Day6 Router

자료	Vue
구분	Vue
과목	

오늘의 핵심 주제는 router 그리고 navigation guard 이다.

먼저 router에 대한 이해와 사용법이 익숙하게 만든 후에

navigation guard 를 학습하는 것이 좋겠다.

따라서 오늘은 router 에 대한 이해를 위해서 작성하였다.

1장 Vue Router

Vue.js는 기본적으로 하나의 HTML 파일(`index.html`)만 사용하는 SPA(Single Page Application) 구조를 기반으로 한다.

즉, 기존의 웹 개발 방식처럼 `login.html`, `signup.html`, `board.html`과 같은 여러 개의 HTML 파일을 사용하는 페이지 전환 방식이 아닌, 하나의 HTML 파일에서 컴포넌트를 동적으로 바꾸며 화면을 전환실시 한다.

하지만 대부분의 사용자에게 익숙한 웹페이지 구조는 각 화면이 고유한 URL을 가지며, 여러 개의 독립된 페이지로 구성되어 있다. 이를 Vue.js에서 구현하기 위해서는 Vue Router라는 서드파티 라이브러리를 사용해야 한다.

Vue Router는 URL에 따라 적절한 컴포넌트를 렌더링해주는 역할을 하며, 마치 전통적인 웹 페이지처럼 다중 페이지를 구성하는 것처럼 보이도록 만들어 준다.

Vue 프로젝트를 Vite로 생성할 때, Vue Router를 함께 사용할지 여부를 선택할 수 있으며, 이를 통해 초기 세팅 시부터 라우팅 구조를 편리하게 구성할 수 있다.

```
$ npm create vue@latest

프로젝트 생성시 router 에 관련 설정을 Yes 로 선택하자.
```

`src/` 에서 필요 없는 파일들을 삭제하겠다.

`src/assets/` 디렉터리부터 정리하겠다.

우선 `base.css` 와 `logo.svg` 를 삭제하고, `main.css` 의 모든 내용을 지운 후, 다음과 같이 입력한다.

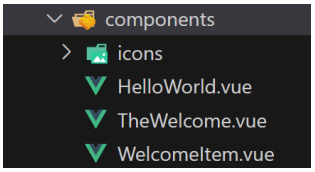
```
* {
  box-sizing: border-box;
  margin: 0;
}
```

`main.css` 는 전역 스타일링 파일이다.

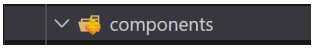
- `box-sizing` 은 박스의 크기를 화면에 표시하는 방식을 변경하는 속성이다. 테두리가 있는 경우에는 테두리의 두께로 인해서 원하는 크기를 찾기가 어려운데, `box-sizing` 속성을 `border-box`로 지정하면 테두리를 포함한 크기를 지정할 수 있기 때문에 크기를 예측하기가 더 쉽다. 기본적으로 프로젝트를 하나 만들면, 모든 엘리먼트에 이 값을 지정하고 시작한다.
- `margin: 0` 은 각 태그에 기본적으로 잡혀있는 공백을 제거해준다.

다음, `src/components/` 하위 모든 파일, 디렉터리를 삭제한다.

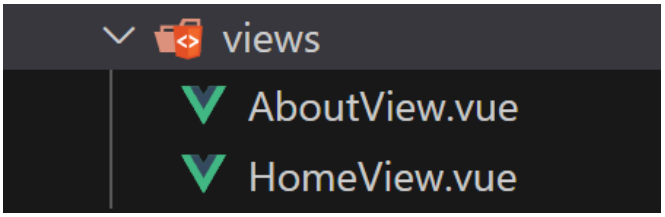
[삭제전]



[삭제후]



다음, `src/views/` 아래의 `HomeView.vue` 와 `AboutView.vue` 를 다음과 같이 변경한다.




```
<template>
  <h1>About</h1>
</template>
```

```
<template>
  <h1>Home</h1>
</template>
```

마지막으로, `src/App.vue` 를 다음과 같이 변경한다.

```vue
<script setup>
import { RouterLink, RouterView } from 'vue-router'
</script>

<template>
  <header>
    <nav>
      <RouterLink to="/">Home</RouterLink> |
      <RouterLink to="/about">About</RouterLink>
    </nav>
  </header>
  <RouterView />
</template>
```

여기까지 완료한 후, 서버를 동작시킨다.

About 링크를 클릭하면 화면이 바뀌면서,

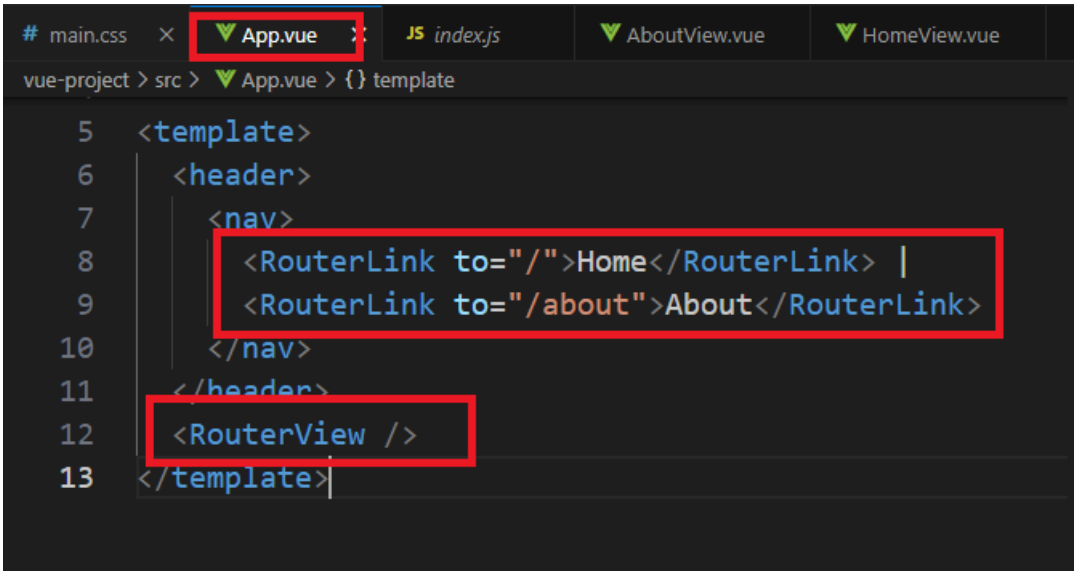
URL 도 `http://localhost:5173/` 에서, `http://localhost:5173/about` 으로 바뀌는 것을

확인 할 수 있을 것이다. 그리고 눈여겨봐야 할 부분은, 페이지가 바뀌어도 새로고침이 없다.

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

먼저, Vue Router 의 설정 파일인 `router/index.js` 로 향하자.

`router/index.js` 파일은 어떤 URL 경로에 접근했을 때, `App.vue` 에 있는 `<RouterView>` 영역에 어떤 컴포넌트를 렌더링 할지를 정의하는 파일이다.



즉, 사용자가 특정 주소로 이동하면, Vue Router가 `index.js`에 설정된 라우팅 정보를 참고하여 그에 맞는 컴포넌트를 찾아 `<RouterView>` 자리에 표시해주는 방식인 것이다.

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

const router = createRouter({
 history: createWebHistory(import.meta.env.BASE_URL),
 routes: [
  {
   path: '/',
   name: 'home',
   component: HomeView
  },
  {
   path: '/about',
   name: 'about',
   // route level code-splitting
   // this generates a separate chunk (About.[hash].js) for this route
   // which is lazy-loaded when the route is visited.
   component: () => import('../views/AboutView.vue')
  }
 ]
})

export default router
```

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

그리고 그 아래에는 router 변수 값으로 `routes` 라는 이름의 배열이 존재한다.

`routes`은 객체를 요소로 갖는 배열인데

각각의 객체속성이 의미하는 바가 무엇인지 알아보자.\

- `path` : URL 을 의미한다. `/` 는 루트라는 뜻이며, `localhost:5173` 접속 시 기본적으로 보여줄 페이지
- `name` : 해당 라우트의 이름으로, 나중에 `RouterLink` 에서 보낸 `name` 의 이름이며, 프로그래밍 방식의 라우팅 (`router.push`) 등에서 사용된다.
- `component` :해당 경로에 접근했을 때 렌더링할 컴포넌트를 지정하는 것이다.

[중요] 용어 정리를 잠시 구분하고 가자.

- router: 여러 개의 route를 관리하는 라우터 객체입니다. `createRouter`를 통해 생성하며, 애플리케이션에 라우팅 기능을 제공한다.
- route: 하나의 URL 경로와 그 경로에 대응하는 컴포넌트를 지정하는 객체다

  (URL과 컴포넌트 간의 매핑)

그리고 각각의 route 에 연결되어 있는 컴포넌트를 앞으로 **"라우트 컴포넌트"** 라고 부를 것이다.

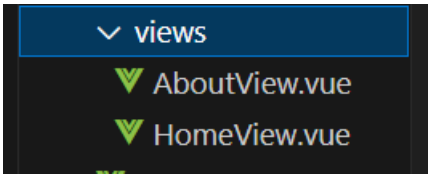
라우트 컴포넌트는 각각의 화면을 대표하며, 해당 화면의 루트 컴포넌트 역할할 것이다.

- `views/` 폴더에는 **라우트 컴포넌트(각 화면의 루트 컴포넌트)**를 위치시킵니다.

  예: `HomeView.vue` , `LoginView.vue` , `ProfileView.vue` 등
- `components/` 폴더에는 **재사용 가능한 일반 컴포넌트**들을 저장합니다.

이렇게 구조를 나누면, 프로젝트의 유지보수성과 가독성이 높아지며 각 컴포넌트의 역할이 명확해집

- views 폴더 에는 루트 컴포넌트 (라우트 컴포넌트) 들을 넣어둔다.
  - 예시: `HomeView.vue` , `LoginView.vue` , `ProfileView.vue` 등
- components 폴더에는 그 외 컴포넌트들을 넣어 둘 것이다. 재사용 가능한 일반 컴포넌트 들을 저장 한다.
  - 예: `Navbar.vue` , `UserCard.vue` , `Modal.vue` 등



라우트 컴포넌트는 각 화면의 루트가 되는 매우 특별한 컴포넌트므로 `components/` 디렉터리가

아니라 `views/` 디렉터리에서 따로 관리할 것이다. 또한 네이밍 시 파일 이름 맨 뒤에 `View` 를 붙여준다.

`router/index.js` 파일을 조금 더 살펴보자.

`router/index.js` 를 보면 아래의 사진처럼 두가지의 객체 사용 방식을 보여주는데,

`component` 부분이 다르게 보일 것이다.

`component: HomeView` 라고 적은 방식이 기존 방식이고.

`component:()⇒import..` 방식을 Lazy Loading Routes 방식이라 하며

Vue 공식문서에서는 후자의 사용방식을 권장하고 있다.

Lazy Loading을 이해하기 전에

우리가 크롬이라는 브라우저에 HTML 파일을 랜더링 하는 과정을 살펴보자.

HTML 파일을 브라우저에 랜더링 하는 과정은 SSR (서버 사이드 랜더링) 이냐?

CSR (클라이언트 사이드 랜더링) 이냐? 에 따라 조금 다르다.

웹을 공부 했다면 두개의 차이점에 대해서 분명하게 알고 있어야 한다.

(따로 생성형 AI 또는 블로그를 통해서 반드시 추가 공부하고 정리 해주세요!)

Vue는 기본적으로 CSR을 기반으로 작동하는 프레임워크다.

따라서 CSR기반 프레임워크에서 HTML 파일을 랜더링 하는 과정을 살펴볼 것이다.

예를들면 유튜브 화면 랜더링 과정을 살펴보면 다음과 같다.

1. 사용자 요청
    a. 먼저 우리는 크롬 브라우저를 켜서 URL을 입력하거나 링크를 클릭해서 프론트서버로 HTML 파일을 달라고 요청을 보낸다.

2. 서버응답 및 Vue 초기화

a. Vue 프론트서버는 (유튜브서버는) 최소한의 div 태그를 사용해서 거의 비어있는 HTML 파일을 브라우저로 반환한다. Javacript파일을 로드할 수 있도록 최소한으로 셋팅이 되어있는, 거의 비어있는 HTML파일을 브라우저에 반환한다. 예를 들자면 `<div id="app"> </div>`과 같은 빈 컨테이너가 들어간 HTML 파일을 서버에서 반환 한다.

b. 그리면 Vue.js가 해당 `<div id="app"> </div>` 에 마운트 된다. 이 시점에서는 화면에 아무것도 표시가 되지 않는다.

3. 데이터 요청

   a. 이제 Vue에서 Javascript를 통해 백앤드 서버(Django)로 비동기로 데이터를 요청한다.

      예를들어 유튜브 동여상 목록, 사용자 정보, 댓글 등을 요청한다.

4. 데이터 수신 및 랜더링

   a. Vue에서는 서버로 부터 응답 받은 JSON 형식의 데이터를 기반으로 동적 HTML 파일을 생성하거나 기존 HTML을 업데이트 해서 브라우저로 반환한다. 이 과정에서 컴포넌트가 랜더링 되고 화면에 동상상 목록, 사용자 정보, 댓글 등이 표시가 된다. 최종적으로 생성된 동적 HTML이 브라우저에 표시가 된 것이다.

- 정적 HTML 파일이란?

변경되지 않는 HTML 파일을 말한다. 예를들어 소개페이지, 회사정보 연락처 등 내용이 변경될 필요가 없는 경우를 말하며, 빠른 로드가 장점이다.

- 동적 HTML 파일이란?

실시간으로 서버에서 생성된 HTML 파일을 말한다. 예를 들어 사용자 로그인 페이지, 검색 결과 페이지 처럼 DB에서 데이터를 가져오거나 약간의 로직을 처리 후 생성된 HTML 파일을 말한다.

즉, CSR 기반 프레임워크에서는 클라이언트가(프론트서버) HTML을 동적으로 생성하고 백엔드 서버로 데이터를 요청하여 그 데이터를 기반으로 페이지의 내용이 렌더링 된다.

프론트서버에서 HTML파일을 처음 요청시 거의 빈 HTML을 반환하고, Javascript가 실행되어 서버에서 필요한 데이터를 요펑한 후, 그 데이터를 기반으로 동적 HTML을 생성하여 브라우저에 표시한다.

따라서 CSR은 초기 로딩은 빠르지만 초기 로딩 시 콘텐츠가 비어있을 수가 있다. 그래서 웹서비스 사용자는 원하는 페이지가 다 로딩이 될 때 까지 빈 화면이나 로딩이 되고 있는 페이지를 경험 할 수도 있다는 단점이 있다.

이때 중요한 것이 Lazy Loading을 적절하게 이용하는 것이다. Lazy Loading은 필요한 리소스만 먼저 로드하고, 나머지 리소스는 사용자가 해당 부분에 접근할 때 로드하도록 지연시키는 것을 말한다.

이렇게 하면 처음 페이지가 로드될 때 불필요한 JavaScript, 이미지, 스타일시트 등을 로드하지 않으므로 초기 로딩 시간을 단축 시킬 수 있다.

당장 사용하지 않을 컴포넌트는 먼저 로드하지 않기 때문에 가장 처음에 최초 컴포넌트 코드를 로딩하는 시간이 빨라지는 개념이다. 이것을 동적import 방식으로 구현했다고 표현하기도 하며, Vue 공식문서는 Vue Router 에서 모든 컴포넌트에 동적 import 를 사용할 것을 권장하고 있다.

> Vue Router
> Vue.js 공식 라우터
> https://router.vuejs.kr/guide/advanced/lazy-loading

[참고]

Vue.js는 CSR이지만 SSR도 지원한다. 하지만 이때에는 Nuxt.js도 같이 사용해야 한다는 점은 참고로만 알아두자. React.js도 마찬가지로 CSR 기반 이지만 SSR도 지원하는데 이때에는 Next.js를 함께 사용하면 된다.

- **Nuxt.js**는 Vue 생태계에서 SSR을 쉽게 구현할 수 있도록 도와주는 프레임워크
- **Next.js**는 React 생태계에서 SSR을 손쉽게 구현할 수 있도록 도와주는 프레임워크

Nuxt.js와 Next.js는 각각의 생태계에서 SSR을 쉽게 구현할 수 있도록 도와주는 프레임워크다. 이것들은 SSR의 단점인 초기 페이지 로딩 속도를 빠르게 해 줄 뿐만 아니라 CSR기반 프레임워크의 단점인 SEO 성능(검색엔진 최적화)을 크게 향상 시킬 수 있어서 현장 (실제 회사에서) 에서 많이 쓰인다.

(회사에서 만든 서비스는 구글 또는 네이버 검색이 잘 되야 할 것이다!)

자, 이제 분석을 위해 `App.vue` 로 향하자.

```
<script setup>
import { RouterLink, RouterView } from 'vue-router'
</script>

<template>
 <header>
  <nav>
```

```
    <RouterLink to="/">Home</RouterLink> |
    <RouterLink to="/about">About</RouterLink>
   </nav>
  </header>
  <RouterView />
</template>
```

가장 먼저 template 태그 안을 살펴보자.

`<RouterLink>` , `<RouterView>` 라는 태그 (사실은 컴포넌트)를 확인할 수 있다.

`<RouterLink to="/경로">`

- 생김새와 사용법은 `<a href="경로">` 와 같아 보인다.
- 유추 해보면, `Home` 링크를 클릭하면 해당 경로로 향하는 것을 알 수 있다.
- `<RouterLink>` 는 routes에 등록 된 컴포넌트와 매핑이 되며, `to` 속성으로 목표 경로를 지정한다
- 개발자 도구를 확인해 보면 기능에 맞게 HTML에서 a 태그로 랜더링 되지만, 필요에 따라 다른 태그로 바꿀 수도 있다는 점만 기억해 두자.

`<RouterView />`

- `<RouterView />` 부분은 사용자가 선택한 화면 (Home | About) 이 보여지는 영역이다.
- 실제 component가 DOM에 부착되어 보이는 자리를 의미한다.

사실 위 코드보다 아래 코드처럼 path 가 아니라 name 을 호출을 할 수도 있다.

```
<script setup>
import { RouterLink, RouterView } from 'vue-router'
</script>

<template>
  <header>
    <nav>
      <!-- <RouterLink to="/">Home</RouterLink> |
      <RouterLink to="/about">About</RouterLink> -->

      <RouterLink :to="{ name: 'home' }">Home</RouterLink> |
      <RouterLink :to="{ name: 'about' }">About</RouterLink>

    </nav>
  </header>
  <RouterView />
</template>
```

`<RouterLink :to="{ name: '이름' }">` 으로 사용 하는 것을 알 수 있다. 역시 생긴 것과 사용법은 a 태그 `<a href="경로">` 와 비슷하다.

단, `name` 사용 시엔 반드시 `v-bind:` 를 사용해야 한다. 즉, `to` 가 아니라 `:to` 이며, `RouterLink` 컴포넌트에 `props` 로 객체를 내려보낸다.

즉, 사용자가 링크를 클릭하면, 각각의 `name` 에 해당하는 컴포넌트가 `<RouterView />` 영역에 보여진다는 것을 유추할 수 있다. 만약 사용자가 링크를 클릭하지 않고 URL 을 통해 직접 접속하면 `path` 에 해당하는 컴포넌트가 보여질 것이다.

Vue Router 는 공식적으로 `path` 보다 `name` 을 추천한다.

<RouterLink to="/about">About</RouterLink> 을 사용하기 보다는

<RouterLink :to="{ name: 'about' }">About</RouterLink> 를 사용하기를 권장한다.

다음과 같은 장점 때문이다.

> Vue Router
> Vue 3에 필요한 최신 공식 라우터
> https://router.vuejs.kr/guide/essentials/named-routes.html

- 하드코딩된 URL 없음
- `params` 를 자동 인코딩/디코딩
- URL에 오타 발생 방지

따라서, 앞으로 `name` 방식의 Vue Router 사용 방법으로 설명을 이어 나가겠다.

[도전] : `BoardView` 라우트 컴포넌트를 만들고, 링크로 접속할 수 있도록 해보자. **(직접 해보기)**

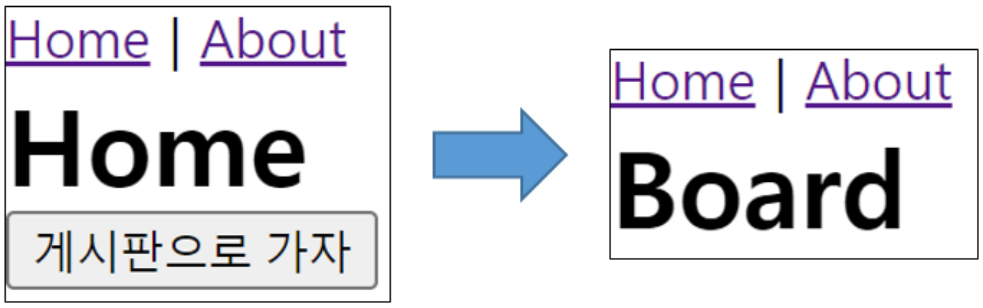
직접 도전을 마쳤다면 다음 설명을 이어가겠다.

웹서비스 이용시 페이지 전환은 "링크"로만 이루어 지는 것이 아니다.

버튼을 눌렀을 때 다른 페이지로 넘어가게 하고 싶을 때 어떻게 하면 좋을까?

`HomeView`로 넘어가서 다음과 같이 작성 해보자.

```
<template>
  <h1>Home</h1>
  <button @click="goToBoardRoute">게시판으로 가자</button>
</template>

<script setup>
import { useRouter } from 'vue-router'

const router = useRouter();

function goToBoardRoute() {
  router.push({ name: "board" });
}
</script>
```



게시판으로 가자 버튼을 통해 페이지 이동을 구현한 것이다.

`router`에서 제공하는 `useRouter` 라는 함수를 이용했다.

`useRouter` 함수는 `router` 객체가 반환되기 때문에 이를 이용해서 페이지 이동을 구현해 본 것이다.

버튼 클릭시 `goToBoardRoute` 함수가 실행되는 동시에 `router.push({ name: "board" })`를 호출한다.

router.push를 통해서 board라는 라우트로 이동한 것이다.

2장 Route - (Dynamic Route Matching)

만약 다음을 구현하고 싶다면 어떻게 하면 좋을까?

`board/1` : 1번글 상세 페이지

`board/2` : 2번글 상세 페이지

우선, 디테일 컴포넌트를 만들자.

views 폴더 안에 `BoardDetailView.vue` 라는 이름의 파일을 생성하고

아래 코드를 넣어두자.

```
<template>
  <h1>BoardDetail</h1>
</template>
```

`router/index.js` 에 다음과 같이 추가로 작성한다.

```
{
  path: '/board/:id',
  name: "detail",
  component: () => import('../views/BoardDetailView.vue')
}
```

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

`router/index.js` 현재 상태는 다음과 같을 것이다.

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

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView,
    },
    {
      path: '/about',
      name: 'about',
      // route level code-splitting
      // this generates a separate chunk (About.[hash].js) for this route
      // which is lazy-loaded when the route is visited.
      component: () => import('../views/AboutView.vue'),
    },

    {
      path: '/board',
      name: 'board',
```

```
      component: () => import('../views/BoardView.vue'),
    },
    {
      path: '/board/:id',
      name: "detail",
      component: () => import('../views/BoardDetailView.vue')
    }
  ],
})

export default router
```

이 상태로 URL 에 `/board/1` 이라고 입력해보자.

Home | About | Board

BoardDetail

이건 내가 원했던 결과가 아니다. 적어도 현재 디테일 페이지가

몇 번 디테일 페이지인지는 나와야 한다.

`BoardDetailView` 를 다음과 같이 수정하자.

```
<template>
  <h1>여기는 {{ $route.params.id }}번 글의 상세 페이지입니다.</h1>
</template>
```

그리고 `/board/1` 그리고 `/board/2` 라고 URL에 직접 작성해서 확인해 보자.

여기는 1번 글의 상세 페이지입니다.

여기는 2번 글의 상세 페이지입니다.

$route.params 를 이용하면 라우트 매개변수를 컴포넌트에서 직접적으로 참조가 가능은 한데 이는 권장하지 않는다. Vue2 (과거의 Vue)에서는 $route.params를 직접 참조했었다. 하지만 라우트가 변경되도 컴포넌트가 자동으로 업데이트가 되지 않기 때문에 Vue3에서는 더이상 권장하는 방식이 아니다. (라우트의 매개변수가 반응형으로 작동이 안되었음)

Vue3에서 권장하는 방식은 `useRoute` 함수를 이용해서 라우트 매개변수를 반응형 변수로 관리하는 방식이다. 예를 들면

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


<script setup>
import { ref } from "vue";
import { useRoute } from "vue-router";

const route = useRoute();
const num = ref(route.params.id)
</script>
```

스크립트 안에서 반응형 변수 (num) 에 할 당 후, 템플릿에 출력하는 것을 말한다.

`useRoute()` 를 사용했을 때에는 라우트의 매개변수를 반응형으로 만들면

라우트가 변경되면서 컴포넌트도 자동으로 업데이트가 된다.

[중요] 여기서 잠시 용어 정리 한번 더 하자.

- `route` 와 `router` 는 다르다.

  `route` 는 `params` 사용할 때 사용한다.

  `router` 는 `push` 를 사용할 때 사용한다.

- `route` : 현재 활성화된 라우트 정보를 담고있는 객체로써

  useRoute() 를 이용한다. (읽기전용)

  - 예> route.params.id (/post/:id 에서 id 값)

- `router`: 라우팅을 조작해야 할때 사용하는 객체이고

  useRouter() 를 이용해서 페이지이동, URL 변경 등 동작수행시 사용한다.

  - 예를들어 `push` , `replace` 등의 메소드를 사용하여 페이지를 이동하거나 URL을 변경하고자 할 때에는 `useRouter` 함수를 사용한다.
  - 예> router.push()

    - 예> router.replace()

따라서 현재 라우트의 정보가 필요로 할 때는 `useRoute` 함수를 사용하고,

프로그래밍적으로 라우팅을 조작해야 할 때, 예를들어 `push`, `replace` 등의 메소드를 사용하여 페이지를 이동하거나 URL을 변경하고자 할 때에는 `useRouter` 함수를 사용한다.

- 지금과 같이 게시판 목록에서 특정 게시글을 클릭하면 접속되는 상세 페이지를 구현할 때, `route.params.id` 를 받아와 서버에 상세 페이지에 들어갈 JSON 데이터를 비동기 요청하는 경우가 앞으로도 많이 있을 것이다.
- 이는 Vue에서 상당히 많이 사용하는 패턴이다.,
    - 게시판 목록 페이지에서 글 클릭 → `/post/3` 같은 경로로 이동
    - 상세 페이지 컴포넌트가 마운트됨
    - `useRoute()`를 통해 `route.params.id` 값 추출
    - `axios` 이용해 해당 ID의 데이터를 서버에 요청
    - 응답 받은 데이터를 화면에 렌더링

[중요] Dynamic Route Matching을 구현시 사용하는 `<RouterLink>` vs `useRoute()`

`<RouterLink>` 그리고 `useRoute()` 의 차이점에 대해서 중간 점검을 하고 넘어가자.

- `<RouterLink>`는 라우트 이동을 위해서 사용하고, `params`를 통해 동적 경로를 설정한다.
    - 예시 `<RouterLink :to="{ name: 'about' }">About</RouterLink>`
    -
- `useRoute()`는 현재 라우트의 파라미터를 읽고 이를 기반으로 동적 데이터를 처리하는데 사용된다.
    - 예시 `const route = useRoute();`

      `const num = ref(route.params.id)`

- 잊지말자.

  라우트 이동을 위한 `<RouterLink>` 그리고 라우트 데이터를 처리할 때 필요한 `useRoute()`

자, 위에서 지금까지 우리가 한 것은

1. BoardDetailView 컴포넌트를 만들고 (상세페이지)
2. 각 상세페이지에 id 라는 파라미터를 사용해서
3. 1번 게시글의 상세페이지 또는 2번 게시글의 상세페이지를 볼수 있도록 셋팅을 했다.

각각의 id에 해당하는 상세페이지를 보기 위해서

1. index.js에서 `'/board/:id'` 경로를 설정해주고

2. BoardDetailView 컴포넌트에서 `useRoute()` 를 이용해서 라우트 파라미터를 읽는것을 해 보았다.

이렇게 실습 후 결과를 확인하기 위해서 URL에

이런식으로 직접 경로를 입력하고 확인을 했다.

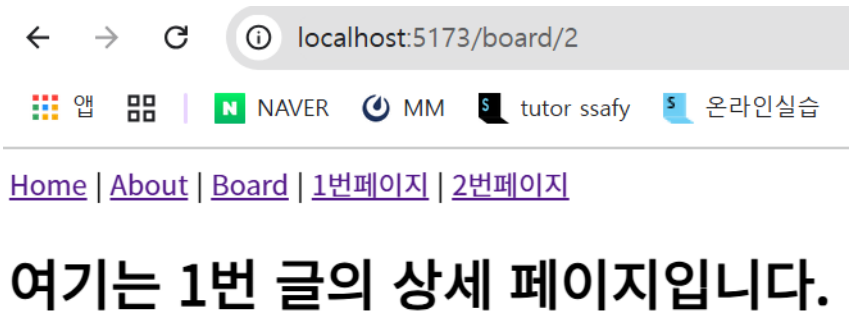
그런데 만약에 링크를 통해서 id가 1인 detail 페이지 그리고
id가 2인 detail 페이지로 이동하도록 링크를 하나씩 달아보자.

다음과 같이 작성하면 된다.
App.vue로 가서 아래의 코드를 추가로 작성해 보자.

```
<RouterLink :to="{ name: 'detail', params: { id: '1' } }">1번페이지</RouterLink> |
<RouterLink :to="{ name: 'detail', params: { id: '2' } }">2번페이지</RouterLink>
```

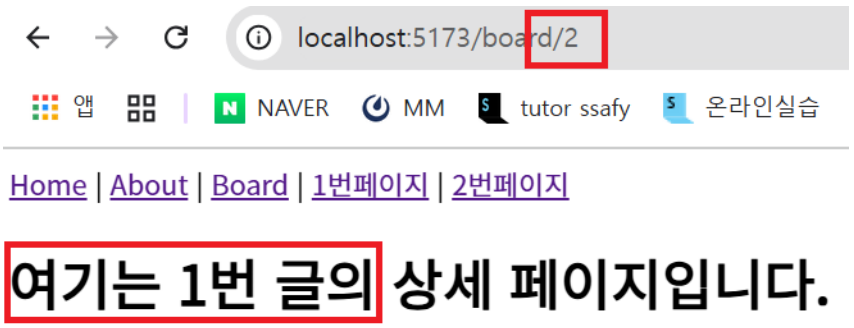
1번페이지 그리고 2번페이지 클릭시 이동이 잘 되는지 확인하자.

처음에 1번페이지를 클릭하면 아래와 같이 잘 이동이 될 것이다.



하지만

2번페이지를 클릭해보자. (1번페이지 → 2번페이지로 링크 클릭을 통해서 이동)



URL은 2번으로 바뀌었지만 여기는 1번 글의 상세 페이지입니다. 라고 여전히 컴포넌트가 업데이트가 되지 않았다.
Vue는 동일한 컴포넌트에서 `<RouterLink>`를 사용할때 파라미터만 바뀔 경우 컴포넌트가 리랜더가 되지 않는다.

그 이유는 같은 컴포넌트 안에서 `<RouterLink>`를 통해서 이동할 때
라우트의 파라미터만 변경될 경우 Vue Router는 Vue의 성능 최적화를 위해서
동일한 컴포넌트를 재사용 하게 된다. 따라서 매개변수인 params는 변경 되지만,
컴포넌트가 리랜더링이 되지 않기 때문에 UI가 업데이트 되지 않는 문제가 발생한 것이다.

즉 조금 더 쉽게 이야기 하자면,

```
const num = ref(route.params.id)
```

여기에서 위의 코드는 딱 한번만 실행된다. (`useRoute()`) 함수를 다시 호출하지 않는다는 것이다.
route는 반응형 객체 이지만
" const num = ref(route.params.id) " 코드는 초기 한 번만 route.params.id 값을 num에 복사했을 뿐,
이후 route가 바뀌어도 num은 자동으로 갱신 되지는 않는다는 것이다.
num이 초기값만 저장하고, 이후 변화에 반응하지 않기 때문에 UI가 업데이트가 되지 않는 것이다.

이 문제를 해결하기 위해서는 watch를 사용해서 감시를 하고 있는 reute.params.id가 변경 될 때마다
변화를 감지하고 num 을 업데이트 하는 방식으로 해결 할 수 있다.

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

<script setup>
```

```
import { ref, watch } from "vue"; // watch 추가
import { useRoute } from "vue-router";

const route = useRoute();
const num = ref(route.params.id);

// watch로 route.params.id가 변경될 때마다 num을 업데이트한다.

watch(() => route.params.id, (newId) => {
 num.value = newId;
}
);
</script>
```

위 코드를 작성하고 서버를 켜서 확인해 보자.

1번페이지 → 2번페이지로 링크 클릭을 통해서 이동시 UI도 잘 바뀌고

2번페이지 → 1번페이지로 링크 클릭을 통해서 이동시에도 UI가 잘 바뀐다.

다른 방법으로는 `beforeRouteUpdate` 라는 네비게이션 가드를 사용하는 방법이 있다.

`beforeRouteUpdate` 을 사용하는 방법은 이 PDF 가장 마지막에 네이게이션 가드 라는 것을 설명하면서 방법을 제시하도록 하겠다.

3장 Nested Routes

위에서 동적 파라미터를 이용해서

http://localhost:5173/board/1 페이지 즉, 여기는 1번 글의 상세페이지 입니다.

http://localhost:5173/board/2 페이지 즉, 여기는 2번 글의 상세페이지 입니다.

http://localhost:5173/board/3 페이지 즉, 여기는 3번 글의 상세페이지 입니다.

들을 구현해 보았다.

이번에는 "중첩라우팅"을 살펴보겠다. 별것 아니다. 기존 경로에 하위경로를 지정하는 것이다.

만약에 각 상세페이지에서

30세 미만인 사람만 보는 컨텐츠와

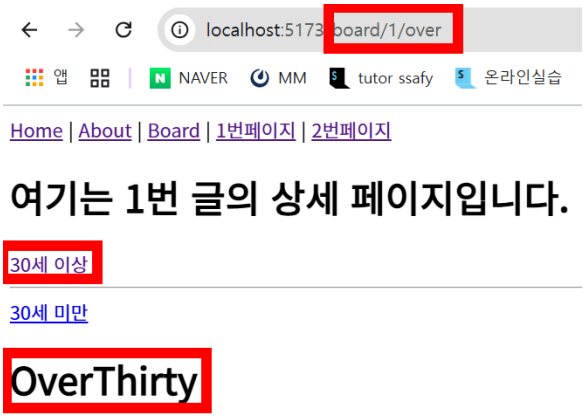
30에 이상인 사람만 보는 컨텐츠 페이지를 각각 나눠 보도록 하자.

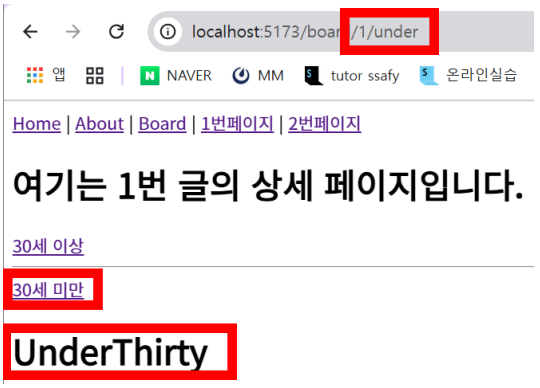
예를들어 1번글의 상세페이지로 접속 후에는 아래 그림과 같이 두 링크가 달릴 것이다.

30세 이상 또는 30세 미만 링크를 클릭하면

게시글 페이지 내에서 각각에 해당하는 하위경로를 지정해 볼 것이다.

http://localhost:5173/board/:id/ 뒤에 이어지는 경로를 설정 할 것이라는 말이다.





즉, 우리가 지금 할 것은 http://localhost:5173/board/:id/ 뒤에 이어지는 경로를 두 부류로 나누고자 한다.

- http://localhost:5173/board/:id/under
- http://localhost:5173/board/:id/over

이렇게 두 종류의 하위 페이지로 각각 나눌 생각이다.

이때 index.js에서 router에 children 옵션만 넣어주면 간단하게 구현이 가능하다.

[중요] 지금 우리는 자식 컴포넌트를 만드는 것이 아니다! (혼동금지)

> Props Emit 때와 같이 컴포넌트들의 부모-자식 관계를 만드는 것이 아니라
>
> 라우트경로가 부모-자식 관계를 갖는것을 말한다.

라우트경로가 부모-자식 관계라고 반드시 컴포넌트도

부모-자식 관계를 가질필요는 없다.

각각 별개의 개념이라는 것을 유의하자.

컴포넌트의 부모-자식 관계와는 별개로
URL경로의 부모-자식 관계를 실습하는 것이다.

라우트의 부모-자식 관계와
컴포넌트의 부모-자식 관계는 완전히 다른 개념이다.

- 컴포넌트의 부조-자식 관계는 **컴포넌트 트리 구조**상 실제로 상,하위에 있는 구조를 말한다.
- 라우트이 부모-자식 관계는 **URL 구조 기준**으로 계층 구조를 만드는 것이다.

실습을 위해 components 폴더 안에
UnderThirty 그리고 OverThirty 컴포넌트를 각각 생성하자.


```
<template>
 <div>
  <h1>OverThirty</h1>
 </div>
</template>
```

```
<template>
 <div>
  <h1>UnderThirty</h1>
```

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

그리고 index.js에서 다음 코드를 추가 / 수정 하자.

import문을 추가하고 children 옵션만 추가해 보자.

```
import OverThirty from '@/components/OverThirty.vue'
import UnderThirty from '@/components/UnderThirty.vue'


  {
    path: '/board/:id',
    name: "detail",
    component: () => import('../views/BoardDetailView.vue'),
    children: [
      { path: 'over', name: 'over-thirty', component: OverThirty },
      { path: 'under', name: 'under-thirty', component: UnderThirty }
    ]
  }
```

children 옵션은 배열 형태로 필요한 만큼 중첩 관계를 표현할 수 있지만 우리는 처음 계획대로 2개만 했다.

[참고]

만약에 children 컴포넌트들도 지연로딩을 시키려면 위에서 작성한 두개의 import문을 지운 후,

아래와 같이 코드를 const router = createRouter( ) 위에다가 추가적으로 작성해도 무방하다.

```
// 자식 컴포넌트들 (지연 로딩 설정)
const OverThirty = () => import('@/components/OverThirty.vue');
const UnderThirty = () => import('@/components/UnderThirty.vue');
```

그리고 views/BoardDetailView.vue 로 넘어가서

두 컴포넌트에 대한 RouterLink 랑 RouterView를 추가하자.

```
<template>

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

    <RouterLink :to="{ name: 'over-thirty' }">30세 이상</RouterLink>
    <hr>
    <RouterLink :to="{ name: 'under-thirty' }">30세 미만</RouterLink>
```

```
    <hr>
    <RouterView />
  </div>
</template>

<script setup>
import { ref, watch } from "vue";
import { useRoute } from "vue-router";

const route = useRoute();
const num = ref(route.params.id);


watch(() => route.params.id, (newId) => {
  num.value = newId;
}
);
</script>
```

원하는 상세페이지를 들어가면 30세 이상 또는 30세 미만이라는 하위경로를 확인 할 수가 있을 것이다.

개념상 주의할 점은 이는, 컴포넌트 간의 부모 자식관계가 아니라,

한 페이지 내에서의 중첩된 라우팅을 의미한다는 것을 기억하자.

(실제 컴포넌트가 부모-자식 관계가 아니라, URL 관계가 부모자식이라는 뜻이다)

4장 Navigation Guard

네이베이션 가드란?

Vue router를 통해 특정 URL에 접근할 때 다른 url로 redirect를 하거나

해당 URL로의 접근을 막는 방법을 말한다.

- 예시> 사용자의 인증 정보가 없으면 특정 페이지에 접근하지 못하게 함

네비게이션 가드 | Vue Router
Vue.js 공식 라우터
https://v3.router.vuejs.org/kr/guide/advanced/navigation-guards.html

네비게이션 가드의 종류에 대해서 먼저 살펴보자.

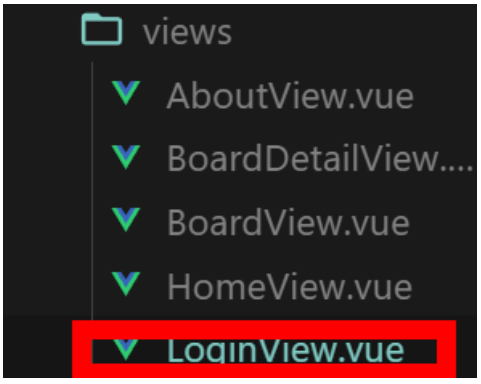
3가지에 대해서 학습할 것이다.

- 전역 가드
  - 애플리케이션 전역에서 동작
  - index.js 에서 정의
- 라우터 가드
  - 특정 URL에서만 동작
  - index.js 의 각 route 에 정의
- 컴포넌트 가드
  - 특정 컴포넌트 내에서만 동작
  - 컴포넌트 안에 Script에 정의

하나씩 실습해보자.

실습을 위해 LoginView.vue 라우터 컴포넌트 생성 후 라우터 등록까지 해보자.

1. views 폴더에 컴포넌트 생성
2. 라우트 추가
   a. (path는 '/login', 그리고 name은 'login' 으로 생성해보자.)




```
{
  path: '/login',
  name: 'login',
  component: () => import('../views/LoginView.vue')
},
```

- LoginView.vue에 컴포넌트 작성 후

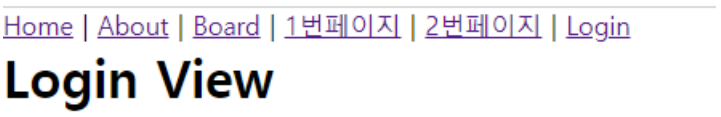
```
<template>
  <div>
    <h1>Login View</h1>
  </div>
</template>
```

- index.js 에 라우터 등록 그리고

```
{
    path: '/login',
    name: 'login',
    component: () => import('../views/LoginView.vue')
 },
```

- App.vue 에 LoginView에 대한 라우터 링크를 작성 한다.

```
<RouterLink :to="{ name: 'login' }">Login</RouterLink>
```



Login View

전역가드 (Global guard)

자 이제 우리가 할 일은 로그인이 되어 있지 않다면

서비스 내 모든 페이지 이동 및 진입을 막고 로그인 페이지로 이동을 시킬 것이다.

다시말해, 로그인 여부에 따른 라우팅 처리를 다르게 해 볼 것이다.

이때 사용하는 전역가드는 beforeEach 라는 함수를 사용하여 구현할 것이다.

beforeEach 함수는 다른 URL로 이동하기 직전에 Vue에서 자동으로 실행해 주는 라우터 훅(Hook)이다.

참고로 Hook이란?

Vue에서 특정 시점에 Vue가 알아서 자동으로 실행 시키는 함수를 Hook이라고 하는데

onmounted, created 와 같이 특정 시점에 자동으로 실행되는 함수들을 컴포넌트 lifecycle 관련된 훅이라고 하며, 페이지 이동이 발생하기 전에 호출되는 라우터 관련 훅은 바로 beforeEach 라우터 훅이다.

beforeEach 라우터 훅의 실행 시점은

라우터 이동 바로직전에 Vue에서 자동으로 호출을 한다.

beforeEach 함수의 기본 문법은 다음과 같다.

```
router.beforeEach((to, from) => {
  console.log(to)
  console.log(from)
  return { name: 'login' }
})
```

- router.beforeEach 함수에는 콜백 함수가 들어간다.
- to: 사용자가 이동하려는 페이지
- from: 이동하기 전 있던 페이지를 의미한다.
- return: 라우터 이동시키기
    - 참고로 next()를 사용하는 방법도 있는데 return이 가독성이 더 좋다.

자, 그럼 실습해보자.

router.beforeEach 훅을

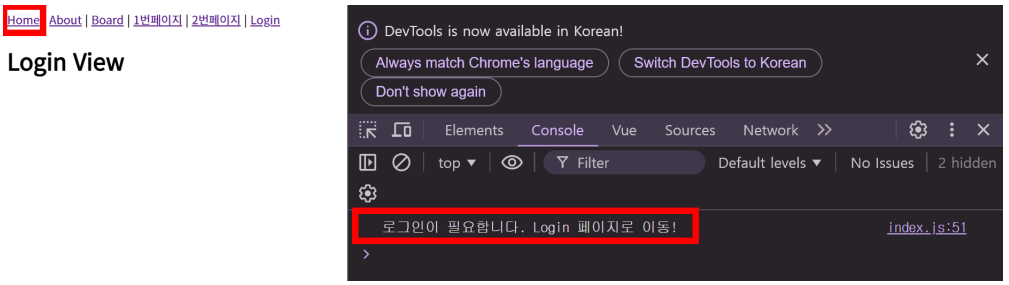
index.js 파일의 `export default router` 구문 위에 작성하자.


```
// router/index.js

router.beforeEach((to, from) => {
 const isAuthenticated = false; // 예시로 인증되지 않은 사용자로 설정했다.

 // 로그인하지 않은 상태에서 로그인 페이지 외의 페이지로 가려고 할 때
 if (!isAuthenticated && to.name !== 'login') {
  console.log('로그인이 필요합니다. Login 페이지로 이동!');
  return { name: 'login' }; // 로그인 페이지로 리디렉션
 }

 // 이미 로그인한 상태에서 로그인 페이지로 접근하려면 다른 페이지로 리디렉션
 if (isAuthenticated && to.name === 'login') {
  console.log('이미 로그인 상태입니다. 홈으로 이동!');
  return { name: 'home' }; // 메인 페이지로 리디렉션
 }
```

```
});

export default router
```

지금 isAuthenticated 값을 false로 셋팅해 놓았다.

지금 사용자는 인증되지 않은 사용자 이므로 Login 페이지가 아닌 곳으로 이동이 불가능 할 것이다.



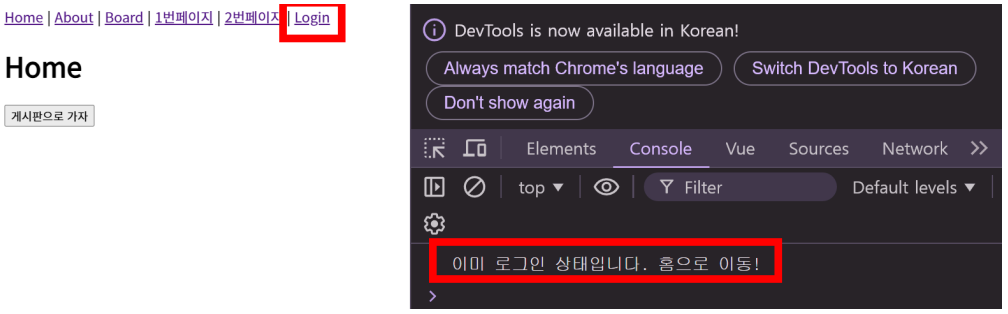
Login 링크가 아닌 다른 링크를 클릭하면 이동이 안될 것이며,
개발자도구의 콘솔창을 확인하면 로그인이 필요하다는 메시지를 확인할 수 있을 것이다.

그러나 Login 링크를 클릭하면 Login View로 이동만 가능 할 것이다.

이번에는 isAuthenticated 값을 true 로 바꿔서 실험해 보자.

```
const isAuthenticated = true;
```

지금 사용자는 인증된 사용자이므로
Login 링크를 클릭하면 이미 로그인이 된 상태라는 메시지를 콘솔창에서 확인이 가능 할 것이며,
링크를 통해 모든 페이지 이동이 가능 할 것이다.



지금은 isAuthenticated 값을 우리가 임이로 true 또는 false 값으로 셋팅을 했지만,
나중에는 Django를 통해서 로그인 여부를 의미하는 값을 응답 받은 후 처리를 하게 될 것이다.

라우트가드 (Route guard)

라우트 가드는 특정 라우트에 진입하기 직전에 사용하는 훅이다.
페이지나 기능이 특정 조건을 충족하는 경우에만 접근을 허용할 때 사용한다.
예를들면 특정 이용약관에 동의한 후에만 접근할 수 있는 페이지가 있는데 이를 컨트롤 할 떄 사용이 가능하겠다.
라우트가드 사용시 사용하는 훅은 beforeEnter() 함수이다.

beforeEnter 함수의 예시를 보자.

```
beforeEnter: (to, from, next) => {

    next('/home');
}
```

- beforeEnter 함수는 라우트 단위로 진입 전 실행되는 가드다.
- to: 사용자가 이동하려는 페이지
- from: 이동하기전 있던 페이지가 된다.
- next( ): 라우터 이동시키기

자, 그럼 실습해보자.
(실습 전에 아까 전역가드 때 셋팅한 isAuthenticated 값이 true인지 다시한번 확인해 보자.)

이번 실습은

about 컴포넌트는 관리자(admin)만 접속 할 수 있도록 코드를 작성해 보겠다.

특정 라우트로의 접근을 제어 할 것이다.

vue-project > src > router > index.js > router > routes > beforeEnter

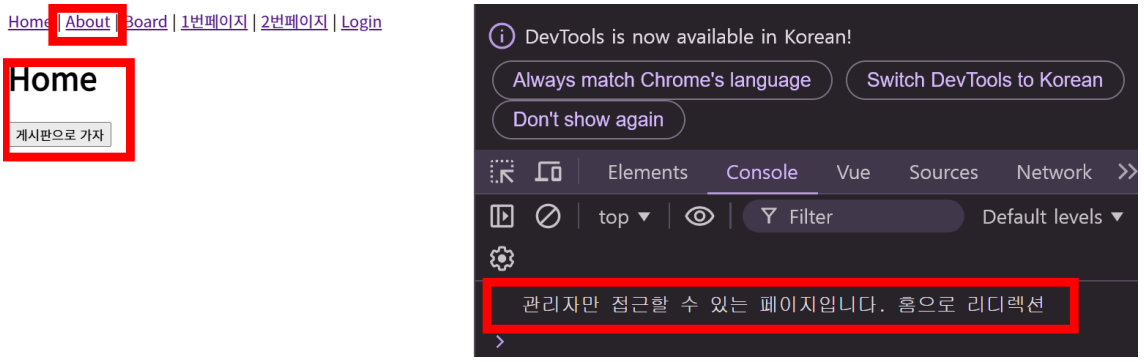
```
import { createRouter, createWebHistory } from 'vue-router'
import HomeView from '../views/HomeView.vue'
const OverThirty = () => import('@/components/OverThirty.vue');
const UnderThirty = () => import('@/components/UnderThirty.vue');

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView,
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('../views/AboutView.vue'),

      beforeEnter: (to, from, next) => {
        const userRole = 'user';
        if (userRole !== 'admin') {
          console.log('관리자만 접근할 수 있는 페이지입니다. 홈으로 리디렉션');
          next('/');
        }
      }
    },

    {
      path: '/board',
      name: 'board',
      component: () => import('../views/BoardView.vue'),
    },
```

```
{
  path: '/about',
  name: 'about',
  component: () => import('../views/AboutView.vue'),

  beforeEnter: (to, from, next) => {
    const userRole = 'user';  // 예시로 일반 사용자
    if (userRole !== 'admin') {
      console.log('관리자만 접근할 수 있는 페이지입니다. 홈으로 리디렉션');
      next('/');  // 홈(메인) 페이지로 리디렉션
    }
  },
```

beforeEnter 함수는 라우트 단위로 진입 전 실행되는 가드이므로 적용하고자 하는 라우트 안에 작성하면 된다.

userRole 값을 'admin'이 아닌 'user'로 셋팅 해 놓았다.

만약에 'admin' 이 아닐 경우에는 '/' 경로로 리디렉션 했다.



About을 클릭하면 Home으로 리디렉션이 되며, 콘솔창 메시지도 확인이 가능 할 것이다.

beforeEnter 훅을 사용하는데 신경 써야 할 주의점이 있다.

1. beforeEnter는 return 을 지원하지 않는다.
   - 위에서 전역가드의 beforeEach 훅은 return 키워드를 지원했다. 하지만 라우트가드의 beforeEnter 훅은 return 키워드 라는 것이 없다.
   - 따라서 next를 통해서 이동을 제어해야 한다. (return 지원 안함)

2. 라우트 가드는 URL 경로가 바뀔 때만 실행되고,
   매개변수만 변경될 경우 라우트 가드는 실행되지 않는다.
   - 단순히 URL의 매개변수가 바뀔 때에는 실행되지 않는다. 는 것은 우리가 사전에 이미 경험을 했다.
     같은 컴포넌트 안에서 `<RouterLink>` 를 통해서 이동할 때
     라우트의 파라미터만 변경될 경우 Vue Router는 Vue의 성능 최적화를 위해서
     동일한 컴포넌트를 재사용 하게 된다. 따라서 매개변수인 params는 변경 되지만,
     컴포넌트가 리랜더링이 되지 않기 때문에 UI가 업데이트 되지 않는 문제가 발생 했던 것과
     같은 이유이다.

컴포넌트가드 (In-component guard)

컴포넌트 안에서 처리할 로직을 설정한다.

예를들면 사용자가 페이지를 떠나기 전에 작성 중인 데이터를 저장하지 않았을 경우 경고 메시지를 띄우고, 이동을 취소시키는 작업을 할 때 `beforeRouteLeave` 함수 사용이 가능 하며,

페이지 안의 일부 데이터만 바뀔때는 `beforeRouteUpdate` 함수를 사용할 수 있다. 하나씩 실습해보자.

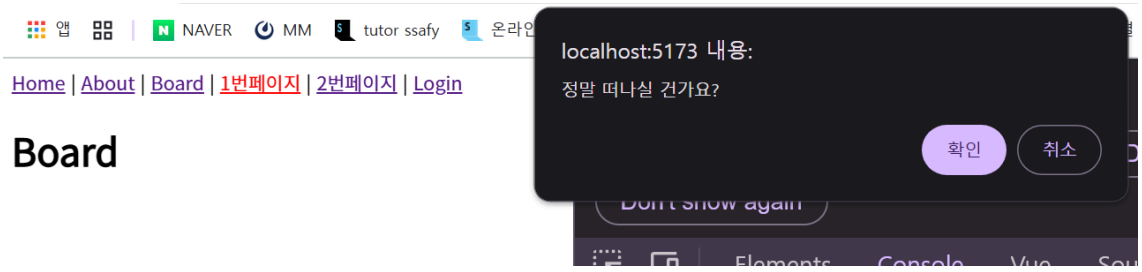
1. onBeforeRouteLeave

Board 페이지를 떠날 때 “정말 떠나실 건가요?” 라는 alert 창을 하나 띄어 보겠다.

컴포넌트 안에서 처리할 로직을 설정하는 것이므로

적용할 컴포넌트 파일에 직접 코드를 작성하자.

```
<!-- # views / BoardView.vue -->

<template>
  <div>
    <h1>Board</h1>
  </div>
</template>

<script setup>
import { onBeforeRouteLeave } from 'vue-router';

onBeforeRouteLeave((to, from, next) => {
  const ask = window.confirm("정말 떠나실 건가요?");
  if (ask) {
    next();  // 사용자가 '확인'을 누르면 이동
  } else {
    next(false);  // 사용자가 '취소'를 누르면 이동하지 않음
  }
});
</script>
```



서버켜서 확인 해보자.

Board 페이지에 한번 들어 갔다가 다른 페이지로 이동하기 위한 링크를 열어보자.

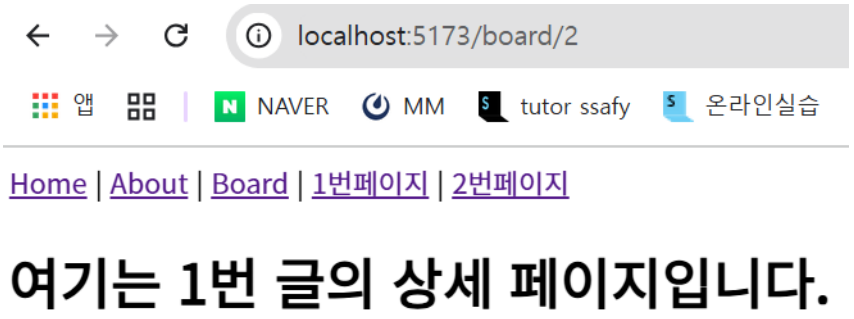
2. beforeRouteUpdate

이번에는 beforeRouteUpdate 를 사용해서 라우트의 파라미터 변경으로 인한 컴포넌트를 새로 갱신해 보겠다.

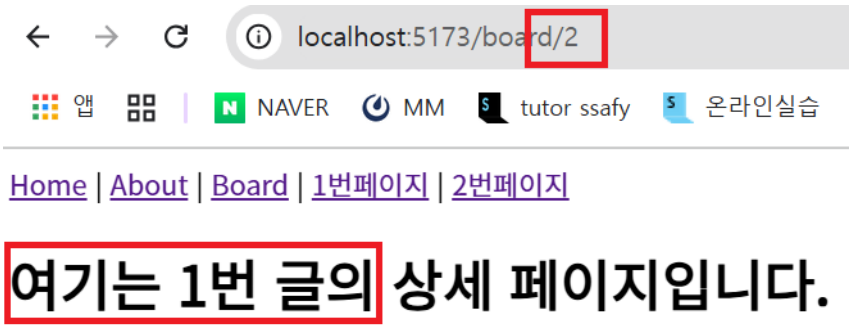
우리는 앞선 실습에서 `<RouterLink>` 를 통해서 이동할 때
라우트의 파라미터만 변경될 경우 UI가 업데이트 되지 않는 문제가 발생했었다.

[앞에서 나왔던 문제발생 지점]

처음에 1번페이지를 클릭하면 아래와 같이 잘 이동이 되었다.



하지만 1번페이지 → 2번페이지로 링크 클릭했을때



URL은 2번으로 바뀌었지만 여기는 1번 글의 상세 페이지입니다. 라고 여전히 컴포넌트가 업데이트가 되지 않았다.
그래서 우리는 앞에서 watch를 사용해서 parameter의 변화를 감지하고 반응형으로 상태를 추적하고 컴포넌트를 갱신했었다.

이번에는 watch를 사용하지 않고 `beforeRouteUpdate` 를 사용해서 문제를 해결해 보겠다.
마찬가지로 컴포넌트 안에서 처리할 로직을 설정하는 것이므로 적용할 컴포넌트 파일에 직접 코드를 작성하자.

```vue
<!-- # views / BoardDetailView.vue -->

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

    <RouterLink :to="{ name: 'over-thirty' }">30세 이상</RouterLink>
    <hr>
    <RouterLink :to="{ name: 'under-thirty' }">30세 미만</RouterLink>

    <RouterView />
  </div>
</template>

<script setup>
import { ref } from "vue";
import { useRoute, onBeforeRouteUpdate } from "vue-router";

const route = useRoute();
const num = ref(route.params.id);

onBeforeRouteUpdate((to, from, next) => {
  num.value = to.params.id;
  next();
});
</script>
```

`beforeRouteUpdate`는 컴포넌트 내부에서 라우트 변경이 발생할 때
컴포넌트 내부의 데이터를 업데이트할 수 있는 함수이다.

위 코드를 작성하고 서버를 켜서 확인해 보자.
1번페이지 → 2번페이지로 링크 클릭을 통해서 이동시 UI도 잘 바뀌고
2번페이지 → 1번페이지로 링크 클릭을 통해서 이동시에도 UI가 잘 바뀐다.

그렇다면 우리가 앞서서 이용했던 watch와의 차이점은 무엇일까?

차이점

특징	`watch`	`beforeRouteUpdate`
사용 목적	상태나 데이터의 변화를 감지하고 그에 반응하는 방식	라우트가 변경될 때, 그 전후를 감지하고 처리하는 방식
호출 시점	데이터가 변경될 때마다 호출	라우트가 변경되기 직전에 호출
사용 가능한 객체	감지하고자 하는 변수(예: `route.params.id`)	라우트 객체 (`to`, `from`)

비교적 사용되는 상황	라우트 외의 상태 변화도 감지하고 싶을 때	라우트 파라미터 변화에 맞춰서 전처리하고 싶을 때
반응성	반응형으로 상태를 추적하며 갱신	라우트 변경 시 컴포넌트를 새로 갱신하고 싶을 때 사용

watch 그리고 beforeRouteUpdate 모두 Vue 컴포넌트에서 라우트 변화에 반응하여 데이터를 갱신하는 방법이지만, 사용하는 상황과 목적에 따라 다르게 사용된다. 따라서 상황과 목적만 부합한다면 아무거나 사용해도 괜찮다.

[요약]

Vue Router에서 가드(guard)란 라우터의 진입, 이탈, 이동을 제어하는 수단을 의미한다.

이는 언제, 어디서 라우터를 제어하느냐에 따라

전역 가드 / 라우터 가드 / 컴포넌트 가드 로 나눌 수 있었다.

- 전역 가드 (beforeEach() 함수사용)
  - 애플리케이션 전역에서 동작
  - index.js 에서 정의
  - 모든 라우터 이동시 사용했다. (예> 로그인 여부 체크)

- 라우터 가드 (beforeEnter() 함수사용)
  - 특정 URL에서만 동작
  - index.js 의 각 route 에 정의
  - 특정 라우터에 들어가기 전에 (타이밍)

    특정 라우트 접근을 제한 또는 허용할때 사용했다. (예> 권한에 따는 접근 설정)

- 컴포넌트 가드 (onBeforeRouteLeave () 그리고 onBeforeRouteUpdate() 함수 사용)
  - 특정 컴포넌트 내에서만 동작
  - 컴포넌트 안에 Script에 정의
  - 컴포넌트 단위로 사용하는데
    - onBeforeRouteLeave() 를 통해서 alrt 창을 띄어 보기도 했고
    - onBeforeRouteUpdate() 를 이용해서 우리는 컴포넌트 내부 비동기 처리를 해 보았다.

test.zip

<끝>