SKALA

대중소 상생

Vue.js 어플리케이션 개발 실무

SK Inc. AX

1. 라우팅과 HTTP 통신
2. Advanced 기능 및 최적화

1. 라우팅과 HTTP 통신

- Vue Router 기초
- Vue Router 심화
- HTTP 클라이언트 (Axios)
- 상태 관리 라이브러리 (Pinia)
- 종합실습

SK Inc. AX

1.1 Vue Router 기초

Vue Router 4를 활용해 SPA 라우팅 개념부터 설치, 기본요소, 동적 라우팅까지 이해하고 적용한다.

학습목표

- 라우팅(Routing) 개요
- SPA와 Router
- Vue Router 4.X 설치
- Router 기본 요소
  - Route, Router
  - RouterLink / RouterView
- 동적 라우팅
- useRoute()와 useRouter()

1.1 Vue Router 기초

라우팅(Routing)

- 개념
  - URL 경로(path)를 화면(component)과 연결시키는 동작
- 특징
  - 페이지 전체를 새로고침 하지 않고 일부만 교체 (SPA 환경)
  - 북마크/공유 가능한 URL 제공
- 구성요소
  - 라우트(Route)
    단일 규칙 (경로와 컴포넌트 매핑 정보)
  - 라우터(Router 인스턴스)
    여러 Route를 모아 앱 전체 내비게이션 관리

1.1 Vue Router 기초

SPA와 Router

- 개념
  - SPA :
    한번 로드 후 페이지 전환을 브라우저가 아닌 클라이언트가 담당
    사용자가 네비게이션 할 때마다 전체 페이지가 아닌 필요한 컴포넌트만 교체 방식
  - Router:
    브라우저의 URL 변화를 감지해서 해당 경로에 맞는 컴포넌트 랜더링
    SPA의 컴포넌트 교체 작업 역할
- 특징
  - URL과 컴포넌트 매칭
  - 페이지 reload 없음
  - 히스토리 관리
  - 부드러운 사용자 경험 (UX)

1.1 Vue Router 기초

Vue Router 4.X 개요

- 개념
  - SPA 라우팅 엔진:
    URL 변환에 맞춰 컴포넌트 교체 랜더링 담당
  - Vue 코어와 공식 통합:
    Composition API 와 긴밀하게 연동

  https://router.vuejs.org/

- 특징
  - 다양한 히스토리 모드 ex. createWebHistory()
  - 중첩 및 네임드 라우트
  - 코드 스플리팅
  - 내비게이션 가드(guard)
  - 스크롤 동작 제어

1.1 Vue Router 기초

Vue Router 4.X 설치

- 설치 가이드

  https://router.vuejs.org/installation.html#Package-managers

- 설치 방법

  A. Vue 프로젝트 생성

  npm create vite@latest my-router-app

  B. Vue Router 4 설치

  cd my-router-app

  npm install vue-router@4

- 의존성 확인

  package.json : **vue-router: ^4.5.1**

1.1 Vue Router 기초

프로젝트 구조 (기본)

```
src
  main.js              # 라우터 등록
  router/
     index.js                # 라우트 및 라우터 생성
  views/               # 페이지 컴포넌트
     Home.vue
     About.vue
  App.vue              # 공통 레이아웃 (메뉴 + 화면 자리)
```

1.1 Vue Router 기초

라우팅 실습 순서

1. 프로젝트 만들기
2. Vue Router 4 설치
3. 프로젝트 구조 만들기
4. 라우트 및 라우터 생성

   src/router/index.js
5. 라우터 등록 및 공통 레이아웃

   src/main.js, src/App.vue 수정
6. 페이지 컴포넌트 작성

   src/views/Home.vue, src/views/About.vue
7. 실행

1.1 Vue Router 기초

코드 작업1 – 라우트 및 라우터 생성

- src/router/index.js 작성
- https://router.vuejs.org/guide/#Creating-the-router-instance 참조

```
// index.js
import { createRouter, createWebHistory } from "vue-router";
import Home from "../views/Home.vue";
import About from "../views/About.vue";

// 라우트 설정
const routes = [
  { path: "/", name: "home", component: Home },
  { path: "/about", name: "about", component: About },
];
// 라우터 생성
const router = createRouter({
  history: createWebHistory(),
  routes,
});
export default router;
```

1.1 Vue Router 기초

코드 작업2 – 라우터 등록 및 공통 레이아웃

- src/main.js 및 src/App.vue 수정
- https://router.vuejs.org/guide/#Registering-the-router-plugin 참조

```
// main.js
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";

// 라우터 등록
createApp(App).use(router).mount("#app");
```

```
<!-- App.vue -->
<template>
  <nav>
    <!-- 링크: 최소한의 네비게이션 -->
    <RouterLink to="/">Home</RouterLink> |
    <RouterLink to="/about">About</RouterLink>
  </nav>

  <!- 해당 컴포넌트가 보여질 위치.
        라우트가 바뀌면 여기 내용이 교체됨 -->
  <RouterView />
</template>

<script setup>
// Composition API 사용 (별도 로직 없음)
</script>
```

1.1 Vue Router 기초

코드 작업3 – 페이지 컴포넌트 생성

- src/views/Home.vue
- src/views/About.vue

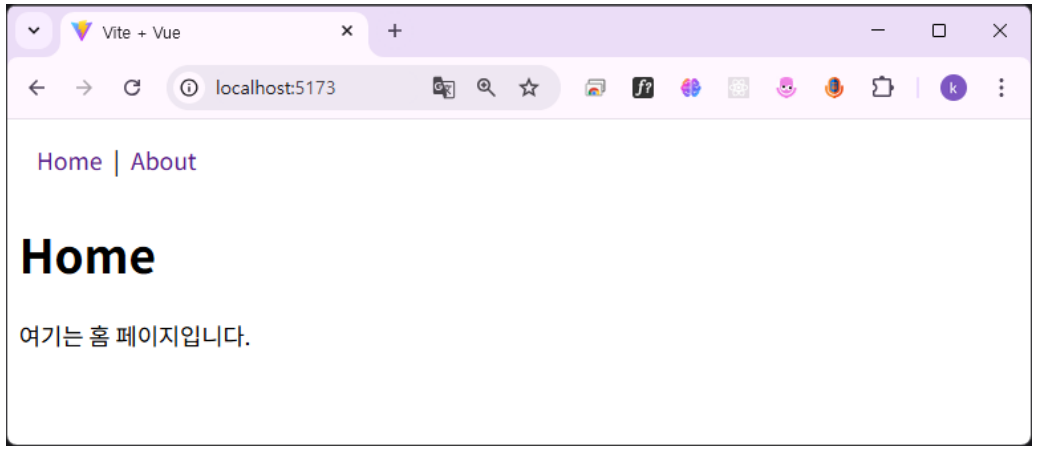
```
<!-- Home.vue -->
<template>
  <h1>Home</h1>
  <p>여기는 홈 페이지입니다.</p>
</template>
<script setup></script>
```

```
<!-- About.vue -->
<template>
  <h1>About</h1>
  <p>여기는 소개 페이지입니다.</p>
</template>
<script setup></script>
```

1.1 Vue Router 기초

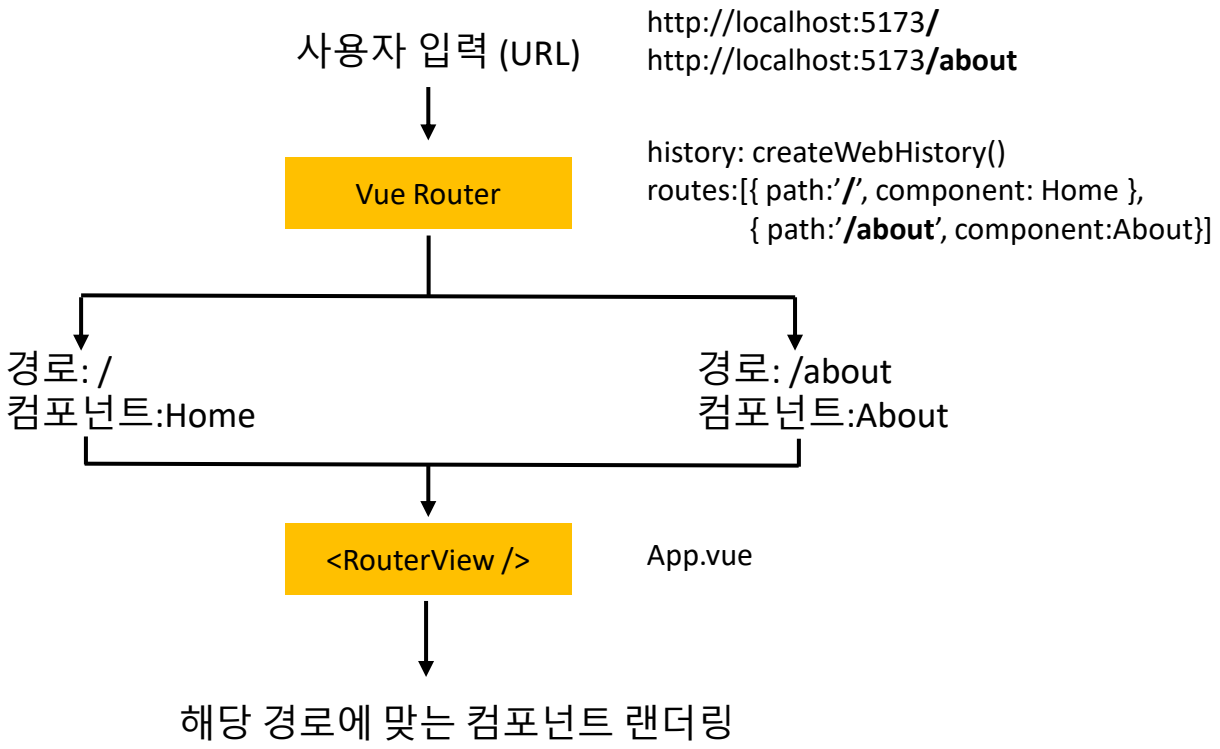
코드 실행

- npm run dev



1.1 Vue Router 기초

실습 동작 흐름



1.1 Vue Router 기초

Vue Router 기본 구성 요소

- Route
- Not Found 처리 (Catch-all)
- Router
- RouterView
- RouterLink
- useRoute()
- useRouter()

1.1 Vue Router 기초

Route

- 개념
  - URL(path)과 컴포넌트(component) 매핑 규칙 목록
  - 하나의 route = path + component (+ 옵션속성)
- 기본 문법
  - path: URL 경로
  - component: 보여줄 Vue 컴포넌트

```
const routes = [
 {
  path: '/경로 ' ,
  component: 컴포넌트명
 }
]
```

1.1 Vue Router 기초

Route 옵션 속성

- 개념
  - path 와 component는 기본 속성
  - 기본 속성 이외의 추가 속성
- 종류
  - name
  - props
  - alias
  - children
  - redirect
  - meta
  - components

1.1 Vue Router 기초

name 속성

- 개념
  - 라우트 이름 지정
  - 코드에서 경로대신 이름으로 접근 가능
  - path 변경 유지보수나 프로그래매틱 내비게이션 시 유용

- 샘플 코드

```
const routes = [
  { path: '/about',
    name: 'aboutPage',
    component: About
  }]

router.push("/about")
router.push({name:'aboutPage'})
```

1.1 Vue Router 기초

props 속성

- 개념
  - 동적 파라미터를 컴포넌트의 props로 전달
  - 컴포넌트에서 route.params 대신 props로 바로 사용

- 샘플 코드

```
const routes = [
  { path: '/user/:id',
    component: User,
    props:true
  }]

defineProps( {id:String})
```

1.1 Vue Router 기초

alias 속성

- 개념
  - 하나의 화면을 여러 경로로 접근 가능
  - 주소는 다르지만 화면은 동일

- 샘플 코드

```
const routes = [
  { path: '/home',
    component: Home,
    alias:'/start'
  }]
```

/home 이나 /start 모두 Home 랜더링

1.1 Vue Router 기초

children 속성

- 개념
  - 부모 컴포넌트안에 자식 컴포넌트 구조
  - 중첩 라우트

- 샘플 코드

```
const routes = [
  { path: '/teams',
    component: TeamList,
    children:[
      {path:':teamId' , component:TeamMembes}
    ]
  }]
```

1.1 Vue Router 기초

redirect 속성

- 개념
  - 특정 경로로 접근했을 때 다른 경로로 자동 이동

- 샘플 코드

```
const routes = [
{ path:'/',  redirect:'/teams'},
{ path: '/teams',  component: TeamList },

]

/ 요청시 자동으로 /teams 이동
```

1.1 Vue Router 기초

meta 속성

- 개념
  - 라우트 별 추가 정보 설정
    ex. 인증 필요 여부, 화면 제목, 권한 레벨 등
  - 참조: route.meta

- 샘플 코드

```
const routes = [
{ path: '/admin',
  component: Admin,
  meta: { requiresAuth: true,  role:'admin' }
 }
]

route.meta.requiresAuth
```

1.1 Vue Router 기초

components 속성

- 개념
  - component : 하나의 컴포넌트 설정
  - components: 여러 컴포넌트 설정
    하나의 화면에서 여러 컴포넌트를 동시에 연결할 때 주로 사용

- 샘플 코드

```
const routes = [
{ path: '/teams',
  components: { default: TeamList,  footer: TeamFooter }
]
<template>
   <RouterView />
   <RouterView name="footer" />
</template>
```

1.1 Vue Router 기초

Route 속성 요약

속 성	설 명	샘 플
path	URL 주소 경로	{ **path**: '/about', component: About }
component	보여줄 화면 컴포넌트	{ path: '/about', **component**: About }
name	라우트 이름	{ path: '/about', **name**:'aboutPage', component: About }
props	URL 파라미터를 props로 전달	{ path: '/user/:id', component:User, **props**:true }
alias	별칭 주소	{ path: '/home', component:Home, **alias**:'/start' }
children	중첩 라우트 (부모/자식)	{ path: '/teams', component:TeamList, **children:[{...}]** }
redirect	다른 주소로 재요청	{ path: '/go-about', **redirect**:'/about' }
meta	라우트 별 추가 정보	{ path: '/admin', component:Admin, **meta**:{ role:'admin'} }
components	여러 컴포넌트들 설정 Named Router view 구현	{ path: '/users', **components**:{default:UsersList, footer:UsersFooter} }

1.1 Vue Router 기초

구현 코드

```
// index.js
import { createRouter, createWebHistory } from "vue-router";

import Home from "../views/Home.vue";
import About from "../views/About.vue";
import Admin from "../views/Admin.vue";

const routes = [
  // 1) alias: / 또는 /start 로 접근 가능
  { path: "/", alias: "/start", component: Home },

  // 2) redirect: /go-about 으로 접근하면 /about 으로 자동 이
동
  { path: "/go-about", redirect: "/about" },
  { path: "/about", component: About },

  // 3) meta: 인증이 필요한 페이지 (관리자 페이지)
  { path: "/admin",
    component: Admin,
    meta: { requiresAuth: true , role:'admin' },
  },
];
```

```
<!-- App.vue -->
<template>
  <nav>
    <!-- 링크: 최소한의 네비게이션 -->
    <RouterLink to="/">Home</RouterLink> |
    <RouterLink to="/start">Home(alias:/start)</RouterLink>
|
    <RouterLink to="/go-about">Go
About(redirect)</RouterLink> |
    <RouterLink to="/about">About</RouterLink> |
    <RouterLink to="/admin">Admin(meta)</RouterLink>
  </nav>

  <!-- 라우트가 바뀌면 여기 내용이 교체됨 -->
  <RouterView />
</template>

<script setup>
// Composition API 사용 (별도 로직 없음)
</script>
```

1.1 Vue Router 기초

구현 코드

```
<!-- Admin.vue -->
<template>
  <h1>Admin</h1>
  <p>여기는 관리자 페이지입니다.</p>
</template>
<script setup>
import { useRoute } from "vue-router";

const route = useRoute();

//meta 참조
console.log("meta 참조: ", route.meta.requiresAuth, route.meta.role);
</script>
```

1.1 Vue Router 기초

Not Found 처리

- 개념
  - 사용자가 잘못된 경로(URL)로 접근했을 때 처리 방법
- 특징

  /:pathMatch(.*)* 패턴 지정 (Catch-all 라우트)

  NotFound.vue 전용 컴포넌트
- 문법

  {path: **'/:pathMatch(.*)*'**, name: 'NotFound', component: NotFound }
- 공식 사이트

  https://router.vuejs.org/guide/essentials/dynamic-matching.html#Catch-all-404-Not-found-Route

1.1 Vue Router 기초

구현 코드

```
// index.js
import { createRouter, createWebHistory } from "vue-
router";

import Home from "../views/Home.vue";
import About from "../views/About.vue";
import NotFound from "../views/NotFound.vue";

const routes = [
  { path: "/", component: Home },
  { path: "/about", component: About },
 // Catch-all route
  { path: "/:pathMatch(.*)*", name: "NotFound",
component: NotFound },
];
```

```
<!-- App.vue -->
<template>
  <nav>
    <!-- 링크: 최소한의 네비게이션 -->
    <RouterLink to="/">Home</RouterLink> |
    <RouterLink to="/about">About</RouterLink> |
    <RouterLink to="/wrong-url">Wrong URL</RouterLink>
  </nav>
  <!-- 라우트가 바뀌면 여기 내용이 교체됨 -->
  <RouterView />
</template>
<script setup>
// Composition API 사용 (별도 로직 없음)
</script>
```

```
<!-- NotFound.vue -->
<template>
  <h1>404 - 페이지를 찾을 수 없습니다</h1>
  <p>입력한 주소가 올바르지 않습니다.</p>
  <RouterLink to="/">홈으로 돌아가기</RouterLink>
</template>
<script setup></script>
```

1.1 Vue Router 기초

Router

- 개념
  - 여러 Route를 모아 앱 전체 어플리케이션 내비게이션 처리
  - createRouter() 이용
  - 생성 후 앱에 등록 필수

- 문법

```
// 생성
Const router = createRouter({
  history: createWebHistory(),
  routes,
});

// 등록
createApp(App).use(router).mount("#app");
```

1.1 Vue Router 기초

RouterView

- 개념
  - 현재 URL에 맞는 컴포넌트 랜더링 위치
  - 여러 개의 <RouterView /> 사용 가능 (중첩 라우트)
  - Named Router Views 활용
- 문법

```
<template>
  <RouterView />
</template>
```

```
<template>
  <RouterView />
  <RouterView name="footer"/>
</template>
```

1.1 Vue Router 기초

구현 코드

```
// index.js
import { createRouter, createWebHistory } from "vue-router";
import Home from "../views/Home.vue";
import About from "../views/About.vue";
import Footer from "../views/Footer.vue";

const routes = [
  { path: "/",
    component: Home
  },
  { path: "/about",
    components: {
      default: About,
      footer: Footer,
    },
  },
];
```

```
<!-- App.vue -->
<template>
  <nav>
    <RouterLink to="/">Home</RouterLink> |
    <RouterLink to="/about">About</RouterLink>
  </nav>

  <!-- 기본 RouterView -->
  <main>
    <RouterView />
  </main>

  <!-- 네임드 RouterView -->
  <footer>
    <RouterView name="footer" />
  </footer>
</template>

<script setup>
</script>
```

1.1 Vue Router 기초

동적 라우팅

- 개념
  - URL 주소 일부가 변수(동적 파라미터) 처럼 동작
  - 문법:

    요청 : ex. /user/**a01** , /user/**a02**

    path : ex. /user**/:id**

    사용 : ex. const route = useRoute()

    route.params.id
- 특징
  - 여러 URL을 하나의 라우트 규칙으로 처리
  - useRoute() 이용한 참조
  - props:true 설정시 파라미터를 props로 처리 가능
  - 쿼리(query) 같이 사용 ex. /user/a01**?tab=profile , route.query.tab**

1.1 Vue Router 기초

구현 코드

```
// index.js
import { createRouter, createWebHistory } from "vue-router";

import Home from "../views/Home.vue";
import About from "../views/About.vue";
import User from "../views/User.vue";

const routes = [
  { path: "/", name: "home", component: Home },
  { path: "/about", name: "about", component: About },
  // id 파라미터
  { path: "/user/:id", name: "user", component: User, props:
true },
];
```

```
<!-- App.vue -->
<template>
  <nav>
    <!-- 링크: 최소한의 네비게이션 -->
    <!-- 1) 문자열 to -->
    <RouterLink to="/">Home</RouterLink>
    <RouterLink to="/about">About</RouterLink>
    <RouterLink to="/user/a01">User a01</RouterLink>
    <RouterLink to="/user/a02?tab=profile">User
a02?tab=profile</RouterLink>
  </nav>

  <!-- 기본 RouterView -->
  <main>
    <RouterView />
  </main>
</template>

<script setup></script>
```

1.1 Vue Router 기초

구현 코드

```
<!-- User.vue -->
<script setup>
import { defineProps } from "vue";
import { useRoute } from "vue-router";

const route = useRoute();

// props:true 덕분에 defineProps로도 받을 수 있음.
defineProps({ id: String });

</script>

<template>
  <section>
    <h1>User Page</h1>
    <p><strong>defineProps</strong>: {{ id ?? "(없음)" }}</p>
    <p><strong>params.id</strong>: {{ route.params.id ?? "(없음)" }}</p>
    <p><strong>query</strong>: {{ route.query }}</p>
  </section>
</template>
```

1.1 Vue Router 기초

RouterLink

- 개념
  - 페이지 이동 링크
  - 페이지 새로고침 없이 화면 전환
  - 라우터와 완전 연동 (활성 상태, 파라미터/쿼리, 이름)
  - 문법:

    <RouterLink to=타겟 />

- 특징
  - to 속성값

    문자열 ex. to="/about"

    객체 ex. to="{ name:'user', params:{id: '12'}, query:{id:'12'} }"
  - 활성 상태

    기본 제공 CSS 클래스 ex. router-link-active

1.1 Vue Router 기초

to 속성값 비교

구분	문자열	객 체
샘플 코드	<RouterLink **to**="/about" /> <RouterLink **:to**="target" />	<RouterLink **to**="{name:'user', params:{id:'12'}, query:{age:'10'}"> <RouterLink **:to**="{name:'user', params:{id:**userId**},query:{**age**})
난이도	가장 간단, HTML 형식이라 직관적	길지만 구조화되어 의미가 명확
params	권장안함. 직접 추가 Ex. "/user/"+id	권장. name+params 안전하게 바인딩 Ex. params:{id:'12'}
query	권장안함. 직접 추가 Ex. "user/34?age=10	권장. 안전하게 처리 Ex. query:{age:10}
유지보수	경로 바뀌면 모든 문자열 수정	이름 기반이기 때문에 경로 변경에도 코드 영향 최소화
타입안정성	문자열이라 오타/키 감지 어려움	객체 키(name, params, query)로 오타 감지 용이
URL인코딩	직접 처리 필요	자동 인코딩
사용 예	정적 페이지 이동	동적 라우팅

1.1 Vue Router 기초

구현 코드

```
// index.js
import { createRouter, createWebHistory } from "vue-router";

import Home from "../views/Home.vue";
import About from "../views/About.vue";
import User from "../views/User.vue";

const routes = [
  { path: "/", name: "home", component: Home },
  { path: "/about", name: "about", component: About },
  // id 파라미터
  { path: "/user/:id", name: "user", component: User, props:
true },
];


<!-- App.vue -->
<template>
  <nav>
    <!-- 링크: 최소한의 네비게이션 -->
    <!-- 1) 문자열 to -->
    <RouterLink to="/">Home</RouterLink>
    <RouterLink to="/about">About</RouterLink>
```

1.1 Vue Router 기초

```
 <!-- 2) 객체 to (이름 + params + query) -->
    <RouterLink   :to="{  name: 'user’, params: { id:
'12’ }, query: { tab: 'info', ref: 'nav' },}“>
        User( 객체 )
    </RouterLink>

    <!-- 3) 객체 to 바인딩 (정적 값) -->
    <RouterLink :to="{ name: 'user', params: { id: '12' },
query: { tab: 'info' } }“>
       User- 정적바인딩
    </RouterLink>

    <!-- 4) 객체 :to 바인딩 (동적 값 활용) -->
    <RouterLink :to="{ name: 'user', params: { id: userId },
query: { tab } }">
      User- 동적바인딩
    </RouterLink>
  </nav>

  <!-- 기본 RouterView -->
  <main>
    <RouterView />
  </main>
</template>
<script setup>
import { ref } from "vue";

const userId = ref(77);
const tab = ref("profile");
</script>
```

1.1 Vue Router 기초

구현 코드

```
<!-- User.vue -->
<script setup>
import { defineProps } from "vue";
import { useRoute } from "vue-router";

const route = useRoute();

// props:true 덕분에 defineProps로도 받을 수 있음.
defineProps({ id: String });

</script>

<template>
  <section>
    <h1>User Page</h1>
    <p><strong>defineProps</strong>: {{ id ?? "(없음)" }}</p>
    <p><strong>params.id</strong>: {{ route.params.id ?? "(없음)" }}</p>
    <p><strong>query</strong>: {{route.params.id }}</p>
  </section>
</template>
```

1.1 Vue Router 기초

useRoute()와 useRouter()

- useRoute()
  - 현재 Route 객체를 참조하는 hook
  - 컴포넌트 안에서 현재 URL 정보 접근 용도

    ex. route.params , route.query, route.path, route.name, route.meta
  - 읽기 전용

- useRouter()
  - Router 인스턴스를 참조하는 hook
  - 컴포넌트 안에서 프로그래매틱 내비게이션 가능

    ex. Router.push('/about')

    router.push("{name:'user', params:{id:'200'}")
  - 쓰기 가능

1.1 Vue Router 기초

구현 코드

```
// index.js
import { createRouter, createWebHistory } from "vue-router";

import Home from "../views/Home.vue";
import User from "../views/User.vue";

const routes = [
  { path: "/", name: "home", component: Home },
  { path: "/user/:id", name: "user", component: User, props: true },
];
```

1.1 Vue Router 기초

구현 코드

```
<!-- App.vue -->
<template>
  <nav>
    <!-- 링크: 최소한의 네비게이션 -->
    <RouterLink to="/">Home</RouterLink>
    <RouterLink
      :to="{ name: 'user', params: { id: '101' }, query: { keyword:
'vue' } }“>
      User(101)
    </RouterLink>
  </nav>

  <!-- 기본 RouterView -->
  <main>
    <RouterView />
  </main>
</template>

<script setup></script>
```

1.1 Vue Router 기초

구현 코드

```vue
<!-- User.vue -->
<script setup>
import { defineProps } from "vue";
import { useRoute, useRouter } from "vue-router";

const route = useRoute();
defineProps({ id: String });

const router = useRouter();
function goHome() {
  router.push("/");
}
</script>
<template>
  <section>
    <h1>User Page</h1>
    <p><strong>defineProps</strong>: {{ id ?? "(없
음)" }}</p>
    <p><strong>params.id</strong>: {{ route.params.id ??
"(없음)" }}</p>
    <p><strong>query</strong>: {{ route.query }}</p>
    <button @click="goHome">Home 페이지로 이동</button>
  </section>
</template>
```

```vue
<!-- Home.vue -->
<template>
  <h1>Home</h1>
  <p>여기는 홈 페이지입니다.</p>
  <button @click="goUserByCode">User 페이지로 이동</button>
</template>
<script setup>
import { useRouter } from "vue-router";

const router = useRouter();
const goUserByCode = () => {
  router.push({
    name: "user",
    params: { id: "200" },
    query: { keyword: "router" },
  });
};
</script>
```

1.2 Vue Router 심화

중첩 라우트, 지연 로딩, 내비게이션 가드를 활용해 Vue Router의 효율적이고 확장 가능한 기법을 이해하고 적용한다.

학습목표

- 중첩 라우트 (Nested Routes)
- 지연 로딩 (Lazy Loading)
- 코드 스플리팅 (Code Splitting)
- 내비게이션 가드 (Navigation Guards)
  - 전역 가드 (Global Guards)
  - 개별 라우트 가드 (Per-Route Guards)
  - 컴포넌트 내부 가드 (In-Component Guards)

1.2 Vue Router 심화

중첩 라우트

- 개념
  - 하나의 화면을 부모(레이아웃)와 자식(컨텐츠 영역) 구조로 분리
  - 부모는 공통 UI(헤더, 사이드바,푸터)를 가짐
  - 자식은 <RouterView /> 자리에서 교체되며 표시
  - 즉 부모는 틀(Layout) 역할이고 자식은 내용(Content) 역할

- 특징
  - 레이아웃 재사용
  - 로드 최적화 (지연 로딩)
  - 명확한 UI 설계
  - Named router View 적용 가능

1.2 Vue Router 심화

프로젝트 구조 (기본)

```
src
  main.js                                  # 라우터 등록
  router/
      index.js                          # 라우트 및 라우터 생성
  layouts/
      UsersLayout.vue                       # 부모 ( 레이아웃 )  - 공통영역 + <RouterView />
  views/
      Home.vue                             # 홈 화면
      users/
              UsersHome.vue                    # 기본 자식,  /users
              UserDetail.vue                 #  자식 1,      /users/:id
              UserSetting.vue                  #  자식2,       /users/:id/settings
  components/
     NavBar.vue                                 #  상단 내비게이션 메뉴
App.vue                                      #  최상위 컴포넌트 ( <RouterView /> )
```

1.2 Vue Router 심화

화면이 변경되는 위치 이해

App.vue

<NavBar />

<RouterView /> App에서 현재 라우트의 최상위 컴포넌트가 보여짐

Home.vue / 매칭시

UsersLayout.vue /users 매칭 시 path:'/users' (절대경로)

<RouterView /> 내용영역인 자식 컴포넌트가 보여짐

UsersHome.vue 기본 자식 , URL: /users, path:'' (상대경로)

UserDetail.vue 자식 1, URL: /users/1, path:':id' (상대경로)

UserSettings.uve 자식 2, URL: /users/1/settings, path:':id/settings' (상대경로)

1.2 Vue Router 심화

중첩 라우트

- 샘플 코드

```
const routes = [
  { path: "/", name: "home", component: Home },
  {
    path: "/users",
    component: UsersLayout,
    children: [
      { path: "", name: "users-home", component: UsersHome },
      { path: ":id", name: "user-detail", component: UserDetail },
      { path: ":id/settings", name: "user-settings", component: UserSettings },
    ],
  },
  { path: "/:pathMatch(.*)*", redirect: "/" },
];
```

1.2 Vue Router 심화

구현 코드

```
// main.js
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";
import "./assets/base.css";

createApp(App).use(router).mount("#app");
```

```
// index.js
import { createRouter, createWebHistory } from "vue-router";
import Home from "../views/Home.vue";
import UsersLayout from "../layouts/UsersLayout.vue";
import UsersHome from "../views/users/UsersHome.vue";
import UserDetail from "../views/users/UserDetail.vue";
import UserSettings from "../views/users/UserSettings.vue";

const routes = [
  { path: "/", name: "home", component: Home },
  {
    path: "/users",
    component: UsersLayout, // 부모 (여기에 <RouterView/>가 있음)
    children: [
      { path: "", name: "users-home", component: UsersHome },
      { path: ":id", name: "user-detail", component: UserDetail },
      { path: ":id/settings", name: "user-settings", component: UserSettings },
    ],
  },
  { path: "/:pathMatch(.*)*", redirect: "/" },
];
```

1.2 Vue Router 심화

구현 코드

```vue
<!-- App.vue -->
<template>
  <div class="app">
      <!- 공통 내비게이션 -->
    <NavBar />
    <main class="container">
      <!-- 최상위 라우트 렌더링 -->
      <RouterView />
    </main>
  </div>
</template>

<script setup>
import NavBar from "./components/NavBar.vue";
</script>
```

```vue
<!-- NavBar.vue -->
<template>
  <nav class="nav">
    <RouterLink to="/" class="brand">MyApp</RouterLink>
    <div class="links">
      <RouterLink to="/">Home</RouterLink>
      <RouterLink to="/users">Users</RouterLink>
    </div>
  </nav>
</template>

<script setup>
import { RouterLink } from "vue-router";
</script>
```

1.2 Vue Router 심화

구현 코드

```vue
<!-- Home.vue -->
<template>
  <section class="card">
    <h1>Home</h1>
    <p>중첩 라우트 실습</p>
  </section>
</template>
```

```vue
<!-- UsersLayout.vue -->
<template>
  <div class="card">
    <h2>Users Layout</h2>
    <p class="muted">
      왼쪽(상단)은 고정 레이아웃, 아래 영역이 자식 라우트로 교체됩니다.
    </p>

    <div class="subnav">
      <RouterLink to="/users">Users Home</RouterLink>
      <RouterLink :to="{ name: 'user-detail', params: { id: 1 } }">
        User 1 Detail
      </RouterLink>
      <RouterLink :to="{ name: 'user-settings', params: { id: 1 } }">
        User 1 Settings
      </RouterLink>
      >
    </div>
    <!-- 여기가 자식이 그려지는 자리 -->
    <RouterView />
  </div>
</template>
```

1.2 Vue Router 심화

구현 코드

```vue
<!-- UsersHome.vue -->
<template>
  <section class="card">
    <h3>Users Home</h3>
    <p>사용자 홈</p>
  </section>
</template>
<script setup></script>
```

```vue
<!-- UserDetail.vue -->
<template>
  <section class="card">
    <h3>User Detail</h3>
    <p><strong>ID:</strong>
{{ id }}</p>
  </section>
</template>

<script setup>
import { useRoute } from "vue-router";
const route = useRoute();
const id = route.params.id;
</script>
```

```vue
<!-- UserSettings.vue -->
<template>
  <section class="card">
    <h3>User Settings</h3>
    <p>사용자 {{ id }}의 환경설정</p>
  </section>
</template>

<script setup>
import { useRoute } from "vue-router";
const route = useRoute();
const id = route.params.id;
</script>
```

1.2 Vue Router 심화

지연 로딩 (Lazy Loading)

- 개념
  - 초기 로드 시 페이지를 처음부터 모두 불러오지 않음
  - 대신에 해당 페이지에 진입할 때 필요한 컴포넌트를 불러오는 방식
- 문법

  component: () => import (…)

  ex. component: () => import (‘../views/Home.vue’)
- 특징
  - 초기 로딩 속도가 빠름
  - 사용자 경험(UX) 개선
  - Vite 는 코드 스플리팅 자동 처리

1.2 Vue Router 심화

코드 스플리팅 (Code Splitting)

- 개념
  - 어플리케이션 전체 코드를 여러 개의 작은 번들 파일로 나누는 기술
- 특징
  - 지연 로딩(Lazy Loading) 활용하면 자동으로 코드 스플리팅 발생
  - 특정 페이지에 진입할 때 필요한 코드만 다운로드
  - Vite 는 내부적으로 자동 지원

1.2 Vue Router 심화

구현 코드

```
// main.js
import { createApp } from "vue";
import App from "./App.vue";
import router from "./router";

createApp(App).use(router).mount("#app");
```

```
// index.js
import { createRouter, createWebHistory } from "vue-router";
import Home from "../views/Home.vue";                // 초기 화면은 즉시 로딩
const About = () => import("../views/About.vue"); // 지연 로딩 (동적 import)

const routes = [
  { path: "/", component: Home },
  { path: "/about", component: About },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

1.2 Vue Router 심화

구현 코드

```vue
<!-- App.vue -->
<template>
  <nav class="nav">
    <RouterLink to="/">Home</RouterLink>
    <RouterLink to="/about">
          About(지연 로딩)
    </RouterLink>
  </nav>
  <RouterView />
</template>
```

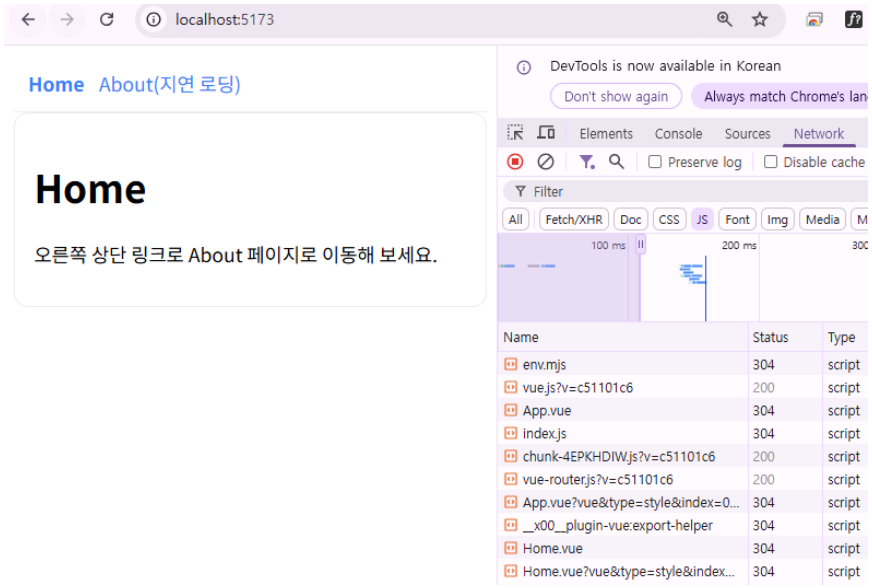
```vue
<!-- Home.vue -->
<template>
  <section class="card">
    <h1>Home</h1>
    <p>오른쪽 상단 링크로 About 페이지로 이동</p>
  </section>
</template>
```

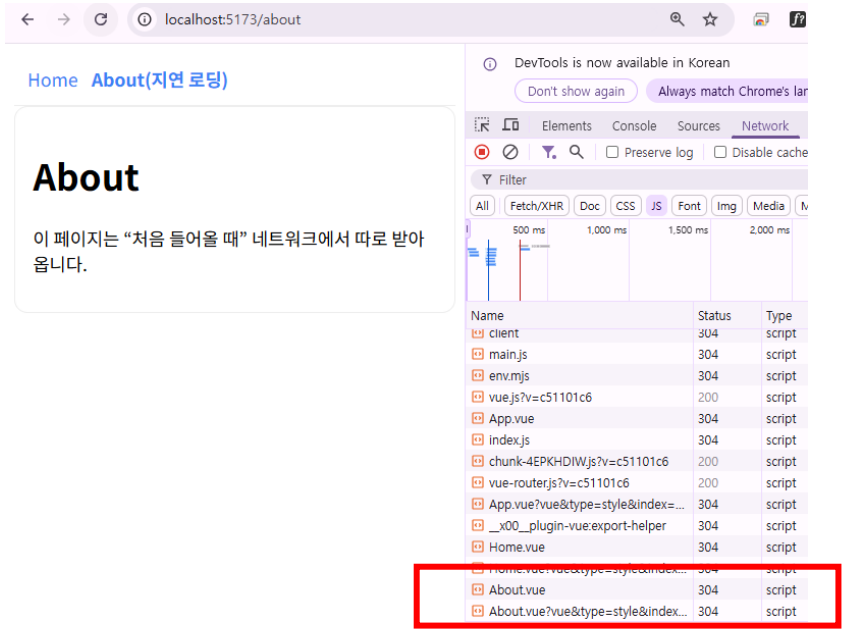
```vue
<!-- About.vue -->
<template>
  <section class="card">
    <h1>About</h1>
    <p>이 페이지는 “처음 들어올 때” 네트워크에서 따로 받아옴</p>
  </section>
</template>

<script setup>
console.log("[About] 동적 import로 로딩됨! (첫 진입 시 1회 출력)");
</script>
```

1.2 Vue Router 심화

실행 결과 확인

크롬 개발자 도구 > Network > js 필터링





1.2 Vue Router 심화

내비게이션 가드(Navigation Guards)

- 개념
  - 사용자가 페이지를 이동하기 전에 조건을 검사
  - 결과에 따라서 이동을 허용/차단/우회 시키는 방법
- 목적
  - 로그인/권한 체크
  - 데이터 불러오기 전 준비
  - 잘못된 접근 차단
- 종류
  - 전역 가드 (Global Guards)
  - 개별 라우트 가드 (Per-Route Guards)
  - 컴포넌트 내부 가드 (In-Component Guards)

1.2 Vue Router 심화

전역 가드 (Global Guards)

- 적용 범위
  - 어플리케이션 전체
  - 설정 위치: src/router/index.js
- 주요 메서드
  - beforeEach: 라우트 진입 전 실행
  - beforeResolve: 모든 가드 및 컴포넌트 hook 실행 후, 내비게이션 확정 직전
  - afterEach: 라우트 전환 후 실행, return 없음
- 특징
  - 앱 전역 공통 로직 ex. 로그인 여부, 로깅, 로딩 스피너 표시
  - Vue Router 4 에서는 next() 대신에 return 문으로 이동 제어 권장

1.2 Vue Router 심화

beforeEach

- 실행시점: 라우트로 진입하기 직전
- 문법: router.beforeEach((to, from)=>{… return 값})
- 파라미터
  - to : 이동 목적지 (path/fullPath/name/params/query/meta 등 사용 가능)
  - from : 이전 라우트
- 리턴
  - 허용 : return true; 또는 return;
  - 이동취소 : return false; (이전 라우트에 멈춤)
  - 리다이렉트 : return '/login' ,return {path:'/login', query:{id:'a01'}}, return {name:'login'};
- 주요용도
  - 권한 체크 ex. 로그인 여부
  - 공통 로직 실행 ex. 로깅, 통계 수집

1.2 Vue Router 심화

beforeResolve

- 실행시점 : 모든 가드(전역/개별/컴포넌트) 실행 후 와 내비게이션 확정 직전
- 문법 : router.beforeResolve((to, from)=>{… return 값})
- 파라미터
  - to : 이동 목적지 (path/fullPath/name/params/query/meta 등 사용 가능)
  - from : 이전 라우트
- 리턴
  - 허용 : return true; 또는 return;
  - 이동취소 : return false; (이전 라우트에 멈춤)
  - 리다이렉트 : return '/login' ,return {path:'/login', query:{id:'a01'}}, return {name:'login'};
- 주요 용도
  - 마지막 순간에 최종 검증/추적/로깅 ex. 결제완료 여부, 최종 권한체크
  - 데이터 프리패칭(Prf-fetching) ex. 상품상세 진입전에 상품 정보 불러오기

1.2 Vue Router 심화

afterEach

- 실행시점 : 내비게이션 완료 후 (라우트 전환 끝난 직후)
- 문법 : router.afterEach((to, from)=>{…})
- 파라미터
  - to : 이동 목적지 (path / fullPath / name / params / query / meta 등 사용 가능)
  - from : 이전 라우트
- 리턴
  - 없음
- 주요용도
  - UI 후처리 ex. 로딩 인티케이터 숨기기, 상태 초기화
  - 스크롤 복원/초기화 ex. 화면 최상단으로 이동

1.2 Vue Router 심화

전역 가드 요약

가드	실행 시점	리턴 값	주요 용도
beforeEach	라우트 진입 전	true / false / 라우트 객체	권한/인증
beforeResolve	확정 직전	true / false / 라우트 객체	최종 검증
afterEach	전환 후	void	로깅/분석

1.2 Vue Router 심화

폴더 구조 (전역 가드)

```
src
  main.js
  state.js              # 상태 (로그인 여부, 로딩)
  router/
     index.js                 # 전역 가드 3 종 적용
  views/
     Home.vue
     About.vue          # requiresAuth  meta 적용 (로그인 필요)
     Login.vue
  App.vue               # 로그인/로그아웃
```

1.2 Vue Router 심화

구현 코드 (전역 가드)

```
// state.js
import { reactive } from "vue";

export const appState = reactive({
  loading: false, // 전역 로딩 스피너 표시용
  loggedIn: false, // 로그인 여부 (샘플용 토글)
});
export function isLoggedIn() {
  return appState.loggedIn;
}
```

```
// index.js
import { createRouter, createWebHistory } from "vue-router";
import Home from "../views/Home.vue";
import About from "../views/About.vue";
import Login from "../views/Login.vue";
import NotFound from "../views/NotFound.vue";
import { appState, isLoggedIn } from "../state";

const routes = [
  { path: "/", component: Home },
  // About 페이지는 로그인 필요하도록 메타 옵션 설정
  { path: "/about", component: About, meta: { requiresAuth:
true } },
  { path: "/login", component: Login },
];});
```

1.2 Vue Router 심화

```
const router = createRouter({
  history: createWebHistory(),
  routes,
});
//전역 가드 1) beforeEach
router.beforeEach((to, from) => {
  console.log("beforeEach:", `${from.fullPath} -
> ${to.fullPath}`);
  appState.loading = true;
  if (to.meta.requiresAuth && !isLoggedIn()) {
    return { path: "/login", query: { redirect:
to.fullPath } };
  }
  return true;
});
//전역 가드 2) beforeResolve
router.beforeResolve(async (to, from) => {
  console.log("beforeResolve:", `${from.fullPath}
-> ${to.fullPath}`);
  await new Promise((r) => setTimeout(r, 300));
  return true;
});
//전역 가드 3) afterEach
router.afterEach((to, from) => {
  console.log("afterEach:", `${from.fullPath} ->
${to.fullPath}`);
  appState.loading = false;
});
export default router;
```

1.2 Vue Router 심화

구현 코드 (전역 가드)

```vue
<!-- App.vue -->
<template>
  <div class="app">
    <header class="nav">
      <RouterLink to="/">Home</RouterLink>
      <RouterLink to="/about">About(로그인 필요)</RouterLink>
      <!-- 로그인 상태에 따라 Login/Logout 토글 -->
      <button v-if="appState.loggedIn" class="btn ghost"
@click="logout">
        로그아웃
      </button>
      <button v-else class="btn ghost" @click="login">로그인
</button>
    </header>

    <!-- 전역 로딩 오버레이 -->
    <div v-if="appState.loading" class="overlay">
      <div class="spinner"></div>
      <p>Loading...</p>
    </div>

    <main class="container">
      <RouterView />
    </main>
  </div>
</template>
```

```vue
<script setup>
import { useRoute, useRouter } from "vue-router";
import { appState } from "./state";
const route = useRoute();
const router = useRouter();
function login() {
  appState.loggedIn = true;
  // 로그인 후 원래 가려던 곳으로
  const target = route.query.redirect || "/";
  router.push(String(target));
}
function logout() {
  appState.loggedIn = false;
  // 로그아웃 후에는 Home 으로.
  router.push("/");
}
</script>
```

1.2 Vue Router 심화

구현 코드 (전역 가드)

```vue
<!-- Home.vue -->
<template>
  <section class="card">
    <h1>Home</h1>
    <p>이 페이지는 누구나 접근 가능합니다.</p>
  </section>
</template>

<script setup></script>
```

```vue
<!-- About.vue -->
<template>
  <section class="card">
    <h1>About (Protected)</h1>
    <p>로그인된 사용자만 볼 수 있는 페이지입니다.</p>
  </section>
</template>

<script setup></script>
```

```vue
<!-- Login.vue -->
<template>
  <section class="card">
    <h1>Login 필요</h1>
    <p>로그인후 다시 요청하세요</p>
    <p style="margin-top: 12px; color: #64748b">
      현재 상태:
      <strong>{{ appState.loggedIn ? "로그인됨" : "로그아웃
" }}</strong>
    </p>
  </section>
</template>

<script setup>
import { appState } from "../state";
</script>
```

1.2 Vue Router 심화

개별 라우트 가드(Per-Route Guards)

- 적용 범위
  - 특정 라우트 단위
  - 설정 위치: route 설정 객체 내부
- 주요 메서드
  - beforeEnter: 특정 개별 라우트 진입 전 실행
- 특징
  - 라우트 단위에서 세밀하게 접근제어 가능
  - 관리자 페이지, 특정 권한이 필요한 화면 등에서 주로 사용

1.2 Vue Router 심화

beforeEnter

- 실행시점 : 특정 개별 라우트로 진입하기 직전
- 문법 : beforeEnter: (to, from)=>{… return 값}
- 파라미터
  - to : 이동 목적지 (path / fullPath / name / params / query / meta 등 사용 가능)
  - from : 이전 라우트
- 리턴
  - 허용 : return true; 또는 return;
  - 이동취소 : return false; (이전 라우트에 멈춤)
  - 리다이렉트 : return '/login' ,return {path:'/login', query:{id:'a01'}}, return {name:'login'};
- 주요용도
  - 전역이 아닌 특정 라우트만 별도 권한 체크 ex. /admin 은 관리자만 접근
  - 라우트 별 전처리 ex. 데이터 프리페칭

1.2 Vue Router 심화

폴더 구조 (개별 라우트 가드)

```
src
  main.js
  state.js              # 상태 ( role 정의: admin|guest )
  router/
     index.js           # 개별 라우드 가드 적용 ( admin 체크 )
  views/
     Home.vue
     About.vue        # requiresAuth  meta 적용 (로그인 필요)
     Admin.vue        #   admin 만 접근 가능
     Login.vue
  App.vue
```

1.2 Vue Router 심화

구현 코드 (개별 라우트 가드)

```
// state.js
import { reactive } from "vue";

export const appState = reactive({
  loading: false, // 전역 로딩 스피너 표시용
  loggedIn: false, // 로그인 여부 (샘플용 토글)

  user: { role: "guest" }, // "guest" | "admin" ,  실
습시 명시적으로 변경
});

export function isLoggedIn() {
  return appState.loggedIn;
}

export function isAdmin() {
  return appState.loggedIn && appState.user.role ===
"admin";
}
```

1.2 Vue Router 심화

구현 코드 (개별 라우트 가드)

```
// index.js
import { createRouter, createWebHistory } from "vue-
router";
import Home from "../views/Home.vue";
import About from "../views/About.vue";
import Login from "../views/Login.vue";
import NotFound from "../views/NotFound.vue";
import Admin from "../views/Admin.vue";
import { appState, isLoggedIn } from "../state";

import { isAdmin } from "../state";

const routes = [
  { path: "/", component: Home },
  // About 페이지는 로그인 필요하도록 메타 옵션 설정
  { path: "/about", component: About, meta:
{ requiresAuth: true } },
  { path: "/login", component: Login },
```

```
// 관리자 전용 페이지: 라우트 단위 접근 제어
  {
    path: "/admin",
    component: Admin,
    beforeEnter: (to, from) => {
      console.log("beforeEnter:", `${from.fullPath} ->
${to.fullPath}`);
      // 인증/권한 체크
      if (!isAdmin()) {
        // 접근 불가: 로그인으로 돌리고, 돌아올 경로를 쿼리로 넘김
        return { path: "/login", query: { redirect:
to.fullPath } };
      }
      // 통과
      return true;
    },
  },

  { path: "/:pathMatch(.*)*", name: "NotFound", component:
NotFound }, // Catch-all
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});
```

1.2 Vue Router 심화

구현 코드 (개별 라우트 가드)

```
// index.js
//전역 가드 1) beforeEach
router.beforeEach((to, from) => {
  console.log("beforeEach:", `${from.fullPath} ->
${to.fullPath}`);
  appState.loading = true;
  if (to.meta.requiresAuth && !isLoggedIn()) {
    return { path: "/login", query: { redirect:
to.fullPath } };
  }
  return true;
});
//전역 가드 2) beforeResolve
router.beforeResolve(async (to, from) => {
  console.log("beforeResolve:", `${from.fullPath} ->
${to.fullPath}`);
  await new Promise((r) => setTimeout(r, 300));
  return true;
});
//전역 가드 3) afterEach
router.afterEach((to, from) => {
  console.log("afterEach:", `${from.fullPath} ->
${to.fullPath}`);
  appState.loading = false;
});
export default router;
```

1.2 Vue Router 심화

구현 코드 (개별 라우트 가드)

```
<!-- Admin.vue -->
<template>
  <section class="card">
    <h1>Admin</h1>
    <span class="badge">관리자 전용</span> 권한 인증 확
인.
  </section>
</template>

<script setup>
//  (접근 제어는 라우터의 beforeEnter에서 처리)
</script>
```

```
<!-- Login.vue -->
<template>
  <section class="card">
    <h1>Login 필요</h1>
    <p>로그인후 다시 요청하세요</p>
    <p style="margin-top: 12px; color: #64748b">
      현재 상태:
      <strong>{{ appState.loggedIn ? "로그인됨" : "로그
아웃" }}</strong>
    </p>
    <p style="margin-top: 12px; color: #64748b">
      현재 ROLE:
      <strong>{{ appState.user.role }}</strong>
    </p>
  </section>
</template>

<script setup>
import { appState } from "../state";
</script>
```

1.2 Vue Router 심화

컴포넌트 내부 가드(In-Component Guards)

- 적용 범위
  - 개별 컴포넌트만 적용
  - 설정 위치: 컴포넌트 내부
- 주요 메서드
  - onBeforeRouteUpdate: 동일 컴포넌트에서 라우트만 갱신될 때 실행
  - onBeforeRouteLeave: 컴포넌트 떠나기 전 실행
- 특징
  - 컴포넌트 로직과 밀접
  - 정말 현재 페이지를 나갈까요? 같은 UX 제어에 적합

1.2 Vue Router 심화

onBeforeRouteUpdate

- 실행시점 : 동일 컴포넌트가 재사용되면서 라우트만 바뀔 때
- 문법 : onBeforeRouteUpdate((to, from)=>{…})
- 파라미터
  - to : 이동 목적지 (path / fullPath / name / params / query / meta 등 사용 가능)
  - from : 이전 라우트
- 리턴
  - 없음
- 주요 용도
  - 동일 페이지 내 데이터 갱신 ex. /product/1 에서 /product/2 이동시, 데이터만 다시 로딩
  - 검색/필터 결과 갱신 ex. /search?q=apple 에서 /search?q=banana

1.2 Vue Router 심화

onBeforeRouteLeave

- 실행시점 : 현재 컴포넌트에서 다른 라우트로 떠나기 직전
- 문법 : onBeforeRouteLeave((to, from)=>{… return 값})
- 파라미터
  - to : 이동 목적지 (path / fullPath / name /params / query / meta 등 사용 가능)
  - from : 이전 라우트
- 리턴
  - 허용 : return true; 또는 return;
  - 이동취소 : return false; (이전 라우트에 멈춤)
  - 리다이렉트 : return '/login' ,return {path:'/login', query:{id:'a01'}}, return {name:'login'};
- 주요용도
  - 폼 작성 도중 페이지 이탈방지
  - 네트워크 연결 종료/정리 ex. socket.close()

1.2 Vue Router 심화

폴더 구조 (컴포넌트 내부 가드)

```
src
  main.js
 router/
      index.js
  views/
      Home.vue              # onBeforeRouteUpdate 적용
      User.vue
      Form.vue              # 폼 입력 도중 페이지 이탈 방지, onBeforeRouteLeave 적용
      NotFound.vue
  App.vue
```

1.2 Vue Router 심화

구현 코드 (컴포넌트 내부 가드)

```
// index.js
import { createRouter, createWebHistory } from "vue-router";
import Home from "../views/Home.vue";
import Form from "../views/Form.vue";
import NotFound from "../views/NotFound.vue";

const routes = [
  { path: "/", component: Home },
  { path: "/form", component: Form },
  { path: "/:pathMatch(.*)*", name: "NotFound", component:
NotFound },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

```
<!-- App.vue -->
<template>
  <div class="app">
    <nav class="nav">
      <RouterLink to="/">Home</RouterLink>
      <RouterLink to="/form">Form</RouterLink>
    </nav>
    <main class="container">
      <RouterView />
    </main>
  </div>
</template>

<script setup></script>
```

1.2 Vue Router 심화

구현 코드 (컴포넌트 내부 가드)

```vue
<!-- Form.vue -->
<template>
  <section class="card">
    <h1>Form</h1>
    <p>
      실습방법: input태그에 값 입력후 [저장] 버튼 클릭하지 말고
Home 버튼 클릭
    </p>
    <input class="input" v-model="content" placeholder="입
력해보세요…" />
    <button class="btn" @click="save">저장</button>
    <p v-if="saved" style="color: #64748b; margin: 8px 0
0">저장됨~</p>
  </section>
</template>


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

const content = ref("");
const saved = ref(false);

function save() {
  // 실제 앱이면 서버에 저장.
  if (content.value) {
    saved.value = true;
    content.value = "";
  }
}
/**
 * 이 컴포넌트를 떠나기 직전 실행
 * - 입력이 있고 아직 저장 안 했으면 확인창 보임
 */
onBeforeRouteLeave((to, from) => {
  const dirty = content.value && !saved.value;
  if (dirty) {
    const ok = window.confirm(
      "정말 현재 페이지를 나갈까요? 저장되지 않은 내용이 있습니
다."
    );
    if (!ok) return false; // 내비게이션 취소
  }
  return true; // 이동 허용
});
</script>
```

1.3 HTTP 클라이언트 (Axios)

구글에서 제공하는 Firebase 백엔드 서비스 플랫폼과 Axios 라이브러리를 활용하여 CRUD 어플리케이션을 구축한다.

학습목표

- HTTP & REST 개요
- Firebase 서버 환경 구축
- Axios 개요
- REST 기반 CRUD 실습

1.3 HTTP 클라이언트 (Axios)

HTTP (HyperText Transfer Protocol)

- 개념
  - 웹에서 클라이언트(브라우저)와 서버가 데이터를 주고 받는 약속
  - 흐름 :
    클라이언트 : 요청(request) 전송
    서버 : 응답(response) 반환

- 특징
  - 텍스트, 이미지, 비디오 등 다양한 형식 사용
  - TCP/IP 기반
  - 무상태(Stateless) 통신

1.3 HTTP 클라이언트 (Axios)

HTTP 요청 (Request)

- 구성요소 3가지
  A. 요청라인(Request line)
     - 메서드(Method): GET/POST/PUT/PATCH/DELETE
     - 경로(Path): ex. /todos
     - 프로토콜 버전(HTTP version): HTTP/1.1
  B. 헤더(header) : 추가 정보 ex. Content-Type, Authorization, User-Agent
  C. 바디(body) : 실제 전송 데이터

```
POST /todos HTTP/1.1                  # 요청라인
Content-Type: application/json        # 헤더

{                                     # 바디
   "title": “Vue3",
   "completed": false
}
```

1.3 HTTP 클라이언트 (Axios)

HTTP 응답 (Response)

- 구성요소 3가지
  A. 상태라인(Status line)
     - 프로토콜 버전(HTTP version): HTTP/1.1
     - 상태 코드(Status Code): ex. 200/400/500
     - 상태 메시지(Reason Phrase): ex. OK/NOT FOUND/Internal Server Error
  B. 헤더(header) : 추가 정보 ex. Content-Type, Content-Length, Set-Cookie
  C. 바디(body) : 실제 응답 데이터

```
HTTP/1.1 200 OK                          # 상태라인
Content-Type: application/json           # 헤더
Content-Length: 48
{                                        # 바디
   "title": “Vue3",
   "completed": false
}
```

1.3 HTTP 클라이언트 (Axios)

REST

- 개념
  - 웹 서비스를 위한 아키텍처
  - 자원(resource)을 URI로 표현
  - HTTP 메서드(method)를 통해서 해당 자원에 대한 행위를 정의
- 이름 의미
  - Representational
    자원을 다양한 표현(JSON,XML)으로 작성
  - State Transfer
    자원의 상태를 클라이언트와 서버 간에 전송

1.3 HTTP 클라이언트 (Axios)

REST

- 핵심 용어
  - 자원(Resource)
  - 표현(Representation)
  - 메서드(Method)
  - 상태코드(Status Code)
  - 엔드 포인트(Endpoint)
- 핵심 아이디어
  - 클라이언트가 서버의 자원 상태를 표현으로 요청
  - 서버는 해당 자원의 현재 상태를 응답으로 전달

1.3 HTTP 클라이언트 (Axios)

REST

- 특징
  - 클라이언트/서버 구조
  - 무상태 (Stateless)
  - 캐시 처리 가능 (Cacheable)
  - 자원 기반 (Resource-Oriented)
  - 표현 (Representation)
  - 일관된 인터페이스 (Uniform Interface)

1.3 HTTP 클라이언트 (Axios)

REST

- 장점
  - 구조 단순
  - HTTP 기반
  - 확장성과 유연성
  - 브라우저/모바일/IOT 호환
- 단점
  - 무상태
  - 복잡한 트랜잭션

1.3 HTTP 클라이언트 (Axios)

REST 메서드

- GET
  - 자원 조회. ex. /todos 전체 목록 조회
- POST
  - 자원 생성. ex. /todos 새로운 할 일 생성
- PUT/PATCH
  - 자원 전체/일부 수정. ex. /todos/1 id=1 데이터 전체/일부 수정
- DELETE
  - 자원 삭제. ex. /todos/1 id=1 제거

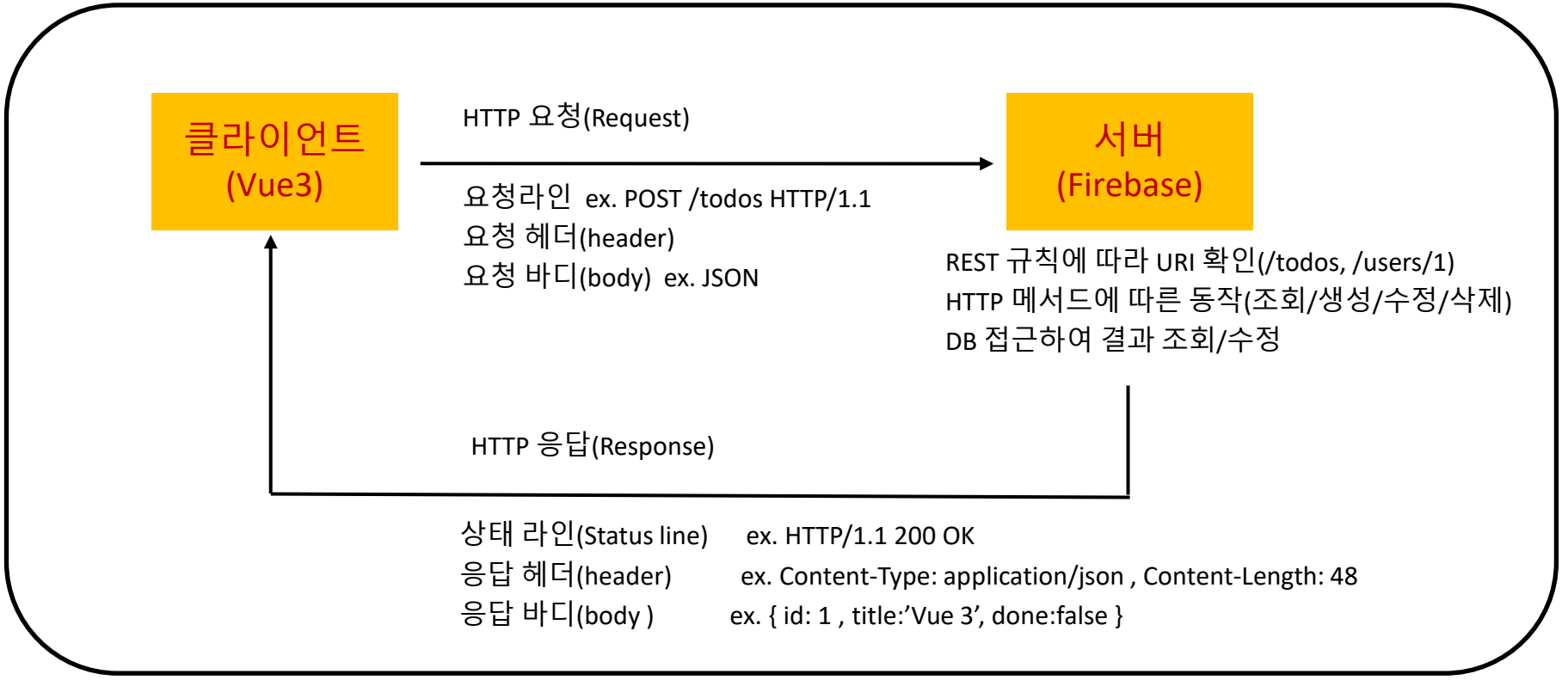
1.3 HTTP 클라이언트 (Axios)

주요 상태코드

코드	상태 메시지	용도
200	OK	조회/수정 성공 (GET/PUT 요청)
201	Created	새 자원 생성 성공 (POST 요청)
204	No Content	본문 없이 성공 (DELETE / AUR 등 요청)
400	Bad Request	형식 오류 검증 실패 (JSON 파싱 실패, 누락 필드 등)
403	Forbidden	인증했으나 권한 없음
404	Not Found	리소스 없음
405	Method Not Allowed	지원 안 하는 메서드
500	Internal Server Error	서버 내부 오류 (포괄)

1.3 HTTP 클라이언트 (Axios)

REST API 흐름



1.3 HTTP 클라이언트 (Axios)

Firebase 서버

- 개념
  - Google이 제공하는 백엔드 서비스 플랫폼 (BaaS : Backend as a Service)
  - 인증, DB, 파일저장, 호스팅,푸시 알림 등 서버 기능을 클라우드로 제공
  - 개발자는 클라이언트(UI)에만 집중하고 서버 기능은 대신 처리
- 특징
  - 사용이 쉬움
  - 실시간 처리
  - 서버리스
  - 확장성
  - 프리티어 지원

1.3 HTTP 클라이언트 (Axios)

Firebase 주요 서비스

- 인증 (Authentication)
- 데이터베이스 (Database)
- 스토리지 (Storage)
- 호스팅 (Hosting)
- 클라우드 함수 (Cloud Functions)
- 푸시 알림 (Cloud Messaging)

1.3 HTTP 클라이언트 (Axios)

Firebase 사용 방법

A. 구글 계정으로 로그인

   https://firebase.google.com/

B. 우측 상단 콘솔로 이동

C. 새 Firebase 프로젝트 만들기 선택

D. 프로젝트 이름 지정 후 계속 버튼 선택

   ex. MyFirstFirebaseProject

E. Firebase의 Gemini 사용설정 비활성화 하고 계속 버튼 선택

F. Google 애널리틱스 사용설정 비활성화 하고 프로젝트 만들기 버튼 선택

G. 왼쪽 빌드 항목 > **Realtime Database** 선택

H. 데이터베이스 만들기 선택

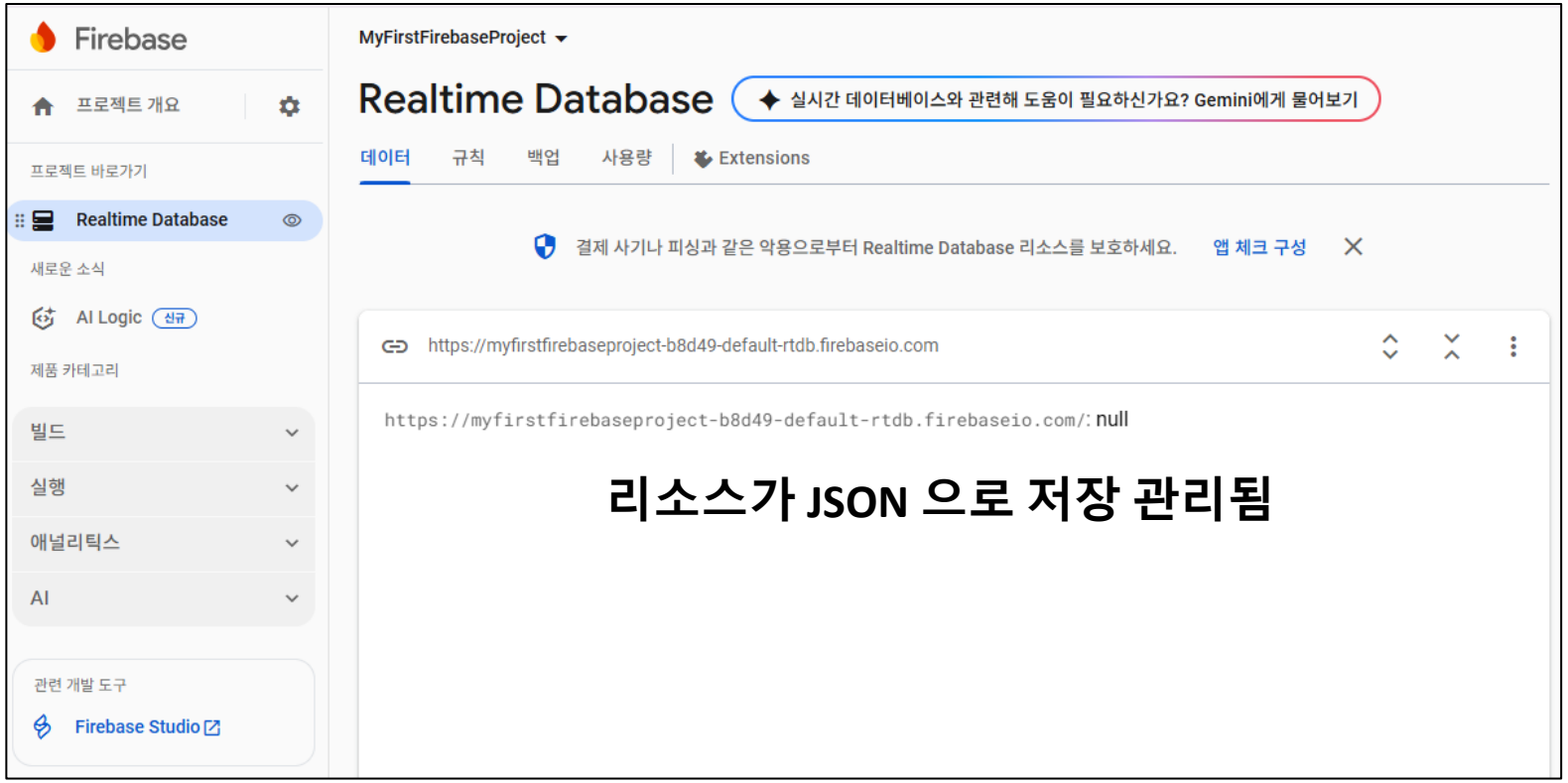
   • 위치: 미국

   • 규칙: **테스트 모드**에서 시작

E. 제공된 URL 복사하기 (코드에서 필요)

1.3 HTTP 클라이언트 (Axios)

Firebase 최종 화면



1.3 HTTP 클라이언트 (Axios)

Axios

- 개념
  - Promise 기반 HTTP 클라이언트 라이브러리
  - 브라우저와 Node.js 환경 모두 사용
- 설치 : npm install axios
- 특징
  - REST API 호출을 쉽고 직관적 처리
  - 인터셉터로 공통 처리 추가 ex. 토큰주입, 에러 메시지 표준화
  - 다양한 서버와 연동 및 자동 JSON 처리
  - 인스턴스 기반 공통 설정 ex. baseURL, headers
  - 에러 핸들링 용이 ex. try/catch

1.3 HTTP 클라이언트 (Axios)

Axios API

- 함수 호출형

  axios(config)

  config = { url, method, baseURL, params, data, headers, timeout, signal, …. }

- 샘플 코드

```
axios({
  method: 'post',
  url: ‘/todos',
  data: {
    title: ‘Vue3 Study',
    done: false,
    createdAt: Data.now()
  }
});
```

1.3 HTTP 클라이언트 (Axios)

Axios API

- HTTP별 단축 메서드

```
axios.get(url [, config] )
axios.post(url [, data [, config]] )
axios.put(url [, data [, config]] )
axios.patch(url [, data [, config]] )
axios.delete(url [, config] )
```

- 샘플 코드

```
axios.post('/todos/', data: {
   title: 'Vue3 Study',
   done: false,
   createdAt:Date.now()
 });
```

1.3 HTTP 클라이언트 (Axios)

Axios API

- 인스턴스
  - 사용자 지정 config로 새로운 Axios 인스턴스를 생성
  - 용도 : 공통 baseURL/headers 공유
  - axios.create([config])

- 샘플 코드

```
const instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,
  headers: {'Content-Type': 'application/json'}
});
```

1.3 HTTP 클라이언트 (Axios)

Axios API

- 인터셉터
  - 용도 : 요청과 응답을 가로채기

```
axios.interceptors.request.use(function(config){}, function(error){} );
axios.interceptors.response.use(function(config){}, function(error){} );
```

- 샘플 코드

```
axios.interceptors.request.use(
   (config) => {
     config.url="https://reqres.in/api/users?page=1"
    return config;
   }, (error) => {
    console.log("요청 interceptor error : " , error);
    return Promise.reject(error);
   });
```

1.3 HTTP 클라이언트 (Axios)

요청 config 핵심 요약

속성명	용도	사용 예시	추가정보
url	요청 대상	'/todos' , '/users/1'	유일한 필수값
method	HTTP 메서드	'get', 'post', 'put', 'patch', 'delete'	기본값 'get'
baseURL	공통 prefix	'https://api.example.com'	url에 prefix로 자동지정
params	쿼리스트링	{ page: 1, q:'vue3' }	값이 null/undefined 제외
data	본문 데이터	{name:'Lee', age:20 }	post/put/patch/delete 사용
headers	헤더 정보	{'Content-Type':'application/json' }	토큰, 컨텐츠 타입 지정
timeout	타임아웃	5000 (ms)	네트워크 지연 보호
withCredentials	CORS 쿠키	false	쿠키 포함 여부
responseType	응답 데이터 타입	'json','text','blob'	기본값 'json'

1.3 HTTP 클라이언트 (Axios)

응답 구조 핵심 요약

항목	설명	예시
data	서버가 실제로 보내준 응답 데이터	{ id: 1, name:'Alice' }
status	서버 응답의 HTTP 상태 코드	200 (성공), 500 (서버 오류)
statusText	상태 코드에 대한 문자 메시지	"OK", "Not Found"
headers	서버가 응답시 전달하는 헤더 정보	{"content-type":"application/json"}
config	Axios가 요청을 보낼 때 사용한 설정 정보 (url, method, headers 등)	{ url: 'https://example.com', method:'get' }
request	실제 네트워크 요청 객체 브라우저: XMLHttpRequest Node.js: ClientRequest	XMLHttpRequest { readyState: 4, status: 200 ….}

1.3 HTTP 클라이언트 (Axios)

응답 구조 예시 코드

```
import axios from "axios";

async function fetchUser() {
  try {
    const response = await axios.get("https://example.com/users/1");
    console.log("data:", response.data);                    // { id: 1, name: "Alice" }
    console.log("status:", response.status);                // 200
    console.log("statusText:", response.statusText);        // "OK"
    console.log("headers:", response.headers);              // { "content-type": "application/json", ... }
    console.log("config:", response.config);                // { url: "https://example.com/users/1" ... }
    console.log("request:", response.request);              // XMLHttpRequest 객체 (브라우저 환경)
  } catch (error) {
    console.error("Error:", error.message);
  }
}
```

1.3 HTTP 클라이언트 (Axios)

Axios 오류 처리 (기본)

A. axios 준비

- 공통설정 (서버주소, 헤더정보, 타임아웃) 관리

코드예시

```
// src/api/axios.js
const api = axios.create({
  baseURL: "https://example.com/api",                 // 서버 기본 주소
  headers: { "Content-Type": "application/json" },    // 헤더 정보
  timeout: 8000,                                      // 8초 넘으면 실패로 간주
});

export default api;
```

1.3 HTTP 클라이언트 (Axios)

Axios 오류 처리 (기본)

B. UI 상태 로직 처리 기본 패턴

- loading : 요청중 (스피너 표시)
- data: 성공시 표시할 데이터
- error: 실패시 사용자에게 보여줄 메시지
- 코드: async/await + try/catch

1.3 HTTP 클라이언트 (Axios)

코드예시

```
// src/composables/useTodo.js
import api from "../api/axios";
 const loading = ref(false);          // loading 처리
 const todos = ref([]);               // data  처리
 const error = ref(null);             // error 처리

 async function fetchTodos() {
   loading.value = true;
   error.value = null;
   try {
     const { data } = await api.get("/todos.json");
     todos.value = data;
   } catch (e) {
     console.error("조회 실패:", e?.response?.status, e?.message);                    // 개발자용
     error.value = "데이터를 불러오지 못했습니다. 잠시 후 다시 시도하세요."; // 사용자용
   } finally {
      loading.value = false;
   }
 }
return {fetchTodos}
```

1.3 HTTP 클라이언트 (Axios)

Axios 오류 처리 (기본)

C. 실제 요청 및 화면 처리

코드예시

```
//src/pages/TodoPage.vue

import { useTodos } from "../composables/useTodos";
const { todos, loading, error, fetchTodos } = useTodos();

<p v-if="loading" class="muted">
  <span class="spinner"></span> 불러오는 중…
</p>
<p v-if="error" class="error">{{ error }}</p>

<ul v-if="todos.length" class="list" aria-label="할 일 목록">
    <li v-for="t in todos" :key="t.id" class="item">{{t.title}}>/li>
</ul>
<p v-else class="muted">아직 항목이 없어요. 첫 할 일을 추가해보세요</p>
```

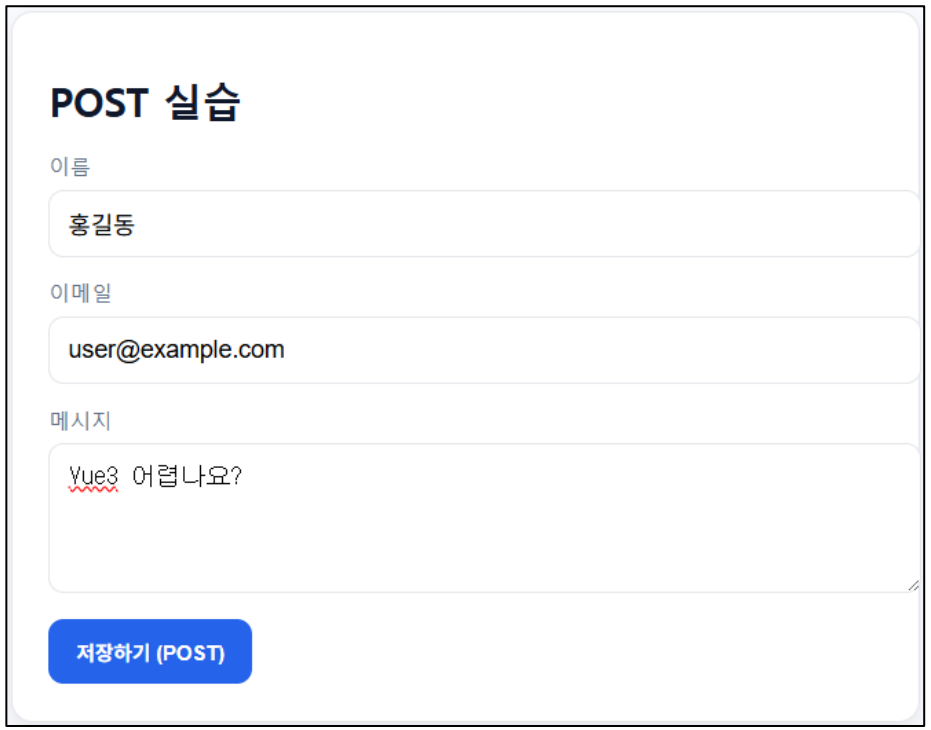
1.3 HTTP 클라이언트 (Axios)

try/catch vs then/catch

구분	async/await + try/catch	then().catch()
가독성	동기 코드처럼 읽히므로 직관적	체인구조로 이어지며 중첩 시 가독성 저하
에러처리	try{….}catch(e){…….} 로 명확히 처리	.catch()를 체인 끝에서 처리
디버깅	Stack trace가 간단해지고 코드 흐름이 이해하기 쉬움	비동기 체인으로 인해 에러 추적이 어려움
중첩 비동기	여러 await을 순차적으로 작성 가능 직관적임	.then().then() 으로 이어지다 보면 콜백 복잡성 증가
코드 스타일	최신 문법. 요즘 프로젝트에서 선호	기본 코드베이스, 빠른 테스트, 간단한 로직에 여전히 사용됨
학습 난이도	초보자도 동기처럼 읽힌다 는 점에서 이해하기 쉬움	Promise 개념을 알아야 하기 때문에 처음에는 난이도 있음
실무 활용도	대규모/팀 프로젝트에서 표준처럼 선호하고 사용	단순 샘플 코드, 짧은 유틸, 체이닝 필요 시 가끔 사용

1.3 HTTP 클라이언트 (Axios)

실습 (POST)



https://myfirstfirebaseproject-b8d49-default-rtdb.firebaseio.com

```
https://myfirstfirebaseproject-b8d49-default-rtdb.firebaseio.com/
  contacts
    -Oa2q-mJOvKoKsnjClvQ
      createdAt: 1758853597107
      email: "use@example.com"
      message: "Vue3 어렵나요?"
      name: "홍길동"
```

id값

1.3 HTTP 클라이언트 (Axios)

구현 코드 (POST)

```
# .env
VITE_FIREBASE_DB_URL=Firebase URL지정
```

```
// src/api/axios.js
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_FIREBASE_DB_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 8000,
});

export default api;
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (POST)

```
// src/composables/usePost.js
import { ref } from "vue";
import api from "../api/axios";

// 상태 관리
const loading = ref(false); // 로딩 상태
const okMsg = ref(""); // 성공 메시지
const error = ref(null); // 에러 메시지

// POST 요청 함수
async function submitForm(payload) {
  loading.value = true;
  okMsg.value = "";
  error.value = null;
  try {
    // Firebase REST API 규칙: 경로 끝에 .json 필수
    await api.post("/contacts.json", payload);
    okMsg.value = `'${payload.name}'님의 데이터가 저장`
  } catch (e) {
    console.error("저장 실패:", e?.response?.status, e?.message);
// 개발자용
    error.value = "저장 중 문제가 발생, 다시 시도."; // 사용자용
  } finally {
    loading.value = false;
  }
}

// 외부에서 사용할 값/함수 리턴
export function usePost() {
  return { loading, okMsg, error, submitForm };
}
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (POST)

```vue
<!-- App.vue -->
<script setup>
import { ref } from "vue";
import { usePost } from "./composables/usePost";

const name = ref("");
const email = ref("");
const message = ref("");

const { loading, okMsg, error, submitForm } = usePost();

async function handleSubmit() {
  if (!name.value.trim() || !email.value.trim() || !message.value.trim()) {
    return alert("모든 항목을 입력하세요.");
  }
  const payload = {
    name: name.value,
    email: email.value,
    message: message.value,
    createdAt: Date.now(),
  };
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (POST)

```
await submitForm(payload);
  if (okMsg.value) {    name.value = ""; email.value = ""; message.value = "";}
}
</script>
<template>
  <main class="container">
    <section class="card">
      <h1>POST 실습</h1>
      <label>이름</label>
      <input v-model="name" type="text" placeholder="홍길동" />
      <label>이메일</label>
      <input v-model="email" type="email" placeholder="user@mail.com" />
      <label>메시지</label>
      <textarea  v-model="message"  rows="4" placeholder="문의 내용을 입력"
      ></textarea>
      <button :disabled="loading" @click="handleSubmit">
        {{ loading ? "저장 중..." : "저장하기 (POST)" }}
      </button>
      <div v-if="error" class="toast err">{{ error }}</div>
      <div v-if="okMsg" class="toast ok">{{ okMsg }}</div>
    </section>
  </main>
</template>
```

1.3 HTTP 클라이언트 (Axios)

실습 (GET)

https://myfirstfirebaseproject-b8d49-default-rtdb.firebaseio.com


```
https://myfirstfirebaseproject-b8d49-default-rtdb.firebaseio.com/
  contacts
    -Oa2q-mJOvKoKsnjClvQ
      createdAt: 1758853597107
      email: "use@example.com"
      message: "Vue3 어렵나요?"
      name: "홍길동"
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (GET)

```
# .env
VITE_FIREBASE_DB_URL=Firebase URL지정
```

```
// src/api/axios.js
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_FIREBASE_DB_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 8000,
});

export default api;
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (GET)

```
// src/composables/useFetch.js
import { ref } from "vue";
import api from "../api/axios";

// 상태 관리
const loading = ref(false); // 로딩 상태
const data = ref([]); // data
const error = ref(null); // 에러 메시지
```

```
  // GET 요청 함수
async function fetchTodos() {
  loading.value = true;
  error.value = null;
  try {
    const { data: res } = await api.get("/contacts.json");
    // Firebase는 객체 형태를 반환하므로 배열로 변환
    data.value = res
      ? Object.entries(res).map(([id, v]) => ({ id, ...v }))
      : [];
    console.log(data.value);
  } catch (e) {
    console.error("조회 실패:", e?.response?.status, e?.message);
// 개발자용
    error.value = "데이터를 불러오지 못함. 잠시 후 다시 시도."; //
사용자용
  } finally {
    loading.value = false;
  }
}

export function useFetch() {
  return { loading, data, error, fetchTodos };
}
```

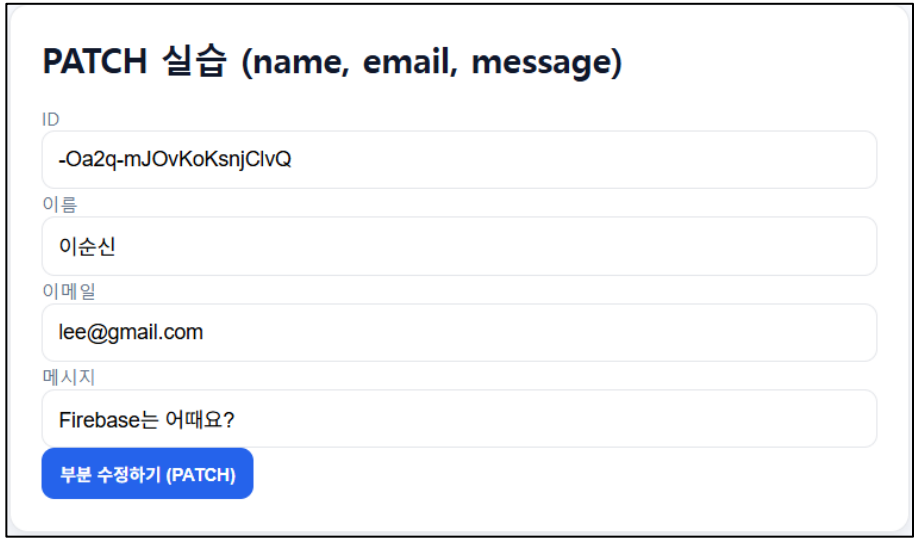
1.3 HTTP 클라이언트 (Axios)

구현 코드 (GET)

```
<!-- App.vue -->
<script setup>
import { useFetch } from "./composables/useFetch";
const { loading, data, error, fetchTodos } =
useFetch();
</script>

<template>
  <main class="container">
    <section class="card">
      <h1>GET 실습</h1>
      <button :disabled="loading"
@click="fetchTodos">
        {{ loading ? "불러오는 중..." : "데이터 불러오
기 (GET)" }}
      </button>
      <ul v-if="data.length">
        <li v-for="item in data" :key="item.id">
          {{ item.name }} / {{ item.email }} /
{{ item.message }}
        </li>
      </ul>
      <div v-if="error" class="toast
err">{{ error }}</div>
    </section>
  </main>
</template>
```

1.3 HTTP 클라이언트 (Axios)

실습 (PATCH)



1.3 HTTP 클라이언트 (Axios)

구현 코드 (PATCH)

```
# .env
VITE_FIREBASE_DB_URL=Firebase URL지정
```

```
// src/api/axios.js
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_FIREBASE_DB_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 8000,
});

export default api;
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (PATCH)

```
// src/composables/usePatch.js
import { ref } from "vue";
import api from "../api/axios";

// 상태 관리
const loading = ref(false); // 로딩 상태
const okMsg = ref(""); // 성공 메시지
const error = ref(null); // 에러 메시지

// PATCH 요청 함수 (name, email, message만 수정)
async function updateContact(id, { name, email,
message }) {
  loading.value = true;
  okMsg.value = "";
  error.value = null;
```

```
try {
    // Firebase REST API: 특정 id 항목 수정
    // 예: /contacts/<id>.json
    await api.patch(`/contacts/${id}.json`, {
      name,
      email,
      message,
      updatedAt: Date.now(),
    });

    okMsg.value = ` '${id}' 항목이 수정되었음`;
  } catch € {
    console.error("PATCH 실패:", e?.response?.status, e?.message); // 개발자용
    error.value = " 수정 중 문제가 발생. 다시 시도"; // 사용자용
  } finally {
    loading.value = false;
  }
}

export function usePatch() {
  return { loading, okMsg, error, updateContact };
}
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (PATCH)

```vue
<!-- App.vue -->
<script setup>
import { ref } from "vue";
import { usePatch } from "./composables/usePatch";

const { loading, okMsg, error, updateContact } = usePatch();

// 입력값
const id = ref("");
const name = ref("");
const email = ref("");
const message = ref("");

async function handlePatch() {
  if (!id.value.trim()) return alert("ID를 입력하세요.");
  await updateContact(id.value, {
    name: name.value,
    email: email.value,
    message: message.value,
  });
}
</script>
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (PATCH)

```
<template>
  <main class="container">
    <section class="card">
      <h1>PATCH 실습 (name, email, message)</h1>
      <label>ID</label>
      <input v-model="id" placeholder="예: -Nabc123xyz..." />
      <label>이름</label>
      <input v-model="name" placeholder="새 이름 입력" />
      <label>이메일</label>
      <input v-model="email" placeholder="새 이메일 입력" />
      <label>메시지</label>
      <input v-model="message" placeholder="새 메시지 입력" />

      <button :disabled="loading" @click="handlePatch">
        {{ loading ? "수정 중..." : "부분 수정하기 (PATCH)" }}
      </button>

      <div v-if="okMsg" class="toast ok">{{ okMsg }}</div>
      <div v-if="error" class="toast err">{{ error }}</div>
    </section>
  </main>
</template>
```

1.3 HTTP 클라이언트 (Axios)

실습 (DELETE)

1.3 HTTP 클라이언트 (Axios)

구현 코드 (DELETE)

```
# .env
VITE_FIREBASE_DB_URL=Firebase URL지정
```

```
// src/api/axios.js
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_FIREBASE_DB_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 8000,
});

export default api;
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (DELETE)

```
// src/composables/useDelete.js
import { ref } from "vue";
import api from "../api/axios";

// 상태 관리
const loading = ref(false); // 로딩 상태
const okMsg = ref(""); // 성공 메시지
const error = ref(null); // 에러 메시지

// DELETE 요청 함수 (id로 삭제)
async function deleteById(id) {
  loading.value = true;
  okMsg.value = "";
  error.value = null;
```

```
try {
    if (!id) {
      error.value = "삭제할 ID가 필요합니다.";
      return;
    }

    // Firebase REST API 규칙: /contacts/<id>.json
    await api.delete(`/contacts/${id}.json`);

    okMsg.value = ` '${id}' 항목이 삭제되었습니다.`;
  } catch (e) {
    console.error("DELETE 실패:", e?.response?.status, e?.message); // 개발자용
    error.value = "삭제 중 문제가 발생. 다시 시도."; // 사용자용
  } finally {
    loading.value = false;
  }
}

export function useDelete() {
  return { loading, okMsg, error, deleteById };
}
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (DELETE)

```vue
<!-- App.vue -->
<script setup>
import { ref } from "vue";
import { useDelete } from "./composables/useDelete";

const { loading, okMsg, error, deleteById } = useDelete();
const id = ref("");

async function handleDelete() {
  if (!id.value.trim()) {
    return alert("삭제할 ID를 입력하세요.");
  }
  await deleteById(id.value);
}
</script>
```

```vue
<template>
  <main class="container">
    <section class="card">
      <h1>DELETE 실습 (id로 삭제)</h1>

      <label>ID</label>
      <input v-model="id" placeholder="예: -Nabc123xyz..." />

      <button :disabled="loading" @click="handleDelete">
        {{ loading ? "삭제 중..." : "삭제하기 (DELETE)" }}
      </button>

      <div v-if="okMsg" class="toast ok">{{ okMsg }}</div>
      <div v-if="error" class="toast err">{{ error }}</div>
    </section>
  </main>
</template>
```

1.3 HTTP 클라이언트 (Axios)

실습 (인터셉터)

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 전)

```
# .env
VITE_FIREBASE_DB_URL=Firebase URL지정
```

```
// src/api/axios.js
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_FIREBASE_DB_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 8000,
});

export default api;
```

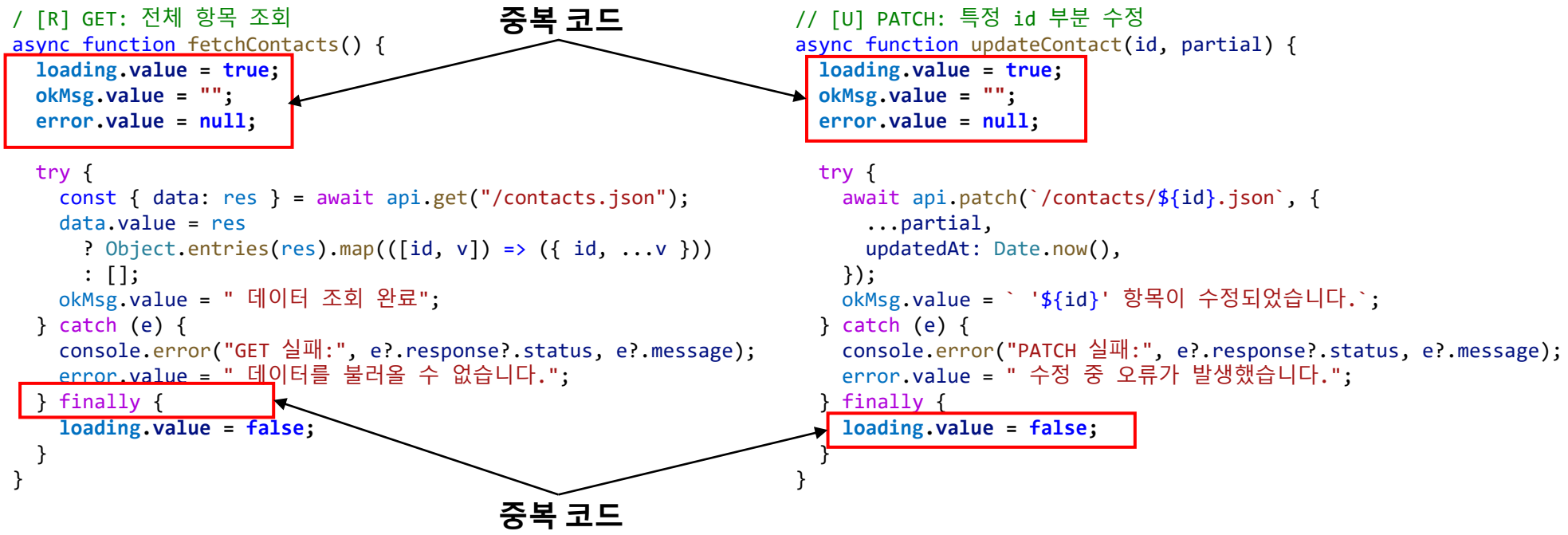
```
// src/composables/useContact.js
…

// [C] POST: 새 항목 추가
async function addContact(payload) {
  loading.value = true;
  okMsg.value = "";
  error.value = null;

  try {
    await api.post("/contacts.json", payload);
    okMsg.value = ` '${payload.name}'님의 데이터가 저장되었습니다.`;
  } catch (e) {
    console.error("POST 실패:", e?.response?.status, e?.message);
    error.value = " 저장 중 오류가 발생했습니다.";
  } finally {
    loading.value = false;
  }
}
```

중복 코드

중복 코드

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 전)




```
/ [R] GET: 전체 항목 조회
async function fetchContacts() {
  loading.value = true;
  okMsg.value = "";
  error.value = null;

  try {
    const { data: res } = await api.get("/contacts.json");
    data.value = res
      ? Object.entries(res).map(([id, v]) => ({ id, ...v }))
      : [];
    okMsg.value = " 데이터 조회 완료";
  } catch (e) {
    console.error("GET 실패:", e?.response?.status, e?.message);
    error.value = " 데이터를 불러올 수 없습니다.";
  } finally {
    loading.value = false;
  }
}
```

```
// [U] PATCH: 특정 id 부분 수정
async function updateContact(id, partial) {
  loading.value = true;
  okMsg.value = "";
  error.value = null;

  try {
    await api.patch(`/contacts/${id}.json`, {
      ...partial,
      updatedAt: Date.now(),
    });
    okMsg.value = ` '${id}' 항목이 수정되었습니다.`;
  } catch (e) {
    console.error("PATCH 실패:", e?.response?.status, e?.message);
    error.value = " 수정 중 오류가 발생했습니다.";
  } finally {
    loading.value = false;
  }
}
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 전)

```
// [D] DELETE: 특정 id 삭제
async function deleteContact(id) {
  loading.value = true;
  okMsg.value = "";
  error.value = null;

  try {
    await api.delete(`/contacts/${id}.json`);
    okMsg.value = ` '${id}' 항목이 삭제되었습니다.`;
  } catch (e) {
    console.error("DELETE 실패:", e?.response?.status,
e?.message);
    error.value = " 삭제 중 오류가 발생했습니다.";
  } finally {
    loading.value = false;
  }
}
```

중복 코드

중복 코드

```
// 외부로 내보내기
export function useContacts() {
  return {
    // 상태
    loading,
    data,
    okMsg,
    error,
    // 메서드
    addContact,
    fetchContacts,
    updateContact,
    deleteContact,
  };
}
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 후)

```
// store/contactState.js
import { ref } from "vue";

export const loading = ref(false);
export const data = ref([]);
export const okMsg = ref("");
export const error = ref(null);
```

```
// src/api/axios.js
import axios from "axios";
import { loading, error as errorState, okMsg } from
"../stores/contactState";

const api = axios.create({
  baseURL: import.meta.env.VITE_FIREBASE_DB_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 8000,
});

// 요청 인터셉터
api.interceptors.request.use(
  (config) => {
    loading.value = true;
    errorState.value = null;
    okMsg.value = "";
    return config;
  },
  (err) => Promise.reject(err)
);
```

요청 인터셉터로 처리

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 후)

```
// 응답 인터셉터
api.interceptors.response.use(
  (response) => {
    loading.value = false;
    return response;
  },
  (error) => {
    loading.value = false;
    console.error("Axios Error:", error.response?.status, error.message);
    errorState.value = " 서버 요청 중 오류가 발생했습니다.";
    return Promise.reject(error);
  }
);

export default api;
```

응답 인터셉터로 처리

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 후)

```
// src/composables/useContacts.js
import { ref } from "vue";
import api from "../api/axios";
import { loading, data, okMsg, error } from "../stores/contactState";

// [C] POST: 새 항목 추가
async function addContact(payload) {
  try {
    await api.post("/contacts.json", payload);
    okMsg.value = ` '${payload.name}'님의 데이터가 저장되었습니다.`;
  } catch (e) {
    console.error("POST 실패:", e?.response?.status, e?.message);
    // error 메시지는 인터셉터에서 설정됨
  }
}
```

```
// [R] GET: 전체 항목 조회
async function fetchContacts() {
  try {
    const { data: res } = await api.get("/contacts.json");
    data.value = res
      ? Object.entries(res).map(([id, v]) => ({ id, ...v }))
      : [];
    okMsg.value = " 데이터 조회 완료";
  } catch (e) {
    console.error("GET 실패:", e?.response?.status, e?.message);
  }
}
```

1.3 HTTP 클라이언트 (Axios)

구현 코드 (인터셉터 적용 후)

```
// [U] PATCH: 특정 id 부분 수정
async function updateContact(id, partial) {
  try {
    await api.patch(`/contacts/${id}.json`, {
      ...partial,
      updatedAt: Date.now(),
    });
    okMsg.value = ` '${id}' 항목이 수정되었습니다.`;
  } catch (e) {
    console.error("PATCH 실패:", e?.response?.status, e?.message);
  }
}
// [D] DELETE: 특정 id 삭제
async function deleteContact(id) {
  try {
    await api.delete(`/contacts/${id}.json`);
    okMsg.value = ` '${id}' 항목이 삭제되었습니다.`;
  } catch (e) {
    console.error("DELETE 실패:", e?.response?.status, e?.message);
  }
}
```

```
// 외부로 내보내기
export function useContacts() {
  return {
    // 상태
    loading,
    data,
    okMsg,
    error,
    // 메서드
    addContact,
    fetchContacts,
    updateContact,
    deleteContact,
  };
}
```

1.4 상태 관리 라이브러리 (Pinia)

Vue 3 공식 상태 관리 라이브러리인 Pinia 사용법을 익혀 어플리케이션 상태를 효율적으로 관리한다.

학습목표

- 상태 관리 필요성
- Pinia 개요
- Pinia 핵심 요소
- Store 개요 및 핵심요소
- State, Getters, Actions 개요
- Counter 실습
- Devtools 활용

1.4 상태 관리 라이브러리 (Pinia)

상태 관리 필요성

- 컴포넌트가 많아질수록 데이터 전달 경로가 복잡해짐
- 형제 컴포넌트 간에 상태 공유가 힘듦
- 전역에서 접근 가능한 단일 소스 필요성

1.4 상태 관리 라이브러리 (Pinia)

Pinia 개요

- 개념
  - Vue 3 공식 상태 관리 라이브러리
    https://pinia.vuejs.kr/

- 설치 및 등록

npm install pinia

```
# main.js
import { createApp } from 'vue'
import { createPinia } from 'pinia'
import App from './App.vue'

const app = createApp(App)
app.use(createPinia())
app.mount('#app')
```

1.4 상태 관리 라이브러리 (Pinia)

Pinia 장단점

- 장점
  - 가볍고 직관적
  - 개발 편의성 (DevTools)
  - 보일러플레이트 최소화
  - TypeScript 친화적
  - Composition API 및 Vite 쉬운 연동

- 단점
  - 레퍼런스 취약
  - 너무 많은 전역 상태 사용시 복잡성 증가
  - 브라우저 새로고침 시 상태 초기화

1.4 상태 관리 라이브러리 (Pinia)

Pinia 핵심 요소

- Store
- State
- Getters
- Actions

1.4 상태 관리 라이브러리 (Pinia)

Store 개요

- 개념
  - 어플리케이션 전역 상태를 관리하는 반응형 객체
  - 데이터 중앙 집중 관리소 역할
- 구성요소
  - state : 실제 데이터
  - getters : state 기반의 계산된 값 반환 (computed)
  - actions : state 변경 메서드
- 작성 방식
  - Options API 방식
  - Setup 방식 (Composition API 기반)

1.4 상태 관리 라이브러리 (Pinia)

Store 정의

- 문법

  import { defineStore } from 'pinia'

  export const variable_name = defineStore(store_id, setup function)

  - 파라미터

    store_id : store 식별하기 위한 고유한 이름

    setup function : state + getters + actions 구현 함수

  - variable_name

    defineStore 의 반환값을 저장할 변수

    권장 이름: "use"+store_id+"Store" 형식

    ex. export const **useCounterStore** = defineStore('**counter**', ()=>{})

1.4 상태 관리 라이브러리 (Pinia)

Store 사용

- 문법 예

```
import { useCounterStore } from '@/store/counter'
const store = useCounterStore()
```

- 원하는 만큼 Store 정의 가능
- state, getters, action에 정의된 모든 속성에 접근 가능
- state/getters는 구조 분해 할당 시 반응성 소멸

1.4 상태 관리 라이브러리 (Pinia)

State 개요

- 개념
  - 어플리케이션 전체에서 사용할 수 있는 변수 모음
  - 반응성 위해 ref, reactive 사용

- 특징
  - Composition API 문법 사용
    ex. ref, reactive, computed, watch
  - 전역 공유 가능
  - 원시값 / 배열 / 객체 / 중첩 모두 가능
  - 구조 분해 할당 사용시 반응성 소멸 (storeToRef 로 반응성 유지)

1.4 상태 관리 라이브러리 (Pinia)

Getters 개요

- 개념
  - State 로부터 계산된 읽기 전용 값 (파생값)
    ex. 장바구니 합계, 필터링 된 리스트
  - computed 방식 동일
- 특징
  - 의존한 값이 변하지 않으면 캐시 처리
  - 구조 분해 할당 사용시 반응성 소멸 (storeToRef 로 반응성 유지)
- 구현 가능/불가
  - 가능 : 순수 계산 로직, 다른 state/getters 참조, 정렬/슬라이싱
  - 불가 : 비동기/네트워크 요청 (action 에서 처리)
    부수효과(side effect)인 로깅/타이머/DOM 접근

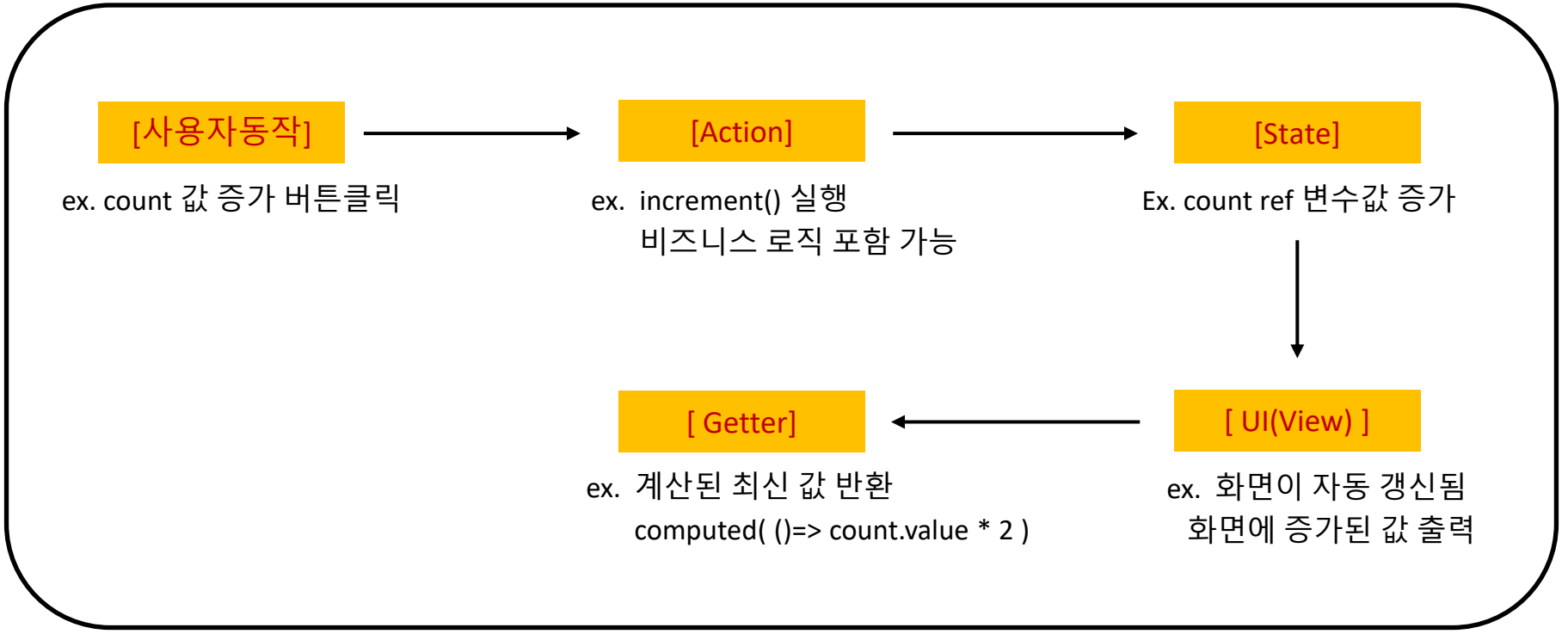
1.4 상태 관리 라이브러리 (Pinia)

Actions 개요

- 개념
  - 전역 상태(State) 변경 및 비즈니스 로직 처리 집합
  - 일반함수 방식 동일
- 특징
  - 동기/비동기 모두 가능
  - 부수효과(side effect) 가능
  - API 호출, 조건분기, 여러 store 협업 가능

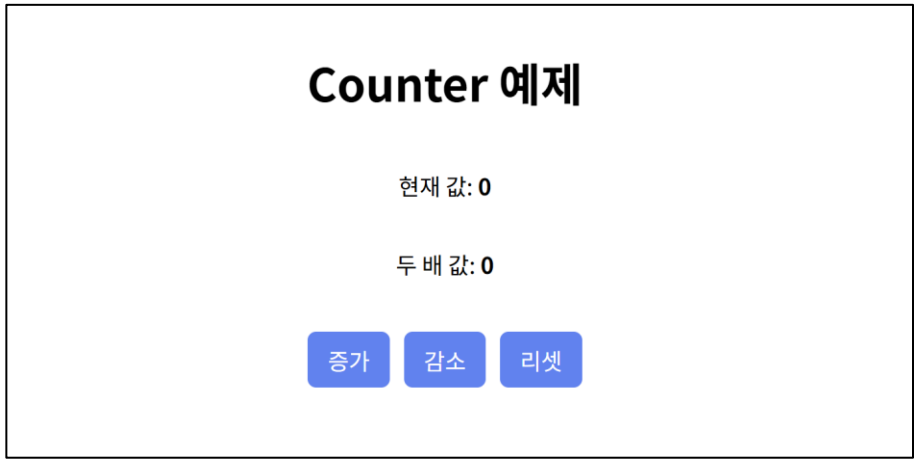
1.4 상태 관리 라이브러리 (Pinia)

Store 기본 흐름



1.4 상태 관리 라이브러리 (Pinia)

실습 (기본)



1.4 상태 관리 라이브러리 (Pinia)

실습 폴더 구조 (기본)

```
src
  main.js
  store/
      counter.js          #  Store 정의 (  State + Getters + Actions )
  App.vue                  #  Store 사용
```

1.4 상태 관리 라이브러리 (Pinia)

구현 코드(기본)

```
// main.js
import { createApp } from "vue";
import { createPinia } from "pinia";
import App from "./App.vue";

const app = createApp(App);
app.use(createPinia());
app.mount("#app");
```

```
// counter.js
import { defineStore } from "pinia";
import { ref, computed } from "vue";

// Setup 방식 Store
export const useCounterStore = defineStore("counter", () => {
  // 상태(state)
  const count = ref(0);
  // 계산된 값 (getter)
  const double = computed(() => count.value * 2);
  // 동작 (actions)
  function increment() { count.value++; }
  function decrement() { count.value--; }
  function reset() { count.value = 0;}

  return { count, double, increment, decrement, reset };
});
```

1.4 상태 관리 라이브러리 (Pinia)

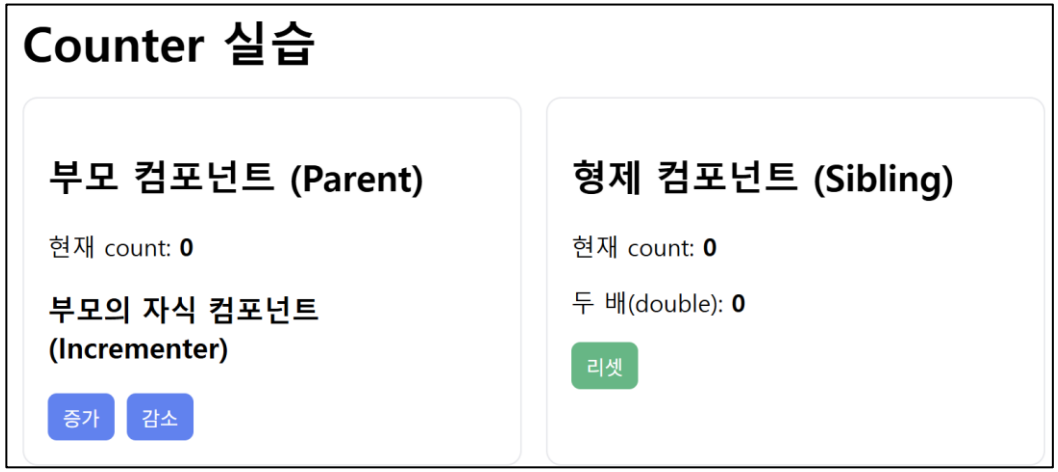
구현 코드(기본)

```vue
<!-- App.vue -->
<script setup>
import { storeToRefs } from "pinia";
import { useCounterStore } from "./stores/counter";

const counter = useCounterStore();

// state/getter를 반응형으로 가져오기
const { count, double } = storeToRefs(counter);

// actions 직접 가져오기
const { increment, decrement, reset } = counter;
</script>
```

```vue
<template>
  <main>
    <h1>Counter 예제</h1>
    <p>
      현재 값: <strong>{{ count }}</strong>
    </p>
    <p>
      두 배 값: <strong>{{ double }}</strong>
    </p>

    <div style="margin-top: 20px; display: flex; gap: 10px">
      <button @click="increment">증가</button>
      <button @click="decrement">감소</button>
      <button @click="reset">리셋</button>
    </div>
  </main>
</template>
```

1.4 상태 관리 라이브러리 (Pinia)

실습 (중첩 및 형제)



1.4 상태 관리 라이브러리 (Pinia)

실습 폴더 구조 (중첩 및 형제)

```
src
  main.js
  components/
     ChildIncrementer.vue      #  부모의 자식 ( 증가/감소 버튼 )
     CounterParent.vue         #  부모 ( count 값 출력 )
     CounterSibling.vue        #  부모와 형제 ( 현재 count, 두배 count, 리셋 버튼)
  store/
      counter.js               #  Store 정의 (  State + Getters + Actions )
  App.vue                      #  레이아웃
```

1.4 상태 관리 라이브러리 (Pinia)

구현 코드 (중첩 및 형제)

```
// main.js
import { createApp } from "vue";
import { createPinia } from "pinia";
import App from "./App.vue";

const app = createApp(App);
app.use(createPinia());
app.mount("#app");
```

```
// counter.js
import { defineStore } from "pinia";
import { ref, computed } from "vue";

// Setup 방식 Store
export const useCounterStore = defineStore("counter", () => {
  // 상태(state)
  const count = ref(0);
  // 계산된 값 (getter)
  const double = computed(() => count.value * 2);
  // 동작 (actions)
  function increment() { count.value++; }
  function decrement() { count.value--; }
  function reset() { count.value = 0;}

  return { count, double, increment, decrement, reset };
});
```

1.4 상태 관리 라이브러리 (Pinia)

구현 코드 (중첩 및 형제)

```vue
<!-- CounterParent.vue -->
<script setup>
import { storeToRefs } from "pinia";
import { useCounterStore } from "@/stores/counter";
import ChildIncrementer from "./ChildIncrementer.vue";

const counter = useCounterStore();
const { count } = storeToRefs(counter);
</script>

<template>
  <section class="card">
    <h2>부모 컴포넌트 (Parent)</h2>
    <p>
      현재 count: <strong>{{ count }}</strong>
    </p>
    <!-- 중첩 자식: 여기서 카운트 올리고 내림 -->
    <ChildIncrementer />
  </section>
</template>
```

```vue
<!-- ChildIncrementer.vue -->
<script setup>
import { useCounterStore } from "@/stores/counter";

const counter = useCounterStore();
// actions는 그대로 구조분해 가능
const { increment, decrement } = counter;
</script>

<template>
  <h3>부모의 자식 컴포넌트(Incrementer)</h3>
  <div>
    <button @click="increment">증가</button>
    <button @click="decrement">감소</button>
  </div>
</template>
```

1.4 상태 관리 라이브러리 (Pinia)

구현 코드 (중첩 및 형제)

```vue
<!-- CounterSibling.vue -->
<script setup>
import { storeToRefs } from "pinia";
import { useCounterStore } from "@/stores/counter";

const counter = useCounterStore();
const { count, double } = storeToRefs(counter);
const { reset } = counter;
</script>
<template>
  <section class="card">
    <h2>형제 컴포넌트 (Sibling)</h2>
    <p>
      현재 count: <strong>{{ count }}</strong>
    </p>
    <p>
      두 배(double): <strong>{{ double }}</strong>
    </p>
    <button @click="reset">리셋</button>
  </section>
</template>
```

```vue
<!-- App.vue -->
<script setup>
import CounterParent from "./components/CounterParent.vue";
import CounterSibling from "./components/CounterSibling.vue";
</script>

<template>
  <main class="wrap">
    <h1>Counter 실습</h1>
    <div class="grid">
      <CounterParent />
      <CounterSibling />
    </div>
  </main>
</template>
```

1.4 상태 관리 라이브러리 (Pinia)

pinia-plug-persistedstate 개요

- 개념
  - Pinia 스토어 상태 변경 시, 브라우저 저장소에 자동 저장 및 복원해주는 플러그인
- 등장배경
  - 세션유지
  - 비로그인 UX 개선
  - 간단한 오프라인 시나리오
- 특징
  - 최소 설정(persist: true)로 기본 동작
  - 옵션으로 저장소, 저장 키 이름, 저장 대상 경로 등 커스터마이징 가능

1.4 상태 관리 라이브러리 (Pinia)

pinia-plug-persistedstate 설치 및 등록

- 설치

  npm install pinia-plugin-persistedstate

- 등록

```
// main.js
import { createApp } from "vue";
import { createPinia } from "pinia";
import piniaPluginPersistedstate from "pinia-plugin-persistedstate";
import App from "./App.vue";

const app = createApp(App);
const pinia = createPinia();
pinia.use(piniaPluginPersistedstate);
app.use(pinia);
app.mount("#app");
```

1.4 상태 관리 라이브러리 (Pinia)

pinia-plug-persistedstate 활성화

- 활성화 방법

```
export const useCounterStore = defineStore("counter", () => {
   // 상태(state)
   // 계산된 값 (getters)
   // 동작 (actions)
   return { state, getters, actions };
 },
 { persist: true }
);
```

1.4 상태 관리 라이브러리 (Pinia)

Devtools 설치

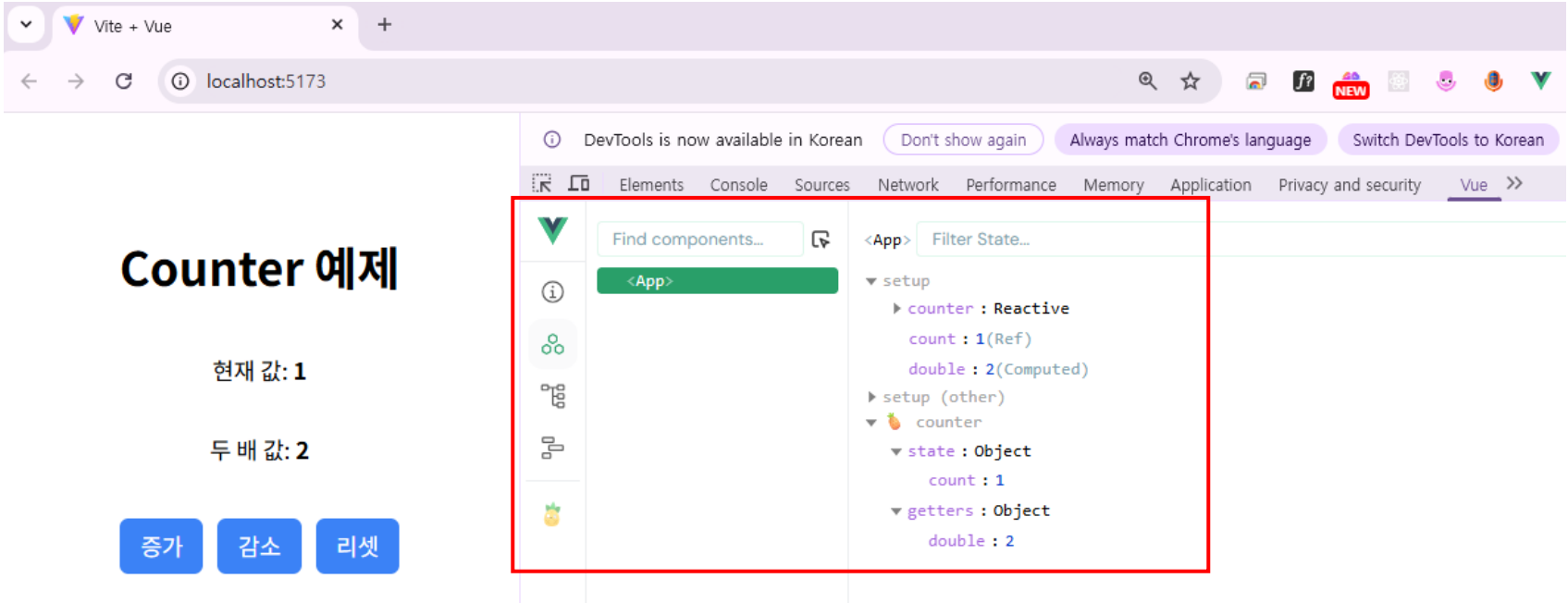
- 개념
  - Vue 공식 개발자 도구 (Chrome 확장 프로그램)
  - Pinia 스토어의 상태(State), getter, action 호출 기록을 시각적으로 추적
  - Store 단위로 디버깅 가능

- 설치

  Chrome 브라우저에서 Vue.js devtools 확장 프로그램 설치

1.4 상태 관리 라이브러리 (Pinia)

Devtools 디버깅



1.5 정리 및 실습

요약맵

- Vue Router 4 기초
- 중첩 라우트 및 내비게이션 가드
- REST API 개요 및 Firebase 활용
- Axios 라이브러리
- Pinia 활용한 전역 상태 관리
- 종합실습

2. Advanced 기능 및 최적화

- Vue 3 고급 기능
- 성능 최적화 기법
- 테스팅
- 빌드 최적화와 배포
- TypeScript와 Vue 3
- WRAP-UP 및 평가

SK Inc. AX

2.1 Vue 3 고급 기능

Vue 3의 Teleport와 Suspense를 활용해 UI 위치 제어와 비동기 로딩을 효율적으로 관리한다.

학습목표

- Teleport 개요
- Suspense 개요

2.1 Vue 3 고급 기능

Teleport 개요

- 개념
  - 컴포넌트의 템플릿 일부를 해당 컴포넌트의 DOM 계층 외부에 존재하는
  - 특정 DOM 노드에 랜더링 해주는 내장 컴포넌트
- 등장배경
  - 기존 Vue 2에서는 컴포넌트 DOM은 반드시 부모 DOM 하위에만 위치
  - UI 요소(모달, 알림창 등)는 종종 부모와 상관없이 body 같은 전역 영역에 위치 필요
- 특징
  - 독립적 위치 랜더링
  - 다양한 target 선택 지원 (CSS 또는 DOM 노드)
  - 동적 제어 가능 (비활성화 가능)

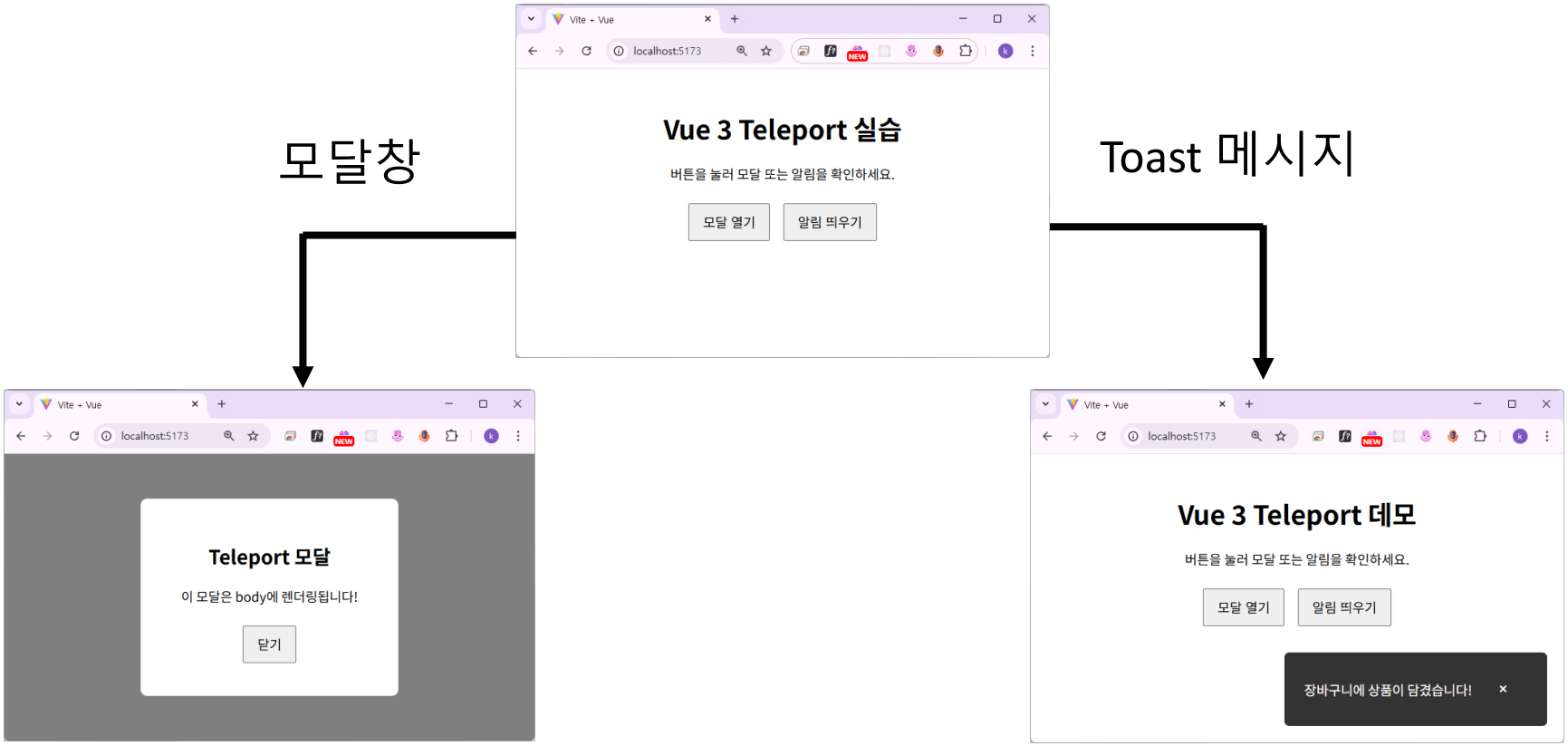
2.1 Vue 3 고급 기능

Teleport 장단점 및 활용

- 장점
  - 부모 스타일/레이아웃에 영향을 안 받음
  - 전역 UI 관리 간편화
  - 코드와 DOM 구조의 분리 (유지보수)
- 단점
  - 너무 남발하면 DOM 구조 파악이 어려워짐
  - CSS 범위(scope) 고려 필요
- 활용 사례
  - 모달 창: body 최상단에 띄우기
  - 알림/Toast 메시지: 항상 화면 특정 영역에 출력
  - 전역 레이어 UI: 앱 전역에 겹쳐지는 UI 관리

2.1 Vue 3 고급 기능

실습



2.1 Vue 3 고급 기능

실습 폴더 구조

```
src
  main.js
  teleport/
    Modal.vue       # 모달 창
    Toast.vue       # Toast 메시지
  App.vue
index.html          # Toast 표시될 영역 지정
```

2.1 Vue 3 고급 기능

구현 코드

```html
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + Vue</title>
  </head>
  <body>
    <div id="app"></div>
    <!-- Teleport된 Toast가 표시될 전용 영역 -->
    <div id="toast-area"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

```javascript
// main.js
import { createApp } from "vue";
import App from "./App.vue";

createApp(App).mount("#app");
```

2.1 Vue 3 고급 기능

구현 코드

```
<!-- Toast.vue -->
<template>
  <div class="toast">
    {{ message }}
    <button class="close"
@click="$emit('close')">×</button>
  </div>
</template>
<script setup>
defineProps({
  message: String,
});
</script>
<style scoped>
.toast .close {
  background: transparent;
  border: none;
  color: white;
  font-size: 1.2rem;
  cursor: pointer;
}
```

```
.toast {
  position: fixed; bottom: 20px; right: 20px;
  background: #333; color: white;
  padding: 1rem 1.5rem;
  border-radius: 6px;
  display: flex;
  align-items: center;
  gap: 0.5rem; animation: fadeIn 0.3s ease;
}
@keyframes fadeIn {
  from {
    opacity: 0;
    transform: translateY(10px);
  }
  to {
    opacity: 1;
    transform: translateY(0);
  }
}
</style>
```

2.1 Vue 3 고급 기능

구현 코드

```
<!-- Modal.vue -->
<template>
  <div class="overlay">
    <div class="modal">
      <h2>Teleport 모달</h2>
      <p>이 모달은 body에 렌더링됩니다!</p>
      <button @click="$emit('close')">닫기</button>
    </div>
  </div>
</template>

<script setup></script>
```

```
<style>
.overlay {
  position: fixed;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(0, 0, 0, 0.5);
  display: flex;
  align-items: center;
  justify-content: center;
}
.modal {
  background: white;
  padding: 2rem;
  border-radius: 8px;
  min-width: 250px;
  text-align: center;
}
</style>
```

2.1 Vue 3 고급 기능

구현 코드

```vue
<!-- App.vue -->
<template>
  <div class="container">
    <h1>Vue 3 Teleport 실습</h1>
    <p>버튼을 눌러 모달 또는 알림을 확인하세요.</p>

    <!-- 모달 열기 버튼 -->
    <button @click="showModal = true">모달 열기</button>
    <!-- Toast 띄우기 버튼 -->
    <button @click="showToast = true">알림 띄우기</button>
    <!-- Teleport: 모달을 body에 렌더링 -->
    <Teleport to="body">
      <Modal v-if="showModal" @close="showModal = false" />
    </Teleport>
    <!-- Teleport: 토스트를 #toast-area에 렌더링 -->
    <Teleport to="#toast-area">
      <Toast  v-if="showToast“ message="장바구니에 상품이 담겼습니다!"@close="showToast=false" />
    </Teleport>
  </div>
</template>
```

```vue
<script setup>
import { ref } from "vue";
import Modal from "./teleport/Modal.vue";
import Toast from "./teleport/Toast.vue";

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

2.1 Vue 3 고급 기능

Suspense 개요

- 개념
  - Vue 3에서 새로 도입된 비동기 컴포넌트 처리 기능
  - 컴포넌트가 데이터 로딩 및 비동기 연산 수행 시
  - 준비가 끝날 때까지 대체 UI(Fallback) 제공

- 특징
  - Fallback UI 제공
  - defalut/ fallback 슬롯 제공
    #default: 실제 컴포넌트
    #fallback: 로딩 시 보여줄 대체 UI
  - 중첩 Suspense 지원

2.1 Vue 3 고급 기능

Suspense 문법

```
<template>
  <Suspense>
    <!-- 로딩이 끝나면 보여줄 영역 -->
    <template #default>
      <AsyncComp />
    </template>

    <!-- 로딩 중일 때 보여줄 영역 -->
    <template #fallback>
      <div>로딩 중...</div>
    </template>
  </Suspense>
</template>
```

2.1 Vue 3 고급 기능

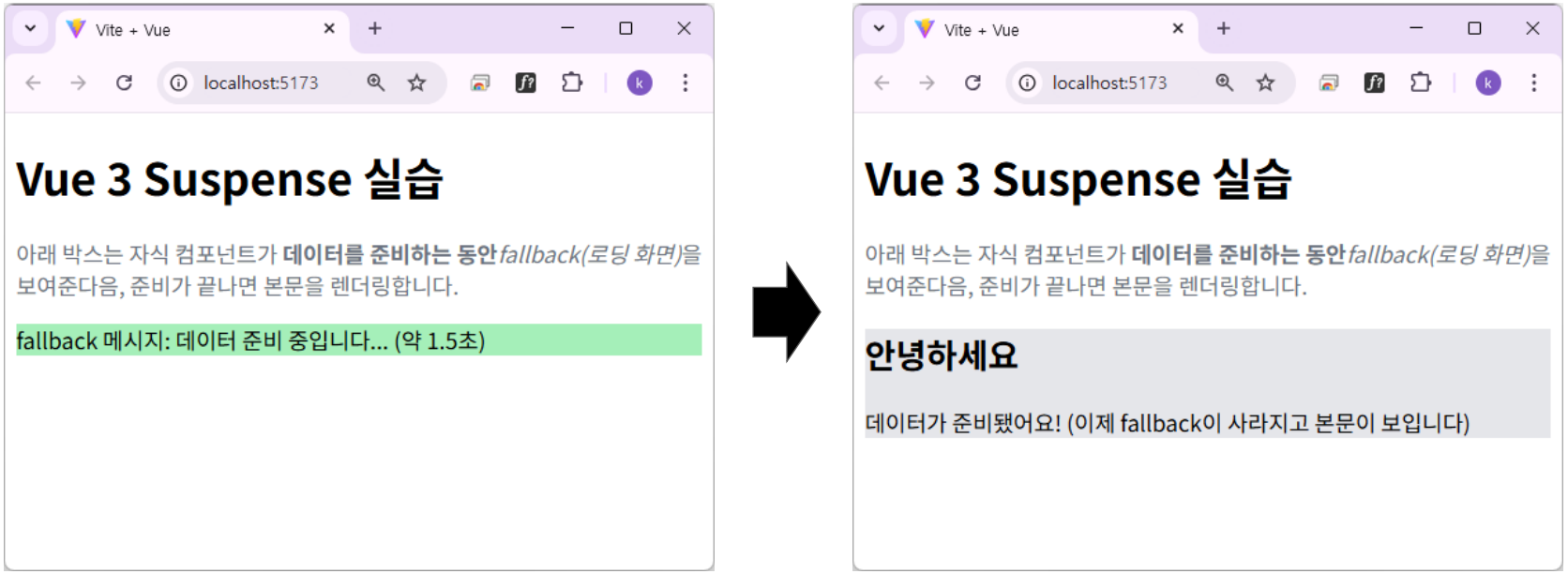
Suspense 장점 및 주의사항

- 장점
  - 사용자 경험(UX) 개선
  - 직관적이로 단순한 문법
  - 다른 비동기 처리 방식 (watch, onMounted)보다 단순

- 주의사항
  - 너무 많은 Suspense 사용시 코드 복잡도 증가
  - Fallback UI도 UX 고려해서 가볍게 구성
  - Vue Router와 함께 사용 시 라우트별 Suspense 설계 고려
  - 모든 자식이 준비될 때까지 fallback 유지 (UX 지연 발생)

2.1 Vue 3 고급 기능

실습



2.1 Vue 3 고급 기능

실습 폴더 구조

```
src
  main.js
  components/
    HelloAsync.vue      # 비동기 코드
  App.vue
```

2.1 Vue 3 고급 기능

구현 코드

```
<!-- App.vue -->
<script setup>
// Suspense 안에서 보여줄 자식 컴포넌트
import HelloAsync from "./components/HelloAsync.vue";
</script>

<template>
  <main>
    <h1>Vue 3 Suspense 실습</h1>
    <p class="muted">
      아래 박스는 자식 컴포넌트가 <strong>데이터를 준비하는 동안
</strong>
      <em>fallback(로딩 화면)</em> 을 보여준다음,
  준비가 끝나면 본문을  렌더링합니다.
    </p>
```

```
    <div class="card" style="margin-top: 12px">
      <!-- 핵심: Suspense -->
      <Suspense>
        <!-- 준비가 끝나면 이 슬롯이 그려짐 -->
        <template #default>
          <HelloAsync />
        </template>

        <!-- 준비되는 동안 이 슬롯이 먼저 보임 -->
        <template #fallback>
          <p class="ing">
            fallback 메시지: 데이터 준비 중입니다... (약 1.5초)
          </p>
        </template>
      </Suspense>
    </div>
  </main>
</template>
```

1.5초 후

2.1 Vue 3 고급 기능

구현 코드

```
<!-- HelloAsync.vue -->
<script setup>
import { ref } from "vue";

// 화면에 보여줄 메시지
const message = ref("");

// 1) 지연을 주는 작은 유틸 함수 (가짜 로딩)
const sleep=
(ms)=>new Promise((resolve)=>setTimeout(resolve,ms));

// 2) top-level await: 컴포넌트가 다 준비될 때까지 기다림
await sleep(1500);

// 3) 준비가 끝났으니 실제 데이터가 왔다고 가정하고 메시지 세팅
message.value =
  " 데이터가 준비됐어요! (이제 fallback이 사라지고 본문이 보입니다)";
</script>

<template>
  <div>
    <h2>안녕하세요</h2>
    <p>{{ message }}</p>
  </div>
</template>

<style scoped>
div {
  background-color: #e5e6ea;
}
</style>
```

2.2 성능 최적화 기법

Vue 3의 v-memo와 비동기 컴포넌트를 활용해 랜더링 성능을 최적화하는 방법을 학습한다.

학습목표

- v-memo 개요
- 비동기 컴포넌트
- defineAsyncComponent + import
- 지연 로딩

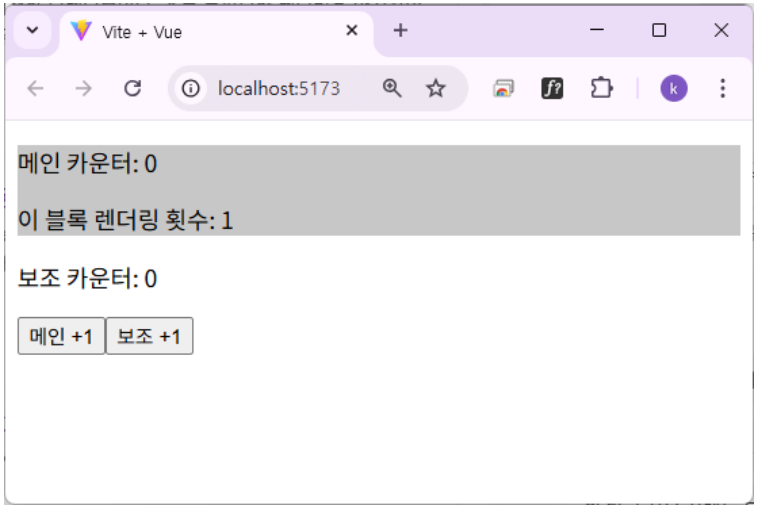
2.2 성능 최적화 기법

v-memo 개요

- 개념
  - Vue 3.2에서 도입된 랜더링 최적화 디렉티브
  - 특정 종속값(props/state)이 변경될 때만 해당 블록을 다시 랜더링
- 문법
  - v-memo=[종속값,…]
  - 종속값이 변경될 때만 다시 랜더링됨
  - 값이 변경 안되면 이전 랜더링 결과 재사용
- 특징
  - 조건부 랜더링 최적화
  - 얕은 비교(Shallow Compare) 적용
  - 요소와 컴포넌트 모두 사용 가능

2.2 성능 최적화 기법

실습 (기본)



2.2 성능 최적화 기법

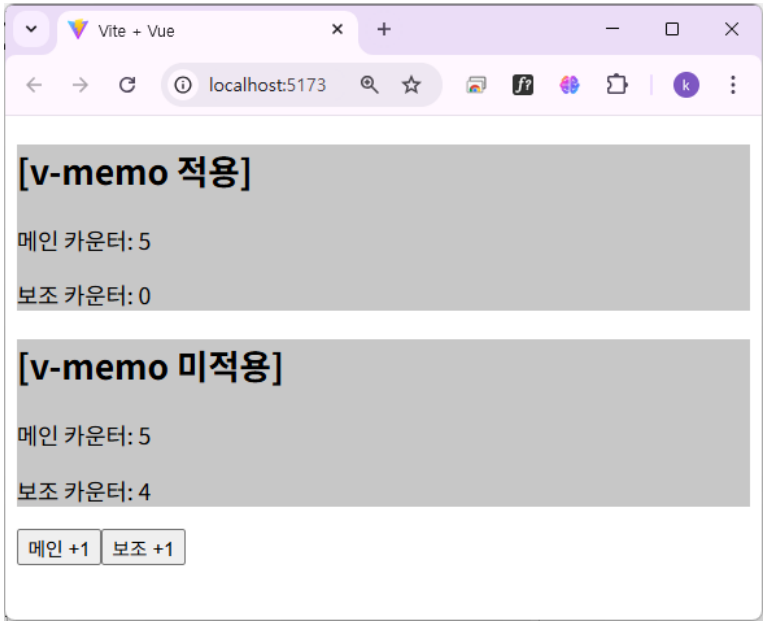
구현 코드 (기본)

```vue
<!-- App.vue -->
<template>
  <!-- v-memo의 의도: counter가 변할 때만 블록을 다시 그리는지 확인
-->
  <div v-memo="[counter]">
    {{ logRender() }}
    <p>메인 카운터: {{ counter }}</p>
    <p>이 블록 렌더링 횟수: {{ renderCount }}</p>
  </div>

  <p>보조 카운터: {{ subCounter }}</p>
  <button @click="counter++">메인 +1</button>
  <button @click="subCounter++">보조 +1</button>
</template>
```

```vue
<script setup>
import { ref, watch, onUpdated } from "vue";

const counter = ref(0);
const subCounter = ref(0);
const renderCount = ref(1); // 초기 마운트 1회 포함

// counter가 변할 때만 카운트 증가
watch(counter, () => {
  renderCount.value++;
});
// 컴포넌트가 다시 렌더링(업데이트) 될 때마다 로그 출력
onUpdated(() => {
  console.log("화면 렌더링 발생");
});
// div 블록이 재랜더링 될 때마다 로그 출력
function logRender() {
  console.log("블록 재랜더링 ");
}
</script>
```

2.2 성능 최적화 기법

실습 (props)



2.2 성능 최적화 기법

실습 폴더 구조 (props)

```
src
  main.js
  components/
    Memo.vue          #  v-memo 추가된 자식 컴포넌트
    NoMemo.vue        #  v-memo 없는 자식 컴포넌트
  App.vue
```

2.2 성능 최적화 기법

구현 코드 (props)

```
<!-- App.vue -->
<template>
<!-- props 전달 -->
  <MemoChild v-memo="[counter]" :counter="counter" :subCounter="subCounter" />
  <NoMemoChild                                    :counter="counter" :subCounter="subCounter" />

  <button @click="counter++">메인 +1</button>
  <button @click="subCounter++">보조 +1</button>
</template>

<script setup>
import { ref } from "vue";
import MemoChild from "./components/MemoChild.vue";
import NoMemoChild from "./components/NoMemoChild.vue";

const counter = ref(0);
const subCounter = ref(0);
</script>
```

2.2 성능 최적화 기법

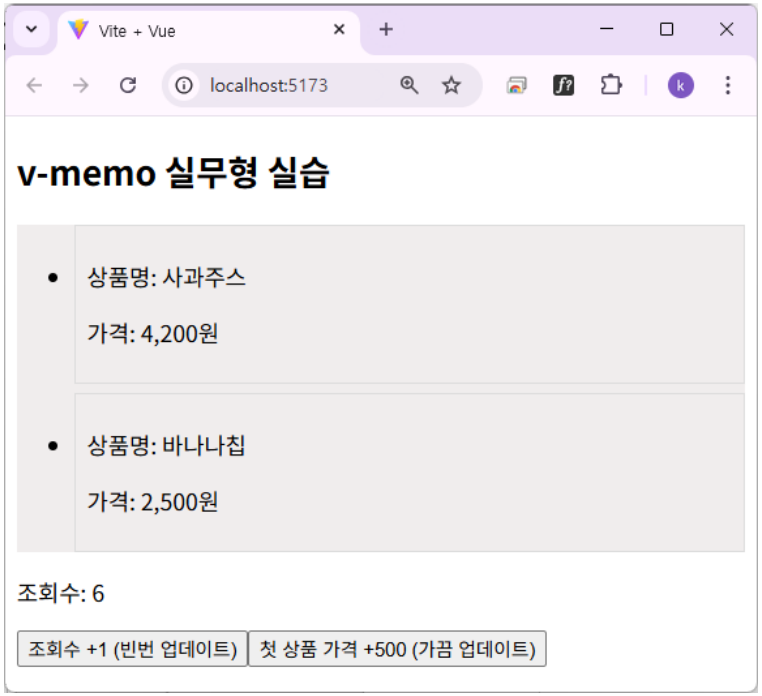
구현 코드 (props)

```vue
<!-- MemoChild.vue -->
<template>
  <div>
    <h2>[v-memo 적용]</h2>
    <p>메인 카운터: {{ counter }}</p>
    <p>보조 카운터: {{ subCounter }}</p>
  </div>
</template>

<script setup>
import { onUpdated } from "vue";
defineProps({ counter: Number, subCounter: Number });

onUpdated(() => {
  console.log("MemoChild 렌더링 발생");
});
</script>
```

```vue
<!-- NoMemoChild.vue -->
<template>
  <div>
    <h2>[v-memo 미적용]</h2>
    <p>메인 카운터: {{ counter }}</p>
    <p>보조 카운터: {{ subCounter }}</p>
  </div>
</template>

<script setup>
import { onUpdated } from "vue";
defineProps({ counter: Number, subCounter: Number });

onUpdated(() => {
  console.log("NoMemoChild 렌더링 발생");
});
</script>
```

2.2 성능 최적화 기법

실습 (v-for)



2.2 성능 최적화 기법

구현 코드 (v-for)

price가 변경된 경우에만 재랜더링 됨

```vue
<!-- App.vue -->
<template>
  <h2>v-memo 실무형 실습</h2>
  <ul>
    <li v-for="p in products" :key="p.id" v-memo="[p.price]">
      {{ listRender() }}
      <div>
        <p>상품명: {{ p.name }}</p>
        <p>가격: {{ p.price.toLocaleString() }}원</p>
      </div>
    </li>
  </ul>
  <p>조회수: {{ viewCount }}</p>
  <button @click="viewCount++">
     조회수 +1 (빈번 업데이트)
  </button>
  <button @click="updatePrice">
    첫 상품 가격 +500 (가끔 업데이트)
  </button>
</template>
```

```vue
<script setup>
import { ref } from "vue";

const viewCount = ref(0);
const products = ref([
  { id: 1, name: "사과주스", price: 3200 },
  { id: 2, name: "바나나칩", price: 2500 },
]);

// 불변 업데이트 (새 객체 교체)
function updatePrice() {
  products.value = products.value.map((p) =>
    p.id === 1 ? { ...p, price: p.price + 500 } : p
  );
}

function listRender() {
  console.log("상품목록 재랜더링");
}
</script>
```

2.2 성능 최적화 기법

비동기 컴포넌트 (Async components)

- 개념
  - 화면에 필요한 시점에만 컴포넌트를 동적으로 불러오는 방식
  - 지연 로딩(Lazy Loading) 방식

- 문법

```
defineAsyncComponent( {
  loader: ()=> import (컴포넌트),     # 실제 보여줄 컴포넌트
  loadingComponent: 컴포넌트,        # 로딩중 보여줄 컴포넌트
  errorComponent: 컴포넌트,          # 실패 시 보여줄 컴포넌트
  delay: (ms),                       # 로딩 컴포넌트 보여주기 전 지연 시간
  timeout: (ms)                      # 타입아웃
})
```

2.2 성능 최적화 기법

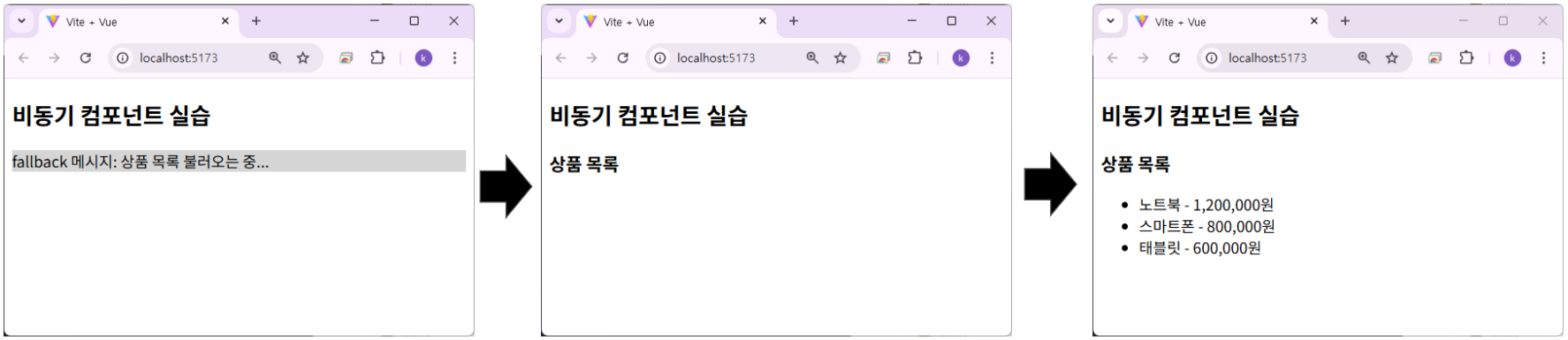
비동기 컴포넌트 (Async components)

- 특징
  - 어플리케이션 초기 로딩 부담 감소
  - 로딩중 / 오류 상태 처리 옵션 지원
  - <Suspense> 활용 시 최적의 UI 흐름 제어
  - 코드 분할(Code Splitting)로 유지보수와 성능 유리

- 활용 사례
  - 관리자 페이지(admin)처럼 잘 사용되지 않는 화면
  - 대형 모듈(chart, map, editor 등)
  - SPA 라우트 전환 시 특정 페이지 지연로딩

2.2 성능 최적화 기법

실습 (Async)



2.2 성능 최적화 기법

실습 폴더 구조 (Async)

```
src
  main.js
  components/
    Error.vue          # 오류 발생시 보여줄 컴포넌트
    Loading.vue        # 로딩 시 보여줄 컴포넌트
    ProductList.vue    # 실제 보여줄 컴포넌트
  App.vue
```

2.2 성능 최적화 기법

구현 코드 (Async)

```vue
<!-- App.vue -->
<template>
  <main>
    <h2>비동기 컴포넌트 실습</h2>
    <!-- Suspense를 이용한 로딩 처리 -->
    <Suspense>
      <template #default>
        <AsyncProductList />
      </template>
      <template #fallback>
        <div class="fallback">
              fallback 메시지: 상품 목록 불러오는 중...
        </div>
      </template>
    </Suspense>
  </main>
</template>
```

```vue
<script setup>
import { defineAsyncComponent } from "vue";

// 로딩 & 에러 컴포넌트
import Loading from "./components/Loading.vue";
import ErrorComp from "./components/Error.vue";
const AsyncProductList = defineAsyncComponent({[
  // loader: () => import("./components/ProductList.vue"),
  loader: () => {
    // 실습용:  2초 지연 후 컴포넌트 import
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve(import("./components/ProductList.vue"));
      }, 2000); // 2초 지연
    });
  },
  loadingComponent: Loading,
  errorComponent: ErrorComp,
  delay: 300,
  timeout: 5000,
});
</script>
```

2.2 성능 최적화 기법

구현 코드 (Async)

```vue
<!-- ProductList.vue -->
<template>
  <section>
    <h3>상품 목록</h3>
    <ul>
      <li v-for="p in products" :key="p.id">
        {{ p.name }} - {{ p.price.toLocaleString() }}원
      </li>
    </ul>
  </section>
</template>
```

```vue
<script setup>
import { ref, onMounted } from "vue";

// 상태 정의
const products = ref([]);

// 실습용:  마운트 시 API 요청 가정
onMounted(() => {
  setTimeout(() => {
    products.value = [
      { id: 1, name: "노트북", price: 1200000 },
      { id: 2, name: "스마트폰", price: 800000 },
      { id: 3, name: "태블릿", price: 600000 },
    ];
  }, 1500);
});
</script>
```

2.2 성능 최적화 기법

구현 코드 (Async)

```
<!-- Loading.vue -->
<template>
  <h1>데이터를 불러오는 중입니다.</h1>
</template>
```

```
<!-- Error.vue -->
<template>
  <div style="color: red">컴포넌트 로드에 실패했습니다.</div>
</templat□□□e>
```

2.3 테스팅

Vitest 및 추가 패키지를 활용하여 실제 컴포넌트 동작을 테스트하고 품질을 보장받는 방법을 학습한다.

학습목표

- 테스트 개요
- Vitest 프레임워크
- 추가 패키지 개요

  @vue/test-utils
  @testing-library/vue
  @testing-library/jest-dom
  @testing-library/user-event
  jsdom

2.3 테스팅

테스트 종류

- 단위테스트(unit test)
  - 가장 자주 사용되는 테스트
  - 하나의 단위(함수, 컴포넌트)로 테스트 케이스를 작성
  - 종속성이나 다른 요소와의 복잡한 상호 작용이 없는 테스트
  - 이를 위해 연관된 부분에 대해서는 가짜 객체인 Mock 사용
- 통합테스트(integration test)
  - 다른 요소와의 종속성을 함께 처리 (복잡성 증가)
- 엔드투엔드 테스트(E2E, End to End test)
  - 사용자와 앱의 상호 작용에 대한 테스트
  - 웹의 경우 실제 브라우저를 이용해서 전체 앱을 테스팅

2.3 테스팅

Vitest

- 개념
  - Vite 기반의 프로젝트에서 사용되는 단위 테스트 메인 프레임워크

  https://vitest.dev/
  - Vue, React, TS/JS 환경에서 빠른 실행 속도 제공

- 특징
  - Vite 생태계와 동일한 빌드 환경 공유 (설정 단순화)
  - 빠른 테스트 실행
  - 모듈 mocking, 스냅샷, 병렬 실행 등 지원
  - Vue Test Utils, Testing Library 등과의 자연스러운 조합

2.3 테스팅

추가 패키지

패키지	기능	공식 사이트
@vue/test-utils	VTU라고 부르며 Vue의 Component 테스트 지원	https://test-utils.vuejs.org/
@testing-library/vue	DOM을 쿼리하고 실제 사용자처럼 클릭/입력 지원	https://testing-library.com/
@testing-library/jest-dom	DOM 테스트를 더 직관적이고 간단하게 만들어주는 매처(matcher) 확장 지원	https://testing-library.com/docs/ecosystem-jest-dom/
@testing-library/user-event	사용자가 실제 브라우저에서 클릭,입력,드래그 같은 행동을 하는 것처럼 시뮬레이션 지원	https://testing-library.com/docs/user-event/intro/
jsdom	Vitest가 DOM 테스트 수행 시 테스트 환경 지원	https://github.com/jsdom/jsdom

2.3 테스팅

Vitest 환경 설정

- 설치

  npm install --save-dev vitest @vue/test-utils @testing-library/vue jsdom @testing-library/jest-dom @testing-library/user-event

- vitest.setup.js 파일 생성

```
import { expect } from 'vitest'
import * as matchers from '@testing-library/jest-dom/matchers'
expect.extend(matchers)
```

2.3 테스팅

Vitest 환경 설정

- vite.config.js 파일 수정

```
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

export default defineConfig({
  plugins: [vue()],
  test: {
    environment: 'jsdom',          // 테스트 과정에서 사용하게 될 dom 시뮬레이터
    globals: true,                 // describe/it/expect 전역 사용
    setupFiles: './vitest.setup.js',
  },
})
```

2.3 테스팅

Vitest 환경 설정

- package.json 파일 수정

```
"scripts": {
   "dev": "vite",
   "build": "vite build",
   "preview": "vite preview",
   "test": "vitest",
   "test:watch": "vitest --watch"
 }
```

2.3 테스팅

Vitest 프레임워크 기능

- Runner
  테스트를 그룹/실행 ex. describe / test / it
- Assertion
  결과가 맞는지 검증 ex. expect(actual).toXXX(value)
- 비동기
  비동기 처리 ex. test + async / resolves / rejects
- Hooks
  공통 준비/정리 ex. beforeEach / afterEach / beforeAll / afterAll
- Mock/Spy
  가짜 함수/감시 ex. vi.fn / vi.spyOn / vi.mock
- Timer
  시간 제어 ex. vi.useFakeTimers / vi.advanceTimersByTime

2.3 테스팅

실습 폴더 구조 (Vitest)

```
src/
  main.js
  tests/
    vitest1.runner.spec.js
    vitest2.assertion.spec.js
    vitest3.async.spec.js
    vitest4.hooks.spec.js
    vitest5.mock.spec.js
    vitest6.timer.spec.js
  App.vue
```

2.3 테스팅

Runner

- 개념
  테스트 실행 및 그룹화

분류	메서드	문법	설명
그룹	describe	describe(name, fn)	테스트를 논리적으로 묶음
케이스	test	test(name, fn)	개별 테스트
케이스	it	it(name, fn)	개별 테스트, test 별칭

2.3 테스팅

Assertion

- 개념
  기대하는 값 검증
  expect(actual).toXXX(value) 형식

분류	메서드	문법	설명
값	toBe	expect(2+2).toBe(4)	원시값 동일
값	toEqual	expect({a:1}).toEqual({a:1})	객체/배열 내용 동일
예외	toThrow	expect(fn).toThrow()	함수가 예외 발생되는지
부분	stringContaining	expect.stringContaining(text)	문자열 포함 여부
부분	arrayContaining	expect.arrayContaining([item])	배열 포함 여부

2.3 테스팅

구현 코드 (Runner)

```
// vitest1.runner.spec.js
import { it, test, expect } from "vitest";

// 1. test/it 개별 테스트
it("덧셈: 1 + 2 = 3", () => {
  expect(1 + 2).toBe(3);
});

test("뺄셈: 5 - 2 = 3", () => {
  expect(5 - 2).toBe(3);
});
```

```
//2. describe 그룹화
describe("Math 연산", () => {
  it("덧셈", () => {
    expect(1 + 2).toBe(3);
  });

  test("뺄셈", () => {
    expect(5 - 2).toBe(3);
  });
});
```

2.3 테스팅

구현 코드 (Assertion)

```
// vitest2.assertion.spec.js
import { it, test, expect } from "vitest";

describe("기본 어서션", () => {
  it("toBe: 원시값 동일(===)", () => {
    expect(2 + 2).toBe(4);
  });

  test("toEqual: 객체/배열 구조 동일", () => {
    const user = { id: 1, name: "Alice" };
    expect(user).toEqual({ id: 1, name: "Alice" });
  });
```

```
  it("toThrow: 함수가 에러를 던지는지", () => {
    const boom = () => {
      throw new Error("boom");
    };
    expect(boom).toThrow();
  });

  it("부분 포함 매처", () => {
    expect("Vitest").toEqual(expect.stringContaining("Vitest"));
    expect([1, 2, 3]).toEqual(expect.arrayContaining([2]));
  });
});
```

2.3 테스팅

비동기(Async)

- 개념

  비동기 처리 성공 및 실패 검증

분류	메서드	문법	설명
비동기	test/it + async	it(name, async fn)	함수내에서 await 비동기 처리
비동기	resolves	await expect(promise).resolves.toBe(x)	성공 결과 검증
비동기	rejects	await expect(promise).rejects.toThrow()	실패(예외) 검증

2.3 테스팅

구현 코드 (Async)

```
// vitest3.async.spec.js;
import { it, test, expect } from "vitest";

// 서버 연동 가정
const fetchUser = (ok = true) =>
  new Promise((res, rej) =>
    setTimeout(
      () => (ok ? res({ id: 1, name: "Alice" })
                :rej(new Error("fail"))),
      50
    )
  );
```

```
describe("비동기 테스트", () => {
  it("await 사용", async () => {
    const data = await fetchUser();
    expect(data).toEqual({ id: 1, name: "Alice" });
  });

  it("resolves / rejects", async () => {
    await expect(fetchUser(true)).resolves
                                 .toEqual({ id: 1, name: "Alice" });
    await expect(fetchUser(false)).rejects
                                  .toThrow("fail");
  });
});
```

2.3 테스팅

Hooks

- 개념
  공통 초기화/정리

분류	메서드	문법	설명
hooks	beforeAll	beforeAll(fn)	전체 테스트 전 **한 번** 실행
hooks	afterAll	afterAll(fn)	전체 테스트 후 **한 번** 실행
hooks	beforeEach	beforeEach(fn)	각 테스트 **직전**마다 실행
hooks	afterEach	afterEach(fn)	각 테스트 **직후**마다 실행

2.3 테스팅

구현 코드 (Hooks)

```
// vitest4.hooks.spec.js
import { describe, it,expect,
  beforeEach,afterEach,beforeAll,afterAll,
} from "vitest";

describe("Math 연산", () => {
  let n = 0;
  beforeEach(() => {
    console.log(`매 테스트 전 실행- ${n++}`);
  });
  afterEach(() => {
    console.log(`매 테스트 후 실행- ${n++}`);
  });
  beforeAll(() => {
    console.log(`전체 테스트 전 한번 실행- ${n++}`);
  });
  afterAll(() => {
    console.log(`전체 테스트 후 한번 실행- ${n++}`);
  });
  it("덧셈", () => {
    expect(1 + 2).toBe(3);
  });
  test("뺄셈", () => {
    expect(5 - 2).toBe(3);
  });
});
```

2.3 테스팅

Mock / Spy

- 개념
  가짜 함수 / 감시

분류	메서드	문법	설명
Mock	fn	vi.fn()	가짜 함수 생성/호출 검증
Spy	spyOn	vi.spyOn(obj, 'method')	기존 메서드 감시/치환
모듈mock	mock	vi.mock('module', obj)	모듈 전체 mock
정리	clearAllMocks	vi.clearAllMocks()	호출 기록 초기화
정리	resetAllMocks	vi.resetAllMocks()	mock 상태 초기화

2.3 테스팅

구현 코드 (Mock/Spy)

```
// vitest5.mock.spec.js
import { describe, expect, vi } from "vitest";

describe("Mock", () => {
  it("vi.fn: 가짜 함수 호출/인자 추적", () => {
    const mockFn = vi.fn().mockReturnValue(42);
    const result = mockFn("hello");
    expect(result).toBe(42);
    expect(mockFn).toHaveBeenCalledWith("hello");
    expect(mockFn).toHaveBeenCalledTimes(1);
  });

  it("vi.spyOn: 객체 메서드 감시/대체", () => {
    const user = { getName: () => "Alice" };
    const spy = vi.spyOn(user, "getName")
                                .mockReturnValue("Bob");
    expect(user.getName()).toBe("Bob");
    expect(spy).toHaveBeenCalled();
  });
});
```

2.3 테스팅

Timer

- 개념
  시간 제어

분류	메서드	문법	설명
타이머	useFakeTimers	vi.useFakeTimers()	가짜 타이머 작동 시작
타이머	useRealTimers	vi.useRealTimers()	실제 타이머로 복귀
타이머	advanceTimersByTime	vi.advanceTimersByTime(ms)	ms 만큼 시간 진행
타이머	setSystemTime	vi.setSystemTime(date)	현재 시간을 특정 시점으로 설정

2.3 테스팅

구현 코드 (Timer)

```
// vitest6.timer.spec.js
import { describe, it, expect, vi, beforeEach, afterEach } from "vitest";

describe("가상타이머", () => {
  // 매 테스트 시작전 가상 타이머 실행
  beforeEach(() => {
    vi.useFakeTimers();
  });
  // 매 테스트 종료후 실제 타이머 실행
  afterEach(() => {
    vi.useRealTimers();
  });
```

```
  it("setTimeout 테스트", () => {
      let count = 0;
      setTimeout(() => {
        count = 1;
      }, 1000);
      expect(count).toBe(0);
      vi.advanceTimersByTime(1000);
      expect(count).toBe(1);
    });
    it("특정 날짜/시간 고정", () => {
      const fixed = new Date(2030, 0, 1, 9, 0, 0);
      vi.setSystemTime(fixed);
      expect(new Date().getFullYear()).toBe(2030);
    });
});
```

2.3 테스팅

VTU (@vue/test-utils) 개요

- 개념
  Vue 컴포넌트를 테스트하기 위한 패키지
  사용자와 상호작용 시뮬레이션 가능
  https://test-utils.vuejs.org/guide/

- 특징
  DOM 없이 컴포넌트 인스턴스 생성
  props/slots/global 옵션으로 다양한 상황 시뮬레이션
  이벤트/상호작용 트리거 및 랜더링 결과 관찰
  부모/자식 통신(emit/props) 검증

2.3 테스팅

VTU (@vue/test-utils) 주요 메서드

범주	메서드	문법	설명
마운트/ 언마운트	mount, unmount	mount(component, {props,slots,global}) wrapper.unmount()	컴포넌트 인스턴스 생성 컴포넌트 제거
찾기	find/get	wrapper.find('button') wrapper.get('[data-test="x"]')	요소 검색 get은 실패시 에러 발생
찾기(다중)	findAll	wrapper.findAll('li')	배열로 반환
컴포넌트 찾기	findComponent	wrapper.findComponent(Child) wrapper.getComponent(Child)	특정 자식 컴포넌트 검색
이벤트 트리거	trigger	await wrapper.trigger('click')	루트 요소에 이벤트 트리거
입력 헬퍼	setValue	await wrapper.find('input').setValue('A')	입력값 변경
상태 관찰	text,html,exists	wrapper.text() , wrapper.html() wrapper.exists()	텍스트/html/존재여부
속성/클래스	attribute/classes	wrapper.attribute('id'), wrapper.classes()	DOM 속성 / 클래스 확인
이벤트 관찰	emitted	wrapper.emitted(), wrapper.emitted('save')	자식이 emit한 이벤트 확인

2.3 테스팅

실습 폴더 구조 (VTU)

```
src/
  main.js
  components/
      Counter.vue
      Greeting.vue
      InputBox.vue
      SaveButton.vue
  tests/
      Counter1.mount.spce.js
      Counter2.find.get.spec.js
      Counter3.trigger.spec.js
      Counter4.unmount.spec.js
      Greeting.props.spec.js
      InputBox.setValue.spec.js
      SaveButton.emitted.spec.js
```

2.3 테스팅

구현 코드 (Counter/mount)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p data-test="count">Count: {{ count }}</p>
    <button data-test="inc" @click="inc">+</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => count.value++;
</script>
```

```js
// Counter.mount.spec.js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("mount 기본", () => {
  it("컴포넌트를 마운트하면 초기 상태를 확인할 수 있다", () => {

    const wrapper = mount(Counter); // 컴포넌트 불러오기
      // 화면에 Count: 0 이 보이는지 확인
     expect(wrapper.text()).toContain("Count: 0");
  });
});
```

2.3 테스팅

구현 코드 (Counter/find및get)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p data-test="count">Count: {{ count }}</p>
    <button data-test="inc" @click="inc">+</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => count.value++;
</script>
```

```js
// Counter2.find.get.spec.js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("find / get", () => {
  it("버튼 요소를 찾는다", () => {
    const wrapper = mount(Counter);

    // find: 없으면 null 반환
    const btn = wrapper.find('[data-test="inc"]');
    expect(btn.exists()).toBe(true);
    expect(btn.text()).toBe("+");

    // get: 없으면 에러 발생 (더 엄격)
    const count = wrapper.get('[data-test="count"]');
    expect(count.text()).toBe("Count: 0");
  });
});
```

2.3 테스팅

구현 코드 (Counter/trigger)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p data-test="count">Count: {{ count }}</p>
    <button data-test="inc" @click="inc">+</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => count.value++;
</script>
```

```js
// Counter3.trigger.spec.js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("trigger", () => {
  it("버튼 클릭 시 count가 1 증가한다", async () => {
    const wrapper = mount(Counter);

 // 클릭 이벤트 2번 발생
    await wrapper.get('[data-test="inc"]').trigger("click");
    await wrapper.get('[data-test="inc"]').trigger("click");
    expect(wrapper.get('[data-test="count"]').text()).toContain("2");
  });
});
```

2.3 테스팅

구현 코드 (Counter/unmount)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p data-test="count">Count: {{ count }}</p>
    <button data-test="inc" @click="inc">+</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => count.value++;
</script>
```

```js
// Counter4.unmount.spec.js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("unmount", () => {
  it("컴포넌트를 제거한다", () => {
    const wrapper = mount(Counter);
    wrapper.unmount();
    expect(wrapper.exists()).toBe(false); // 더 이상 존재하지 않음
  });
});
```

2.3 테스팅

구현 코드 (Greeting/props)

```vue
<!-- Greeting.vue -->
<template>
  <h1 data-test="msg">Hello, {{ msg }}</h1>
</template>
<script setup>
defineProps({ msg: String });
</script>
```

```js
// Greeting.props.spec..js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import Greeting from "../components/Greeting.vue";

describe("props", () => {
  it("props로 전달된 메시지를 표시한다", () => {

   //Greeting 컴포넌트에 props 전달
    const wrapper = mount(Greeting, { props: { msg: "World" } });
    expect(wrapper.get('[data-test="msg"]').text()).toBe("Hello, World");

    // 현재 props 확인
    expect(wrapper.props().msg).toBe("World");
  });
});
```

2.3 테스팅

구현 코드 (InputBox/setValue)

```vue
<!-- InputBox.vue -->
<template>
  <div>
    <input data-test="name" v-model="name" />
    <p data-test="result">Hello, {{ name }}</p>
  </div>
</template>
<script setup>
import { ref } from "vue";
const name = ref("");
</script>
```

```js
// InputBox.setValue.spec.js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import InputBox from "../components/InputBox.vue";

describe("setValue", () => {
  it("input에 값을 넣으면 화면에 반영된다", async () => {
    const wrapper = mount(InputBox);

    await wrapper.find('[data-test="name"]').setValue("Vue");
    expect(wrapper.get('[data-test="result"]').text()).toBe("Hello, Vue");
  });
});
```

2.3 테스팅

구현 코드 (SaveButton/emitted)

```vue
<!-- SaveButton.vue -->
<template>
  <button data-test="save" @click="$emit('save', '저장완료')">
    Save
</button>
</template>
```

```js
// SaveButton.emitted.spec.js
import { mount } from "@vue/test-utils";
import { describe, it, expect } from "vitest";
import SaveButton from "../components/SaveButton.vue";

describe("emitted", () => {
  it("버튼 클릭 시 save 이벤트가 발생한다", async () => {
    const wrapper = mount(SaveButton);

    await wrapper.get('[data-test="save"]').trigger("click");

    // emitted 이벤트 확인
    expect(wrapper.emitted().save).toBeTruthy();
    expect(wrapper.emitted().save[0]).toEqual(["저장완료"]);
  });
});
```

2.3 테스팅

@testing-library/vue 개요

- 개념
  - 사용자가 실제로 화면을 어떻게 쓰는지에 맞춰 테스트 지원
  - https://testing-library.com/

- 특징
  - 접근성 친화적 (Role/Name)
  - 사용자 입장에서 쉽게 접근 가능
  - @testing-library/jest-dom 패키지와 같이 사용

2.3 테스팅

@testing-library/vue 주요 메서드

범주	메서드	문법	설명
랜더링	render	render(component, {props,slots,global})	컴포넌트 화면에 랜더링
screen기반쿼리	getByRole	screen.getByRole(role, {name})	접근성역할+이름으로 검색
	getByText	screen.getByText(text)	특정 텍스트 포함요소 검색
	getByLabelText	screen.getByLabelText(text)	<label> 연결된 input 검색
	getByDisplayValue	screen.getByDisplayValue(value)	input 현재 값으로 검색
	getByTitle	scrren.getByTitle(text)	title 속성으로 검색
	queryByText	screen.queryByText(text)	요소 없을 경우 null 반환
비동기	findByText	screen.findByText(text)	비동기 처리 요소가 나타날 때까지 대기

2.3 테스팅

@testing-library/jest-dom 개요

- 개념
  - 기본 expect 문법에 DOM 전용 매처(matcher) 추가 지원
  - DOM 상태를 더 쉽게 검사 가능

https://testing-library.com/docs/ecosystem-jest-dom/

2.3 테스팅

@testing-library/jest-dom 주요 메서드

매처	문법	설명
toBeInTheDocument	expect(button).toBeInTheDocument()	요소가 DOM에 존재하는지 확인
toHaveTextContent	expect(div).toHaveTextContent('hello')	요소안에 특정 텍스트 포함 여부
toHaveAttribute	expect(img).toHaveAttribute('src','logo.png')	특정 속성 확인
toBeVisible	expect(modal).toBeVisible()	화면에 보이는 상태인지 확인
toBeDisabled	expect(input).toBeDisabled()	비활성화 상태인지 확인
toBeEnabled	expect(input).toBeEnabled()	활성화 상태인지 확인
toHaveClass	expect(div).toHaveClass('active')	클래스명 존재 여부 확인
toHaveStyle	expect(button).toHaveStyle({color:'red'})	인라인 스타일 확인
toContainElement	expect(list).toContainElement(item)	특정 자식 포함 여부
toHaveValue	expect(input).toHaveValue('hello')	input/textarea 값 확인

2.3 테스팅

@testing-library/user-event 개요

- 개념
  - 사용자가 실제 브라우저에서 클릭, 키보드 입력 등과 같은 행동을 하는 것처럼
  - 시뮬레이션 지원

  https://testing-library.com/docs/user-event/intro/

- 특징
  - 실제 브라우저 동작처럼 자연스러운 이벤트 지원
  - 사용자 관점에서 테스트 작성

2.3 테스팅

@testing-library/user-event 주요 메서드

메서드	문법	설명
click	await user.click(button)	마우스 클릭
type	await user.type(input, ‘hello’)	입력란에 글자 입력
selectOptions	await user.selectOptions(box,’opt1’)	select 옵션 선택
clear	await user.clear(input)	입력창 리셋
upload	await user.upload(fileInput, file)	파일 업로드
tab	await user.tab()	탭 키로 포커스 이동
hover/unhover	await user.hover(link)	마우스 오버/아웃

2.3 테스팅

실습 폴더 구조

```
src/
  components/
      Counter.vue
      Hello.vue
      List.vue
      LoginForm.vue
      NameForm.vue
  tests/
      Counter1.render.spce.js
      Counter2.query.spec.js
      Counter3.event.spec.js
      Hello.query.spec.js
      List.query.spec.js
      LoginForm.query.spec.js
      NameForm.event.spec.js
```

2.3 테스팅

구현 코드 (Counter/render)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p role="status">Count: {{ count }}</p>
    <button @click="inc" aria-label="increment">
      +
    </button>
    <button>저장</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => {
  count.value++;
};
</script>
```

```js
// Counter1.render.spec.js
import { render, screen } from "@testing-library/vue";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("render 기본", () => {
  it("render 후 초기값 검증", () => {
    // 컴포넌트 화면에 랜더링
    render(Counter);

    const status = screen.getByRole("status");
    // 초기값 검증
    expect(status).toHaveTextContent("Count: 0");
  });
});
```

2.3 테스팅

구현 코드 (Counter/query/matcher)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p role="status">Count: {{ count }}</p>
    <button @click="inc" aria-label="increment">
      +
    </button>
    <button>저장</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => {
  count.value++;
};
</script>
```

```js
// Counter2.query.spec.js
import { render, screen } from "@testing-library/vue";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("query 기본", () => {
  it("getByXXX: 일치하면 찾기", () => {
    // 컴포넌트 화면에 랜더링
    render(Counter);

    // aria-label 기반 이름
    const incBtn = screen.getByRole("button", { name: /increment/i });
    expect(incBtn).toHaveTextContent("+");

    // 텍스트 기반 이름
    const saveBtn = screen.getByRole("button", { name: "저장" });
    expect(saveBtn).toBeTruthy();
    const saveText = screen.getByText("저장");
    expect(saveText).toBeTruthy();
  });
});
```

2.3 테스팅

구현 코드 (Counter/event)

```vue
<!-- Counter.vue -->
<template>
  <div>
    <p role="status">Count: {{ count }}</p>
    <button @click="inc" aria-label="increment">
      +
    </button>
    <button>저장</button>
  </div>
</template>

<script setup>
import { ref } from "vue";
const count = ref(0);
const inc = () => {
  count.value++;
};
</script>
```

```js
// Counter2.query.spec.js
import { render, screen } from "@testing-library/vue";
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";
import Counter from "../components/Counter.vue";

describe("query 기본", () => {
  it("getByXXX: 일치하면 찾기", async () => {
    // 컴포넌트 화면에 랜더링
    render(Counter);

    const status = screen.getByRole("status");
    // aria-label 기반 이름
    const incBtn = screen.getByRole("button", { name: /increment/i });

    await userEvent.click(incBtn);
    expect(status).toHaveTextContent("Count: 1");
  });
});
```

2.3 테스팅

구현 코드 (Hello/matcher)

```vue
<!-- Hello.vue -->
<template>
  <h1>안녕하세요, Vue 테스트!</h1>
</template>
```

```javascript
// Hello.matcher.spec.js
import { render, screen } from "@testing-library/vue";
import Hello from "../components/Hello.vue";

it("텍스트가 화면에 존재하는지 확인", () => {
  render(Hello);
  const heading = screen.getByText("안녕하세요, Vue 테스트!");
  expect(heading).toBeInTheDocument();
  expect(heading).toHaveTextContent("Vue 테스트");
});
```

2.3 테스팅

구현 코드 (List/query/matcher)

```vue
<!-- List.vue -->
<template>
  <ul>
    <li>사과</li>
    <li>바나나</li>
  </ul>
</template>
```

```js
// List.query.spec.js
import { render, screen } from "@testing-library/vue";
import List from "../components/List.vue";

it("리스트에 특정 자식 요소가 있는지 확인", () => {
  render(List);
  const list = screen.getByRole("list");
  const item = screen.getByText("바나나");
  expect(list).toContainElement(item);
});
```

2.3 테스팅

구현 코드 (LoginForm/query/matcher)

```vue
<!-- LoginForm.vue -->
<template>
  <form>
    <label for="id">아이디</label>
    <input id="id" v-model="userId" />
  </form>
  <div>
    <p v-if="show">보이는 문구</p>
    <p v-else hidden>숨겨진 문구</p>
  </div>
  <div class="alert success">성공 메시지</div>
</template>

<script setup>
import { ref } from "vue";
const userId = ref("guest");
const show = ref(true);
</script>
```

```js
// LoginForm.matcher.spec.js
import { render, screen } from "@testing-library/vue";
import LoginForm from "../components/LoginForm.vue";

it("라벨과 input이 연결되고 초기값이 있는지 확인", () => {
  render(LoginForm);
  const input = screen.getByLabelText("아이디");
  expect(input).toHaveValue("guest");
});
it("조건부 요소가 보이는지/숨겨졌는지 확인", () => {
  render(LoginForm);
  expect(screen.getByText("보이는 문구")).toBeVisible();
  expect(screen.queryByText("숨겨진 문구")).toBeNull();
});
it("요소에 특정 클래스가 적용되어 있는지 확인", () => {
  render(LoginForm);
  const alert = screen.getByText("성공 메시지");
  expect(alert).toHaveClass("alert");
  expect(alert).toHaveClass("success");
});
```

2.3 테스팅

구현 코드 (NameForm/event)

```vue
<!-- NameForm.vue -->
<template>
  <form @submit.prevent>
    <label for="name">이름</label>
    <input id="name" v-model="name" placeholder="이름을 입력" />
    <p>미리보기: {{ name }}</p>
  </form>
</template>
<script setup>
import { ref } from "vue";
const name = ref("");
</script>
```

```js
// NameForm.event.spec.js
import { render, screen } from "@testing-library/vue";
import userEvent from "@testing-library/user-event";
import NameForm from "../components/NameForm.vue";

it("라벨로 입력 찾고 타이핑 반영", async () => {
  render(NameForm);
  const input = screen.getByLabelText("이름"); // label-for 연결
  await userEvent.type(input, "홍길동");
  expect(screen.getByText(/미리보기:/)).toHaveTextContent("홍길동");
});
```

2.4 빌드 최적화와 배포

프로젝트를 빌드하고 최적화하여 Docker 기반으로 실제 배포까지 전 과정을 이해하고 구현한다.

학습목표

- 빌드(Build) 이해
- 빌드 주요 동작 과정
- Vite 빌드 특징
- 빌드 최적화 기법
- 배포(Deployment) 개요
- 배포 방식
- Docker 이해
- 실제 배포 시나리오

2.4 빌드 최적화와 배포

빌드(build)

- 개념
  - 개발자가 작성한 소스코드를 브라우저가 이해하고 실행 가능한
  - 최적의 정적 자원(JS/CSS/HTML)으로 변환하는 과정.
  - 단순 변환이 아니라 최적화,정리,패키징을 모두 포함하는 종합적인 처리 단계.

- 필요성
  - 브라우저 호환성 확보
  - 성능 최적화
  - 운영 환경과 개발 환경 분리
  - 보안 강화

2.4 빌드 최적화와 배포

주요 동작 과정

A. 의존성 관리 (Dependency Resolution)
- import/require 등으로 불러온 모듈 분석
- 외부 라이브러리 포함 여부 확인

B. 코드 변환 (Transpilation)
- 최신 자바스크립트 문법을 구버전 브라우저 호환 코드로 변환
- Typescript,JSX,Vue SFC 파일을 표준 JS/CSS/HTML 로 변환

C. 번들링 (Bundling)
- 수 많은 모듈 및 파일을 하나 혹은 소수의 번들 파일로 통합
- 네트워크 요청 최적화

2.4 빌드 최적화와 배포

주요 동작 과정

D. 최적화 (Optimization)
- 압축(Minify), 코드 분리(Code Splitting), 캐싱 전략 적용
- 사용하지 않는 코드 제거 (Tree Shaking)

E. 산출물 생성 (Output Generation)
- 최종적으로 /dist 폴더에 정적 자원 생성

```
예>
  dist/
    index.html
    assets/
      index.a1b2c3.js
      style.f4e5d6.css
      logo.98f3a1.png
```

2.4 빌드 최적화와 배포

Vite 빌드 특징

- 개발단계 (Dev Mode)
  - 네이티브 ES 모듈 기반으로 번들링 없이 브라우저에서 빠른 HMR 제공
  - 즉시 반영되는 개발 환경 제공

- 배포단계 (Build Mode)
  - 내부적으로 Rollup 도구 사용하여 최적화된 번들 생성
  - npm run build 실행 시 최적화된 산출물 (/dist 폴더) 생성
  - 빠른 빌드 속도 지원
  - 코드 스플리팅 자동 적용
  - 해시(hash) 기반 파일명 생성으로 캐싱 최적화

2.4 빌드 최적화와 배포

최적화 기법

- 코드 압축 (Minification)
- 번들링 (Bundling)
- 트리 셰이킹 (Tree Shaking)
- 코드 스플리팅 (Code Splitting)
- 캐싱 전략 (Cache Strategy)
- 이미지 및 assets 최소화
- 코드 분리 및 외부 리소스 활용

2.4 빌드 최적화와 배포

코드 압축 (Minification)

- 개념
  - JS/CSS/HTML 파일에서 실행에 불필요한 요소 제거
    ex. 공백,줄바꿈,주석, 불필요한 변수명 제거 등
- 장점
  - 파일 사이즈 20% ~ 40% 절감 가능
  - 네트워크 전송 속도 향상
  - 초기 로딩 성능 개선
- Vite 동작방식
  - Vite는 기본적으로 esbuild 기반 압축기 사용
  - 기존 Terser 대비 10 ~ 100배 빠른 압축 성능 제공

2.4 빌드 최적화와 배포

번들링 (Bundling)

- 개념
  - 다수의 모듈 및 JS 파일을 하나 또는 소수의 파일로 통합
- 장점
  - HTTP 요청 수 최소화 및 초기 로딩 성능 개선
  - 모듈 시스템(ESM, CommonJS) 차이를 일관되게 통합 지원
- Vite 동작방식
  - 개발단계에서는 번들링 없이 네이티브 JS 모듈 사용 (속도 최적화)
  - 배포단계에서는 Rollup 기반 번들링 수행
- 고려사항
  - 대규모 프로젝트에서는 오히려 비효율적 (코드 스플리팅과 병행 필요)

2.4 빌드 최적화와 배포

트리 셰이킹 (Tree Shaking)

- 개념
  - 실제로 사용되지 않는 코드(Dead Code) 제거
    ex. 라이브러리에서 일부 기능만 사용된 경우, 사용 안된 나머지 제외
- 적용대상
  - 불필요한 함수, 클래스, 상수
  - 외부라이브러리 중 미사용 모듈
- Vite 동작방식
  - Rollup 최적화 기능과 함께 ESM 기반 import/export 구조에서 자동 적용

2.4 빌드 최적화와 배포

코드 스플리팅 (Code Splitting)

- 개념
  - 코드를 페이지별/기능별로 분리하여 지연로딩 (Lazy Loading) 적용
  - Vue Router의 import() 활용해 라우트 단위로 코드 분리
- 장점
  - 초기 로딩 속도 단축
  - 페이지 전환 시 지연 로딩 가능
- 전략
  - 페이지 단위 스플리팅 권장
  - 공통모듈 (axios 등)은 별도 chunk로 분리

2.4 빌드 최적화와 배포

캐싱 전략 (Cache Strategy)

- 개념
  - 변경되지 않는 파일을 브라우저가 보관하여 반복 다운로드 방지
- 방법
  - 파일명 해싱(Hashing) 적용
    ex. app.**98f3a1**.js , 내용이 변경된 경우에만 해시값이 바뀜
  - HTTP 캐시 헤더 적용
    ex. Cache-Control: max-age=31536000, immutable
- 효과
  - 재방문 시 불필요한 네트워크 트래픽 감소
  - CDN (Content Delivery Network) 활용 시 글로벌 성능 최적화

2.4 빌드 최적화와 배포

이미지 및 assets 최소화

- 이미지 최적화
  - WebP, AVIF 와 같은 최신 이미지 포맷 적용
  - JPEG 대비 30% ~ 50% 용량 절감 효과
- 자동화 도구
  - vite-plugin-imagemin
  - 빌드 시 이미지 자동 최적화

2.4 빌드 최적화와 배포

코드 분리 및 외부 리소스 활용

- 라이브러리 외부화 (Externals)
  - CDN 통해 자주 사용되는 라이브러리를 불러와 번들 크기 감소
    ex. Vue, Firebase SDK 등 대규모 패키지

- Chunk 분리 전략
  - vender.js : 외부 라이브러리 전용
  - app.js : 어플리케이션 코드 전용

2.4 빌드 최적화와 배포

배포 (Deployment)

- 개념
  - 빌드(Build) 후 정적 자원을 최종 사용자가 접근 가능하도록 서비스 환경에 배치하는 일련의 과정

- 필요이유
  - 최종 사용자가 접근 가능성 확보
  - 일관된 실행 환경 보장
  - 성능 및 안전성 관리 보장
  - 운영 및 유지보수 편리성 보장

2.4 빌드 최적화와 배포

배포 방식

- 로컬 배포 (Local Deployment)
  - 개인 PC에서 Docker, Nginx 등을 이용하여 배포
  - 교육 목적이나 테스트용
- 온 프레미스 (On-premise)
  - 자체 서버에 배포
  - 인프라 관리 및 보안 직접 제어
- 클라우드 배포 (Cloud Deployment)
  - AWS, GCP, Azure 등 클라우드 환경에 배포
  - 확장성 및 가용성 높음
  - 관리 복잡도와 비용

2.4 빌드 최적화와 배포

배포 방식

- CI/CD 자동화
  - Github Actions, Jenkins 등 파이프라인 도구 활용
  - 코드 변경 시 자동으로 빌드,테스트,배포까지 수행
  - 기업 환경에서는 필수적
  - 일반 학습자/초보자는 진입장벽 높음

2.4 빌드 최적화와 배포

Docker 이해

- 이미지 (image)
  - Docker 이미지는 어플리케이션 실행에 필요한 모든 것을 포함
    ex. 파일/라이브러리/실행환경/어플리케이션 코드등 포함
  - Dockerfile 을 사용하여 정의하고 빌드 후에는 읽기 전용
  - 동일한 이미지는 여러 컨테이너에서 실행 가능

- 컨테이너 (Container)
  - Docker 컨테이너는 Docker 이미지의 인스턴스
  - 컨테이너는 격리된 환경에서 실행, 호스트 시스템과 독립적
  - 가상머신처럼 동작하지만 가상화보다 오버헤드가 적고 더 가벼움
  - 컨테이너는 Docker 엔진에 의해 관리, 시작/중지/삭제 등 작업 수행 가능

2.4 빌드 최적화와 배포

Docker 특징

- 재현 가능성
  - 어디서 실행해도 결과가 동일
- 이식성
  - Win / Mac / Linux 모두 동일한 이미지로 실행
- 단순성
  - Docker file > build > run 만으로 배포 가능
  - 배포 절차가 표준화
- 운영환경 시뮬레이션
  - Docker 기반 배포를 통해 실제 기업 환경과 동일한 구조를 로컬에서 체험

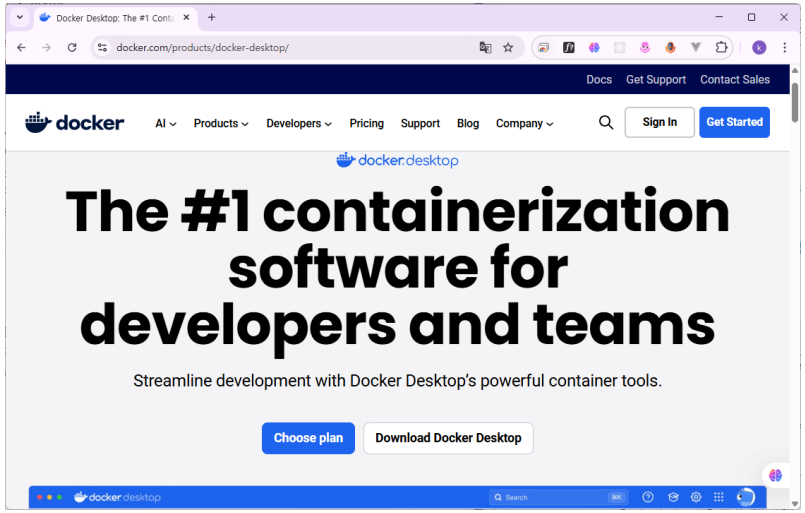
2.4 빌드 최적화와 배포

Docker 설치

- 윈도우 (Window)

  Docker Desktop 설치

  https://www.docker.com/products/docker-desktop/



2.4 빌드 최적화와 배포

Docker 설치

- 설치 확인

  docker version

  docker info

```
C:\Users\User>docker version
Client:
 Version:           28.3.2
 API version:       1.51
 Go version:        go1.24.5
 Git commit:        578ccf6
 Built:             Wed Jul  9 16:12:31 2025
 OS/Arch:           windows/amd64
 Context:           desktop-linux

Server: Docker Desktop 4.44.3 (202357)
 Engine:
  Version:          28.3.2
  API version:      1.51 (minimum version 1.24)
  Go version:       go1.24.5
  Git commit:       e77ff99
  Built:            Wed Jul  9 16:13:55 2025
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.7.27
  GitCommit:        05044ec0a9a75232cad458027ca83437aae3f4da
```

2.4 빌드 최적화와 배포

Dockerfile 이해

```
 # 1. Node로 빌드
FROM node:22 AS build                  # base 이미지
WORKDIR /app                           # 작업 디렉터리
COPY package*.json  ./                 # 의존성 설치 파일 복사
RUN npm install
COPY . .                               # 어플리케이션 파일 추가
RUN npm run build                      # 빌드 프로세스 실행

# 2. Nginx로 서비스
FROM nginx:alpine                      # 어플리케이션 실행 환경인 서버(Nginx) 설정
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80                              # 서버 포트 80
CMD ["nginx", "-g", "daemon off;"]     # Nginx 서버 실행
```

2.4 빌드 최적화와 배포

실습 폴더 구조 (docker)

```
project
  dist
  node_modules
  public
  src
  .dockerignore          #  빌드 컨텍스트 최소화
  .env                       #  Firebase URL 설정 필요
  .gitignore
  Dockerfile             # Docker 설정 정보
  index.html
```

2.4 빌드 최적화와 배포

실습 배포 시나리오

A. Vite 프로덕션 빌드
npm run build
dist 폴더 생성

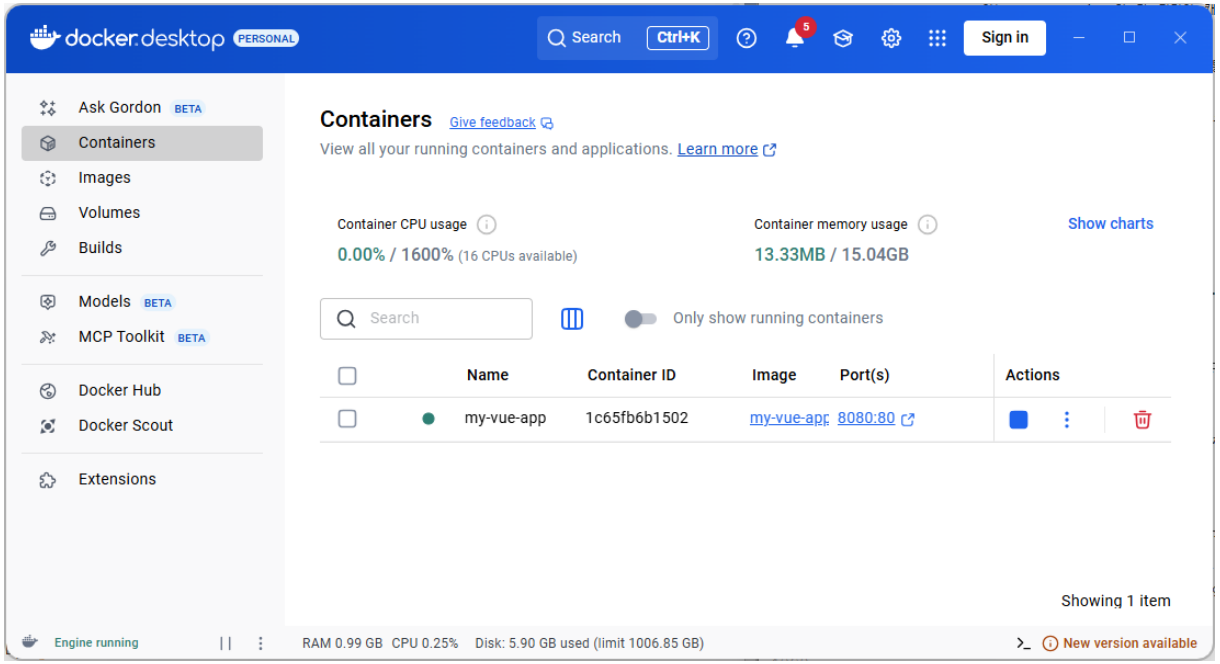
B. Docker 이미지 생성
docker build -t my-vue-app .

C. 컨테이너 실행
docker run -d --name my-vue-app -p 8080:80 my-vue-app

D. 브라우저 요청
http://localhost:8080

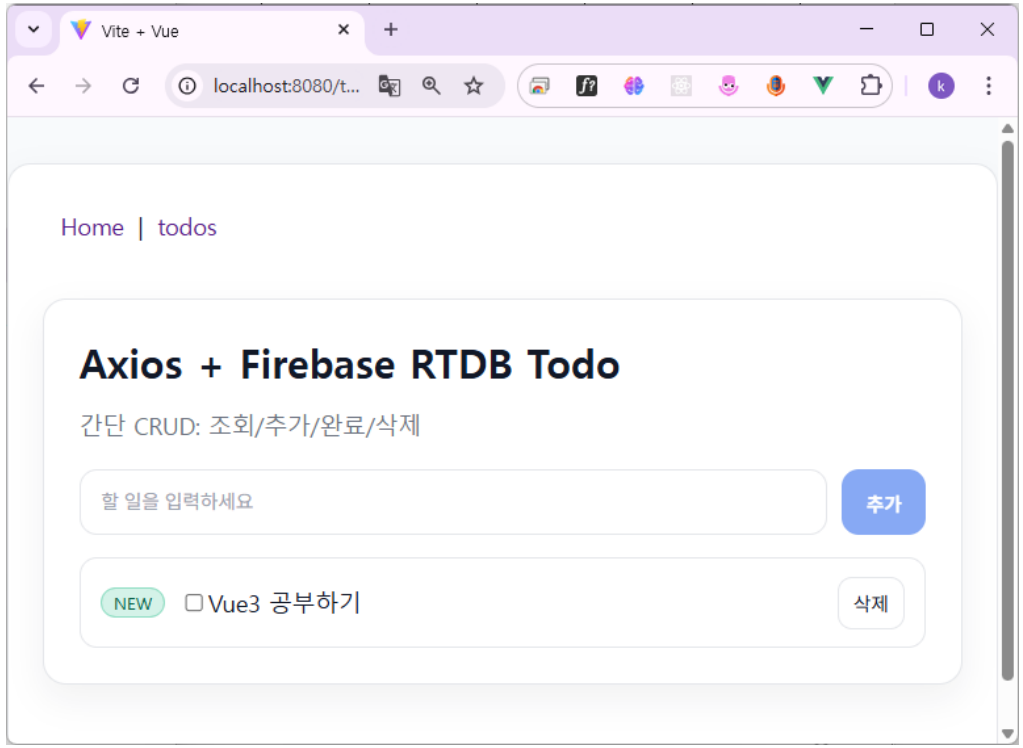
2.4 빌드 최적화와 배포

Docker desktop (Container)



2.4 빌드 최적화와 배포

최종 실행 화면



2.5 TypeScript와 Vue 3

TypeScript와 Vue 3를 중심으로 환경 설정부터 기본 문법과 실습까지 학습한다.

학습목표

- TypeScript 개요
- node.js + TypeScript 환경 설정
- TS 기본 문법
- Vite + Vue3 + TypeScript 환경 설정
- Vue+TS 문법
- TS 기반 실습

2.5 TypeScript와 Vue 3

TypeScript 개요

- 개념
  - 자바스크립트 위에 타입 시스템이 추가된 확장 언어
  - 기본문법은 **:타입** 형식으로 변수/파라미터/리턴값 정의

- 특징
  - 정적 타입 체크
  - 스마트 에디터 지원
  - 최신 자바스크립트 문법 지원
  - 점진적 도입 가능
  - 타입 선언 파일 (@types)

2.5 TypeScript와 Vue 3

TypeScript 장단점

- 장점
  - 버그 감소
  - 협업 강화
  - 유지보수 용이
- 단점
  - 초기 학습 곡선
  - 추가 설정/컴파일 과정 필요
  - 과한 타입 설계 시 생산성 저하

2.5 TypeScript와 Vue 3

TypeScript 주요 타입

- string
- string[]
- number
- boolean
- null, undefined
- any
- unknown

2.5 TypeScript와 Vue 3

Node.js + TypeScript 단독 환경설정

A. Node.js 설치

B. TypeScript 초기화

```
mkdir ts-hello && cd ts-hello
npm init -y
npm install --save-dev typescript ts-node @types/node
npx tsc --init
```

```
# 생성된 디렉터리 구조
ts-hello/
  node_modules/
  package-lock.json
  package.json
  tsconfig.json
```

2.5 TypeScript와 Vue 3

Node.js + TypeScript 단독 환경 설정

C. src 폴더 생성

```
ts-hello/
  src/
     index.ts
  tsconfig.json
```

D. src/index.ts 작성

```
function add(a: number, b: number): number {
  return a + b;
}
const name: string = "타입스크립트";
console.log(`안녕하세요, ${name}! 2+3=${add(2,3)}`);
```

2.5 TypeScript와 Vue 3

Node.js + TypeScript 단독 설정

E. 실행

- ts 형식

  npx ts-node src/index.ts

- 컴파일 후 js 형식

  npx tsc

  node src/index.js

F. 실행 결과

안녕하세요, 타입스크립트! 2+3=5

2.5 TypeScript와 Vue 3

기본 문법

- 기본 타입

```
let username: string = "Tom";
let age: number = 25;
let isAdmin: boolean = true;

// any는 가급적 사용 지양
let anything: any = "문자열";
anything = 123;
```

2.5 TypeScript와 Vue 3

기본 문법

- 배열과 튜플

```
let numbers: number[] = [1, 2, 3];
let fruits: Array<string> = ["apple", "banana"];

// 튜플 (고정된 타입 순서)
let user: [string, number] = ["Tom", 25];
```

2.5 TypeScript와 Vue 3

기본 문법

- 함수 타입

```
// 매개변수 타입, 리턴 타입 지정
function greet(name: string): string {
  return `Hello, ${name}`;
}

// 선택적 매개변수
function printInfo(name: string, age?: number): void {
 console.log(`${name}, ${age ?? "unknown"}세`);
}
```

2.5 TypeScript와 Vue 3

기본 문법

- 인터페이스와 타입 별칭

```
// 인터페이스
interface User {
  id: number;
  name: string;
  email?: string; // 선택적 프로퍼티
}

// 타입 별칭
type Point = {
  x: number;
  y: number;
}
```

```
const u: User = { id: 1, name: "Alice" };
const p: Point = { x: 10, y: 20 };
```

2.5 TypeScript와 Vue 3

기본 문법

- 유니온 & 교차타입

```
// 유니온
let value: string | number;
value = "Hello";
value = 123;

// 교차 타입
interface Person { name: string }
interface Worker { job: string }
type Employee = Person & Worker;

const e: Employee = { name: "Tom", job: "Dev" };
```

2.5 TypeScript와 Vue 3

기본 문법

- 제네릭(Generic)

```
// 제네릭 함수
function identity<T>(value: T): T {
  return value;
}

let n = identity<number>(123);
let s = identity<string>("Hello");
```

```
// 제네릭 인터페이스
interface ApiResponse<T> {
  data: T;
  success: boolean;
}

const res: ApiResponse<string[]> = {
  data: ["a", "b"],
  success: true
};
```

2.5 TypeScript와 Vue 3

기본 문법

- 클래스와 접근 제한자

```
class Person {
  private id: number;
  public name: string;
  protected age: number;

  constructor(id: number, name: string,
                  age: number) {
    this.id = id;
    this.name = name;
    this.age = age;
  }
  greet(): string {
    return `Hello, I'm ${this.name}`;
  }
}

const p = new Person(1, "Tom", 20);
console.log(p.greet());
```

2.5 TypeScript와 Vue 3

기본 문법

- 타입단언 (as T)

```
// 단언(as)
let val: unknown = "Hello";
let len = (val as string).length;
```

2.5 TypeScript와 Vue 3

기본 문법

- 유틸리티 타입 (Partial, Pick, Omit, Readonly)

```
interface Todo {
  id: number;
  title: string;
  done: boolean;
}
```

```
// 일부만 필요
type PartialTodo = Partial<Todo>;

// 특정 키만 선택
type TodoPreview = Pick<Todo, "title" | "done">;

// 특정 키 제외
type TodoID = Omit<Todo, "title" | "done">;

// 모두 읽기 전용
type ReadonlyTodo = Readonly<Todo>;
```

2.5 TypeScript와 Vue 3

기본 문법

- Enums (열거형)

```
enum Role {
  Admin,
  User,
  Guest,
}

const r: Role = Role.Admin;
console.log(r); // 0
```

2.5 TypeScript와 Vue 3

Vite + Vue 3 + TypeScript 환경 설정

A. 프로젝트 생성

npm create vite@latest my-vue-ts-app -- --template vue-ts

```
# 생성된 디렉터리 구조
my-vue-ts-app
    src/
      components/
      App.vue
      main.ts
    package.json
    tsconfig.app.json
    tsconfig.json
    tsconfig.node.json
```

2.5 TypeScript와 Vue 3

Vite + Vue 3 + TypeScript 환경 설정

B. src/env.d.ts 생성

```
/// <reference types="vite/client" />

declare module "*.vue" {
  import { DefineComponent } from "vue";
  const component: DefineComponent<{}, {}, any>;
  export default component;
}
```

2.5 TypeScript와 Vue 3

Vite + Vue 3 + TypeScript 환경 설정

C. 프로젝트 실행

```
cd my-vue-ts-app
npm install --save-dev @types/node
npm run dev
```

D. 브라우저 요청

http://localhost:5173/

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 ref

```
import { ref } from "vue";

const count = ref<number>(0);     // 숫자 타입
const title = ref<string>("Vue3"); // 문자열 타입
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 reactive

```
import { reactive } from "vue";

interface Profile {
  id: number;
  name: string;
}
const profile = reactive<Profile>({
  id: 1,
  name: "Tom”
});

const user = reactive<{ id: number; name: string }>({ id: 1, name: "Lee" });
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 computed

```
import { ref, computed } from "vue";

interface Todo { id: number; title: string; done: boolean }

const todos = ref<Todo[]>([
  { id: 1, title: "공부", done: false },
  { id: 2, title: "코딩", done: true },
]);

const doneList = computed<Todo[]>(() => todos.value.filter(t => t.done));
const doneCount = computed<number>(() => doneList.value.length);
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 watch

```
import { ref, watch} from "vue";

const keyword = ref<string>("");

watch(keyword, (newVal: string, oldVal: string) => {
  console.log("변경:", oldVal, "->", newVal);
});
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 props

```
const props = defineProps<{
  msg: string;
  count?: number; // 선택적
}>();

interface Item { id: number; name: string }

const props = defineProps<{
  items: Item[];
  formatter?: (name: string) => string;
  sizes?: Array<"sm" | "md" | "lg">;
}>();
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 emits

```
const emit = defineEmits<{
  (e: "update", payload: { id: number; value: string }): void;
  (e: "close"): void;
}>();
function onClick() {
  emit("update", { id: 1, value: "hello" });
}
</script>
<template>
  <button @click="onClick">업데이트</button>
  <button @click="$emit('close')">닫기</button>
</template>
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 provide/Inject

```
mport { provide } from "vue";

interface Auth { user: string; isAdmin: boolean }
const auth: Auth = { user: "tom", isAdmin: true };

provide<Auth>("auth", auth);

import { inject } from "vue";
interface Auth { user: string; isAdmin: boolean }

const auth = inject<Auth>("auth");
```

2.5 TypeScript와 Vue 3

기본 문법

- Vue의 Composable

```
import { ref, computed } from "vue";

export function useCounter(initial = 0) {
  const n = ref<number>(initial);
  const inc = (step: number = 1): void => (n.value += step);
  const dec = (step: number = 1): void => (n.value -= step);
  const double = computed<number>(() => n.value * 2);

  return { n, inc, dec, double };
}
```

2.5 TypeScript와 Vue 3

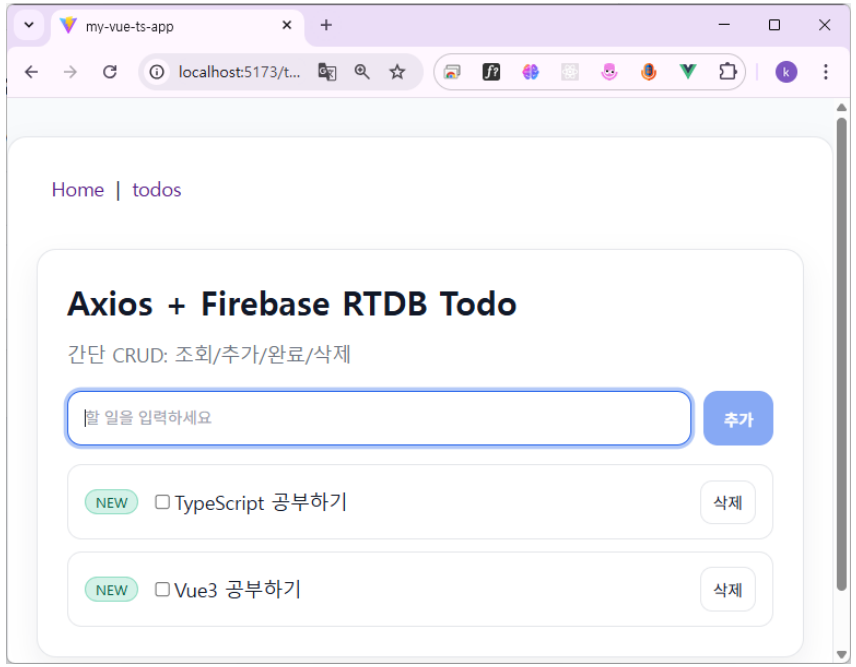
기본 문법

- Vue의 비동기

```
import { ref } from "vue";
interface User { id: number; name: string }

const user = ref<User | null>(null);
const error = ref<string | null>(null);
const loading = ref<boolean>(false);
async function loadUser(id: number): Promise<void> {
  loading.value = true;
  error.value = null;
  try {
    // const res = await fetch(`/api/users/${id}`);
    // user.value = await res.json() as User;
    user.value = { id, name: "Mock" }; // 데모용
  } catch (e) {
    error.value = (e as Error).message;
  } finally {
    loading.value = false;
  }
}
```

2.5 TypeScript와 Vue 3

실습



2.5 TypeScript와 Vue 3

구현 코드 (JS vs TS)

```js
// useTodo.js
const todos = ref([]);
const loading = ref(false);
const error = ref(null);

export function useTodos() {
  const ordered = computed(() =>
    [...todos.value].sort((a, b) => b.createdAt - a.createdAt)
  );

  async function fetchTodos() {
    loading.value = true;
    error.value = null;
    try {
      const { data } = await api.get("/todos.json");
      const list = data
        ? Object.entries(data).map(([id, v]) => ({ id, ...v }))
        : [];
      todos.value = list;
    } catch (e) {
      console.error("조회 실패:", e?.response?.status, e?.message);
      error.value = "데이터를 불러오지 못했습니다. 잠시 후 다시 시도하세요.";
    } finally {
      loading.value = false;
    }
  }
;
```

```ts
// useTodo.ts
const todos: Ref<Todo[]> = ref([]);
const loading: Ref<boolean> = ref(false);
const error: Ref<string | null> = ref(null);

export function useTodos() {
  const ordered = computed<Todo[]>(() =>
    [...todos.value].sort((a, b) => b.createdAt - a.createdAt)
  );

  async function fetchTodos(): Promise<void> {
    loading.value = true;
    error.value = null;
    try {
      const { data } = await api.get<FirebaseList>("/todos.json");
      const list: Todo[] = data
        ? Object.entries(data).map(([id, v]) => ({ id, ...v })) : [];
      todos.value = list;
    } catch (e: unknown) {
      console.error("조회 실패:", (e as any)?.response?.status,(e as
Error)?.message
      );
      error.value = "데이터를 불러오지 못했습니다. 잠시 후 다시 시도하세요.";
    } finally {
      loading.value = false;
    }
  }
```

2.6 정리

요약맵

- Teleport 및 Suspense 활용
- v-memo 및 비동기 컴포넌트 활용
- 빌드 및 배포
- Docker 컨테이너 활용
- Vue 3 에서의 TypeScript 적용

수고하셨습니다.

SK Inc. AX