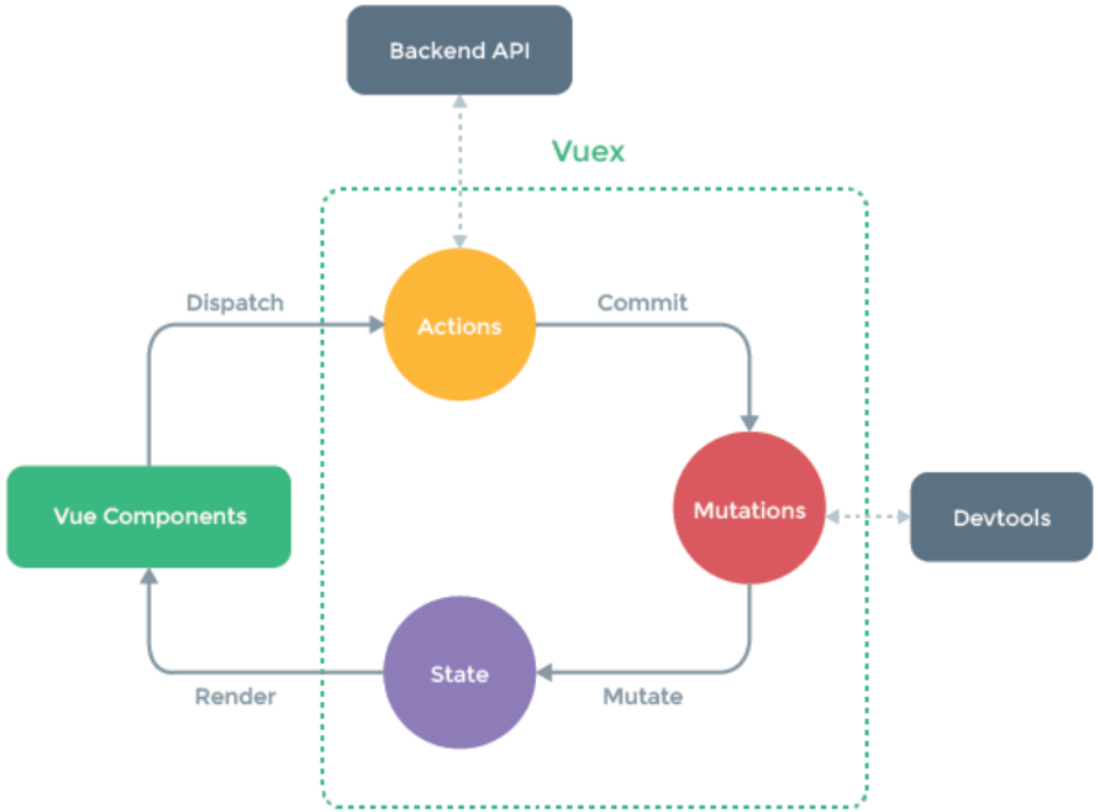
- State : 컴포넌트 간에 공유하는 데이터 data()
- View : 데이터를 표시하는 화면 template
- Action : 사용자의 입력에 따라 데이터를 변경하는 methods



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.

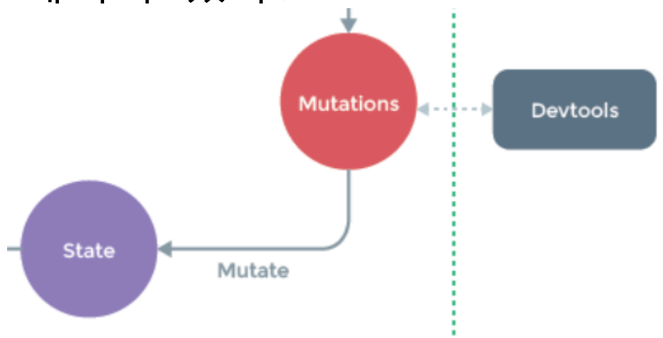
Vuex

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 : 설치

1. Vuex 설치하기

```
npm install vuex
```

2. Store 생성

- src/store 디렉토리와 index.ts 파일을 생성한다.
- 루트 컴포넌트에 store 옵션을 제공하면, store는 루트의 모든 하위 컴포넌트에 주입 되어진다.

src/store/index.ts

```
import { createStore, createLogger } from "vuex"

export const store = createStore({
    plugins: process.env.NODE_ENV === 'development' ?
      [createLogger()] : [],
})
```

src/main.ts

```
import { createApp } from 'vue'
import App from './App.vue'
import { store } from "./store"

createApp(App)
    .use(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 : state 속성 적용

1. TodoApp : 리팩토링 state 속성 적용

1) state에 todoItems 속성을 정의한다.

2) App.vue에 있는 onBeforeMount() 메서드의 구현 내용을 store/index.ts로 이동하고, 메서드 이름은 fetch() 로 변경한다.

src/store/index.ts

```
import { createStore, createLogger } from "vuex"
import TodoItem from "@/types/TodoItem"

const storage = {
    fetch() {
        const arr = [];
        if(localStorage.length > 0){
          for(let i=0; i < localStorage.length; i++) {
               const storageKey = localStorage.key(i) as string;
               const itemJson = localStorage.getItem(storageKey) as string | null
               if (itemJson){
                    arr.push(JSON.parse(itemJson))
               }
          }
        }
        return arr;
    }
}
```

src/store/index.ts

```
export type State = { todoItems: TodoItem[] };

const state: State = { todoItems: storage.fetch() };

export const store = createStore({
    state,
})
```

Vuex : state 속성 적용

1. TodoApp : 리팩토링 state 속성 적용

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

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

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

src/component/TodoList.vue

```
<template>
    <div>
        <TransitionGroup name="list" tag="ul">
            <li v-for="(todoItem, index) in todoItems" :key="index" class="shadow">
              ...
            </li>
        </TransitionGroup>
    </div>
</template>
<script lang="ts" setup>
import { useStore } from "vuex"
import { computed  } from "vue"

const store = useStore()
const todoItems = computed(() => store.state.todoItems)
</script>
```

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/index.ts에 mutations 속성을 정의한다.

2) store/index.ts의 mutations 속성 내에 App.vue에 있는 addTodo() 메서드를 이동시킨다.

3) App.vue의 <TodoInput @add:todo="addTodo"></TodoInput> ➔ <TodoInput></TodoInput>로 변경한다.

src/store/index.ts

```
export const store = createStore({
    state,
    mutations: {
        addTodo(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

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

src/component/TodoInput.vue

```
<script lang="ts" setup>
import { useStore } from "vuex"

const store = useStore()

const addTodo = () => {
  if(newTodoItem.value !== ""){
    const todoItem = newTodoItem.value
    store.commit("addTodo", todoItem)
    clearInput()
  }
}
</script>
```

Vuex : [리팩토링] mutations 적용

3. TodoApp 할 일 삭제 : 리팩토링 mutations 적용

1) App.vue에 있는 removeTodo() 메서드를 store/index.ts의 mutations 속성내에 선언한다.

2) App.vue의 <TodoList @remove:todo="removeTodo"></TodoList> ➔ <TodoList></TodoList>로 변경한다.

src/store/index.ts

```
export const store = createStore({
    state,
    mutations: {
        removeTodo(state, payload) {
            const {todoItem, index} = payload
            localStorage.removeItem(todoItem.item);
            state.todoItems.splice(index, 1);
        },
    },
})
```

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 removeTodo() 메서드를 직접 호출한다.

emit("remove:todo", todoItem, index) ➔ store.commit('removeTodo', {todoItem, index})

※ 원래는 store.commit('removeTodo', {todoItem:todoItem, index:index}); 이지만 ES6 Enhanced Object Literals를 적용해서 전달하는 객체의 key와 value 값이 동일 하므로 {todoItem, index} 로 전달한다.

src/component/TodoList.vue

```
<template>
        <span class="removeBtn" @click="removeTodo(todo, index)">
          <i class="fas fa-trash-alt"></i>
        </span>
</template>
<script lang="ts" setup>
import { useStore } from "vuex"

const store = useStore()

const removeTodo = (todoItem: TodoItem, index: number) => {
   store.commit("removeTodo", {todoItem, index})
}
</script>
```

Vuex : [리팩토링] mutations 적용

4. TodoApp 할 일 완료 : 리팩토링 mutations 적용

1) App.vue에 있는 toggleComplete() 메서드를 store/index.ts의 mutations 속성내에 선언한다.

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

src/store/index.ts

```
export const store = createStore({
    state,
    mutations: {
        toggleTodo(state, payload) {
            const {todoItem, index} = payload
            state.todoItems[index].completed = !todoItem.completed
            localStorage.removeItem(todoItem.item);
            localStorage.setItem(todoItem.item, JSON.stringify(state.todoItems[index]));
        }
    },
})
```

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 toggleComplete() 메서드를 직접 호출한다.

emit("toggle:todo", todoItem, index) ➔ store.commit('toggleTodo', {todoItem, index})

src/component/TodoList.vue

```
<template>
        <i class="fas fa-check checkBtn"
          :class="{ checkBtnCompleted: todo.completed }"
          @click="toggleComplete(todo, index)"></i>
</template>
<script lang="ts" setup>
import { useStore } from "vuex"

const store = useStore()

const toggleComplete = (todoItem: TodoItem, index: number) => {
   store.commit("toggleTodo", {todoItem, index})
}
</script>
```

Vuex : [리팩토링] mutations 적용

5. TodoApp 할 일 모두 삭제 : 리팩토링 mutations 적용

1) App.vue에 있는 clearTodo() 메서드를 store/index.ts의 mutations 속성내에 선언한다.

2) App.vue의 <TodoFooter @clear:todo="clearTodo"></TodoFooter> ➔ <TodoFooter></TodoFooter>로 변경한다.

src/store/index.ts

```
export const store = createStore({
    state,
    mutations: {
         clearTodo(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 clearTodo() 메서드를 직접 호출한다.

src/component/TodoFooter.vue

```
<script lang="ts">
import { defineComponent } from 'vue'
import { useStore } from "vuex"

export default defineComponent({
    setup () {
        const store = useStore()

        const clearTodo = () => {
            store.commit("clearTodo")
        }
        return {clearTodo}
    }
})
</script>
```

Axios

Nodejs + 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!');
});
```

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)

@types 설치

- @types (https://github.com/DefinitelyTyped/DefinitelyTyped)

- TypeScript로 작성되지 않은 라이브러리를 import 해야 할 때도 있다.
- 만약에 styled-component를 import 하려면, 아래 명령어를 사용한다.

  $ npm i --save-dev @types/styled-components
- @types란 DefinitelyTyped라는 github Repository이며 대부분의 npm 라이브러리들을 가지고 있고 TypeScript로 작업 할 때 필요한 대부분의 라이브러리나 패키지의 type definition을 얻을 수 있다.

- @types/node는 Node.js의 type 정의를 포함하고 있는 패키지이다.

  @types/node 설치하기

  $ npm i --save-dev @types/node

Nodejs + Express

- Typescript로 Nodejs와 Express 설정하기

1. package.json 파일을 생성하고 의존성 설치하기

```
$ npm init
$ npm i express cors
$ npm i  -D typescript @types/node @types/express @types/cors @types/body-parser nodemon ts-node
```

- express : NodeJS를 사용하여 서버를 개발 할 때 서버를 쉽게 구성할 수 있게 만든 프레임워크
- cors : cross origin resource sharing 지원
- @types/node : type definitions for Node.js
- @types/express : express는 원래 자바스크립트로 만들어졌다. TS 환경에서 사용하기 위해 express 내부의 변수들, 함수들의 타입을 정의한 ~d.ts 파일이 포함된 형태
- @types/cors : type definitions for cors
- @types/body-parser : type definitions for body-parser
- nodemon: 노드 서버를 실행하고 소스코드를 수정하면 다시 서버 재시작을 자동으로 해주는 모듈
- ts-node: Node.js용 TS 실행 엔진 및 REPL(Node.js 상에서 Typescript Compile 하지 않고, ts를 실행하는 역할을 한다.)

Nodejs + Express

- Typescript로 Nodejs와 Express 설정하기

2. tsconfig.json 생성 (tsc 컴파일러 설정 파일)

```
$ tsc --init
{
  "compilerOptions": {
    "target": "es6", // 어떤 버전으로 컴파일
    "module": "commonjs", //어떤 모듈 방식으로 컴파일
    "outDir": "./dist",        //컴파일 후 js 파일들이 생성되는 디렉토리
    "rootDir": ".",     //루트 폴더
    "strict": true,     //strict 옵션 활성화
    "moduleResolution": "node",  //모듈 해석 방법 설정: 'node' (Node.js)
    "esModuleInterop": true   //ES6 모듈 사양을 준수하여 CommonJS 모듈을 import 할 수 있도록
  }
}
```

Nodejs + Express

- Typescript로 Nodejs와 Express 설정하기

3. package.json 수정

```
"scripts": {
    "start": "node dist/app.js",
    "build": "tsc -p .",
    "dev": "nodemon --watch \"src/**/*.ts\" --exec \"ts-node\" src/app.ts"
}
```

npm start : 컴파일된 js 파일을 node로 실행

npm run build : typescript를 javascript로 컴파일

npm run dev : 서버를 시작하고, ts-node로 app.ts 실행

Nodejs + Express

- Typescript로 Nodejs와 Express 설정하기

4. src/app.ts

src/app.ts

```
import express, { Request, Response, NextFunction } from 'express';
import cors from 'cors';

const app = express();
app.use(cors<Request>());
app.use(express.json())

const port = 4500;
let nextId = 4;

type TodoItem = {
  id: number,
  completed: boolean,
  item: string
}
type Post = {
  id: number,
  text: string
};
```

https://github.com/mysoyul/node_express_typescript_Server/blob/master/src/app.ts

Axios

Axios(https://github.com/axios/axios)는 성공적인 **HTTP 클라이언트 라이브러리** 중 하나입니다.

- 요청 취소와 TypeScript를 사용할 수 있습니다.

>> 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 설치하기 :

```
npm install axios
```

Axios

1. Axios 의 사용

- axios를 import 하여 사용한다.

src/common/http-common.ts

```
import axios, { AxiosInstance } from "axios"

const apiClient: AxiosInstance = axios.create({
  baseURL: "http://localhost:4500/api",
  headers: {
    "Content-type": "application/json",
  },
})

export default apiClient
```

src/store/index.ts

```
import http from "@/common/http-common"

const state: State = { todoItems: [] };

export const store = createStore({
    state,
    actions: {

    } //action
})
```

Vuex axios : Todo 목록

2. TodoApp 할 일 목록 :

1) index.ts 의 state에 todoItems 변수를 초기화 한다.

2) getters , mutations , actions 프로퍼티를 추가한다.

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

src/store/index.ts

```
import http from "@/common/http-common"
import axios from "axios"

export const store = createStore({

    getters: {
        getTodoItems(state) {
            return state.todoItems;
        }
    },
    mutations: {
        setTodoItems(state, items) {
            state.todoItems = items;
        },
    },
```

src/store/index.ts

```
    actions: {
          loadTodoItems ({commit}) {
             http
               .get('/todos')
               .then(r => r.data)
               .then(items => {
                 commit('setTodoItems', items)
               })
               .catch(error => {
                 if (axios.isAxiosError(error)) {
                    console.log(error?.response?.status +
                          ' : ' + error.message)
                 } else {
                   console.error(error);
                 }
               });
          },
    } //action
})
```

Vuex axios : Todo 목록

2. TodoApp 할 일 목록 :

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

src/components/TodoList.vue

```
<template>
 <transition-group name="list" tag="ul">
   <li v-for="(todoItem, index) in todoItems" :key="index" class="shadow">
    ...
</template>
<script lang="ts" setup>

import { useStore } from "vuex"
import { computed, onMounted } from "vue"

const store = useStore()
const todoItems = computed(() => store.state.todoItems)

onMounted(() => {
    console.log('onMounted...')
    store.dispatch("loadTodoItems")
});
</script>
```

Vuex axios : Todo 삭제

3. TodoApp 할 일 삭제 :

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

src/store/index.ts

```
  actions: {
     removeTodo({commit}, payload) {
            http
            .delete(`/todos/${payload.id}`)
            .then(r => r.data)
            .then(items => {
               commit('setTodoItems', items)
            })
     },
  }
```

src/components/TodoList.vue

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

<script lang="ts" setup>
   const removeTodo = (todoItem: TodoItem) => {
      store.dispatch("removeTodo", todoItem)
   }
</script>
```

Vuex axios : Todo 추가

4. TodoApp 할 일 추가 :

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

src/store/index.ts

```
actions: {
        addTodo({commit}, payload) {
            http
            .post(`/todos`, payload)
            .then(r => r.data)
            .then(items => {
                commit('setTodoItems', items)
            })
        },
}
```

src/components/TodoInput.vue

```
<script lang="ts" setup>
const addTodo = () => {
 if(newTodoItem.value !== ""){
   const todoItemStr = newTodoItem.value
   const itemObj = { completed: false, item: todoItemStr }
   store.dispatch("addTodo", itemObj)
   clearInput()
 }
}
</script>
```

Vuex axios : Todo 완료

5. TodoApp 할 일 완료 :

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

src/store/index.ts

```
actions: {
        toggleTodo({commit}, payload) {
           http
           .put(`/todos/${payload.id}`, payload)
           .then(r => r.data)
           .then(items => {
              commit('setTodoItems', items)
           })
        },
}
```

src/components/TodoList.vue

```
<i class="fas fa-check checkBtn"
                        @click="toggleTodo(todoItem)"></i>

<script lang="ts" setup>
const toggleComplete = (todoItem: TodoItem) => {
  todoItem.completed = !todoItem.completed
  store.dispatch("toggleTodo", todoItem)
}
</script>
```

axios : Todo 모두 삭제

5. TodoApp 할 일 모두 삭제 :

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

src/store/index.ts

```
  actions: {
        clearTodo ({commit}) {
            http
              .delete('/todos')
              .then(r => r.data)
              .then(items => {
                commit('setTodoItems', items)
              })
        },
  }
```

src/components/TodoFooter.vue

```
<span class="clearAllBtn" @click="clearTodo">Clear All</span>

<script lang="ts" setup>
        const clearTodo = () => {
            store.dispatch("clearTodo")
        }
</script>
```

Vuex : Store 모듈화

프로젝트 구조화와 모듈화 방법 #1

- 아래와 같은 store 구조를 어떻게 모듈화 할 수 있을까?

```
 //store/index.ts
import { createStore } from "vuex"

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

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/index.ts

```
export const store = createStore({
    state: {
        todoItems: []
    },
    getters: {

    },
    mutations: {

    }
});
```

src/store/getters.ts

```
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.ts

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

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

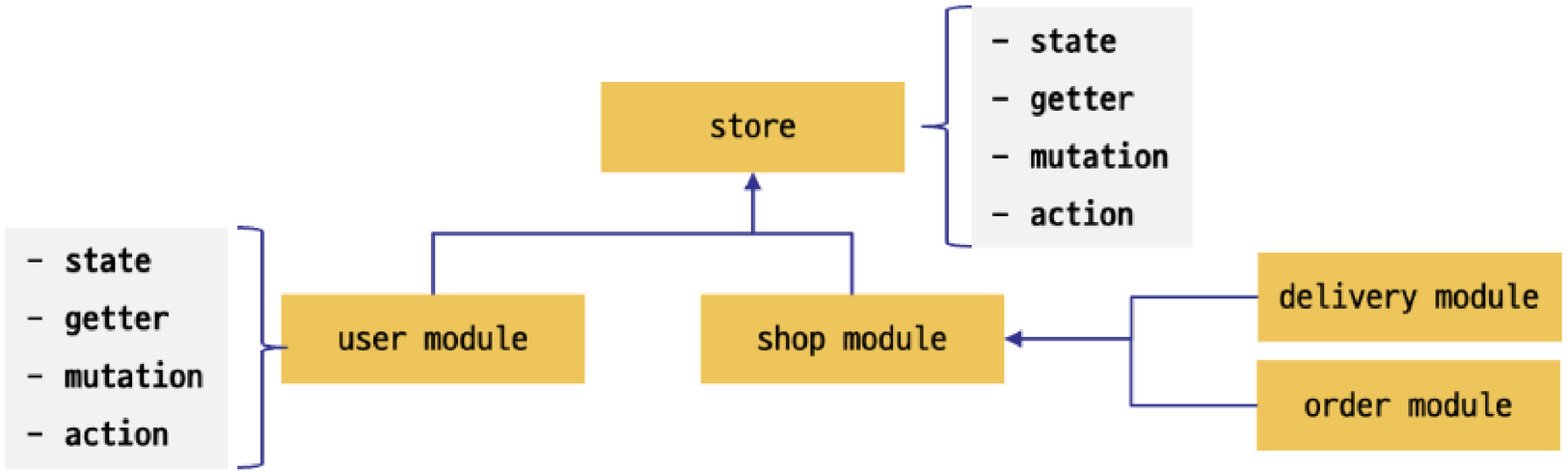
src/store/index.ts

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

export const store = createStore({
    state: { .. },
    mutations: { mutations },
    getters: { getters }
```

Vuex : Store 모듈화

프로젝트 구조화와 모듈화 방법 #2

- App의 규모가 커져서 1개의 store로는 관리하기 힘들 때 modules 속성 사용
- module이란 state, mutations, actions, getters를 갖는 store의 하위 객체를 의미한다.



Vuex : Store 모듈화

프로젝트 구조화와 모듈화 방법 #2

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

```
//store/index.ts
import { createStore } from "vuex"
import { moduleTodo, ModuleTodoState } from './modules/moduleTodo';
export interface RootState {
    moduleTodo: ModuleTodoState
}

export const store = createStore({
    modules: {
        moduleTodo
    }
})
//moduleTodo.ts
export const moduleTodo: Module<ModuleTodoState, RootState> = {
    namespaced: true,
    state: () => ({
        todoItems: []
    }),
    mutations: { },
    getters: { },
    actions: { }
}
```

Vuex : Store 모듈화

2. Store 모듈화 #2

1) store 디렉토리 아래에 modules 디렉토리를 생성하고 moduleTodo.ts를 생성한다.

2) store/index.ts 내의 state, getters, mutations 속성의 내용을 moduleTodo.ts로 이동시킨다.

3) index.ts는 모든 타입들과, 각각의 Store들을 한 곳에 모아주는 역할을 담당한다.

4) RootState는 Store의 서로 다른 모듈 간 state를 공유할 수 있도록 한 곳에 모아두는 역할을 한다.

src/store/index.ts

```
import { createStore } from "vuex"
import { moduleTodo, ModuleTodoState } from './modules/moduleTodo'

export interface RootState {
    moduleTodo: ModuleTodoState
}

export const store = createStore({
    modules: {
        moduleTodo
    }
})
```

https://codeburst.io/vuex-and-typescript-3427ba78cfa8

Vuex : Store 모듈화

2. Store 모듈화 #2

vuex에서 제공하는 Module<S,R> 인터페이스의 첫 번째 S Generic은 본인의 State interface를 넣어준 것이고,

두 번째 R 제네릭은 모든 모듈 타입을 가지고 있는 RootState를 넣어준 것이다.

src/store/modules/moduleTodo.ts

```
import { Module } from "vuex"
import { RootState } from "../index"
import TodoItem from "@/types/TodoItem"
import http from "@/common/http-common"

export interface ModuleTodoState {
    todoItems: TodoItem[];
}

export const moduleTodo: Module<ModuleTodoState, RootState> = {
    namespaced: true,
    state: () => ({
        todoItems: []
    }),
    mutations: { },
    getters: { },
    actions: { }
}
```

https://gist.github.com/mysoyul/74992a29be60ec2fb0f297154d0220b9

Vuex : Store 모듈화

2. Store 모듈화 #2

모듈화에 namespace가 추가 되었으므로 아래와 같이 수정해야 합니다. (TodoInput.vue 와 TodoFoot.vue 도 수정해야 함)

src/components/TodoList.vue

```
<script lang="ts" setup>
import { computed, onMounted } from "vue";
import TodoItem from '@/types/TodoItem';
import { useStore } from "vuex"

const store = useStore()
const todoItems = computed(() => store.state.moduleTodo.todoItems)

onMounted(() => {
    store.dispatch("moduleTodo/loadTodoItems")
});

const removeTodo = (todoItem: TodoItem) => {
   store.dispatch("moduleTodo/removeTodo", todoItem)
};

const toggleComplete = (todoItem: TodoItem) => {
   todoItem.completed = !todoItem.completed
   store.dispatch("moduleTodo/toggleTodo", todoItem)
};
</script>
```

vue-cli : Mode

Modes 와 Environment Variables

vue cli 에는 기본적으로 3개 모드가 있다.

: 1) development, 2) production, 3) test

기본 모드 이외에 사용자가 정의한 모드를 추가 할 수 있다.

```
 //package.json
"scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint",
    // local 로컬 모드 추가
    "local": "vue-cli-service serve --mode local",
    // 사용자 정의 모드 생성
    "mymode": "vue-cli-service serve --mode mymode",
},
```

> npm run local

> npm run mymode

development is used by vue-cli-service serve

> npm run serve

production is used by vue-cli-service build

> npm run build

test is used by vue-cli-service test:unit

> npm run test

vue-cli : Env Variable

Modes 와 Environment Variables

모드명에 맞춰 환경 변수 파일 생성

: package.json 과 같은 위치 (root) 에 두어야 한다.

기본모드 이외에 사용자가 정의한 모드를 추가 할 수 있다.

```
.env                # loaded in all cases
.env.local          # loaded in all cases, ignored by git
.env.[mode]         # only loaded in specified mode
.env.[mode].local   # only loaded in specified mode, ignored by git
```

각 모드별로 생성한 파일 안에 필요한 환경변수를 추가하면 된다.

환경 변수는 process.env 객체로 접근 가능하다.

기본 변수가 아닌 사용자가 정의한 변수는 VUE_APP_ prefix 키워드를 추가해야 인식 가능하다.

VUE_APP_[사용자 지정]

```
VUE_APP_TITLE=My App
VUE_APP_SECRET_CODE=some_value
```

console.log(process.env.VUE_APP_SECRET_CODE)

vue-cli : Env Variable

Modes 와 Environment Variables

.env.development

```
VUE_APP_TITLE=개발 모드
VUE_APP_APIURL=http://localhost:4500/api
```

.env.production

```
VUE_APP_TITLE=운영 모드
VUE_APP_APIURL=http://localhost:4500/api
```

src/common/http-common.ts

```
import axios from "axios"

const APIURL = process.env.VUE_APP_APIURL

const apiClient = axios.create({
  baseURL: APIURL, //"http://localhost:4500/api",
  headers: {
    "Content-type": "application/json",
  },
})
export default apiClient
```

src/components/TodoHeader.vue

```
<template>
    <div>
        <header>
            <h1>TODO it! ({{ mode }})</h1>
        </header>
    </div>
</template>

<script lang="ts">
export default {
    setup() {
        const mode = process.env.VUE_APP_TITLE
        return {mode}
    }
}
</script>
```

> npm run serve

> npm run build

Deployment https://cli.vuejs.org/guide/deployment.html

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 추가 해주는 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 설치

1. Vue Router 설치하기

```
npm install vue-router
```

2. Vue Router 코드 자동 생성

```
vue add router
```

? Use history mode for router? (Requires proper server setup for index fallback in production) Yes

App.vue 의 코드를 components/HelloWorldd.vue 로 이동시킨다.

Vue Router

- 라우터 객체 생성

main.ts : 뷰 인스턴스 생성 객체에는 router 속성이 있다. 뷰 라우터를 사용하려면 이 속성으로 Router 객체를 전달 해야 한다.

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

src/main.ts

```
import { createApp } from 'vue'
import App from './App.vue'
import { store } from "./store"
import router from './router'

createApp(App)
    .use(router)
    .use(store)
    .mount('#app')
```

src/router/index.ts

```
import { createRouter, createWebHistory, RouteRecordRaw } from 'vue-router'
import HomeView from '../views/HomeView.vue‘

const routes: Array<RouteRecordRaw> = [
  { path: '/', name: 'home', component: HomeView },
  { path: '/about', name: 'about',
    component: () => import('../views/AboutView.vue') }
]
const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})
export default router
```

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 사용하는 것을 권장한다.

✓ History 모드에서는 주소 체계가 달라서 <a> 태그를 사용할 경우 모드 변경 시 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>
  <nav>
     <router-link to="/">Home</router-link> |
     <router-link to="/about">About</router-link> |
  </nav>
   <router-view/>
</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(PostList)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/PostList.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>
```

PostList.vue

PostNew.vue

PostDetail.vue

Vue Router

- children 속성

: 중첩된 라우터는 children 속성으로 하위 라우터를 정의할 수 있다.

/posts 경로를 포함한 하위 경로인 /posts/new와 /posts/:id을 children 옵션으로 설정한다.

- 네비게이션 메뉴에 추가

: 생성한 라우터 링크를 루트 컴포넌트 네비게이션 메뉴에 추가한다.

src/router/index.ts

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

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

src/App.vue

```
<template>
   <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">New Post</router-link>
   </nav>
   <router-view/>
</template>
```

Vue Router

- 동적 라우터 매핑

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

/posts/1, /posts/2

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

src/router/index.ts

```
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: {{route.params.id}}
      Post Detail page</h1>
  </div>
</template>
<script lang="ts" setup>
import { useRoute } from "vue-router"

const route = useRoute();
</script>
```

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>
   <nav>
      <router-link to="/">Home</router-link>
   </nav>
</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.

- 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.

: Route 객체는 라우팅 변경이 일어날 때 마다 호출된다. 따라서 watch 에서 감시하고 있다가 변경되면 즉시 fetchData() 함수를 호출하는 로직을 추가하자.

Vue Router

- Store 모듈 : modulePost.ts

src/types/Post.ts

```
interface Post {
    id: number,
    text: string
}
export default Post
```

src/store/modules/modulePost.ts

```
import { Module } from "vuex"
import { RootState } from "../index"
import Post from "@/types/Post"
import http from "@/http-common"

export interface ModulePostState {
  posts: Post[]
  post: Post
}

export const modulePost: Module<ModulePostState, RootState> = {
    namespaced: true,
    state: () => ({
        posts: [],
        post: { id: 0, text: '' }
    }),
    mutations: { },
    getters: { },
    actions: { }
}
```

Vue Router

- Store 모듈 : modulePost.ts

src/store/modules/modulePost.ts

```
export const modulePost:
 Module<ModulePostState, RootState> = {
    namespaced: true,
    state: () => ({
        posts: [],
        post: { id: 0, text: '' }
    }),
   mutations: {
    setPosts(state, items) {
      state.posts = items;
    },
    setPost(state, item) {
      state.post = item;
    },
  },
  getters: {
    getPosts(state) {
      return state.posts;
    },
    getPost(state) {
      return state.post;
    },
  },
```

https://gist.github.com/mysoyul/4cdacf424a7fe67295f9c5f6c6ada569

src/store/modules/modulePost.ts

```
 actions: {
    loadPosts({ commit }) {
      http
        .get(`/posts`)
        .then((res) => res.data)
        .then((items) => commit("setPosts", items))
        .catch((err) => console.error(err));
    },
    loadPost({ commit }, payload) {
      http
        .get(`/posts/${payload.id}`)
        .then((res) => res.data)
        .then((item) => commit("setPost", item))
        .catch((err) => console.error(err));
    },
    removePost({ commit }, id) {
      http
        .delete(`/posts/${id}`)
        .then((res) => res.data)
        .then((items) => commit("setPosts", items))
        .catch((err) => console.error(err));
    },
    addPost({ commit }, payload) {
      http
        .post(`/posts`, payload)
        .then((res) => res.data)
        .then((items) => commit("setPosts", items))
        .catch((err) => console.error(err));
    },
  }
}
```

Vue Router

- Store 모듈 : store/index.ts

src/store/index.ts

```
import { createStore } from "vuex"
import { moduleTodo, ModuleTodoState } from './modules/moduleTodo'
import { modulePost, ModulePostState } from './modules/modulePost'

export interface RootState {
    moduleTodo: ModuleTodoState
    modulePost: ModulePostState
}

export const store = createStore({
    modules: {
        moduleTodo,
        modulePost
    }
})
```

Vue Router

- PostList (리스트)

src/views/PostList.vue

```
<template>
  <div>
    <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 }}] {{ summary(post.text) }}
     </router-link>
    </div>
    <router-view></router-view>
  </div>
</template>
```

https://gist.github.com/mysoyul/e9724854a6a099958e20f8af1810d614

src/views/PostList.vue

```
<script lang="ts" setup>
import { useStore } from "vuex"
import { ref, onBeforeMount, computed } from "vue"

const store = useStore()

const loading = ref(false)
const posts = computed(() => store.state.modulePost.posts)

onBeforeMount(() => {
  fetchData();
});

const fetchData = () => {
  loading.value = true;
  store.dispatch("modulePost/loadPosts").then(() => {
    loading.value = false;
  });
};

const summary = (val: string) => {
  if (typeof val === "string") {
    return val.substring(0, 20) + "...";
  }
  return val;
};
</script>
```

Vue Router

- PostDetail (상세정보) #1

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>
        <button v-on:click="removePost(post.id)">
            Delete
        </button>
    </div>
  </div>
</template>
```

src/views/PostDetail.vue

```
<script lang="ts" setup>
import { ref, onBeforeMount, computed, watch } from "vue";
import { useRouter, useRoute } from "vue-router"
import { useStore } from "vuex"

const router = useRouter();
const route = useRoute();
const store = useStore();
const loading = ref(false);
const post = computed(() => store.state.modulePost.post);

onBeforeMount(() => {
  fetchData();
});

const removePost = (id: number) => {
  console.log('-----------------removePost' + id)
  store.dispatch("modulePost/removePost", id);
  router.push("/posts");
};
```

https://gist.github.com/mysoyul/b820c78f6e3fe83a635fa221d47addb9

Vue Router

- PostDetail (상세정보) #2

src/views/PostDetail.vue

```
const fetchData = () => {
  loading.value = true;
  if(route.params.id) {
    store.dispatch("modulePost/loadPost", { id: +route.params.id }).then(() => {
      loading.value = false;
    });
  }
};

watch(() => route.params.id, fetchData)
</script>
<style scoped>
button {
  margin: 1rem 0;
}
</style>
```

Vue Router

- PostNew (등록) #1

src/views/PostNew.vue

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

src/views/PostNew.vue

```
<script lang="ts" setup>
import { ref, computed, defineEmits } from "vue";
import { useRouter } from "vue-router"
import { useStore } from "vuex"

const store = useStore()
const router = useRouter()

const isSaving = ref(false)
const inputTxt = ref('')

const emit = defineEmits(["input:post"])

const handleInput = (event: InputEvent) => {
  const value = (event.target as HTMLInputElement).value
  if (!value) return
  emit("input:post", value)
  inputTxt.value = value
}
```

https://gist.github.com/mysoyul/4db6c294bcba01d8f0795550a5fa1f15

Vue Router

- PostNew (등록) #2

src/views/PostNew.vue

```
const btnTxt = computed(() => isSaving.value ? 'Saving...' : 'Save')
const disabled = computed(() => isSaving.value)

const onSubmit = () =>{
  isSaving.value = true
  const postObj = { text: inputTxt.value };
  store.dispatch('modulePost/addPost', postObj).then(() => {
        isSaving.value = false;
        inputTxt.value = '';
        router.push('/posts');
      });
}
</script>
<style scoped>
input {
  margin: 1rem 0;
}
</style>
```

감사합니다.