Vue.js 2.0

Vue.js - 뷰는 **사용자 인터페이스(User Interface)**를 개발하기 위한 프레임워크이다. 다양한 지시자(Directive)와 컴포넌트(Component)들을 지원하며, Angular나 React 보다 훨씬 가볍고 빠르게 우리가 원하는 화면을 쉽게 개발 할 수 있는 여러가지 라이브러리를 제공하고 있다.

뷰를 설치 하기 위한 방법이다.

독립 실행 버전(단순 html 에서만 사용하기)

1) vuejs.org에서 다운로드 받기.
   ➔ 단순히 <script>태그를 이용하여 .js 파일을 불러오면 된다. 개발용과 배포용 두가지 버전이 있는데 개발시에는 반드시 개발용 버전을 사용해야 한다.
2) CDN에서 가져오기.
   ➔ Vue에서 공식적으로 제공하는 CDN이 있지만 항상 최신버전을 유지하기 위해서 `<script src="https://unpkg.com/vue"></script>` 에서 스크립트를 불러와 사용하는 방법도 있다.

NPM 사용

vue.js를 사용해 대규모 어플리케이션을 제작할 때 권장된다. 모든 구조화가 자동으로 이루어져 있고 Webpack과 Browserify 같은 CommonJS 모듈들과 잘 어울린다.

1) 최신 안정 버전
   ➔ npm install vue
2) 최신 안정 버전 + CommonJS Spec 준수
   ➔ npm install vue@csp
3) 개발 빌드(Git Hub 에서 설치)
   ➔ npm install vuejs/vue#dev

간단하게 Vue 의 CDN 을 이용해서 Vue 어플리케이션을 만들어 보자. html 페이지의 간단한 설정 만으로도 Vue 를 사용 할 수 있다. Vue.js 를 이용하면 jQuery 같은 자바스크립트보다 훨씬 더 시간을 아낄 수 있다는 것을 알 수 있다. 먼저 **vue_start01.html** 을 만들고 **Vue 인스턴스를 만들어 사용자의 DOM 을 Vue 가 알아 차릴 수 있는 코드**를 작성하자.

vue01_start1.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Hello VUE</title>
</head>
<body>

    <div id="app">
        <h1>안녕 Vue</h1>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el:'#app',
    })
</script>
</html>
```

body 태그가 끝나고 나서 Vue 의 CDN 을 이용해 라이브러리를 불러오는데, 이 때 src 에 CDN 이 들어가고 바로 밑에 Vue 인스턴스를 만든 것이 보인다. **el 속성**은 Vue 에서 DOM 엘리먼트를 지정하는 속성이며, 이 코드에서는 **id 가 app('#app')인 div 를 Vue 를 작동시킬 공간(컨테이너)** 이라고 생각할 수 있다. 참고로 Vue 는 el 로 지정한 DOM 엘리먼트 밖에 있는 것들은 알지 못한다. 이제부터 Vue 를 이용해 data 를 DOM 으로 넘기는 코드를 만들어보자. vue_start01.html 를 약간 수정하자.

```
...
    <div id="app">
        <h1>{{ message }}</h1>
    </div>
...
<script>
    var vue_data = { message : '안녕하세요 Vue에서 #app에 보내는 메시지입니다.'};
    new Vue({
        el:'#app',
        data: vue_data 또는 { message : '안녕하세요 Vue에서 #app에 보내는 메시지입니다.' }
    })
</script>
...
```

Vue 인스턴스를 생성할 때 전달되는 자바스크립트 객체의 속성에 data 속성을 추가하면서 DOM으로 전달한 데이터를 넣어둔 vue_data 객체를 전달 시킨다. 이 때 vue_data 객체에는 message 속성이 들어가 있는 것을 알 수 있는데, message 속성에 들어 있는 값이 #app의 h1 태그 안에 있는 {{ message }}에 그대로 표현이 되는 것이 보일 것이다.

자바스크립트에 있는 객체 내부의 데이터를 DOM에서 표현하기 위해 쓰인 {{ message }}는 **스크립트 코드**로써 이런 {{ }} 안에 들어 있는 모든 것들을 **바인딩 표현식(Binding Expression)**이라고 한다.

이어서 input 박스에 들어있는 입력 내용을 Vue 에서 바꾸기 위해 디렉티브(Directive)를 사용함과 동시에 input 박스의 내용이 바뀌면 message 값을 동적으로 변경하는 양방향 바인딩에 관한 예제이다. vue01_start2.html 을 만들고 작성하자.

vue01_start2.html

```
<html>
<head>
    <meta charset="UTF-8">
    <title>디렉티브와 양방향 바인딩</title>
</head>
<body>
    <div id="app">
        <h2>{{ message }}</h2>
        <input v-model="message">
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el:'#app',
        data: { message : 'Hello Vue'}
    })
</script>
</html>
```

예제를 실행해 보면 먼저 Vue 객체에 있는 data 의 message 가 h2 와 input box 에 표현된 것을 알 수 있으며, input 의 메시지를 바꾸면 h2 의 message 도 실시간으로 바뀌는 것을 알 수 있다. **v-model 은 양방향 바인딩을 지원하는 지시자(Directive)**로써, Vue 에게 해당 input 이 어느 변수와 바인딩 됐는지 알려주는 역할을 한다. 양방향 데이터 바인딩은 뷰에서 모델의 값(데이터 – 여기서는 message) 모든 내용이 최신으로 유지된 다는 사실을 알 수 있다. 지시자와 데이터 바인딩은 나중에 알아본다.

만약에 jQuery 를 사용하여 위의 코드를 작성 한다면 어떻게 작성해야 하겠는가? 사용자가 키를 눌렀다가 뗄 때마다(keyup 이벤트) 입력을 해 준 다음 id 나 class 선택자를 이용해 원하는 엘리먼트를 선택하여 값을 넣어 줘야 할 것이다.

```
<html>
<head>
    <meta charset="UTF-8">
    <title>안녕 jQuery</title>
</head>
<body>
    <div id="app">
        <h1>안녕 Vue</h1>
        <input id="message">
    </div>
</body>
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
<script>
    $('#message').on('keyup', function(){
        var message = $('#message').val();
        $('h1').text(message);
    })
</script>
</html>
```

물론 jQuery 를 이용해서도 만들어 낼 수 있지만 Vue 를 사용한 것이 조금 더 깔끔하고 많은 기능을 지원하는데 있어서 불편함이 없어 보인다.

Vue 디렉티브(지시자 – Directive)

디렉티브란 라이브러리에서 DOM 엘리먼트가 무언가를 수행하도록 지시하는 특수한 속성이다. 제어문(조건문, 반복문) 같이 사용되며 조건에 따라 DOM 엘리먼트에서 표시 해야 할 내용을 조절 할 수 있게 해준다. 간단한 몇가지 디렉티브이다.

- **v-show** : 조건에 따라서 엘리먼트 표시
- **v-if** : v-show 대신 사용. **v-else-if** 와 **v-else** 사용 가능
- **v-for** : 배열 내부에 있는 항목들을 출력 할 때 사용

v-show 디렉티브

먼저 v-show 디렉티브는 조건에 따라서 엘리먼트를 표시한다. 만약 Vue 인스턴스의 data 란에 존재하는 message 의 값이 있으면 표시하고, 없으면 표시 하고 싶지 않을 때의 코드는 다음과 같이 작성하면 된다.

```
...
<button v-show="message">My Button</button>
...
new Vue({
        el : '#app',
        data : { message : '' }
    });
...
```

Vue 인스턴스의 data 에 있는 message 가 있으면 버튼을 표시하고, message 가 없으면 버튼을 표시하지 않는데, 이 때 **v-show 의 특징 중 하나는 display:none 으로 css 의 스타일이 바뀌는 것** 뿐, 실제 엘리먼트가 삭제되거나 추가되지 않는다. 다음은 v-show 를 이용한 예이다. Vue 인스턴스에 있는 message 변수가 textarea 에 바인딩 되어있고 button 에 v-show 디렉티브가 적용되어 있다. 이는 textarea 에 글이 없으면 버튼이 보이지 않고(**display:none**), textarea 에 글이 있으면 button 이 보이게 될 것이다.

vue02_directive1.html

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Directives</title>
</head>
<body>
    <div id="app">
        <textarea v-model="message"></textarea>
        <pre>{{ $data }}</pre> <!-- {{ $data }} : JSON 형태로 data 속성을 표현해 준다.-->

        <!-- message가 존재 한다면 이 엘리먼트가 보인다. textarea에 문자열이 하나도 없으면 -->
        <!-- 이 버튼의 display는 none(display:none)으로 설정된다.-->
        <button v-show="message">
            데이터가 입력 되면 보입니다
        </button>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el : '#app',
        data : { message : '' }
    });
</script>
</html>
```

v-if, v-else-if, v-else 디렉티브

먼저 v-show 디렉티브와의 차이점을 설명하자면, v-show 디렉티브는 조건이 맞지 않으면 엘리먼트 객체는 그대로 유지한 채 사용자의 눈에 보이지만 않게 하는 것이라면(display:none) **v-if, v-else-if, v-else 는 아예 엘리먼트 자체를 만들지 않는다**는 것이다. 또한 v-if 디렉티브는 v-else-if 와 v-else 도 같이 이용 할 수 있다는 것이 차이점이 될 수 있다. 마지막으로 **여러가지 엘리먼트를 묶어서 한꺼번에 토글**(display <-> non-display)해야 하는 경우 v-if 와 <template></template> 태그를 이용하여 손쉽게 만들어 낼 수도 있다.

```
...
<template v-if="!message"> <!--  message 가 없으면.. -->
    <h1> message 가 없습니다. </h1>
    <p>메시지를 입력 하세요!</p>
</template>
<h2 v-else>{{message}}</h2>
...
```

위의 예시가 바로 v-if 와 v-else 를 사용한 예시이다. v-if 의 !message 조건은 message 가 없을 때 표현 해야 할 template 태그를 지정 하였고, v-else 에는 message 가 있을 경우 표시할 내용을 표시 해 놓았다. 주의 해야 할 점은 v-else-if 나 v-else 는 반드시 v-if 뒤에 와야 한다는 것이다.

vue02_directive2.html

```
... html, head 생략 ...
<body>
    <div id="app">
        <template v-if="!message">
            <h1>message가 없습니다.</h1>
            <p>메시지를 입력하세요.</p>
        </template>
        <h2 v-else>{{message}}</h2>
        <textarea v-model="message"></textarea>
        <pre>{{ $data }}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '#app',
        data: {
            message: 'hello vue'
        }
    });
</script>
</html>
```

그렇다면 v-if 와 v-show 를 언제 사용해야 할까? v-if 는 엘리먼트를 삭제/생성 하는 작동을 하고, v-show 는 단순히 엘리먼트를 미리 만들어 놓고 display 상태만 바꾸는 것이기 때문에 **화면의 변화가 잦을 때는 v-show** 가 유리하고, 조건이 자주 변경되지 않아 **화면이 자주 변경될 가능성이 낮은 경우에는 v-if** 가 유리하다.

[연습문제] input box를 두개 만들어 하나는 성별을 입력(female, male) 받고, 다른 하나는 이름을 입력 받을 수 있게 한다. 이 때 성별(gender)를 입력 받는 input box에 female이 입력되면 Hello, Miss. {{name}}이, male이 입력되면 Hello, Mr. {{name}}이 출력되도록 하시오.

연습문제 정답

```
... html, head 생략 ...
<body>
    <div id="app">
        <h1 v-if="gender === 'female' || gender === 'male'">
            Hello, {{gender === 'female' ? 'Miss' : 'Mr'}}. {{name}}
        </h1>
        <h1 v-else>성별이 입력되지 않거나 female 또는 male이 아닙니다.</h1>
        <h3>성별을 입력 하세요(female 또는 male)</h3>
        <input type="text" v-model="gender">
        <h3>이름을 입력 하세요</h3>
        <input type="text" v-model="name">
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '#app',
        data: {
            name: '',
            gender: ''
        }
    });
</script>
...
```

v-for 디렉티브 [리스트 렌더링]

먼저 v-for 는 **특정 엘리먼트나 데이터를 반복해서 나타내고 싶을 때 사용 할 수 있는 디렉티브**이다. v-for 를 이용하여 다음과 같은 작업들을 진행 할 수 있게 된다. 이 때 **배열에 들어있는 데이터를 반복하면서 화면에 표시해 주는 것을 리스트 렌더링**이라고 한다.

- 배열 기반의 리스트 출력
- 템플릿 반복
- 객체의 프로퍼티를 순회

v-for 에 숫자를 전달하여 사용하는 방식이다. 일반적인 for 문과 비슷하게 사용이 가능하다. 해당 예제부터 Bootstrap 을 사용한다.

vue03_list_rendering1.html

```
<body>
    <div class="container">
        <h1>v-for로 구구단 </h1>
        <ul class="list-group">
            <!-- 1 ~ 10 까지 반복함 -->
            <li v-for="i in 9" class="list-group-item">
                {{ i }} 곱하기 4 는 {{ i * 4 }}
            </li>
        </ul>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container'
    });
</script>
</html>
```

다음은 단순히 값만 들어있는 배열을 순회하여 출력하는 방식이다.

vue03_list_rendering2.html

```
<body>
    <div class="container">
        <h1> v-for로 배열 순회하기</h1>
        <ul class="list-group">
            <li v-for="story in stories" class="list-group-item">
                {{ story }}
            </li>
        </ul>
        <pre>{{ $data }}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            stories: [
                "첫 번째 이야기 입니다.",
                "두 번째 이야기 입니다.",
                "세 번째 이야기 입니다."
            ]
        }
    });
</script>
</html>
```

Vue 인스턴스에 data 속성에 stories 라고 이름이 붙은 배열을 만들고 하나씩 차례대로 표시하는 예제이다. v-for 부분에 stories 배열에서 원소들을 하나씩 꺼내서 표시하고 있다.

다음은 객체가 들어있는 배열을 순회하면서 화면에 출력하는 방식이다. 위의 예제와 동일한 방식이지만 story 를 그대로 출력 하는 것이 아닌, stories 를 v-for 를 이용해 순회하면서 가져온 story 객체 안에서 속성을 표시하는 예제이다.

vue03_list_rendering3.html

```
<body>
    <div class="container">
        <h1> v-for로 객체 배열 순회하기</h1>
        <ul class="list-group">
            <li v-for="story in stories" class="list-group-item">
                <strong>{{ story.writer }}</strong>가 {{ story.plot }} 이라고 했어요...
            </li>
        </ul>

        <ul class="list-group">
            <li v-for="(story, index) in stories" class="list-group-item">
                <!-- value, index 형태로 지정하기 -->
                {{ index }}. {{ story.writer }} {{ story.plot }}
            </li>
        </ul>
        <pre>{{ $data }}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            stories: [{
                    writer: 'A',
                    plot: "첫 번째 이야기 입니다."
```

```
            },
            {
                writer: 'B',
                plot: "두 번째 이야기 입니다."
            },
            {
                writer: 'C',
                plot: "세 번째 이야기 입니다."
            }
        ]
    }
});
</script>
</html>
```

배열에 들어있는 객체들을 v-for 를 이용하여 순회하고 있는데, 위의 예제에서는 두 가지 방식을 보이고 있다.

첫 번째 v-for 는 단순히 stories 배열에서 요소들을 하나씩 꺼낸 뒤 출력 하는 코드이다. 이 때 꺼내어 지는 요소들은 전부 자바스크립트 객체 형태를 띄고 있는 것이 확인된다.

두 번째 v-for 는 story in stories 형태로 데이터를 꺼내는 것이 아닌 (story, index) 형태로 데이터를 꺼내는 것이 확인된다. 이 때 두 번째 v-for 에서는 배열의 요소(자바스크립트 객체)와 인덱스 값이 동시에 꺼내어 지게 된다.

마지막은 배열이 아닌 단순 객체를 순회하는 방법이다.

vue03_list_rendering4.html

```
<body>
    <div class="container">
        <h1> v-for로 객체 프로퍼티 순회하기</h1>
        <ul class="list-group">
            <!-- 하나의 객체에서 값만 가져오기 -->
            <li v-for="value in story" class="list-group-item">
                {{ value }}
            </li>
        </ul>
        <ul class="list-group">
            <!-- 객체 하나에서 value, key, index 가져오기 -->
            <li v-for="(value, key, index) in story" class="list-group-item">
                {{ index }} : {{ value }} : {{ key }}
            </li>
        </ul>
        <pre>{{ $data }}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            story: {
                plot: "Someone ate my chocolate...",
                writer: 'John',
                upvotes: 47
            }
```

```
        }
    });
</script>
</html>
```

객체의 프로퍼티를 순회 할 때도 v-for 를 사용할 수 있다. 배열의 index 를 표시할 수 있는 것이 이전 예제에서 확인 됐는데, 객체를 순회 할 때도 똑같이 할 수 있다. 이 때는 인덱스와 값은 물론이고 key 값 까지 객체에서 꺼내는 것이 가능하다.

[연습문제] 이름, 키, 몸무게, 눈 색깔을 저장하는 객체와 그 객체를 저장하는 배열을 만들어 출력하되, v-for 구문을 이용하여 출력하시오. 이 때 출력되는 내용들은 전부 index : key = value 형태로 출력

연습문제 정답

```
<div class="container">
        <h1> v-for 연습 문제</h1>
        <ul class="list-group">
            <li class="list-group-item" v-for="(person, index) in persons">
                <ul class="list-group">
                    <li class="list-group-item" v-for="(info, key, index) in person">
                        {{ index }} : {{ key }} = {{ info }}
                    </li>
                </ul>
            </li>
        </ul>
        <pre>{{ $data }}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
```

```
        persons: [{
            name: '소민호1',
            height: 154,
            weight: '비밀임',
            eyeColor: 'brown'
        }, {
            name: '소민호2',
            height: 175,
            weight: '몰라',
            eyeColor: 'blue'
        }, {
            name: '소민호3',
            height: 180,
            weight: '안알려줄껀데?',
            eyeColor: 'black'
        }, ]
    }
});
```

상호작용

v-on 을 활용한 이벤트 처리

Vue 에서는 HTML 에서 발생되는 여러가지 이벤트를 손쉽게 사용해 데이터를 조작 할 수 있다. 기존에 자바스크립트나 제이쿼리를 이용하면 직접 엘리먼트 객체를 생성하여 이벤트 부여(on 메소드)를 해야 했지만, Vue 를 이용하면 v-on 프로퍼티만 사용하여 손쉽게 이벤트를 처리 할 수 있게 된다. 다음 예제는 버튼을 클릭 했을 때 upvotes 라는 변수의 값을 하나 증가 시키는 예제 이다.

```
vue04_event1.html
<body>
    <div class="container">
        <button v-on:click="upvotes++">
            Upvote! {{ upvotes }}
        </button>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            upvotes: 0
        }
    });
</script>
```

v-on 지시자는 DOM 엘리먼트에 이벤트 리스너를 등록시키는 역할을 한다. 위의 예제는 버튼을 클릭 했을 때(v-on:click)을 예시로 들고 있는 모습이다. 버튼을 클릭 했을 때 **특정한 메소드를 실행 시키고 싶다면 Vue 인스턴스에 methods 필드를 추가** 시켜 주면 된다.

vue04_event2.html

```
<div class="container">
        <button v-on:click="doUpvote">
            Upvote! {{ upvotes }}
        </button>

        <button @click="doUpvote">
            축약형 Upvote! {{ upvotes }}
        </button>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            upvotes: 0
        },
        //vue에서 작동할 method 만들기
        methods: {
            doUpvote : function () {
                //this는 Vue 인스턴스
                this.upvotes++;
            }
        }
    });
</script>
```

methods 필드를 추가해 사용할 메소드를 등록 시키는 모습이다. doUpvote 라는 메소드를 만들었으며, 이 메소드는 this.upvotes (data 필드에 있는 upvotes 변수)를 하나씩 증가 시키는 역할을 하게 된다.

그 아래쪽을 보면 @를 활용해 이벤트가 지정되어 있는 것이 확인되는데, @는 이벤트를 축약해서 쓸 때 사용하게 된다. 즉 v-on 을 축약해서 @로 쓴다고 이해 할 수 있다. 따라서 다음 두 가지 예시는 완벽하게 같은 동작을 하게 될 것이다.

```
- @click="doUpVote"
- v-on:click="doUpVote"
```

이벤트 한정자

한정자란, 이벤트를 사용하면서 자주 사용되는 기능들을 간단히 프로퍼티 형태로 호출을 한번 하면 간단하게 여러 이벤트 관련된 처리를 할 수 있는 것이다. 계산기 앱을 만드는데 단순히 2 개의 입력필드와 연산을 선택하는 드롭다운만 하나 포함한 형태의 폼을 사용할 것이다. 다음 코드를 보면 문제는 없어 보이지만, 작동이 되지 않는다.

vue04_event3.1.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>바보 계산기</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1> 계산기 </h1>
        <form class="form-inline">
            <!-- 입력을 숫자로 자동으로 파싱해주는 특별한 한정자 number -->
            <input v-model.number="a" class="form-control">
            <select v-model="operator" class="form-control">
                <option>+</option>
                <option>-</option>
                <option>*</option>
```

```
                <option>/</option>
            </select>
            <input v-model.number="b" class="form-control">
            <button type="submit" @click="calculate" class="btn btn-primary">계산</button>
        </form>
        <h2>Result : {{ a }} {{ operator }} {{ b }} = {{ c }}</h2>
        <pre>{{$data}}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            a: 1,
            b: 2,
            c: null,
            operator: "+"
        },
        methods: {
            calculate: function() {
                switch (this.operator) {
                    case "+":
                        this.c = this.a + this.b;
                        break;
                    case "-":
                        this.c = this.a - this.b;
                        break;
                    case "*":
                        this.c = this.a * this.b;
                        break;
```

```
                case "/":
                    this.c = this.a / this.b;
                    break;
            }
        }
    }
});
</script>
</html>
```

위의 예제를 실행 시켜보면 잘 동작하지 않는 것이 확인된다. 바로 계산 버튼이 submit 타입이기 때문에 form 의 동작을 우선적으로 처리 해야 하기 때문에 원래 우리가 하려던 calculate 함수는 실행이 되지 않기 때문이다.

이때는 간단히 submit 에 지정해 놓은 클릭 이벤트(@click)를 실행하는 함수(calculate)에서 event 를 받아 **preventDefault** 를 실행 해 주면 간단히 이벤트를 멈출 수 있다.

해결 된 것처럼 보이지만 여러가지 문제가 하나 더 있다. 바로 값을 입력 받아내는 input 을 확인 해 보자. 기본적으로 input box 는 문자열로 모든 데이터를 처리한다. 우리가 아무리 숫자를 입력해도 기본적으로 문자열로 처리하는 특징 때문에 우리가 원하는 계산이 이루어지지 않고 문자열이 이어지는 형태로 계산이 될 것이다. 마찬가지로 input box 의 내용들을 숫자로 바꿔 주기 위해 **parseInt** 나 **parseFloat** 같은. 또는 **Number** 등의 문자열을 숫자로 바꿔주는 코드를 작성하면 간단히 해결 할 수 있다.

하지만 위의 preventDefault 나, parseInt, parseFloat 같은 함수를 계속 사용하게 되면 여러모로 코드가 보기 좋지 않다. 따라서 Vue 에서는 **한정자** 라는 것을 지원하여 개발자들이 자주 사용하는 여러가지 기능들을 엘리먼트의 속성처럼 사용 할 수 있도록 하였다.

다음 예제는 한정자를 이용해 원래 이벤트를 멈춤과 동시에(preventDefault) input box 의 내용을 자동으로 숫자화 시켜주는 한정자를 담고 있는 예제이다.

vue04_event3.2.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>prevent 한정자 사용</title>
```

```
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container">
        <h1> 계산기 - preventDefault </h1>
        <form class="form-inline">
            <!-- 입력을 숫자로 자동으로 파싱해주는 특별한 한정자 number -->
            <input v-model.number="a" class="form-control">
            <select v-model="operator" class="form-control">
                <option>+</option>
                <option>-</option>
                <option>*</option>
                <option>/</option>
            </select>
            <input v-model.number="b" class="form-control">
            <!-- .prevent : event 객체의 preventDefault()와 같다 -->
            <button type="submit" @click.prevent="calculate" class="btn btn-primary">Calculate</button>
        </form>
        <h2>Result : {{ a }} {{ operator }} {{ b }} = {{ c }}</h2>
        <pre>{{$data}}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            a: 1,
            b: 2,
```

```
            c: null,
            operator: "+"
        },
        methods: {
            calculate: function(event) {
                //이벤트의 전달을 멈춰주는 preventDefault() 대신
                // @click.prevent를 사용해도 좋다.
                //event.preventDefault();
                switch (this.operator) {
                    case "+":
                        this.c = this.a + this.b;
                        break;
                    case "-":
                        this.c = this.a - this.b;
                        break;
                    case "*":
                        this.c = this.a * this.b;
                        break;
                    case "/":
                        this.c = this.a / this.b;
                        break;
                }
            }
        }
    });
</script>

</html>
```

input box 의 v-model.number 는 input box 에 입력된 내용을 숫자로 바꿔주는 한정자 이고, 클릭 이벤트의 @click.prevent 는 해당 이벤트가 일어났을 때 자동으로 preventDefault()를 호출하여 원래 하려던 이벤트를 멈춰주는 한정자 이다.

계산된 프로퍼티(Computed Properties)

Vue 의 인라인 표현식으로도 충분히 로직을 구현 할 수 있지만, 로직이 복잡해질 경우에는 계산된 프로퍼티를 사용하는 것이 훨씬 좋다. **계산된 프로퍼티는 함수처럼 동작한다. methods 필드를 이용하는 것과 다르게 계산된 프로퍼티는 의존하고 있는(계산된 프로퍼티의 함수에서 사용되고 있는) 요소가 변경 될 때마다 함수가 계속 실행 된다.**

vue04_event4.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>계산된 프로퍼티</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container">
        a = {{a}}, b= {{b}}
        <input v-model.number="a"> <!-- 변수 a 바인딩 -->
        <pre>{{$data}}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            a: 1
```

```
        },
          computed:{
              //계산된 프로퍼티.
              // b 는 a 의 값에 따라서 변화 된다.( b = a + 1)
              b : function(){
                  return this.a + 1;
              }
          }
    });
</script>

</html>
```

새롭게 computed 필드가 추가된 것을 확인 할 수 있다. **b 특성이 바로 함수에 의해 계산 되는 프로퍼티**가 된다. a 의 값이 바뀌면 b 의 값은 +1 이 되는 형태로 값이 설정이 되게 될 것이다.

input box 의 값을 바꿔주면 b 의 값도 바뀌는 것이 확인 될 것이다. 마지막으로 계산된 프로퍼티를 활용한 사칙연산 계산기 이다.

```
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>계산된 프로퍼티를 이용한 계산기</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container">
        <h1> 계산기 - 계산된 프로퍼티</h1>
```

```
        <form class="form-inline" @submit.prevent="calculate">
            <input v-model.number="a" class="form-control" @keyup.13="calculate">
            <select v-model="operator" class="form-control">
                <option>+</option>
                <option>-</option>
                <option>*</option>
                <option>/</option>
            </select>
            <input v-model.number="b" class="form-control" @keyup.enter="calculate">
        </form>
        <h2>Result : {{ a }} {{ operator }} {{ b }} = {{ c }}</h2>
        <pre>{{$data}}</pre>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '.container',
        data: {
            a: 1,
            b: 2,
            operator: "+"
        },
        computed: {
            <!-- Vue 의 인스턴스(this)인 operator 와 a,b 의 값이 바뀔 때마다 해당하는 함수가 실행되면서 c 의 값이 재평가 됨 -->
            c: function () {
                switch (this.operator) {
                    case "+":
                        return this.a + this.b;
                        break;
                    case "-":
```

```
                    return this.a - this.b;
                    break;
                case "*":
                    return this.a * this.b;
                    break;
                case "/":
                    return this.a / this.b;
                    break;
            }
        }
    });
</script>

</html>
```

[실습] 반장선거를 진행 하려고 한다. 이름과 투표수 데이터를 갖고 각각의 후보자들을 투표 할 수 있는 버튼을 각각 만들어 버튼이 눌러지면 해당하는 투표자의 투표수가 오를 수 있도록 해보자. 이 때 가장 높은 투표를 받은 사람 순서대로(내림차순) 정렬이 되도록 만들어 보자. 또한 리셋 버튼을 만들어 버튼이 눌리면 모든 투표수가 초기화 될 수 있도록 만들어 보자.

필터(Filter)

원본 데이터를 변경하지 않고 특정 계산이나 조건 변경으로 인한 필터링된 배열을 표시 해야 할 때가 있을 것이다. 여러가지 예시를 통해 필터링 하는 방법을 알아보자.

조건에 의한 필터링

Alex 와 John 이 쓴 이야기를 각각 보여주는 예제이다. **storiesBy 메소드가 배열 요소를 검사하는 필터링 메소드**가 된다.

vue05_filter1.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>필터 - 작성자별 분류하기</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1>알렉스의 이야기</h1>
        <ul class="list-group">
            <li v-for="story in storiesBy('Alex')" class="list-group-item">
                {{ story.writer }} said "{{ story.plot }}"
            </li>
        </ul>
        <h1>존의 이야기</h1>
        <ul class="list-group">
            <li v-for="story in storiesBy('John')" class="list-group-item">
                {{ story.writer }} said "{{ story.plot }}"
            </li>
```

```
        </ul>
        <pre>{{$data}}</pre>
    </div>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    new Vue({
        el: '.container',
        data: {
            stories: [{
                plot: "새벽 4 시니까 아무말 대잔치가 시작됩니다."
                writer: "Alex"
            }, {
                plot: "곱창 먹고 싶당"
                writer: "John"
            }, {
                plot: "ㄴ r 는 ㄱ r 끔 눈물을 흘린ㄷㅏ...
                writer: "Alex"
            }, {
                plot: "배고프다. 곱창이 날아와 내 입에 박힌다. 걱정마라 다이어트는 낼부터 하면 되니까.",
                writer: "John"
            }]
        },
        methods: {
              //작가에 따라서 필터링을 하는 메소드
              storiesBy: function(writer) {
                  return this.stories.filter(function(story) {
                      return story.writer === writer;
                  })
              }
```

```
        }
    })
</script>
</html>
```

storiesBy 메서드는 글쓴이(writer) 인자를 가지로 해당 작가로 필터링된 배열을 리턴 하게 된다. v-for 디렉티브는 storiesBy 메소드를 이용한 결과물을 순회하며 각각 작가들에 대한 story 만 가지고 표시한다.

계산된 프로퍼티 사용

다음은 **계산된 프로퍼티를 이용하여 더욱더 정확하고 원하는 결과만을 가진 배열을 만들어서 활용** 할 수 있다. 예를 들어, 위의 예제를 약간 변형하여 일정한 투표수를 받은 이야기만 담겨 있는 배열을 만들어 낼 수 있다.

vue05_filter2.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>필터 - computed 활용하여 조건 활용하기</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container">
        <h1>인기 있는 이야기 듣기 - {{ famous.length }} 개의 이야기가 인기 있습니다</h1>
        <ul class="list-group">
            <li v-for="story in famous" class="list-group-item">
                {{ story.writer }} said "{{ story.plot }}" 투표수 : {{ story.upvotes }}
            </li>
        </ul>
        <pre>{{$data}}</pre>
    </div>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    new Vue({
        el: '.container',
        data: {
            stories: [{
                plot: " 새벽 4 시니까 아무말 대잔치가 시작됩니다."
```

```
                writer: "Alex",
                upvotes: 30
            }, {
                plot: "곱창 먹고 싶당"
                writer: "John",
                upvotes: 28
            }, {
                plot: "ㄴr는 ㄱr 끔 눈물을 흘린ㄷㅏ...
                writer: "Alex",
                upvotes: 8
            }, {
                plot: "배고프다. 곱창이 날아와 내 입에 박힌다. 걱정마라 다이어트는 낼부터 하면 되니까.",
                writer: "John",
                upvotes: 28
            }]
        },
        computed: {
            famous: function() {
                return this.stories.filter(function(item) {
                    return item.upvotes > 25;
                });
            }
        }
    })
</script>
</html>
```

계산된 프로퍼티인 famous 필드도 필터 메소드를 이용해 투표수가 총 25 개인 이야기만 걸러서 배열로 만들어 리턴 해 준다. 본문(.container) 에서는 famous 사용을 통해 계산된 프로퍼티의 결과물이 된 배열을 가지로 v-for 디렉티브를 이용해 표시 해 주고 있는 것이 확인된다.

실시간 검색

계산된 프로퍼티를 이용한 Vue 인스턴스에 지정 되어있는 데이터를 **실시간으로 검색**하는 방법이다. 비어있는 query 변수에 바인딩된 input 을 추가해서 stories 배열을 동적으로 필터링 할 수 있다. **includes 함수를 활용하여 문자열 포함 여부를 알아낼 수 있다.**

vue05_filter3.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>필터 - 간단한 실시간 검색</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container">
        <h1>알렉스의 이야기</h1>
        <ul class="list-group">
            <li v-for="story in storiesBy('Alex')" class="list-group-item">
                {{ story.writer }} said "{{ story.plot }}"
            </li>
        </ul>
        <h1>존의 이야기</h1>
        <ul class="list-group">
            <li v-for="story in storiesBy('John')" class="list-group-item">
                {{ story.writer }} said "{{ story.plot }}"
            </li>
        </ul>

        <div class="form-group">
            <label for="query">
```

```
            누구의 글을 찾나요?
        </label>
          <input v-model="query" class="form-control">
    </div>
    <h3>검색 결과 입니다 : </h3>
    <ul class="list-group">
        <li v-for="story in search" class="list-group-item">
            {{ story.writer }} said "{{ story.plot }}"
        </li>
    </ul>
    <pre>{{$data}}</pre>
  </div>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    new Vue({
        el: '.container',
        data: {
            stories: [{
                plot: "나는 오늘 공부를 한다.",
                writer: "Alex"
            }, {
                plot: "나는 내일 뭐하냐",
                writer: "John"
            }, {
                plot: "공부 하기 싫당..",
                writer: "Alex"
            }, {
                plot: "배가 너무 부르다...ㅠㅠ",
                writer: "John"
            }],
```

```
            query: '' //검색에 사용될 query 변수
        },
        methods: {
            //작가에 따라서 필터링을 하는 메소드
            storiesBy: function(writer) {
                return this.stories.filter(function(story) {
                    return story.writer === writer;
                })
            }
        },
        computed: {
              search: function() {
                  var query = this.query;
                  return this.stories.filter(function(story) {
                     //includes 함수를 이용해 문자열 포함여부 리턴함
                      return story.plot.includes(query);
                  });
              }
          }
    })
</script>
</html>
```

결과 정렬

sort 메소드를 이용하여 정렬된 배열을 만들어 낼 수 있다. 먼저 sort 메소드는 배열에 저장되어 있는 요소들을 기준에 맞춰 정렬 할 때 사용한다. 예를 들어 다음과 같이 이용 할 수 있다.

- ArrayObject.sort(function(item1, item2) { return item1.value - item2.value });

위 예시 코드는 sort 메소드를 호출하면서 정렬을 하는 코드인데 콜백 함수 내부의 내용은 다음과 같다.

- item1.value - item2.value < 0 이면 item1 이 item2 보다 앞에 위치
- item1.value - item2.value == 0 이면 위치를 변경 하지 않는다.
- item1.value - item2.value > 0 이면 item2 이 item2 보다 뒤에 위치

만약 item1.value - item2.value 의 결과물에 -1 을 곱하면 오름차순과 내림차순을 각각 설정 할 수 있다.

vue05_filter4.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>필터 - 결과 정렬하기</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container">
        <h1>투표순 정렬하기</h1>
        <ul class="list-group">
            <li v-for="story in orderedStories" class="list-group-item">
```

```
                {{ story.writer }} said "{{ story.plot }}" 투표수 :
                <strong>[{{ story.upvotes }}]</strong>
            </li>
            <li class="list-group-item">
                    <!-- 클릭 할 때마다 오름차순, 내림차순을 조절하는 order 값을 -1 을 곱하며 토글한다. -->
                <button @click="order=order * -1" class="btn btn-primary">
                    <span v-if="order > 0">내림차순 정렬</span>
                    <span v-else>오름차순 정렬</span>
                </button>
            </li>
        </ul>

        <pre>{{$data}}</pre>
    </div>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    new Vue({
        el: '.container',
        data: {
            stories: [{
                plot: "나는 오늘 공부를 한다.",
                writer: "Alex",
                upvotes: 30
            }, {
                plot: "나는 내일 뭐하냐",
                writer: "John",
                upvotes: 8
            }, {
                plot: "공부 하기 싫당..",
                writer: "Alex",
```

```
                upvotes: 26
            }, {
                plot: "배가 너무 부르다...ㅠㅠ",
                writer: "John",
                upvotes: 28
            }],
            order: -1
        },
        computed: {
           //order 프로퍼티가 바뀔 때마다 계속 계산되어 진다.
            orderedStories: function() {
                  var order = this.order;
                  return this.stories.sort(function(a, b) {
                      return (a.upvotes - b.upvotes) * order;
                  });
              }
        }
    })
</script>

</html>
```

사용자 정의 필터

Vue 인스턴스에 있는 데이터들을 표현식을 이용해서 이어 붙이거나 꾸며줄 수 있지만, HTML 코드상에 이러한 데이터를 이어 붙이는 코드 없이 Vue 내장 필터를 이용하여 조금 더 깔끔하게 정보들을 이용 할 수도 있다.

vue05_filter5.html

```
...
<body>
    <div class="container">
        <h1>영웅</h1>
        <ul class="list-group">
            <li v-for="hero in heros" class="list-group-item">
                {{ hero | snitch}}
            </li>
        </ul>
        <pre>{{$data}}</pre>
    </div>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    //커스텀 필터. hero를 받아서 모든 정보를 표시할 Filter Function 생성함
     Vue.filter('snitch', function(hero) {
         return hero.secretId + ' is ' +
             hero.firstname + ' ' +
             hero.lastname + ' in real life!'
     });
    new Vue({
        el: '.container',
        data: {
            heros: [{
                firstname: 'Bruce',
```

```
                lastname: 'Wayne',
                secretId: 'Batman'
            }, {
                firstname: 'Clark',
                lastname: 'Kent',
                secretId: 'Superman'
            }, {
                firstname: 'Jay',
                lastname: 'Garrick',
                secretId: 'Flash'
            }, {
                firstname: 'Peter',
                lastname: 'Parker',
                secretId: 'Spider-Man'
            }]
        }
    })
</script>

</html>
```

컴포넌트

컴포넌트는 Vue.js 에서 가장 강력한 기능 중 하나로써, **기본 HTML 엘리먼트를 확장하여 재사용 가능**한 형태로 만든다. 즉 Vue.js 의 컴파일러가 특정 동작을 추가한 사용자 정의 엘리먼트에 해당된다. 또한 is 속성을 활용해 기존 HTML 엘리먼트에 추가적인 기능을 구현한 형태로도 나타나기도 한다.

컴포넌트의 이름은 태그명이 되며, **이어 대입되는 자바스크립트 객체는 생성자의 역할**을 하게 된다. **생성자는 결과적으로 컴포넌트의 옵션을 의미**하게 된다.

template 옵션은 표현할 기본 HTML 이나 다른 컴포넌트를 포함한 형태로 나타낼 수 있으며 **인라인 방식과 <template></template> 태그를 사용한 방식**을 이용 할 수 있다.

인라인 템플릿

vue06_component1.html

```
...
<body>
    <div class="container">
         <story></story>
         <story></story>
         <story></story>
    </div>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    //Vue 컴포넌트 등록하기 - 인라인 템플릿 사용
     Vue.component('story', {
         template: '<h1>컴포넌트의 템플릿 입니다.</h1>'
     });
    new Vue({
        el: '.container'
    })
</script>
```

위의 예제에서의 Vue.component 함수를 이용하여 story 라는 컴포넌트를 생성하고, 해당하는 컴포넌트의 엘리먼트의 표현은 <h1>컴포넌트의 템플릿 입니다.</h1> 으로 사용하였다. 따라서 <story></story> 부분은 template 에 인라인 방식으로 삽입한 형태로 화면에 나타나게 된다.

템플릿 태그 활용하기 <template></template>

인라인 템플릿은 관리도 힘들 뿐 더러 자주 사용되는 방식이 아니다. 우리는 template 태그를 이용해서 템플릿을 만들고 활용 할 것이다.

vue06_component2.html

```
...
<body>
    <div class="container">
        <story></story>
        <story></story>
        <story></story>
    </div>
</body>

<!--Vue 컴포넌트 등록하기 - template 태그 사용하기 -->
<template id="story-template">
    <h1> My horse is amazing </h1>
</template>

<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    Vue.component('story', {
        template: '#story-template'
    });
    new Vue({
        el: '.container'
    })
</script>
```

먼저 template 태그를 이용해 사용할 template 을 작성한 후 **템플릿에 id 를 부여**한다. 이후에 Vue.component 코드에서 template 프로퍼티에 작성한 템플릿의 id 를 넣어주면 template 태그로 만든 내용을 컴포넌트로 사용 할 수 있게 된다.

컴포넌트 프로퍼티 ★★매우 중요★★

props 특성을 사용하면 개별적인 컴포넌트에 전달할 수 있는 값을 설정 할 수 있게 된다. 만약 템플릿 내부에서 { plot } 프로퍼티를 이용하여 표현하고 싶으면 props : ['plot'] 과 같이 **프로퍼티를 props 에 추가 해 주어야 한다**. 두개 이상의 프로퍼티 전달이 가능하다.

vue06_component3.html

```
...
<body>
    <div class="container">
        <!-- template 에서 사용할 프로퍼티 전달하기 -->
        <story plot="Hello" writer-name="소민호"></story>
        <story plot="My Story"></story>
        <story plot="Hello Component!"></story>
    </div>
</body>
<!--Vue 컴포넌트 등록하기 - template 태그 사용하기 -->
<!-- Vue.component 에 CamelCase 로 prop 을 등록 했다면 실제 템플릿의 속성명은 kebab-case 로 써야 한다. -->
<!-- props:['writerName'] ===> <story writer-name="???" -->
<template id="story-template">
    <h1> {{ plot }} : {{ writerName }} </h1>
</template>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    Vue.component('story', {
        props: ['plot', 'writerName'], //컴포넌트에서 사용할 프로퍼티 등록하기
        template: '#story-template'
    });
    new Vue({
        el: '.container'
    })
</script>
```

전달 할 프로퍼티가 많은 경우 객체를 곧바로 전달 할 수도 있다. 이때는 **v-bind 디렉티브를 활용**하는데, v-bind 디렉티브는 **하나 이상의 속성 또는 컴포넌트 프로퍼티를 표현식에 동적으로 바인딩** 할 때 사용된다. 해당 예제에서는 사용할 **컴포넌트의 속성에 객체를 전달** 하기 위해 사용한다. 축약형은 콜론(:) 을 이용하여 사용 할 수 있다.

vue06_component4.html

```
...
<body>
    <div class="container">
            <!-- 컴포넌트에서 사용하는 프로퍼티인 myStory(my-story)에 객체 전달하기 -->
        <story v-bind:my-story="{plot : 'v-bind 사용', writer: 'Mr. So' }"></story>
        <story :my-story="{plot : '축약형 사용', writer: 'Mhso'}"></story>
    </div>
    <template id="story-template">
            <!-- myStory 라는 이름의 객체를 컴포넌트에서 사용 함 -->
        <h1> {{ myStory.plot }} : {{ myStory.writer }} </h1>
    </template>
</body>
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    Vue.component('story', {
        props: ['myStory'], //template 에서 사용할 myStory 객체 설정하기
        template: '#story-template'
    });
    new Vue({
        el: '.container'
    })
</script>
```

컴포넌트 재사용성

재사용성에 대해 이야기 해보기 위해 예전에 만들어 놨던 필터링 예제를 한번 확인 해 보자. Alex 라는 사람이 써낸 글만 필터링 하기 위해서 확인 하기 위해 다음과 같은 코드를 작성 해 놨다고 가정한다.

```
<li v-for="story in storiesBy('Alex')" class="list-group-item">
       {{ story.writer }} said "{{ story.plot }}"
</li>
```

일전에 만들어 놓았던 형태를 생각해보면 우리가 story.writer 나 story.plot 을 사용 할 수 밖에 없는 이유는 이미 Vue 인스턴스의 data 에 story 데이터가 바인딩이 되어 있기 때문에 저렇게 사용 할 수 밖에 없는 상황이다. 하지만 웹에서 데이터를 불러오거나 다른 파일에서 데이터를 불러와야 한다면 어떻게 해야 할까? 그 때도 이야기 객체(story 객체)를 불러오기 위해 plot 과 writer 를 써야 할까?

만약에 웹에서 데이터를 불러왔더니 plot 이 아니고 body 로 이름이 바뀌었다고 생각해 보자. 그렇다면 코드는 다음과 같이 바뀔 것이다.

```
...
<li v-for="story in storiesBy('Alex')" class="list-group-item">
       ~~{{ story.writer }} said "{{ story.plot }}"~~
       {{ story.writer }} said "{{ story.body }}"
</li>
...
<li v-for="story in storiesBy('John')" class="list-group-item">
       ~~{{ story.writer }} said "{{ story.plot }}"~~
       {{ story.writer }} said "{{ story.body }}"
</li>
...
<li v-for="story in search" class="list-group-item">
       ~~{{ story.writer }} said "{{ story.plot }}"~~
       {{ story.writer }} said "{{ story.body }}"
</li>
```

그냥 단순하게 plot 을 body 로 바꿔주면 되지 않느냐!" 라고 생각 할 수 있으나, 문제는 저 plot 이 쓰인 곳이 한두 군데가 아닐 수도 있다는 이야기 이다. 간단히 말해서 **plot 이 등장한 부분들을 모두 검토해서 수정 해야 한다**는 이야기 이다.

컴포넌트를 활용하면 이러한 문제점을 빠르게 해결 할 수 있다. 일전에 만들어 놨던 story 컴포넌트를 활용하면 훨씬 쉽게 컴포넌트의 재사용성을 볼 수 있을 것이다.

```
...
    <story v-for="story in storiesBy('John')" :my-story="story"></story>
...
    <story v-for="story in storiesBy('Alex')" :my-story="story"></story>
...
    <story v-for="story in search" :my-story="story"></story>
...

<template id="story-template">
    <li class="list-group-item">
        ~~{{ story.writer }} said "{{ story.plot }}"~~
        {{ story.writer }} said "{{ story.body }}"
    </li>
</template>
```

template 태그의 plot 을 body 로 바꿔주면 사용중인 곳은 건들지 않은 채로 사용이 가능하다.

컴포넌트 종합 예제

과자 판매 시스템을 만들어 보려고 한다. 4 가지의 과자를 준비 하고, 과자의 재고수량(quantity)가 0 이 되면 더 이상 구매 할 수 없도록 하는 아주 간단한 예제이다. 먼저 Component 를 구성하기 위한 데이터(snacks 객체를 구성한다.)

```
new Vue({
    el: '#app',
    data: {
        snacks: [{
                productName: '초코송이',
                quantity: 30
            }, {
                productName: '스윙칩',
                quantity: 60
            }, {
                productName: '도리토스',
                quantity: 40
            }, {
                productName: '수미칩',
                quantity: 30
        }]
    },
    methods: {
        increase: function(price) {
            this.proceed += price;
        }
    }
})
```

이어서 컴포넌트에 들어갈 템플릿 태그와 컴포넌트 객체를 만든다.

```
Vue.component('snack-component', {
    template: '#snack-template',
    props: ['snack'],
    methods: {
        buy: function() {
            this.snack.quantity--;
        }
    },
    computed: {
        available: function() {
            return this.snack.quantity > 0;
        }
    }
});
```

```
<template id="snack-template'">
    <div class="col-lg-3">
        <div class="thumbnail">
            <div class="caption">
                <h3>{{snack.productName}}</h3>
                <p>수량 : {{snack.quantity}}</p>
                <p>
                    <button class="btn btn-primary" @click="buy" :disabled="!available">구매</button>
                </p>
            </div>
        </div>
    </div>
</template>
```

Vue.component 를 잘 보면 Vue 인스턴스 객체에서 보던 것과 같이 methods 와 computed 필드가 존재 하는 것이 확인된다. 이는 **각 컴포넌트별로도 메소드와 계산된 프로퍼티 등을 지정 할 수 있다**는 것을 의미한다. 따라서 클릭 이벤트가 발생되면 구매 함수가 실행 되는 것이 확인된다. (@click="buy")

마지막으로 body 를 작성하여 테스트 해보자.

```
<div class="container">
    <div id="app">
        <div class="row">
            <snack-component v-for="snack in snacks" :snack="snack" @sell="increase"></snack-component>
        </div>
        <pre>{{$data}}</pre>
    </div>
</div>
```

사용자 정의 이벤트

우리가 사용하는 모든 Vue 의 인스턴스는 이벤트 인터페이스를 구현 하고 있다. NodeJS 의 EventEmitter 와 비슷한 역할을 한다. Vue 에는 사용자가 직접 이벤트를 설정하고 활용 할 수 있도록 Vue 인스턴스 메소드인 emit 과 on 을 지원하고 있다.

- $on('name', function) 'name' 이벤트 리스너를 function 으로 설정한다.
- $emit('name') 'name' 이벤트를 발생 시킨다.

$on 과 $emit 에서 사용하는 'name'은 개발자 마음대로 지정해 줄 수 있다는 사실을 알아두자.

이벤트 발생과 청취

먼저 간단한 이벤트 청취(Listen - $on)와 이벤트 발생($emit) 에 대해 알아보자.

vue07_event1.html

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>이벤트 바인딩과 생명주기 훅</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>

<body>
    <div class="container text-center">
        <p style="font-size: 140px;">
            {{ votes }}
        </p>
        <button class="btn btn-primary" @click="voteFunction">Vote</button>
        <pre style="text-align:left">{{$data}}</pre>
    </div>
</body>
```

```
<script src=" https://unpkg.com/vue/dist/vue.js "></script>
<script>
    new Vue({
        el: '.container',
        data: {
            votes: 0
        },
        methods: {
            voteFunction: function(writer) {
                //vote 함수가 실행 될 때마다 voted 이벤트를 호출함
                    this.$emit('voted');
            }
        },
        //인스턴스가 만들어 진 후의 생명주기 함수.
        // $on 을 이용해 voted 라는 이벤트를 바인딩 하고 해당하는 리스너 함수를 등록함
        created() {
             this.$on('voted', function(button) {
                this.votes++;
             });
          }
    });
</script>

</html>
```

Vue 인스턴스를 하나 만들어 생명주기 훅 중에 하나인 **created() 에서는 'voted' 이벤트를 바인딩($on)** 하고 있고, methods 필드에 있는 voteFunction 에서는 **'voted' 이벤트를 발생($emit)** 하고 있는 것이 확인된다.

생명주기 훅이란 Vue 인스턴스가 만들어지기 시작해서 파괴 될 때까지의 주기를 말한다. 생명주기 훅의 종류는 다음과 같다.

훅 메소드 명	호출 시점
beforeCreate	인스턴스가 초기화된 후 데이터 감시 및 이벤트/감시자(watch)를 설정하기 전
created	인스턴스가 생성된 후 데이터에 대한 관찰 기능(data), 계산형 속성(computed), 메서드(methods), 감시자 설정(watch)이 완료된 후에 호출됨
beforeMount	마운트가 시작되기 직전(el 을 이용해서 DOM 에 적용되기 직전)
mounted	el 에 vue 인스턴스의 데이터가 마운트 된 후에 호출됨
beforeUpdate	가상 DOM 이 렌더링, 패치되기 전에 데이터가 변경될 때 호출됨. 이 훅에서 주기적인 상태 변경을 수행 할 수 있음. 추가로 다시 뷰를 렌더링 하지는 않음
updated	데이터의 변경으로 가상 DOM 이 다시 렌더링되고 패치된 후에 호출됨. 이 훅이 호출되었을 때는 이미 컴포넌트의 DOM 이 업데이트된 상태이다. 그래서 DOM 에 종속성이 있는 연산을 이 단계에서 수행할 수 있다.
activated	keep-alive 상태의 컴포넌트가 활성화될 때
deactivated	keep-alive 상태의 컴포넌트가 비활성화될 때
beforeDestory	Vue 인스턴스가 제거 되기 전에 호출됨
destroyed	Vue 인스턴스가 파괴된 후

지금 당장은 중요하지 않은 내용이기 때문에 넘어가도록 하자. 단지 현재 이벤트를 설정 하면서 Vue 인스턴스가 만들어 질 때 이벤트를 바인딩 했다는 사실만 알아 두면 된다.

부모-자식 간 통신

HTML 의 엘리먼트 또는 Vue 의 컴포넌트가 중첩된 형태로 존재 하면 부모-자식 관계가 성립한다.

<부모 컴포넌트>

<자식 컴포넌트>

Vue 에는 부모가 자식에게 데이터를 전달 할 때와 자식이 부모에게 데이터를 전달 하는 방식이 각각 존재한다. 우리는 이미 부모가 자식에게 데이터를 전달 하는 방법에 대해서 알아본 적이 있다.

```
...
<div class="container">
    <story v-bind:my-story="{plot : 'v-bind 사용', writer: 'Mr. So' }"></story>
    <story :my-story="{plot : '축약형 사용', writer: 'Mhso'}"></story>
    <story v-for="story in storiesBy('John')" :my-story="story"></story>
</div>
...
```

```
...
Vue.component('story', {
    props: ['myStory'],
    template: '#story-template'
});
...
```

Vue.component 의 props 를 이용해 전달하여 component 에서 활용할 데이터를 전달 받을수록 해준 부분과, v-bind 또는 축약형 (:)을 이용하여 데이터를 전달한 곳이 바로 부모에서 데이터를 전달한 방식이라고 할 수 있다. 이처럼 부모가 가지고 있는 데이터를 자식 컴포넌트에 전달 하기 위해서 props 를 이용하면 손쉽게 객체나 데이터를 전달 할 수 가 있다.

하지만 자식에서 부모로 데이터를 전달하는 방식은 props 처럼 간단하게 사용 가능한 것이 아닌, **event 를 활용하여 자식이 가지고 있는 데이터를 부모에게 전달** 하여야 한다.

즉, **부모 컴포넌트는 자식이 보내는 이벤트에 대해서 대기(v-on, @)**하고 있어야 하고, **자식 컴포넌트는 적절한 때에 부모 컴포넌트가 데이터를 전달 받을 수 있도록 이벤트를 발생($emit('event'))** 하여야 한다.

먼저 이전 예제에서 단순히 투표 카운트만 올리는 것이 아닌 오늘 먹을 점심을 투표할 수 있는 화면을 구성하는 예제를 작성 하겠다. 오늘 먹을 점심 메뉴와 투표수, 각각을 투표할 수 있도록 투표 버튼을 만들어 놓도록 한다. **Food 컴포넌트는 자식 컴포넌트** 이다.

Food 컴포넌트(자식 컴포넌트)

```
Vue.component('food', {
        template: '#food',
        props: ['name'],
        methods: {
             doVote: function(event) {
                  this.$emit('voted');
             }
        }
});
```

Food 컴포넌트의 템플릿

```
<template id="food">
    <button class="btn btn-default" @click="doVote">{{ name }}</button>
</template>
```

name 프로퍼티를 가진 음식 컴포넌트를 만들었다. 음식의 이름을 표현해 주기 위해 템플릿 에서는 name 을 표시하는 버튼을 추가 하였고, 이 버튼을 누르면 doVote 가 실행되어 **voted 이벤트를 청취**(바인딩 돼서 이벤트의 발생(emit)을 기다리는..) 곳에 이벤트를 발생 시키게 된다. 당연히 **voted 이벤트를 기다리고 있는 쪽은(v-on, @) 부모 쪽이** 된다.

Container 컴포넌트(부모 컴포넌트)

```
new Vue({
        el: '.container',
        data: {
            votes: 0
        },
        methods: {
            //자식이 emit 을 통해 @voted="countVote"를 시켜 인자를 전달 했음..!!
            countVote: function(food) {
               this.votes++;
            },
      }
})
```

부모 컴포넌트의 템플릿

```
<div class="container text-center">
    <p style="font-size: 140px;">
        {{ votes }}
    </p>
    <food @voted="countVote" name="김가네"></food>
    <pre style="text-align:left">{{$data}}</pre>
</div>
```

부모 컴포넌트 템플릿의 **<food @voted="countVote"...** 부분이 자식 컴포넌트가 발생시킬 이벤트를 청취하는 쪽이 된다. 자식에서 voted 이벤트가 $emit 되면 **부모 컴포넌트 쪽에서는 countVote 메소드를 실행시켜 부모 컴포넌트의 변수인 votes 를 하나 증가 시킬 것이다.**

이벤트를 통한 인자전달

하나의 메뉴가 등록 되고 해당하는 메뉴의 버튼을 클릭하면 투표수가 올라가는 것이 보일 것이다. 이번엔 3 개의 메뉴를 갖고, 버튼을 누를 때마다 각각의 메뉴의 투표수가 올라가고, 최종 투표수까지 올라가도록 만들어 보자. 또한 **어떠한 버튼이 눌렸는지 파악해서 부모객체에 텍스트만 전달 하여 부모 객체에서 로그를 찍어 볼 것이다**. 먼저, food 컴포넌트마다 각각의 투표수를 갖는 데이터가 있어야 할 것이다. food 컴포넌트 생성자에 data 필드를 추가하자

```
//개별적인 컴포넌트의 투표 수 객체를 리턴
data: function() {
        return { foodVotes: 0 };
}
```

food 컴포넌트에 함수가 들어갔는데, 우리가 일반적으로 new Vue({}) 에서 사용하던 코드와는 약간 다르다. 객체를 리턴 하는 익명 함수 형태로 만든 이유는 바로 food 컴포넌트가 여러 개 추가 될 때마다 새로운 foodVotes 값을 갖는 객체를 만들기 위해서이다. 만약에 함수 없이 객체를 만들었다면, 즉 다음과 같은 형태라면

```
//컴포넌트 전체에서 공유하는 데이터
data: {
        foodVotes: 0
}
```

모든 컴포넌트가 하나의 foodVotes 를 사용 할 것이다. 우리는 메뉴마다 각각 다른 투표수를 적용 시킬 것이기 때문에 객체를 만들어서 리턴하는 함수의 형태로 사용을 할 것이다. food 의 투표수를 저장할 데이터가 새로 생겼기 때문에 food 컴포넌트의 템플릿도 수정 해주자

```
<div class="text-center col-lg-4">
        <p style="font-size:40px;">
            {{ foodVotes }}
        </p>
        <button class="btn btn-default" @click="vote">{{ name }}</button>
</div>
```

부트스트랩을 사용하여 적절하게 열거하기 위해 class 에 col-lg-4 가 추가 되었고, 컴포넌트 생성자에서 추가한 foodVotes 를 보여주기 위해 {{ foodVotes }}도 추가 해 주었다.

이어서 버튼을 눌렀을 때 개별 투표수를 증가시키기 위해 doVote 메소드를 수정해 주자. 여기서 이벤트를 통한 인자 전달이 보이게 되는데 이벤트를 발생시킨 엘리먼트에서 텍스트를 꺼내서 **$emit 의 두 번째 인자로 전달** 하고 있는 것이 보인다.

```
methods:{
            doVote: function(event) {
                this.foodVotes++;
                //부모에게 이벤트를 전달하며 event.srcElement.textContent(이벤트가 발생한 엘리먼트의 텍스트)를 함께 전달함.
                this.$emit('voted', event.srcElement.textContent);
        }
}
```

$emit 의 첫 번째 인자는 발생시킬 이벤트의 이름이 들어가고, 그 다음부터 들어가는 인자들은 이벤트 리스너 콜백 함수에 전달할 인자들이 들어간다.

- $emit('event_name', arg1, arg2, arg3) -> 'event_name' 이벤트 리스너 콜백함수에 arg1, arg2, arg3 을 전달함

이렇게 전달된 인자들은 @voted 로 대기중이던 부모에게 전달되고, 이는 부모 컴포넌트의 메소드인 countVote 가 호출되는 것이다. $emit 으로 전달한 인자를 countVote 가 받아 낼 수 있도록 수정하자

```
new Vue({
        el: '.container',
        data: {
            votes: 0,
            log: []
        },
        methods: {
            //자식이 emit 을 통해 @voted="countVote"를 시켜 인자를 전달 했음..!!
            countVote: function(food) {
                this.votes++;
                this.log.push(food + ' 투표 됐음!')
            },
        }
    })
```

버튼이 클릭되면 해당 버튼의 텍스트도 인자로 전달되고, 이를 저장하여 로그로 활용하기 위해 부모 컴포넌트의 데이터에 log 배열도 추가하였고, @voted 이벤트가 발생되면 로그에 추가하여 표현 하도록 하였다. 마지막으로 부모 템플릿 쪽을 수정 해주자

```
<div class="container text-center" v-cloak>
        <p style="font-size: 140px;">
            {{ votes }}
        </p>
        <div class="row">
            <food @voted="countVote" name="김가네"></food>
            <food @voted="countVote" name="한돈애"></food>
            <food @voted="countVote" name="하나우동"></food>
        </div>
        <h1>투표 로그</h1>
        <ul class="list-group">
            <li class="list-group-item" v-for="vote in log">{{ vote }}</li>
        </ul>
        <pre style="text-align:left">{{$data}}</pre>
</div>
```

비부모 자식 간 통신

위의 예제에서 초기화 버튼을 추가해서초기화 버튼을 누르면 모든 내용이 사라지게 하고 싶다.초기화 버튼은 클릭 이벤트가 일어나면 부모 컴포넌트의 내용을 먼저 초기화 하고,그 다음 모든 자식 컴포넌트인 food 컴포넌트의 내용을 초기화를 하기 위해 이벤트를 발생 시킬 것이다.그렇다면 자식 컴포넌트에서는 초기화 이벤트를 어떻게 초기화 해야 할까?

<food @voted="countVote"> 같은 코드는 부모 입장에서 이벤트를 청취 해서 자식이 보내는 voted 이벤트를 보내면 countVote 함수를 호출 하겠다는 이야기 이기 때문에,자식 컴포넌트로 이벤트를 보낸다는 개념과는 약간 차이가 있다.

이 때 우리가 사용 할 수 있는 것은 **버스(bus)**의 개념이다.이벤트를 주고 받을 컴포넌트 사이에 bus 를 두고,컴포넌트들은 버스에 이벤트를 바인딩 시키고, 발생 시키면서 서로 간에 이벤트 발생을 손쉽게 진행 할 수 있다.

먼저 버스를 활용하기 위해서는 Vue 인스턴스가 필요하다. 그런 다음 이벤트를 바인딩 시킬 컴포넌트의 created() 훅에 버스에 이벤트를 바인딩 시킨다.우리가 만들 예제는 Reset 버튼이 눌렸을 때 현존하는 모든 food 컴포넌트가 초기화 이벤트를 받아야 하기 때문에 food 컴포넌트가 만들어 질 때 버스에 이벤트를 바인딩 시킨다.마찬가지로 투표 이벤트 또한 자식에서 부모에게 이벤트를 전달 할 때 버스를 이용 해 본다.

```
//서로 다른 컴포넌트 사이에 이벤트를 주고 받을 수 있는 bus 객체 생성
    var bus = new Vue();
    Vue.component('food', {
        template: '#food',
        props: ['name'],
        data: function() {
            return {
                foodVotes: 0
            };
        },
        methods: {
            //투표 이벤트를 실행 했을 때 bus 를 이용해 voted 이벤트 발생
            vote: function(event) {
                //this.name 을 사용하는 대신 이벤트의 엘리먼트의 텍스트에 접근함
                var foodname = event.srcElement.textContent;
                this.foodVotes++;
```

```
            bus.$emit('voted', foodname);
        },
        reset: function() {
            this.foodVotes = 0;
        }
    },
    //food 컴포넌트가 만들어 졌을 때 bus 인스턴스에 reset 이벤트 바인딩
    created() {
        console.log('자식 생성 됨');
        bus.$on('reset', this.reset);
    }
});

new Vue({
    el: '.container',
    data: {
        votes: {
            count: 0,
            log: []
        }
    },
    methods: {
        countVote: function(foodname) {
            this.votes.count++;
            this.votes.log.push(foodname + ' received a vote.')
        },
        reset: function() {
            this.votes = {
                count: 0,
                log: []
            }
```

```
            //리셋 함수 호출 시 reset 이벤트 실행
            bus.$emit('reset');
        }
    },
    //부모 인스턴스가 만들어지면 voted 이벤트를 바인딩함
    created() {
        console.log('부모 생성됨');
        bus.$on('voted', this.countVote);
    }
})
```

먼저 버스를 만들기 위해 Vue 의 인스턴스를 먼저 만들었다.이 버스는 각각 부모와 자식 컴포넌트가 만들어 질 때즉 created() 훅이 발생 될 때 서로 간에 받아낼 이벤트를 바인딩 한다.부모 컴포넌트는 자식 컴포넌트가 voted 이벤트를 발 생 시켰을 때 받아낼 voted 이벤트를 청취하고,자식 컴포넌트는 리셋 버튼이 눌렸을 때 (사실상 부모에서 발생된 이벤트..)의 이벤트를 받아낼 reset 이벤트를 각각 $on 을 이용해 바인딩 시켰고,필요한 시점에 bus.$emit 을 이용하여 서로간의 이벤트를 발생시켰다.

[실습] story 투표에서좋아요 버튼을 만들어 사용자가 관심있어 하는 story 에 '좋아요' 버튼을 누를 수 있도록 한다,그리고 바로 밑에 사용자가 '좋아요' 버튼을 클릭한 글만 모아서 따로 표시해두자,물론 '좋아요' 버튼을 다시 누르면 좋아요가 취소되고,좋아요만 따로 모아놓은 곳에서도 삭제 되어야 한다.

클래스와 스타일 바인딩

클래스 바인딩 - 객체문법

데이터 바인딩이 필요한 경우는 주로 엘리먼트의 클래스와 스타일을 조작할 때이다.이러한 경우 보통은 v-bind:class 를 이용하는데,조건에 따라서 클래스를 적용하고 토글하거나 하나의 객체를 이용해 여러가지 클래스를 한꺼번에 엘리먼트에 적용 시킬 때 많이 사용한다.

v-bind:class 는 클래스의 객체와,적용 시킬 조건값(boolean)을 입력 받는다.

```
{
    'classA' : true,
    'classB':false,
    'classC':true
}
```

위의 객체는 true 로 되어있는 부분만 엘리먼트의 클래스로 지정되어 스타일이 적용되게 될 것이다.객체 형태이기 때문에 Vue 의 data 프로퍼티 같은 곳에 집어 넣을 수 있다.

```
data : {
elClasses: {
        'classA' : true,
        'classB' : false,
        'classC' : true
    }
}
<div v-bind:class="elClasses"></div>
```

와 같이 사용할 수 있다.다음은 객체 바인딩을 활용한 스타일 변경이다.

```
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>클래스 바인딩 객체 문법</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container text-center">
        <div class="box" v-bind:class="{'red':color,    'blue':!color}"></div>
        <div class="box" v-bind:class="{'purple':color, 'green':!color}"></div>
        <div class="box" v-bind:class="{'red':color,    'blue':!color}"></div>
        <div class="box" v-bind:class="{'purple':color, 'green':!color}"></div>
        <button v-on:click="flipColor" class="btn btn-block btn-success">
            Flip Color!
        </button>
    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script>
    var vm = new Vue({
        el: '.container',
        data: {
            color: true
        },
        methods: {
            flipColor: function() {
                console.log('color' + this.color);
                this.color = !this.color;
            }
```

```
        }
    });
</script>
<style>
    .red {
        background-color: red;
    }

    .green {
        background-color: green;
    }

    .blue {
        background-color: blue;
    }

    .purple {
        background-color: purple;
    }

    .box {
        float: left;
        width: 200px;
        height: 200px;
        margin: 40px;
        border: 1px solid rgba(0, 0, 0, .2);
    }
</style>
</html>
```

각 box 의 v-bind:class 쪽을 보면 color 조건 값에 따라서 적용할 클래스를 각각 설정 하는 것이 보인다.

클래스 바인딩 - 배열 문법

클래스명으로 구성된 배열을 사용하여 엘리먼트에 클래스 리스트를 적용 할 수 있다.또한 삼항연산자 ? 를 이용하여 조건에맞게 클래스를 사용할 수도 있다.

```
<div class="container text-center">
       <div class="box" v-bind:class="[color ? 'red' : 'blue']"></div>
       <div class="box" v-bind:class="[color ? 'green' : 'purple']"></div>
       <div class="box" v-bind:class="[color ? 'red' : 'blue']"></div>
       <div class="box" v-bind:class="[color ? 'green' : 'purple']"></div>
       <button v-on:click="flipColor" class="btn btn-block btn-success">
           Flip Color!
       </button>
</div>
```

스타일 바인딩 - 객체 문법

자바스크립트 객체를 활용해서 스타일을 선언 할 수도 있다.

```
var vm = new Vue({
       el: '.container',
       data: {
           niceStyle: {
                color: 'blue',
                fontSize: '20px'
            }
       }
   });
```

컴포넌트에 niceStyle 로 자바 객체를만들고 마치 css 처럼 만들었다.이 객체는 곧바로 style 로 들어 갈 수 있다.

```
<div :style="niceStyle">객체 문법을 사용했습니다. $data 를 보세요!</div>
<div :style="{'color':'red', 'fontSize':'20px'}">인라인 스타일을 적용 시켰습니다.</div>
<div :style="{'color': niceStyle.color, 'fontSize': niceStyle.fontSize}">스타일 객체 안에서 인라인으로 참조합니다!</div>
```

객체로 된 스타일은 위와 같이 사용된다.$data 에 들어 있는 객체를 가져올 수도 있고,두 번째처럼 인라인 형태로도 스타일 적용이 가능하며, 마지막으로 객체 안쪽에 있는 스타일을 가져와서 사용 할 수도 있다.

스타일 바인딩 - 배열 문법

두개 이상의 스타일 객체를 한꺼번에 적용 시키기 위해서는 배열 문법을 활용하여 스타일을 적용 시켜주면 된다.

```
var vm = new Vue({
        el: '.container',
        data: {
            niceStyle: {
                color: 'blue',
                fontSize: '20px'
            },
            badStyle: {
                fontStyle: 'italic'
            }
        }
    });
```

```
<div :style="[niceStyle, badStyle]">배열 문법을 사용 하였습니다.</div>
```

Vue 의 HTTP 통신

지금까지는 Vue 인스턴스 내부에 있는 데이터를 활용하여 여러가지 Vue 의 기본 작업 등에 대해 이야기 해보았다. 이제부는 이전과 다르게 실제 데이터를 HTTP 웹 서버에서 데이터를 받아 vue 에 출력 할 것이다. Vue 를 이용해서 웹에서 데이터를 불러 오는 과정은 전부 비동기 통신으로 이루어지며, 우리가 알고있는 ajax 와 Vue 에서 웹 요청을 수행하고 처리하기 위한 vue-resource, 비동기 통신을 매우 간편하고 안전하게 처리해주는 axios 를 사용 할 수 있다.

$.get 을 이용한 비동기 통신

$.get 은 jQuery 의 메소드로써, et 방식만을 이용해 데이터를 비동기적으로 가지고 올 때 사용 할 수 있다. $.ajax 를 활용 하는 것과 같은 효과를 가지고 있다.

```
var vm = new Vue({
            el: '#app',
            data: {
                stories: []
            },
            mounted: function() {
                this.$nextTick(function() {
                    $.get('/api/stories', function(data) {
                        vm.stories = data.result;
                    });
                });
            }
        });
```

mounted 는 Vue 인스턴스 생명주기 중 하나로써 Vue 의 인스턴스 객체가 el 에 의해 마운트 된 직후에 발생 되는 생명주기 훅이다. 또한 위의 코드에서 $.nextTick 은 모든 DOM 이 준비 된 후에 실행 할 때 사용하는 메소드 이다.

결론적으로 모든 Document 가 로딩이 완료 된 후에 $.get 을 실행시켜 data 를 가져올 수 있다는 뜻이 된다. 템플릿과 실제 사용한 코드이다.

```
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
    <title>Document</title>
</head>

<body>
    <div id="app">
        <div class="container">
            <h1>Let's hear some stories!</h1>
            <ul class="list-group">
                <story v-for="story in stories" :story="story"></story>
            </ul>
            <pre>{{$data}}</pre>
        </div>
    </div>
    <template id="template-story-raw">
        <li class="list-group-item">
            {{ story.writer }} said "{{ story.plot }}"
        </li>
    </template>
    <script src="https://code.jquery.com/jquery-3.2.1.js"></script>
    <script src="https://unpkg.com/vue"></script>
    <script>
        Vue.component('story', {
            template: '#template-story-raw',
            props: ['story']
```

```
        });

        var vm = new Vue({
            el: '#app',
            data: {
                stories: []
            },
            mounted: function() {
                this.$nextTick(function() {
                    $.get('/api/stories', function(data) {
                        vm.stories = data.result;//불러온 데이터를 모델 객체에 넣기. vue 의 인스턴스를 참조한 vm 을 이용했다.
                    });
                });
            }
        });
    </script>
</body>


</html>
```

실행 해보면, 웹에 있는 데이터를 JSON 형태로 불러와서 Vue 의 모델 객체 내에서 사용 하고 있는 것을 확인 할 수 있다.

  및 <li> 를 이용해 표시 하는 것도 괜찮은 방법이지만, 많은 량의 데이터가 있는 경우에는 보기가 힘들어 질 수도 있기 때문에 이번엔 <table>을 사용하여 데이터를 표시한다. 또한 **삭제와 투표하기를 ajax 를 이용하여 수행한다.**

```
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
    <title>HTTP Vue</title>
</head>
<body>
    <div id="app">
        <div class="container">
            <table class="table table-striped">
                <tr>
                    <th>#</th>
                    <th>Plot</th>
                    <th>Writer</th>
                    <th>Upvotes</th>
                    <th>Actions</th>
                </tr>
                 //테이블 같이 내부에 들어갈 엘리먼트가 제약이 되어 있는 경우엔 is 속성을 이용해 템플릿을 지정해 주면된다.
                 //이는 story 가 사용자 엘리먼트 라는 것을 말해준다.
                <tr v-for="story in stories" is="story" :story="story"></tr>
            </table>
            <p class="lead">Here's a list of all your stories</p>
            <pre>{{ $data }}</pre>
        </div>
    </div>
    <template id="template-story-raw">
```

```
        <tr>
            <td>{{ story._id }}</td>
            <td><span>{{ story.plot }}</span></td>
            <td><span>{{ story.writer }}</span></td>
            <td>{{ story.upvotes }}</td>
            <td>
                <div class="btn-group">
                    <button @click="upvoteStory(story)" class="btn btn-primary">
                            Upvote
                    </button>
                    <button @click="deleteStory(story)" class="btn btn-danger">
                            Delete
                    </button>
                </div>
            </td>
        </tr>
    </template>
    <script src="https://code.jquery.com/jquery-3.2.1.js"></script>
    <script src="https://unpkg.com/vue"></script>
    <script>
        Vue.component('story', {
            template: '#template-story-raw',
            props: ['story'],
            methods: {
                upvoteStory: function(story) {
                    story.upvotes++;
                    $.ajax({
                            url: '/api/stories/upvotes/' + story._id,
                            type: 'put',
                            data: story
                        })
```

```
            },
            deleteStory: function(story) {
                //story 찾기
                var index = vm.stories.indexOf(story);
                //삭제
                vm.stories.splice(index, 1);

                    //DELETE 요청
                    $.ajax({
                        url: '/api/stories/' + story._id,
                        type: 'delete'
                    })
            }
        }
    });
    var vm = new Vue({
        el: '#app',
        data: {
            stories: []
        },
        mounted: function() {
            this.$nextTick(function() {
                $.get('/api/stories', function(data) {
                    vm.stories = data.result;
                });
            });
        }
    });
</script>
</body>
</html>
```

<table> 같은 태그는 내부에 <td> 같은 엘리먼트가 들어 갈 수 있도록 이미 지정이 되어 있다. 이럴 때는 **is 속성을 이용해 템플릿을 지정**해 주면 손쉽게 테이블 태그에 직접 만들어 놓은 커스텀 엘리먼트를 이용 할 수 있게 된다.

마찬가지로 jQuery 의 ajax 를 이용해서 삭제와 투표하기를 만들어 보았다. 먼저 내부 $data 에 저장되어 있는 부분부터 변경 또는 삭제를 진행하고, ajax 통신을 이용해 해당 데이터를 지우거나 투표수를 높일 수 있도록 요청하는 코드가 보인다.

이번엔 axios(엑시오스)를 이용한 통신 방법이다. jQuery 의 ajax 를 이용해서도 충분히 비동기 통신을 만들어 낼 수 있지만, 비동기 통신만을 위해서 jQuery 를 사용 하는 것은 약간은 낭비이다. 따라서 전문적으로 비동기 통신을 할 수 있는 라이브러리인 axios 를 이용해 비동기 통신을 해보겠다.

먼저 axios 는 함수 자체에서 통신 방식을 설정 해 낼 수 있다. 설정 할 수 있는 통신 방식은 **get/post/put/delete** 등이 있다.

then 과 catch 는 Promise 구문 문법이며, then 에는 성공 했을 때의 함수 콜백이 사용되고, catch 에서는 오류 발생이나 통신 실패 등 통신을 할 수 없을 때 실행할 함수를 등록한다.

- axios.**get**('url')
  .then(function(response){
    요청이 성공 했을 때.
  }).
  catch(function(error){
    요청에서 예외가 등장하여 예외처리가 필요 할 때
  });

이전 예제에서 jQuery 로 사용했던 DELETE 와 PATCH 요청을 바꾸면 다음과 같은 형태가 된다.

```
upvoteStory : function(story){
    story.upvotes++;
    axios.patch('/api/stories/'+story._id, story)
}
```

```
deleteStory: function(story){
    var index = this.$parent.stories.indexOf(story);
    this.$parent.stories.splice(index, 1);
    axios.delete('/api/stories/' + story._id)
}
```

story 컴포넌트는 stories 배열에 직접 접근 할 수 없기 때문에 부모 인스턴스에 접근 할 수 있는 $parent 를 이용하여 접근 했다.

다음 예제는 ajax 를 쓰던 통신 방식을 axios 방식으로 바꾼 코드이다. 초기에 데이터를 불러오기 위해 작성한 mounted 생명주기에 **Vue.set 이라는 메소드를 사용 했는데 이는 Vue 인스턴스 구동 중 바뀌는 변수를 감지** 하기 위해 사용한 것이다.

```
...
upvoteStory: function(story) {
                story.upvotes++;
                axios.put('/api/stories/upvotes/' + story._id, story);
            },
deleteStory: function(story) {
                //story 찾기
                var index = vm.stories.indexOf(story);
                //삭제
                this.$parent.stories.splice(index, 1);
                axios.delete('/api/stories/' + story._id);
            }
...
```

```
...
mounted: function() {
            //axios 를 이용한 get 요청하기
            axios.get('/api/stories').then(function(response) {
                  console.dir(response.data)
                  Vue.set(vm, 'stories', response.data.result);
            });
        }
...
```

mounted 생명주기에 있는 Vue.set 은 잠시 후에 설명한다.

이어서, 데이터 수정을 위해 Edit 버튼을 **템플릿에** 추가하자

```
<td>
    <div class="btn-group" v-if="!story.editing">
        <button @click="upvoteStory(story)" class="btn btn-primary">
            Upvote
        </button>
        <button @click="editStory(story)" class="btn btn-default">
            Edit
        </button>
        <button @click="deleteStory(story)" class="btn btn-danger">
            Delete
        </button>
    </div>
    <div class="btn-group" v-else>
        <button @click="updateStory(story)" class="btn btn-primary">
            Update Story
        </button>
        <button @click="story.editing=false" class="btn btn-default">
            Cancel
        </button>
    </div>
</td>
```

맨 위쪽에 있는 v-if 를 보면 story 에 editing 이 추가 된 것을 확인 할 수 있다. edting 변수는 현재 데이터를 수정 할 수 있는지 여부를 갖고 있는 변수로써, Vue 인스턴스가 mounted 상태가 되어 전체 데이터를 불러올 때 각각의 story 객체에 부여되는 변수이다. 다음은 마운팅 될 때 editing 변수를 추가 시킨 형태로 story 들을 구성하는 방법이다. 각각의 배열을 순회 하면서 edting 프로퍼티를 추가적으로 부여 해야 하기 때문에 map() 함수를 사용한다.

```
mounted: function() {
        //axios 를 이용한 get 요청하기
        axios.get('/api/stories').then(function(response) {
        //런타임 상에서 프로퍼티를 추가하거나 삭제해야 하는 경우 Vue.set 이나 Vue.delete 를 사용한다.
        var storiesReady = response.data.result.map(function(story) {
            story.editing = false;
            return story;
        });
        vm.stories = storiesReady;
                    //Vue.set(vm, 'stories', response.data.result);
        });
    }
```

```
<td>
    <!-- 이야기를 수정하는 경우 줄거리에 대한 입력 필드를 출력 -->
    <input v-if="story.editing" v-model="story.plot" class="form-control">
    <!-- 그 밖의 경우 이야기의 줄거리 출력 -->
    <span v-else>
        {{ story.plot }}
    </span>
</td>
<td>
    <!-- 이야기를 수정하는 경우 작가에 대한 입력 필드를 출력 -->
    <input v-if="story.editing" v-model="story.writer" class="form-control">
    <!-- 그 밖의 경우 작가 이름 출력 -->
    <span v-else>
        {{ story.writer }}
    </span>
</td>
```

데이터를 관찰할 때 없었던 속성을 새로 추가하면 DOM 에 바로 반영되지 않는다. 따라서 항상 반응성이 있는 프로퍼티를 선언하는 것이 제일 좋은데, 이미 구동이 되고 나서 프로퍼티가 수정 되었다면 Vue.set 을 이용하여 반응성 있는 프로퍼티를 만들도록 해 주는 방법이 좋다. 그래서 storiesReady 라는 story 객체를 만드는데, 이 때 map 함수에 의해 전체 story 객체가 editing=false 변수를 가지고 있는 것이 확인 된다. 결과적으로 모든 story 객체는 editng 변수를 가진 채로 배열의 원소로 구성이 되고 이는 vm.stories = storiesReady 코드에 의해 editing 변수를 가진 코드로 DOM 에 반영 될 것이다.

마지막으로 새로운 이야기를 추가 하는 방법에 대한 이야기 이다. 사용자가 새로운 이야기 만들기 버튼을 누르면 새로운 이야기를 만들 수 있는 엘리먼트를 제공해 줘야 한다. 아무것도 입력이 되지 않은 비어있는 이야기를 만들고 push 를 이용해 stories 배열에 추가하면 된다. 새로운 이야기는 바로 수정창(입력창)이 떠야 하기 때문에 editing 을 true 로 설정한다.

```
methods: {
                createStory: function() {
                    var newStory = {
                        plot: "",
                        upvotes: 0,
                        editing: true
                    };
                    this.stories.push(newStory);
                }
```

```
<p class="lead">
                Here's a list of all your stories
                <button @click="createStory()" class="btn btn-primary">
                    Add a new one?
                </button>
</p>
```

createStory 함수를 만들어 Vue 인스턴스에 추가한다. 그 다음 목록 바로 아래에 버튼을 추가하여 createStory 메소드를 호출 할 수 있도록 한다. newStory.editing 이 true 이기 때문에 Edit 동작 버튼과 함께 plot 과 writer 에 바인딩된 입력 필드가 등장 할 것이다.

다음은 작성된 내용을 전송 하기 위해 storeStory 메소드를 컴포넌트에 만들어보자.

```
storeStory: function(story) {
                    axios.post('/api/stories/', story).then(function(response) {
                            story.editing = false;
                    });
                }
```

다음은 새로운 글을 쓰기 위한 버튼 템플릿을 구성 해야 하는데, 현재 새로운 글을 작성 할 때는 맨 앞쪽에 위치하는 id 값이 없는 것을 확인 할 수 있다. 이 id 값을 이용하여 저장버튼과 수정 버튼을 구분해 v-if 의 조건으로 줄 수 있다.

```
<div class="btn-group" v-else>
    <!-- _id 가 존재하면 수정할 이야기임 -->
    <button v-if="story._id" @click="updateStory(story)" class="btn btn-primary">
        Update Story
    </button>
        <!-- _id 가 존재 하지 않으면 새로 작성하는 이야기 -->
    <button v-else @click="storeStory(story)" class="btn btn-success">
        Save new Story
    </button>
    <!-- 취소 버튼은 수정을 하든, 새로 작성을 하든 무조건 있어야 함-->
    <button @click="story.editing=false" class="btn btn-default">
        Cancel
    </button>
</div>
```

여기까지 코드가 진행되고 나면 저장은 잘 되나, 다시 이야기가 로딩 되지 않아 입력창이 계속 떠 있게 된다. 이를 해결 하기 위해서는 다시 처음부터 이야기를 불러오면 된다. Vue 인스턴스에 새로운 메소드를 추가하자

```
fetchStories: function() {
                    var vm = this;
                    axios.get('/api/stories')
                        .then(function(response) {
                            var storiesReady = response.data.result.map(function(story) {
                                story.editing = false;
                                return story;
                            });
                            //vm.stories = storiesReady;
                            Vue.set(vm, 'stories', response.data.result);
                        });
            }
```

단순히 한번만 세팅 하는 것이 아닌, 입력을 하거나 수정을 하게 되면 모든 데이터를 처음부터 불러 와야 하기 때문에 제일 마지막 코드에 Vue.set 을 이용해 바뀐 데이터를 stories 에서 계속 관찰 할 수 있도록 설정 해 놓았다.

이제 mounted 훅과 데이터를 저장한 storeStory 에서도 따로 불러올 필요 없이 fetchStories 를 호출 할 수 있도록 해 주자.

```
mounted: function() {
              this.fetchStories();
          }
```

```
storeStory: function(story) {
                axios.post('/api/stories/', story).then(function(response) {

                    vm.fetchStories();

                    story.editing = false;
                });
            }
```

만약 추가 하고 나서 방금 추가한 데이터만 가지고 오고 싶다면 story 에 _id 만 추가해 주면 된다.

```
storeStory: function(story) {
                axios.post('/api/stories/', story).then(function(response) {
                    Vue.set(story, '_id', response.data.story._id);
                    story.editing = false;
                });
            }
```

ECMAScript 6(ES6 - ES2015)

Vue 뿐만이 아닌 여러가지 스크립트 언어류를 이용해 대규모 개발을 하기 위해서 하나의 표준화된 스크립트들에 대한규격을 만들었는데, 우리가 현재 사용할 수 있는 최신 규격은 지금부터 이야기 해볼 ECMAScript 6 - ES6 라고 불리우는 규격이다.

2009 년에 ES5 가 표준화 되고 2015 년에 새롭게 ES6 의 규격 사양이 완성 되었기 때문에 최근 유행하는 여러 기술을 학습하고 개발 하기 위해서 필수적인 내용이라고 할 수 있다.

하지만 최신 기술 사양이다 보니 단점도 존재 하는데, 바로 구 버전의 브라우저는 지원 하지 않는 것이다. 이에 따라 Babel 같은 스크립트 컴파일러를 이용하여 구버전에 맞게 개발을 해 줘야 할 때도 있다.

지금부터 ES6 에서 추가된 몇가지 특징들을 확인해 보자

변수 선언

대표적으로 자바스크립트에서 사용하는 변수 선언의 기준은 var 이다. var 형태로 변수를 선언하면 스코프의 기준이 우리가 일반적으로 사용하는 코드블록({ }) 아닌 function, with, catch 등이 되기 때문에 호이스팅 현상이 일어나게 된다.

ES6 에서는 **let 키워드**가 등장했고, 이는 코드 블록을 기준으로 변수를 다룰 수 있게 해준다.

```
let age = 22
if (age >= 18) {
    let adult = true;
    console.log(adult);
}
//여기서는 adult 에 접근 할 수 없음.
console.log(adult); //error!
```

```
let age = 22;
let adult;
if (age >= 18) {
    adult = true;
    console.log(adult);
}
console.log(adult); //ok!
```

상수를 만들어 낼 수 있는 const 선언도 추가되었다. const 선언은 기본적으로 let 과 똑 같은 형태의 스코프를 가지며, 한번 값이 대입되면 그 다음부터는 값을 바꿀 수 없다.

```
const name = '소민호';
name = '소민호 2'; //error!
```

화살표 함수

자바스크립트에서 함수를 만들 때 function 키워드를 이용해서 함수를 만들어 낸다. 하지만 es6 에서 새로운 방식으로 함수를 만들 수 있게 됐는데, 화살표 기호(=>) 를 이용해서 함수를 만들 수 있게 되었다.

```
var foo_arrow = arg => arg + 1;
console.log(foo(5));

var foo_func = function(arg) {
    return arg + 1;
}
```

인자를 여러 개 받는 형태의 코드도 만들어 낼 수 있다.

```
var sum_arrow = (a, b) => a + b;
console.log(sum_arrow(10, 20));

var sum_func = function(a, b) {
    return a + b;
}
```

매개 변수 없이 화살표 함수를 구현 하려면 비어 있는 괄호를 만들면 되고, 중괄호를 이용해 함수의 몸체를 구성 할 수도 있다.

```
var sayHi = () => {
    console.log('say');
    console.log('hi');
}
sayHi();
```

모듈

let, const, 화살표 함수 등 우리가 손쉽게 파악할 수 있고 불편 했던 점을 개선 한 것도 있지만, 제일 큰 변화 - 가장 향상된 변화는 ES6 스타일의 모듈이다. **ES6 는 여러 파일에서 모듈을 내보내고(export) 가져올(import) 수 있다.**

module1.js 변수 하나만 내보내기

```
export var name = '소민호'
```

main1.js

```
import { name } from './module';
console.log('Hello', name);
```

module2.js 함수, 변수 내보내기

```
export var name = '소민호';

export function getAge() {
    return 33;
}
```

main2.js

```
import { name, getAge } from './module'
console.log(name, 'is', getAge());
```

module3.js 객체 내부에서 내보내기

```
var name = '소민호';

function getAge() {
    return 22;
}

export default { name, getAge }
```

main3.js

```
import person from './module';

console.log(person.name, 'is', person.getAge());
```

클래스

ES6 이 도입되면서 클래스가 등장 했으며, 객체 지향 프로그래밍을 위한 클래스 라기보다 손쉽게 상속을 구현하기 위한 용도로 많이 쓴다.

```
//부모 클래스
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    calcArea() {
        return this.height * this.width;
    }
    get area() {
        return this.calcArea();
    }
```

```
}

//자식 클래스
class Square extends Rectangle {
    constructor(side) {
        super(side, side);
    }
}

var square = new Square(5);
console.log(square.area);
```

기본 매개변수 값

함수에 기본 매개변수 값을 지정 할 수 있다.

```
function divide(x, y = 2) {
    return x / y;
}

var a = divide(10);
console.log(a); // 5
```

템플릿 리터럴

템플릿 리터럴은 표현식을 포함할 수 있는 문자열 리터럴이다. 백틱(`)을 이용해서 문자열을 감싸면 $ {표현식}을 사용 할 수 있게 된다. 표현식은 함수와 변수 전부 사용 가능하다.

```
//템플릿 리터럴 - 변수 사용
let name = 'Alex';
console.log(`Hello ${name}`);

//템플릿 리터럴 - 함수 사용
let add = (a, b) => a + b;
let [a, b] = [10, 2];
console.log(`add result : ${add(a,b)}`);

//템플릿 리터럴 - 표현식 사용
let [c, d] = [10, 2];
console.log(`c+d =
${c+d}`)
```

고수준 워크플로우

모든 브라우저에서 ES6 를 완벽하게 지원하지는 않는다. 따라서 ES6 형태로 개발을 한다면 포기해야만 하는 브라우저가 등장하기 마련인데, 자바스크립트 컴파일러를 사용하면 ES6 를 지원하지 않는 브라우저도 최신 자바스크립트 기능을 사용 하게 끔 할 수 있게 된다.

우리는 Babel 을 설치하여 ES6 로 만든 코드를 사용해 보고, 자동화와 웹 팩(Webpack) 까지 알아 볼 것이다.

Babel 설치하기

먼저 적절한 위치에 Babel 을 사용할 폴더를 만들고 시작한다.

- mkdir babel-example
- echo {} > package.json

mkdir 로 폴더를 만들고, echo 를 이용해 {}만 들어있는 package.json 을 만들어 주면 된다. package.json 이 생겼으니 babel 을 설치해 주자.

- npm install babel-cli --save-dev

package.json 을 열어서 babel-cli 가 제대로 설치 됐는지 확인해 보자..

Babel 설치가 완료 됐으므로 ES6 형태로 만들어진 코드를 빌드하면서 소스코드 변환 설정을 해야 한다. 우리는 ES2015 코드를 변환 하기 때문에 기준 값이라고 할 수 있는 preset 을 es2015 로 잡고 설정한다. 프리셋을 활성화할 설정파일 (.babelrc)도 만든다.

- echo { "presets" : ["es2015"]} > .babelrc

babelrc 파일이 만들어 진 것이 확인 됐으면, package.json 파일을 열어 "script" 필드를 다음과 같이 추가하자

package.json

```
{
    "scripts": {
        "build": "babel src -d assets/js"
    },
    "devDependencies": {
        "babel-cli": "^6.26.0",
        "babel-preset-es2015": "^6.24.1"
    }
}
```

npm run build 를 이용하여 자바스크립트 파일을 빌드 시킬 때 babel 을 이용 하겠다 라는 뜻이 된다. babel 을 이용해 변환된 결과물은 assets/js 에 들어간다.

- mkdir src
- mkdir assets
- cd assets
- mkdir js

명령어를 이용하여 폴더를 만들어 주자. src 폴더가 생겼고, assets 와 그 하위폴더인 js 폴더까지 생겼을 것이다. visual studio code 를 열어서 src 폴더에 간단한 자바스크립트를 ES6 스타일로 작성해 보자.

/src/sum.js

```
const sum = (a, b) => a + b;
console.log(sum(5, 3));
```

작성이 끝났으면 **npm run build** 명령어로 sum.js 를 컴파일 하게 되면 assets/js/sum.js 에 컴파일된 자바스크립트 파일이 생긴다. cmd 창에서 assets/js/ 까지 이동한 다음 node sum 으로 결과물을 확인 해 보자. **자바스크립트 파일이 추가되거나 변경 되면 바로 npm run build 명령어를 사용해 컴파일 하는 것을 잊지 말자!!**

라고 했지만 매번 자바스크립트 파일이 변경 될 때마다 명령어를 계속 쳐가면서 빌드하는 것은 매우 귀찮다. Gulp 를 이용하면 자동으로 자바스크립트 파일이 변경 되는 것을 감지하여 손쉽게 관리가 가능하다. 이러한 시스템을 **태스크러너** 라고 한다. 이러한 테스크러너는 워크플로우를 자동화하도록 도와준다.

태스크러너는 자동화를 사용하려고 쓰며, 프로젝트 코드 최소화, 컴파일, 단위 테스트, 린팅(경고) 등과 같은 반복되는 작업을 수행할 때 해야 할 일을 줄여준다.

- npm install gulp-cli --global
- npm install gulp --save-dev

gulp-cli 는 전역 설치, gulp 는 해당하는 프로젝트에 설치되게 된다. 이어서 프로젝트 루트 폴더(프로젝트 최 상단 폴더)에 gulpfile.js 파일을 만들어 주고 코드를 입력하자.

```
const gulp = require('gulp');

gulp.task('default', function(){
    //이곳에 기본 태스크를 작성
});
```

위의 코드가 gulp 를 설정 하는 코드이다. 하지만 아직 아무 설정을 하지 않았기 때문에 소용이 없다.

우리가 자동화 할 일에 대해서 설정해 주면 되는데, 우리는 babel 을 활용한 자바스크립트 코드의 컴파일 과정을 자동화 하고자 한다. babel 을 자동으로 실행 시켜 주기 위해 gulp-babel 을 설치하자

- npm install gulp-babel --save-dev

이어서 gulpfile.js 를 수정해야 한다.

gulpfiles.js

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('default', ['babel']); //기본 작업을 babel 로 설정

//기본 바벨 테스크
gulp.task('babel', function() {
    return gulp.src('src/*.js') //src 폴더에 있는 모든 js 파일을
        .pipe(babel({ presets: ['es2015'] })) //es2015 규격의 코드들을
        .pipe(gulp.dest('assets/js/')) //assets/js 에 컴파일 시킨다.
});
```

마찬가지로 콘솔에서 gulp 를 실행만 해도 npm run build 를 치는 것과 똑 같은 효과가 나타나긴 하지만 아직 제대로 자동화가 됐다고 보기엔 어렵다. 우리의 목표는 자바스크립트 코드의 변경을 **감시** 해서 자동으로 컴파일 하는 것이기 때문에 감시자(watch)를 설정 해야 한다.

```
...

//감시자 설정 - 기본 watch 태스크
gulp.task('watch', function() {
    gulp.watch('src/*.js', ['babel']);
});

...
```

프로젝트 폴더에서 gulp watch 를 실행하면 지정한 디렉터리(src 폴더)에 자바스크립트 파일이 추가되거나 수정되면 자동으로 컴파일 되어 assets/js 에 위치한 자바스크립트 파일들도 갱신 되는 것이 확인된다.

Webpack 을 이용한 모듈 번들링

ES6 의 문법인 import 구문을 이용하여 간단하게 사람의 이름과 피자, 맥주 값을 계산하는 모듈을 만들고 사용해보자.

/src/sum.js

```
import { name } from './client';

const pizza = 10000;
const beer = 5000;

const sum = (a, b) => a + b + '원';
console.log(`${name}님 ${sum(pizza, beer)} 입니다.`);
```

/src/client.js

```
export const name = '소민호';
```

간단히 client 모듈을 만들어 sum.js 에서 import 해서 사용하고 있는 모듈이다. node 를 이용해서 콘솔창에서 실행해보면 잘 실행 된다. 그렇다면 html 에서도 잘 실행 되는지 확인해 보기 위해서 index.html 을 만들어 확인해보자.

/index.html

```
<html>
<head>
    <meta charset="UTF-8">
    <title>sum test</title>
</head>
<body>
    <script src="./assets/js/sum.js"></script>
</body>
</html>
```

index.html 을 실행해보면 require 가 없다 라는 오류가 나면서 제대로 실행되지 않는다. 브라우저에서 오류가 나는 이유는 브라우저/클라이언트 측 자바스크립트에서는 require() 함수가 없기 때문이다. 이를 수정하려면 모듈을 하나의 파일로 만들어 활용 해야 하기 때문에 모듈을 만든 의미가 없어지게 된다. 이러한 현상을 해결하기 **위해서는 Webpack 이나 Browserify 같은 모듈 번들러**가 필요하다.

Webpack 은 모듈 번들러로써, 사용하고자하는 자바스크립트 모듈을 받아 종속성을 파악 한 후 그것들을 적절하게 모두 연결하여 정적 에셋(static asset)을 만들어 낸다. 정적 에셋이란, 어디에서 확인 하던 규격화된 모습으로 바뀌지 않는 형태로 존재하는 결과물이란 뜻이 된다.

즉 Webpack 은 자바스크립트 뿐만 아닌 여기저기 흩어져있는 미디어파일(그림, 음악, 동영상 등등)과 자바스크립트, 뷰 엔진 파일들을 한데 모아서 클라이언트에서 확인 할 수 있는 모습으로 묶어 주는 역할을 한다.

먼저 Webpack 을 설치하자.

- npm install webpack -g
- npm install webpack --save-dev

자바스크립트를 컴파일 하기 위해 Webpack 에 소스의 위치와 출력을 알려줘야 한다. 우리가 사용할 컴파일된 소스는 assets/js/sum.js 이고, Webpack 을 통해 컴파일된 코드의 위치로는 assets/webpacked/app.js 로 지정해 줄 것이다.

- webpack assets/js/sum.js assets/webpacked/app.js

Webpack 을 이용한 빌드가 완료되고 나면 html 에서는 assets/js/sum.js 대신에 assets/webpacked/app.js 를 사용 할 수 있게 된다.

```
/index.html
...
<script src="./assets/webpacked/app.js"></script>
...
```

마찬가지로 Webpack 도 gulp 을 통해 자동화가 가능하다. webpack-stream 을 이용해 자동화가 가능하다.

- npm install webpack-stream --save-dev

gulpfile.js 를 수정하여 Gulp 가 babel 태스크를 실행한 후 변경 사항들을 감지 하여 실행 하도록 하자.

/gulpfile.js

```
const gulp = require('gulp');
const babel = require('gulp-babel');
const webpack = require('webpack-stream');

gulp.task('default', ['babel']); //기본 작업을 babel 로 설정

//기본 babel 태스크
gulp.task('babel', function() {
    return gulp.src('src/*.js') //src 폴더에 있는 모든 js 파일을
        .pipe(babel({ presets: ['es2015'] })) //es2015 규격의 코드들을
        .pipe(gulp.dest('assets/js/')) //assets/js 에 컴파일 시킨다.
});
//기본 webpack 태스크
gulp.task('webpack', ['babel'], function() {
    return gulp.src('assets/js/sum.js')
        .pipe(webpack({
            output: {
                path: "/assets/webpacked",
                filename: "app.js"
            }
        }))
        .pipe(gulp.dest('assets/webpacked'));
});

//감시자 설정 - 기본 watch 태스크
gulp.task('watch', function() {
    gulp.watch('src/*.js', ['babel', 'webpack']); //watch 태스크에 webpack 도 등록
});
```

gulp watch 명령어로 gulp 를 실행시켜서 확인해 보자.

단일파일 컴포넌트(Single File Component)

Vue 의 단일 파일 컴포넌트를 사용하려면 Webpack 과 vue-loader 또는 Browserify 와 vueify 가 필요하다. 단일 파일 컴포너트 또는 Vue 컴포넌트는 템플릿과 자바스크립트, CSS 스타일을 확장자가 .vue 인 파일 하나에 캡슐화 시켜놓은 것이다. 이러한 .vue 같은 새로운 유형의 파일들을 번들링해서 사용 할 수 있도록 해주는데 Webpack 이 사용된다.

vue-cli

Webpack 설정 및 새로운 프로젝트(워크플로우)를 처음부터 만드는 것을 방지하여 미리 구성된 빌드의 형태를 만들 수 있게 도와준다. vue-cli 를 이용하면 Webpack 은 물론 vue-loader 까지 포함되어 vue 를 개발 하기에 최적화된 환경을 제공한다.

- npm install vue-cli -g

global 로 vue-cli 를 설치 하고 나서 적절한 위치에 프로젝트를 생성하면 된다.

- vue init webpack stories-classic-project

프로젝트 설정 시 Rumtime + Compiler 빌드만 선택하고 전부 n 을 입력하여 설치 하지 않는다. 프로젝트 구성이 끝나면 vue 에서 제공하는 임시 서버를 실행시켜 웹 페이지가 제대로 작동하는지 확인해 보면 된다

- cd stories-classic-project
- npm install
- npm run dev

npm run dev 를 치고나서 어플리케이션이 localhost:8080 에서 구동 중이란 말이 나올 때 까지 기다리자. 설치가 완료되면 브라우저를 열어서 localhost:8080 으로 접속해 보도록 하자

vue-cli 프로젝트의 구조

vue-cli 를 이용해 기본적인 프로젝트 구조를 완성 시켰으면, 우리는 앞으로 이러한 파일들을 다뤄야 한다.

1) index.html
2) main.js
3) src 및 src/components 디렉터리 아래의 파일들

먼저 index.html 을 보도록 한다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>stories-classic-project</title>
  </head>
  <body>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

이미 컴포넌트가 포함된 기본적인 구조 라는 것을 확인 할 수가 있다. 기본적으로 Webpack 이 스크립트를 번들링 한다음 출력된 스크립트를 주석 부분에 자동으로 삽입하기 때문에 우리가 손수 추가시킬 필요는 없다.

그 다음은 src/components 디렉터리에 있는 HelloWorld.vue 파일이다. vue 파일은 크게 template 부분, script 부분, style 부분 세가지로 나뉘게 된다.

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <h2>Hello CLI</h2>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  data () {
    return {
      msg: 'Hello Cli!'
    }
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h1, h2 {
  font-weight: normal;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
```

```
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

<script></script> 태그 안에는 컴포넌트에서 사용할 data 만 들어있고 템플릿을 따로 정의 할 필요는 없다. template 블록이 존재 하면 자동으로 바인딩 되기 때문이다. <template></template> 블록은 컴포넌트의 템플릿을 정의하며 HTML 의 <template-helloworld> 같은 태그로 생각하면 된다.

<script></script>에 있는 부분은 컴포넌트 옵션 객체를 내보내는 구간으로써 **반드시 옵션 객체를 내보내야 한다**. 하나의 스크립트 태그 파일에 존재해야 한다.

어플리케이션의 메인 템플릿이 담긴 App.vue 파일은 src 에 위치한다. **일반적으로 다른 컴포넌트를 포함**하는 역할을 한다.

```
<template>
  <div id="app">
    <img src="./assets/logo.png">
    <HelloWorld/>
  </div>
</template>
<script>
import HelloWorld from './components/HelloWorld'
export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>
<style>
...
</style>
```

일단 스크립트 쪽을 보면 HelloWorld 컴포넌트를 포함시켜 components 객체에 추가 시키고 있다. 템플릿을 보면 <HelloWorld /> 태그를 이용해 HelloWorld 템플릿을 화면에 표시 하고 있는 것이 확인된다.

main.js 파일은 메인 스크립트로써 vue 모듈을 포함시키고 App 컴포넌트도 포함시키고 있는 것이 확인된다. 여기에서 Vue 인스턴스를 만들고 App 인스턴스를 만들어내고 있다.

스크립트나 컴포넌트를 전역적으로 사용해야 한다면 main.js 에 작성하면 된다.

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

Vue.config.productionTip = false

/* eslint-disable no-new */
new Vue({
  el: '#app',
  template: '<App/>',
  components: { App }
})
```

template: '<App /> 옵션은 컴포넌트의 템플릿 출력을 나타내며 index.html 에 삽입된다. 지금부터 실제 시나리오에 단일 파일 컴포넌트를 적용해 보자. 로그인과 회원 가입 페이지를 만들어 볼 것이다. HelloWorld 컴포넌트는 필요 없기 때문에 삭제한다.

먼저 부트스트랩을 전역적으로 포함해서 모든 컴포넌트에서 스타일을 사용할 수 있도록 해보자. index.html 파일을 수정한다.

```
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>stories-classic-project</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
</head>
```

index.html 에 부트스트랩 css 를 임포트 함에 따라서 어디에서든 부트스트랩 테마를 사용 할 수 있게 된다.

이어서 간단한 /src/components/Login.vue 파일을 생성하고 로그인 폼을 만들어 보자.

login.vue

```
<template>
  <div id="login" style="width:300px; display:inline-block;">
    <h2>Sign in</h2>
    <div class="form-group form-horizontal">
      <input type="email" placeholder="Email address" class="form-control">
    </div>
    <div class="form-group">
      <input type="password" placeholder="Password" class="form-control">
    </div>
    <button class="btn">Sign in</button>
  </div>
</template>

<script>
  export default {
    created() {
      console.log('login')
    }
  }

</script>
```

이어서 App.vue 에 로그인 컴포넌트를 추가하면 된다.

```
<template>
  <div id="app" class="container">
    <img src="./assets/logo.png">
    <div>
      <login></login>
    </div>
  </div>
</template>
<script>
  import Login from './components/Login'

  export default {
    name: 'app',
    components: {
      Login
    }
  }
</script>
```

실행 후 확인하면 방금 만든 login 컴포넌트가 화면에 표시된다. 이어서 회원 가입 화면을 작성하자

Register.vue

```
<template>
  <div id="register" style="width:300px; display:inline-block;">
    <h2>Register</h2>
    <div class="form-group">
      <input type="text" placeholder="First Name" class="form-control">
    </div>
    <div class="form-group">
      <input type="text" placeholder="Last Name" class="form-control">
    </div>
    <div class="form-group ">
      <input type="email" placeholder="Email address" class="form-control">
    </div>
    <div class="form-group">
      <input type="password" placeholder="Pick a password" class="form-control">
    </div>
    <div class="form-group">
      <input type="password" placeholder="Confirm password" class="form-control">
    </div>
    <button class="btn">Register</button>
  </div>
</template>
<script>
  export default {
    created() {
      console.log('register')
    }
  }
</script>
```

작성이 끝났다면 login 과 마찬가지로 App.vue 에 추가하여 확인 해 보자.

스토리를 보여주는 컴포넌트인 Stories.vue 를 만들어 보자

```
<template id="stories">
  <ul class="list-group">
      <li v-for="story in stories" class="list-group-item">
          {{ story.writer }} said "{{ story.plot }}"
          Story upvotes {{ story.upvotes }}
      </li>
  </ul>
</template>

<script>
export default {
  data(){
      return{
          stories:[
              {
                  plot:'My horse is amazing.',
                  writer:'Mr. Weebl',
                  upvotes:28,
                  voted:false
              },{
                  plot:'Narwhals invented Shish Kebab.',
                  writer: 'Mr. Weebl',
                  upvotes: 8,
                  voted:false
              },{
                  plot:'The dark side of the Force is stronger',
                  writer:'Darth Vader',
                  upvotes: 52,
                  voted:false
              },{
```

```
                plot: 'One does not simply walk into Mordor',
                writer:'Boromir',
                upvotes:74,
                voted:false
            }
        ]
    }
  }
}
</script>
```

App.vue 에 넣어놓고 테스트 해보자.

이제 로그인, 회원가입, 컨텐츠 부분이 완성 되었다. 이번엔 가장 인기 있는 컨텐츠를 보여주기 위한 컴포넌트를 작성하는데, 이번에 작성한 컴포넌트는 어디에서든 보여 줄 것이다.

famous.vue

```
<template>
  <div id="register">
    <h2>Trending stories
      <strong>({{ famous.length }})</strong>
    </h2>
    <ul class="list-group">
      <li v-for="story in famous" class="list-group-item">
        {{ story.writer }} said "{{ story.plot }}". Story upvotes {{ story.upvotes }}.
      </li>
    </ul>
  </div>
</template>
```

```
<script>
  export default {
    computed: {
      famous: () {
        return this.stories.filter(function (item) {
          return item.upvotes > 50;
        })
      }
    },
    data() {
      return {
        stories: [{
          plot: 'My horse is amazing.',
          writer: 'Mr. Weebl',
          upvotes: 28,
          voted: false
        }, {
          plot: 'Narwhals invented Shish Kebab.',
          writer: 'Mr. Weebl',
          upvotes: 8,
          voted: false
        }, {
          plot: 'The dark side of the Force is stronger',
          writer: 'Darth Vader',
          upvotes: 52,
          voted: false
        }, {
          plot: 'One does not simply walk into Mordor',
          writer: 'Boromir',
          upvotes: 74,
          voted: false
```

```
        }]
      }
    }
  }

</script>
```

stories 배열은 famous.vue 에서도 하드 코딩 하였다. 데이터 자체는 똑같지만 computed 필드를 사용해서 일정 수 이상의 투표를 받은 컨텐츠만 보여주게 설정 해 놓았다. 추후에 stories 배열을 정의하고 모든 컴포넌트에서 공유 하는 방법에 대해 알아보자.

먼저 Register.vue 에 famous.vue 를 넣어서 가입 때 인기있는 이야기를 볼 수 있도록 해보자. App.vue 내부에서 했던 것과 똑같이 하면 된다. famous 컴포넌트를 Register 의 제일 아래 쪽에 추가 시켜 주자.

Register.vue

```
...
<div class="form-group">
    <input type="password" placeholder="Confirm password" class="form-control">
</div>
    <button class="btn">Register</button>
    <famous></famous>
</div>

<script>
  import Famous from './Famous.vue'
  export default {
    components: {
      Famous
    },
    created() {
```

```
      console.log('register')
    }
  }
</script>
```

App.vue 에 register 를 추가 시킨 후 확인해 보면 회원가입 페이지와 famous 컴포넌트가 같이 보이는 것을 확인 할 수 있다.

중복 상태 제거

위의 famous 컴포넌트와 stories 컴포넌트에는 똑 같은 데이터인 stories 배열을 동시에 사용 하고 있다. 이번엔 데이터를 공유하는 방법에 대해 이야기 한다. 컴포넌트들은 두 가지 방법을 이용해서 데이터를 공유한다

- 컴포넌트 프로퍼티를 이용
- 전역 저장소 이용

먼저 프로퍼티를 이용한 공유에 대해서 알아보자. 제일 먼저 해야 할 일은 stories 배열을 App.vue 로 옮기는 일을 해야 한다.

App.vue

```
export default {
    name: 'app',
    components: {
      Login,
      Register,
      Stories
    },
    data() {
      return {
        stories: [{
          plot: 'My horse is amazing.',
          writer: 'Mr. Weebl',
          upvotes: 28,
```

```
        voted: false
      }, {
        plot: 'Narwhals invented Shish Kebab.',
        writer: 'Mr. Weebl',
        upvotes: 8,
        voted: false
      }, {
        plot: 'The dark side of the Force is stronger',
        writer: 'Darth Vader',
        upvotes: 52,
        voted: false
      }, {
        plot: 'One does not simply walk into Mordor',
        writer: 'Boromir',
        upvotes: 74,
        voted: false
      }]
    }
  }
}
</script>
```

App.vue 에 없었던 data() 가 생기고 stories 에 있던 내용을 옮겨 왔다. 이제 Stories.vue 와 Famous.vue 에서는 App.vue 로부터 stories 를 받아내야 한다. 즉, 'props'를 이용해 전달 해야 한다.

Stories.vue

```
<script>
export default {
    props:['stories']
}
</script>
```

Stories.vue 에서 stories 프로퍼티를 받기 때문에 App.vue 에서 stories 배열을 넣어 주어야 한다.

App.vue
```
<stories :stories="stories"></stories>
```

Famous.vue 같은 경우에는 App.vue 에서 사용되는 컴포넌트가 아닌 Register.vue 에서 사용되는 컴포넌트 이기 때문에 Register.vue 를 통해서 넣어주면 된다.

App.vue
```
<register :stories="stories"></register>
```

Register.vue
```
export default {
    props: ['stories'],
...
```

Famous.vue
```
export default {
    props: ['stories'],
```

정상적으로 잘 출력이 되는 것이 확인된다. 하지만 Famous 컴포넌트가 독립적이지 않고 Register 컴포넌트에 대해 포함 되어야만 사용 할 수 있는 구조이기 때문에 그다지 효율적인 구조라고 볼 순 없다. 마찬가지로 Register 컴포넌트도 사용하지도 않을 stories 프로퍼티를 가지고 있는 것이 꽤나 만족스럽지는 않다. 게다가 데이터 전달이 매우 깊은 자식 컴포넌트까지 미치게 된다면 불필요한 코드가 굉장히 많아질 것이다.

따라서 story 에 대한 데이터만 모아 놓을 store.js 를 만들어 필요 할 때마다 사용 할 것이다.

store.js

```
export const store = {
    stories: [{
        plot: 'My horse is amazing.',
        writer: 'Mr. Weebl',
        upvotes: 28,
        voted: false
    }, {
        plot: 'Narwhals invented Shish Kebab.',
        writer: 'Mr. Weebl',
        upvotes: 8,
        voted: false
    }, {
        plot: 'The dark side of the Force is stronger',
        writer: 'Darth Vader',
        upvotes: 52,
        voted: false
    }, {
        plot: 'One does not simply walk into Mordor',
        writer: 'Boromir',
        upvotes: 74,
        voted: false
    }]
}
```

이제 stories 프로퍼티를 모든 파일에서 제거 해야 한다. 데이터 저장 방법을 변경 했기 때문에 충돌이 발생 하면서 빌드가 실패 할 수도 있다. 본격적으로 컴포넌트에 store 데이터를 집어 넣어 주자.

Stories.vue

```
<template id="stories">
  <ul class="list-group">
      <li v-for="story in store.stories" class="list-group-item">
          {{ story.writer }} said "{{ story.plot }}"
          Story upvotes {{ story.upvotes }}
      </li>
  </ul>
</template>

<script>
import { store } from '../store.js'
export default {
   data(){
        return {
            store
        }
    }
}
</script>
```

store 객체를 import 하고 있으므로 컴포넌트의 템플릿도 수정해야 한다. 이 상황에서 템플릿 수정을 하기 싫다면

```
data(){
       return {
           stories: store.stories
       }
   }
```

같은 형태로 리턴 하면 template 은 건들 필요가 없다. 동일한 작업을 Famous.vue 에서도 해주자

Famous.vue

```
import { store } from '../store.js'
  export default {
   data() {
      return {
        stories: store.stories
      }
    },
    computed: {
      famous: function () {
        return this.stories.filter(function (item) {
          return item.upvotes > 50;
        })
      }
    }
  }
```

컴포넌트 교체

이제부터 특정 컴포넌트를 교체해 가며 사용할 것이다. 지금까지는 App.vue 에 모든 것들을 표시 해왔기 때문에 전부 중첩 되어진 형태로 표시가 되었으나, 컴포너트가 페이지에서 동시에 렌더링 되지 않도록 **컴포넌트를 동적으로 교체**할 방법을 찾아야 한다.

동적 컴포넌트 - <component>와 is 속성

<component>는 예약 태그로써 동일한 컴포넌트 교체 지점(마운트 지점)을 만들고 여러 컴포넌트를 동적으로 교체 할 수 있다. 이 때 사용 되는 속성이 **is 속성**이다.

새로운 프로젝트를 만들고 component 교체를 테스트 해보자. 프로젝트의 이름은 dynamic-component 로 하고, App.vue 의 HelloWorld 컴포넌트를 <component> 태그와 is 속성을 이용해 사용해 보자. 먼저 App.vue 과 HelloWorld.vue 의 소스를 바꾸고 시작하자.

/src/App.vue

```
...
<template>
  <div id="app">
    <img src="./assets/logo.png">
     <component is="HelloWorld"></component>
    <p>
      컴포넌트 교체 입니다.
    </p>
  </div>
</template>
...
```

/src/components/HelloWorld.vue

```
<template>
  <div class="hello">
```

```
    <h1>{{ msg }}</h1>
    <h2>Hello World 컴포넌트 입니다!</h2>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  data () {
    return {
      msg: 'HelloWorld '
    }
  }
}
</script>
```

다음은 HelloWorld 컴포넌트를 대체 해서 사용할 Greet.vue 파일을 생성하고 작성한다.

/src/components/Greet.vue

```
<template>
  <div class="greet">
    <h1>{{ msg }}</h1>
    <h2>Greet 컴포넌트 입니다!</h2>
  </div>
</template>

<script>
export default {
  name: 'Greet',
  data () {
    return {
      msg: 'Greeting world!'
```

```
    }
  }
}
```

기본적인 골격은 HelloWorld 컴포넌트와 같지만, 모양만 약간 다르게 한다. 이제부터는 App.vue 에서 HelloWorld 컴포넌트와 Greet 컴포넌트를 서로간에 교체 해볼 수 있다. component 태그의 is 속성을 바꿔서 이벤트에 따라 각각 다른 컴포넌트를 교체 하는 코드를 만들어보자.

/src/App.vue

```
<template>
  <div id="app">
    <img src="./assets/logo.png">
    <!-- currentComponent의 값이 변경되면 컴포넌트가 바뀐다.-->
     <component is="currentComponent"></component>
    <p>
      컴포넌트 교체 입니다.
    </p>
    <a class="btn btn-primary" @click="currentComponent='HelloWorld'">Show Hello</a>
    <a class="btn btn-danger" @click="currentComponent='Greet'">Show Greet</a>

  </div>
</template>
<script>
import HelloWorld from './components/HelloWorld'
import Greet from './components/Greet'

export default {
  name: 'app',
  components: {
    HelloWorld, Greet
  },
  data(){
    return {
      currentComponent: 'HelloWorld' // 기본 컴포넌트는 HelloWorld 컴포넌트로 설정
    }
  }
}
</script>
```

버튼 클릭을 통해 컴포넌트가 변환 되는 것을 확인 할 수 있다. 테스트가 끝났으므로 다시 story 프로젝트로 돌아가 내비게이션을 통한 컴포넌트 교체를 해보자.

```
<template>
  <div id="app" class="container">
    <img src="./assets/logo.png">
    <h1>동적 컴포넌트 활용하기</h1>
    <ul class="nav nav-tabs">
      <!-- 조건에 따라 'active' 클래스를 설정 -->
      <li v-for="page in pages" :class="isActivePage(page) ? 'active':''">
        <!-- 링크를 이용해 탭 사이를 이동하기 -->
        <a @click="setPage(page)">{{ page | capitalize }}</a>
      </li>
    </ul>
    <component is="activePage"></component>
  </div>
</template>

<script>
  import Vue from 'vue' // 뷰 객체를 활용하기 위해 import
  import Login from './components/Login'
  import Register from './components/Register'
  import Stories from './components/Stories'

  Vue.filter('capitalize', function(value){
    return value.charAt(0).toUpperCase() + value.substr(1);
  })

  export default {
    name: 'app',
```

```
    components: {
      Login,
      Register,
      Stories
    },
    data(){
      return {
        pages:[
          'stories',
          'register',
          'login'
        ],
        activePage:'stories'
      }
    },
    methods:{
      setPage(newPage){
        this.activePage = newPage;
      },
      isActivePage(page){
        return this.activePage === page;
      }
    }
  }

</script>
```

pages 배열에는 렌더링 하려는 컴포넌트가 들어있고 v-for 디렉티브를 이용해 각 탭을 생성하고 있다. 각 탭을 열기 위해서 setPage 메소드를 사용하였고 activePage 프로퍼티는 처음에 stories 로 시작한다. 탭을 클릭하면 표시하려는 컴포넌트의 이름을 반영 하기 위해 activePage 가 변경되는 모습이 확인된다.

어떤 탭이 활성화 되어야 하는지 결정하기 위해 삼항 연산자 (?)를 적용했다. 여기서는 현재 activePage 프로퍼티가 현재 컴포넌트의 이름과 일치하느냐에 따라 active 클래스를 추가한다.

capitalize 필터는 컴포넌트의 이름을 이용해 탭 이름을 구성 하면서 맨 앞글자를 대문자로 만들기 위해 사용하였다.

Vue 라우터

일반적으로 라우팅은 어플리케이션이 클라이언트 요청에 응답하는 방법을 결정 짓는 것을 말한다. 웹 브라우저의 모든 요청은 라우팅 없이 어플리케이션으로 전달되지 않는다. 라우터는 웹서버가 적절하고 정확한 사용자의 정보를 가져 올 수 있게 돕고, 해당 요청에 대한 응답을 한다.

마찬가지로 이전에 했던 컴포넌트 교체는 라우팅을 위한 것이었다. Vue.js 의 공식 라우터를 vue-router 라고 한다. vue-router 는 vue.js 코어와 긴밀하게 통합되어 단일 페이지 어플리케이션을 손쉽게 제작 할 수 있도록 도와준다.

- npm install vue-router –save

먼저 라우팅에 대한 테스트를 진행 해야 하므로 vue-router-test 프로젝트를 만들고 나서 전역 영역을 담당하는 main.js 를 수정해야 한다.

/src/main.js

```
import Vue from 'vue'
import App from './App'
import VueRouter from 'vue-router'

Vue.use(VueRouter);
Vue.config.productionTip = false

/* eslint-disable no-new */
new Vue({
    el: '#app',
    template: '<App/>',
    components: { App }
})
```

use 함수를 이용하면 vue.js 플러그인을 사용 할 수 있다. vue-router 모듈도 일종의 플러그인이다. **첫 번째 할 일은 라우터 인스턴스**를 만드는 일이다.

추가 옵션 설정도 할 수 있지만 추후에 진행 하기로 하고 먼저 아주 간단하게 main.js 에 라우터 인스턴스를 만들자.

/src/main.js

```
...
const router = new VueRouter({
    routes // routes: routes의 축약형
});
```

라우터 인스턴스를 만들었으면, 다음은 routes 배열을 만들어 vue-router 플러그인에 등록 할 컴포넌트 들을 설정하면 된다.

```
...
//컴포넌트 import
import Hello from '. components/Hello.vue';
import Login from '. components/Login.vue';
...
Vue.use(VueRouter);

const routes = [
    { path: '/', component: Hello },
    { path: '/login', component: Login }
];

...
```

이어서 루트 Vue 인스턴스(new Vue({ ... }))에 router 옵션을 반드시 지정하자. 이에 따라서 전체 어플리케이션에서 라우터를 인식하게 된다.

```
/* eslint-disable no-new */
new Vue({
    el: '#app',
     router,
    template: '<app></app>', // id 가 app 인 div 인식 시켜서 템플릿으로 대체하기
    components: { App }
})
```

본격적으로 vue-router 를 이용하기 위해 App.vue 에서 router-link 를 만들어 주자.

```
/src/App.vue
<div id="app">
    <img src="./assets/logo.png">
    <h1>라우터 테스트</h1>
    <router-link to="/"> Home 으로 </router-link>
   <router-link to="/login"> 로그인 으로 </router-link>
   <router-view></router-view>

</div>
```

router-link 는 사용자가 라우터가 활성 된 어플리케이션에서 이동을 할 수 있도록 만드는 컴포넌트이다. **to 프로퍼티는 main.js 에 정의된 routes 의 경로와 일치** 하는 곳에 있는 컴포넌트를 정의하면 된다. 기본적으로 <a href>를 기반으로 만들어 져 있다는 것을 예상 해 볼 수 있다.

이름을 가지는 라우트

지금은 소규모 프로젝트 이지만, 프로젝트가 커져갈수록 많은 옵션이 필요 할 때가 있을 것이다. 예를 들어 /login 을 /signup 이라고 바꾸기로 했다면 로그인으로 접근하는 모든 경로를 바꿔야 하기 때문에 매우 힘들어 진다. 이러한 현상들을 방지 하기 위해서 라우터의 uri 에 별명을 붙여 준다고 생각하자. 먼저 main.js 를 열어서 routes 객체를 수정해야 한다.

/src/main.js

```
const routes = [{
        name: 'home',
        path: '/',
        component: Hello
    },
    {
        name: 'login',
        path: '/login',
        component: Login
    }
]
```

name 프로퍼티를 추가해서 라우트에 이름을 부여할 수 있으며, 나중에 링크를 위한 식별자로 사용된다.

/src/App.vue

```
<img src="./assets/logo.png">
    <h1>라우터 테스트</h1>
    <router-link :to="{ name: 'home'}"> Home 으로 </router-link>
    <router-link :to="{ name: 'login'}"> 로그인 으로 </router-link>
    <router-view></router-view>
```

링크의 목적지를 정의하는데 문자열(to="/home") 을 사용하는 대신 객체(:to="{ name: 'home' }")을 사용하였다.

히스토리 모드

기본적으로 vue-router 의 기본 설정은 해시 모드 이다. 해시 모드 에서는 URL 이 변할 때마다 #기호가 추가되는데 이는 해시 모드 자체가 vue-router 의 기본 기능이기 때문이다. 따라서 해시를 제거 하기 위해선 **히스토리 모드 로 변경 해야 한다.**

/src/main.js

```
const router = new VueRouter({
    mode: 'history',
    base: '/',
    routes
})
```

중첩 라우트

중첩 라우트는 다른 라우트 안에 있는 라우트 이다. 스토리를 보여주는 라우터에 두개의 라우트를 추가하는데, 하나는 모든 이야기를 보여주기위한 라우터이고, 다른 하나는 인기있는 이야기만 보여주는 라우트 이다. 그리고 이 둘을 감싸는 페이지 까지 추가 해 주자

- StoriesAll.vue
- StoriesFamous.vue
- StoriesPage.vue

두 개의 파일을 추가하고 main.js 에서 불러오자

```
// The Vue build version to load with the `import` command
// (runtime-only or standalone) has been set in webpack.base.conf with an alias.
import Vue from 'vue'
import App from './App'

//컴포넌트 import
import Hello from './components/Hello.vue'
import Login from './components/Login.vue'
import StoriesPage from './components/StoriesPage.vue'
import StoriesAll from './components/StoriesAll.vue'
import StoriesFamous from './components/StoriesFamous.vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)

const routes = {
        path: '/',
        name: 'home',
        component: Hello
    },
    {
```

```
        path: '/login',
        name: 'login',
        component: Login
    },{
        path: './stories',
        component: StoriesPage,
        children: [{
            path: '',
            name: 'stories.all',
            component: StoriesAll
        }, {
            path: 'famous',
            name: 'stories.famous',
            component: StoriesFamous
        }]
    }
]

const router = new VueRouter({
    mode: 'history',
    base: '/',
    routes
})

/* eslint-disable no-new */
new Vue({
    el: '#app',
    router,
    template: '<App/>',
    components: { App }
})
```

예제어서 StoriesAll 의 path 속성이 ''로 되어 있는데, 이는 하위 라우트의 기본 라우트를 의미하며, URL 이 /stories 와 일치할 때 렌더링 된다는 것을 의미한다. '/'를 사용해 기본 라우트를 정의 할 수도 있다

Wrapper 컴포넌트(단순히 다른 컴포넌트를 감싸는 컴포넌트)인 StoriesPage.vue 를 작성하자

```
<template>
  <div>
      <h2>Stories</h2>
      <!-- navigation -->
       <router-link :to="{ name: 'home'}" exact> Home 으로 </router-link>
       <router-link :to="{ name: 'login'}"> 로그인 으로 </router-link>
       <router-link :to="{ name: 'stories.all'}">Stories</router-link>

      <!-- route outlet -->
      <router-view></router-view>
  </div>
</template>
```

래퍼 컴포넌트인 StoriesPage.vue 에는 자식 컴포넌트의 내용을 렌더링 하기 위해 링크 2 개와 <router-view> 태그가 들어있다.

현재 보고 있는 페이지에 대한 링크를 강조 할 수도 있다. vue-router 는 CSS 클래스를 현재 활성화된 링크에 추가 시켜주는 기능도 있는데 내부적으로 vue-router-active 클래스를 이용한다. 우리가 해야 할 것은 CSS 에 스타일을 적용 시킬 규칙만 작성하면 된다.

```
<style>
  .router-link-active {
    color: green;
    border-bottom: 1px solid green;
  }
</style>
```

위 처럼 설정하면 Home 링크만 초록색으로 변형되는데, 이는 모든 링크의 처음이 (' / ') 로 시작하여 HOME 이 기준이 되기 때문이다. 따라서 home 에 exact 를 붙여주면 이러한 현상이 방지 된다.

active-class 속성을 사용하거나 라우터 생성자 옵션인 linkActiveClass 를 이용해 전역적으로 특정 링크에 대한 활성 클래스 이름을 변경할 수 있다.

- 링크 강조를 위해 my-active-class 라는 스타일 클래스를 만들었다고 가정
  - 특성을 활용한 방법

```
<router-link :to="{ name: 'hello'}" active-class="my-active-class" exact>Hello~</router-link>
```

  - 전역 범위 사용자 정의 활성 클래스

```
const router = new VueRouter({
    mode: 'history',
    base: '/',
     linkActiveClass:'my-active-class',
    routes
})
```

라우트 객체

파싱된 경로의 모든 정보는 라우트 컨텍스트 객체(라우트 객체)에서 접근 할 수 있다. **라우트 객체**는 router-enabled **앱의 모든 컴포넌트에 주입되어 있기 때문에 this.$route 를 이용하여 라우터에 접근 할 수 있다. 이는 라우트 이동이 일어날 때마다 갱신된다.**

프로퍼티	설명
path	현재 경로를 문자열로 표현. 항상 절대 경로를 리턴한다.
params	동적 세그먼트(:)와 별표 세그먼트의 키, 값을 포함하는 객체
query	쿼리 문자열의 키/값 쌍을 포함하는 객체.
hash	해시가 존재하는 경우 #, 존재하지 않는 경우 빈 문자열
fullPath	쿼리 및 해시를 포함한 전체 URL
matched	현재 경로의 모든 중첩 경로 세그먼트에 대한 라우트가 포함된 배열
name	현재 라우트의 이름

동적 세그먼트

파라미터를 전달 할 때 방식 중 하나가 url 파라미터(:) 이다. 마찬가지로 vue-router 는 동적 세그먼트 (:) 를 이용해 원하는 파라미터 값을 보낼 수도 있다.

```
const routes = [{
        path: ':id/edit',
        name: 'stories.edit',
        component: StoriesEdit
    }
...
]
```

지금부터 본격적으로 이야기들을 보기 위한 내용들을 작성하자. 먼저 store.js 이다.

```
store.js
export const store = {
    stories: [{
        id:1,
        plot: 'My horse is amazing.',
        writer: 'Mr. Weebl',
        upvotes: 28,
        voted: false
    }, {
        id:2,
        plot: 'Narwhals invented Shish Kebab.',
        writer: 'Mr. Weebl',
        upvotes: 8,
        voted: false
    }, {
        id:3,
```

```
        plot: 'The dark side of the Force is stronger',
        writer: 'Darth Vader',
        upvotes: 52,
        voted: false
    }, {
        id:4,
        plot: 'One does not simply walk into Mordor',
        writer: 'Boromir',
        upvotes: 74,
        voted: false
    }]
}
```

단순히 store.js 에는 임시로 id 값만 부여했다. 이어서 StoriesAll.vue 를 작성하자

/src/Stories.vue

```
<template>
  <div>
    <h3>All Stories ({{ stories.length }})</h3>
    <ul class="list-group">
      <li v-for="story in stories" class="list-group-item">
        <h4>{{ story.writer }} said "{{ story.plot }}"
          <span class="badge">{{ story.upvotes }}</span>
        </h4>
        <router-link :to="{ name:'stories.edit', params:{id: story.id}}" tag="button" class="btn btn-default" exact>
            Edit
        </router-link>
      </li>
    </ul>
  </div>
</template>
```

```
<script>
import {store} from '../store.js'
export default {
    data(){
        return {
            stories:store.stories
        }
    },
    mounted(){
        console.log('stories')
    }
}
</script>
```

경로에 id 값까지 추가 시키기 위해 router-link 에 name 뿐만 아니라 params 라는 프로퍼티를 추가해 아이디 값도 한꺼번에 전송한다. 추후 $route.params 내에서는 선택한 이야기에 대한 id 값이 나오게 될 것이다. 이어서 StoriesEdit.vue 를 작성한다.

/src/components/StoriesEdit.vue

```
<template>
  <div class="row">
      <h3>Editing</h3>
      <form>
          <div class="form-group col-md-offset-2 col-md-8">
              <input type="text" class="form-control" v-model="story.plot">
          </div>
          <div class="form-group col-md-12">
              <button @click="saveChanges(story)" class="btn btn-success">
                  Save changes
              </button>
          </div>
      </form>
  </div>
</template>

<script>
    import { store } from '../store.js';
    export default {
        props: ['id'],
        data(){
            return {
                story: {}
            }
        },
        methods: {
            isTheOne(story) {
                return story.id === this.id
            },
            saveChanges(story){
```

```
                console.log('조금 이따가 함')
            }
        },
        mounted(){
            this.story = store.stories.find(this.isTheOne);
        }
    }

</script>
```

story 의 id 를 이용해 자바스크립트의 함수인 find 메소드로 stories 배열에서 원하는 story 를 선택 해 주면 된다. 또한 router 의 동적 세그먼트 형태로 파라미터를 구성하면 문자열 형태로만 비교 하기 때문에 반드시 Number 를 이용하여 숫자로 바꿔 줘야 한다. 괜찮은 방법 중 하나는 라우터에서 미리 전달 받을 파라미터를 Number 형태로 바꿀 수 있게 하는 방법이 있다. main.js 를 수정해 주자

```
,
            {
                path: ':id/edit',
                props: (route) => ({ id: Number(route.params.id) }),
                name: 'stories.edit',
                component: StoriesEdit
            }
```

라우트 별칭(alias 부여하기)

라우팅 위한 url 을 부여 할 때는 경로를 명확하고 대표성을 띄게 만드는 것이 좋은데, url 자체가 많아질수록 관리 하기가 힘들어 진다. 인기 있는 글만 보여주는 famous 에 별칭을 부여 할 수 있다.

```
{
                path: 'famous',
                name: 'stories.famous',
                alias: '/famous',
                component: StoriesFamous
            },
```

이제 /stories/famous 와 /famous 로도 인기 있는 글을 볼 수 있다.

라우트 푸시

저장이 끝나거나 어떠한 프로그래밍적인 동작인 끝난 후 이동하고 싶을 때가 있을 것이다. 이럴 때는 라우트 푸시 방식을 이용해서 아주 간단하게 해결 할 수 있다. router.push(path) 형식으로 이용하면 되는데, 이 때 동적 세그먼트가 들어가면 안된다는 뜻이다. 즉 (router.push(/stories/:id/edit) 과 같은 방식으로는 사용 할 수 없고 무조건 router.push('/stories/11/edit') 처럼 문자열로만 사용해야 한다.

라우트 푸시 방식

1) router.push({ path : '/stories/11/edit' })
2) router.push({ name: 'stories.edit', params: {id: '11'}})

StoriesEdit.vue 의 saveChanges 메소드를 수정해 주자

```
saveChanges(story){
                console.log('saved!');
                this.$router.push('/stories')
            },
goBack (){
                console.log('back')
                this.$router.back();
}
```

```
<button @click="goBack" class="btn btn-danger">
        Go Back
</button>
```

뒤로 가기 버튼을 추가하여 뒤로 가기도 구현 해 보았다.