SKALA

대중소 상생

Vue.js 기반 실전 Frontend 개발

2025.11

SK Inc. AX

1. Vue.js 기초
2. Vue.js 핵심 기능
3. 상태 관리와 라이프사이클

1. Vue.js 기초

- Vue.js 개요 및 SPA 개념
- Vue 3 설치 및 개발환경 구성
- Vue 인스턴스와 템플릿 문법
- 데이터 바인딩과 디렉티브
- 정리 및 Q&A

SK Inc. AX

1.1 Vue.js 개요 및 SPA 개념

Vue.js 는 직관적이고 강력한 컴포넌트 기반 구조의 프론트앤드 프레임워크이다.

학습목표

- Vue.js 개요와 특징 이해
- 컴포넌트 개념과 장점 학습
- MVVM 패턴과 Vue.js 아키텍처 이해
- 웹 어플리케이션 구조 비교
- SPA vs MPA
- CSR vs SSR

1.1 Vue.js 개요 및 SPA 개념

Vue.js 개요

- 등장: 2014년 Evan You (전 Google 의 Angular 팀)
- 목표: 단순하고 가벼운 프레임워크
- 장점
  - 컴포넌트 기반 : 재사용성
  - 양방향 바인딩 : UI 자동 동기화
  - HTML 기반 템플릿 : 쉬운 문법
  - Angular + React 장점 결합

1.1 Vue.js 개요 및 SPA 개념

컴포넌트 개요

- 개념
  - UI와 로직을 합친 독립 블록
  - Vue CLI 환경에서는 .vue 확장자
- 특징: 독립성/재사용성/조립성/확장성
- Vue 컴포넌트 구성
  - template: UI 구조
  - script : 데이터 & 로직
  - style: 디자인

1.1 Vue.js 개요 및 SPA 개념

컴포넌트 구성 예

```
<template>
  <button
@click="count++">+</button>
  클릭 횟수: {{ count }}
</template>
```

<template> 영역 : 화면 UI에 보여질 HTML 구조 정의

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

const count = ref(0);

</script>
```

<script> 영역 : 컴포넌트의 동작 로직 작성

```
<style>
button {
  background: greenyellow;
}
</style>
```

<style> 영역 : 컴포넌트 전용 스타일

1.1 Vue.js 개요 및 SPA 개념

MVVM 패턴

- 구조
  - Model : 데이터 & 로직
  - View : UI (화면)
  - ViewModel : Model과 View 연결

- Vue 아키텍쳐: Vue는 MVVM 패턴 기반 프레임워크

1.1 Vue.js 개요 및 SPA 개념

데이터 흐름

- 단방향
  - One-way binding
  - Model 에서 View 로 데이터 전달
  - 출력 중심
  - {{ }} 보간법

- 양방향
  - Two-way binding
  - Model 과 View 서로 간에 데이터 전달
  - 입력과 출력 중심
  - v-model 디렉티브

1.1 Vue.js 개요 및 SPA 개념

SPA (Single Page Application)

- 개념
  - 하나의 HTML 에서 동작
  - API 로 데이터만 갱신

- 특징
  - 서버: 데이터 제공
  - 클라이언트: UI 랜더링 (CSR 기반)

1.1 Vue.js 개요 및 SPA 개념

SPA 동작 흐름

- 최초 요청: HTML + JS + CSS
- 초기 랜더링
- 데이터 로딩: API 호출
- 내부 네비게이션: 부분화면 갱신

1.1 Vue.js 개요 및 SPA 개념

SPA 장단점

- 장점
  - 화면 전환 빠름
  - 앱과 유사한 UX
  - 협업 효율적

- 단점
  - 초기 로딩 지연
  - SEO 취약
  - 복잡성 증가

1.1 Vue.js 개요 및 SPA 개념

MPA (Multi Page Application)

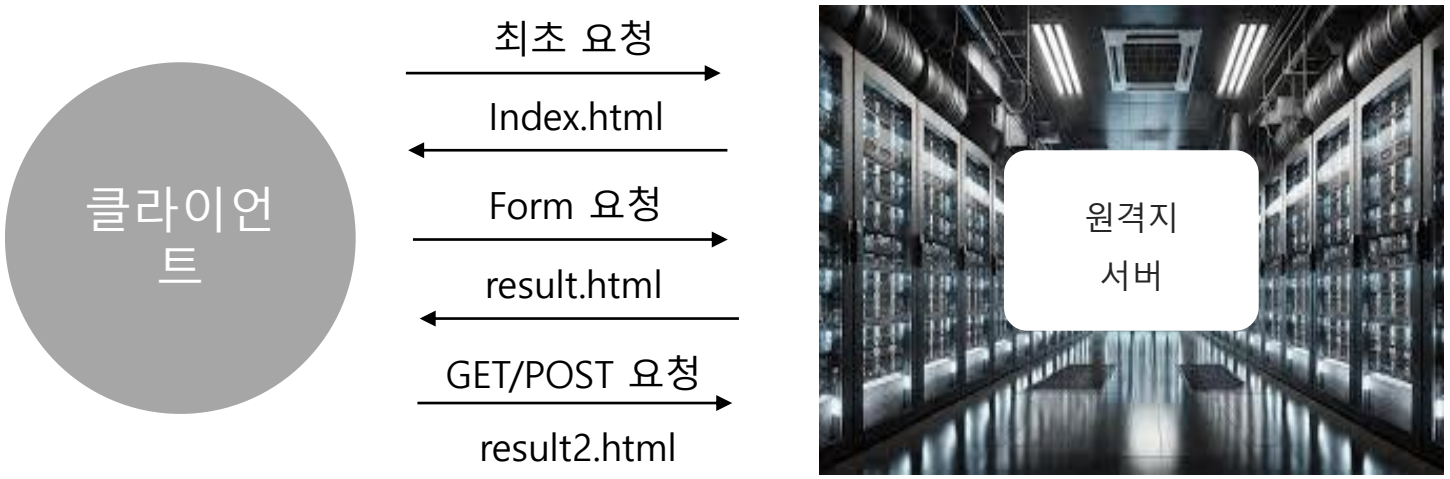
- 개념
  - 요청마다 서버가 새로운 HTML 생성
  - 전통적인 아키텍쳐
  - 각 URL은 별도의 페이지로 구성

- 특징
  - 서버: 페이지 단위 HTML 생성/전송
  - 클라이언트: 폼 검증 등 보조적 JS

1.1 Vue.js 개요 및 SPA 개념

MPA 동작 흐름

- 최초 요청: 완성된 HTML (JS + CSS)
- 초기 랜더링
- 데이터 로딩: 데이터 포함한 HTML 새로 생성
- 페이지 네비게이션: 전체 페이지 새로고침



1.1 Vue.js 개요 및 SPA 개념

MPA 장단점

- 장점
  - 초기 화면 빠름
  - 구조 명확
  - SEO 유리
- 단점
  - 전체 새로 고침
  - 서버 부하
  - UX 저하

1.1 Vue.js 개요 및 SPA 개념

SPA vs MPA

	구분	SPA	MPA
01	구조 / 랜더링	• 단일 페이지 + CSR 중심	• 다중 페이지 + SSR 중심
02	초기속도 / 전환	• 초기 번들로 느릴 수 있음. 전환은 빠름	• 초기 빠름, 전환 시 새로 고침
03	SEO/접근성	• CSR 만으로는 SEO 약함 (SSR 필요)	• 기본적으로 SEO 강함. HTML 중심 접근성 유리
04	서버/트래픽	• API(JSON) 중심, 서버 부하 낮음/예측가능	• HTML 랜더 요청마다 수행, 서버 부하 높음
05	적합도/사례	• 앱형/리치 인터랙션 (ex. Gmail, Trello)	• 컨텐츠/문서 중심 (ex. 뉴스,블로그)

1.1 Vue.js 개요 및 SPA 개념

CSR (Client Side Rendering)

- 개념: 브라우저에서 JS 실행해서 UI 랜더링
- 특징
  - 서버 최소 역할
  - SPA 핵심 방식
  - API 중심

1.1 Vue.js 개요 및 SPA 개념

CSR 장단점

- 장점
  - 빠른 전환
  - 동적 UI
  - API 중심
- 단점
  - 초기 로딩 지연
  - SEO 취약
  - JS 의존성

1.1 Vue.js 개요 및 SPA 개념

SSR (Server Side Rendering)

- 개념 : 서버에서 HTML 미리 랜더링 후 전달
- 특징
  - 서버 부하
  - 최신 데이터 즉시 반영
  - 초기 표시 / SEO 유리

1.1 Vue.js 개요 및 SPA 개념

SSR 장단점

- 장점
  - 빠른 초기 표시
  - SEO 유리
  - 저사양 기기 지원
- 단점
  - 서버 부하
  - 응답 지연
  - 구현 복잡

1.1 Vue.js 개요 및 SPA 개념

CSR vs SSR

구분	CSR	SSR
랜더링 위치	• 클라이언트에서 JS 실행 후 화면 생성. 서버는 데이터만 전달.	• 서버(HTML) 에서 만들어 클라이언트에 전달
초기속도	• JS 번들을 모두 다운 받아 실행해야 되므로 첫 로딩이 느림	• 완성된 HTML 을 제공하므로 첫 로딩이 빠름
SEO	• 컨텐츠가 JS 실행 후 보이기 때문에 검색 엔진에 불리	• 서버에서 바로 HTML 제공하기 때문에 검색 엔진 최적화에 유리
서버 부하	• 서버는 API만 응답하므로 상대적으로 가벼움	• 요청마다 HTML 랜더링 필요하기 때문에 서버 자원 사용이 많음
UX	• 초기 느리지만 이후 전환은 매우 부드럽고 빠름	• 첫 화면은 빠르지만 전환 시 서버 요청이 필요. 하이브리드 방식으로 보완 가능

1.2 Vue 3 설치 및 개발 환경 구성

Node.js , VS Code, Vite – 세 가지 도구로 완벽한 개발이 가능하다.

학습목표

- Node.js 설치
- VS Code 설치 및 설정
- Vite 설치
- Vue 프로젝트 생성 및 실행

1.2 Vue 3 설치 및 개발 환경 구성

Node.js

- 개념 : 서버 사이드 JavaScript 실행 환경
- 설치 목적
  - Vue.js, React 등 프론트앤드 프레임워크 실행에 필수
  - npm (Node Package Manager)을 통한 라이브러리 관리
- 설치 흐름
  - LTS 다운로드
  - npm 설치확인
  - 환경 변수 (PATH) 확인

1.2 Vue 3 설치 및 개발 환경 구성

LTS 다운로드

- 공식 사이트
  - https://nodejs.org/ko/download
  - LTS (Long Term Support) 버전 권장
  - 기본 설치 방법으로 설치
- LTS 특징
  - 안정성, 호환성 보장
  - 장기간 지원 (교육 및 실무 환경에 적합)

1.2 Vue 3 설치 및 개발 환경 구성

npm 설치 확인

- Node.js 설치 시 npm 자동 포함
- 설치 완료 후 터미널/명령 프롬프트 실행
  - node -v
  - npm –v
  - 정상적으로 버전 번호가 나오면 설치 완료

1.2 Vue 3 설치 및 개발 환경 구성

환경 변수 (PATH) 확인

- 설치 후 자동으로 PATH에 등록
- 확인 방법
  - Windows : 제어판 > 시스템 > 고급 시스템 설정 > 환경 변수
  - Path 항목에 c:₩Program Files₩nodejs₩ 존재 여부 확인
  - 문제가 있을 경우 수동으로 추가

1.2 Vue 3 설치 및 개발 환경 구성

VS Code

- 개념
  - MS에서 개발한 오픈 소스 에디터
  - 가볍지만 강력한 확장성
  - 웹/프론트앤드 개발에 최적화
- 공식 사이트
  - https://code.visualstudio.com
  - Windows, Mac, Linux 지원
- 필수 확장 프로그램
  - Vue (Official)
  - Prettier - Code formatter

1.2 Vue 3 설치 및 개발 환경 구성

Vite

- 개념
  - Node.js 기반 Vue 개발을 위한 스캐폴딩(scaffolding) 도구
  - Vue.js 프로젝트의 빠른 시작과 효율적 관리 지원
  - https://ko.vite.dev/guide/

- 설치 및 확인
  - 설치: **npm create vite@latest 프로젝트명**
  - 확인: cd 프로젝트명
    npm list vite

- 주요 기능
  - 프로젝트 생성, 빌드/배포 지원

1.2 Vue 3 설치 및 개발 환경 구성

Vue 프로젝트 생성 및 실행

- 생성 명령어: **npm create vite@latest my-vue-app**
- 옵션 선택
  - Framework : Vue 선택
  - Variant: JavaScript 선택
- 실행 명령어
  - cd my-vue-app
  - npm install
  - npm run dev
  - http://localhost:5173

1.2 Vue 3 설치 및 개발 환경 구성

주요 디렉터리 구조

node_modules : npm으로 설치된 패키지들

public : 정적 리소스

index.html : Vue 홈페이지

src/assets : 이미지, CSS, 폰트등

src/components : 자식 Vue 컴포넌트

App.vue : 최상위(Root) Vue 컴포넌트

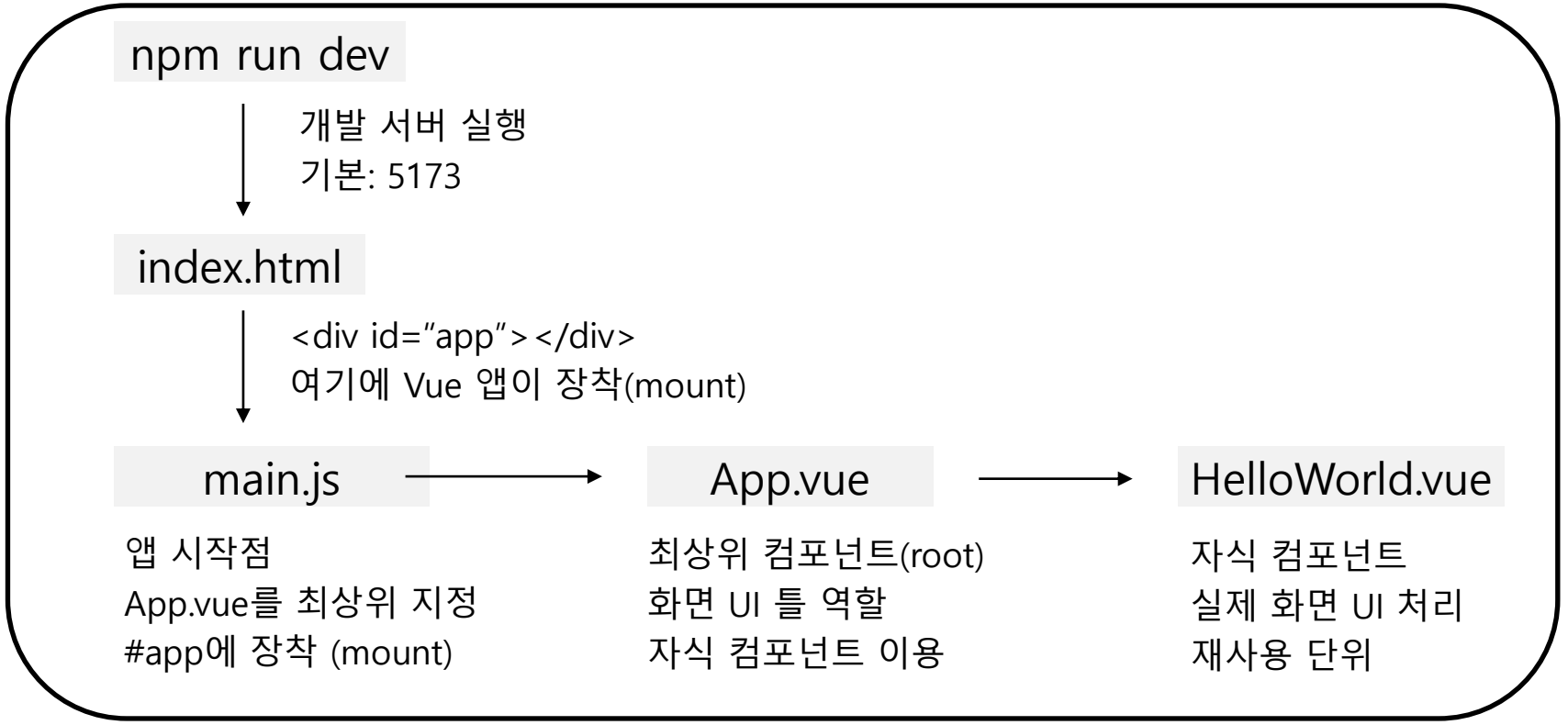
main.js : 가장 먼저 실행되는 JS 파일/Vue 인스턴스를 생성하는 역할

package.json : 프로젝트에 필요한 package 정의

vite.config.js : Vite 프로젝트의 전체 빌드와
개발 서버동작방식 등을 수정할 수 있는 설정파일

1.2 Vue 3 설치 및 개발 환경 구성

전체 흐름



1.3 Vue 인스턴스와 템플릿 문법

Vue 인스턴스와 템플릿은 Vue 어플리케이션의 출발점이다.

학습목표

- Vue 인스턴스의 이해
- Options API vs Composition API
- Vue3 반응성 개요
  - ref()
  - reactive()
- 템플릿 개요
- 템플릿 문법
  - {{ }} 보간법
  - v-bind 속성 바인딩
  - v-on 이벤트 바인딩
  - v-if 조건 바인딩
  - v-for 반복 바인딩

1.3 Vue 인스턴스와 템플릿 문법

Vue 인스턴스

- 개념
  - Vue 앱을 동작하게 하는 객체로서 앱의 시작점
  - createApp() 이용
- 역할
  - 반응형 데이터 관리
  - UI 업데이트
  - 이벤트 처리
  - 라이프사이클 관리

1.3 Vue 인스턴스와 템플릿 문법

생성 과정

- createApp(App) 이용
  - App.vue 를 root 컴포넌트로 Vue 인스턴스를 생성

- . mount('#app')
  - index.html 의 에 Vue 앱 장착
  -   는 Vue 가 관리하는 영역

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

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

1.3 Vue 인스턴스와 템플릿 문법

SFC (Single File Component)구조

- <template> : 화면 (UI) 설계
- <script setup> : Composition API 방식의 로직
- <style scoped> : 화면 꾸미기

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```
<!-- App.vue -->
<template>
  <section class="container">
    <h1>{{ message }}</h1>
  </section>
</template>

<script setup>
import { ref } from "vue";
const message = ref("Hello World");
</script>
```

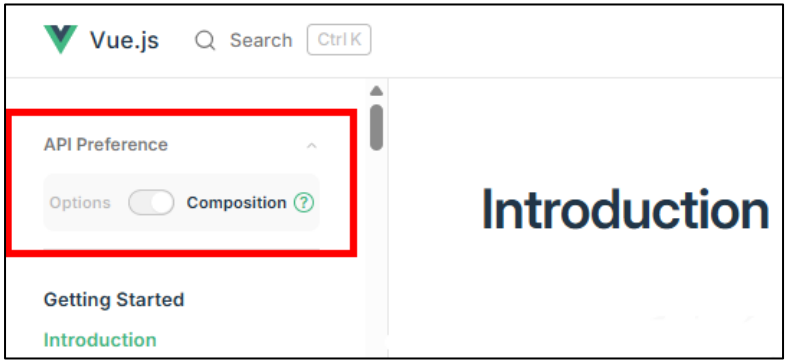
```
<style scoped>
* {box-sizing: border-box;}
html {font-family: sans-serif;}
body {margin: 0;}
.container {
  margin: 3rem auto;
  max-width: 30rem;
  border-radius: 12px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.26);
  padding: 1rem;
  text-align: center;
}
</style>
```

1.3 Vue 인스턴스와 템플릿 문법

Options API vs Composition API

- 개념
- Vue 컴포넌트를 작성하는 방법
- 결과는 같으나 코드를 구성하는 방식이 다름

- 공식 사이트
- https://vuejs.org/guide/introduction.html



1.3 Vue 인스턴스와 템플릿 문법

Options API 방식

- Vue 2 사용하던 방식
- data, methods, computed 같은 옵션(속성) 구성
- 코드 가독성 떨어짐
- 유지보수 취약

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```
<!-- App.vue -->
<template>
  <section class="container">
    <h1>{{ message }}</h1>
    <button @click="changeMessage">메시지 변경</button>
  </section>
</template>
<script>
export default {
  data() {
    return { message: "HelloWorld." };
  },
  methods: {
    changeMessage() {   this.message = "안녕하세요.";   },
  },
};
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

Composition API 방식 (권장)

- Vue 3 지원 방식
- setup 안에 관련된 데이터와 로직 기능단위로 구성
- 대규모 프로젝트
- 유지보수 최적

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```
<!-- App.vue -->
<template>
  <section class="container">
    <h1>{{ message }}</h1>
    <button @click="changeMessage">메시지 변경</button>
  </section>
</template>

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

const message = ref("HelloWorld");

function changeMessage() {
  message.value = "안녕하세요.";
}
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

Vue 3 반응성

- 개념
  - 데이터가 바뀌면 화면(UI)이 자동으로 다시 그려지는 시스템
- 구현
  - 데이터를 Proxy로 감싸서 읽힐 때 의존성을 기록
  - 이후 값이 변경되면 화면(UI) 자동 수정
- 장점
  - 직접 DOM 수정 안함
  - 데이터 중심 개발
  - 필요한 부분만 자동 수정

1.3 Vue 인스턴스와 템플릿 문법

Vue 3 반응성 핵심

- ref()
  - 숫자/문자/불리언 값을 반응형 처리
  - {{ }} 이용하여 화면 출력

- reactive()
  - 객체/배열 값을 반응형 처리

1.3 Vue 인스턴스와 템플릿 문법

ref() 함수

- 용도
  - 숫자/문자/불리언 값을 저장
- 특징
  - JavaScript 사용: .value 지정
  - Template 사용: {{ 변수 }}
- 문법
  - import { ref } from ‘vue’;
  - const 변수 = **ref**(초기값);

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```vue
<!-- App.vue -->
<template>
  <section class="container">
    <!-- template 사용시 그냥 사용 -->
    <h2>{{ uName }}</h2>
    <h2>{{ uAge }}</h2>
    <h2>{{ isMarried }}</h2>
  </section>
</template>
```

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

const uName = ref("홍길동");
const uAge = ref(31);
const isMarried = ref(false);

//JS 사용시 .value 지정
console.log(uName.value);
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

reactive() 함수

- 용도
  - 객체/배열 값을 저장
- 특징
  - JavaScript 사용: .value 없이 그냥 사용
  - Template 사용: {{ 변수.key }}
  - 중첩 속성까지 반응성 처리
- 문법
  - import { reactive } from 'vue';
  - const 변수 = **reactive**({key:value});

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```vue
<!-- App.vue -->
<template>
  <section class="container">
    <!-- template 사용시 그냥 사용 -->
    <h2>{{ user.name }}</h2>
    <h2>{{ user.age }}</h2>
    <h2>{{ user.isMarried }}</h2>
  </section>
</template>
```

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

const user = reactive({
  name: "홍길동",
  age: 31,
  isMarried: false,
});

//JS 사용시 .value 없이 사용
console.log(user.name);
console.log(user.age);
console.log(user.isMarried);
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

템플릿 개요

- 개념
  - 템플릿 = HTML + Vue 문법
  - Vue 컴포넌트의 화면 (UI) 정의
  - <template> 태그안에서 작성

- 문법
  - 보간법 (interpolation): {{변수}}
  - 속성 바인딩: v-bind
  - 조건/반복 : v-if/v-else/v-else-if, v-for
  - 이벤트 처리: v-on
  - 양방향 바인딩: v-model

1.3 Vue 인스턴스와 템플릿 문법

보간법 (interpolation)

- 용도
  - 데이터를 화면에 보여주는 방법
  - 문법: {{ }} 이중 중괄호 표현

- 특징
  - 문자열/숫자/불린/배열/객체
  - 연산 가능 (산술,비교,3항)
  - 함수 호출 가능
  - 대체 기능
  - v-text : HTML 무시 , 텍스트만 출력
  - v-html: HTML 적용

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```vue
<!-- App.vue -->
<template>
  <section class="container">
    <h2>{{ uName }}</h2>
    <h2>{{ uAge + 1 }}</h2>
    <h2>{{ uAge > 30 }}</h2>
    <h2>{{ uAge > 18 ? "성인" : "미성년" }}</h2>
    <h2>{{ uName.toUpperCase() }}</h2>
    <h2>{{ formatDate() }}</h2>
    <div v-text="message"></div>
    <div v-html="message"></div>
  </section>
</template>
```

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

const uName = ref("Hong gil Dong");
const uAge = ref(31);
const message = ref("<b>안녕하세요</b>");

// 사용자 정의 함수
function formatDate() {
  return new Date().toISOString();
}
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

속성 바인딩

- 용도
  - Vue 데이터와 HTML 속성값을 연결
  - 데이터가 바뀌면 태그 속성값도 변경
  - 문법: v-bind: 또는 : (콜론)

- 특징
  - : (콜론) 주로 사용
  - class, style 에도 자주 사용
  - 객체사용시 한꺼번에 바인딩 가능

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```vue
<!-- App.vue -->
<template>
  <section class="container">
    <h1>{{ title }}</h1>
    <img v-bind:src="imageUrl" v-bind:alt="altText" />
    <a :href="link">여기로 이동</a>
    <!-- 한번에 바인딩 -->
    <img v-bind="imgAttrs" />
    <hr />
  </section>
</template>
```

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

const title = ref("Vue 속성 바인딩 예제");
// 무료 샘플 이미지 제공 사이트
const imageUrl = ref("https://i.pravatar.cc/300");
const altText = ref("샘플 랜덤 이미지");
const link = ref("https://vuejs.org");

//한번에 바인딩
const imgAttrs = reactive({
  src: "https://i.pravatar.cc/300",
  alt: "한 번에 바인딩된 이미지",
  width: "200",
});
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

이벤트 바인딩

- 용도
  - 이벤트 감지/실행 동작 연결
  - 문법: v-on: 또는 @ (축약표현)

- 특징
  - @ 주로 사용
  - 이벤트 수식어(modifier) 지원
    (prevent, enter, stop)
  - 이벤트 처리는 일반 함수 이용
  - 이벤트 객체 명시적 전달 ($event 이용)

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```vue
<!-- App.vue -->
<template>
  <header>
    <h1>Vue 3 이벤트 실습</h1>
  </header>
  <section id="events">
    <p>결과: {{ counter }}</p>
    <button v-on:click="up">+</button>
    <button @:click="down">-</button>
  </section>
</template>
```

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

const counter = ref(0);

//이벤트 처리
function up() {
  counter.value = counter.value + 1;
}
function down() {
  counter.value = counter.value - 1;
}
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

조건부 랜더링

- 용도
  - 조건에 따라서 화면에 보이거나 숨김
  - 문법: v-if="조건식", v-else, v-else-if

- 특징
  - false : 실제 DOM에서도 생성 안됨
  - v-if 와 v-else 는 바로 연결 해야 됨
  - 조건식은 숫자/문자열/비어있음 가능

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```
<!-- App.vue -->
<template>
  <header>
    <h1>Vue 3 조건부 랜더링 실습</h1>
  </header>
  <section id="user-goals">
    <h2>나의 나이</h2>
    <input type="number" @input="setAge" />
    <p v-if="age <= 0 || age >= 100">당신의 나이를 입력</p>
    <ul>
      <li v-if="age < 0">입력오류</li>
      <li v-else-if="age < 8">미취학</li>
      <li v-else-if="age < 19">미성년</li>
      <li v-else>성년</li>
    </ul>
  </section>
</template>
```

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

const age = ref(0);
console.log(age);

function setAge(event) {
  age.value = event.target.value;
}
</script>
```

1.3 Vue 인스턴스와 템플릿 문법

반복 랜더링

- 용도
  - 배열값을 반복해서 출력
  - 문법: v-for="변수 in 배열"

- 특징
  - 인덱스(index) 사용
  - 객체 출력
  - 숫자 반복
  - : key 속성 필수 (유일값)

1.3 Vue 인스턴스와 템플릿 문법

구현 코드

```vue
<!-- App.vue -->
<template>
  <header>
    <h1>Vue 과정 목표</h1>
  </header>
  <section id="user-goals">
    <h2>나의 목표</h2>
    <input type="text" @input="setGoal" />
    <button @click="addGoal">추가</button>
    <p v-if="goals.length === 0">목표를 추가하세요!</p>
    <ul v-else>
      <li v-for="goal in goals" :key="goal">{{ goal }}</li>
    </ul>
  </section>
</template>
```

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

const enteredGoalValue = ref("");
const goals = reactive([]);

function setGoal(event) {
  enteredGoalValue.value = event.target.value;
}

function addGoal() {
  goals.push(enteredGoalValue.value);
}
console.log(goals);
</script>
```

1.4 데이터 바인딩과 디렉티브

데이터 바인딩을 활용하여 데이터와 화면을 자동으로 연결한다.

학습목표

- ◆ 데이터 바인딩 이해
  - ▪ 양방향 바인딩
  - ▪ 스타일 바인딩
  - ▪ 클래스 바인딩
- ◆ 디렉티브 개요
- ◆ computed 함수와 바인딩
- ◆ watch 함수와 바인딩

1.4 데이터 바인딩과 디렉티브

데이터 바인딩

- 개념 : JS의 데이터와 화면(UI)을 자동으로 연결
- 특징
  - 데이터가 변경되면 자동으로 화면도 갱신
  - 전체 화면이 아닌 변경할 부분만 갱신
  - 디렉티브(directive) 기반
- 종류
  - 단방향 / 양방향 바인딩
  - 속성 바인딩
  - 이벤트 바인딩
  - 클래스/스타일 바인딩

1.4 데이터 바인딩과 디렉티브

디렉티브 (Directive)

- 개념 : Vue 템플릿에서 v- 로 시작하는 특수 속성
- 역할
  - 데이터 바인딩 처리
  - 이벤트 처리
  - 조건/반복 처리
  - 클래스/스타일 처리
- 핵심
  - v-model, v-bind
  - v-on
  - v-if, v-for

1.4 데이터 바인딩과 디렉티브

양방향 바인딩 (Two-way binding)

- 개념
  - 데이터와 화면(UI)이 서로 연결
- 문법: v-model
- 특징
  - 데이터가 변경되면 화면(UI)에 반영
  - 화면(UI)이 변경되면 데이터에 반영
  - 수식어(modifier) 지원 (trim, number)
- 용도
  - 폼 입력 처리 / 실시간 미리 보기

1.4 데이터 바인딩과 디렉티브

구현 코드

```
<!-- App.vue -->
<template>
  <header>
    <h1>Vue 과정 목표</h1>
  </header>
  <section id="user-goals">
    <h2>나의 목표</h2>
    <input type="text" v-model.trim="enteredGoalValue" />
    <button @click="addGoal">추가</button>
    <p v-if="goals.length === 0">목표를 추가하세요!</p>
    <ul v-else>
      <li v-for="goal in goals" :key="goal">{{ goal }}</li>
    </ul>
  </section>
</template>
```

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

const enteredGoalValue = ref("");
const goals = reactive([]);

function addGoal() {
  goals.push(enteredGoalValue.value);
}
console.log(goals);
</script>
```

1.4 데이터 바인딩과 디렉티브

computed 함수 바인딩

- 개념
  - 데이터가 변경될 때만 다시 계산
  - 이외에는 캐시된 값을 재사용
- 문법
  - const 변수 = computed(함수);
- 용도
  - 의존 데이터가 바뀔 때만 자동으로 다시 계산
  - 매번 계산하기 비싼 작업을 최소화
  - ex. 상품 총합

1.4 데이터 바인딩과 디렉티브

구현 코드

```
<!-- App.vue -->
<template>
  <div>
    <ul>
      <li v-for="(item, idx) in cart" :key="idx">
        {{ item.name }}: {{ item.price }} x {{ item.qty }}
      </li>
    </ul>
    <hr />
    <p>소계: {{ subtotal.toLocaleString() }}원</p>
    <p>부가세: {{ vat.toLocaleString() }}원</p>
    <p>
      <strong>총합: {{ formattedTotal }}</strong>
    </p>
  </div>
</template>
```

1.4 데이터 바인딩과 디렉티브

구현 코드

```
<script setup>
import { reactive, computed } from "vue";

const cart = reactive([
  { name: "사과", price: 1200, qty: 2 }, { name: "배", price: 2000, qty: 1 }
]);

//소계
const subtotal = computed(() =>
  cart.reduce((sum, item) => sum + item.price * item.qty, 0)
);
//부가세
const vat = computed(() => Math.floor(subtotal.value * 0.1));
//총합
const total = computed(() => subtotal.value + vat.value);
//총합 포맷
const formattedTotal = computed(() =>
  total.value.toLocaleString("ko-KR", { style: "currency", currency: "KRW" })
);
</script>
```

1.4 데이터 바인딩과 디렉티브

watch 함수 바인딩

- 개념
  - 변수가 바뀌면 특정 동작을 실행
  - 부수효과 (side effect)
- 문법
  - watch(변수, (new, old)=>{ });
- 용도
  - 콘솔에 로그 출력
  - 알림
  - 특정 값이 변경되면 작업 실행

1.4 데이터 바인딩과 디렉티브

구현 코드

```vue
<!-- App.vue -->
<template>
  <header>
    <h1>Vue 3 이벤트 실습</h1>
  </header>
  <section id="events">
    <p>결과: {{ counter }}</p>
    <button v-on:click="up">+</button>
    <button @:click="down">-</button>
  </section>
</template>
```

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

const counter = ref(0);

function up() {
  counter.value = counter.value + 1;
}
function down() {
  counter.value = counter.value - 1;
}
// counter 값 변경 로그
watch(counter, (newValue, oldValue)=>{
  console.log(`값이 ${oldValue}에서 ${newValue}로 바뀜`)
});
</script>
```

1.4 데이터 바인딩과 디렉티브

style 바인딩

- 개념 : HTML의 style 속성을 데이터로 연결
- 문법
  - 객체 이용
  - 배열 + computed 바인딩 이용
- 사용예
  - <div :style="{ color: textColor }">
  - <div :style="[baseStyle, hoverStyle]">

1.4 데이터 바인딩과 디렉티브

구현 코드

```vue
<!-- App.vue -->
<template>
  <section id="user-goals">
    <h2>스타일</h2>
    <select v-model="textColor">
      <option value="blue" selected>파랑</option>
      <option value="red">빨강</option>
    </select>
    <select v-model="fontSize">
      <option value="16">16px</option>
      <option value="20" selected>20px</option>
    </select>
    <p :style="{ color: textColor, fontSize: fontSize + 'px' }">
      안녕하세요! 글자 색상과 크기를 바꿔보세요.
    </p>
    <p :style="[baseStyle, baseStyle2]">
      안녕하세요! 글자 색상과 크기를 바꿔보세요.
    </p>
  </section>
</template>
```

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

const textColor = ref("blue");
const fontSize = ref(20);
console.log(textColor, fontSize);

// 배열 형식 + computed 바인딩
const baseStyle = computed(() => ({
  color: textColor.value,
  fontSize: fontSize.value + "px",
}));
// 배열 형식 + computed 바인딩
const baseStyle2 = computed(() => ({
  color: textColor.value,
  fontSize: fontSize.value + "px",
}));
</script>
```

1.4 데이터 바인딩과 디렉티브

class 바인딩

- 개념 : HTML의 class 속성을 데이터로 연결
- 문법
  - 문자열 이용
  - 객체 이용
  - 배열 이용
  - 정적 + 동적
- 사용예
  - <div :class="{ active: isActive, error: hasError }">내용
  - <div class="card" :class="{ active: isActive }">내용
  - <div :class="[baseClass, sizeClass, { active: isActive }]">내용

1.4 데이터 바인딩과 디렉티브

구현 코드

```
<!-- App.vue -->
<template>
  <header>
    <h1>Vue 3 클래스 바인딩 실습</h1>
  </header>
  <section id="styling">
    <!-- boxASelected 가 true면 active 클래스가 적용됨 -->
    <div   class="demo“ :class="{ active: boxASelected }"
      @click="boxSelected('A')"></div>
    <div   class="demo“ :class="{ active: boxBSelected }"
      @click="boxSelected(‘B’)”></div>
    <div   class=“demo“ :class=“{ active: boxCSelected }”
      @click=“boxSelected(‘C')"></div>
  </section>
</template>

<style>
  .active {
       border-color: purple;   background-color: purple;
   }
</style>
```

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

const boxASelected = ref(false);
const boxBSelected = ref(false);
const boxCSelected = ref(false);

console.log(boxASelected, boxBSelected, boxCSelected);

function boxSelected(box) {
  if (box === "A") {
    boxASelected.value = !boxASelected.value;
  } else if (box === "B") {
    boxBSelected.value = !boxBSelected.value;
  } else if (box === "C") {
    boxCSelected.value = !boxCSelected.value;
  }
}
</script>
```

1.5 정리 및 QnA

요약맵

- Vue 컴포넌트/MVVM 패턴/SPA/CSR
- Composition API 기본 문법
- 디렉티브 (보간법, v-bind, v-model, v-on, v-if, v-for)
- 반응성 (ref, reactive, computed, watch)
- 양방향 바인딩
- 클래스 및 스타일 바인딩

2. Vue.js 핵심 기능

- 컴포넌트 기초
- Props 와 Events
- 조건부 랜더링과 리스트 랜더링 심화
- 폼 입력 바인딩
- 정리 및 실습

2.1 컴포넌트 기초

SFC 기반의 컴포넌트 개념과 전역/지역/동적 컴포넌트에 관하여 살펴보자

학습목표

- 컴포넌트와 SFC 구조 이해
- 전역 및 로컬 컴포넌트 이해
- 동적 컴포넌트 (Dynamic Component)
- Scoped style 개요

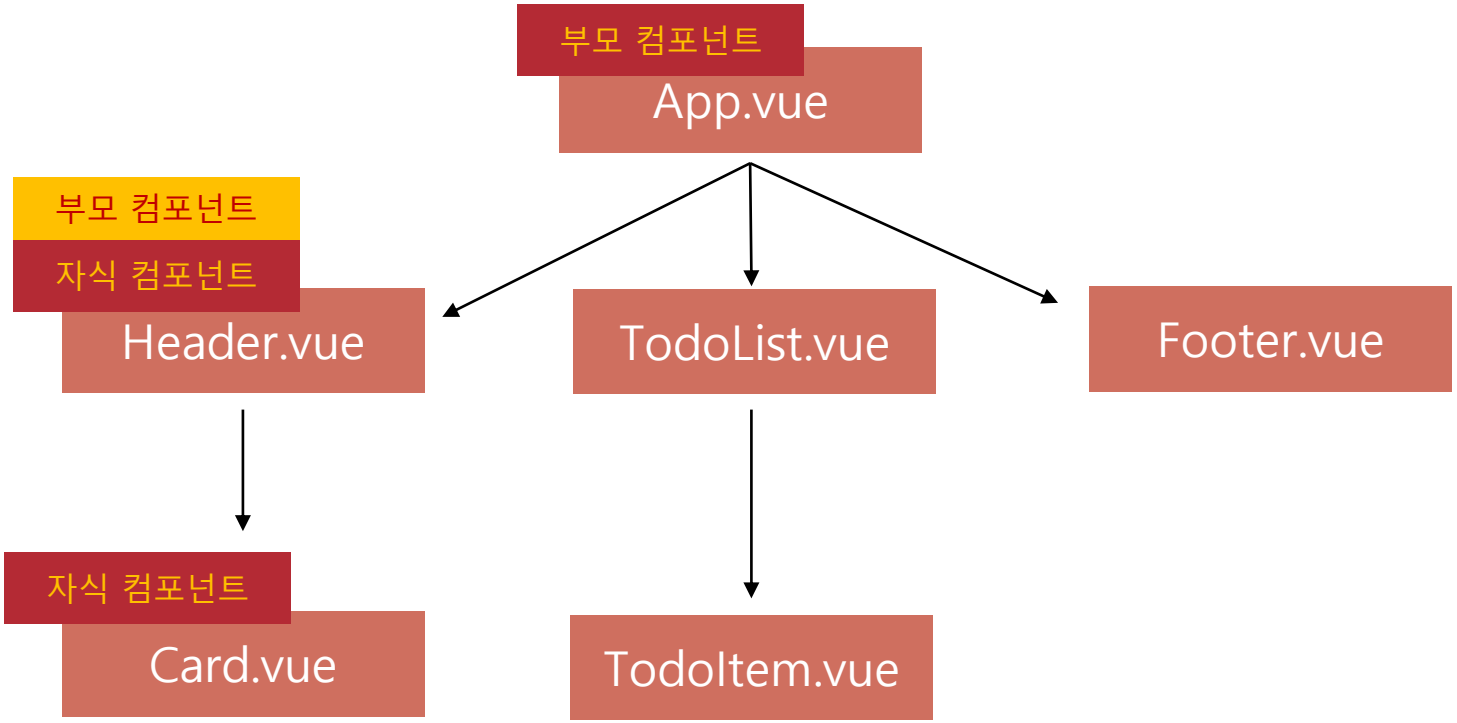
2.1 컴포넌트 기초

컴포넌트 개요

- 개념
  - 화면을 이루는 독립 단위
  - 재사용/유지보수 탁월
- 구현
  - SFC (Single File Component)
  - .vue 파일
- 특징
  - 하나의 파일은 하나의 역할 (단일 책임 원칙)
  - 컴포넌트 트리(부모/자식 관계) 구성
  - 작은 컴포넌트를 조합하여 화면 구성

2.1 컴포넌트 기초

컴포넌트 구조 예



2.1 컴포넌트 기초

SFC (Single File Component)

- 개념
  - 컴포넌트를 .vue 단일 파일로 구조화
- 구성요소
  - <template>
  - <script **setup**>
  - <style **scoped**>
- 동작방식
  - Vue CLI 또는 Vite 이용
  - .vue를 컴파일하여 일반 JS로 변환
  - 이후 웹 브라우저에서 실행

2.1 컴포넌트 기초

컴포넌트 등록

- 개념
  - 새로 만든 SFC 기반 컴포넌트를 사용하기 위한 방법
- 방법
  - 전역(Global)등록 : 여러 컴포넌트에서 공유해서 사용
  - 지역(Local)등록 : 해당 컴포넌트내에서만 사용
- 명명법
  - 파일명: PascalCase
    - ex. StopWatch.vue
  - 템플릿: PascalCase 또는 kebab-case
    - ex. <StopWatch> ,<stop-watch>

2.1 컴포넌트 기초

전역 컴포넌트 (Global)

- 개념
  - 여러 컴포넌트에서 공유할 목적
- 문법
  - main.js 에서 등록
  - app.component("컴포넌트명", 컴포넌트);
    ex. app.component("StopWatch", StopWatch)
    app.component("stop-watch", StopWatch)
- 용도
  - 공통 버튼/입력 같은 Base 컴포넌트에 적합

2.1 컴포넌트 기초

구현 코드

```
// main.js
import { createApp } from "vue";
import App from "./App.vue";
import SimpleHeader from "./components/SimpleHeader.vue";
import SimpleFooter from "./components/SimpleFooter.vue";

const app = createApp(App);

// 전역 컴포넌트 등록
app.component("SimpleHeader", SimpleHeader); // PascalCase
app.component("simple-footer", SimpleFooter);   // kebab-case

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

```
<!-- App.vue -->
<template>
  <main class="container">
    <SimpleHeader></SimpleHeader>
    <!-- 본문 -->
    <section class="body">
      <p>본문 내용입니다.</p>
      <ul>
        <li>할 일 1</li>
        <li>할 일 2</li>
        <li>할 일 3</li>
      </ul>
    </section>
    <simple-footer></simple-footer>
  </main>
</template>
<script setup></script>
```

2.1 컴포넌트 기초

지역 컴포넌트 (Local)

- 개념
  - 해당 컴포넌트에서만 사용할 목적
- 문법
  - 해당 SFC 에서 import 하면 바로 사용가능
- 특징
  - 템플릿에서는 PascalCase 와 kebab-case 모두 사용 가능
  - 일반적으로 권장하는 방법

2.1 컴포넌트 기초

구현 코드

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

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

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

```
<!-- App.vue -->
<template>
  <main class="container">
    <SimpleHeader></SimpleHeader>
    <section class="body">
      <p>본문 내용입니다.</p>
      <ul>
        <li>할 일 1</li>
        <li>할 일 2</li>
        <li>할 일 3</li>
      </ul>
    </section>
    <!-- PascalCase 와 kebob-case 모두 가능 -->
    <simple-footer></simple-footer>
  </main>
</template>
```

2.1 컴포넌트 기초

동적 컴포넌트 (Dynamic component)

- 개념
  - 랜더링 되는 컴포넌트를 동적으로 변경
- 문법
  - <component :is="컴포넌트명" />
- 특징
  - is 속성값의 컴포넌트가 화면에 랜더링
  - 화면전환이 되면 내부 상태가 초기화됨.
  - 상태보존 : <keep-alive>

2.1 컴포넌트 기초

동적 컴포넌트 실습 화면

- <keep-alive> 적용하여 UserCard의 count 값을 유지

UserCard.vue 화면

ProductList.vue 화면

2.1 컴포넌트 기초

구현 코드

```
<!-- App.vue -->
<template>
  <div class="container">
    <h1>동적 컴포넌트 연습</h1>
    <div class="tabs">
      <button class="tab":class="{ active: currentName === 'UserCard' }"
        @click="currentName = 'UserCard'">UserCard
      </button>
      <button class="tab":class="{ active: currentName === 'ProductList' }"
        @click="currentName = 'ProductList'">ProductList
      </button>
    </div>
    <div>
      <keep-alive>
        <component :is="currentComp" class="panel" />
      </keep-alive>
    </div>
  </div>
</template>
```

2.1 컴포넌트 기초

구현 코드

```
<!-- App.vue -->
<script setup>
import { ref, computed } from "vue";
import UserCard from "./components/UserCard.vue";
import ProductList from "./components/ProductList.vue";

const currentName = ref("UserCard");
const comps = { UserCard, ProductList };
const currentComp = computed(() => comps[currentName.value]);
</script>


<!- ProductList.vue -->
<template>
  <div class="card">
    <h2>상품 목록</h2>
    <ul class="list">
      <li v-for="product in products" :key="product">{{ product }}</li>
    </ul>
  </div>
</template>
```

2.1 컴포넌트 기초

구현 코드

```
<!- ProductList.vue -->
<script setup>
import { reactive } from "vue";
const products = reactive(["책", "키보드", "마우스", "모니터"]);
</script>


<!- UserCard.vue -->
<template>
  <div class="card">
    <h2>사용자 카드</h2>
    <p>
      카운터: <strong>{{ count }}</strong>
    </p>
    <button class="btn" @click="up">+</button>
  </div>
</template>
<script setup>
import { ref } from "vue";
const count = ref(0);
function up() {
  count.value = count.value + 1;
}
</script>
```

2.1 컴포넌트 기초

Scoped Style 개요

- 개념
  - SFC 에서 해당 컴포넌트 템플릿에만 CSS 적용
  - 런타임시 템플릿 각 요소에 data-v-xxx 형식의 고유 속성이 설정

- 문법
  - <style scoped> : 해당 컴포넌트 내부에만 적용
  - <style> : 전역 스타일, 앱 전체에 적용

- 권장 가이드
  - 전역 스타일은 App.vue 또는 main.js의 base.css 한 곳에서 설정
  - 나머지는 모두 <style scoped> 설정
  - 자식 내부 접근: :deep (과사용 금지)

2.2 Props와 Events

컴포넌트 간의 통신 방법인 props 및 emit, slot 방법을 활용하여 Vue어플리케이션을 구축한다.

학습목표

- 컴포넌트 데이터 흐름 이해
- props 개요
- emit 개요
- slot 개요

2.2 Props와 Events

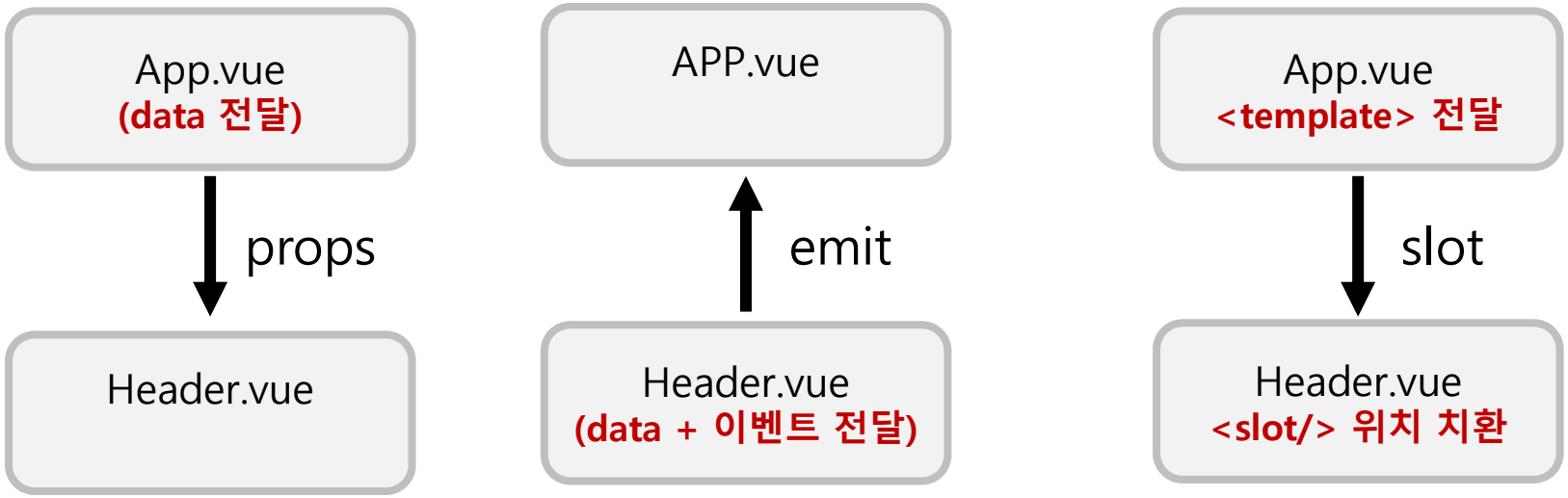
컴포넌트 데이터 흐름

- 개념
  - 컴포넌트간 데이터 전달 방식

- 방법
  - props : 부모에서 자식으로 데이터 전달
  - emit : 자식에서 부모로 데이터 전달
  - slot : 부모에서 자식으로 template 전달

2.2 Props와 Events

컴포넌트 데이터 흐름 구조도



2.2 Props와 Events

props 개요

- 개념
  - 부모에서 자식으로 데이터 전달 (읽기 모드)
- 방법
  - 부모 컴포넌트: 자식 컴포넌트 사용시 속성값 전달
  - 자식 컴포넌트: defineProps(배열) , defineProps(옵션객체)
- 명명법
  - <template> : kebab-case 또는 camelCase
    ex. <Child **my-property**="" **myProperty**="" />
  - <script>: camelCase
    ex. const myProps = defineProps(['**myProperty**']);

2.2 Props와 Events

defineProps 함수

- 개념
  - props 값을 정의/타입지정/필수여부/기본값/유효성검사

- 방법-1
  - 배열: 속성 이름만 나열

    ex. defineProps(['title','count'])

- 방법-2
  - 옵션 객체: 타입지정/필수여부/기본값/유효성검사

    ex. defineProps ({title: { **type**:String, **required**:true} })

    defineProps ({count: { type:Number, **default**: 0} })

    defineProps ({isFavorite: { ..., **validator**:function(val){ return val==='0' || val ==='1' }}})

2.2 Props와 Events

구현 코드

```vue
<!-- App.vue -->
<template>
  <main class="container">
    <h1>defineProps 실습</h1>

    <section class="card">
      <h2>1) InputLabel-배열처리</h2>
      <InputLabel label="이름" placeholder="홍길동" />
      <InputLabel label="이메일" placeholder="user@example.com" />
    </section>

    <section class="card">
      <h2>2) UserCard -옵션처리(런타임 검증 + 기본값 + validator)</h2>
      <UserCard title="홍길동" :count="3" :is-active="true" size="md" />
      <UserCard title="임꺽정" size="lg" />
    </section>
  </main>
</template>

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

```vue
<!-- InputLabel.vue -->
<template>
  <label class="box">
    <span class="label">{{ label }}</span>
    <input class="input"
           :placeholder="placeholder" />
  </label>
</template>

<script setup>
import { defineProps } from "vue";

defineProps(["label", "placeholder"]);
</script>
```

```vue
<!-- UserCard.vue -->
<script setup>
import { defineProps } from "vue";
defineProps({
  title: { type: String, required: true },
  count: { type: Number, default: 0 },
  isActive: { type: Boolean, default: false },
  size: {
    type: String,
    default: "md",
    validator: (v) => ["sm", "md", "lg"].includes(v),
  },
});
</script>
```

2.2 Props와 Events

emit 개요

- 개념
  - 자식에서 부모로 데이터 전달
  - 사용자 정의 이벤트 이용

- 방법
  - 자식 컴포넌트:
    1) 이벤트 이름 선언: const emit = defineEmit(['이벤트명1','이벤트명2'])
    2) 이벤트 발생: emit ('이벤트명1', 전달값)
    script에서 이벤트명은 camelCase 권장

- 부모 컴포넌트:
  <자식컴포넌트 @이벤트명1="이벤트처리함수" />
  template에서 이벤트명은 kebab-case 권장

2.2 Props와 Events

구현 코드

```vue
<!-- App.vue -->
<template>
  <main class="wrap">
    <h1>emit 실습</h1>
    <p class="count">
      현재 값: <strong>{{ count }}</strong>
    </p>
    <!-- 자식이 전달하는 add 이벤트를 듣고, count를 증가 -->
    <CounterButton @add="inc" />
  </main>
</template>

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

```vue
<!-- CounterButton.vue -->
<template>
  <button class="btn" @click="addOne">+1 증가</button>
</template>

<script setup>
import { defineEmits } from "vue";
const emit = defineEmits(["add"]); // 부모에게 'add' 이벤트 알림

function addOne() {
  emit("add", 1); // 페이로드로 1 전달
}
</script>
```

2.2 Props와 Events

slot 개요

- 개념
  - 부모에서 자식으로 template 전달
  - slot은 화면(template) 전달해서 최종적인 조립 목적
- 용도
  - 각 컴포넌트마다 공통적으로 사용할 수 있는 template이 존재하는 경우
  - 부모에서 자식 template 관리하려는 경우
- 종류
  - Default 및 Named slot: <slot />, <slot name="header" />
  - Scoped slot: <slot :msg="goal" >
  - Fallback content: <slot>폴백 내용</slot>

2.2 Props와 Events

Default slot

- 개념
  - 가장 기본이 되는 slot
  - 부모 template이 자식 slot 위치에 치환

- 방법
  - 자식 컴포넌트:
    <slot /> 으로 template 내용이 적용될 위치 지정

  - 부모 컴포넌트:
    <child> template 내용</child>
    지정된 template 내용은 자식 <slot /> 위치에 채움

2.2 Props와 Events

구현 코드

```vue
<!-- App.vue -->
<template>
  <main class="wrap">
    <section>
      <h2>1) 기본 슬롯 (Default Slot)</h2>
      <DefaultSlotBox>
        <p>부모가 기본 슬롯에 넣은 콘텐츠입니다.</p>
        <ul>
          <li v-for="p in people" :key="p.id">
            {{ p.name }}
          </li>
        </ul>
      </DefaultSlotBox>
    </section>
  </main>
</template>
```

```vue
<!-- DefaultSlotBox.vue -->
<template>
  <div class="box">
    <!-- 부모가 여기 사이를 채웁니다 -->
    <slot />
  </div>
</template>
```

2.2 Props와 Events

Named slot

- 개념
  - 이름이 있는 slot
  - 부모 template이 자식 slot 이름과 일치하는 위치에 치환

- 방법
  - 자식 컴포넌트:
    <slot **name="header"** /> 으로 template 내용이 적용될 위치 지정

  - 부모 컴포넌트:
    <child>
    <template **#header**>template 내용 </template>
    </child>
    지정된 template 내용은 name과 일치하는 자식 <slot /> 위치에 채움

2.2 Props와 Events

구현 코드

```vue
<!-- App.vue -->
<template>
  <main class="wrap">
    <h1>Vue3 Named slot 실습</h1>
    <!-- 2) 이름 있는 슬롯 -->
    <section>
      <h2>2) 이름 있는 슬롯 (Named Slots)</h2>
      <NamedSlotCard>
        <template #header>헤더를 부모가 채웠어요 </template>

        <p>여기는 기본 슬롯(본문)입니다. 부모가 자유롭게 마크업을 지정</p>

        <template #footer>
          <button class="btn">확인</button>
        </template>
      </NamedSlotCard>
    </section>
  </main>
</template>
```

```vue
<!-- NamedSlotCard.vue -->
<template>
  <article class="card">
    <header class="card__header">
      <slot name="header" />
    </header>

    <section class="card__body">
      <!-- 기본 슬롯(본문) -->
      <slot />
    </section>

    <footer class="card__footer">
      <slot name="footer" />
    </footer>
  </article>
</template>
```

2.2 Props와 Events

Scoped slot

- 개념
  - 자식이 데이터를 slot 속성 이용하여 부모에게 전달
  - 이후 부모가 받아서 처리 후 랜더링

- 방법
  - 자식 컴포넌트:
    <slot **:msg="goal"** /> 방식으로 부모에게 데이터 전달

  - 부모 컴포넌트:
    <child>
    <template **#default="{msg}"** >msg + template 내용 </template>
    </child>
    부모에서는 #default 속성으로 자식 데이터를 받음

2.2 Props와 Events

구현 코드

```vue
<!-- App.vue -->
<template>
  <main class="wrap">
    <h1>Vue3 Scoped slot 실습</h1>
    <section>
      <ScopedSlotList>
        <template #default="{ item, anotherProps }">
          <h3>{{ item.name }}</h3>
          <p>{{ anotherProps }}</p>
        </template>
      </ScopedSlotList>
    </section>
  </main>
</template>
```

```vue
<!-- ScopedSlotList.vue -->
<template>
  <ul class="list">
    <li v-for="person in people" :key="person.id">
      <!-- 부모에게 넘겨줄 슬롯 props -->
      <slot :item="person"
            another-props="추가 설명 텍스트"></slot>
    </li>
  </ul>
</template>
```

2.2 Props와 Events

Fallback content

- 개념
  - 부모에서 자식에게 아무것도 전달하지 않는 경우
  - 기본적으로 보여줄 컨텐츠(UI) 지정

- 방법
  - 자식 컴포넌트:
    <slot>기본컨텐츠(UI)</slot>

  - 부모 컴포넌트:
    <child />
    <child>컨텐츠UI</child>

2.2 Props와 Events

구현 코드

```vue
<!-- App.vue -->
<template>
  <main class="wrap">
    <section>
      <h2>4) 폴백 콘텐츠 (Fallback)</h2>

      <p>아무것도 안 채운 경우(폴백 표시):</p>
      <FallbackSlotPanel />
      <hr />
      <p>부모가 채운 경우(폴백 대신 사용자 콘텐츠):</p>
      <FallbackSlotPanel>
        커스텀 본문. 폴백이 아닌 이 내용이 보입니다.
      </FallbackSlotPanel>
    </section>
  </main>
</template>
```

```vue
<!-- FallbackSlotPanel.vue -->
<template>
  <div class="box">
    <!-- 부모가 여기 사이를 채웁니다 -->
    <slot>커스텀 본문. 폴백 내용이 보입니다.</slot>
  </div>
</template>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

조건 및 반복 처리 심화 학습을 통하여 성능 향상을 기대할 수 있다.

학습목표

- ◆ 조건부 랜더링 기초 및 심화
  - v-if
  - v-show, v-once
  - 조건 + computed
- ◆ 리스트 랜더링 기초 및 심화
  - v-for
  - :key="유일값"
  - 필터링/정렬 + computed

2.3 조건부 랜더링과 리스트 랜더링 심화

조건부 랜더링

- 용도
  - 조건에 따라서 화면에 보이거나 숨김
  - 문법: v-if="조건식"
    v-show="조건식"
    v-once
- 특징
  - v-if
    - DOM 자체 제거/비용 큼
    - 크롬 : <!—v-if --> 마킹
  - v-show
    - DOM은 남기고 CSS 로 숨김
  - v-once
    - 한 번만 랜더링

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드

```vue
<!-- App.vue -->
<template>
  <div class="wrap">
    <h1>v-if · v-show · v-once 빠른 비교</h1>

    <!-- v-if: 조건이 false면 DOM 자체가 없어짐 -->
    <section class="panel">
      <h2>v-if (DOM 생성/제거)</h2>
      <label class="toggle">
        <input type="checkbox" v-model="showIf" />
        <span>박스 보이기</span>
      </label>

      <div v-if="showIf" class="box if-box">
        <strong>v-if 박스</strong>
        <p>체크 해제하면 DOM에서 완전히 사라짐.</p>
      </div>
    </section>
```

```vue
<!-- App.vue -->
    <!-- v-show: DOM은 유지, display: none 으로 숨김 -->
    <section class="panel">
      <h2>v-show (CSS로 숨김)</h2>
      <label class="toggle">
        <input type="checkbox" v-model="showShow" />
        <span>박스 보이기</span>
      </label>
      <div v-show="showShow" class="box show-box">
        <strong>v-show 박스</strong>
        <p>체크 해제해도 DOM은 그대로 있고 화면만 숨김.</p>
      </div>
    </section>

    <!-- v-once: 최초 렌더만, 이후 반응형 갱신 안 됨 -->
    <section class="panel">
      <h2>v-once (한 번만 렌더)</h2>
      <p>v-once 미지정:{{ num }}</p>
      <p v-once>v-once 지정:{{ num }}</p>
      <button class="btn" @click="num++">+</button>
    </section>
  </div>
</template>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

조건부 + computed

- 용도
  - 데이터 변경 사항을 조건 체크할 때
    - 같은 결과값을 2번 이상 사용. Ex. 버튼 비활성화 등
    - 여러 상태에 의존해서 결과가 바뀜. Ex. 권한, 폼유효성

- 특징
  - 의존 값이 변할 때만 다시 계산 (미사용시 랜더링 마다 재계산)
  - 코드 재사용 가능 (미사용시 코드 중복)
  - 로직을 한 곳에 모아서 테스트/유지 보수 용이
  - 템플릿에 로직 최소화

2.3 조건부 랜더링과 리스트 랜더링 심화

조건부 + computed 사용

```
<!- computed에서  계산 -->
<button type="submit" :disabled="!canSubmit">제출</button>
      <p v-if="!canSubmit" class="warn">이름 및 이메일을 확인
</p>
```

코드 재사용

```
/ 이메일 형식 단순 체크. 데이터가 변경되는 경우에만 실행됨.
const emailOk = computed(() => {
  console.log("App.vue 호출");
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email.value);
});
// 핵심: computed에서 데이터 조건체크.
const canSubmit = computed(() => name.value.trim() && emailOk.value);
```

값 변경될 때만 호출

조건부 + computed 미사용

```
<!-- computed 없이 템플릿에서 직접 계산 -->
      <button type="submit"
              :disabled="!(name.trim() && emailValid(email))">제출
      </button>

      <p v-if="!(name.trim() && emailValid(email))"
               class="warn"> 이름 및 이메일을 확인</p>
```

코드 중복

```
// 단순 이메일 검사 함수(렌더 때마다 호출됨)
const emailValid = (v) => {
  console.log("App.vue 호출");
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(v);
};
```

매번 호출

2.3 조건부 랜더링과 리스트 랜더링 심화

리스트 랜더링

- 용도
  - 배열/객체 반복해서 출력
  - 문법:

    배열: v-for="(val, idx) in arr" :key="고유값"

    객체: v-for="(val, key) in obj" :key="고유값"

- 특징
  - DOM에서 기존 목록은 재사용
  - 인덱스(index)를 key로 사용시 재정렬/삭제 이슈
  - 중첩 v-for 사용시 <template> 사용
  - 필터링/정렬 기능은 computed 사용

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드 (배열)

```
<!- App.vue-->
<template>
  <main class="container">
    <h1>v-for 기본 실습 (배열)</h1>
    <section class="card">
      <h2>배열 순회</h2>
      <ul class="list">
        <!-- (item, idx) in fruits  /  :key에는 고유 id 사용 -->
        <li v-for="(item, idx) in fruits" :key="item.id" class="list-item">
          <span class="badge">{{ idx + 1 }}</span>
          {{ item.name }} <small class="muted">({{ item.color }})</small>
        </li>
      </ul>
      <button class="btn" @click="addFruit">과일 추가</button>
    </section>
  </main>
</template>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드 (배열)

```
<!- App.vue-->
<script setup>
import { reactive } from "vue";

// 배열 예제
const fruits = reactive([
  { id: 1, name: "사과", color: "red" },
  { id: 2, name: "바나나", color: "yellow" },
  { id: 3, name: "포도", color: "purple" },
]);

let nextId = 4;
function addFruit() {
  const samples = [
    { name: "오렌지", color: "orange" },
    { name: "키위", color: "green" },
    { name: "블루베리", color: "blue" },
  ];
  const pick = samples[Math.floor(Math.random() * samples.length)];
  fruits.push({ id: nextId++, ...pick });
}
</script>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드 (객체)

```
<!- App.vue-->
<template>
  <main class="container">
    <h1>v-for 기본 실습 (객체)</h1>
    <section class="card">
      <h2>객체 순회</h2>
      <ul class="list">
        <!-- (value, key, idx) in profile  /  :key에는 유일한값 key 사용 -->
        <li v-for="(value, key, idx) in profile" :key="key" class="list-item">
          <strong class="key">{{ idx + 1 }}</strong>
          <strong class="key">{{ key }}</strong>
          <span class="value">{{ value }}</span>
        </li>
      </ul>
      <button class="btn" @click="updateCity">
          도시 변경 (서울 → 부산)
      </button>
    </section>
  </main>
</template>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드 (객체)

```
<!- App.vue-->
<script setup>
import { reactive } from "vue";

// 객체 예제
const profile = reactive({
  name: "홍길동",
  age: 29,
  city: "서울",
  job: "강사",
});

function updateCity() {
  profile.city = "부산";
}
</script>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

리스트 목록 재사용

- 개념
  - 리스트 항목 변경(추가/삭제)시 DOM 재사용

- 동작 순서
  1) 상태(데이터) 변경 (추가/삭제)
  2) 새로운 가상 DOM 생성
  3) 이전 가상 DOM과 비교 (diff)
     - 이전 배열과 새로운 배열 비교
     - Key 존재: 같은 key는 같은 DOM으로 간주하고 최소 이동 (성능 향상 기대)
     - Key 없으면: 인덱스 기준으로 덮어쓰기 처리 (상대 뒤섞임 위험 발생)
  4) 실제 DOM 반영

2.3 조건부 랜더링과 리스트 랜더링 심화

key 존재하는 경우

1) 상태(배열) 변경 – 새로운 d 값 삽입

변경 전 : items = [a, b, c]
변경 후 : items = [d,a, b, c]

2) 새로운 가상DOM 생성

이전 DOM : [A, B, C] // a,b,c 각 항목에 대한 가상노드
이후 DOM : [D, A, B, C]

3) 비교(diff)

이전: [**key**:a] [**key**:b] [**key**:c]
이후: [**key**:d] [**key**:a] [**key**:b] [**key**:c]
같은 key는 같은 DOM으로 간주 : A/B/C 는 재사용 (**이동**)
[key:d] 는 새로운 DOM 생성되어 맨 앞에 삽입

4) DOM 최소 반영

key 없는 경우

1) 상태(배열) 변경 – 새로운 d 값 삽입

변경 전 : items = [a, b, c]
변경 후 : items = [d,a, b, c]

2) 새로운 가상DOM 생성

이전 DOM : [A, B, C] // a,b,c 각 항목에 대한 가상노드
이후 DOM : [D, A, B, C]

3) 비교(diff)

이전: [**0**:a] [**1**:b] [**2**:c] -
이후: [**0**:d] [**1**:a] [**2**:b] [**3**:c]
인덱스 별 처리: [**0**:a] 위치에 [**0**:d]가 덮어씀,
마지막 값 c 는 새 DOM 생성해서 맨 뒤 추가. 결과: [d,a,b,c]
결국 위치는 그대로 두고 내용만 **덮어쓰기** 됨(**상태 뒤섞임 위험**)

4) DOM 최소 반영

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드

```
<!- App.vue-->
      <h2>배열 순회1- key값을 id로 설정</h2>
      <ul class="list">
  <!-- (item, idx) in fruits  /  :key에는 고유 id 사용 -->
        <li v-for="(item, idx) in fruits" :key="item.id" class="list-item">
          <span class="badge">{{ idx + 1 }}</span>
          {{ item.name }} <small class="muted">({{ item.color }})</small>
          <!-- input 과 삭제버튼 추가 -->
          <input type="text" />
          <button @click="delFruit(idx)">-</button>
        </li>
      </ul>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드

```
<!- App.vue-->
      <h2>배열 순회2- key값을 index로 설정</h2>
      <ul class="list">
   <!-- (item, idx) in fruits  /  :key에는 idx 사용 -->
        <li v-for="(item, idx) in fruits" :key="idx"
class="list-item">
          <span class="badge">{{ idx + 1 }}</span>
          {{ item.name }} <small class="muted">({{ item.color }})</small>
          <!-- input 과 삭제버튼 추가 -->
          <input type="text" />
          <button @click="delFruit(idx)">-</button>
        </li>
      </ul>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

실행결과

2.3 조건부 랜더링과 리스트 랜더링 심화

필터링/정렬 + computed

- 용도
  - 화면에 보여줄 검색/필터/정렬/요약통계 등 데이터가 필요할 때
  - 같은 실행 결과를 여러 곳에서 재사용할 때

- 특징
  - 캐싱 기능
    - 데이터가 변경된 경우에만 다시 계산됨
  - 선언적/간결함
    - 원본 데이터와 파생데이터(필터결과) 별도 관리
  - 부작용 방지
    - 원본 데이터를 사본으로 처리해서 원본 데이터 유지

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드

```
<!- App.vue-->
<template>
  <section class="wrap">
    <h2>상품 필터링 (computed 사용)</h2>
    <div class="controls">
      <input v-model="search" placeholder="상품명 검색" />
      <label class="check">
        <input type="checkbox" v-model="onlyInStock" />
        재고 있는 상품만
      </label>
      <span class="count">표시: {{ filteredProducts.length }}개</span>
    </div>

    <ul class="list">
      <li v-for="p in filteredProducts" :key="p.id">
        <b>{{ p.name }}</b>
        <span class="price">{{ p.price.toLocaleString() }}원</span>
        <span :class="p.inStock ? 'badge ok' : 'badge no'">
          {{ p.inStock ? "재고있음" : "재고없음" }}
        </span>
      </li>
    </ul>
  </section>
</template>
```

2.3 조건부 랜더링과 리스트 랜더링 심화

구현 코드

```
<!- App.vue-->
<script setup>
import { reactive, ref, computed } from "vue";

const products = reactive([
  { id: 1, name: "키보드", price: 39000, inStock: true },
  { id: 2, name: "마우스", price: 19000, inStock: false },
  { id: 3, name: "모니터", price: 219000, inStock: true },
  { id: 4, name: "USB 허브", price: 12900, inStock: true },
  { id: 5, name: "노트북 스탠드", price: 29000, inStock: false },
]);
const search = ref("");
const onlyInStock = ref(false);

//     파생값: 검색/체크에 따라 자동 재계산(캐싱됨)
//    - 의존성: products, search, onlyInStock

const filteredProducts = computed(() => {
  const term = search.value.trim().toLowerCase();
  return products.filter((p) => {
    const byText = term === "" || p.name.toLowerCase().includes(term);
    const byStock = !onlyInStock.value || p.inStock;
    return byText && byStock;
  });
});
</script>
```

2.4 폼 입력 바인딩

사용자 입력과 관련된 기본 폼 처리,유효성 검사, 커스텀 컴포넌트 사용방법을 알 수 있다.

학습목표

- 폼 처리 기본
- v-model 과 수식자(modifier)
- Checkbox,Radiobutton, Dropdown(select) 처리
- 폼 유효성

2.4 폼 입력 바인딩

폼 처리 개요

- 개념
  - 사용자 입력 폼 처리시 필수적으로 고려해야 되는 처리 방법

- 고려 사항
  - 양방향 바인딩
    - v-model 사용하면 입력값과 상태(ref, reactive)가 자동 동기화
  - 폼 제출 처리
    - <form @submit.**prevent**="onSubmit"> 코드로 브라우저 기본 제출 방지
    - <input type="button"> 코드로 Enter 키 누를 때 제출 방지
  - UX 요소 : 로딩 상태/에러메시지/폼 리셋 기능
  - 데이터 관리
    - 단일 값 : ref()
    - 배열/객체: reactive()

2.4 폼 입력 바인딩

구현 코드

```
<!- App.vue-->
<template>
  //  자동전송 방지 기능
  <form @submit.prevent="onSubmit" class="form-container">
    <input v-model="name" placeholder="이름" class="input-box" />
    <div class="button-group">
      <button type="submit" :disabled="loading" class="submit-btn">
        {{ loading ? "제출 중..." : "제출" }}
      </button>
      <button type="button“class="reset-btn“  @click="resetForm" :disabled="loading">
          리셋
      </button>
    </div>
    <!-- 메시지 표시 -->
    <p v-if="error" class="error-msg"> {{ error }}</p>
    <p v-if="success" class="success-msg"> {{ success }}</p>
  </form>
</template>
<script setup>
import { ref } from "vue";
// 상태 관리 기능
const name = ref("");
const loading = ref(false);
const error = ref("");
const success = ref("");
```

2.4 폼 입력 바인딩

구현 코드

```
// 제출 처리
function onSubmit() {
  // 초기화
  error.value = "";
  success.value = "";
  loading.value = true;

  // 서버 요청 시뮬레이션 (2초 지연)
  setTimeout(() => {
    loading.value = false;
    if (!name.value.trim()) {
      error.value = "이름을 입력해주세요.";
    } else if (name.value.length < 2) {
      error.value = "이름은 2글자 이상이어야 합니다.";
    } else {
      success.value = `제출됨: ${name.value}`;
      resetForm();
    }
  }, 2000);
}
// 폼 리셋 기능
function resetForm() {
  name.value = "";
}
</script>
```

2.4 폼 입력 바인딩

v-model 과 수식자(modifier)

- 역할
  - input 요소값과 Vue 상태를 양방향으로 동기화
- 수식자(modifier)
  - .trim : 입력값 앞뒤 공백 자동 제거 (사용자 입력 정제)
  - .number : 문자열을 숫자로 자동 변환 (수량, 가격 입력에 적합)
  - .lazy : 입력 중 즉시 반영하지 않고 포커스 해제나 Enter 시 반영 (성능 최적화)

2.4 폼 입력 바인딩

구현 코드

```
<!- App.vue-->
<template>
  <div class="wrap">
    <h2>v-model 수식자 실습</h2>
    <!-- .trim 실습-->
    <div class="box">
      <label>이름 (v-model.trim):
        <input type="text" v-model.trim="username"
          placeholder="앞뒤 공백 자동 제거"
        />
      </label>
      <p>저장된 값: "{{ username }}"</p>
    </div>
    <!-- .number 실습-->
    <div class="box">
      <label>수량 (v-model.number):
        <input type="number“ v-model.number="quantity"
          placeholder="숫자로 변환"
        />
      </label>
      <p>저장된 값: {{ quantity }} (타입: {{ typeof quantity }})</p>
      <p>합계: {{ total.toLocaleString() }} 원</p>
    </div>
```

2.4 폼 입력 바인딩

구현 코드

```
    <!-- .lazy 실습-->
    <div class="box">
      <label>메모 (v-model.lazy):
        <input type="text" v-model.lazy="note"
          placeholder="Enter 또는 포커스 해제 시 반영"
        />
      </label>
      <p>저장된 메모: "{{ note }}"</p>
    </div>
  </div>
</template>
<script setup>
import { ref, computed } from "vue";

const username = ref(""); // .trim
const quantity = ref(1); // .number
const note = ref(""); // .lazy

// 합계 계산 (수량 * 1000원)
const total = computed(() => (Number(quantity.value) || 0) * 1000);
</script>
```

2.4 폼 입력 바인딩

Checkbox, Radiobutton, Dropdown(select) 처리

- Checkbox
  - 단일선택: boolean 과 바인딩
  - 복수선택: string[] 과 바인딩, 체크된 value값이 자동으로 배열에 추가/삭제
- Radiobutton
  - 동일 그룹에서 하나만 선택 가능
  - v-model은 최종 선택값에 바인딩
- Select
  - 단일선택: string 과 바인딩
  - 복수선택: string[] 과 바인딩
  - "선택하세요" : disabled value="" 로 처리

2.4 폼 입력 바인딩

Checkbox 구현 코드

```
<!-- App.vue -->
<template>
  <div class="box">
    <h3>Checkboxes 실습</h3>

    <label class="row">
      <input type="checkbox" v-model="agree" />
      <span>약관 동의 (단일 선택, boolean)</span>
    </label>

    <div class="row">취미(복수 선택, string[]):</div>
    <div class="list">
      <label v-for="o in options" :key="o.value" class="row">
        <input type="checkbox" :value="o.value" v-model="selected" />
        <span>{{ o.label }}</span>
      </label>
    </div>

    <p class="preview">agree: {{ agree }} / selected: {{ selected }}</p>
  </div>
</template>
```

2.4 폼 입력 바인딩

Checkbox 구현 코드

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

// 단일 체크(약관 동의 등)
const agree = ref(false);

// 복수 체크(문자열 배열)
const selected = ref([]);
const options = [
  { label: "독서", value: "read" },
  { label: "운동", value: "workout" },
  { label: "음악", value: "music" },
];
</script>
```

2.4 폼 입력 바인딩

RadioButton 구현 코드

```
<!-- App.vue -->
<template>
  <div class="box">
    <h3>Radiobuttons 실습</h3>
    <div class="list">
      <label v-for="o in options" :key="o.value" class="row">
        <input
          type="radio"
          name="delivery"
          :value="o.value"
          v-model="delivery"
        />
        <span>{{ o.label }}</span>
      </label>
    </div>

    <p class="preview">선택: {{ delivery }}</p>
  </div>
</template>
```

2.4 폼 입력 바인딩

RadioButton 구현 코드

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

const delivery = ref("standard");
const options = [
  { label: "일반(무료)", value: "일반" },
  { label: "익일(+3,000)", value: "익일" },
  { label: "당일(+5,000)", value: "당일" },
];
</script>
```

2.4 폼 입력 바인딩

Select 구현 코드

```
<!-- App.vue -->
<template>
  <div class="box">
    <h3>Select 실습</h3>
    <label class="row">
      <span>국가(단일)</span>
      <select v-model="country">
        <option disabled value="">-- 선택하세요 --</option>
        <option v-for="c in countries" :key="c.value" :value="c.value">
          {{ c.label }}
        </option>
      </select>
    </label>
    <label class="row">
      <span>스킬(다중)</span>
      <select v-model="skills" multiple size="3">
        <option v-for="s in skillOptions" :key="s.value" :value="s.value">
          {{ s.label }}
        </option>
      </select>
    </label>
    <p class="preview">country: {{ country }} / skills: {{ skills }}</p>
  </div>
</template>
```

2.4 폼 입력 바인딩

Select 구현 코드

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

// 단일
const country = ref("");
const countries = [
  { label: "대한민국", value: "KR" },
  { label: "일본", value: "JP" },
  { label: "미국", value: "US" },
];

// 다중
const skills = ref([]);
const skillOptions = [
  { label: "JavaScript", value: "js" },
  { label: "Python", value: "py" },
  { label: "SQL", value: "sql" },
];
</script>
```

2.4 폼 입력 바인딩

폼 유효성

- 검증 대상
  - 필수 입력(required)
  - 입력 길이(length)
  - 정규식(regex) : 이메일, 전화번호 형식
  - 값의 범위(range)
- 검증 시점
  - 실시간 검증: on change/ blur 시점, 즉각 피드백
  - 일괄 검증: on submit 시점, 구현 간단
- 에러 표시
  - 시각적 강조(빨간 테두리), 오류 메시지

2.4 폼 입력 바인딩

구현 코드

```
<!-- App.vue  template 코드-->
<template>
  <form class="card" @submit.prevent="onSubmit" novalidate>
    <h2>간단 폼 유효성 검사 실습</h2>

    <div class="field" :class="{ invalid: errors.name }">
      <label for="name">이름 *</label>
      <input  id="name"  v-model.trim="form.name"  @blur="validateField('name')" />
      <small v-if="errors.name" class="error">{{ errors.name }}</small>
    </div>
    <div class="field" :class="{ invalid: errors.email }">
      <label for="email">이메일 *</label>
      <input id="email" type="email" v-model.trim="form.email“ @blur="validateField('email')"
        placeholder="user@example.com" />
      <small v-if="errors.email" class="error">{{ errors.email }}</small>
    </div>
    <div class="field" :class="{ invalid: errors.phone }">
      <label for="phone">전화번호 *</label>
      <input id="phone" v-model.trim="form.phone" @blur="validateField('phone')"
        placeholder="010-1234-5678“ />
      <small v-if="errors.phone" class="error">{{ errors.phone }}</small>
    </div>
```

2.4 폼 입력 바인딩

구현 코드

```
<div class="field" :class="{ invalid: errors.age }">
      <label for="age">나이(1~120) *</label>
      <input id="age" type="number" v-model.number="form.age"
        @blur="validateField('age')"  min="1“ max="120"
        placeholder="예: 29"  />
      <small v-if="errors.age" class="error">{{ errors.age }}</small>
    </div>
    <button class="btn" type="submit">제출</button>
    <p  v-if="submitMsg" class="msg"
      :class="{
        bad: submitMsg.includes('확인'),
        ok: submitMsg.includes('성공'),
      }"
    >
      {{ submitMsg }}
    </p>
  </form>
</template>
```

2.4 폼 입력 바인딩

구현 코드

```
<!-- App.vue  script 코드-->
<script setup>
import { reactive, ref } from "vue";
const form = reactive({
  name: "", email: "", phone: "",age: "",
});
const errors = reactive({
  name: "", email: "", phone: "", age: "",
});

/* 단순한 규칙 적용*/
function validateField(field) {
  const v = String(form[field] ?? "").trim();
  if (field === "name") {
    errors.name = !v ? "이름은 필수입니다." : v.length < 2 || v.length > 20 ? "2~20자 입력": "";
  }
  if (field === "email") {
    const re = /^[^\s@]+@[^\s@]+\.[^\s@]{2,}$/;
    errors.email = !v ? "이메일은 필수입니다.": !re.test(v) ? "이메일 형식 확인" : "";
  }
  if (field === "phone") {
    const re = /^0\d{1,2}-?\d{3,4}-?\d{4}$/;
    errors.phone = !v      ? "전화번호는 필수입니다.": !re.test(v)? "예: 010-1234-5678": "";
  }
```

2.4 폼 입력 바인딩

구현 코드

```
if (field === "age") {
    const n = Number(form.age);
    errors.age =      v === "" ? "나이는 필수입니다." : Number.isNaN(n)
        ? "숫자 입력": n < 1 || n > 120? "1~120 범위“: "";
  }
}
function validateAll() {
  Object.keys(form).forEach((f) => validateField(f));
  return Object.values(errors).every((msg) => !msg);
}

const submitMsg = ref("");
async function onSubmit() {
  submitMsg.value = "";
  if (!validateAll()) {
    submitMsg.value = "입력값을 확인해주세요.";
    return;
  }
   submitMsg.value = "제출 성공!";

}
</script>
>
```

2.4 폼 입력 바인딩

template 의 ref

- 개념
  - DOM 요소나 자식 컴포넌트 인스턴스 참조
- 용도
  - 포커스/선택/스크롤, 화면 크기 측정
  - 서드파티 UI 초기화/해제
  - 자식 컴포넌트의 메서드 호출
- 핵심패턴
  - 문자열 ref
  - 컴포넌트 ref

2.4 폼 입력 바인딩

문자열 ref

- 개념
  - input 같은 사용자 입력 DOM 요소 참조
  - 가장 일반적으로 사용

- 문법
  - template : <input ref="참조변수" />
  - script: const 참조변수 = ref(null)

  - 참조변수를 이용해서 DOM 참조 가능 (현재 DOM은 input)

2.4 폼 입력 바인딩

구현 코드

```
<!-- App.vue  -->
<template>
  <div class="container">
    <h2>문자열 ref 실습</h2>
    <input ref="inputEl" class="input-box" placeholder="여기에 입력하세요" />
    <button class="btn" @click="focusInput">포커스 주기</button>
    <button class="btn secondary" @click="clearInput">입력 초기화</button>
  </div>
</template>
<script setup>
import { ref } from "vue";
const inputEl = ref(null);
// 이벤트 핸들러에서 직접 접근
function focusInput() {
  inputEl.value?.focus();
}
function clearInput() {
  if (inputEl.value) {
    inputEl.value.value = "";
    inputEl.value.focus();
  }
}
</script>
```

2.4 폼 입력 바인딩

컴포넌트 ref

- 개념
  - 부모 컴포넌트에서 자식 컴포넌트 메서드/상태 참조
  - defineExpose 이용

- 문법
  - 부모

    template: <Child ref="참조변수" />
    script: const 참조변수 = ref(null);

  - 자식: defineExpose 로 메서드 공개 , ex. defineExpose({getValue, clear})

  - 참조변수를 이용해서 자식 메서드 호출 가능

2.4 폼 입력 바인딩

구현 코드

```
<!-- App.vue  -->
<template>
  <div class="parent">
    <h2>컴포넌트 ref 실습</h2>
    <Child ref="childRef" />
    <div class="btn-group">
      <button class="btn" @click="logValue">자식 값 확인</button>
      <button class="btn secondary" @click="clearChild">자식 초기화</button>
    </div>
  </div>
</template>
<script setup>
import { ref } from "vue";
import Child from "./components/ChildComponent.vue";
// 자식 컴포넌트를 참조하는 ref
const childRef = ref(null);

function logValue() {
  alert(`자식 값: ${childRef.value?.getValue()}`);
}
function clearChild() {
  childRef.value?.clear();
}
</script>
```

2.4 폼 입력 바인딩

구현 코드

```
<!-- ChildComponent.vue  script -->
<template>
  <div class="child">
    <input v-model="text" class="input-box" placeholder="자식 입력창" />
  </div>
</template>

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

const text = ref("");

// 부모가 접근할 수 있도록 노출할 메서드 정의
function getValue() {
  return text.value;
}

function clear() {
  text.value = "";
}

// defineExpose로 외부 접근 허용
defineExpose({ getValue, clear });
</script>
```

2.5 정리 및 실습

요약맵

- 전역/지역/동적 컴포넌트 개념
- props/emit/slot 활용한 컴포넌트 통신
- 조건/리스트 + computed 활용
- 폼 처리 고려사항
- template의 ref
- 실습 (미니 수강 신청 앱)

3. 상태 관리와 라이프 사이클

- 컴포넌트 라이프 사이클
- Composition API 기초
- Composition API 심화
- 상태 관리 패턴
- Provide/Inject 패턴

3.1 컴포넌트 라이프 사이클

컴포넌트 라이프 사이클의 개념과 실무 적용 타이밍 및 사용 가능한 hook에 관하여 살펴보자

학습목표

- 라이프 사이클 개요
- 단계별 함수 및 역할
- 용도별 분류
- 기본 및 Special 아키텍처

3.1 컴포넌트 라이프 사이클

라이프 사이클

- 개념
  - 컴포넌트의 생명주기(생성,랜더링,수정,제거) 동안의 특정 시점에 코드를 자동으로 실행할 수 있도록 지원하는 함수(hook)

- 공식 사이트
  - https://vuejs.org/guide/essentials/lifecycle.html

- 용도
  - DOM이 생성된 이후에 가능한 작업 처리 ex. 포커스 지정
  - 업데이트 이후 의미 있는 작업 처리 ex. 레이아웃 재계산
  - 컴포넌트 제거 시 작업 처리 ex. 이벤트 리스너 제거
  - 특수 상황 발생 시 작업 처리 ex. 에러, <keep-alive> 적용

3.1 컴포넌트 라이프 사이클

라이프 사이클 핵심 단계

- 1단계
  - 마운트(mount) 단계
    컴포넌트가 화면(DOM)에 올라가는 과정
- 2단계
  - 업데이트(update)단계
    반응형 데이터가 변경되어 화면이 다시 그려지는 과정
- 3단계
  - 언마운트(unmount)단계
    컴포넌트가 화면에서 사라질 때 정리하는 과정
- 4단계
  - 특수 상황(special)단계
    일반 흐름 외에도 예외적인 상황에서 실행되는 과정

3.1 컴포넌트 라이프 사이클

마운트(mount) 단계

- 개념
  - 컴포넌트가 처음 화면(DOM)에 올라가는 과정
  - 한 번만 실행

- hook
  - onBeforeMount

    호출시점 : DOM에 생성되기 직전

    특징 : DOM 접근 불가

    용도 : 데이터 초기화, 외부 상태 설정
  - onMounted

    호출시점 : DOM에 생성된 후

    특징 : 실제 DOM 접근 가능 (ref 가능)

    용도 : API 호출, chart 랜더링, 3rd-party 라이브러리 초기화

3.1 컴포넌트 라이프 사이클

업데이트(update) 단계

- 개념
  - 반응형 데이터가 변경되어 화면이 재랜더링 되는 과정

- hook
  - onBeforeUpdate

    호출시점 : 가상 DOM과 실제 DOM 비교해서 실제 DOM을 업데이트하기 직전

    특징 : 변경되기 전 값 확인 가능

  - onUpdated

    호출시점 : DOM 업데이트 직후

    특징 : 여러 상태 변경은 한 번에 처리 (성능 최적화)

    용도 : 레이아웃 크기 측정, 3rd-party 라이브러리 재초기화

3.1 컴포넌트 라이프 사이클

언마운트(unmount) 단계

- 개념
  - 컴포넌트가 화면에서 완전히 제거될 때

- hook
  - onBeforeUnmount

    호출시점 : 제거되기 직전

    특징 : 타이머/이벤트 리스너 등 메모리 누수 방지 기능 처리 가능

    용도 : setInterval/setTimeout 해제, 이벤트 리스너 제거

  - onUnmounted

    호출시점 : 제거된 직후

    특징 : 모든 정리 완료 확인 가능

3.1 컴포넌트 라이프 사이클

특수상황(Special) 단계

- 개념
  - 일반 흐름 외의 예외적인 상황에서만 실행

- 에러처리 hook
  - onErrorCaptured
    하위에서 발생된 에러를 상위로 전파, return false 지정 시 전파 멈춤

- Keep-Alive 캐시 hook
  - onActivated
    호출시점 : <keep-alive>로 캐시된 컴포넌트가 다시 활성화 될 때
  - onDeactivated
    호출시점 : <keep-alive> 로 캐시된 컴포넌트가 비활성화 될 때

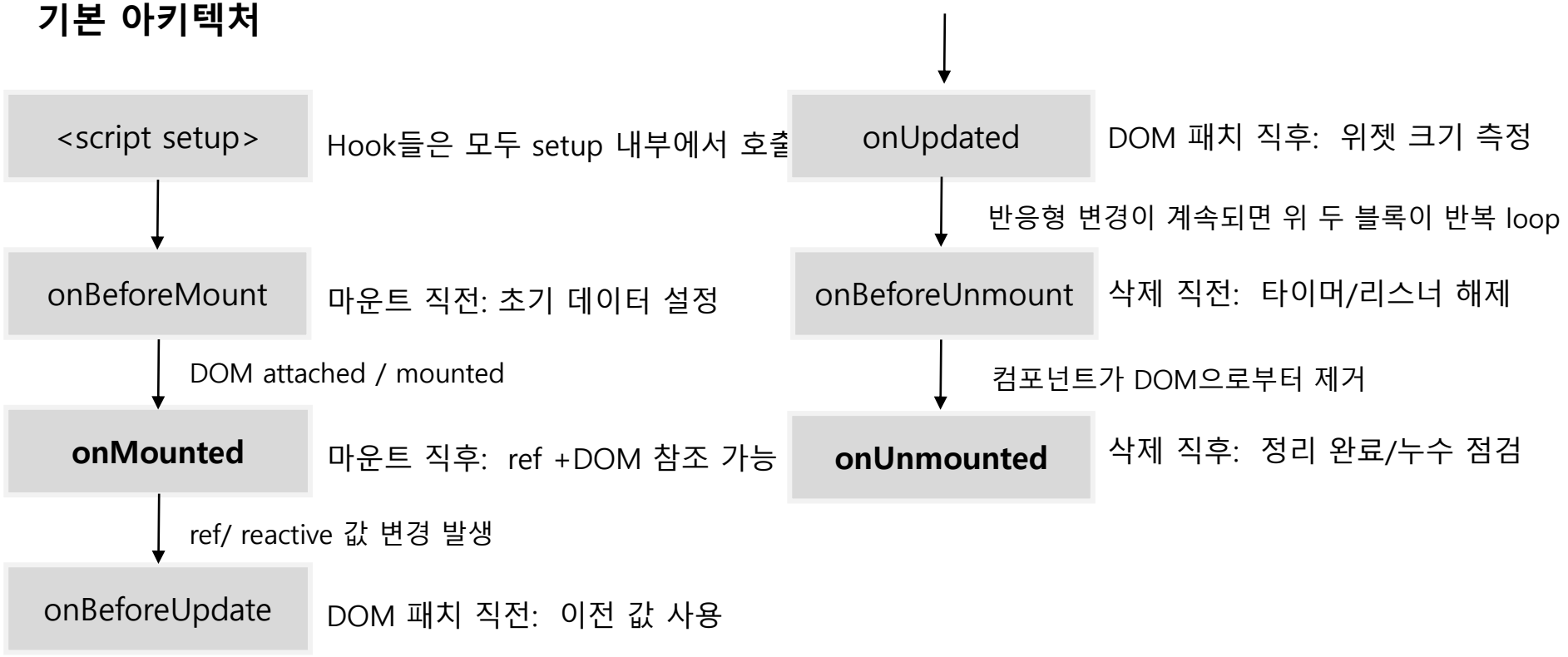
3.1 컴포넌트 라이프 사이클

용도 별 분류

단계	hook	용도
초기 설정	• onBeforeMount : 컴포넌트가 DOM 생성 직전	• 데이터, 외부 라이브러리 초기화
	• **onMounted** : 컴포넌트가 DOM 생성 직후	• ref 로 DOM 접근 가능, API 호출
화면 갱신	• onBeforeUpdate : 상태가 변경되어 DOM 업데이트 되기 전	• 업데이트 전 값 확인
	• onUpdated : DOM이 다시 그려진 직후	• 위젯 크기 측정, nextTick() 활용
정리	• onBeforeUnmount : 컴포넌트가 제거되기 직전	• 타이머/이벤트 리스너 해제
	• **onUnmounted** : 컴포넌트 제거된 직후	• 정리 작업 완료, 메모리 누수 방지
예외 상황	• **onErrorCaptured** : 자식 컴포넌트에서 에러 발생시 부모가 처리	• 에러 로깅, 사용자 친화적 메시지
	• onActivated : <keep-alive> 컴포넌트가 다시 활성화될 때	• 탭 전환 후 데이터/상태 복구
캐시 상황	• onDeactivated : <keep-alive> 컴포넌트가 비활성화될 때	• 리소스 사용 중지
	• onBeforeMount : 컴포넌트가 DOM 생성 직전	• 데이터, 외부 라이브러리 초기화

3.1 컴포넌트 라이프 사이클

기본 아키텍처



3.1 컴포넌트 라이프 사이클

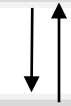
Special 아키텍처

onErrorCaptured

부모가 자식 오류 처리
return false : 상위 전파 중지

onActivated

<keep-alive>
컴포넌트가 다시 보일 때



onDeactivated

<keep-alive>
컴포넌트가 숨겨질 때

3.1 컴포넌트 라이프 사이클

구현 코드

```
<!-- App.vue  -->
<template>
  <div class="box">
    <h2>Lifecycle All-in-One Demo</h2>

    <div class="row">
      <button class="btn" @click="count++">증가(Update 확인)</button>
      <button class="btn" @click="show = !show">토글(Unmount 확인)</button>
      <button class="btn" @click="tab = tab === 'A' ? 'B' : 'A'">
        탭 전환(keep-alive)
      </button>
    </div>
    <p ref="meter">count: {{ count }}</p>
    <p class="note">콘솔 로그에서 훅 호출 순서를 확인하세요.</p>

    <keep-alive>
      <TabA v-if="tab === 'A' && show" />
      <TabB v-else-if="tab === 'B' && show" />
    </keep-alive>
  </div>
</template>
```

3.1 컴포넌트 라이프 사이클

구현 코드

```
<!-- App.vue -->
<script setup>
import {  ref,onBeforeMount,onMounted,onBeforeUpdate,onUpdated,
  onBeforeUnmount,onUnmounted,onErrorCaptured,nextTick } from "vue";

// 외부 컴포넌트 import
import TabA from "@/components/TabA.vue";
import TabB from "@/components/TabB.vue";

const count = ref(0);
const show = ref(true);
const tab = ref("A");
const meter = ref(null);

// 부모 컴포넌트 훅
onBeforeMount(() => console.log("App컴포넌트: onBeforeMount: DOM 붙기 직전"));
onMounted(() =>
  console.log(
    "App컴포넌트: onMounted: DOM 붙은 직후, meter height=",
    meter.value?.offsetHeight
  )
);
```

3.1 컴포넌트 라이프 사이클

구현 코드

```
<!-- App.vue  -->
onBeforeUpdate(() =>
  console.log("App컴포넌트: onBeforeUpdate: DOM 패치 직전, count=", count.value)
);
onUpdated(async () => {
  await nextTick();
  console.log(
    "App컴포넌트: onUpdated: DOM 패치 직후, meter height=",
    meter.value?.offsetHeight
  );
});
onBeforeUnmount(() =>
  console.log("App컴포넌트: onBeforeUnmount: 파기 직전, 정리 시작")
);
onUnmounted(() =>
  console.log("App컴포넌트: onUnmounted: 파기 직후, 정리 완료")
);
onErrorCaptured((err, instance, info) => {
  console.error("App컴포넌트: onErrorCaptured:", err.message, info);
  return false; // 전파 중지
});
</script>
```

3.1 컴포넌트 라이프 사이클

구현 코드

```vue
<!-- TabA.vue  -->
<template>
  <div class="box">
    Tab A 내용 ( keep-alive 이용한 타이머 유지) — t: {{ t }}
  </div>
  <button class="btn" @click="boom">에러 발생(ErrorCaptured 확인)</button>
</template>

<script setup>
import { ref, onMounted, onUnmounted, onActivated, onDeactivated } from "vue";

const t = ref(0);
let timer = null;

function boom() {
  throw new Error("Tab A: 의도적 오류 발생, 부모인 App에서 처리");
}

onMounted(() => {
  console.log("Tab A: onMounted");
  timer = setInterval(() => (t.value += 1), 1000);
});
```

3.1 컴포넌트 라이프 사이클

구현 코드

```
<!-- TabA.vue  -->
onUnmounted(() => {
  console.log("Tab A: onUnmounted");
  clearInterval(timer);
});
onActivated(() => console.log("Tab A: onActivated"));
onDeactivated(() => console.log("Tab A: onDeactivated"));
</script>


<!-- TabB.vue  -->
<template>
  <div class="box">Tab B 내용</div>
</template>
<script setup>
import { onMounted, onUnmounted, onActivated,
       onDeactivated } from "vue";

onMounted(() => console.log("Tab B: onMounted"));
onUnmounted(() => console.log("Tab B: onUnmounted"));
onActivated(() => console.log("Tab B: onActivated"));
onDeactivated(() => console.log("Tab B: onDeactivated"));
</script>
```

3.2 Composition API 기초

Vue3 권장 개발 방식인 Composition API 의 기본 문법을 이해하여 어플리케이션 개발 기초를 다진다.

학습목표

- 등장 배경
- Options API 비교
- 기본 구조
  - <script setup>
  - setup()
- 핵심 개념 리뷰
  - ref
  - reactive
  - computed
  - watch
  - Template ref

3.2 Composition API 기초

등장 배경

- 이유
  - Vue 2 의 Options API 한계
  - 로직이 여러 옵션(data, methods, computed)에 분산처리
  - 대규모 프로젝트에서 코드 재사용성/가독성 문제

- 특징
  - Vue 3 권장 개발 방식
  - 로직을 기능 단위로 묶어 재사용 증대
  - Typescript 친화적

3.2 Composition API 기초

Options API 비교

단계	hook	용도
코드 구조	• data/methods/computed/watch 등 옵션별로 분산구현 (기능 파악이 어려움)	• 기능별로 묶음 • 관련 상태/로직을 한 곳에 배치(setup)
재사용성	• 충돌 / 가독성 문제 • 추적 어려움	• Composables(useXXX)로 상태/로직 캡슐화 • 의존성 주입 테스트 용이
타입지원 (Typescript)	• this 기반 추론 한계 • 제네릭 / 리팩터링 불편	• TS 친화적. 명시적 변수 / 함수 • 제네릭과 유틸 타입 활용 쉬움
가독성/규모	• 소규모/단순 화면에 유리 • 진입장벽 낮음	• 대규모/복잡 기능에 유리 • 관심사 분리/모듈화 우수
라이프사이클 hook	• beforeCreate / created / mounted / updated / unmounted 등 제공	• onMounted / onUpdated / onUnmounted 등 • Hook을 함수처럼 조합 가능
학습 곡선	• 익숙하고 직관적	• 초반 진입 장벽 높음(reactivity, ref, reactive등)

3.2 Composition API 기초

코드 비교

```
<!- Options API 방식  -->
<script>
export default {
  data() { return { count: 0 } },
  computed: {    doubled() { return this.count * 2 }},
  methods: {   inc() { this.count++ } },

  mounted() { console.log(“mounted”) }
}
</script>
<template>
  <button @click="inc">
    Count: {{ count }} / x2={{ doubled }}
  </button>
</template>
```

```
<!- Composition API 방식  -->
<script setup>
import { ref, computed, onMounted } from 'vue'

const count = ref(0)
const doubled = computed(() => count.value * 2)
const inc = () => count.value++

onMounted(() => console.log('mounted'))
</script>

<template>
  <button @click="inc">
     Count: {{ count }} / x2={{ doubled }}
 </button>
</template>
```

3.2 Composition API 기초

기본 구조

- 컴파일 타입 방식
  - 문법: **<script setup>**
  - 특징:
    선언한 변수,함수, import 는 템플릿에서 바로 사용
    return { } 불필요
    SFC 컴파일러가 빌드 과정에서 자동으로 setup()으로 변환
    현업에서 주로 사용하는 방식

- 런타임 방식
  - 문법: **<script>** export default{ **setup(){}**}
  - 특징:
    setup() 함수안에서 선언된 값은 템플릿에서 바로 사용 불가
    반드시 return { } 통해서 노출해야 템플릿에서 사용 가능

3.2 Composition API 기초

핵심 차이 요약

단계	hook	용도
처리 방식	• 컴파일 타임 방식, SFC가 하나의 setup()으로 변환되어 불필요한 래퍼/return 제거	• 런타임 방식으로 setup() 실행 • 컴파일 최적화 이점은 적음
템플릿 노출	• 선언만 하면 템플릿에서 바로 사용 (자동 expose)	• 선언하고 반드시 return {...} 코드로 • 템플릿에 명시적 노출 필요
보일러 플레이트	• 최소화 • Import/로컬변수는 템플릿에서 바로 사용	• export default, setup(props.ctx), return 등 상용구가 많음
Props/emit	• defineProps() • defineEmits()	• props:{...} • emits:[...]
가독성	• top-level 코드로 흐름이 단순 • 관련 로직 근접 배치	• 옵션/컨텍스트/return 코드 분산 • 길고 중첩되기 쉬움
권장 사용처	• 대부분의 신규 컴포넌트 • TS 스킬 적용 가능	• 레거시 유지보수 • 라이브러리에서 옵션 명시가 많은 경우

3.2 Composition API 기초

코드 비교

```vue
<!-- <script setup> 방식 -->
<script setup>
import { ref, computed, onMounted } from "vue";

const props = defineProps({
  msg: { type: String, required: false, default:
"<script setup> 방식" },
  start: { type: Number, default: 0 },
});
const count = ref(props.start);
const doubled = computed(() => count.value * 2);
const inc = () => count.value++;
onMounted(() => {
  console.log("mounted:", props.msg);
});
</script>
<template>
  <div>
    <h3>{{ props.msg }}</h3>
    <button @click="inc">Count: {{ count }}
(x2={{ doubled }})</button>
  </div>
</template>
```

```vue
<!-- <script> + setup() 방식 -->
<script>
import { ref, computed, onMounted } from "vue";

export default {
  name: "CounterSetupFunction",
  props: {
    msg: { type: String, required: false, default:
"<script> + setup() 방식" },
    start: { type: Number, default: 0 },
  },
  setup(props) {
    const count = ref(props.start);
    const doubled = computed(() => count.value * 2);
    const inc = () => count.value++;

    onMounted(() => {
      console.log("mounted:", props.msg);
    });
    return { count, doubled, inc, props };
  },
};
</script>
```

3.2 Composition API 기초

핵심 개념 리뷰

- ref
  - toRefs
- reactive
  - toRefs
- computed
  - get/set
- watch 및 옵션
  - immediate
  - deep
  - flush
- template ref

3.2 Composition API 기초

ref

- 개념
  - 숫자,문자,불리언 등 원시값을 반응형 상태로 만드는 함수
- 특징: 단일값 관리에 적합
- 장점
  - 단순하고 직관적
  - 어떤 타입이든 사용 가능 (배열/객체 포함)
- 단점
  - .value 사용으로 접근이 번거로움
  - 구조 분해 할당 시 반응성이 제거 (**toRefs** 로 유지)

3.2 Composition API 기초

구현 코드

```
<!- App.vue →
<template>
  <div class=“app”>
    <h2>정상 (ref 그대로 사용)</h2>
    <p>{{ count }}</p>

    <h2>비반응 (구조 분해)</h2>
    <p>{{ value }}</p>

    <h2>복원 (toRefs 사용)</h2>
    <p>{{ countFixed }}</p>

    <button @click="inc">+1</button>
  </div>
</template>
```

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

const count = ref(0);

// 구조 분해 → 단순 값 복사 (반응성 깨짐)
const { value } = count;

// reactive에 감싸서 toRefs로 꺼내면 ref 유지
const { count: countFixed } =
toRefs(reactive({ count }));

const inc = () => {
  count.value++;
};
</script>
```

3.2 Composition API 기초

reactive

- 개념
  - 객체/배열을 Proxy로 감싸서 반응형 상태로 만드는 함수
- 특징: 깊은 구조까지 반응성 처리
- 장점
  - 객체 단위 상태 관리에 적합
  - 중첩 객체도 자동으로 반응성 유지
- 단점
  - 구조 분해 할당 시 반응성이 제거 (**toRefs** 로 유지)

3.2 Composition API 기초

구현 코드

```
<!- App.vue →
<template>
  <div class="app">
    <h2>정상 (reactive 그대로 사용)</h2>
    <p>{{ state.count }}</p>

    <h2>비반응 (구조 분해)</h2>
    <p>{{ count }}</p>

    <h2>복원 (toRefs 사용)</h2>
    <p>{{ countFixed }}</p>

    <button @click="inc">+1</button>
  </div>
</template>
```

```
<script setup>
import { reactive, toRefs } from "vue";

// reactive 객체
const state = reactive({ count: 0 });

// 구조 분해 → 단순 값 복사 (반응성 깨짐)
const { count } = state;

// toRefs로 속성을 ref로 꺼내면 반응성 유지
const { count: countFixed } = toRefs(state);

const inc = () => {
  state.count++;
};
</script>
```

3.2 Composition API 기초

computed

- 개념
  - 다른 반응형 상태를 기반으로 계산된 값 반환
- 특징
  - 의존값이 변하지 않으면 캐싱 처리
  - **get / set 지원**
- 장단점
  - 복잡하거나 반복 계산 감소
  - 의존성 관리 자동
  - 조건부 + 리스트 + computed 처리시 성능향상
  - 남용 시 코드 복잡도 증가

3.2 Composition API 기초

구현 코드 (기본)

```vue
<!- App.vue →
<template>
  <div class="app">
    <h2>상품 가격 계산</h2>
    <p>가격: {{ price }}원</p>
    <p>세율: {{ tax * 100 }}%</p>
    <p>총액: {{ total }}원</p>
    <p class="tag">{{ status }}</p>

    <div class="controls">
      <button @click="price += 10">+10</button>
      <button @click="price -= 10">-10</button>
    </div>
  </div>
</template>
```

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

const price = ref(100);
const tax = ref(0.1);

/* 세금 포함 총액 */
const total = computed(() => Math.floor(price.value * (1 + tax.value)));

/* 조건부 computed */
const status = computed(() => (total.value >= 120 ? "비쌈" : "저렴"));
</script>
```

3.2 Composition API 기초

computed (get/set)

- 개념
  - 기본적으로 읽기 전용
  - get/set 사용하면 읽기+쓰기 가능한 계산 속성 가능
- 문법
  - computed({ get: ()=> …, set(v)=> {….} })
  - get은 읽을 때 실행, set은 값 대입할 때 실행
  - 핵심: 읽을 때는 계산, 쓸 때는 원하는 규칙으로 원본 상태 갱신
- 용도
  - 전처리 작업 (공백제거, 소/대문자 변환, 숫자 변화, 길이 제한)
  - 커스텀 컴포넌트에서 v-model 지원

3.2 Composition API 기초

구현 코드 (get/set)

```
<!- App.vue →
<template>
  <main class="wrap">
    <section class="card">
      <h2 class="title">v-model + computed(get/set) 실습
</h2>
      <p class="desc">입력 시 자동으로 <code>trim()</code>
처리됨</p>

      <div class="field">
        <label class="label">이름 입력</label>
        <input
          v-model="normalizedName"
          class="input"
          placeholder="예)   홍 길동   "
        />
        <small class="hint">앞·뒤 공백은 자동 제거됨</small>
      </div>
      <div class="result">
        <span class="pill">저장된 값:
"{{ rawName }}"</span>
      </div>
    </section>
  </main>
</template>
```

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

const rawName = ref("");

// get/set computed 구현
const normalizedName = computed({
  get: () => rawName.value,
  set: (v) => {
    rawName.value = v.trim();
  },
});
</script>
```

3.2 Composition API 기초

watch

- 개념
  - 특정 데이터(들)의 변화를 감지하여 부수 효과(side effect) 처리
- 옵션
  - immediate : false
  - deep: false
  - flush:'pre|post|sync'
- 장/단점
  - 비동기 처리
  - API 호출 및 로컬 스토리지 동기화 적합
  - 의존성을 명시적으로 지정해서 관리 비용 증가

3.2 Composition API 기초

구현 코드 (기본)

```
<!- App.vue →
<template>
  <div class="app">
    <h2>watch 실습</h2>
    <input v-model="query"
       placeholder="검색어를 입력하세요" />
    <p>
      현재 검색어: <strong>{{ query }}</strong>
    </p>
  </div>
</template>
```

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

const query = ref("");

watch(
  query,
  (newVal, oldVal) => {
    console.log("검색:", newVal, "(이전:", oldVal, ")");
  },
  {
    immediate: false,
  }
);
</script>
```

3.2 Composition API 기초

구현 코드 (deep + flush)

```
<!- App.vue →
<template>
  <div class="container">
    <h2>deep + flush 실습</h2>
    <p id="name" class="name">이름: {{ user.info.name }}</p>
    <button class="btn" @click="changeName">
     이름 변경
    </button>
  </div>
</template>

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

const user = ref({
  info: { name: "Ann" },
});

function changeName() {
  user.value.info.name = user.value.info.name === "Ann" ? "Ben" : "Ann";
}
```

```
// flush: 'pre'
watch(  user,
  () => {  console.log("[pre] DOM 상태:",
              document.getElementById("name").textContent);
  },{ deep: true, flush: "pre" }
);
// flush: 'post'
watch( user,
  () => {  console.log( "[post] DOM 상태:",
              document.getElementById("name").textContent
    );
  },  { deep: true, flush: "post" }
);
// flush: 'sync'
watch( user,
  () => {  console.log("[sync] DOM 상태:",
              document.getElementById("name").textContent
    );
  },{ deep: true, flush: "sync" }
);
</script>
```

3.2 Composition API 기초

Template ref

- 개념
  - DOM 요소 또는 자식 컴포넌트에 직접 접근
- 특징: Vue의 선언적 랜더링 기능 보완, onMounted 함께 사용
- 장점
  - DOM API 활용 가능 ex. Focus, scroll
  - 외부 라이브러리 초기화 용이
- 단점
  - 남용 시 Vue의 선언적 UI 철학 훼손
  - SSR (Server Side Rendering) 에서는 주의

3.2 Composition API 기초

구현 코드

```
<!- App.vue →
<template>
  <div class="app">
    <h2>자동 포커스 인풋</h2>
    <input ref="inputEl" placeholder="여기에 자동으로 포커스됩니다" />
  </div>
</template>

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

const inputEl = ref(null);

onMounted(() => {
  // 컴포넌트가 마운트되면 input에 자동 포커스
  inputEl.value.focus();
});
</script>
```

3.3 Composition API 심화

Composition API 심화 문법을 활용해 다양한 조건에서 최적화된 어플리케이션을 구현한다.

학습목표

- watchEffect
- 컴포넌트의 v-model
  - 커스텀 컴포넌트
- Composables 설계 패턴

3.3 Composition API 심화

watchEffect

- 개념
  - 함수안에서 사용된 반응형 값들을 자동으로 추적
  - 이전 watch는 명시적으로 반응형 변수 지정 및 지정한 값만 추적 가능
- 용도
  - 간단한 부수 효과 처리
  - 감시할 대상이 불명확할 때
  - computed와 비슷하지만 결과 반환없이 사용할 때
- 문법
  - watch(()=>{})
  - watch((onCleanup)=>{ … onCleanup(()=>{})})

3.3 Composition API 심화

watch 비교

구분	watch	watchEffect
의존성 지정	• 감시 대상을 직접 지정 • 지정된 대상만 추적	• 함수 안에서 사용된 반응형 변수 자동 추적
초기 실행	• 기본적으로 실행 안됨 • immediate:true 필요	• 기본적으로 즉시 1 회 실행
콜백 인자	• (newVal, oldVal, onCleanup)	• (onCleanup) • clean-up 필요시 적용
주요 특징	• 이전/현재 값 비교 가능 • 다중 감시 대상 지정 가능	• 간단/자동 동작 • 의존성 추적 자동
대표 용도	• 특정 값만 감시 • 이전/현재 값 비교 필요	• 초기 데이터 연동 • 부수 효과 자동화

3.3 Composition API 심화

구현 코드

```
<!- App.vue →
<template>
  <div class="wrap">
    <h2>watchEffect + CSS</h2>

    <button @click="color = 'red'">빨강</button>
    <button @click="color = 'green'">초록</button>
    <button @click="color = 'blue'">파랑</button>

    <!-- watchEffect에서 style을 적용할 대상 -->
    <div ref="boxEl" class="box">색상이 자동으로 바뀝니다</div>
  </div>
</template>
```

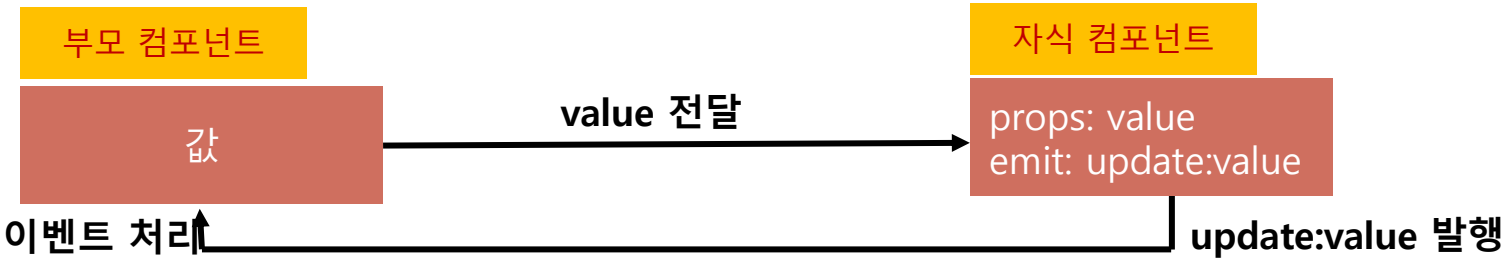
```
<script setup>
import { ref, watchEffect } from "vue";

const color = ref("red");
const boxEl = ref(null);

// watchEffect: color 값이 바뀌면 박스에 CSS 적용
watchEffect(() => {
  if (boxEl.value) {
    boxEl.value.style.backgroundColor = color.value;
    boxEl.value.style.color = "white";
    boxEl.value.style.padding = "20px";
    boxEl.value.style.borderRadius = "8px";
    boxEl.value.style.textAlign = "center";
  }
});
</script>
```

3.3 Composition API 심화

컴포넌트의 v-model

- 개념
  - 양방향 데이터 바인딩을 위한 Vue 문법
  - 내부적으로 **:value** 와 **@input** 합친 기능 (속성 바인딩 + 이벤트 바인딩)

```
<template>
    v-model:    <input type="number" v-model="age" />
    실제 동작:<input type="number" :value="age"   @input="age = $event.target.value" />
</template>
```



3.3 Composition API 심화

컴포넌트의 v-model

- 활용 형태
  - 기본 입력 태그 : input, textarea, select 와 함께 자동 동작
  - 커스텀 컴포넌트: props(modelValue) 와 emit(update:modelValue)로 구성
  - 별칭 v-model : v-model:title, props(title) 와 emit(update:title)로 구성

- 문법
  - 기본
    - <input v-model="mesg" />
    - :value="mesg" 와 @input="mesg=$event.target.value" 합친 표현
  - **커스텀 컴포넌트**
    - <CustomInput **v-model="mesg"** />
    - **:modelValue="mesg" 와 @update:modelValue="newValue=>mesg=newValue" 합친 표현**
    - 자식 컴포넌트에서 **props:{modelValue}, emit('update:modelValue', mesg) 구성**

3.3 Composition API 심화

커스텀 컴포넌트 v-model

- 특징
  - 양방향 바인딩 제공
    - 부모에서 자식: props
    - 자식에서 부모: emit 이벤트
  - 컴포넌트 재사용성 증대
    - 부모에서 v-model만 지정하면 자식 컴포넌트의 내부 로직 몰라도 상태 관리 가능
  - 가독성과 생산성 향상
    - :value + @input 반복 문법 제거
  - 유연성/확장성
    - 별칭 v-model 을 통해 여러 상태를 한 컴포넌트에서 제어 가능
    - 단순 input 이외의 복잡한 컴포넌트 확장 구현

3.3 Composition API 심화

구현 코드

```
<!- App.vue →
<template>
  <main class="app">
    <h1>부모-자식 v-model 기본 예제</h1>

    <!-- v-model = :modelValue + @update:modelValue 의 축약형 -->
    <ChildInput v-model="message" />

    <p class="preview">
      부모가 가진 값: <strong>{{ message }}</strong>
    </p>
  </main>
</template>

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

// 부모가 관리하는 상태
const message = ref("안녕하세요");
</script>
```

```
<!-- ChildInput.vue -->
<template>
  <div class="child">
    <label >자식 입력창:
      <input :value="props.modelValue" @input="onInput" />
    </label>
  </div>
</template>
<script setup>
/*
  props: modelValue,  emit: update:modelValue
*/
const props = defineProps({
  modelValue: String, // 부모가 관리하는 값
});
const emit = defineEmits(["update:modelValue"]);

function onInput(e) {
  // 입력된 값을 부모에게 전달
  emit("update:modelValue", e.target.value);
}
</script>
```

3.3 Composition API 심화

Composable 패턴 기본

- 개념
  - 자주 사용하는 로직(함수,상태, 이벤트 처리 등)을 재사용 외부 함수로 분리

- 특징
  - 함수명은 useXxx 형식 ex. useCounter, useLoginForm
  - 상태 관리 가능 ex. ref,reactive
  - 라이프사이클 가능 ex. onMounted, onUnmounted

- 장단점
  - 코드 재사용 증가, UI와 로직 분리로 인한 유지보수 및 가독성 향상
  - 초기 학습 곡선 높음, 남용 시 오히려 복잡해짐

3.3 Composition API 심화

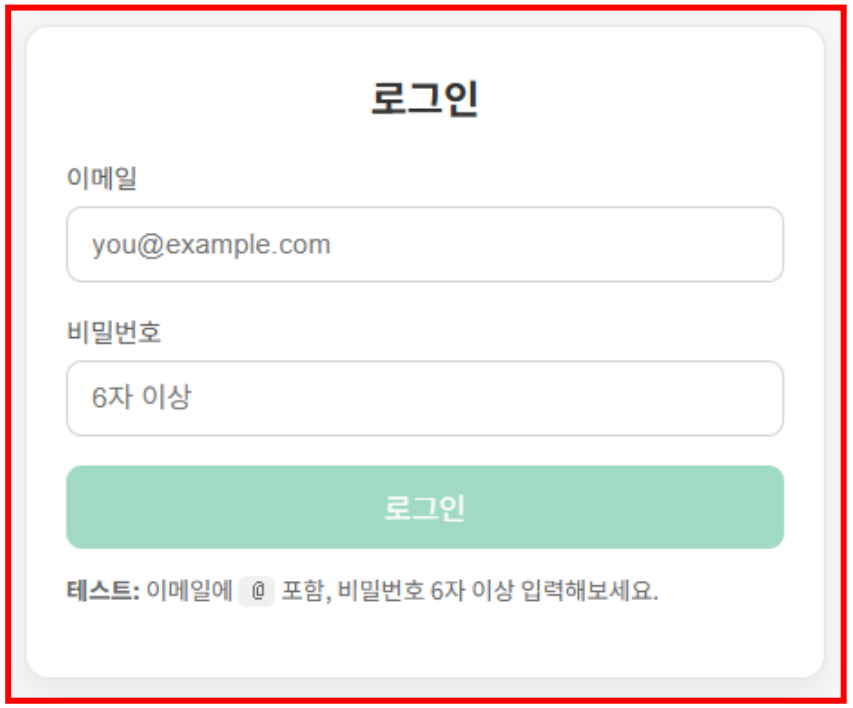
구현 코드 (기본)

```
<!- App.vue →
<template>
  <section class="container">
    <h1>{{ sayHello() }}</h1>
  </section>
</template>

<script setup>
import { useGreeting } from "@/composables/useGreeting";
const { sayHello } = useGreeting("안녕하세요");
</script>
```

```
// src/composables/useGreeting.js
export function useGreeting(name = "Vue") {
  const sayHello = () => `${name}!`;
  return { sayHello };
}
```

3.3 Composition API 심화

Composable 패턴 활용



App.vue

화면(UI) 담당

Ex. 버튼, 입력창, 에러 메시지등

useLoginForm.js (Composable 파일)

로직 담당

Ex. 이메일 형식, 비번 길이 확인 등

3.3 Composition API 심화

기능 비교

구분	App.vue	useLoginForm.js
역할	• 화면 UI 담당 Ex. 입력창, 버튼, 메시지	• 비즈니스 로직 담당 ex. 검증/상태/제출
장점	• 디자인(UI)에만 집중 가능	• 로직을 한 곳에 모아 관리
	• 구조가 단순하고 읽기 쉬움	• 규칙 변경 시 한 파일만 수정
	•필요 시 화면만 수정 •유지보수 향상	• 여러 컴포넌트에서 재사용

3.3 Composition API 심화

구현 코드 (패턴 활용)

```vue
<!- App.vue →
<template>
  <div class="page">
    <div class="card">
      <h1>로그인</h1>
      <form class="form" @submit.prevent="submit">
        <div class="form-group">
          <label>이메일</label>
          <input v-model="form.email" placeholder="you@example.com" />
          <p class="error" v-if="errors.email">{{ errors.email }}</p>
        </div>
        <div class="form-group">
          <label>비밀번호</label>
          <input    v-model="form.password"  type="password" placeholder="6자 이상" />
          <p class="error" v-if="errors.password">{{ errors.password }}</p>
        </div>
        <button class="btn" type="submit" :disabled="!isValid">로그인</button>
      </form>
    </div>
  </div>
</template>
```

```vue
<script setup>
// 별칭(@) 사용한 경우
import { useLoginForm } from "@/composables/useLoginForm";

// 별칭 미사용 (상대경로):
// import { useLoginForm } from "./composables/useLoginForm"

const { form, errors, isValid, submit } = useLoginForm();
</script>
```

3.3 Composition API 심화

구현 코드 (패턴 활용)

```
// src/composables/useLoginForm.js
import { reactive, computed, watch } from "vue";
export function useLoginForm() {
  // 폼 입력값
  const form = reactive({ email: "", password: "" });
  // 에러 메시지
  const errors = reactive({ email: "", password: "" });

  // 유효성 검사 함수
  function validate() {
    errors.email = form.email.includes("@") ? "" : "이메일 형식이 아닙니다";
    errors.password = form.password.length >= 6 ? "" : "비밀번호는 6자 이상";
    return !errors.email && !errors.password;
  }

  // 전체 유효 여부
  const isValid = computed(() => {
    // 버튼 활성화를 위해 간단히 체크 (공백 방지)
    return form.email.trim() && form.password.trim() && validate();
  });

  // 제출
  async function submit() {
    if (!validate()) return false;
    // 여기서 실제 로그인 API 호출 가능 (fetch 등)
    alert(`로그인 성공! 이메일: ${form.email}`);
    // 폼 초기화 (선택)
    form.email = "";
    form.password = "";
    return true;
  }

  return { form, errors, isValid, validate, submit };
}
```

3.4 상태 관리 패턴

Composition API 의 다양한 상태를 살펴보고 상황에 맞는 상태 관리 패턴 적용 방법을 알아보자.

학습목표

- 상태 관리 개요
- 상태 종류
  - 로컬 상태
  - 부모/자식 간 상태
  - 컴포넌트의 v-model 상태 (커스텀 컴포넌트)
  - 재사용 가능한 상태 (Composable 파일)

3.4 상태 관리 패턴

상태 관리

- 개념
  - 상태(state)는 화면에 보여질 데이터 의미
    ex. 입력폼 값, 체크 여부, API 로딩중 상태, 에러 메시지, 필터 조건들
  - 상태는 어디에 저장되고 어떻게 전달하며 언제 갱신되는지가 중요

- 상태 종류
  - 로컬 상태
  - 부모/자식 간 상태
  - 커스텀 컴포넌트의 v-model 상태
  - 재사용 가능한 상태

3.4 상태 관리 패턴

로컬 상태

- 개념
  - 가장 기본이 되는 상태
  - 단일 컴포넌트에서만 사용

- 특징/구현
  - 간단/직관적
  - 단일 값: ref
  - 객체/배열: reactive
  - 파생상태: computed
  - 부수효과: watch

3.4 상태 관리 패턴

구현 코드 (로컬)

```vue
<!-- App.vue -->
<template>
  <div class="container">
    <button class="btn" @click="count++">+1</button>
    <p class="counter">count: {{ count }}</p>

    <div class="form-group">
      <input v-model="form.name" placeholder="이름"
                          class="input" />
      <input v-model.number="form.age"
                           type="number" min="0" class="input" />
    </div>

    <p class="result">
      {{ form.name || "익명" }} 님은 성인인가요? →
      <span :class="isAdult ? 'yes' : 'no'">
        {{ isAdult ? "예" : "아니오" }}
      </span>
    </p>
  </div>
</template>
```

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

const count = ref(0); // 단일 값
const form = reactive({ name: "", age: 20 }); // 객체 상태

const isAdult = computed(() => form.age >= 20); // 파생 상태

watch(count, (n, o) => {
  console.log(`count: ${o} → ${n}`);
});
</script>
```

3.4 상태 관리 패턴

부모/자식 간 상태

- 개념
  - 부모에서 자식 : props
  - 자식에서 부모: emit
- 특징/구현
  - 데이터 흐름이 명확 (단방향)
  - 컴포넌트가 재사용되기 쉬움
  - props
  - emit

3.4 상태 관리 패턴

구현 코드 (부모/자식)

```vue
<!-- App.vue -->
<template>
  <div class="parent-container">
    <Child :title="title" :items="items" @add="handleAdd" />
  </div>
</template>
<script setup>
import { ref } from "vue";
import Child from "./components/Child.vue";

const title = ref("할 일 목록");
const items = ref(["Vue 공부", "운동"]);

function handleAdd(newItem) {
  items.value.push(newItem);
}
</script>
```

```vue
<!-- Child.vue -->
<template>
  <div class="child-container">
    <h3>{{ props.title }}</h3>
    <ul>
      <li v-for="(it, i) in props.items" :key="i">{{ it }}</li>
    </ul>
    <div class="form-group">
      <input v-model="input" placeholder="추가할 항목" />
      <button @click="add">추가</button>
    </div>
  </div>
</template>
<script setup>
import { ref } from "vue";
const props = defineProps({
   title: String,
   items: Array,
});
const emit = defineEmits(["add"]);
const input = ref("");
function add() {
  if (!input.value.trim()) return;
    emit("add", input.value);
    input.value = "";
}
</script>
```

3.4 상태 관리 패턴

커스텀 컴포넌트의 v-model 상태

- 개념
  - 부모가 v-model로 데이터를 전달
  - 자식은 modelValue props을 받음
  - 데이터 변경시 update:modelValue 를 emit 처리

- 특징/구현
  - 폼 컴포넌트 만들 시 표준 패턴
  - 부모 입장에서 사용이 간결
  - 부모

```
<CustomInput  v-model="mesg"  />
```

  - 자식

```
defineProps({ modelValue: String});
defineEmits(["update:modelValue"]);
```

3.4 상태 관리 패턴

구현 코드 (컴포넌트 v-model)

```vue
<!-- App.vue -->
<template>
  <div class="parent-container">
    <TextField v-model="name" label="이름" />
    <p class="result">
      입력값: <span>{{ name }}</span>
    </p>
  </div>
</template>
<script setup>
import { ref } from "vue";
import TextField from "./components/TextField.vue";

const name = ref("");
</script>
```

```vue
<!-- TextField.vue -->
<template>
  <div class="textfield">
    <label>
      <span class="label">{{ label }}</span>
      <input
        :value="modelValue"
        @input="(e) => emit('update:modelValue', e.target.value)"
      />
    </label>
  </div>
</template>
<script setup>
const props = defineProps({
  modelValue: String,
  label: String,
});
const emit = defineEmits(["update:modelValue"]);
</script>
```

3.4 상태 관리 패턴

재사용 가능한 상태

- 개념
  - useXxx 형태의 함수로 상태+로직 묶음의 외부 함수 (Composable 파일)
  - 이후 여러 컴포넌트에서 재사용

- 특징/구현
  - 관심사 분리 (UI와 비즈니스 로직 분리)
  - 테스트/ 유지보수 향상
  - Composable 파일: useXxx.js
  - Vue 컴포넌트: const f = useXxx()

3.4 상태 관리 패턴

구현 코드 (Composable 파일)

```vue
<!-- App.vue -->
<template>
  <main class="container">
    <h1>Counter 실습 (Composables)</h1>

    <section class="counter-box">
      <button @click="a.inc">A+1</button>
      <span>A: {{ a.count }}</span>
      <button @click="a.reset">Reset</button>
    </section>
    <section class="counter-box">
      <button @click="b.inc">B+1</button>
      <span>B: {{ b.count }}</span>
      <button @click="b.reset">Reset</button>
    </section>
  </main>
</template>
<script setup>
import { useCounter } from "@/Composables/useCounter";

// 각 버튼/카운터는 독립적인 상태를 가짐
const a = useCounter(0); // A 전용 상태
const b = useCounter(10); // B 전용 상태
</script>
```

```vue
// src/composables/useCounter.js
import { ref } from "vue";

export function useCounter(initial = 0) {
  const count = ref(initial);
  const inc = () => count.value++;
  const reset = () => (count.value = initial);
  return { count, inc, reset };
}
```

3.5 Provide/Inject 패턴

Provide/Inject 패턴의 구현 방법과 앞선 상태 패턴들과의 차이점을 알아보자.

학습목표

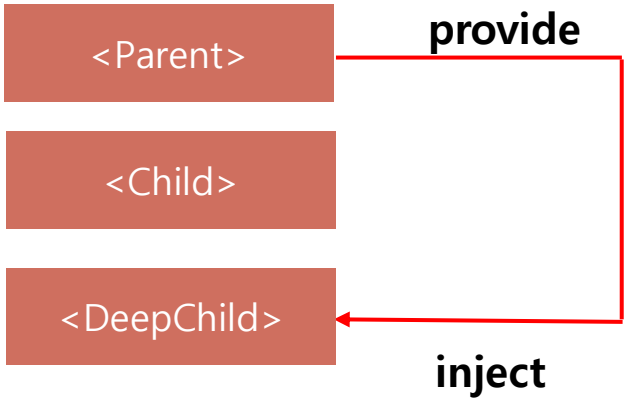
- Provide/Inject 패턴 개요
- 등장배경
  - Props drilling 방지
- 기본문법
  - Provide
  - Inject
- 문자열 vs Symbol

3.5 Provide/Inject 패턴

Provide/Inject

- 개념
  - 부모에서 데이터를 provide(제공)하고 자식에서 데이터를 inject(주입)해서 사용
  - 기존 props 처럼 데이터 전달하지만 중간 단계를 거치지 않고 깊은 곳에서 바로 사용

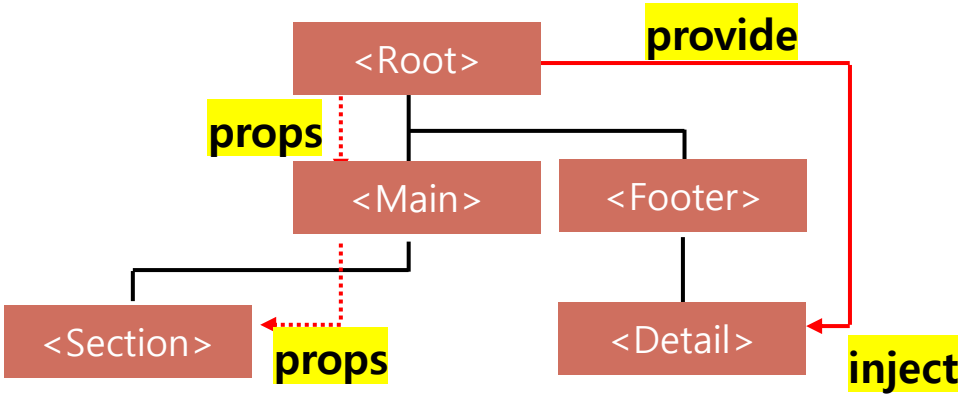
- 특징
  - 조상/자손 간 공유 (앱 전체 공유 X)
  - 반응성 유지 가능 (ref, reactive 이용)
  - 읽기 전용 보호 (자손이 임의로 변경 불가)
  - key 관리 : 문자열 보다 Symbol 권장(고유 ID)



3.5 Provide/Inject 패턴

등장 배경

- Props drilling 방지
  - 깊은 구조에서는 계속 props로 하위에 데이터 전달 필요
  - 중간 컴포넌트는 사용 없이 전달만 담당 (비효율적)
  - Provide/Inject 로 중간 전달 제거



3.5 Provide/Inject 패턴

기본 문법

- Provide
  - 역할: 조상 컴포넌트에서 값(데이터/함수)제공
  - 문법: provide(key, value)
  - key 형식
    - 문자열: "counter"
    - 심볼(Symbol) : Symbol("counter")
      고유성 보장으로 key 충돌 방지
  - value 형식
    - 원시값: 숫자, 문자열, 불리언 ex. provide('color', 'red')
    - 반응형 데이터: ref, reactive, ex. provide('count', ref(0))
    - 함수: ()=>{} , ex. provide('increment', ()=> count.value++)

3.5 Provide/Inject 패턴

기본 문법

- Inject
  - 역할: 조상에서 제공한 값을 하위에서 주입 받아서 사용
  - 문법: inject(key, defaultValue)
  - key 형식
    - 문자열: "counter"
    - 심볼(Symbol) : Symbol("counter")
    - 부모와 자식에서 key값이 동일
  - defaultValue 형식
    - 원시값: 숫자, 문자열, 불리언 ex. inject('color', 'red')
    - 객체: ref, reactive, ex. inject('user', {name:'Guest'})
    - 함수: ()=>{} , ex. inject('log', ()=> console.log('기본로그'))

3.5 Provide/Inject 패턴

구현 코드

```vue
<!-- App.vue -->
<template>
  <div class="card">
    <h1>App (조상)</h1>
    <p>이름을 입력하면 하위 컴포넌트에도 반영됩니다.</p>
    <input v-model="username" placeholder="이름 입력하기" />
  </div>

  <Child />
</template>
<script setup>
import { ref, provide } from "vue";
import Child from "./components/Child.vue";

// 반응형 데이터 준비
const username = ref("홍길동");

// provide로 하위 컴포넌트에 전달
provide("username", username);
</script>
```

```vue
<!-- Child.vue -->
<template>
  <div class="card">
    <h2>Child (중간)</h2>
    <p>여기는 값을 쓰지 않음 (단순 중간 단계)</p>
    <DeepChild />
  </div>
</template>
<script setup>
import DeepChild from "./DeepChild.vue";
</script>
```

```vue
<!-- DeepChild.vue -->
<template>
  <div class="card">
    <h3>DeepChild (자손)</h3>
    <p>  조상으로부터 받은 값: <strong>{{ username }}</strong></p>
  </div>
</template>
<script setup>
import { inject } from "vue";
// 조상(App.vue)에서 제공한 값 주입
const username = inject("username");
</script>
```

3.5 Provide/Inject 패턴

문자열 vs Symbol 비교

구분	문자열 key	Symbol key
고유성	• 전역적으로 중복될 수 있음 ex. 'x' === 'x' 결과는 true	• 유일성 보장 ex. Symbol('x') === Symbol('x') 결과는 false
충돌 위험	• 다른 라이브러리/컴포넌트에서 같은 key 사용할 경우, 충돌 가능성 존재	• 프로젝트 전역에서 동일한 이름이지만 실제 Symbol 객체가 다르면 충돌 불가
디버깅 편의성	• 문자열 그대로 표시되어 디버깅 시, 확인이 쉬움	• Symbol(x) 처럼 표시되어, 문자열보다 직관성은 떨어짐
코드 유지보수	• 큰 프로젝트에서 key 관리 어려움	• 고유성이 보장되어 유지보수 안정성 안정
실무 권장	• 소규모 프로젝트나 테스트용에는 무난	• 중대형 프로젝트, 라이브러리 개발 시 권장

3.5 Provide/Inject 패턴

구현 코드 (Symbol)

```
// keys.js Symbol 키 정의
export const USERNAME_KEY = Symbol("username");
```

```
<!-- App.vue -->
<template>
  <div class="card">
    <h1>App (조상)</h1>
    <p>문자열 key와 Symbol key를 동시에 제공</p>
    <input v-model="userStr" placeholder="문자열 username 수정" />
    <input v-model="userSym" placeholder="Symbol username 수정" />
  </div>
  <Child />
</template>
<script setup>
import { ref, provide } from "vue";
import Child from "./components/Child.vue";
import { USERNAME_KEY } from "./keys.js";

// 문자열 key 제공
const userStr = ref("홍길동 (App에서 문자열 제공)");
provide("username", userStr);
// Symbol key 제공
const userSym = ref("이몽룡 (App에서 Symbol 제공)");
provide(USERNAME_KEY, userSym);
</script>
```

3.5 Provide/Inject 패턴

구현 코드 (Symbol)

```vue
<!-- Child.vue -->
<template>
  <div class="card">
    <h2>Child (중간)</h2>
    <p>문자열 키는 App에서 제공한 값을 덮어써버림</p>
    <input v-model="userStr" placeholder="Parent 문자열 username 수정" />
    <DeepChild />
  </div>
</template>
<script setup>
import { ref, provide } from "vue";
import DeepChild from "./DeepChild.vue";

// 같은 문자열 키 'username' 다시 제공 → App 값 덮어쓰기
const userStr = ref("성춘향 (Child에서 문자열로 덮어씀)");
provide("username", userStr);
</script>
```

```vue
<!-- DeeoChild.vue -->
<template>
  <div class="card">
    <h3>DeepChild (자손)</h3>
    <p>
      문자열 키 값: <strong>{{ userStr }}</strong>
    </p>
    <p>
      Symbol 키 값: <strong>{{ userSym }}</strong>
    </p>
  </div>
</template>
<script setup>
import { inject } from "vue";
import { USERNAME_KEY } from "../keys.js";

// 문자열 키 주입 → Parent에서 덮어쓴 값만 보임
const userStr = inject("username");

// Symbol 키 주입 → App에서 제공한 값 안전하게 유지
const userSym = inject(USERNAME_KEY);
</script>
```

3.5 Provide/Inject 패턴

구현 코드 (Symbol)

실행 결과

App.vue
provide('username', "홍길동") ← 문자열
provide(USERNAME_KEY, "이몽룡") ← Symbol
Child.vue
provide('username', "성춘향") ← 문자열 덮어쓰기
DeepChild.vue
inject('username') → "성춘향" (App 값 덮어씀)
inject(USERNAME_KEY) → "이몽룡" (안전)

3.6 정리 및 실습

요약맵

- 컴포넌트 라이프 사이클
  - onMounted, onUnmounted, onErrorCaptured 주로 사용
- Composition API
  - <script setup> 컴파일 방식
  - 반응성 (reactivity) 개념 및 ref, reactive, computed, watch
- 상태 관리
  - props/emit
  - 커스텀 컴포넌트의 v-model
  - 재사용 (Composable 파일)
- Provide/Inject
  - Props drilling 방지
  - 문자열 vs Symbol

수고하셨습니다.