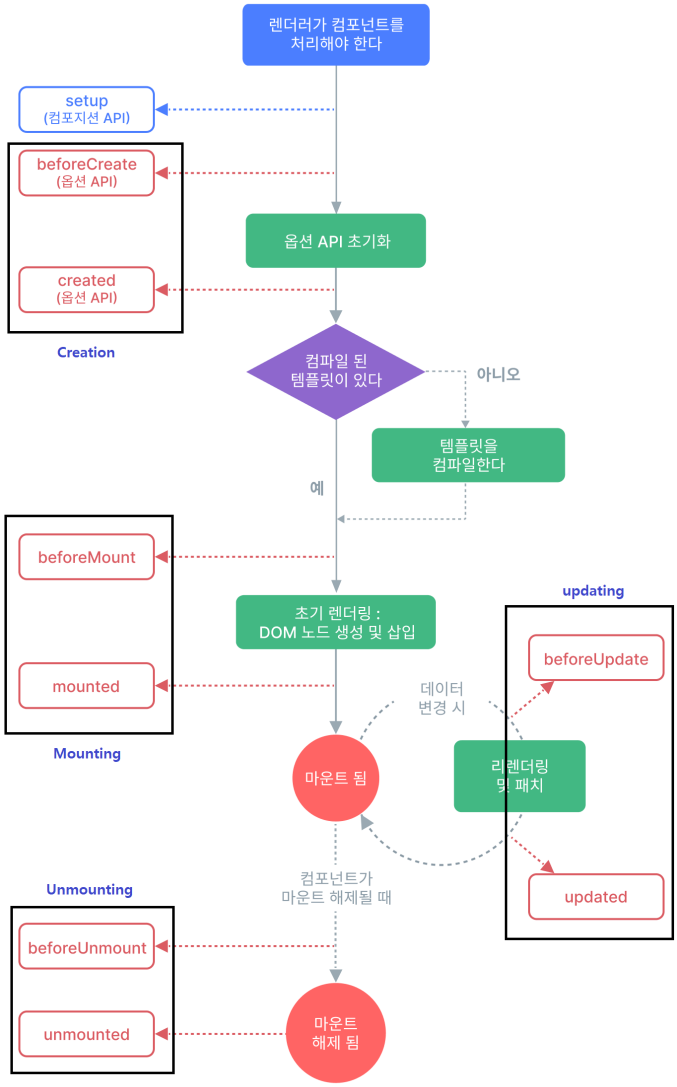
Vue day4 - Lifecycle Hooks / Router

Vue에서 Lifecycle Hooks란 뷰의 컴포넌트가 생성 되면서 부터 사라지게 되는 과정을 이야기 한다. 컴포넌트의 생명주기를 의미한다. 뷰 컴포넌트의 생명주기를 크게 나누자면 4단계로 나눌 수 있다.

1. Creation
2. Mounting
3. Updating
4. Unmounting

13. Lifecycle Hooks



컴포넌트 생성부터 마운팅 해제까지 각 단계별 어떠한 일이 일어나는지 간략하게만 살펴보자.

1. Creation : 컴포넌트 초기화 단계

- beforecrete - 뷰 인스턴스 초기화 직후 생성단계를 의미한다. 즉 뷰 인스턴스가 딱 만들어 졌지만 셋팅은 덜 된 상태를 의미한다. 코드를 예를 들자면 const app = new Vue({ }) 코드만 딱 실행 되었을 타이밍을 의미한다. (뷰 인스턴스가 막 생성된 단계)
- created - 뷰 인스턴스의 셋팅이 완료가 되었을때를 의미하며 이때 부터 data와 event가 활성화 되어 접근이 가능하다. 컴포넌트가 생성되긴 했지만 아직 화면에 붙지 않은(마운팅 되지 않은) 단계를 의미한다.

2. Mounting : 돔(DOM) 삽입 단계
   마운트(Mount)는 DOM 객체가 생성되고 브라우저에 나타나는 것을 의미한다. (화면에 붙는다 라고 표현 했다.) 말 그대로 브라우저에 '나타나는' 것이기 때문에 유저가 직관적으로 확인할 수 있는 부분이다.

- beforeMount - 컴포넌트가 DOM에 추가 되기 직전 단계를 의미 하는데 거의 의식하고 사용을 하지 않는 라이프사이클 훅이다.
- Mounted - 컴포넌트가 DOM에 추가 된 후 호출되는 단계로 $el을 사용하여 DOM에 접근이 가능한 시점이다. $el 옵션은 우리가 이미 사용을 하고 있다.

```
<div id="app">
  <p>메시지: {{ msg }}</p>
</div>

const app = new Vue({
  el: '#app',
  data: {
    msg: 'Text interpolation',
  }
})
```

3. Update - 컴포넌트에서 사용되는 속성들이 변경되는 과정 그리고 컴포넌트가 재 랜더링 되면 실행되는 라이프 사이클이다.
4. Destroy - 컴포넌트가 제거 될 때를 의미하는 라이프 사이클 이다. destroyed 단계가 되면 컴포넌트의 모든 이벤트 리스너와(@click, @change 등) 디렉티브(v-model, v-show 등)의 바인딩이 해제 되고, 하위 컴포넌트도 모두 제거되는 단계를 말한다.

이 중, 아주 고급 개발자가 아닌 이상 딱 두 가지만 주로 사용한다.

`created` : 컴포넌트가 생성 되었으나, 아직 화면에 붙기 전 그리고

`mounted` : 화면에 붙은 후

우리는 `created` 단계에서 사용 할 만한 `created()` 함수만 우선 살펴 보도록 하겠다.

우선, axios 를 설치 할 것인데, CDN 방식이 아니라 npm 을 사용해 설치할 것이다.

서버를 잠시 종료하고, 다음 명령어를 실행시킨 다음, 서버를 다시 작동 시키자.

```
$ npm i axios
```

항상 설치 후 `package.json` 을 확인하는 습관을 들여야 한다.

```
"dependencies": {
    "axios": "^1.4.0",
    "core-js": "^3.8.3",
    "vue": "^2.6.14"
  },
```

`dependencies` 는 배포 환경에서 필요한 패키지를 정의하는 섹션이며 `dependencies` 를 확인해보면, `axios` 버전 `1.4.0` 으로 잘 설치된 것이 확인된다.

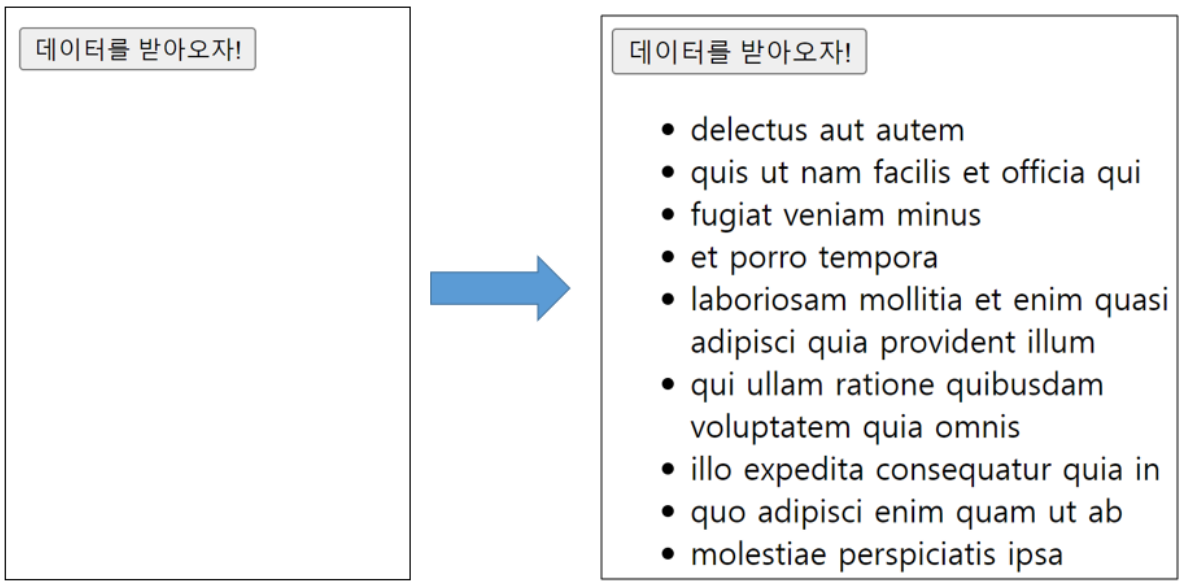
다음 코드를 `App.vue` 에 작성한다.

```
 <template>
  <div>
    <button @click="getData">데이터를 받아오자!</button>
    <ul>
      <li v-for="data in datas" :key="data.id">{{ data.title }}</li>
    </ul>
  </div>
</template>

<script>
import axios from "axios";

export default {
  data() {
    return {
      datas: [],
    };
  },
  methods: {
    getData() {
      axios
        .get("https://jsonplaceholder.typicode.com/todos")
        .then((response) => {
          this.datas = response.data;
        })
        .catch((error) => console.log(error));
    },
  },
```

```
};
</script>
```



버튼을 누르면, 오른쪽 그림과 같이 데이터를 화면에 출력하는 코드다.

일단, `<script>` 부분만 자세히 살펴보자.

```
<script>
import axios from "axios";

export default {
  data() {
    return {
      datas: [],
    };
  },
  methods: {
    getData() {
      axios
        .get("https://jsonplaceholder.typicode.com/todos")
        .then((response) => {
          this.datas = response.data;
        })
        .catch((error) => console.log(error));
    },
  },
};
</script>
```

`import ... from ...` : 외부 JavaScript 모듈을 가져올 때 사용한다. 여기 에서는 `axios` 패키지를 가져왔다. Python 은 `from ... import ...` 라고 썼었지만 JavaScript 는 `import ... from ...` 이다.

- `export default` : 외부로 나가는 객체인데, 복잡하게 생각하지 말고 `import` 구문을 제외하곤 전부 `export default` 에 작성되며, 이 안에 우리가 배운 Vue 문법이 들어간다.

- `data` : 맨 처음에 배웠던 그 `data` 가 맞다. 이 파일에서 쓸 변수들을 담는다. 다만, 맨 처음 배울 때와는 문법이 다르다. Vue CLI 에는 CLI 만의 방식이 있으므로, `data() { return { } }` 이 문법을 반드시 지켜서 작성 해야 한다.

  (vue 스타일 가이드에 따랐다.) 아래 링크의 스타일 가이드를 천천히 한번 훑어 보기를 바란다.

  https://v2.ko.vuejs.org/v2/style-guide#전체-이름-컴포넌트-이름-매우-추천함

- `methods` 우리가 배웠던 그 메서드가 맞다. 바로 data에 적용시킬 함수들이다.

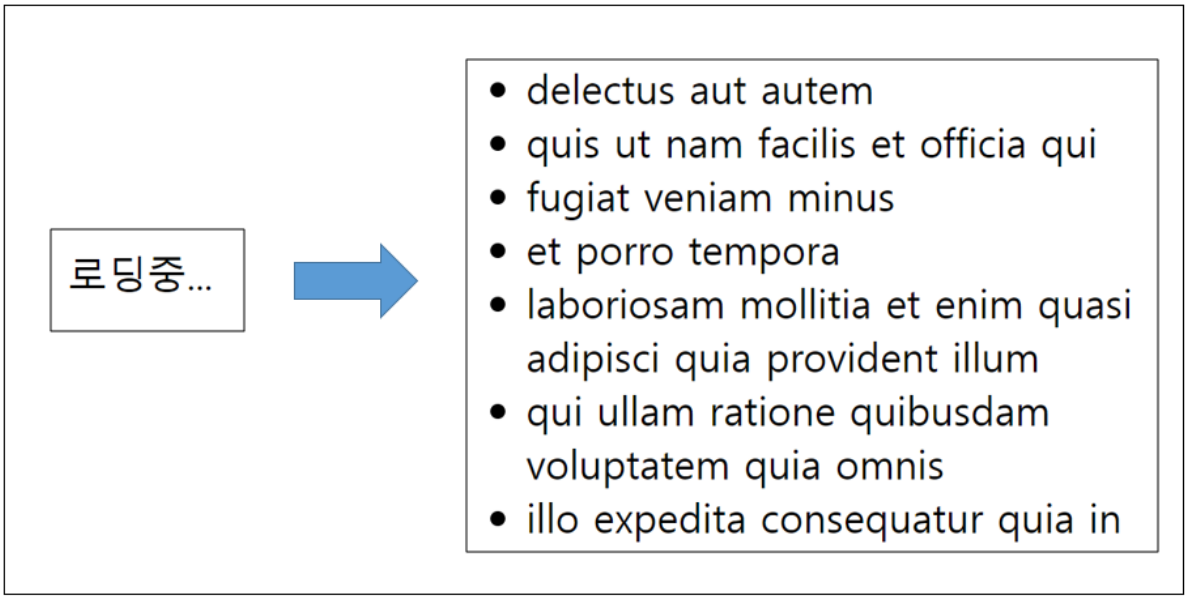
Vue 문법 복습을 했고, 기본적인 HTML, JavaScript 를 안다면 어려울 것이 없는 구문이다. 지금 여기서 시도하고자 하는 것은 클릭 없이 화면이 랜더링 되자마자 바로 함수를 자동으로 실행되는 것이다.

클릭이라는 이벤트를 발생 시키는 '트리거' 없이 vue 인스턴스 생성 후 자동 실행되는 함수를 사용할 것이다. 다음 코드를 보자.

```
<template>
  <div>
    <ul v-if="datas.length">
      <li v-for="data in datas" :key="data.id">{{ data.title }}</li>
    </ul>
    <div v-else>로딩중...</div>
  </div>
</template>

<script>
import axios from "axios";

export default {
  data() {
    return {
      datas: [],
    };
  },
  created() {
    this.getData();
  },
  methods: {
    getData() {
      axios
        .get("https://jsonplaceholder.typicode.com/todos")
        .then((response) => {
          this.datas = response.data;
        })
        .catch((error) => console.log(error));
    },
  },
};
</script>
```



처음에 `로딩중...` 문구가 잠깐 떴다가, 자동으로 로딩이 끝나면 결과가 출력 됨을 확인 할 수 있다. 주목할 코드는 이 곳이다.

```
created() {
  this.getData();
},
```

`created()` 메서드는 컴포넌트가 생성되긴 했지만 아직 화면에 붙지 않은 단계에서 자동 실행되는 메서드이다. 이 메서드는 우리가 만든 `getData` 메서드를 실행시킨다. 즉 `created` 는 `axios` 와 결합해, 서버 통신 구문에 사용되며, DOM 화면이 구성되기 전에 미리 데이터를 받아오는 역할을 한다. 그리고 이 과정은 lifecycle 중 마운팅이 실행되기 전, 템플릿이 컴파일 되기 이전 즉, 화면에 템플릿이 완벽하게 붙기 전에 `created()` 메서드가 실행되므로 컴포넌트 lifecycle 중 created 과정 중에서 코드가 수행 된다고 이해하면 완벽하다.

14. router 기초

우선 알아야 할 건, Vue.js 는 기본적으로 하나의 HTML, 즉 `index.html` 파일만 가진다는 것이다.

기존의 다른 웹 개발 방식에서 보이는 `login.html` , `signup.html` , `board.html` 등의 여러 페이지가 존재하지 않기 때문에, 다른 페이지로 이동할 수 없다. 즉, 화면은 본질적으로 하나다!

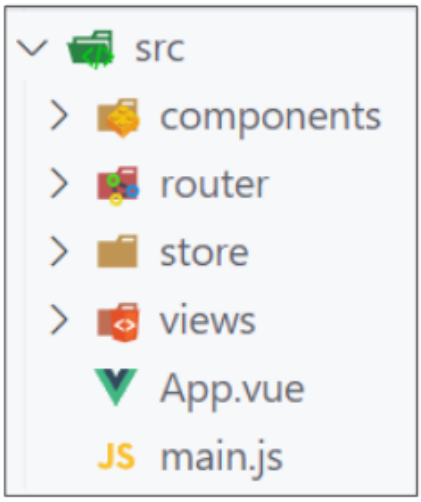
그러나, 여러가지 화면을 가진, 그리고 각 화면마다 URL 을 가진 웹 페이지는 우리에게 너무나 익숙한 개념이다. 하나의 HTML 만을 가진 Vue.js 에선 이 기능을 직접 제공하지 않고, 서드파티 패키지인 Vue Router 를 사용하도록 권장한다.

설치부터 해보자. 먼저, 기존의 `components/` 디렉터리에 존재하는 모든 파일 그리고 `App.vue` 도 삭제하고 다음 명령어를 실행하자.

```
$ vue add router
```

```
? Use history mode for router? (Requires proper server setup for index fallback in production) (Y/n)
```

위는 히스토리 모드, 즉 뒤로가기 기능을 활성화할것인지를 묻는다. `y` 를 입력하자.



새로운 `App.vue` 와, `router/` , `views/` 라는 새로운 디랙터리도 확인된다.

우선 `App.vue` 를 열어보자.

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

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}

nav {
  padding: 30px;
}

nav a {
  font-weight: bold;
  color: #2c3e50;
}

nav a.router-link-exact-active {
  color: #42b983;
}
</style>
```

여기서, `<style>` 은 CSS 영역이기 때문에 지금 필요가 없다. 전부 지워버리자.

확인해보니, `<router-link>` , `<router-view>`라는 새로운 태그 (사실은 컴포넌트) 를 확인할 수 있다.

그리고 실습을 위해 `<template>` 부분을 다음과 같이 수정하자.

```
<template>
  <div id="app">
    <nav>
      <router-link :to="{ name: 'home' }">Home</router-link> |
      <router-link :to="{ name: 'about' }">About</router-link>
    </nav>
    <router-view />
  </div>
</template>
```

`views/HomeView.vue` 를 열어서 다음과 같이 수정하자.

```
<template>
  <div>
    <h1>HomeView</h1>
  </div>
</template>
```

누가 봐도 평범한 컴포넌트다. `views/AboutView.vue` 도 다음과 같이 수정하자.

```
<template>
  <div>
    <h1>AboutView</h1>
  </div>
</template>
```

다음, `router/index.js` 를 열어서 아래와 같이 변경하자.

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import HomeView from '../views/HomeView.vue'
import AboutView from '../views/AboutView.vue'

Vue.use(VueRouter)

const routes = [
  {
    path: '/',
    name: 'home',
    component: HomeView
  },
  {
    path: '/about',
    name: 'about',
    component: AboutView,
  }
```

```
];

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

일단 이 파일이 무엇을 의미하는지 알기 전에, 서버 켜고 확인부터 해보자.

링크를 클릭하면 화면이 바뀌고, URL 도 바뀌는 것이 확인된다.

그러나, 놀랍게도 새로고침이 없다.

즉, SPA 의 특성은 계속 유지하면서, URL 로 화면 전환이 가능하도록 만든 것이 바로 Vue Router 다.

분석을 위해 `App.vue` 로 향하자.

```
<template>
  <div id="app">
    <nav>
      <router-link :to="{ name: 'home' }">Home</router-link> |
      <router-link :to="{ name: 'about' }">About</router-link>
    </nav>
    <router-view />
  </div>
</template>
```

`<router-link :to="{ name: '이름' }">`

- 생긴 것과 사용법은 a 태그 `<a href="경로">` 와 비슷해 보이지만, `:to` props 는 객체를 전달한다. 조금 더 유추해 보면, `Home` 링크를 클릭하면 `router/index.js` 에서 `name` 이 `home` 인 페이지를 보여준다는 것을 알 수 있다.

`<router-view />`

- `<router-view />` 부분은 사용자가 선택한 화면 (Home | About) 이 보여지는 영역이다.

이제 `router/index.js` 로 향하자.

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import HomeView from '../views/HomeView.vue'
import AboutView from '../views/AboutView.vue'

Vue.use(VueRouter)

const routes = [
  {
    path: '/',
    name: 'home',
    component: HomeView
  },
  {
    path: '/about',
    name: 'about',
    component: AboutView,
  }
];

const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

`import` 구문을 자세히 보면, `views/` 디렉터리의 컴포넌트를 가져온다는 것을 알 수 있다.

`routes` 는 하나의 배열인데, 각각의 엘리먼트는 객체로 이루어졌고, 다음 프로퍼티를 가진다.

- `path` : URL 을 의미한다. `/` 는 루트라는 뜻이며, `localhost:8080` 접속 시 기본적으로 보여줄 페이지다.
- `name` : `<router-link>` 에서 보낸 `name` 의 이름에 해당한다.
- `component` : 페이지 변경 요청이 왔을 때 보여줄 컴포넌트다.

즉, 사용자가 링크를 클릭하면, 각각의 `name` 에 해당하는 컴포넌트가 `<router-view />` 영역에 보여지는 것이라는 것을 유추할 수 있다. 만약 사용자가 링크를 클릭하지 않고 URL 로 직접 접속하면 `path` 에 해당하는 컴포넌트가 보여질 것이다.

용어정리를 잠시 하고가자.

router : 라우트를 관리하는 프로그램. 즉, Vue Router 그 자체를 의미함

route : 각각의 화면

그리고 각각의 route 에 연결되어 있는 컴포넌트를 앞으로 "라우트 컴포넌트" 라고 부를 것이며, 각 화면을 대표하기에 각 화면의 루트 컴포넌트가 된다. 이 컴포넌트 들은 `components/` 디렉터리가 아니라

`views/` 디렉터리에서 따로 관리할 것이다. 또한 파일 이름을 네이밍 시에는 맨 뒤에 `View` 를 붙여준다.

- 라우터를 배우기 전엔 루트 컴포넌트는 `App.vue` 였다. 그러나 앞으로 `App.vue` 는 오로지 두 가지 역할만 할 것이다.
  - CSS 전역 스타일링 - `scoped` 안 붙임!!
  - 라우트 링크 관리

즉, 앞으로 각 화면의 루트 컴포넌트의 역할은 각 라우트 컴포넌트가 대신 하게 된다.

도전Mission: `BoardView` 라우트 컴포넌트를 만들고, 링크로 접속할 수 있도록 해보자.

그러나, 화면 전환은 링크로만 이루어 지는 것이 아니다. 만약 어떤 사람이 버튼을 눌렀을 때 다른 화면으로 가고 싶다면 어떻게 하는 게 좋을까?

`HomeView` 를 아래와 같이 작성해 보자.

```
<template>
  <div>
    <h1>HomeView</h1>

    <button @click="goToBoardRoute">게시판으로 이동</button>
  </div>
</template>

<script>
export default {
  methods: {
    goToBoardRoute() {
      this.$router.push({ name: "board" });
    },
  },
};
</script>
```

위와 같이, 다른 라우트로 이동할 땐 `this.$router.push({ name: "라우트네임" })` 을 통해서 이동이 가능하다.

15. router 심화 - params

만약에 `board/1` : 1번글 상세 페이지

`board/2` : 2번글 상세 페이지

를 구현하고 싶다면 어떻게 하는 게 좋을까?

우선, views에 디테일 컴포넌트를 만든다. 이름은 `BoardDetailView` 라고 하겠다.

```
<template>
  <div>
    <h1>BoardDetailView</h1>
  </div>
</template>
```

`router/index.js` 에 다음과 같이 작성한다.

```
import BoardDetailView from "../views/BoardDetailView.vue";

const routes = [
  // ...
  {
    path: "/board/:id",
    name: "detail",
    component: BoardDetailView,
  },
];
```

`path` 부분이 좀 특별해졌는데, 받고 싶은 params 를 `:변수` 식으로 정의했다.

이 상태로 URL 에 `/board/1` 식으로 입력해보자.

이건 우리가 원했던 결과가 아니다. 적어도 몇 번 페이지인지는 나와야 구분이 가능하다.

`BoardDetailView` 를 다음과 같이 수정하자.

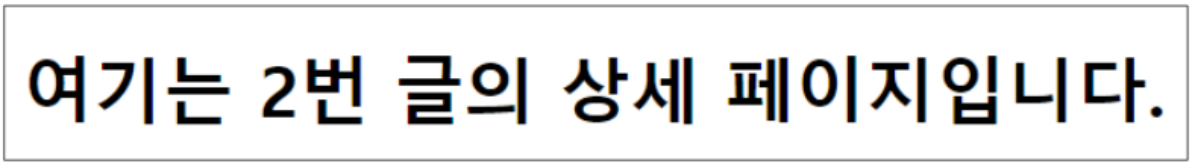
```
<template>
  <div>
    <h1>여기는 {{ num }}번 글의 상세 페이지입니다.</h1>
  </div>
</template>

<script>
export default {
  data() {
    return {
      num: this.$route.params.id,
    };
  },
};
</script>
```

`data` 를 자세히 보면, `this.$route.params` 객체에 접근해, 우리가 `:id` 라고 이름을 붙인 변수를 return 하는 것을 알 수 있다.

- 왜 여기선 `computed` 를 안 썼을까? `computed` 는 변경이 일어나는 것을 감지할 때 화면을 다시 그린다. 그러나, 현재 라우트의 params 는 절대 값을 변경 할 일이 없기 때문에 `computed`를 사용 할 필요가 없었다.

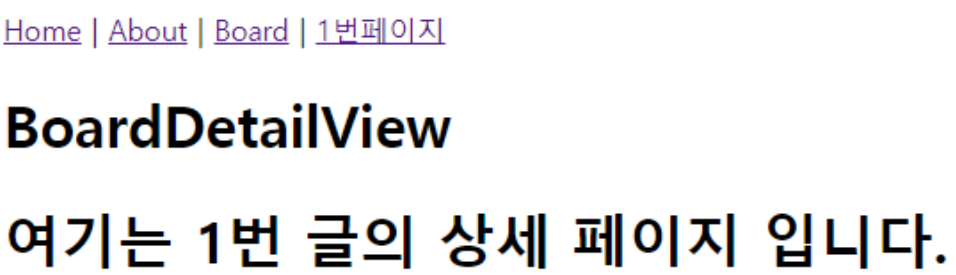
`/board/1` 과 `/board/2` 로 접속해 각각 결과를 확인하면 다음과 같다.



- `$router` 와 `$route` 는 다르다. `push` 사용시엔 `$router` 였고, params 에 접근할 땐 `$route` 다. 혼동하지 말자.
- 보통은 여기서 `created()` 와 결합해 서버에 상세 페이지의 JSON 데이터를 요청한다.

만약, 링크를 통해서 1번 페이지에 접속하려면 App.vue에 다음과 같이 작성하면 된다.

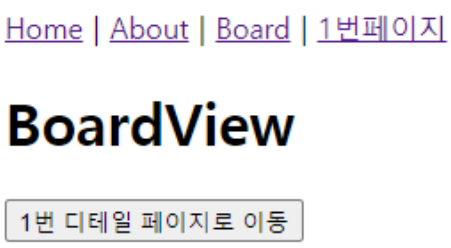
```
<router-link :to="{ name: 'detail', params: { id: '1' } }">1번페이지</router-link>
```

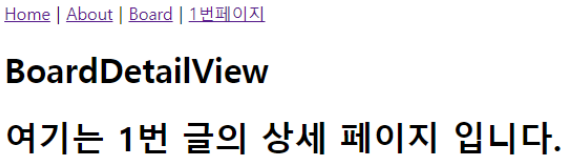


만약에 Board에서 버튼 하나 달고, 버튼을 클릭을 하여 1번 페이지에 접속하는 코드를 구현 하려면 다음과 같이 작성하면 되겠다.

```
<template>
  <div>
    <h1>BoardView</h1>
    <button @click="goToDetailone">1번 디테일 페이지로 이동</button>
  </div>
</template>

<script>
export default {
  methods: {
    goToDetailone() {
      this.$router.push({ name: "detail", params: { id: "1" } });
    },
  },
};
</script>
```





이상으로 router 를 살펴 보았다. <끝>