처음 배우는 훈련생을 위한 react 가이드

by JKR

1. 1~2주차: JavaScript 보강 (60시간)

가. Week 1 - JS 기본기 다지기

- ▶ 자바스크립트 소개 및 개발환경 점검
- ▶ 변수와 자료형 (let, const, var, Number, String, Boolean 등)
- ▶ 함수와 스코프 (선언, 표현식, 화살표 함수)
- ▶ 객체와 배열 기초
- ▶ 배열 메서드 (map, filter, reduce)
- ▶ ES6 문법 (템플릿 리터럴, 구조분해, 스프레드 등)
- ▶ this와 바인딩 이해
- ▶ 종합 실습: ToDo 리스트 CRUD 구현
- ▶ 주간 리뷰 & 퀴즈

나. Week 2 - DOM & 브라우저 API

- ▶ DOM(Document Object Model) 이해
- ▶ DOM 요소 선택과 조작 (innerText, innerHTML, classList, style)
- ▶ 이벤트 핸들링 (addEventListener, 이벤트 객체)
- ▶ localStorage 활용 (데이터 저장/불러오기, JSON.stringify/parse)
- ▶ fetch API 기초 (HTTP 요청/응답, 공공 API 연동)
- ▶ JSON 데이터 다루기 (형식, 변환)
- ▶ async/await 비동기 처리 및 에러 핸들링
- ▶ 미니 프로젝트: 간단한 영화 검색 앱

2. 3~4주차: React 기초 & 심화 (60시간)

가. Week 3 - React 기초

- ▶ Vite + React 환경 세팅
- ▶ JSX 문법과 표현식
- ▶ 컴포넌트 구조와 Props
- ▶ State 개념과 활용
- ▶ 이벤트 핸들링 in React
- ▶ 조건부 렌더링 & 리스트 렌더링(map)
- ▶ 실습: 카운터 앱, 출석부 앱

나. Week 4 - React 심화 + API 연동

- ▶ useEffect 개념과 데이터 가져오기
- ▶ fetch API 활용 in React
- ▶ 로딩/에러 처리 패턴
- ▶ 외부 API 실습 (날씨, 영화)
- ▶ JSON 데이터 테이블 출력

- ▶ JSON CRUD 시뮬레이션
- ▶ 종합 실습: API 기반 리스트 앱

3. 5주차: 데이터 시각화 (40시간)

가. Week 5 - Chart.js & Recharts

- ▶ Chart.js 소개 및 설치
- ▶ 기본 차트: 막대, 선, 파이 그래프
- ▶ Recharts 소개 및 실습 (BarChart, LineChart, PieChart)
- ▶ 차트 커스터마이징 (색상, 라벨, 툴팁 등)
- ▶ 종합 실습: 지출/수입 대시보드 프로젝트

4. 6~7주차: Firebase (80시간)

가. Week 6 - Firebase 기초 (Firestore CRUD)

- ▶ Firebase 소개 및 프로젝트 생성
- ▶ SDK 연결과 보안 규칙 이해
- ▶ Firestore 구조 설계
- ▶ 데이터 읽기/쓰기/수정/삭제
- ▶ React + Firestore CRUD 연동
- ▶ 미니 프로젝트: 출석 관리 DB 앱

나. Week 7 - Firebase 심화 (Auth, Storage, Hosting)

- ▶ Firebase Authentication (회원가입, 로그인)
- ▶ React와 Auth 상태 관리
- ▶ Firestore + Auth 연계
- ▶ Firebase Storage (이미지 업로드)
- ▶ Hosting (Firebase 배포)
- ▶ 종합 실습: 출석·성적 관리 웹앱

React 7주 완성 세부 시간표					
	9월 30일	**10월 1일**	**10월 2일**	**10월 10일**	**10월 13일**
오전	JS 기본 (let/const, 자료형)	함수/스코프	객체/배열 기초	배열 메서드 map/filter, 고차 함수	JS 미니 프로젝트(ToDo CRUD)
오후	ES6 문법(화살표 함수, 템플릿)	this & 바인딩	구조분해/스프레드		
	10월 14일	**10월 15일**	**10월 16일**	**10월 17일**	**10월 20일**
오전	DOM 조작(querySelector)	localStorage 활용	JSON(parse/stringify)	Vite + React 환경 세팅	JSX 문법
오후	이벤트(addEventListener)	fetch API 실습(날씨)	async/await	Hello React 실습	JSX 표현식/조건부 렌더링
	10월 21일	**10월 22일**	**10월 23일**	**10월 24일**	**10월 27일**
오전	컴포넌트 & Props	State 개념	이벤트 핸들링	useEffect 소개	fetch API (React)
오후	리스트 렌더링(map)	카운터 앱	미니 프로젝트(출석부)	JSON CRUD 시뮬	API 실습(영화)
	10월 28일	**10월 29일**	**10월 30일**	**10월 31일**	**11월 3일**
오전	로딩/에러 처리	외부 API 실습(날씨)	JSON 데이터 출력	Chart.js 소개 & 설치	막대 그래프
오후	조건부 렌더링 실습	JSON + 테이블 렌더링	미니 프로젝트(API 기반 리스트)	Recharts 소개	BarChart 실습
	11월 4일	**11월 5일**	**11월 6일**	**11월 7일**	**11월 10일**
오전	선 그래프	파이 차트	종합 차트 실습	Firebase 소개/세팅	Firestore 구조
오후	LineChart 실습	PieChart 실습	미니 프로젝트(지출 대시보드)	SDK 연결	React + Firestore 연동
	11월 11일	**11월 12일**	**11월 13일**	**11월 14일**	**11월 17일**
오전	데이터 읽기	데이터 쓰기	Firebase 콘솔 실습	Auth 개요	회원가입/로그인 구현
오후	CRUD 추가	CRUD 수정/삭제	미니 프로젝트(출석 DB)	React + Auth 상태 관리	Auth + Firestore 연동
	11월 18일	**11월 19일**	**11월 20일**		
오전	Storage 소개	이미지 업로드	Hosting 배포		
오후	Storage + DB 연계	CRUD 종합 실습	미니 프로젝트(출석·성적 관리 앱)		

- Day 1 -

1. 자바스크립트 소개

- 웹 페이지를 동적으로 만들기 위해 사용하는 대표 프로그래밍 언어
- HTML → 구조, CSS → 디자인/스타일, JS → 동작/기능
- 현재는 브라우저뿐만 아니라 Node.js 환경을 통해 서버에서도 사용됨
- 특징
  - HTML문서내에 포함되어 해석됨
  - 이벤트에 반응하여 처리가 가능함
  - 브라우저 프로그래밍을 통해 DB정보와 전자상거래, CGI를 대체할 수 있음
  - 클래스 기반의 프로그래밍 가능
  - 자바스크립트의 라인구분 : 세미콜론(;) 사용
  - 변수 선언을 하지않아도 바로 사용가능
  - 사용자 이벤트 제어 가능
  - DOM 사용가능
  - 클라이언트 측 객체지향 스크립트 언어
- 자바스크립트의 활용

  자바스크립트의 삽입 : 인라인 형태, 스크립트 태그를 사용하는 형태, 외부 파일을 지정하는 방법

  ○ 인라인 형태 : 이벤트와 함께 사용

  -
  - test

  ○ 스크립트 태그를 사용하는 방법

  - 1 -- <script language="javascript"> </script>
  - <script type="text/javascript"> </script>

  ○ 외부 파일을 지정하는 방법

  - <script language="javascript" src="스크립트파일.js"> </script>

  ○ 주석처리 : (한줄주석) //, (여러라인 주석) /*... */, (HTML 주석) <!-- ... -->

2. JavaScript의 데이터 타입

구분	데이터 타입	내용
1	number	정수, 실수등의 숫자 타입
2	boolean	true, false를 표시
3	string	문자열 표시
4	object	복합데이터 타입
5	undefined	정의되지 않은 타입

3. 변수 선언

- 변수 : 메모리에 데이터를 저장할 수 있도록 저장공간을 확보하는 것
- 변수가 메모리에 할당되면, 그 공간은 같은 타입의 데이터를 지속적으로 변경하여 저장할 수 있음.
- 변수는 임의의 장소에 할당되며, 숫자형 주소가 지정됨.
- JavaScript는 변수의 타입검사 등을 하지 않는다.
- 변수의 선언이 없이 사용을 해도 된다.
- 자바스크립트에서 변수를 선언하는 방법은 3가지가 있습니다.

```
let name = "홍길동";    // 변경 가능 (블록 스코프)
const age = 25;          // 변경 불가 (상수)
var old = "옛날 방식"; // 함수 스코프 (권장 X)
```

let : 값 변경 가능, 블록 스코프 (가장 많이 사용)
const : 값 변경 불가, 반드시 선언과 동시에 초기화해야 함
var : 과거 방식, 함수 스코프라 버그 유발 가능 → 사용 지양

```
let a = 10;
a = 20;      // 가능
const b = 30;
// b = 40;  //     에러 발생 (const는 변경 불가)
```

4. 상수

- 상수 개념 : 메모리에 데이터를 저장할 수 있도록 저장공간을 확보하는 것
- 상수가 메모리에 생성되면 그 공간은 변경하여 저장할 수가 없음
- 상수의 종류 : 숫자형 상수, 문자형 상수, NaN(Not a Number), null
- 리터럴 : 표현 가능한 데이터(10,203, 1.23, true, ‘test string’, ‘훈민정음’)

5. 자료형

- 원시 자료형 (Primitive Type)
- Number → 모든 숫자 (정수, 실수)
- String → 문자열
- Boolean → true/false
- null → “의도적으로 비어있음”
- undefined → 값이 할당되지 않음
- Symbol → 고유한 값 (잘 사용 안 함, 고급 주제)
- BigInt → 매우 큰 정수 표현

```
let num = 10;              // Number
let text = "안녕하세요";  // String
let flag = true;          // Boolean
```

```
let empty = null;          // null
let unknown;               // undefined (값을 안 넣었음)

console.log(typeof num);       // number
console.log(typeof text);      // string
console.log(typeof flag);      // boolean
console.log(typeof empty);     // object (JS의 오래된 버그)
console.log(typeof unknown);   // undefined
```

6. JavaScript 데이터 타입 변환

- Number(String str) : 문자열을 숫자로 변환
- 지정된 매개변수가 숫자가 아닌 값이라면 NaN(Not a Number)값 출력
- String(숫자) : 숫자를 문자열로 변환
- 변수.toString() : 변수를 문자열로 변환
- parseInt(var str), parseFloat(var str) : 문자열을 숫자로 변환
- 단, 숫자형 문자를 변환시키는 것이 의미가 있다. (123a → 123)
- Boolean(변환값) : 변환할 값이 0, NaN, “”, null, undefined일 경우를 제외하고는 true

7. 연산자

- 산술 연산자

```
console.log(5 + 3);  // 8
console.log(5 - 3);  // 2
console.log(5 * 3);  // 15
console.log(10 / 2); // 5
console.log(10 % 3); // 1 (나머지)
```

- 비교 연산자

```
console.log(5 == "5");    // true (값만 비교, 타입 무시)
console.log(5 === "5");   // false (값 + 타입 모두 비교)
console.log(10 != "10"); // false
console.log(10 !== "10");// true
```

■ 논리 연산자

```
console.log(true && false); // false (AND)
console.log(true || false); // true  (OR)
console.log(!true);             // false (NOT)
```

■ 조건 연산자(삼항연산자)

```
let result = 10 > 5 ? “5보다 크다” : “5보다 작다”;
(조건) ? (A) : (B);
```

8. 템플릿문자열

■ 백틱(`)을 사용하면 문자열 안에 변수를 쉽게 넣을 수 있다

```
let name = "철수";
let age = 20;

console.log("안녕하세요. 저는 " + name + "이고 나이는 " + age + "살입니다.");
console.log(`안녕하세요. 저는 ${name}이고 나이는 ${age}살입니다.`); // 추천 방식
```

◆ **실습 문제**

1. BMI 계산기 만들기

- **변수 height(m), weight(kg)를 선언**
- **BMI = weight / (height * height) 계산**
- **BMI 값 출력**

2. 장바구니 합계 계산

```
let item1 = 12000;
let item2 = 3500;
let item3 = 8000;
```

- **총합을 구해 출력**
- **만약 총합 >= 20000 이면 "무료배송" 출력**
- **아니면 "배송비 3000원 추가" 출력**

3. 짝수/홀수 판별기

- **변수 num에 숫자 하나 저장**
- **% 연산자를 이용해 "짝수" 또는 "홀수" 출력**

4. 숫자 문자열 더하기

- **변수 num1 = "10", num2 = "20" 선언**
- **이 두 값을 더해서 "1020"이 출력되도록 하고**
- **다시 숫자로 변환하여 합이 30이 출력되도록 하세요.**

5. 문자열 다루기 심화

- **변수 fullName = "홍길동"**
- **템플릿 문자열을 이용해 "안녕하세요, 저는 홍길동입니다." 출력**
- **문자열 길이를 출력 → "이름의 길이는 3입니다."**
- **첫 글자만 추출해서 "성은 홍입니다." 출력**
- **문자열을 거꾸로 뒤집어 "동길홍" 출력**

- Day 2 -

9. 함수 선언

- 특정 동작을 묶어서 재사용할 수 있게 만든 코드 블록
- return 키워드를 만나면 함수 실행이 종료되고 값을 반환

```
function add(a, b) {
  return a + b;
}
console.log(add(2, 3)); // 5
console.log(add(10, 20)); // 30
```

- 함수 선언은 호이스팅 되어 코드 위쪽에서 호출해도 실행 가능

```
console.log(square(4)); // 16 (함수 정의보다 위에서 호출 가능)

function square(x) {
  return x * x;
}
```

10. 스코프

- 변수의 유효 범위
- 어디서 변수를 사용할 수 있는지를 결정함
- 블록 스코프

```
let x = 10;

if (true) {
  let x = 20;
  console.log(x); // 20
}

console.log(x); // 10
```

- 함수 스코프

```
function test() {
  let y = 30;
  console.log(y);
}
test();
// console.log(y); //     에러 (함수 밖에서는 접근 불가)
```

11. 함수 표현식

- 함수도 값이 될수 있음 → 변수에 저장 가능

```
const multiply = function(a, b) {
  return a * b;
};
console.log(multiply(3, 4)); // 12
```

- 함수 표현식은 호이스팅이 되지 않음 → 반드시 정의후 호출해야함 (권장)

12. 화살표 함수

- ES6부터 추가된 함수 표현식 문법으로, 더 간결하게 작성할 수 있습니다

```
// 일반 함수
function add(a, b) {  return a + b; }
// 화살표 함수
const add = (a, b) => {  return a + b;  };
```

- 한 줄일 때는 return 생략 가능

```
const square = x => x * x;
console.log(square(5)); // 25
```

- 매개변수가 없을 때

```
const hello = () => console.log("안녕하세요!");
hello();
```

- 객체 반환시 괄호 필요

```
// 객체를 바로 반환할 때는 ()로 감싸야 함
const makeUser = (name, age) => ({ name, age });
console.log(makeUser("철수", 20));  // { name: "철수", age: 20 }
```

■ this와의 차이점

- 일반 함수 : 호출 방식에 따라 this가 달라짐
- 화살표 함수 : 자신을 감싸는 외부 스코프의 this를 그대로 사용

```
let user = {
  name : "홍길동",
  normalHello : function() {
    console.log("안녕 나는 " + this.name);
  },
  arrowHello : () => {
    console.log("안녕 나는 " + this.name);
  }
};
user.normalHello(); // "안녕 나는 홍길동"
user.arrowHello();  // "안녕 나는 undefined" (this를 user로 바인딩하지 않음)
```

◆ 실습문제

1. 두 수를 더한 결과를 반환하는 add 함수 만들기

2. 세 수의 평균을 구하는 함수 (함수 표현식으로 작성)

3. 매개변수로 이름을 받아 "안녕, ○○!" 출력하는 화살표 함수 작성

4. 블록 스코프와 함수 스코프 차이를 실험해보기

- **let으로 블록 안과 밖에서 같은 이름 변수 선언 후 출력 비교**
- **함수 안에서 선언한 변수가 밖에서 보이는지 확인**

5. 객체 안에 sayHi 메서드를 추가하고, this를 사용해 "나는 ○○입니다." 출력

- Day 3 -

13. 객체

■ 객체란

- 키(key) : 값(value) 쌍을 모아 놓은 자료형
- 예 : 학생 한명의 이름/나이/전공/점수 등을 한 덩어리로 담기 좋다.

```
const student = { name : "철수", age : 20 };
```

■ 프로퍼티 읽기/쓰기

- 점 표기법과 대괄호 표기법 2가지

```
console.log(student.name);        // "철수"  (점 표기법)
console.log(student["age"]);     // 20       (대괄호 표기법)

student.major = "Computer";       // 추가
student["grade"] = 3.7;           // 추가
student.age = 21;                  // 수정
delete student.grade;             // 삭제
```

- 언제 대괄호 표기법을 쓰나?
  - 키가 변수로 동적일 때
  - 띄어쓰기/특수문자가 있는 키일 때(권장하지 않음)

```
const key = "home-town";
const person = {};
person[key] = "Seoul";    // OK
// person.home-town = ... //     - (연산자로 해석되어 에러)
```

■ 계산된 프로퍼티 이름

```
const field = "score";
const s = { name : "영희", [field] : 95 }; // { name : "영희", score : 95 }
```

■ 단축 프로퍼티

- 변수명과 키 이름이 같다면 축약 가능

```
const name = "민수";
const age = 23;
const user = { name, age }; // { name : "민수", age : 23 }
```

■ 객체 검사 & 안전 접근

```
console.log("name" in user);            // true/false
console.log(Object.hasOwn(user, "age")); // true/false (권장)

// 안전 접근(옵셔널 체이닝)
const city = user.address?.city;       // address가 없으면 undefined 반환, 에러 X

// 기본값(널 병합)
const nickname = user.nickname ?? "손님"; // null/undefined면 "손님"
```

■ 얕은 복사 vs 깊은 복사

```
const a = { nested: { x: 1 } };

// 얕은 복사(1단계만 복사)
const b = { ...a };
b.nested.x = 999;
console.log(a.nested.x); // 999 (같은 참조라 함께 변함)

// 깊은 복사(전부 새로 복사)
const c = structuredClone(a); // 최신 브라우저/Node에서 지원
c.nested.x = 123;
console.log(a.nested.x); // 999 (영향 없음)

// 함수/Date/LargeInt 등 손실
const d = JSON.parse(JSON.stringify(a));
```

14. 배열

■ 배열은 여러개의 데이터를 한 줄로 차곡차곡 담아놓는 상자이다.

■ 자바스크립트에서 배열은 Array객체로, 순서가 있는 컬렉션을 표현한다.

■ 여러값을 하나의 변수로 묶어서 다룰 수 있게 해주는 자료 구조이다.

■ 컬렉션이란 여러개의 항목들을 모아놓은 것, 배열은 항목드의 순서(Index)을 기억하고, 인덱스를 통해 항목에 접근할 수 있습니다.

■ 배열 생성

- 리터럴 표기법

```
const fruits = ["사과", "바나나", "포도"];
```

- 생성자 방식

```
const arr = new Array("사과", "바나나", "포도");
```

■ length 속성
- 배열의 항목 개수를 저장하고 있는 속성
- 배열에 항목을 추가/제거하면 자동으로 변함

```
console.log(fruits.length); // 3
```

■ 인덱스(Index)와 요소(Element)
■ 배열 요소는 0,1,2,... 순서 있는 번호(인덱스)로 식별됨. 첫 번째 요소 인덱스는 0.
■ 만약 잘못된 인덱스를 쓰면 undefined가 나옴

```
console.log(fruits[5]); // undefined
```

■ 자주 쓰이는 변경 메서드와 비변경 메서드
- 변경 메서드 : push, pop, shift, unshift, splice, sort, reverse
- 비변경 메서드 : slice, map, filter, concat, toSorted, toReversed

```
// 변경 예시
const arr = [1, 2, 3];
arr.push(4);            // [1,2,3,4] (원본 변경)
arr.splice(1, 1, 99); // [1,99,3,4]

// 비변경 예시
const arr2 = [1, 2, 3];
const arr3 = arr2.slice(0, 2);    // [1,2], arr2 그대로
const arr4 = arr2.map(n => n*2); // [2,4,6], arr2 그대로
```

■ 검색 / 탐색

```
const nums = [5, 10, 15, 20];
nums.includes(10);                    // true
nums.indexOf(15);                     // 2
nums.find(n => n > 12);               // 15 (조건 만족하는 첫 값)
nums.findIndex(n => n > 12);          // 2
```

■ 정렬

```
const n = [10, 2, 5];
n.sort();                    // ["10","2","5"] 문자열 기준 → [10, 2, 5] 그대로처럼 보일 수 있음
n.sort((a, b) => a - b);  // 숫자 오름차순 → [2,5,10]

// 원본 보존하고 싶다면 (최신)
const sorted = n.toSorted((a, b) => a - b);
```

■ 배열 합치기 / 복사

```
const a1 = [1, 2];
const a2 = [3, 4];
const merged = a1.concat(a2);    // [1,2,3,4]  (비파괴)
const merged2 = [...a1, ...a2]; // 스프레드

const copy = [...a1];              // 얕은 복사
```

■ 배열의 장점

- 여러 값을 하나의 변수로 관리 → 코드 간결화
- 반복문을 통한 일괄 처리 가능
- 인덱스를 통한 빠른 접근
- 다양한 내장메서드로 가공이 쉬움

■ 배열의 한계

- 중간 삽입/삭제가 느릴 수 있음
- 매열 앞에서 요소를 삭제하면, 뒤의 요소들을 모두 한 칸씩 당겨야 함
- 크기가 동적으로 늘어나지만, 메모리 관리 측면에서는 비효율적일 수 있음
- 대량의 데이터에 특정 조건 탐색이 많을 경우, 다른 자료구조가 더 적합

15. 구조분해 할당 & 스프레드

■ 구조분해

- 배열이나 객체에서 값을 꺼내 변수에 간단히 담을 수 있는 문법
- 구조를 분해해서 변수에 할당한다는 의미

■ 객체 구조 분해

```
const student = { name: "철수", age: 20, major: "CS" };
const { name, age } = student;              // 같은 변수명으로 꺼내기
const { major: m = "미정" } = student;  // 이름 바꾸기 + 기본값
```

■ 배열 구조분해

```
const fruits = ["사과", "바나나", "포도"];
const [first, second] = fruits;          // "사과", "바나나"
const [head, , tail] = fruits;           // "사과", "포도" (중간 생략)
const [x, y = "기본"] = ["값"];            // x="값", y="기본"
```

■ 함수 매개변수 구조분해

```
function printUser({ name, age }) {
  console.log(`${name}(${age})`);
}
printUser({ name: "영희", age: 21 });
```

■ 스프레드 문법

- ... 점 세 개를 사용
- 배열이나 객체를 펼친다는 의미

■ 스프레드 vs 레스트

- 스프레드 : 펼치기 → 복사/합치기
- 레스트 : 나머지 모으기 → 파라미터/구조분해에서 잔여값 모으기

```
// 스프레드
const base = { a: 1, b: 2 };
const ext  = { ...base, b: 99, c: 3 }; // { a:1, b:99, c:3 }

// 레스트(객체)
const user = { id: 1, name: "철수", age: 20 };
const { id, ...profile } = user; // id=1, profile={ name:"철수", age:20 }

// 레스트(함수 매개변수)
function sum(...nums) {
  return nums.reduce((a, c) => a + c, 0);
}
sum(1,2,3,4); // 10
```

16. Tip

■ sort()는 원본 배열을 바꾼다 → 원본 보존하고 싶으면 toSorted()(최신) 또는 slice().sort()로 복사 후 정렬.

■ 스프레드는 얕은 복사라서 중첩된 값은 같이 변gka → 깊은 복사 필요 시 structuredClone 사용.

■ for...in은 객체 키 순회, 배열에는 for...of나 forEach/map 사용을 권장.

■ 없는 경로 접근 시 obj.a.b는 에러 → obj.a?.b처럼 옵셔널 체이닝으로 안전하게.

■ null과 undefined에 기본값 주려면 ??(널 병합) 사용. (||는 0/빈문자열도 거짓으로 봄)

- Day 4 -

17. 조건문

- 조건문은 프로그램이 상황에 따라 다른 동작을 하도록 분기 시키는 구문
- if 문

- if : 조건이 맞으면 실행
- else if : 앞 조건이 틀리면 다른 조건 검사
- else : 위 조건이 모두 틀리면 실행

```
let score = 85;

if (score >= 90) {
   console.log("A 학점");
} else if (score >= 80) {
   console.log("B 학점");
} else {
   console.log("C 학점 이하");
}
```

- 만약(if) ~ 라면 .. 하고, 아니면 .. 한다 라는 뜻
- 우리가 일상에서 쓰는 말 그대로

- 만약 비가 온다면 우산을 챙긴다.
- 그렇지 않으면 우산을 안챙긴다.

```
let isRaining = true;

if (isRaining) {
   console.log("우산을 챙긴다 ");
} else {
   console.log("우산 필요 없음 ");
}
```

```
조건 검사 → true → 실행1
          → false → 실행2
```

- pc방 요금제 : 만약(if) 돈이 1000원 이상이면 → 이용가능, 그렇지 않으면 잔액 부족

- switch 문
- 여러 경우의 수를 처리할 때 깔끔함
- break를 안 쓰면 뒤까지 실행되는 fall-through 현상이 생김(실무 버그 원인)

```
let fruit = "사과";

switch (fruit) {
  case "사과":
    console.log("□ 사과입니다");
    break;
  case "바나나":
    console.log("□ 바나나입니다");
    break;
  default:
    console.log("알 수 없는 과일");
}
```

18. 반복문

- 반복문은 코드 블록을 여러번 반복 실행하는 구조
- for문

- 초기값 let i=0
- 조건식 검사 i < 5 → false면 종료
- { } 블록 코드 실행
- 증감식 i++
- 조건식 검사

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

- while문
- 조건이 true인 동안 반복

```
let n = 0;
while (n < 3) {
  console.log(n);
  n++;
}
```

- doe ~ while 문
- 무조건 한 번은 실행 후 조건 검사

```
let num = 0;
do {
  console.log(num);
  num++;
} while (num < 3);
```

■ for...of

- 배열 순회

```
const fruits = ["사과", "바나나", "포도"];
for (let f of fruits) {
  console.log(f);
}
```

■ for...in

- 객체속성을 순회

```
const student = { name: "철수", age: 20 };
for (let key in student) {
  console.log(key, student[key]);
}
```

19. 배열 고차함수

■ 고차함수(Higher-Order Function) = 다른 함수를 인자로 받거나 결과로 반환하는 함수

■ 배열에서 자주쓰이는 대표 고차함수 : forEach, map, filter, reduce

■ **forEach (단순반복)**

- 반환값 없음 , 단순히 모든 요소에 대해 실행

```
const numbers = [1, 2, 3];
numbers.forEach(n => {
  console.log(n * 2);
});
```

■ **map (새 배열 만들기)**

- 원본배열을 변형하지 않고 새 배열 반환

```
const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);
console.log(doubled); // [2, 4, 6]
```

■ filter (조건으로 거르기)

- 조건을 만족하는 요소만 모아 새 배열 생성
- 검색, 조건 필터링 구현에 유용

```
const scores = [45, 80, 90, 30];
const passed = scores.filter(s => s >= 60);
console.log(passed); // [80, 90]
```

■ reduce (누적 계산)

■ 배열전체를 하나의 값으로 줄임

- acc : 누적 값
- cur : 현재요소
- 0 : 초기값

■ 합계, 평균, 그룹화 등 집계 연산에 필수

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, cur) => acc + cur, 0);
console.log(sum); // 10
```

■ find , findIndex(찾기)

■ 조건을 만족하는 첫 번째 요소 또는 인덱스 찾기

```
const users = [{id:1,name:"철수"},{id:2,name:"영희"}];
const user = users.find(u => u.id === 2);
console.log(user); // {id:2, name:"영희"}
```

- Day 5 -

오늘 할일 미니 프로젝트

- Day 6 -

1. DOM 이란 무엇인가?

- **DOM (Document Object Model)** = 웹 페이지를 자바스크립트가 다룰 수 있도록 구조화한 “트리(tree) 형태의 모델”
- HTML 요소 하나하나가 노드(node)로 표현됨.
- JS에서는 DOM을 통해 요소를 **찾고(선택)**, **바꾸고(조작)**, **추가/삭제**할 수 있음.
- **쉽게 말하면**
  웹 브라우저는 집 설계도(HTML)를 보고 실제 집(DOM)을 지어둡니다.
  자바스크립트는 이 집에 들어가서 “불 켜기/끄기, 가구 옮기기, 벽 칠하기” 같은 일을 할 수 있는 **주인공**이에요.

2. DOM 요소선택 (Element Selection)

- DOM에 있는 HTML 요소를 먼저 **찾아야** 조작할 수 있습니다.

가) 기본 선택

```
document.getElementById("title");             // id로 선택
document.getElementsByClassName("item"); // class로 선택
document.getElementsByTagName("p");         // 태그 이름으로 선택
```

나) css 선택자 사용 (추천 방식)

```
document.querySelector("#title");    // id 선택
document.querySelector(".item");      // class 선택
document.querySelectorAll("p");       // 모든 &p& 선택
```

- querySelector / querySelectorAll은 CSS 선택자를 그대로 쓸 수 있어서 React나 현대 개발에서 가장 많이 쓰임.
- querySelector는 **CSS 선택자와 똑같이** 쓰기 때문에 배우기 쉽습니다.
- Tip: 실무에서도 거의 무조건 querySelector 씁니다.

3. DOM 요소 조작 (Manipulation)

가) 텍스트/HTML 변경

```
let title = document.querySelector("#title");
title.innerText = "새로운 제목";      // 순수 텍스트만 변경
title.innerHTML = "&b&굵은 제목&/b&"; // HTML 태그 포함 가능
```

나) 속성 변경

```
let img = document.querySelector("img");

img.setAttribute("src", "dog.png");
console.log(img.getAttribute("src"));
```

다) 스타일 변경

```
let box = document.querySelector(".box");

box.style.color = "red";
box.style.backgroundColor = "yellow";
```

- 실무에서는 CSS 클래스(classList.add/remove/toggle)를 조작하는 방법을 더 권장

```
box.classList.add("active");
box.classList.remove("hidden");
box.classList.toggle("dark-mode");
```

4. 이벤트와 이벤트 핸들링

가) 이벤트란?

- **이벤트(Event)** = 사용자가 하는 행동
- 클릭, 입력, 스크롤, 마우스 올리기, 키보드 누르기 등
- **이벤트 핸들링(Event Handling)** = “어떤 일이 일어났을 때 실행할 코드”를 연결하는 것

나) 이벤트 등록

```
let btn = document.querySelector("#myBtn");

btn.addEventListener("click", function() {
  alert("버튼이 눌렸습니다!");
});
```

다) 실무에서는 화살표 함수도 자주 사용

```
btn.addEventListener("click", () =& {
  console.log("버튼 클릭됨");
});
```

라) 대표적인 이벤트 종류
- click : 버튼 클릭
- input : 입력창에 글자 입력
- change : 선택 값 변경 (예: 드롭다운)
- keydown : 키보드 입력
- submit : 폼 제출

마) 이벤트 객체 (event object)

바) 이벤트가 발생하면, 브라우저는 자동으로 이벤트 정보를 담은 객체를 전달함.

```
document.querySelector("#myInput")
  .addEventListener("keydown", function(e) {
    console.log(e.key); // 어떤 키 눌렀는지 출력
  });
```

5. 생활 속 비유

■ DOM = **아파트**
- 각 집(요소)을 찾아서, 페인트칠(스타일), 간판 바꾸기(속성), 글자 갈아끼우기(innerText) 할 수 있음

■ 이벤트 = **초인종**
- 누군가 초인종을 누르면(이벤트 발생) → 주인이 나와서 행동(함수 실행)

■ 이벤트 핸들링 = **자동화 장치**
- “초인종을 누르면 자동으로 문 열어주기”처럼, 조건에 따라 자동 실행되는 코드

6. 알아두면 좋은 Tip

가) DOM은 그림으로 이해
- HTML 태그 하나하나를 “박스”라고 생각하고, 트리처럼 이어져 있다고 상상

나) querySelector는 CSS랑 똑같다고 기억
- #id, .class, tag → 웹디자인 수업에서 CSS를 했던 사람은 금방 적응함

다) innerText vs innerHTML 구분
- innerText = 글자만
- innerHTML = 태그까지 포함

라) 이벤트 핸들링은 ‘조건’이 아니라 ‘반응’
- if문은 “조건에 따라 실행”
- 이벤트는 “사용자가 행동할 때 실행”

마) 하나씩 테스트
- DOM 조작이나 이벤트는 오타 하나로 안 돌아감
- console.log()로 단계마다 확인하는 습관이 중요

7. 실습 문제

■ 문제 1.
- 버튼 클릭 시 배경색이 바뀌는 박스 만들기
- 입력창에 이름을 입력하면 안녕하세요, ___님! 출력하기

- 문제 2.

- +1, -1 버튼으로 카운터 만들기
- 체크박스 클릭 시 텍스트에 취소선 그어지게 하기

8. 정리

- DOM = HTML을 코드로 다룰 수 있게 한 구조
- 요소 선택 : querySelector / querySelectorAll
- 요소 조작 : .innerText, .innerHTML, .style, .classList
- 이벤트 : 사용자의 행동, addEventListener로 연결
- 실습을 통해 “내가 만든 웹이 살아 움직이는 경험”을 하는 것이 중요

- Day 7 -

1. localStorage

■ localStorage 개념

브라우저에 **데이터를 저장**할 수 있는 공간
용량 : 약 5MB
특징 : 브라우저를 껐다 켜도 데이터가 남아 있음 (세션과 다름)
간단히 말해, localStorage는 **브라우저 안의 작은 메모장**

■ **사용처**

UI 환경설정 저장 : 다크모드, 언어, 글자 크기 등
최근 본 항목/검색어 기록
간단한 임시 저장 : 폼 작성 중인 내용 임시 보관(초안/드래프트)
가벼운 캐시 : 작은 API 응답을 잠깐 저장(시간 제한 두기)

```
// 저장
localStorage.setItem("key", "value");

// 불러오기
let value = localStorage.getItem("key");
console.log(value);

// 삭제
localStorage.removeItem("key");

// 전체 삭제
localStorage.clear();
```

■ JSON과 함께 쓰기

■ localStorage는 문자열만 저장 가능 → 객체/배열은 JSON 변환 필요

```
let user = { name: "홍길동", age: 25 };

// 저장
localStorage.setItem("user", JSON.stringify(user));

// 불러오기
let savedUser = JSON.parse(localStorage.getItem("user"));
console.log(savedUser.name); // "홍길동"
```

✔ Tip

1. 동기(synchronous) API라서, 큰 데이터를 잦게 저장하면 UI가 잠깐 멈출 수 있어요
   - → 데이터가 크거나 자주 쓰면 IndexedDB(비동기) 고려
2. 보안
   - XSS가 발생하면 localStorage 내용이 유출될 수 있어요. 민감 정보(토큰/개인정보) 저장 금지
   - 인증 토큰은 가능한 HTTP-Only 쿠키나 메모리 사용 등 대안 검토
3. 문자열만 저장 → JSON.stringify/parse는 try/catch로 감싸 오류 대비(사용자가 개발자 도구로 값을 바꾸거나, 손상될 수 있음)
4. 네이밍/버저닝
   - 키 앞에 접두어 붙이기 : "myapp:v1:theme", "myapp:v1:user"
   - 스키마가 바뀌면 버전 올리고 마이그레이션 로직 추가
5. TTL(만료시간) 패턴
   - 저장할 때 시간도 같이 넣고, 꺼낼 때 만료 확인

```
function setWithTTL(key, value, ttlMs) {
  const record = { value, expiresAt: Date.now() + ttlMs };
  localStorage.setItem(key, JSON.stringify(record));
}
function getWithTTL(key) {
  try {
    const raw = localStorage.getItem(key);
    if (!raw) return null;
    const { value, expiresAt } = JSON.parse(raw);
    if (Date.now() > expiresAt) { localStorage.removeItem(key); return null; }
    return value;
  } catch { return null; }
}
```

2. fetch API

fetch 개념

서버(또는 외부 API)에 요청(request)을 보내고 응답(response)을 받는 함수

HTTP 요청을 쉽게 할 수 있음

□ 쉽게 말해, fetch는 **웹에서 데이터 가져오기 배달 서비스**

공공/외부 API에서 데이터 가져오기(날씨/영화/뉴스 등)

백엔드 서버와 통신(상품목록, 로그인, 글쓰기 등)

2.3 fetch 동작 흐름

fetch(URL) → 요청 보내기

서버가 응답 → response 객체
response.json()으로 JSON 변환
변환된 데이터를 화면에 표시

2.4 실습
공공 API 불러오기

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(res =& res.json())
  .then(data =& {
    console.log("제목:", data.title);
  });
```

---

3. 미니 프로젝트: 메모장 저장 & API 데이터 출력

목표
입력창에 메모 작성 → localStorage에 저장
새로고침해도 메모 유지
버튼 클릭 시 API에서 데이터 불러와 화면에 표시

코드 예시

```
<input type="text" id="memoInput" placeholder="메모를 입력하세요">
<button id="saveBtn">저장</button>
<p id="memoDisplay"></p>

<button id="apiBtn">API 불러오기</button>
<p id="apiResult">< /p>

<script>
  const input = document.querySelector("#memoInput");
  const saveBtn = document.querySelector("#saveBtn");
  const display = document.querySelector("#memoDisplay");

  // 저장된 메모 불러오기
  display.innerText = localStorage.getItem("memo") || "";

  // 메모 저장
```

```
  saveBtn.addEventListener("click", () =& {
    localStorage.setItem("memo", input.value);
    display.innerText = input.value;
  });

  // API 데이터 불러오기
  document.querySelector("#apiBtn").addEventListener("click", () =& {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
      .then(res =& res.json())
      .then(data =& {
        document.querySelector("#apiResult").innerText = data.title;
      });
  });
</script>
```

✔ Tip

localStorage는 **데이터베이스(DB)의 아주 간단한 버전**이라고 생각하면 이해 쉬움
fetch는 “외부에서 JSON을 가져와 DOM에 뿌린다”로 요약
console.log()로 데이터가 제대로 들어왔는지 항상 확인해보기

---

4. 오늘 배운 핵심

localStorage = 브라우저 저장소 (데이터 유지 가능)
fetch API = 외부 데이터 불러오기
☐ 내 웹사이트가 **기억하고**, **외부와 소통하는** 첫걸음!

---

5. 도전 과제

오늘 할 일(ToDo)을 localStorage에 저장해서, 새로고침해도 유지되도록 만들기
fetch API로 **영화 검색 API**(OMDb, TMDB 등) 불러와서 영화 제목 목록 출력하기

- Dya 8 -

1. JSON (JavaScript Object Notation)

JSON이란?

데이터를 주고받기 위한 표준 형식
자바스크립트의 객체 문법을 차용 → 다른 언어에서도 사용 가능
웹 API, 서버/클라이언트 간 데이터 교환에 **가장 많이 쓰임**
□ **쉽게 말해 :** JSON은 **데이터를 담은 택배 상자**, 누구나 열 수 있고, 어디로든 보낼 수 있음

JSON 형식

Key-Value 쌍 (키는 반드시 문자열 " "로 감싸야 함)
숫자, 문자열, 불리언, 배열, 객체 포함 가능

```
{
  "name": "홍길동",
  "age": 25,
  "isStudent": false,
  "hobbies": ["독서", "게임"],
  "address": { "city": "서울", "zip": "12345" }
}
```

JSON과 자바스크립트 객체 차이

```
// JS 객체
let obj = { name: "홍길동", age: 25 };

// JSON (문자열)
let json = '{"name":"홍길동","age":25}';
```

변환 방법

```
// 객체 → JSON 문자열
let jsonStr = JSON.stringify(obj);

// JSON 문자열 → 객체
let parsedObj = JSON.parse(jsonStr);
```

JSON 실습

user 객체 만들고 localStorage에 JSON으로 저장
저장된 JSON 문자열을 꺼내서 다시 객체로 변환
이름과 나이를 출력해보기

---

2. 비동기 처리 (async/await)

동기 vs 비동기

동기(Synchronous) : 작업이 순서대로, 하나 끝나야 다음 실행
비동기(Asynchronous) : 기다리는 동안 다른 작업도 진행 가능
□ 예시
동기 = 편의점에서 계산대 1개, 손님이 계산 끝날 때까지 다음 사람 기다림
비동기 = 무인 계산대 여러 개, 계산 중에도 다른 사람 처리 가능

---

fetch + 비동기 문제

```
let data = fetch("https://jsonplaceholder.typicode.com/todos/1");
console.log(data); // Promise { &pending& }
```

□ fetch 결과는 바로 값이 아니라 **Promise**(약속).
즉, “나중에 결과 줄게”라는 종이 쪽지 같은 것.

---

async/await 기본 문법

```
async function getTodo() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    if (!response.ok) throw new Error("HTTP 오류: " + response.status);
    let data = await response.json();
    console.log(data);
  } catch (err) {
    console.error("에러 발생:", err);
  }
}
getTodo();
```

async : 함수가 비동기 함수를 쓴다는 표시
await : Promise가 끝날 때까지 기다림
try/catch : 에러를 안전하게 처리

---

async/await 실습

버튼 클릭 시 API 요청 → JSON 데이터 화면에 표시
로딩 중일 때 “로딩 중…” 표시
에러 발생 시 “데이터 불러오기 실패” 표시

✔ Tip

JSON

JSON은 반드시 " " 큰따옴표 사용 (작은따옴표 불가)
undefined, function은 JSON에 저장할 수 없음
JSON.parse 시 에러 발생 가능 → try/catch로 감싸기

async/await

await는 반드시 **async 함수 안에서만 사용 가능**
fetch는 404/500 에러도 Promise 성공으로 취급 → res.ok 확인 필수
네트워크 불안정 고려해서 항상 에러 처리(catch or try/catch)

---

3. 미니 프로젝트: 영화 검색 앱 (간단 버전)

요구사항

입력창에 영화 제목 입력
버튼 누르면 API 요청
JSON 데이터에서 영화 목록 가져와 화면에 출력

```
<input id="query" placeholder="영화 제목 입력">
<button id="searchBtn">검색</button>
<ul id="results"></ul>
```

```
async function searchMovie() {
  const query = document.querySelector("#query").value;
  const results = document.querySelector("#results");
  results.innerHTML = "로딩 중...";
```

```
    try {
      const res = await fetch(`https://www.omdbapi.com/?apikey=demo&s=${query}`);
      if (!res.ok) throw new Error("HTTP 에러 " + res.status);

      const data = await res.json();
      results.innerHTML = "";

      if (data.Search) {
        data.Search.forEach(movie =& {
          const li = document.createElement("li");
          li.textContent = movie.Title + " (" + movie.Year + ")";
          results.appendChild(li);
        });
      } else {
        results.innerHTML = "검색 결과 없음";
      }
    } catch (err) {
      results.innerHTML = "데이터 불러오기 실패 □";
      console.error(err);
    }
  }

  document.querySelector("#searchBtn").addEventListener("click", searchMovie);
```

---

6. 오늘 핵심

JSON = 데이터 교환 표준, stringify/parse로 변환
async/await = 비동기 코드를 동기처럼 읽기 쉽게 작성
실무 포인트 = 항상 에러 처리 + 로딩 상태 처리

WORK BOOK

✒️ JSON 이해하기

JSON은 **데이터를 주고받기 위한 _____ 형식**입니다.
자바스크립트의 객체 문법과 비슷하지만, 모든 키는 반드시 _____ 로 감싸야 합니다.
JSON은 서버 ↔ 브라우저 간 _____ **교환**에 가장 많이 사용됩니다.

✒️ JSON 예시

아래는 사용자 정보를 담은 JSON입니다.

```
{
  "name": "홍길동",
  "age": 25,
  "isStudent": false,
  "hobbies": ["독서", "게임"],
  "address": { "city": "서울", "zip": "12345" }
}
```

☑ JSON에서 name 키의 값은? → _____

☑ hobbies 배열의 두 번째 값은? → _____

✒️ JSON과 JS 객체의 차이

```
// 자바스크립트 객체
let obj = { name: "홍길동", age: 25 };

// JSON 문자열
let json = '{"name":"홍길동","age":25}';
```

➡ JSON은 단순히 **텍스트**(문자열)이에요.

➡ 자바스크립트에서 사용하려면 **객체로 변환(parse)** 해야 합니다.

✒️ 실습 JSON 변환

```
let user = { name: "철수", age: 30 };

// (1) 객체 → JSON 문자열로 바꾸기
let jsonData = JSON._____(user);

// (2) JSON 문자열 → 객체로 다시 바꾸기
let objData = JSON._____(jsonData);

console.log(objData.name);
```

✒️ async / await 이해하기

☐ 개념 요약

동기(synchronous) : 한 작업이 끝나야 다음 작업 실행
비동기(asynchronous) : 기다리는 동안 다른 작업도 진행 가능
async/await는 비동기 코드를 **마치 순서대로 실행되는 것처럼** 쉽게 읽을 수 있게 해줍니다.

비유

동기 : 편의점에 계산대 1개 → 손님이 계산 끝날 때까지 다음 손님 대기
비동기 : 셀프 계산대 여러 개 → 동시에 계산 가능

기본 문법

```
async function getData() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("에러 발생:", error);
  }
}
```

☐ async 함수 안에서는 _____ 키워드를 써서 Promise가 끝날 때까지 기다립니다.
☐ 오류가 나면 try / _____ 블록으로 예외를 처리합니다.

주의사항

항목	주의 포인트
await	반드시 async 함수 안에서만 사용 가능
fetch	404/500도 에러 아님 → res.ok로 상태 체크 필요
JSON	undefined, function은 JSON에 저장 불가
보안	fetch 시 API 키 노출 주의 (.env 파일이라도 노출 가능)

✒️ 실습 영화 검색 앱 만들기

입력창에 영화 제목을 입력하면
버튼 클릭 → API 호출 (OMDb 무료 API 사용)
JSON 데이터로 영화 제목과 연도 출력

준비 코드

```
<input id="query" placeholder="영화 제목 입력">
<button id="searchBtn">검색</button>
<ul id="results"></ul>
```

아래 코드를 완성해보세요.

```
async function searchMovie() {
  const query = document.querySelector("#query").value;
  const results = document.querySelector("#results");
  results.innerHTML = "__________"; // 로딩 메시지

  try {
    const res = await fetch(`https://www.omdbapi.com/?apikey=demo&s=${query}`);
    if (!res.ok) throw new Error("HTTP 에러");

    const data = await res._________(); // JSON 변환
    results.innerHTML = "";

    if (data.Search) {
      data.Search.forEach(movie =& {
        const li = document.createElement("li");
        li.textContent = `${movie.Title} (${movie.Year})`;
        results.appendChild(li);
      });
    } else {
      results.innerHTML = "검색 결과 없음";
    }
  } catch (err) {
    results.innerHTML = "__________ □"; // 에러 메시지
  }
}

document.querySelector("#searchBtn")
  .addEventListener("click", searchMovie);
```

- Day 9 -

React란 무엇인가?

개념 요약

React는 **사용자 인터페이스(UI)** 를 만들기 위한 **자바스크립트 라이브러리**입니다.
HTML + JS + CSS 로 구성된 웹을 "컴포넌트"라는 조각 단위로 관리하게 해줍니다.
페이스북(Meta) 이 만들었고, 현재 전 세계에서 가장 많이 사용되는 프론트엔드 기술입니다.

React의 특징

특징	설명
컴포넌트 기반	UI를 재사용 가능한 컴포넌트 단위로 구성
단방향 데이터 흐름	데이터가 부모 → 자식 방향으로 흐름
가상 DOM(Virtual DOM)	변경된 부분만 효율적으로 업데이트
선언형 프로그래밍	"어떻게"보다 "무엇을 보여줄지"를 코드로 표현

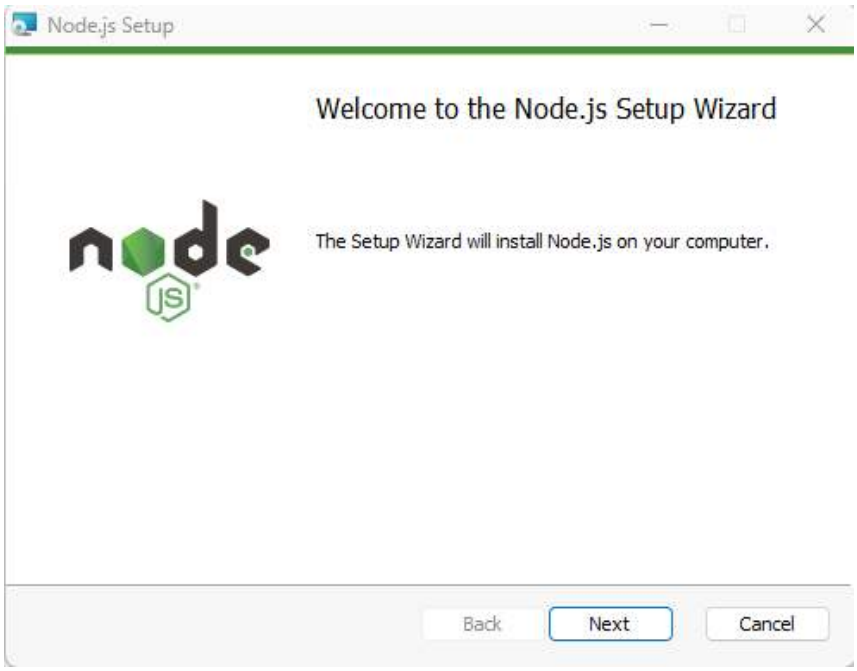
예시로 이해하기

"HTML은 브러시로 직접 그림을 그리는 것,
React는 도형 조각(컴포넌트)을 미리 만들어 조합하는 것."

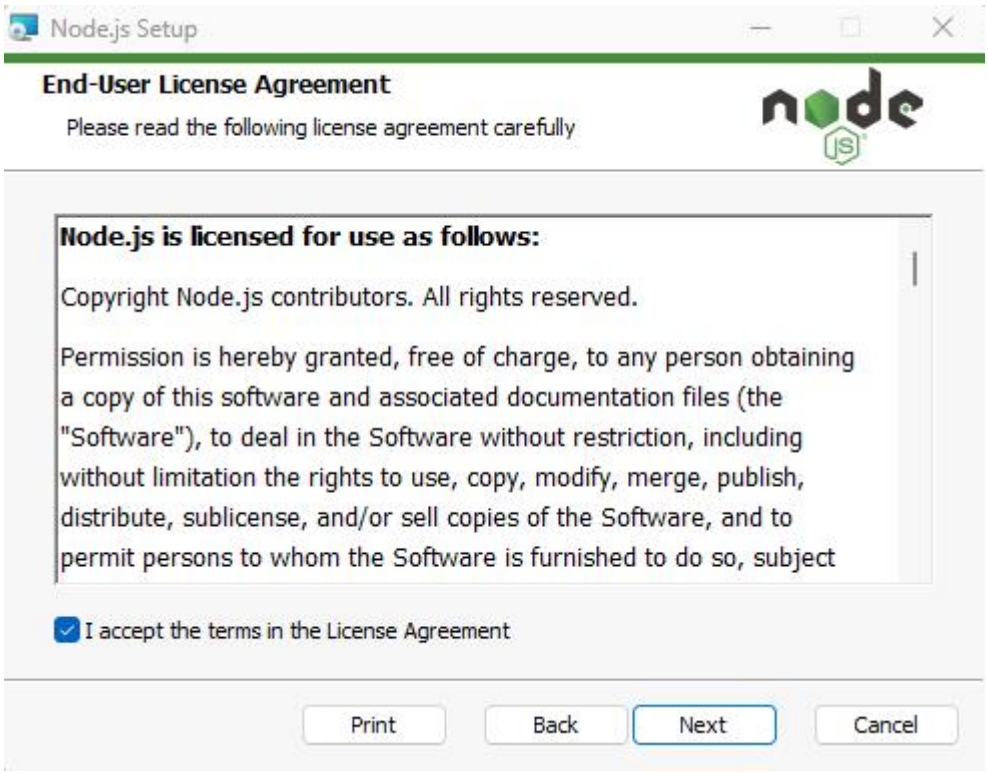
개발 환경 세팅 (Vite + Node.js + VS Code)

Node.js 설치

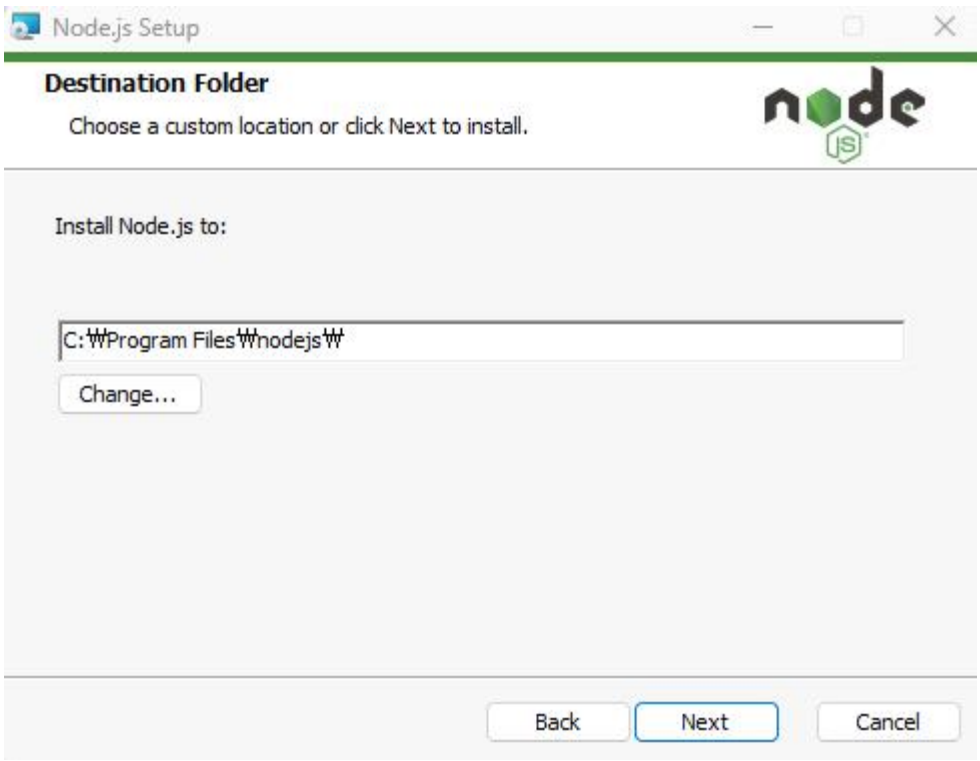
https://nodejs.org → LTS 버전 다운로드



next 클릭



체크박스 체크 후 next 클릭



next 클릭

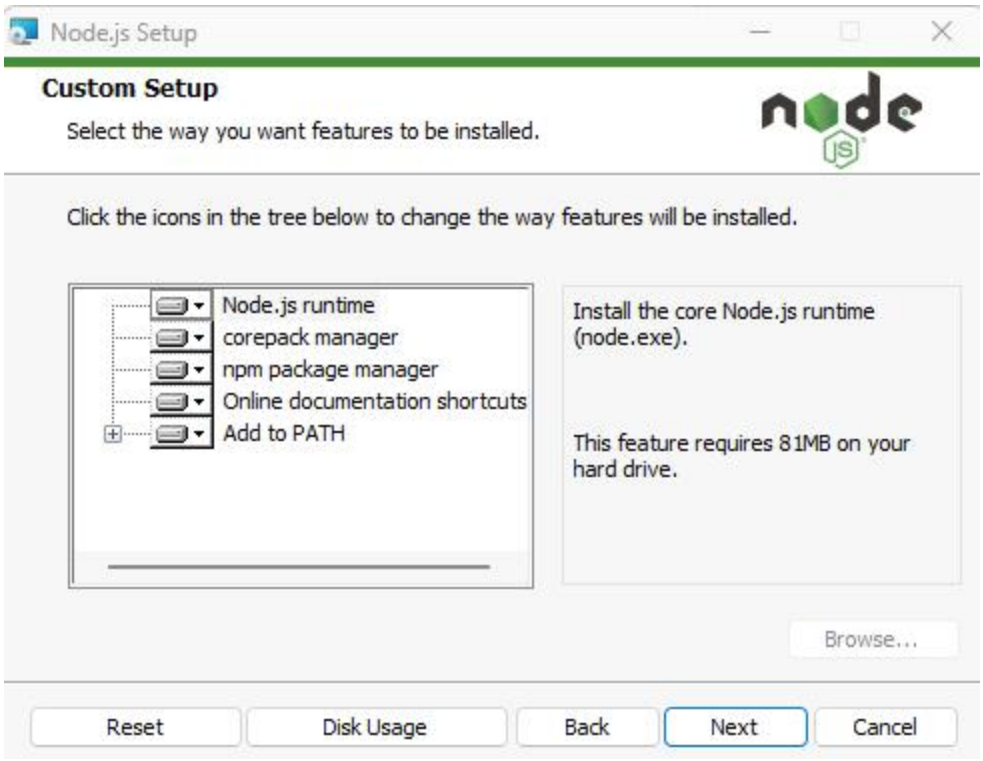


그림 34

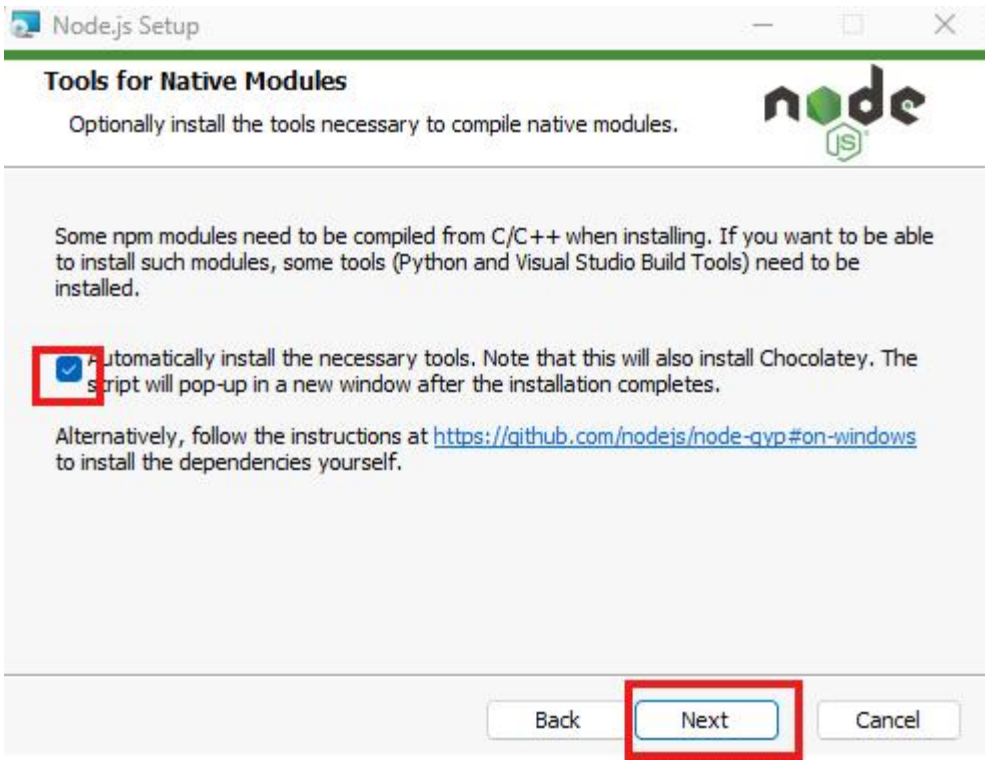


그림 35

Automatically 왼쪽 체크하고 Next 클릭

- Automatically install 체크 안 했을 경우 나중에 후회하는 경우가 많으므로 꼭 설치하는 것을 추천

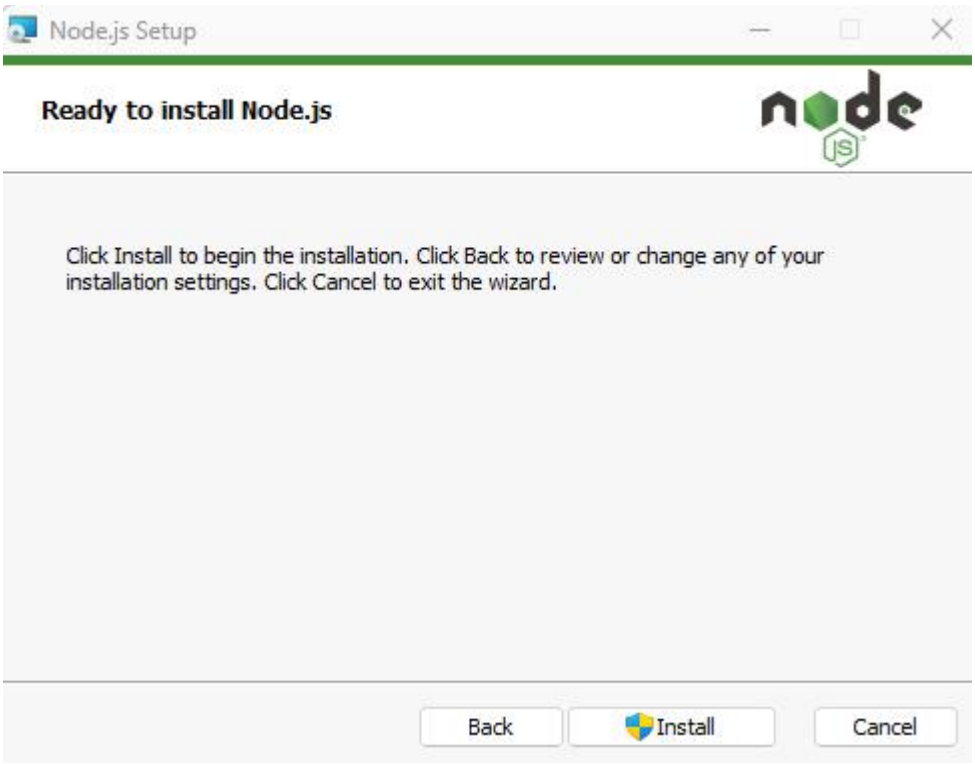


그림 36

설치 중에 중간에 팝업화면이 뜨면서 설치할 건지 물어보면 예(Yes) 누르고 넘어가기

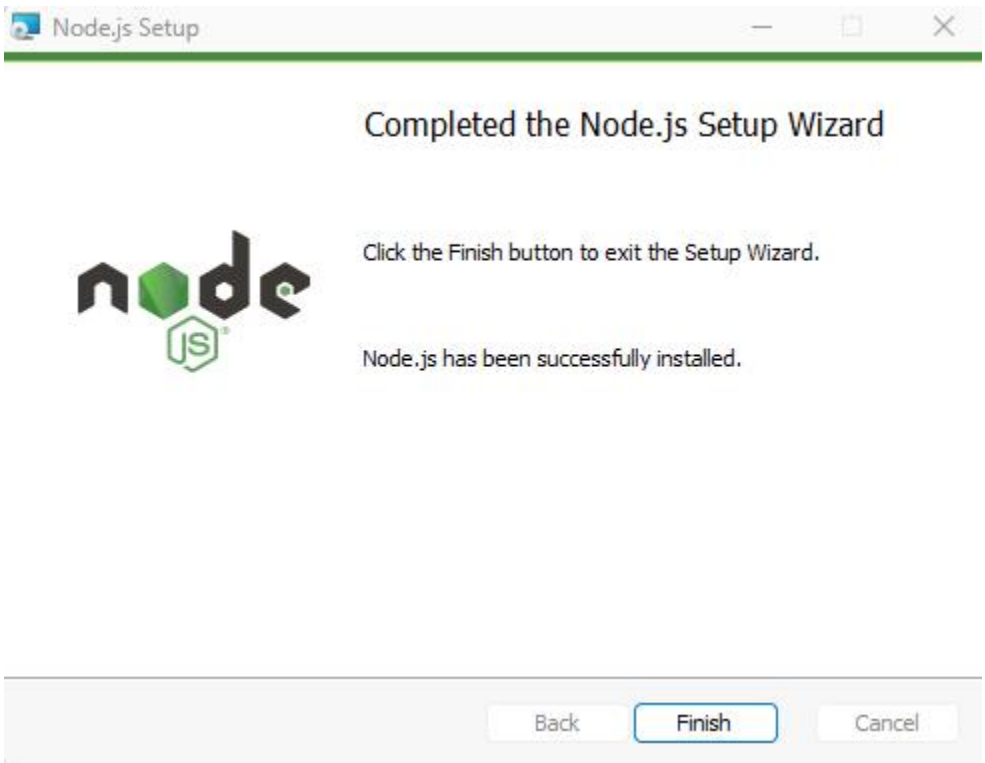


그림 37

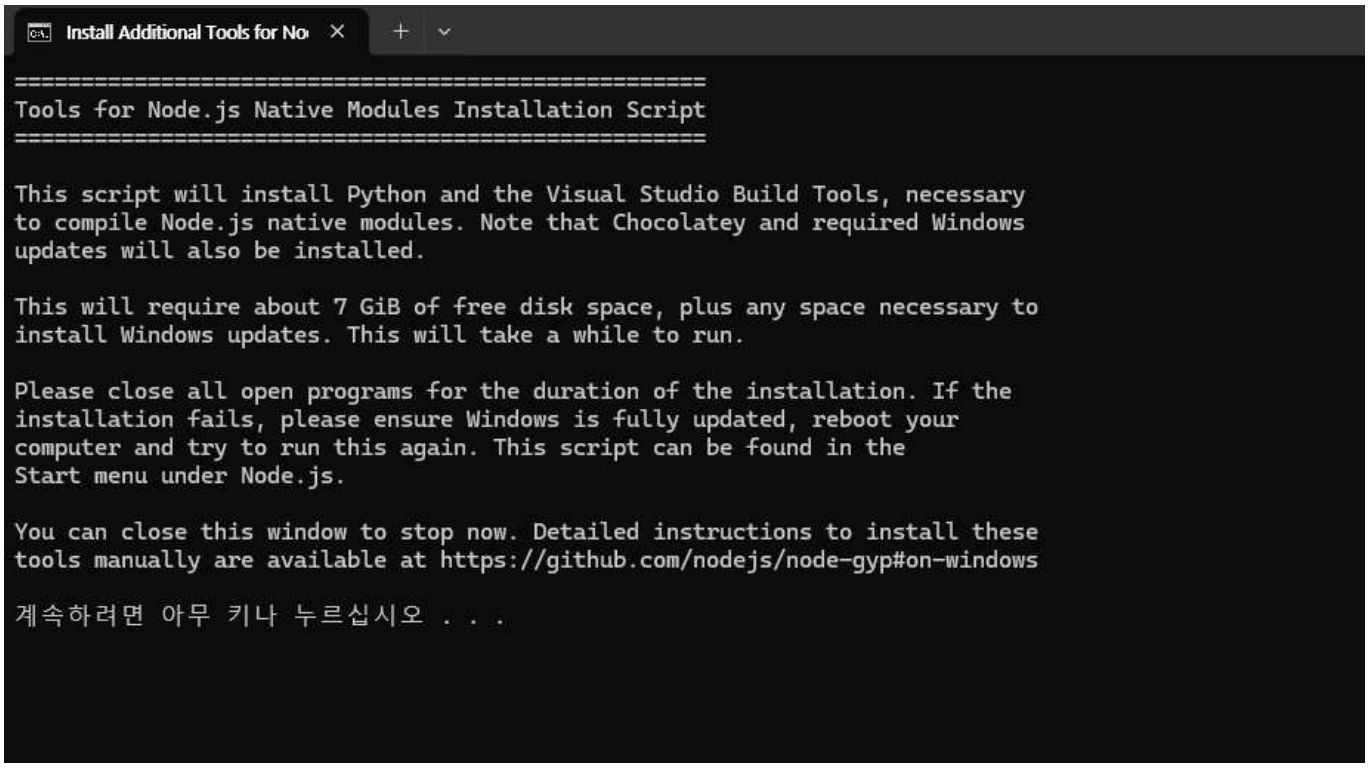


그림 38

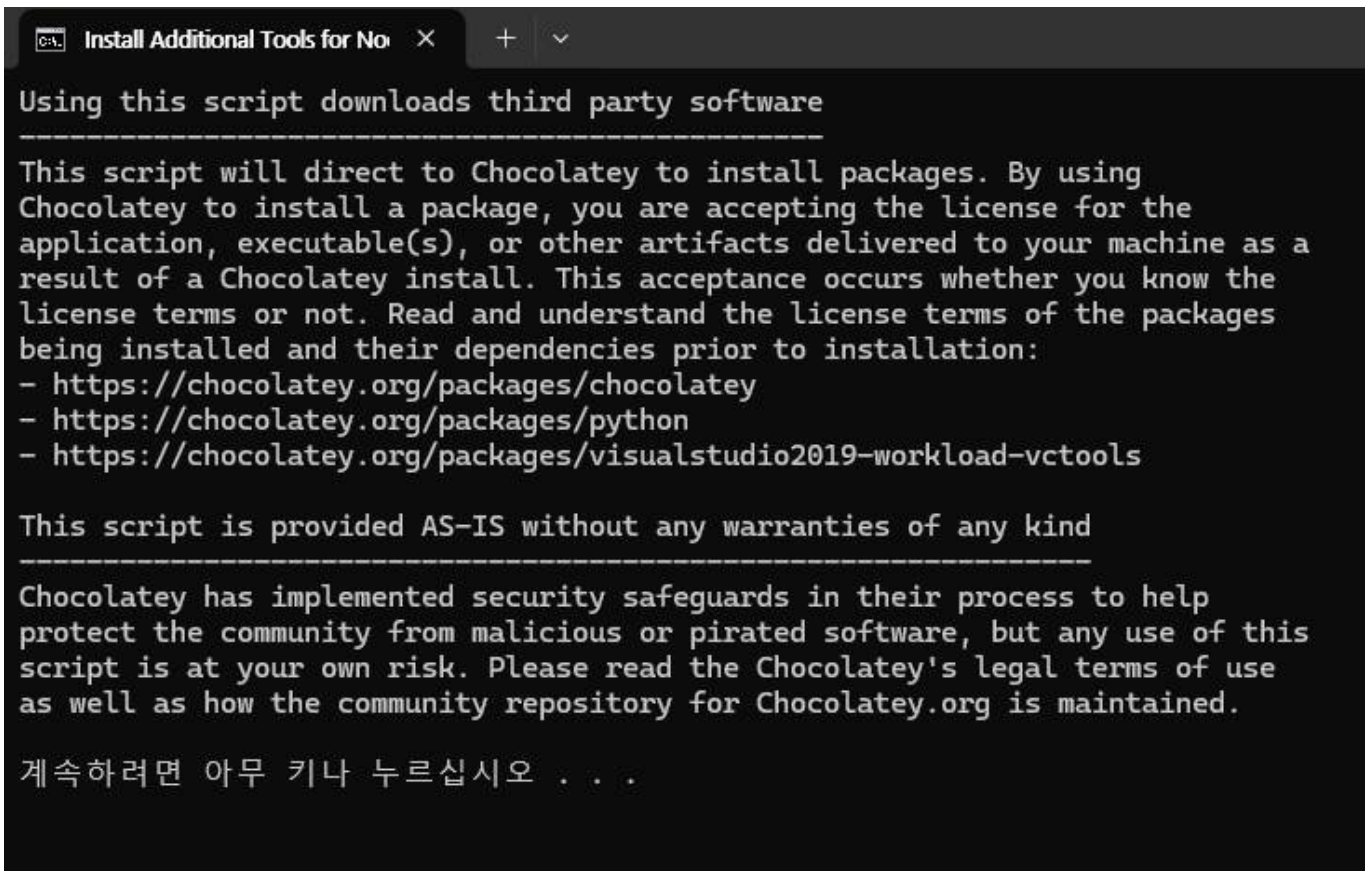


그림 39

Windows PowerShell 실행 여부 창 예 클릭

React 프로젝트 생성

npm create vite@latest my-react-app

Framework : React
Variant : JavaScript 선택
cd my-react-app
npm install
npm run dev

브라우저에서 http://localhost:5173 접속 → 첫 React 화면 확인

```
my-react-app/
  ├── public/              → 이미지 등 정적 파일
  ├── src/                 → 코드 작성 폴더
  |     ├── App.jsx          → 메인 컴포넌트
  |     ├── main.jsx         → React 시작점
  |     └── components/      → 컴포넌트 폴더
  ├── package.json
  └── vite.config.js
```

❖ JSX 문법

JSX란?

JavaScript 안에서 HTML처럼 UI를 표현할 수 있게 하는 **React 전용 문법**
사실상 React.createElement() 의 간결한 표현 방식

```
// HTML과 비슷하지만, 실제로는 JS 코드
const element = <h1>안녕하세요, React!</h1>;
```

JSX 문법 규칙

규칙	예시	설명
닫는 태그 필수	<img />	홀로 있는 태그도 반드시 닫기
여러 요소는 하나로 감싸기	...	루트 태그 하나만 존재해야 함
class → className		JS 예약어(class)와 충돌 방지
JS 표현식은 {}	{name}	변수나 연산 결과를 표현 가능
속성값 문자열은 따옴표		속성값은 항상 문자열로 작성

JSX 예시

```
const name = "홍길동";
const element = (
  <div>
    <h1>안녕하세요, {name}!</h1>
    <p>오늘도 즐거운 코딩 </p>
  </div>
);

export default element;
```

❖ 첫 컴포넌트 만들기

컴포넌트란?

화면의 일부를 구성하는 **독립적인 UI 조각**

함수형 컴포넌트가 주로 사용됨

```
function Welcome(props) {
  return <h1>안녕하세요, {props.name}님!</h1>;
}

export default Welcome;
```

사용 예시

```
import Welcome from "./components/Welcome";

function App() {
  return (
    <div>
      <Welcome name="홍길동" />
      <Welcome name="철수" />
    </div>
  );
}

export default App;
```

❖ 자주 하는 실수 & 주의 사항

항목	설명
대문자 시작	컴포넌트 이름은 반드시 **대문자 시작**
루트 태그 하나	JSX는 반드시 하나의 부모 태그로 감싸야 함
className	class 대신 className 사용
return 괄호	여러 줄 JSX 반환 시 괄호로 감싸기
JS 코드와 혼동	JSX 안에서는 {} 만 JS 표현 가능
경로 오타	import 경로 대소문자 정확히 확인

❖ 실습 : 나만의 명함 카드 만들기

목표

이름, 직업, 이메일을 JSX로 표현
CSS 클래스 적용 (className 사용)

```
function MyCard() {
  return (
    <div className="card">
      <h2>전계림</h2>
      <p>프론트엔드 개발자</p>
      <p>Email : example@naver.com</p>
    </div>
  );
}

export default MyCard;
```

확장 : 사진, 소셜 링크, 배경색 변경 기능 추가

복습 퀴즈

번호	문제	답
1	React는 무엇을 만들기 위한 라이브러리인가요?	
2	JSX는 ______ 안에서 ______ 를 표현할 수 있는 문법입니다.	
3	JSX에서 class 대신 사용하는 속성은?	
4	여러 요소를 JSX로 반환할 때 반드시 하나의 ______ 로 감싸야 합니다.	
5	컴포넌트 이름은 반드시 ______ 로 시작해야 합니다.	

오늘의 도전 과제

ProfileCard 컴포넌트 만들기
props : name, job, email
카드 스타일 꾸미기 (CSS className 사용)
App.jsx 에 여러 명의 프로필 카드 출력하기
map() 을 이용해 반복 렌더링 시도하기

오늘의 핵심 정리

React는 **컴포넌트 기반 UI 라이브러리**이다.
JSX는 **HTML + JS** 의 결합 문법으로, **className / {} / 루트 태그** 규칙이 중요하다.
모든 React 앱은 **작은 컴포넌트의 조합**으로 이루어진다.

REACT 교육 일정 재구성

단계	일자	오전	오후	학습 목표
React 기초 다지기	1일	JSX 복습, 컴포넌트 구조 이해 import / export / return 개념	props, state 실습 (카운터, 입력값 제어)	컴포넌트 구조와 상태 변화 이해
	2일	useState 응용 : 입력값·리스트 관리	부모→자식 데이터 전달(props) / 이벤트 핸들링(onClick, onChange)	React 데이터 흐름(상향·하향) 익히기
	3일	useEffect 기초 : 렌더링 타이밍과 생명주기	fetch() 로 JSON 불러오기 실습 (공공데이터 예시)	외부 데이터 가져오기 이해
	4일	map(), 조건부 렌더링, key 개념	미니 프로젝트 : “날씨/환율 리스트 출력 앱”	데이터 출력과 렌더링 완성
Chart.js 시각화 집중	5일	Chart.js & react-chartjs-2 설치 / 구조 설명	하드코딩 데이터로 막대, 선 그래프 실습	그래프 생성 구조 이해
	6일	props 기반 Chart 데이터 변경 실습	fetch() 데이터 → Chart.js 시각화	API 데이터 그래프화
	7일	Line + Bar + Pie 등 복합 차트 구성	미니 프로젝트 : “나의 데이터 대시보드” 제작	데이터 시각화 완성
Firebase 연동	8일	Firebase 프로젝트 생성 / 연결	Firestore CRUD (데이터 추가·삭제·조회)	React + DB 연동
	9일	Firebase Auth (로그인 / 회원가입)	로그인 사용자 전용 데이터 관리	인증 + 개인화 데이터 연동
	10일	Firebase Hosting 배포 / 환경변수 설정	종합 실습 : “로그인 + 차트 + 저장 앱”	배포 가능한 서비스 완성
프로젝트 예열 및 정리	11일	핵심 기술 총복습(JSX → useState → fetch → Chart.js → Firebase 흐름 정리)	팀별 미니 기획 & API 테스트 → Firebase Hosting으로 배포	4주 프로젝트 준비 완료

Day 1 – JSX, 컴포넌트, props/state 기초

오늘의 목표

✔ 리액트로 화면을 만드는 기본 구조를 이해하고
✔ "컴포넌트"로 화면을 나누는 방법을 배우며
✔ "props"와 "state"로 데이터를 다루는 기초를 익히기

1. React란?

🧱 "HTML보다 똑똑한 화면 도구"

보통 HTML로 만든 웹페이지는 고정되어 있음, 버튼을 눌러도 화면이 바뀌려면 새로고침이 필요
리액트(React)는 **데이터가 바뀌면 자동으로 화면이 바뀌는 웹페이지**를 만들 수 있게 해주는 도구
💬 **비유**
HTML은 '사진', React는 '동영상' 즉, **살아 있는 화면**을 만드는 것

2. JSX문법이란?

React에서는 HTML처럼 보이는 코드를 사용하지만,
사실은 **JavaScript 안에서 화면을 만드는 문법**
이게 바로 **JSX (JavaScript + XML)**

💡 JSX 기본 예시

```
// App.jsx
export default function App() {
  return (
    <div>
      <h1>안녕하세요! DW Academy입니다 </h1>
      <p>이건 JSX 문법으로 만든 화면이에요.</p>
    </div>
  );
}
```

이 코드는 **HTML처럼 보이지만 실제로는 JavaScript 함수**
return 안에 들어있는 게 화면에 보여질 부분

JSX의 중요한 규칙

규칙	설명	예시
1	반드시 **한 개의 부모 태그**로 감싸야 함	❌ → ☑
2	태그는 **닫혀야 함**	<img />, <input />
3	자바스크립트 값은 { } 로 표시	{name}
4	class 대신 **className** 사용	

실습 : Hello React

파일 : src/App.jsx

```
export default function App() {
  const name = "홍길동";
  return (
    <div>
      <h1>안녕하세요, {name}님 </h1>
      <p>React 세계에 오신 것을 환영합니다!</p>
    </div>
  );
}
```

예상화면

안녕하세요, 홍길동님
React 세계에 오신 것을 환영합니다!

3. 컴포넌트 (Component)

화면을 나누는 작은 블록

React에서는 화면 전체를 한 번에 만들지 않고, **Header**, **Main**, **Footer** 처럼 조각 단위(컴포넌트)로 나눈다.

비유

웹사이트는 레고 블록처럼 컴포넌트를 조합해서 만든 집 이다.

실습 : 컴포넌트 만들기

파일 구조

src/
├ App.jsx
├ Header.jsx
├ Footer.jsx

Header.jsx

```
export default function Header() {
  return <h1>DW Academy React 수업</h1>;
}
```

Footer.jsx

```
export default function Footer() {
  return <p>2025 DW Academy All Rights Reserved.</p>;
}
```

App.jsx

```
import Header from "./Header";
import Footer from "./Footer";

export default function App() {
  return (
    <div>
      <Header />
      <p>이건 본문 부분입니다.</p>
      <Footer />
    </div>
  );
}
```

예상화면

DW Academy React 수업
이건 본문 부분입니다.
2025 DW Academy All Rights Reserved.

4. props (프로퍼티)

“컴포넌트에 값을 전달하는 방법”

부모 컴포넌트(App)에서 자식 컴포넌트로 데이터를 넘길 때 props를 쓴다.

실습 : props 사용해보기

파일 : Welcome.jsx

```
export default function Welcome(props) {
  return <h2>안녕하세요, {props.name}님 </h2>;
}
```

파일 : App.jsx

```
import Welcome from "./Welcome";

export default function App() {
  return (
    <div>
      <Welcome name="이순신" />
      <Welcome name="문익점" />
    </div>
  );
}
```

예상화면

안녕하세요, 이순신님
안녕하세요, 문익점님

props는 **부모 → 자식으로 데이터 전달**하는 통로이다.
HTML의 속성(attribute)처럼 생각하면 된다.
예: <Welcome name="이순신" />

5. state (상태)

“화면에서 바뀌는 데이터”

리액트의 가장 큰 특징은 **데이터가 바뀌면 화면도 자동으로 바뀐다는 것!**
그 데이터를 state라고 부른다.

실습 : 숫자 증감 버튼 만들기

파일 : Counter.jsx

```
import { useState } from "react";

export default function Counter() {
  const [count, setCount] = useState(0); // 기본값 0

  return (
    <div>
      <h3>현재 숫자 : {count}</h3>
      <button onClick={() => setCount(count + 1)}>+ 증가</button>
      <button onClick={() => setCount(count - 1)}>- 감소</button>
    </div>
  );
}
```

App.jsx

```
import Counter from "./Counter";

export default function App() {
  return (
    <div>
      <h1>카운터 예제</h1>
      <Counter />
    </div>
  );
}
```

예상화면

카운터 예제
현재 숫자 : 0
[+ 증가] [- 감소]

버튼을 누르면 숫자가 자동으로 바뀐다.
즉, state가 바뀌면 화면이 새로 고쳐지지 않아도 React가 알아서 업데이트한다.

한눈에 정리

개념	설명	예시

JSX	HTML처럼 화면을 만드는 문법	{name}
컴포넌트	화면을 나누는 블록	<Header />, <Footer />
props	부모 → 자식으로 값 전달	<Welcome name="홍길동" />
state	변할 수 있는 값	useState(0)

실습 문제

문제 1 : 자기소개 카드 만들기

새 Vite 프로젝트를 만들고 다음을 구현해보기

요구사항

1. App.jsx → ProfileCard 컴포넌트 import
2. ProfileCard에 name, age, hobby props 전달
3. 화면에

> 안녕하세요, 저는 홍길동입니다.
> 나이는 25살이고, 취미는 음악 감상입니다.

형태로 표시

파일 구조

```
src/
 ├ App.jsx
 └ ProfileCard.jsx
```

문제 2 : 좋아요 버튼 만들기

요구사항

1. LikeButton 컴포넌트 생성
2. useState를 사용해서 클릭할 때마다 숫자가 1씩 증가하도록 구현
3. 초기값은 0, 버튼 이름은 "좋아요♥"

결과 예시

좋아요 ♥ 3

(버튼을 클릭할 때마다 숫자가 올라감)

핵심 정리

React는 "데이터(state)"와 "화면(JSX)"이 자동으로 연결된다.
데이터를 바꾸면 화면이 새로 고침 없이 바뀐다.
이것이 HTML과 가장 큰 차이점이다

Day 2 – useState 심화, 이벤트 핸들링, props 응용

오늘의 목표

✔ useState로 “바뀌는 데이터”를 다룰 수 있고
✔ 사용자의 행동(버튼 클릭, 입력 등)에 반응하는 방법을 이해하며
✔ props로 부모 → 자식 간 데이터 전달을 복습한다.

1. 이벤트(Event)란?

이벤트 = “사용자의 행동”

버튼 클릭 → onClick
키보드 입력 → onChange
마우스 이동 → onMouseMove
리액트에서는 HTML과 다르게 **이벤트 이름이 카멜표기법**(소문자+대문자)으로 작성된다.
예 : onclick ❌ → onClick ☑

2. 버튼 클릭 이벤트 예제

```
// App.jsx
import { useState } from "react";

export default function App() {
  const [count, setCount] = useState(0);

  function increase() {
    setCount(count + 1);
  }

  return (
    <div>
      <h2>현재 숫자 : {count}</h2>
      <button onClick={increase}>+ 1 증가</button>
    </div>
  );
}
```

코드 설명

부분	설명
useState(0)	count라는 state를 0으로 시작
setCount(count + 1)	버튼 누를 때마다 count를 1 증가
onClick={increase}	클릭 시 increase 함수 실행

예상화면

현재 숫자 : 0
[+ 1 증가]

버튼을 클릭할 때마다 숫자가 하나씩 올라간다.

3. 화살표 함수로 더 간단히 쓰기

리액트에서는 이벤트 함수를 짧게 작성할 수 있다.

```
<button onClick={() => setCount(count + 1)}>+ 1 증가</button>
```

이렇게 쓰면 별도의 함수를 만들지 않아도 된다.
(하지만 코드가 길어지면 가독성을 위해 따로 함수로 분리하는 게 좋다.)

4. 입력값 다루기 (onChange)

이번엔 사용자가 입력한 값을 화면에 바로 보여주는 예제

```
import { useState } from "react";

export default function App() {
  const [name, setName] = useState("");

  return (
    <div>
      <h2>이름을 입력하세요</h2>
      <input
        type="text"
        placeholder="이름 입력"
        onChange={(e) => setName(e.target.value)}
      />
      <p>안녕하세요, {name}님!</p>
    </div>
  );
}
```

코드 설명

부분	설명
onChange={(e) =& setName(e.target.value)}	입력한 값이 바뀔 때마다 state에 저장
{name}	state의 내용을 바로 화면에 표시

예상화면

이름을 입력하세요
[홍길동]
안녕하세요, 홍길동님!

💬 한 글자 입력할 때마다 state가 갱신되어 화면도 실시간으로 바뀐다.

5. props 복습 : 부모 → 자식 데이터 전달

부모 컴포넌트(App)에서 자식 컴포넌트로 값을 전달

```
// Greeting.jsx
export default function Greeting(props) {
  return <h2>안녕하세요, {props.user}님 □</h2>;
}

// App.jsx
import Greeting from "./Greeting";

export default function App() {
  return (
    <div>
      <Greeting user="홍길동" />
      <Greeting user="이순신" />
    </div>
  );
}
```

예상화면

안녕하세요, 홍길동님
안녕하세요, 이순신님

💬 props는 “컴포넌트 간의 대화창구”라고 생각하자
부모가 값을 주면, 자식이 그 값을 받아 화면에 출력한다.

6. 여러 개의 state 다루기

한 컴포넌트 안에서 여러 개의 데이터를 관리할 수 있다.

```
import { useState } from "react";

export default function App() {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  return (
    <div>
      <h2>회원 정보 입력</h2>
      <input
        type="text"
        placeholder="이름 입력"
        onChange={(e) => setName(e.target.value)}
      />
      <input
        type="number"
        placeholder="나이 입력"
        onChange={(e) => setAge(e.target.value)}
      />
      <p>이름 : {name}</p>
      <p>나이 : {age}</p>
    </div>
  );
}
```

💻 예상화면

회원 정보 입력
이름: [홍길동]
나이: [25]
이름 : 홍길동
나이 : 25

핵심 요약

개념	설명	예시
onClick	버튼 클릭 이벤트	<button onClick={함수}>
onChange	입력값 변경 이벤트	<input onChange={함수}>

useState	화면 데이터 저장	const [num, setNum] = useState(0)
props	부모 → 자식 데이터 전달	<Child name="홍길동" />

💬 **리액트의 원리**

"사용자의 행동(Event) → state 변경 → 화면 자동 업데이트"

실습 문제 (새 프로젝트로 해보기)

문제 1 : 이름 저장 버튼 만들기

요구사항

새 프로젝트 생성 후 App.jsx 파일 작성

input에 이름을 입력하고, 버튼을 누르면 이름이 화면에 표시되도록 만들기

입력창 아래에 "당신의 이름은 홍길동입니다" 형태로 표시

힌트

useState("")

onChange로 입력값 관리

onClick으로 state 업데이트

예상화면

이름을 입력하세요

[홍길동]

[이름 저장]

당신의 이름은 홍길동입니다

문제 2 : 좋아요 버튼 만들기 (개선형)

요구사항

useState로 숫자 상태 관리

버튼을 누를 때마다 좋아요 수 증가

10 이상이 되면 "많은 사랑을 받고 있습니다 ❤ 문구 표시

결과 예시

좋아요 수 : 9

[좋아요 ❤

좋아요 수 : 10

많은 사랑을 받고 있습니다 ❤

핵심 한 줄

"React는 사용자의 행동(Event)과 데이터(State)를 자동으로 연결한다."

즉, **버튼을 누르거나 입력하면 → 데이터가 바뀌고 → 화면도 즉시 바뀐다.**

Day 3 – useEffect와 fetch (데이터 불러오기 기초)

오늘의 목표

✔ 리액트 컴포넌트가 언제 실행되고 다시 그려지는지(렌더링 시점) 이해한다.
✔ useEffect 훅(Hook)을 이용해 "화면이 처음 나타날 때" 코드를 실행한다.
✔ fetch() 함수를 사용해 외부에서 데이터를 불러와 화면에 표시한다.

1. 컴포넌트는 언제 실행될까?

리액트에서 컴포넌트는 "그려질 때(render)"마다 다시 실행된다.
즉, **state가 바뀔 때마다 함수가 다시 실행되는 구조**

```
import { useState } from "react";

export default function App() {
  const [count, setCount] = useState(0);
  console.log("컴포넌트 실행됨!");

  return (
    <div>
      <h2>카운터 : {count}</h2>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

콘솔에는 버튼을 누를 때마다 "컴포넌트 실행됨!"이 반복 출력된다.
→ **리액트는 데이터(state)가 바뀔 때마다 화면 전체를 다시 그린다.**

2. useEffect란?

리액트에서 "특정 시점"에 코드를 실행하고 싶을 때 쓰는 게 **useEffect**
이건 "리액트의 생명주기(Lifecycle)"를 다루는 훅이다.

useEffect는 "컴포넌트가 처음 나타날 때", 또는 "특정 데이터가 바뀔 때" 실행된다.

기본 문법

```
useEffect(() => {
  // 실행할 코드
}, []);
```

() 안의 **화살표 함수** : 실행할 동작
[] 안의 **의존성 배열(dependency array)** : 언제 실행할지 결정

의존성 배열의 의미

형태	실행 시점	설명
[]	화면이 처음 나타날 때 1번 실행	“처음에만 실행”
[state변수]	지정한 변수가 바뀔 때마다 실행	“특정 데이터가 바뀔 때 실행”
	화면이 다시 그려질 때마다 실행	“렌더링 될 때마다 실행” (주의!)

3. useEffect 기초 예제

```
import { useEffect, useState } from "react";

export default function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("처음 화면이 나타났어요!");
  }, []);

  return (
    <div>
      <h2>카운터 : {count}</h2>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

콘솔에는 “처음 화면이 나타났어요!” 한 번만 출력된다.
→ []를 넣으면 **컴포넌트가 처음 보일 때만 실행된다.**

4. fetch()란?

“외부에서 데이터를 가져오는 함수”

리액트는 **백엔드(API 서버)** 또는 **공공데이터**에서 정보를 가져와서 표시할 수 있다.
이때 사용하는 명령이 바로 **fetch()** 이다.

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then((response) => response.json())
  .then((data) => console.log(data));
```

fetch는 데이터를 받아오면 .then() 안의 코드가 실행된다.
이게 바로 **비동기(asynchronous)** 처리이다.
(“잠시 기다렸다가 결과가 오면 실행하는 방식”)

5. useEffect + fetch 함께 쓰기

이제 useEffect를 이용해 **화면이 처음 표시될 때 데이터를 가져오자.**

```
import { useState, useEffect } from "react";

export default function App() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    // 1. 화면이 처음 나타나면 fetch 실행
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => response.json()) // 2. 응답을 JSON으로 변환
      .then((data) => setUsers(data)); // 3. state에 저장
  }, []);

  return (
    <div>
      <h2>사용자 목록</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>
            {user.name} ({user.email})
          </li>
        ))}
      </ul>
    </div>
  );
}
```

예상화면

사용자 목록
Leanne Graham (Sincere@april.biz)
Ervin Howell (Shanna@melissa.tv)
Clementine Bauch (Nathan@yesenia.net)
...

이 예시는 실제 무료 테스트용 API(jsonplaceholder)에서 사용자 데이터를 불러온다.
(즉, 진짜 외부 서버에서 데이터를 가져온 거!)

6. 주의할 점

상황	설명	해결법
무한 fetch 발생	useEffect 안에 []를 안 넣으면 화면이 다시 그려질 때마다 fetch 실행	항상 [] 넣기
CORS 오류	API 서버에서 React 접근을 허용하지 않은 경우	다른 API 사용 or 서버 프록시 설정
setState 오타	setUser → setUsers 실수 잦음	변수 이름 정확히 확인

7. useEffect + state 응용 예제

```
import { useState, useEffect } from "react";

export default function App() {
  const [time, setTime] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    const timer = setInterval(() => {
      setTime(new Date().toLocaleTimeString());
    }, 1000);
    return () => clearInterval(timer); // 컴포넌트 종료 시 타이머 제거
  }, []);

  return (
    <div>
      <h2>현재 시각 : {time}</h2>
    </div>
  );
}
```

setInterval로 1초마다 state가 갱신되므로 화면의 시간이 자동으로 변한다.
이런 걸 “실시간 렌더링”이라고 한다.

핵심 정리

개념	설명	예시
useEffect	컴포넌트가 “처음 나타날 때” 또는 “데이터가 바뀔 때” 실행	useEffect(() =& {...}, [])
fetch	외부 서버(API)에서 데이터를 불러오는 함수	fetch("url").then(res=&res.json())
비동기	데이터가 도착하기 전에도 코드가 계속 실행되는 구조	.then() 사용

React는 “데이터가 들어오면 화면이 다시 그려진다”는 규칙을 기억!

실습 문제

문제 1 : 날씨 데이터 불러오기 (Open API 사용)

요구사항
fetch()로 아래 URL 데이터를 가져오기
https://api.zippopotam.us/us/90210

받아온 JSON 데이터 중 places[0].place name 값을 화면에 표시하기
제목 : “□도시 이름 : Beverly Hills”
힌트

- useEffect로 fetch 실행
- useState로 데이터 저장
- places[0].["place name"] 접근

□ 예상화면

도시 이름 : Beverly Hills

문제 2 : 게시글 목록 출력하기

요구사항
아래 주소의 데이터를 불러오기
https://jsonplaceholder.typicode.com/posts

제목(title)과 내용(body)을 &li&로 표시
상단에 “게시글 목록” 제목 출력
예시 화면
게시글 목록
1. sunt aut facere repellat provident occaecati...
2. qui est esse...
3. ea molestias quasi exercitationem repellat qui ipsa...
...

한 줄 정리

useEffect : “언제 실행할지”를 정하는 도구
fetch : “어디서 데이터를 가져올지”를 정하는 도구
두 개를 합치면 “데이터가 들어오면 자동으로 화면이 만들어지는 리액트 앱” 완성

Day 4 — 리스트 렌더링(map), 조건부 렌더링, key 완전 정복

오늘의 목표

- ✔ map() 으로 배열을 **리액트 방식**으로 화면에 뿌릴 수 있다.
- ✔ **조건부 렌더링**으로 상황에 맞는 화면을 보여준다.
- ✔ **key** 를 올바르게 사용해 오류, 깜빡임, 성능 문제를 피한다.
- ✔ 빈 상태, 로딩, 에러까지 **실전형 리스트 화면**을 만든다.

1. 왜 map 으로 렌더링할까?

리액트는 **데이터** → **화면** 흐름이다.
배열 데이터를 <li>, 로 바꿀 때 for 대신 map 을 쓰는 이유는 **함수형으로, 매번 새로운 UI 배열을 만들어** 리액트에게 넘기기가 표준이기 때문이다.

비유
주방에서 식판을 줄줄이 나열하는 느낌이 map 이다.
자료 한 줄 → 화면 한 칸 으로 바꾼다.

2. map 기본 예제

🗀 파일 : src/App.jsx

```
export default function App() {
  const fruits = ["사과", "바나나", "포도"];

  return (
    <div>
      <h2>과일 리스트</h2>
      <ul>
        {fruits.map((item) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    </div>
  );
}
```

포인트

{배열.map(...)} : JSX 안에서 바로 사용
key 는 각 항목을 **안정적으로 구별할 수 있는 값** (여기선 과일 이름이 고유하다고 가정)
절대map 바깥에서 push 로 JSX를 쌓지 않는다. (리액트 철학과 어긋남)

3. key 가 왜 중요할까?

리액트는 리스트가 바뀔 때 **어떤 항목이 추가/삭제/이동됐는지** 비교해서 최소만 다시 그린다.
이때 **key** 를 보고 "누가 누구인지" 판단한다.
좋은 key : 데이터의 진짜 ID (예 : DB id, uuid)
나쁜 key : 배열의 index (순서가 바뀌면 엉킴)
증상 : 체크박스가 다른 행으로 옮겨 붙음, 애니메이션 깜빡임, 예상 못한 재렌더

권장 좋은 예 : 서버에서 받은 고유 id 사용

todos.map(todo => <li key={todo.id}>{todo.text});

가급적 피하기

나쁜 예 : index 사용 (정렬/삽입/삭제에 취약)
todos.map((todo, idx) => <li key={idx}>{todo.text});

4. 조건부 렌더링 3가지 패턴

삼항 연산자 (가장 자주 사용)

{ isLogin ? 어서오세요 : 로그인이 필요합니다 }

&& 단축 평가 (조건이 true 일 때만)

{ cart.length === 0 && 장바구니가 비었습니다 }

조기 리턴 (컴포넌트 상단에서 빠르게 종료)

if (!items) return 로딩 중... ;

삼항은 두 갈래, && 는 한 갈래.
복잡해지면 **조기 리턴**으로 가독성을 지키자.

5. 실전형 리스트 화면 : 로딩 / 빈 상태 / 에러

🗀 파일 : src/App.jsx

```
import { useEffect, useState } from "react";

export default function App() {
  const [posts, setPosts] = useState(null);    // 아직 없음
  const [loading, setLoading] = useState(true);
  const [error, setError]    = useState(null);

  useEffect(() => {
    async function load() {
      try {
        const res = await fetch("https://jsonplaceholder.typicode.com/posts");
        if (!res.ok) throw new Error("네트워크 에러");
        const data = await res.json();
        setPosts(data.slice(0, 5)); // 5개만 보기
      } catch (e) {
        setError(e.message);
      } finally {
        setLoading(false);
      }
    }
    load();
  }, []);

  if (loading) return <p>로딩 중 ...</p>;
  if (error)    return <p>에러 발생 : {error}</p>;
  if (!posts || posts.length === 0) return <p>데이터가 없습니다</p>;

  return (
    <div>
      <h2>게시글 5개</h2>
      <ul>
        {posts.map(post => (
          <li key={post.id}>
            <strong>{post.id}. {post.title}</strong>
            <p>{post.body}</p>
          </li>
```

```
          ))}
        </ul>
      </div>
    );
}
```

예상 화면

게시글 5개
1. sunt aut facere ...
(본문...)
2. qui est esse ...

6. 검색 필터 + 빈 상태 메시지

🗀 파일 : src/App.jsx

```
import { useEffect, useState } from "react";

export default function App() {
  const [users, setUsers] = useState([]);
  const [q, setQ] = useState("");

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(r => r.json())
      .then(setUsers);
  }, []);

  const filtered = users.filter(u =>
    [u.name, u.username, u.email]
      .join(" ")
      .toLowerCase()
      .includes(q.toLowerCase().trim())
  );

  return (
    <div style={{ maxWidth : 520, margin : "40px auto" }}>
      <h2>사용자 검색</h2>
      <input
```

```
        placeholder="이름 / 아이디 / 이메일"
        value={q}
        onChange={e => setQ(e.target.value)}
        style={{ width : "100%", padding : 10, borderRadius : 8 }}
      />

      {filtered.length === 0 ? (
        <p style={{ marginTop : 12 }}>검색 결과가 없습니다</p>
      ) : (
        <ul style={{ marginTop : 12 }}>
          {filtered.map(u => (
            <li key={u.id}>
              <strong>{u.name}</strong> — {u.email}
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}
```

예상 화면

검색어에 따라 리스트가 즉시 줄어들거나 검색 결과가 없습니다 가 보임

7. 정렬과 제한 : 정렬 버튼, 상위 N개 보기

🗀 파일 : src/examples/SortAndSlice.jsx

```
import { useEffect, useState } from "react";

export default function SortAndSlice() {
  const [items, setItems] = useState([]);
  const [asc, setAsc] = useState(true);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then(r => r.json())
      .then(data => setItems(data.slice(0, 10))); // 10개만 사용
  }, []);
```

```
  const sorted = [...items].sort((a, b) =>
    asc ? a.title.localeCompare(b.title) : b.title.localeCompare(a.title)
  );

  return (
    <div>
      <h2>할 일 10개 (정렬 토글)</h2>
      <button onClick={() => setAsc(!asc)}>
        정렬 순서 : {asc ? "오름차순" : "내림차순"}
      </button>
      <ul>
        {sorted.map(i => (
          <li key={i.id}>{i.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

포인트

불변성 유지 : const sorted = [...items].sort(...)
원본 배열을 직접 정렬하지 말고 **복사본**을 정렬

8. 실전 패턴 : 카드 리스트 컴포넌트 분리

🗀 파일 : src/components/UserCard.jsx

```
export default function UserCard({ name, email, company }) {
  return (
    <div style={{
      border : "1px solid #ddd",
      borderRadius : 12,
      padding : 12,
      marginBottom : 10
    }}>
      <strong>{name}</strong>
      <div>{email}</div>
      <small>{company?.name}</small>
```

```
    </div>
  );
}
```

파일 : src/App.jsx

```
import { useEffect, useState } from "react";
import UserCard from "./components/UserCard";

export default function App() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(r => r.json())
      .then(setUsers);
  }, []);

  return (
    <div style={{ maxWidth : 520, margin : "40px auto" }}>
      <h2>사용자 카드</h2>
      {users.length === 0 ? (
        <p>로딩 중 ...</p>
      ) : (
        users.map(u => (
          <UserCard
            key={u.id}
            name={u.name}
            email={u.email}
            company={u.company}
          />
        ))
      )}
    </div>
  );
}
```

포인트

리스트 화면은 부모 : 데이터 로딩 + 필터 / 자식 : 표시 전담 으로 분리하면 깔끔

key 는 **부모 map** 쪽에서 지정

9. 자주 하는 실수와 해결

실수	증상	해결
key 에 index 사용	행 이동, 삭제 후 체크박스가 엉뚱한 행에 붙음	고유 id 사용
로딩 상태 처리 없음	빈 화면 or 깜박임	loading, error, empty 3단계 처리
거대한 map 안에서 무거운 계산	스크롤 끊김	미리 계산해서 state 로 저장하거나 useMemo 고려
조건부 렌더링 복잡	가독성 저하	조기 리턴으로 단순화
원본 배열 변형	예상치 못한 화면	map, filter, slice, 전개 연산자(...)로 불변성 유지

10. 종합 예제 : 로딩 → 검색 → 리스트 → 조건부 표시

🗀 파일 : src/App.jsx

```
import { useEffect, useState } from "react";

export default function App() {
  const [data, setData]       = useState([]);
  const [q, setQ]             = useState("");
  const [loading, setLoad]    = useState(true);
  const [error, setError]     = useState(null);

  useEffect(() => {
    (async () => {
      try {
        const r = await fetch("https://jsonplaceholder.typicode.com/albums");
        if (!r.ok) throw new Error("불러오기 실패");
        const json = await r.json();
        setData(json.slice(0, 20));
      } catch (e) {
        setError(e.message);
      } finally {
        setLoad(false);
      }
```

```
    })();
  }, []);

  if (loading) return <p>로딩 중 ...</p>;
  if (error)   return <p>에러 : {error}</p>;

  const filtered = data.filter(a =>
    a.title.toLowerCase().includes(q.toLowerCase().trim())
  );

  return (
    <div style={{ maxWidth : 680, margin : "40px auto" }}>
      <h2>앨범 리스트</h2>
      <input
        placeholder="제목 검색"
        value={q}
        onChange={e => setQ(e.target.value)}
        style={{ width : "100%", padding : 10, borderRadius : 8 }}
      />

      {filtered.length === 0 ? (
        <p style={{ marginTop : 12 }}>검색 결과가 없습니다</p>
      ) : (
        <ul style={{ marginTop : 12 }}>
          {filtered.map(a => (
            <li key={a.id}>
              <strong>{a.id}. {a.title}</strong>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}
```

실습 문제 2개 (새 프로젝트에서 진행)

문제 1 : 출석부 만들기 (검색 + 정렬)

요구사항

- 학생 더미 데이터 15명 : { id, name, className }
- 상단 검색 입력 : 이름으로 필터
- 정렬 버튼 2개 : 이름 오름차순 / 내림차순
- 빈 결과면 "검색 결과가 없습니다" 표시
- key 는 id 사용

파일 구조 예시

```
src/
 ├ App.jsx
 └ components/
    └ StudentRow.jsx
```

문제 2 : 할 일 리스트 (체크 + 남은 개수 표시)

요구사항

- 초기 todo 배열 8개 : { id, text, done }
- 체크박스로 완료 토글
- 상단에 "남은 할 일 : N개" 표시
- 필터 버튼 3개 : 전체 / 진행중 / 완료
- key 는 id 사용, 원본 불변성 유지

힌트

토글 : setTodos(todos.map(t => t.id === id ? {...t, done : !t.done} : t))

남은 개수 : todos.filter(t => !t.done).length

핵심 한 줄

map 으로 그린다, key 로 구별한다, 조건부로 상황을 나눈다.

여기에 로딩 · 빈 상태 · 에러까지 챙기면 **실무형 리스트 화면** 완성

Day 5 — Chart.js 입문 : React에서 막대 · 선 · 원형 차트 만들기

오늘의 목표

- ✔ 차트가 필요한 이유와 기본 구조를 이해한다.
- ✔ Chart.js 와 react-chartjs-2 를 설치하고 화면에 띄운다.
- ✔ 막대 그래프, 선 그래프, 도넛 / 파이 그래프를 만들어 본다.
- ✔ 차트 데이터 구조와 옵션 구조를 읽을 수 있다.
- ✔ 나중에 API 데이터를 연결할 수 있도록 "데이터 변환" 감을 잡는다.

왜 차트를 쓸까?

표만 보면 숫자가 **빽빽**해서 한눈에 안 들어온다.
차트는 **변화**와 **비교**를 눈으로 바로 보여 준다.
팀 프로젝트에서 **대시보드**를 만들 때 필수 스킬이다.
표는 성적표, 차트는 성적 추이 그래프.
추세와 차이를 한 번에 보여 주는 도구가 **차트이다.**

1. 설치와 준비

설치

터미널에서 프로젝트 루트에서 설치한다.

```
npm i chart.js react-chartjs-2
```

chart.js : 차트 엔진
react-chartjs-2 : 리액트에서 차트를 컴포넌트처럼 쓰게 해주는 래퍼

가져오기 방법 두 가지

가장 쉬운 방법은 **자동 등록이다.**

```
// 자동 등록 : 한 줄이면 요소들이 자동으로 등록됩니다.
import "chart.js/auto";
// 수동 등록 : 필요한 요소만 골라 등록
import {
  Chart as ChartJS,
  BarElement, LineElement, ArcElement,
  CategoryScale, LinearScale, PointElement, Tooltip, Legend
```

```
} from "chart.js";

ChartJS.register(
  BarElement, LineElement, ArcElement,
  CategoryScale, LinearScale, PointElement, Tooltip, Legend
);
```

2. 차트의 기본 구조 이해

Chart.js 는 기본적으로 아래 두 가지를 받는다.
data : 무엇을 그릴지, **options** : 어떻게 보여 줄지

데이터 구조

```
const data = {
  labels: ["1월", "2월", "3월"],          // x축 레이블
  datasets: [
    {
      label: "매출",
      data: [120, 90, 150],                // y축 값들
      backgroundColor: "rgba(99, 102, 241, .5)", // 색 (막대·면)
      borderColor: "rgba(99, 102, 241, 1)",      // 선 색
      borderWidth: 1
    }
  ]
};
```

옵션 구조

```
const options = {
  responsive: true,                // 반응형
  maintainAspectRatio: false,     // 높이 고정하고 싶을 때 false
  plugins: {
    legend: { display: true },
    title:  { display: true, text: "월별 매출" }
  },
  scales: {
    y: { beginAtZero: true }      // y축 0부터
  }
};
```

차트 선택 가이드

상황	가장 적합	차선
카테고리 간 **절대값 비교**	**막대 차트**	도넛/파이
시간에 따른 변화 추세	**선 차트**	막대 차트
비율 · 점유율 보여주기	**도넛/파이**	누적 막대
항목이 많고 **값 차이 큼**	**막대 · 선**	트리맵 등 고급 차트
두 지표를 함께 비교	**혼합 차트** (막대+선)	보조축 포함 선 차트
사용자에게 **빠른 이해** 제공	**막대 · 선**	도넛

3. 막대 그래프 : 기초 예제

언제 쓰나

서로 다른 항목의 절대값 비교
예시 : 반별 평균 점수, 제품군별 매출, 지역별 인원, 요일별 판매량
순위를 드러낼 때 : 값이 큰 순서로 정렬하면 즉시 이해 가능

데이터 구조 설계 팁

labels 는 이산형 카테고리 : ["A반", "B반", "C반"]
datasets[n].data 는 각 라벨에 대응하는 값
비교 대상이 여러 개면 : dataset 을 둘 이상 넣되 색을 명확히 구분

```
// 그룹 막대  :  같은 라벨에 여러 값 비교
const data = {
  labels : ["1분기", "2분기", "3분기", "4분기"],
  datasets : [
    { label : "온라인", data : [120, 140, 160, 180], borderWidth : 1 },
    { label : "오프라인", data : [90, 110, 105, 130], borderWidth : 1 }
  ]
};
```

가로 막대는 라벨이 길거나 항목이 많을 때 가독성

```
const options = { indexAxis : "y" };  // 가로 막대
```

누적 막대 : 부분의 합 = 전체를 보여줄 때
범주별로 구성 비율을 한 번에 표시

```
const options = {
  scales : {
```

```
    x : { stacked : true },
    y : { stacked : true, beginAtZero : true }
  }
};
```

디자인 · UX 팁

- 막대 간격이 너무 좁으면 수치가 겹쳐 보입니다. 데이터가 많다면 : 가로 막대 + 스크롤 전략
- 정렬 : 큰 값 → 작은 값으로 정렬하면 이해가 매우 빠름
- 라벨 길이가 길면 줄바꿈 또는 짧은 약어를 사용
- 항목이 20개 넘는데 세로 막대 → 가로 막대로 전환
- 값 스케일이 매우 커 차이가 안 보임 → 로그 축 또는 보조 지표 추가

파일 경로 : src/charts/BarBasic.jsx

```
import "chart.js/auto";
import { Bar } from "react-chartjs-2";

export default function BarBasic() {
  const data = {
    labels: ["1월", "2월", "3월", "4월", "5월"],
    datasets: [
      {
        label: "매출",
        data: [120, 90, 150, 80, 130],
        backgroundColor: "rgba(99, 102, 241, .6)",
        borderColor: "rgba(99, 102, 241, 1)",
        borderWidth: 1
      }
    ]
  };

  const options = {
    responsive: true,
    maintainAspectRatio: false,
    plugins: { legend: { display: true }, title: { display: true, text: "월별 매출" } },
    scales: { y: { beginAtZero: true } }
  };

  return (
    <div style={{ height : 360 }}>
```

```
      <Bar data={data} options={options} />
    </div>
  );
}
```

App 에서 사용

파일 경로 : src/App.jsx

```
import BarBasic from "./charts/BarBasic";

export default function App() {
  return (
    <div style={{ padding : 24 }}>
      <h1>Chart.js 입문</h1>
      <BarBasic />
    </div>
  );
}
```

4. 선 그래프 : 변화 추세 보기

언제 쓰나

시간 축을 따라 값이 어떻게 변했는지를 보여줄 때
예시 : 일별 방문자, 월별 매출, 시간대별 온도, 주가 변동

데이터 구조 설계 팁

labels 는 시간 순서 : ["1월", "2월", ...] 또는 날짜 배열
시간 데이터라면 차트 스케일을 time 으로 두면 표시가 깔끔

```
// time 스케일 : 날짜 라벨 자동 포맷
import "chart.js/auto";
const options = {
  scales : {
    x : { type : "time", time : { unit : "day" } },
    y : { beginAtZero : true }
  }
};
```

time 스케일은 dayjs, luxon 등 어댑터 추가가 필요할 수 있다.
(입문 단계에서는 문자열 라벨로도 충분히 실습 가능하다.)

부드러운 곡선과 포인트

```
const dataset = {
  tension : 0.3,        // 0 : 직선 / 0.4 전후 : 적당히 곡선
  pointRadius : 3,      // 점 크기
  fill : false          // 아래 면 채우기 여부
};
```

여러 지표 동시 비교

- 두 라인을 포개면 상관 추세 보기에 좋음
- 값 단위가 다르면 보조축(y1) 사용

```
const datasets = [
  { label : "매출(만원)", data : sales, yAxisID : "y" },
  { label : "방문자(명)", data : visits, yAxisID : "y1", tension : 0.3 }
];
const scales = {
  y  : { beginAtZero : true, position : "left" },
  y1 : { beginAtZero : true, position : "right", grid : { drawOnChartArea : false } }
};
```

디자인 · UX 팁

- 점 개수가 많을수록 포인트를 작게 또는 숨기기
- 결측치(null) 는 선이 끊길 수 있음 → spanGaps : true 로 보간 표시 가능
- 색 의미 통일 : 매출 : 파랑, 경고/실패 : 빨강 등
- 순서가 뒤섞인 라벨 → 반드시 시간 순서로 정렬
- 카테고리 데이터에 선 차트를 쓰는 경우 → 막대가 더 적합

파일 경로 : src/charts/LineBasic.jsx

```
import "chart.js/auto";
import { Line } from "react-chartjs-2";

export default function LineBasic() {
  const data = {
    labels: ["월", "화", "수", "목", "금", "토", "일"],
    datasets: [
      {
```

```
        label: "방문자 수",
        data: [120, 180, 90, 160, 220, 300, 280],
        tension: 0.3,                    // 선을 약간 곡선으로
        fill: false,                     // 면 채우기 여부
        borderWidth: 2
      }
    ]
  };

  const options = {
    responsive: true,
    maintainAspectRatio: false,
    plugins: { title: { display: true, text: "요일별 방문자 추이" } },
    scales: { y: { beginAtZero: true } }
  };

  return (
    <div style={{ height : 360 }}>
      <Line data={data} options={options} />
    </div>
  );
}
```

5. 도넛 / 파이 : 비율 비교

언제 쓰나

하나의 전체가 어떤 구성 요소로 얼마나 이루어졌는지
예시 : 지출 비율, 시장 점유율, 카테고리 비중

사용 시 주의

조각이 6개 이상이면 인지 부하 ↑ → 상위 4~5개만 남기고 그 외 기타로 묶기
조각 간 미세한 차이는 알아보기 어려움 → 누적 막대가 더 정확하게 읽힘
비율 합이 100 % 가 맞는지 항상 점검

```
// 도넛 기본
const data = {
  labels : ["식비", "교통", "주거", "취미"],
```

```
  datasets : [{ data : [35, 12, 45, 18] }]  // 합계 : 110 → 비율로 환산해도 OK
};
const options = {
  plugins : {
    legend : { position : "bottom" },
    title  : { display : true, text : "지출 비율" },
    tooltip : {
      callbacks : {
        label : (ctx) => {
          const total = ctx.dataset.data.reduce((a, b) => a + b, 0);
          const val   = ctx.parsed;
          const pct   = Math.round((val / total) * 100);
          return ` ${val.toLocaleString()}원 · ${pct}%`;
        }
      }
    }
  }
};
```

도넛 중앙에 합계 텍스트

플러그인으로 캔버스 중앙에 합계를 그려주면 이해가 더 빨라짐다.
입문과정에서는 아래쪽에 합계를 텍스트로 먼저 표시해도 충분

```
// 합계 표시용 JSX (간단)
const total = values.reduce((a, b) => a + b, 0);
return (
  <>
    <Doughnut data={data} options={options} />
    <p style={{ textAlign : "center", marginTop : 8 }}>
      총합 : { total.toLocaleString() } 원
    </p>
  </>
);
```

누적 막대 vs 도넛 : 언제 무엇을

정확한 수치 차이가 중요 → 누적 막대
전체 구성의 느낌만 빠르게 → 도넛/파이

파일 경로 : src/charts/DoughnutBasic.jsx

```
import "chart.js/auto";
import { Doughnut } from "react-chartjs-2";

export default function DoughnutBasic() {
  const data = {
    labels: ["식비", "교통", "주거", "취미"],
    datasets: [
      {
        label: "지출 비율",
        data: [35, 15, 30, 20],
        borderWidth: 1
      }
    ]
  };

  const options = {
    plugins: {
      title: { display: true, text: "카테고리별 지출 비율" },
      legend: { position: "bottom" }
    }
  };

  return <Doughnut data={data} options={options} />;
}
```

6. 한 화면에 여러 차트 배치

혼합 차트 : 서로 다른 형태를 한 화면에서

- 동일한 labels 를 공유하면 막대 + 선 같은 조합이 가능
- 예시 : 막대 : 월 매출, 선 : 방문자
- 값 단위가 다르면 보조축을 써서 헷갈림 방지

파일 경로 : src/App.jsx

```
import BarBasic from "./charts/BarBasic";
import LineBasic from "./charts/LineBasic";
```

```
import DoughnutBasic from "./charts/DoughnutBasic";

export default function App() {
  return (
    <div style={{ padding : 24, display : "grid", gap : 24 }}>
      <h1>데이터 시각화 데모</h1>
      <div style={{ display : "grid", gridTemplateColumns : "1fr 1fr", gap : 24 }}>
        <div style={{ border : "1px solid #eee", borderRadius : 12, padding : 16 }}>
          <BarBasic />
        </div>
        <div style={{ border : "1px solid #eee", borderRadius : 12, padding : 16 }}>
          <LineBasic />
        </div>
      </div>
      <div style={{ border : "1px solid #eee", borderRadius : 12, padding : 16,
maxWidth : 560 }}>
        <DoughnutBasic />
      </div>
    </div>
  );
}
```

7. 데이터 변환 감 잡기 : 표 → 차트 데이터로

```
// 예시 : 매출 표 데이터
const rows = [
  { month: "1월", sales: 120 },
  { month: "2월", sales: 90  },
  { month: "3월", sales: 150 }
];
```

차트에 넣으려면 **labels** 와 **data** 로 분리

```
const labels = rows.map(r => r.month);  // ["1월","2월","3월"]
const values = rows.map(r => r.sales);  // [120, 90, 150]
```

8. 비동기 데이터 연결 연습

하드코딩 → 상태 → API 순서로 확장합니다.

파일 경로 : src/charts/BarFromState.jsx

```
import "chart.js/auto";
import { useEffect, useState } from "react";
import { Bar } from "react-chartjs-2";

export default function BarFromState() {
  // 1) 처음엔 빈 배열
  const [rows, setRows] = useState([]);

  // 2) 마운트 시 데이터 로드 (여기서는 setTimeout 으로 흉내)
  useEffect(() => {
    setTimeout(() => {
      setRows([
        { month: "1월", sales: 120 },
        { month: "2월", sales: 90  },
        { month: "3월", sales: 150 },
        { month: "4월", sales: 80  },
        { month: "5월", sales: 130 }
      ]);
    }, 800);
  }, []);

  // 3) 변환
  const labels = rows.map(r => r.month);
  const values = rows.map(r => r.sales);

  const data = {
    labels,
    datasets: [{ label: "매출", data: values, borderWidth: 1 }]
  };

  const options = {
    responsive: true,
    maintainAspectRatio: false,
    plugins: { title: { display: true, text: "월별 매출 (동적 데이터)" } },
    scales: { y: { beginAtZero: true } }
```

```
  };

  // 4) 로딩 상태
  if (rows.length === 0) return <p>   데이터 준비 중 ...</p>;

  return <div style={{ height : 360 }}><Bar data={data} options={options} /></div>;
}
```

9. 자주 만나는 오류와 해결법

증상 / 에러	원인	해결
차트가 안 뜬다	chart.js/auto 를 안 가져옴	import "chart.js/auto" 추가
Cannot find Bar	react-chartjs-2 미설치	npm i react-chartjs-2
데이터는 있는데 빈 화면	labels 와 datasets[i].data 길이가 안 맞음	배열 길이 확인
비율이 너무 납작함	높이가 부족	래퍼 div 에 height 스타일 주기 + maintainAspectRatio : false
레전드 제목이 안 보임	plugins.title.display 빠짐	display : true 설정

10. 접근성 · UX 팁

색만으로 구분하지 말고 **레이블 / 범례 / 수치 표시**도 함께 보여 주기
범례 위치는 **bottom**이 이해에 편한 경우가 많다.
차트는 **한 화면에 2~3개까지만**. 너무 많으면 집중도가 떨어지고 성능도 저하된다.
값의 단위는 항상 같이 적기 : 예시 “매출 (만원)”, “온도 (℃)”.

실 습 문 제

문제 1 : 반별 시험 성적 막대 그래프

파일 경로 : src/charts/ClassScoreBar.jsx
요구 사항 :

반 이름 : A, B, C, D
평균 점수 : [72, 85, 64, 91]
y축은 0 에서 100
제목 : 반별 평균 점수
레전드 표시 켜기
힌트 :
labels : ["A", "B", "C", "D"]
data : [72, 85, 64, 91]
scales : { y : { beginAtZero : true, max : 100 } }

문제 2 : 주간 기온 선 그래프

파일 경로 : src/charts/WeeklyTempLine.jsx
요구 사항 :
요일 라벨 : 월 ~ 일
최고 기온 : [3, 5, 7, 4, 6, 8, 7]
선은 부드럽게, 점은 보이도록
제목 : 일주일 최고 기온 추이
힌트 :
tension : 0.3
pointRadius : 3

문제 3 : 카테고리별 지출 도넛 + 합계 표시

파일 경로 : src/charts/SpendDoughnut.jsx
데이터 :

```
const rows = [
  { cat : "식비",  amt : 350000 },
  { cat : "교통",  amt : 120000 },
  { cat : "주거",  amt : 450000 },
  { cat : "취미",  amt : 180000 }
];
```

요구 사항 :
도넛 차트로 비율 표시
아래쪽에 **총합**과 **가장 큰 카테고리** 텍스트로 출력
범례는 하단
힌트 :

```
const labels = rows.map(r =& r.cat);
const data   = rows.map(r =& r.amt);
const total  = rows.reduce((s, r) =& s + r.amt, 0);
const topCat = rows.reduce((a, b) =& a.amt & b.amt ? a : b).cat;
```

핵심 정리

키워드	설명
data	labels + datasets 로 구성
options	제목, 범례, 축, 반응형 등 표시 방식
Bar / Line / Doughnut	비교, 추세, 비율에 각각 적합
변환	표·JSON → labels, data 로 바꿔 넣기
트러블 슈팅	chart.js/auto, 높이 지정, 배열 길이 체크

Day 6 — 동적 차트 만들기

핵심 키워드 : 컴포넌트 분리, props 주입, 상태 변경, 옵션 커스터마이즈, 미니 대시보드

무엇을 배우나?

재사용 가능한 차트 컴포넌트를 만든다. → *상위에서 데이터만 바꿔 끼우면 어디서든 사용 가능*
사용자 상호작용(버튼/드롭다운)으로 **상태(state)를 바꾸고, 차트를 즉시 갱신**한다.
Chart.js 옵션(축/툴팁/범례/단위)을 커스터마이즈한다.
여러 차트(막대 · 선 · 도넛/혼합)를 **그리드 레이아웃**으로 묶어 **미니 대시보드**를 만든다.

1. 재사용 가능한 차트 컴포넌트 (props 주입)

왜 이렇게 하나?

관심사 분리 : “데이터 준비/상태 관리”는 부모가, “그리기”는 자식이 담당 → 유지보수 쉬움
재사용 : 같은 컴포넌트를 다른 화면에서 다양한 데이터로 재활용 가능
테스트 용이 : 시각화 로직을 독립적으로 점검하기 쉬움

예제 : BarChartCard (막대 차트 카드)

파일 :src/components/BarChartCard.jsx

```
import "chart.js/auto";
import { Bar } from "react-chartjs-2";

export default function BarChartCard({
  title,
  labels,
  values,
  color = "rgba(99,102,241,1)" // indigo-500
}) {
  // data : 무엇을 그릴지 (labels + datasets)
  const data = {
    labels,
    datasets: [
      {
        label: title,
        data: values,
        // 같은 색을 테마로 "면"은 연하게, "테두리"는 진하게 → 시각 대비·가독성
        backgroundColor: color.replace("1)", ".2)"),
        borderColor: color,
        borderWidth: 1,
        borderRadius: 8 // 막대 모서리 둥글게 → 읽기 쉬운 UI
      }
    ]
  };
```

```
  // options : 어떻게 보여줄지 (축/툴팁/범례/반응형 등)
  const options = {
    responsive: true,                // 부모 컨테이너 폭에 반응
    maintainAspectRatio: false,    // 고정 높이(h-80) 우선 → 대시보드 카드 배치 안정
    plugins: {
      title: { display: true, text: title }, // 카드 제목
      legend: { display: false }                // 단일 데이터셋이면 범례 생략 → 시선 분산 방지
    },
    scales: { y: { beginAtZero: true } }       // y축 0부터 → 비교를 공정하게
  };

  return (
    &div className="h-80 rounded-2xl border border-gray-200 bg-white p-4 shadow-sm"&
      {/* react-chartjs-2 : data와 options가 바뀌면 알아서 리렌더 */}
      &Bar data={data} options={options} /&
    &/div&
  );
}
```

왜 import "chart.js/auto"?
Chart.js의 필요한 요소(BarElement, CategoryScale 등)를 **자동 등록**해 줍니다. 초보 단계에서는 가장 간단·안정적인 방식입니다.
왜 maintainAspectRatio: false + 카드 h-80?
대시보드는 "카드 높이 통제"가 중요합니다. h-80으로 **높이를 Tailwind로 고정**하고, 차트는 그 높이에 **가득 맞추도록** 하는 패턴이 실무에서 안정적입니다.

2️⃣ 상호작용으로 차트 갱신 (state 변경)

핵심 원리

•

버튼/드롭다운 → **state 변경**
state가 바뀌면 → React가 **리렌더**
data/labels가 새 값으로 바뀌면서 차트도 **자동 업데이트**

예제 : 분기(Quarter) 필터

파일 :src/examples/QuarterFilter.jsx

```
import { useState } from "react";
import BarChartCard from "../components/BarChartCard";

const months = ["1월","2월","3월","4월","5월","6월","7월","8월","9월","10월","11월","12월"];
const sales12 = [120, 90, 150, 80, 130, 160, 140, 170, 110, 180, 200, 220];

// q(1~4)에 맞는 3개월 분량을 잘라 반환
function sliceQuarter(q) {
```

```
  const s = (q - 1) * 3;
  return { labels: months.slice(s, s + 3), values: sales12.slice(s, s + 3) };
}

export default function QuarterFilter() {
  const [q, setQ] = useState(1);                    // 현재 선택된 분기
  const { labels, values } = sliceQuarter(q);       // 분기 변경 시 재계산

  return (
    &div className="space-y-3"&
      &div className="flex gap-2"&
        {[1,2,3,4].map(n =& (
          &button
            key={n}
            onClick={() =& setQ(n)} // 상태 변경 → 차트 갱신
            className={`rounded-lg px-3 py-2 text-sm ${
              q === n ? "bg-indigo-600 text-white" : "bg-gray-100 hover:bg-gray-200"
            }`}
          &
            {n}분기
          &/button&
        ))}
      &/div&

      &BarChartCard title={`${q}분기 매출`} labels={labels} values={values} /&
    &/div&
  );
}
```

왜 slice로 잘라 쓰나?
원본 데이터(12개월)는 유지하고, **표시 구간만 변형**해야 추후 다른 필터/조합(연/분기/월)로 확장하기 쉽습니다.
상태는 **불변성**을 지켜야 예측 가능하고 버그가 줄어듭니다.

예제 : 드롭다운으로 항목 선택

파일 :src/examples/SelectCategory.jsx

```
import { useMemo, useState } from "react";
import BarChartCard from "../components/BarChartCard";

const rows = [
  { cat: "식비",   jan: 35, feb: 28, mar: 42 },
  { cat: "교통",   jan: 12, feb: 14, mar: 13 },
  { cat: "주거",   jan: 45, feb: 47, mar: 46 },
  { cat: "취미",   jan: 18, feb: 22, mar: 20 }
```

```
];

export default function SelectCategory() {
  const [cat, setCat] = useState("식비");

  // 선택된 카테고리의 1~3월 데이터를 계산 (cat이 바뀔 때만 재계산)
  const item = useMemo(() =& rows.find(r =& r.cat === cat), [cat]);

  const labels = ["1월","2월","3월"];
  const values = [item.jan, item.feb, item.mar];

  return (
    &div className="max-w-xl space-y-3"&
      &select
        value={cat}
        onChange={(e) =& setCat(e.target.value)}
        className="w-full rounded-lg border border-gray-300 bg-white px-3 py-2"
      &
        {rows.map(r =& (
          &option key={r.cat} value={r.cat}&{r.cat}&/option&
        ))}
      &/select&

      &BarChartCard title={`${cat} 지출 (만원)`} labels={labels} values={values} /&
    &/div&
  );
}
```

왜 useMemo?
입력값(cat)이 바뀔 때만 **필요한 계산**을 수행하고, 나머지 렌더에서는 이전 값을 재사용 → **불필요한 연산/리렌더 감소**(성능·안정성).
입문 단계에서 꼭 필수는 아니지만, “변환/가공 비용이 커지는 상황”을 대비한 습관입니다.

---

3. 옵션 커스터마이즈 (축/툴팁/범례/단위)

왜 중요하나?

- **읽기 쉬운 단위 표기**(만원, ℃, 명…)는 해석 속도를 크게 높입니다.
- 툴팁/축 포맷팅은 **현업 가독성**의 핵심 포인트입니다.

예제 : 보기 좋은 막대 (단위/툴팁/범례)

파일 :src/components/BarChartPretty.jsx

```
import "chart.js/auto";
import { Bar } from "react-chartjs-2";
```

```
export default function BarChartPretty({ title, labels, values, unit = "만원" }) {
  const data = {
    labels,
    datasets: [
      {
        label: title,
        data: values,
        backgroundColor: "rgba(16,185,129,.25)", // emerald-500 @ .25
        borderColor: "rgba(16,185,129,1)",
        borderWidth: 1,
        borderRadius: 10
      }
    ]
  };

  const options = {
    responsive: true,
    maintainAspectRatio: false,
    plugins: {
      legend: { position: "bottom" },              // 범례는 하단 → 시선 흐름 자연스러움
      title: { display: true, text: title },
      tooltip: {
        callbacks: {
          // 툴팁 : 숫자 + 단위, 천단위 콤마
          label: (ctx) =& {
            const v = ctx.parsed.y ?? 0;
            return ` ${v.toLocaleString()} ${unit}`;
          }
        }
      }
    },
    scales: {
      y: {
        beginAtZero: true,
        // y축 틱 : 숫자 + 단위, 천단위 콤마
        ticks: { callback: (v) =& `${Number(v).toLocaleString()} ${unit}` }
      }
    }
  };

  return (
    &div className="h-80 rounded-2xl border border-gray-200 bg-white p-4 shadow-sm"&
      &Bar data={data} options={options} /&
    &/div&
  );
```

}

왜 툴팁/틱에서 toLocaleString()?
3자리 콤마가 자동으로 들어가 **가독성**이 좋아집니다. 사용자 로케일에 따라 포맷도 현지화됩니다.

---

4. 혼합 차트 (막대 + 선 + 보조축)

왜 필요하나?

•
단위가 다른 두 지표(예 : 매출=만원, 방문자=명)를 **한 화면**에서 직관적으로 비교
yAxisID 를 나눠 **좌/우 축**으로 분리하면 단위 혼동이 사라짐

예제 : MixedChart

파일 :src/components/MixedChart.jsx

```
import "chart.js/auto";
import { Chart } from "react-chartjs-2";

export default function MixedChart() {
  const labels = ["1월","2월","3월","4월","5월","6월"];

  const data = {
    labels,
    datasets: [
      {
        type: "bar",
        label: "매출 (만원)",
        data: [120, 90, 150, 80, 130, 160],
        backgroundColor: "rgba(59,130,246,.25)",
        borderColor: "rgba(59,130,246,1)",
        borderWidth: 1,
        borderRadius: 8,
        yAxisID: "y" // 왼쪽 축
      },
      {
        type: "line",
        label: "방문자 (명)",
        data: [520, 610, 480, 700, 820, 900],
        borderColor: "rgba(234,88,12,1)",
        tension: 0.3, // 부드러운 곡선
        yAxisID: "y1"  // 오른쪽 축
      }
    ]
  };

  const options = {
```

```
      responsive: true,
      maintainAspectRatio: false,
      plugins: {
        legend: { position: "bottom" },
        title: { display: true, text: "매출 vs 방문자" }
      },
      scales: {
        y:  { beginAtZero: true, position: "left",  ticks: { callback: v =& `${v} 만원` } },
        y1: { beginAtZero: true, position: "right", grid: { drawOnChartArea: false }, ticks: { callback: v =&
`${v} 명` } }
      }
    };

  return (
    &div className="h-80 rounded-2xl border border-gray-200 bg-white p-4 shadow-sm"&
      &Chart type="bar" data={data} options={options} /&
    &/div&
  );
}
```

왜 grid.drawOnChartArea: false?
오른쪽(y1) 격자선을 숨겨 **이중 격자선 겹침**을 방지 → 가독성 개선.

---

5 미니 대시보드 (그리드 + 카드 조합)

왜 그리드?

•

반응형으로 2열/1열 전환 쉬움
차트 카드를 **같은 높이**로 맞춰 **균형감** 확보
파일 :src/pages/MiniDashboard.jsx

```
import BarChartPretty from "../components/BarChartPretty";
import MixedChart from "../components/MixedChart";
import "chart.js/auto";
import { Doughnut } from "react-chartjs-2";

export default function MiniDashboard() {
  const labels = ["식비","교통","주거","취미"];
  const values = [35, 12, 45, 18];

  const doughnut = { labels, datasets: [{ data: values, borderWidth: 1 }] };
  const doughnutOpt = {
    plugins: {
      title: { display: true, text: "지출 비율" },
      legend: { position: "bottom" }
```

```
    }
  };

  return (
    &div className="p-6 space-y-6"&
      &h1 className="text-2xl font-bold"&미니 대시보드&/h1&

      &div className="grid gap-6 md:grid-cols-2"&
        &BarChartPretty
          title="월별 매출"
          labels={["1월","2월","3월","4월","5월","6월"]}
          values={[120, 90, 150, 80, 130, 160]}
          unit="만원"
        /&
        &MixedChart /&
      &/div&

      &div className="mx-auto max-w-xl"&
        &div className="rounded-2xl border border-gray-200 bg-white p-4 shadow-sm"&
          &Doughnut data={doughnut} options={doughnutOpt} /&
          &p className="mt-2 text-center text-sm text-gray-600"&
            총합 : {values.reduce((a, b) =& a + b, 0).toLocaleString()} 만원
          &/p&
        &/div&
      &/div&
    &/div&
  );
}
```

왜 합계를 따로 텍스트로?
도넛 중앙 플러그인을 쓰지 않아도 **합계·해석 포인트**를 명시하면 사용자 이해 속도가 빨라집니다.

---

6. 혼한 문제와 “왜 그렇게 고치는지”

문제	원인	왜 발생?	해결·이유
차트가 안 뜸	chart.js/auto 누락	요소 미등록	상단에 import "chart.js/auto" 추
값 있는데 빈 화면	labels.length !== data.length	x/y 개수 불일치	변환 로직에서 길이 확인 → 1:1 ㅂ
카드가 납작	기본 비율 유지	캔버스가 가로폭에 비례	카드에 h-80 + maintainAspectR 보
갱신 안 됨	상태 직접변경	참조가 같으면 React가 변화 인지 못함	**새 배열**로 set(map/slice/spread)
라벨 겹침	항목 과다	공간 부족	가로 막대(indexAxis: "y"), 상위 N

---

7. 실습 과제(이유 포함)

□ 실습 1 : 3·6·12개월 버튼

이유 : 같은 데이터에서 **표시 범위만 조절**하는 패턴 연습 (불변성 + slice)
버튼은 Tailwind로 명확한 상태 표시(bg-indigo-600 text-white)

□ 실습 2 : 라인 데이터셋 가시성 토글

이유 : 하나의 차트에서 **데이터셋을 숨기거나 보이기**(dashboard 필수)
hidden: true 를 토글하는 식으로 구현

---

8. 한눈 정리 (왜 이렇게 하는가?)

컴포넌트 분리 + props 주입 : 재사용·유지보수·확장성 최고
state 변경 → 차트 갱신 : React의 선언적 렌더링 원리 사용
옵션 커스터마이즈 : 읽기 쉬운 숫자/단위/범례/툴팁 → 실무적 완성도
고정 높이 + 반응형 : 대시보드 레이아웃 안정(카드 h-80 + maintainAspectRatio: false)
불변성·길이 일치 : 버그의 80%를 사전에 차단하는 핵심 습관

□ Day 7 — API 데이터 연동 + 차트 옵션 완전 정복 (Bar / Line / Doughnut / Pie / Radar)

□ 오늘의 학습 목표

useEffect + fetch로 **실제(또는 모의) API**에서 데이터를 가져와 차트에 바인딩한다.
로딩/에러/빈 상태를 안전하게 처리한다.
Chart.js 옵션을 차트별로 이해하고, **언제/어떻게** 쓰는지 안다.
실습으로 **막대/라인/도넛/파이/레이더** 차트를 각각 구성한다.

0□ 사전 확인 (필수)

npm i chart.js react-chartjs-2 설치됨
파일 상단에 import "chart.js/auto" (요소 자동 등록)
Tailwind 설정 완료(@tailwind base/components/utilities)

1□ API 연동 기본기 : useEffect + fetch

왜 이렇게 해야 하나?

네트워크는 **지연/실패**가 발생할 수 있어요 → 로딩/에러 처리가 **반드시** 필요
데이터를 받아오면 **state**에 저장 → React가 다시 렌더링 → 차트가 **자동 갱신**

□ 예제 — 사용자 수(월별) 가져와 막대 차트로 표시

파일 :src/pages/BarFromApi.jsx

```
import "chart.js/auto";
import { useEffect, useState } from "react";
import { Bar } from "react-chartjs-2";

export default function BarFromApi() {
  const [rows, setRows] = useState(null);
  const [err, setErr] = useState(null);

  useEffect(() =& {
    (async () =& {
      try {
        // 1) 실제 API가 준비되기 전까지는 mock API 사용 (jsonplaceholder 등)
        //    예: 월별 사용자 수를 흉내내기
        const res = await fetch("https://jsonplaceholder.typicode.com/users");
        if (!res.ok) throw new Error("네트워크 오류");
        const json = await res.json();

        // 2) 변환 (실무 포인트) : API 원본 → labels & data
```

```
        //      여기서는 사용자 10명을 1~10월로 매핑하는 예시
        const labels = Array.from({ length: 10 }, (_, i) =& `${i + 1}월`);
        const values = labels.map((_, i) =& (json[i] ? (i + 1) * 10 : 0));

        setRows({ labels, values });
      } catch (e) {
        setErr(e.message);
      }
    })();
  }, []);

  if (err) return &p className="p-6 text-red-600"&    에러 : {err}&/p&;
  if (!rows) return &p className="p-6"&    로딩 중 ...&/p&;

  const data = {
    labels: rows.labels,
    datasets: [
      {
        label: "월별 사용자 수(예시)",
        data: rows.values,
        borderWidth: 1,
        borderRadius: 8
      }
    ]
  };

  const options = {
    responsive: true,
    maintainAspectRatio: false,
    plugins: {
      title: { display: true, text: "API → Bar 차트" },
      legend: { display: false },
      tooltip: {
        callbacks: {
          label: (ctx) =& ` ${ctx.parsed.y.toLocaleString()} 명`
        }
      }
    },
    scales: {
      y: { beginAtZero: true, ticks: { callback: v =& `${v} 명` } }
    }
  };

  return (
    &div className="p-6"&
```

```
        &div className="h-80 rounded-2xl border border-gray-200 bg-white p-4 shadow-sm"&
          &Bar data={data} options={options} /&
        &/div&
      &/div&
    );
}
```

핵심 이유 정리
useEffect: 컴포넌트가 **처음 나타날 때** 한 번 호출 (API 요청 타이밍)
try/catch: 네트워크/파싱 오류 대비
변환 로직: “원본 → labels / data”로 **항상 분리해서 생각**하면 차트 연결이 쉬워짐
로딩/에러/성공 3단계 UI: 실무에서 **반드시** 구현

2️⃣ Chart.js 옵션 핵심 공통 파트

모든 차트에서 공통으로 자주 쓰는 옵션입니다.

```
const options = {
  responsive: true,
  maintainAspectRatio: false, // 카드 높이 주도권을 우리가 가짐 (Tailwind h-80 등)
  plugins: {
    title:  { display: true, text: "차트 제목" },
    legend: { display: true, position: "bottom" }, // bottom이 읽기 쉬운 경우 많음
    tooltip: {
      callbacks: {
        label: (ctx) =& {
          // 막대/라인은 ctx.parsed.y, 도넛/파이는 ctx.parsed
          const v = ctx.parsed.y ?? ctx.parsed;
          return ` ${v.toLocaleString()} 단위`;
        }
      }
    }
  },
  // scales는 막대/라인 등 '축이 있는' 차트에 사용
  scales: {
    x: { grid: { display: true } },
    y: { beginAtZero: true, ticks: { callback: v =& `${v}` } }
  }
};
```

3️⃣ 차트 유형별 옵션 · 사용처 · 팁

A) 막대(Bar) — 카테고리 간 절대값 비교의 왕도

언제? 반별 평균 점수, 제품군별 매출, 지역별 인원, 요일별 판매량
키 옵션
•
indexAxis: "y" → **가로 막대** (라벨 길거나 항목 많을 때)
barThickness, maxBarThickness → 막대 두께 조절
categoryPercentage, barPercentage → 막대 사이 간격
scales.y.beginAtZero: true → 공정한 비교
누적 : scales.x.stacked = true, scales.y.stacked = true
예제 : 가로 누적 막대

```
const options = {
  indexAxis: "y", // 가로 막대
  responsive: true,
  maintainAspectRatio: false,
  plugins: {
    title: { display: true, text: "카테고리별 집행(누적)" },
    legend: { position: "bottom" }
  },
  scales: {
    x: { stacked: true, ticks: { callback: v =& `${v} 만원` } },
    y: { stacked: true }
  }
};
```

B) 라인(Line) — 시간의 추세/변화

언제? 일별 방문자, 월별 매출, 온도 변화, 주가 등
키 옵션
•
tension (0~1) : 선 굴곡 (0 = 직선, 0.3~0.4 권장)
pointRadius : 점 크기 (데이터 많으면 0~2로 작게)
fill : 면 채우기 유무
spanGaps: true : null 구간 이어 그리기
보조축 : 서로 단위 다르면 yAxisID: 'y' | 'y1'
예제 : 방문자(명) + 전환율(%) 보조축

```
const data = {
  labels,
  datasets: [
    { type: "line", label: "방문자(명)", data: visits, tension: 0.3, yAxisID: "y"  },
    { type: "line", label: "전환율(%)", data: conv,    tension: 0.3, yAxisID: "y1" }
  ]
};
const options = {
  responsive: true,
  maintainAspectRatio: false,
```

```
  plugins: { legend: { position: "bottom" }, title: { display: true, text: "방문자 vs 전환율" } },
  scales: {
    y:  { beginAtZero: true, position: "left",  ticks: { callback: v =& `${v} 명` } },
    y1: { beginAtZero: true, position: "right", grid: { drawOnChartArea: false }, ticks: { callback: v =& `${v}%` } }
  }
};
```

C) 도넛(Doughnut) — 구성 비율 (중앙 빈 원)

언제? 지출 비율, 점유율, 카테고리 비중

키 옵션

- cutout : 중앙 구멍 비율 ("60%" 등)
- rotation / circumference : 시작 각도 / 표시 범위
- plugins.legend.position: "bottom" 권장 (조각 설명 읽기 쉬움)

주의 : 조각이 6개 이상이면 인지 부하 ↑ → TopN + “기타”

예제 : cutout + 합계 표시(별도 텍스트)

```
const options = {
  plugins: {
    legend: { position: "bottom" },
    title: { display: true, text: "지출 비율" },
    tooltip: {
      callbacks: {
        label: (ctx) =& {
          const total = ctx.dataset.data.reduce((a,b)=&a+b,0);
          const val = ctx.parsed;
          const pct = Math.round((val / total) * 100);
          return ` ${val.toLocaleString()} 원 · ${pct}%`;
        }
      }
    }
  },
  cutout: "60%"
};
```

D) 파이(Pie) — 구성 비율 (원형)

언제? 도넛과 동일하지만 **중앙이 꽉 찬 원**

키 옵션 (도넛과 거의 동일)

- rotation / circumference
- 조각 강조 : offset (특정 데이터만 튀어나오게)

권장 : 도넛과 용도는 같지만, 중앙에 합계가 필요하면 **도넛**이 더 표현력 좋음
예제 : 특정 조각 강조

```
const data = {
  labels: ["A","B","C","D"],
  datasets: [{
    data: [40, 30, 20, 10],
    offset: [8, 0, 0, 0] // 첫 조각만 8px 돌출
  }]
};
```

E) 레이더(Radar) — 다변수(능력치) 비교

언제? 다양한 지표를 **동시에** 비교 (예: 선수 능력치, 부서 역량, 제품 스펙)
키 옵션

•

scales.r : 레이더 전용 축(반경) 옵션
suggestedMin / suggestedMax : 값 범위 권장
angleLines.display, grid : 눈금선/격자
pointLabels : 꼭짓점 라벨 스타일
데이터셋 스타일 : fill, borderWidth, pointRadius
예제 : 능력치 차트

```
import { Radar } from "react-chartjs-2";

const data = {
  labels: ["속도","체력","지능","순발력","기술","팀워크"],
  datasets: [
    {
      label: "선수 A",
      data: [70, 80, 65, 88, 90, 75],
      fill: true,
      borderWidth: 2
    },
    {
      label: "선수 B",
      data: [60, 85, 72, 80, 85, 82],
      fill: true,
      borderWidth: 2
    }
  ]
};
const options = {
  responsive: true,
  maintainAspectRatio: false,
  plugins: {
```

```
      legend: { position: "bottom" },
      title: { display: true, text: "선수 능력치 비교" }
    },
    scales: {
      r: {
        suggestedMin: 0,
        suggestedMax: 100,
        angleLines: { display: true },
        grid: { circular: true },
        pointLabels: { font: { size: 12 } },
        ticks: { stepSize: 20, showLabelBackdrop: false }
      }
    }
};
```

---

4. 차트별 React + Tailwind 카드 샘플

아래는 각 차트를 **카드 컴포넌트**로 감싸는 공통 스타일 예시입니다.

&div className="h-80 rounded-2xl border border-gray-200 bg-white p-4 shadow-sm"&
  {/* &Bar .../& / &Line .../& / &Doughnut .../& / &Pie .../& / &Radar .../& */}
&/div&

이유 : 카드 높이를 통일(h-80), maintainAspectRatio: false와 함께 안정적인 대시보드 레이아웃 확보.

---

5. 실습 (API → 변환 → 차트)

실습 1 : **막대** — 상위 5개 지역 인구

임의 API(또는 CSV/JSON)을 불러와 { region, people } 변환
상위 5개만 정렬해서 표시
y축 단위 : 명, 툴팁/틱 포맷 적용
이유 : **정렬/슬라이스/단위 포맷** 연습

실습 2 : **라인** — 7일간 기온 추이

{ date, temp } 변환
tension: 0.3, pointRadius: 2, spanGaps: true
이유 : **시간 추세/포인트/결측치 처리** 연습

실습 3 : **도넛** — 카테고리 지출 비율

Top 4만 남기고 나머지는 “기타”로 합산
cutout: "60%", 아래 텍스트로 총합/최대 카테고리 표시
이유 : **TopN + 기타, 합계/비율 해석** 패턴 연습

실습 4 : **레이더** — 팀 역량 비교

{ skill, A, B } 형식 JSON을 가져와 labels, datasets[2]로 변환

suggestedMax: 100, ticks.stepSize: 20
이유 : **다변수 비교와 scales.r 옵션** 이해

---

6. 자주 하는 실수와 해결 (이유 포함)

증상	원인	왜 생기나	해결
차트가 안 뜸	chart.js/auto 누락	요소가 미등록	상단에 import "chart.js/auto"
값 있는데 빈 화면	labels.length !== data.length	x/y 수가 다름	변환 시 길이 맞추기(필수)
카드가 납작함	기본 비율 유지	캔버스가 가로폭에 종속	카드 h-80 고정 + maintainAspectR false
라벨 겹침	항목 과다	공간 부족	가로 막대, 상위 N개 + 기타, 2열 그리드
도넛이 복잡	조각 과다	인지 부하↑	Top4 + 기타
실시간 업데이트 버벅임	매 렌더마다 새 객체 생성	불필요 리렌더	데이터 가공 useMemo, 상태 불변성 유지

7. 한눈 정리 — 언제 무엇을 쓰나?

목적	추천 차트	핵심 옵션
카테고리 절대값 비교	**Bar**	indexAxis, stacked, barThickness
시간 추세	**Line**	tension, pointRadius, spanGaps, 보조축
구성 비율	**Doughnut / Pie**	cutout, rotation, offset
다변수 비교(능력치)	**Radar**	scales.r.suggestedMax, ticks.stepSize

Day 8 — Firebase 시작하기 : React + Firestore CRUD 기초

목표

Firebase 가 뭔지, 어떤 걸 할 수 있는지 감을 잡는다.
Firebase 콘솔에서 **프로젝트 만들기 → Firestore 켜기** 까지 직접 해 본다.
React 프로젝트에 **Firebase SDK 설치 + 초기화 파일 작성**을 한다.
Firestore 컬렉션 하나를 정해서 **CRUD (Create / Read / Update / Delete)** 를 구현해 본다.
Firebase 콘솔에서 프로젝트 만들기 → React 에서 SDK 연결 → Firestore CRUD 연습

1. Firebase 는 뭐 하는 서비스인가?

Google 이 제공하는 **백엔드 서버 대신 써먹는 서비스 묶음**
Firestore : 클라우드 NoSQL 데이터베이스
나중에 : Authentication (로그인), Storage (파일), Hosting (배포) 등도 가능
React 입장에서 보면 내가 서버 안 만들어도, URL 하나로 DB 를 읽고 쓰게 해 주는 서비스

2. Firebase 콘솔에서 프로젝트 만들기

2.1 콘솔 접속

브라우저에서 https://console.firebase.google.com 접속
구글 계정으로 로그인

2.2 새 프로젝트 만들기

프로젝트 추가 버튼 클릭
프로젝트 이름 입력 : 예 : react-chart-firebase-demo
Google Analytics
수업용이면 : **"나중에 설정" 또는 사용 안 함** 해도 됨
만들기 클릭 → 몇 초 기다리면 프로젝트 생성 완료

2.3 Firestore 데이터베이스 생성

왼쪽 메뉴 : **Build → Firestore Database** 클릭
데이터베이스 만들기 버튼
모드 선택
수업용 : **테스트 모드 시작** (일정 기간 동안 모든 읽기 / 쓰기 허용)
실무에서는 나중에 **보안 규칙** 꼭 수정 필요
위치는 기본 리전 선택해도 됨

완료하면 빈 Firestore 화면이 보임
지금은 아직 컬렉션 안 만들어도 됨. React 코드에서 처음 추가하면서 자동으로 생기게 할 거임

3. React 프로젝트에 Firebase SDK 연결

3.1 SDK 설치

프로젝트 루트에서
npm install firebase

3.2 Firebase 설정 값 복사

Firebase 콘솔 왼쪽 메뉴 : **프로젝트 개요** 화면
가운데 쯤에 **웹 앱 추가 (&/&)** 버튼 클릭 (앱 등록)
앱 이름 : react-app 등
Firebase Hosting 설정 같은 건 지금은 생략해도 됨
앱을 만들고 나면 아래와 같은 코드가 보일 거야

```
const firebaseConfig = {
  apiKey: "....",
  authDomain: "....firebaseapp.com",
  projectId: "....",
  storageBucket: "....appspot.com",
  messagingSenderId: "....",
  appId: "...."
};
```

이 firebaseConfig 객체를 복사해 둔다.

3.3 React 에서 초기화 파일 만들기

파일 경로 : src/firebase/config.js

```
// Firebase SDK 함수 가져오기
import { initializeApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";

// 콘솔에서 복사한 설정 값 붙여넣기
const firebaseConfig = {
  apiKey: "복붙",
  authDomain: "복붙",
  projectId: "복붙",
```

```
  storageBucket: "복붙",
  messagingSenderId: "복붙",
  appId: "복붙"
};

// 1) Firebase 앱 초기화
const app = initializeApp(firebaseConfig);

// 2) Firestore 인스턴스 만들기
export const db = getFirestore(app);
```

왜 이렇게 해야 하는지

initializeApp(firebaseConfig)

- 이 프로젝트에서 Firebase 를 사용할 준비를 하는 단계.
- 이걸 한 번은 해 줘야 나중에 Auth, Firestore 등을 쓸 수 있음.

getFirestore(app)

- 이 앱에서 **Firestore 데이터베이스** 객체를 가져옴.
- 앞으로 DB 관련 함수는 다 db 를 기준으로 동작.

export const db

- 다른 파일에서 import { db } from "../firebase/config"; 해서 Firestore 를 쓸 수 있도록 내보내기.

config.js 는 Firebase 와 연결해서 db 를 꺼내 쓰도록 하는 관문"파일이다.

4. Firestore CRUD 예제 구조 설계

간단 ToDo 앱 으로 연습
컬렉션 이름 : todos
문서 구조
text : 문자열 (할 일 내용)
done : 불린 (완료 여부)
createdAt : 생성 시각 (서버 타임스탬프)

파일 구성

```
src/
  firebase/
    config.js            // 방금 만든 Firebase 초기화
  components/
    TodoForm.jsx         // 할 일 입력 폼 (Create)
```

```
  TodoList.jsx          // 할 일 목록 + Update / Delete
pages/
  Day8TodoPage.jsx    // 오늘 수업용 화면
App.jsx
```

5. Firestore CRUD 에 쓸 기본 함수들

Firestore 관련 함수들은 여기에서 가져온다

```
import {
  collection,       // 컬렉션 참조 만들기
  addDoc,           // 문서 추가 (Create)
  getDocs,          // 문서 목록 가져오기 (Read - 1회)
  onSnapshot,       // 실시간으로 감시 (Read - 실시간)
  updateDoc,        // 문서 수정 (Update)
  deleteDoc,        // 문서 삭제 (Delete)
  doc,              // 문서 한 개를 가리키는 참조
  serverTimestamp // 서버 시간
} from "firebase/firestore";
```

6. Create + Read (실시간) 구현

6.1 TodoForm : 새 할 일 추가 (Create)

파일 : src/components/TodoForm.jsx

```
import { useState } from "react";
import { addDoc, collection, serverTimestamp } from "firebase/firestore";
import { db } from "../firebase/config";

export default function TodoForm() {
  const [text, setText] = useState("");
  const [loading, setLoading] = useState(false);

  async function onSubmit(e) {
    e.preventDefault();
    if (!text.trim()) return;
```

```
    setLoading(true);
    try {
      // 1) todos 컬렉션 참조 만들기
      const ref = collection(db, "todos");

      // 2) 새 문서 추가
      await addDoc(ref, {
        text: text.trim(),
        done: false,
        createdAt: serverTimestamp()
      });

      setText(""); // 입력창 비우기
    } catch (err) {
      console.error(err);
      alert("추가 중 오류가 발생했습니다.");
    } finally {
      setLoading(false);
    }
  }

  return (
    <form onSubmit={onSubmit} className="flex gap-2">
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        className="flex-1 rounded-lg border border-gray-300 px-3 py-2"
        placeholder="할 일을 입력하세요"
      />
      <button
        disabled={loading}
        className="rounded-lg bg-indigo-600 px-4 py-2 text-white
disabled:bg-gray-400"
      >
        {loading ? "추가 중..." : "추가"}
      </button>
    </form>
  );
```

```
}
```

코드 설명

collection(db, "todos")

- Firestore 안의 "todos" 라는 이름의 컬렉션을 가리키는 **참조 객체**
- 이 컬렉션이 없어도 addDoc 을 하면 자동으로 생성됨.

addDoc(ref, { ... })

- 해당 컬렉션에 새 문서를 추가하고, 자동으로 문서 ID 를 만들어 줌.

serverTimestamp()

- 내 컴퓨터 시간이 아니라 **Firebase 서버 시간을 기록**.
- 여러 사용자가 있을 때 기준 시간이 동일해져서 정렬 등에 유리.

6.2 TodoList : 실시간 읽기 (Read)

처음부터 getDocs 로 한 번만 가져와도 되지만,
Firebase 의 장점은 **실시간(onSnapshot)** 이라서 이걸 보여주는 게 좋음.
파일 : src/components/TodoList.jsx

```
import { useEffect, useState } from "react";
import {
  collection,
  onSnapshot,
  query,
  orderBy,
  updateDoc,
  deleteDoc,
  doc
} from "firebase/firestore";
import { db } from "../firebase/config";

export default function TodoList() {
  const [todos, setTodos] = useState([]);

  useEffect(() => {
    // 1) 컬렉션 + 정렬 기준 설정
    const ref = collection(db, "todos");
```

```
    const q = query(ref, orderBy("createdAt", "desc"));

    // 2) 실시간 구독 시작
    const unsub = onSnapshot(q, (snapshot) => {
      // snapshot.docs : 문서들 배열
      const list = snapshot.docs.map((d) => ({
        id: d.id,
        ...d.data()
      }));
      setTodos(list);
    });

    // 3) 컴포넌트 언마운트 시 구독 해제
    return () => unsub();
  }, []);

  // 완료 상태 토글 (Update)
  async function toggleDone(todo) {
    const ref = doc(db, "todos", todo.id);
    await updateDoc(ref, { done: !todo.done });
  }

  // 삭제 (Delete)
  async function removeTodo(id) {
    if (!window.confirm("삭제할까요?")) return;
    const ref = doc(db, "todos", id);
    await deleteDoc(ref);
  }

  if (!todos.length) {
    return <p className="mt-3 text-sm text-gray-500">아직 할 일이 없습니다.</p>;
  }

  return (
    <ul className="mt-3 space-y-2">
      {todos.map((todo) => (
        <li
          key={todo.id}
          className="flex items-center justify-between rounded-lg border
```

```
border-gray-200 px-3 py-2"
        >
          <button
            onClick={() => toggleDone(todo)}
            className="flex-1 text-left"
          >
            <span className={todo.done ? "text-gray-400 line-through" : ""}>
              {todo.text}
            </span>
          </button>
          <button
            onClick={() => removeTodo(todo.id)}
            className="ml-2 text-xs text-red-500 hover:underline"
          >
            삭제
          </button>
        </li>
      ))}
    </ul>
  );
}
```

코드 설명

query(ref, orderBy("createdAt", "desc"))

- 단순 컬렉션이 아니라 **정렬 조건이 붙은 쿼리 객체**
- createdAt 기준으로 최신순 정렬.

onSnapshot(q, callback)

- 이 쿼리를 만족하는 데이터에 **변화가 있을 때마다 콜백 실행**
- 누군가 새 할 일을 추가 / 수정 / 삭제하면 화면이 자동으로 갱신됨.

snapshot.docs.map(...)

- Firestore 의 DocumentSnapshot 을 **평범한 JS 객체 배열** 로 바꾸는 단계
- id 는 문서 ID, data() 는 문서의 실제 내용.

doc(db, "todos", todo.id)

- 특정 문서 한 개를 가리키는 참조 객체
- 이걸로 updateDoc, deleteDoc 수행.

7. 예제

파일 : src/pages/Day8TodoPage.jsx

```
import TodoForm from "../components/TodoForm";
import TodoList from "../components/TodoList";

export default function Day8TodoPage() {
  return (
    <div className="mx-auto max-w-xl p-6 space-y-4">
      <h1 className="text-2xl font-bold">Day 8 : Firebase + Firestore CRUD</h1>
      <p className="text-sm text-gray-600">
        Firebase 와 연결해서 할 일을 추가 · 조회 · 수정 · 삭제하는 연습입니다.
      </p>
      <TodoForm />
      <TodoList />
    </div>
  );
}
```

App 에 라우터로 연결

src/App.jsx 예시

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Day8TodoPage from "./pages/Day8TodoPage";

export default function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Day8TodoPage />} />
      </Routes>
    </Router>
  );
}
```

8. 정리 : 개념 + 코드 흐름

개념

Firebase 프로젝트와 웹 앱을 만든다.
React 에서 initializeApp → getFirestore 로 db 객체를 만든다.
컬렉션 : collection(db, "todos")
문서 : doc(db, "todos", "문서ID")
CRUD

- C : addDoc(collection(...), {...})
- R : onSnapshot(query(...), cb) 또는 getDocs(query(...))
- U : updateDoc(doc(...), { 필드수정 })
- D : deleteDoc(doc(...))

코드 흐름

- TodoForm : 입력 → addDoc 으로 새 문서 생성
- TodoList : onSnapshot 으로 실시간 목록 구독
  - 완료 토글 : updateDoc
  - 삭제 : deleteDoc

9. 실습

필드 추가

- priority (우선순위 : 낮음 / 보통 / 높음) 필드 추가
- 화면에서 select 박스로 선택해서 저장, 목록에 색상 다르게 표시.

정렬 변경

- 완료가 아닌 것 → 완료된 것 순으로 보이도록 정렬
- Firestore 에서는 조건 + 정렬이 같이 필요하다는 점 설명.

필터 버튼

- “전체 / 미완료 / 완료” 필터 버튼 만들기
- Firestore 쿼리 조건 바꾸거나, 가져온 배열을 JS 로 필터링.

createdAt 표시

- createdAt 을 화면에 한국 시간 형식으로 보여 주기
- todo.createdAt?.toDate().toLocaleString() 활용.

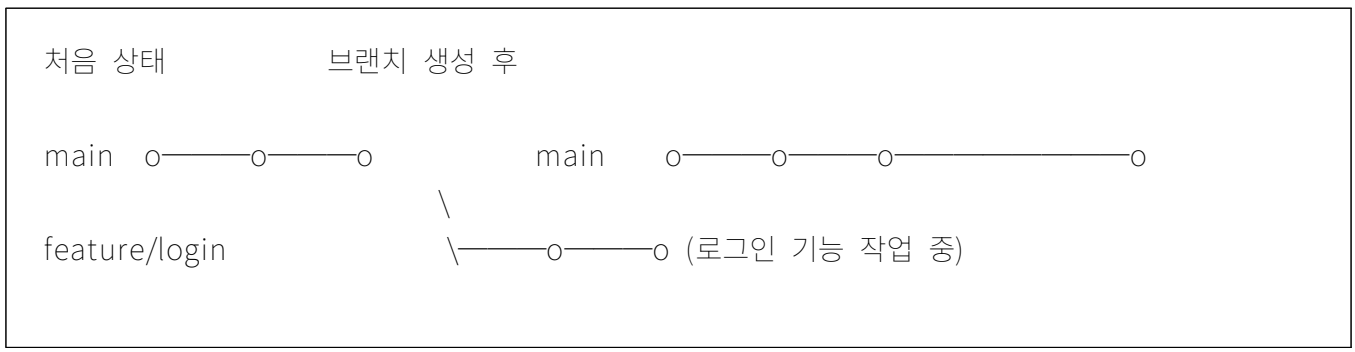
1. Git 브랜치 전략이란 무엇인가?

프로그래밍에서는 혼자만 쓰는 코드 가 아니라 여러 명이 함께 계속 수정하는 코드 가 대부분이다.
- 로그인 기능 만드는 사람
- 게시판 만드는 사람
- 차트 화면 만드는 사람
- 버그 고치는 사람

이 사람들이 모두 같은 프로젝트를 만지기 때문에 아무 생각 없이 같은 줄, 같은 파일을 덮어쓰면 난장판이 됩니다.
그래서 나오는 개념이 바로 브랜치(branch) 이다.
그리고 브랜치를 어떻게 나누고, 언제 합칠 것인가?에 대한 팀 규칙이 Git 브랜치 전략이다.

1) 브랜치란?
브랜치(branch) : 현재 코드에서 갈라져 나온 또 다른 작업선
쉽게 말하면, 지금까지의 코드 상태를 기준으로 새로운 작업을 하기 위한 복사본 길 을 하나 더 만든 다음
그 길 위에서 마음껏 실험하고, 잘 되면 원래 길(main) 로 합치는 구조이다.
Git 에서는 이 길(작업선)을 브랜치 라고 부르고, 각 브랜치는 서로 독립적으로 움직일 수 있다.

□ 핵심 개념
브랜치를 나누어서 일단 실험하고 잘 되면 main 에 합치고(merge) 문제가 생기면 브랜치만 버리면 되는 구조



2. 브랜치 전략이 필요한 이유

그냥 main 에서만 작업하면?
여러 명이 동시에 작업하면 서로 코드가 섞여서 충돌(conflict) 자주 발생
방금 올린 코드에 버그가 있으면 → 배포용 main 이 바로 망가짐
그래서 규칙을 정한다 어떤 작업은 어떤 브랜치에서 하고, 어떻게 합칠 것인가?→ 이게 바로 브랜치 전략

3. 대표적인 브랜치 전략 세 가지

1) Git Flow
브랜치가 크게 다섯 종류로 나뉜다.
main, develop, feature, release, hotfix
전통적인 방식이고 규모가 큰 회사에서 사용한다.
출시 주기가 길거나 여러 버전을 동시에 관리해야 할 때 유용하다.
그러나 브랜치가 너무 많아 학습용이나 단기 프로젝트에는 복잡한 편이다.

2) GitHub Flow
브랜치가 간단하다.

main 과 feature/* 중심으로 운영된다.
main 은 항상 실행 가능한 상태로 유지한다.
새 기능을 만들 때는 feature 브랜치를 만들고 작업한 뒤에
PR 을 올려 코드 리뷰를 받고 main 에 병합한다.
웹 개발, 프론트엔드 개발에서 가장 널리 사용되는 방식이다.
규칙이 간단하고 팀원들이 따라오기 쉽다.

3) Trunk-based Development
main 중심 개발 방식이다.
feature 브랜치를 만들더라도 짧게 유지하고 짧은 시간 안에 main 에 자주 병합한다.
작업 단위를 잘게 나누고 CI·CD 환경이 갖춰진 팀에서 효과적이다.
숙련된 개발자들에게 적합하며 초보자에게는 조금 어려울 수 있다.

4) 팀(4~5) 프로젝트 권장 브랜치 전략
학습과 실습에 적합하도록 단순하고 실수 위험이 적은 전략을 사용한다.
전체적인 형태는 GitHub Flow 를 기반으로 한다.
브랜치 구성
- main : 최종 실행 가능한 버전
- feat/* : 기능 개발용
- fix/* : 버그 수정용
- hotfix/* : 발표 직전 급한 오류 수정

이 전략의 장점은 다음과 같다.
브랜치 구조가 단순해서 프로젝트 관리가 쉽다.
팀원 4~5명이 동시에 작업해도 충돌 관리가 어렵지 않다.
협업 경험을 배우기 좋고 실무에서도 바로 사용하기 좋다.

4. 브랜치 전략

4-1. 기본 규칙

1. main 에 직접 commit 하지 않는다
2. main 은 배포용, 시연용 코드만 두는 브랜치이다.
3. 개발자는 항상 별도 브랜치를 만들어서 작업한다.
4. main 을 지키는 것이 팀 전체의 목표이다.
5. 모든 작업은 반드시 브랜치를 따서 한다

작업 기본 흐름은 다음과 같다.
- 1단계 main 에서 최신 코드 가져오기
- 2단계 새 브랜치 생성
- 3단계 브랜치에서 코드 작업 후 여러 번 commit
- 4단계 자신의 브랜치에서 테스트
- 5단계 main 으로 merge
- 6단계 원격 저장소에 push

브랜치 이름은 의미 있게 짓는다
브랜치를 보는 것만으로 무엇을 하는지 알 수 있어야 한다.

구분	설명	예시
새로운 기능	새로운 화면 또는 기능 개발	feat/login-formfeat/chart-page
버그 수정	기존 기능의 오류 수정	fix/header-layoutfix/chart-tooltip
긴급 수정	배포 중 발생한 치명적 오류 수정	hotfix/prod-crash

이 규칙을 팀 규칙으로 정하고, 의미 없는 이름 test, temp, aaa 은 사용하지 않는다.

4-2. 기본 작업 흐름 예시

여기서는 GitHub 에 프로젝트가 있다고 가정하고,
팀원이 기능 하나를 개발할 때의 전체 흐름을 정리한다.
프로젝트 처음 가져올 때

명령어	설명
git clone 주소	원격 저장소 전체를 내 컴퓨터로 복사한다
cd 프로젝트폴더	프로젝트 폴더로 이동한다
git branch	현재 브랜치를 확인한다 (별표가 있는 브랜치가 현재 브랜치)

처음에는 보통 main 하나만 보인다.

새 기능 작업

단계	명령어 또는 작업 내용	설명
1	git checkout main git pull origin main	main 최신 코드로 맞춘다
2	git checkout -b feat/login-page	새 브랜치를 만들고 그 브랜치로 이동한다
3	파일 수정	로그인 관련 화면을 개발한다
4	git statusgit diff	변경된 파일과 수정 내용을 확인한다
5	git add .git commit -m "feat : 로그인 페이지 기본 UI 구현"	작업 단위대로 commit 한다
6	git push -u origin feat/login-page	브랜치를 원격 저장소에 올린다
7	GitHub PR 생성 및 코드 리뷰	검토 후 main 에 merge 한다

(1) main 최신 코드 받기
git checkout main
작업 위치를 main 브랜치로 이동한다.
다른 브랜치에서 작업 중이었다면, main 으로 되돌아온다.
git pull origin main
원격 저장소 origin 의 main 브랜치 내용을 가져와 내 로컬 main 과 합친다.

누군가가 먼저 코드를 올린 경우, 이 과정으로 최신 상태를 맞춘다.
항상 새 브랜치를 만들기 전에 한 번은 git pull origin main 을 실행하는 습관을 들인다.

(2) 새 브랜치 생성과 이동
git checkout -b 브랜치이름
새로운 브랜치를 만들고, 그 브랜치로 바로 이동한다.
feat/login-page
로그인 페이지 기능을 개발한다는 의미를 가진 브랜치 이름이다.
이제 이후 모든 작업, 파일 수정, 커밋은 이 feat/login-page 브랜치 위에서만 이루어진다.

(3) 변경 내역 확인
git status
어떤 파일이 변경되었는지, 아직 staging 되지 않은 파일이 무엇인지 보여준다.
git diff
변경된 내용의 차이를 줄 단위로 보여준다.
커밋하기 전에 꼭 확인하는 습관이 좋다.

(4) 커밋
git add
현재 폴더 기준으로 변경된 모든 파일을 커밋 준비 상태로 올린다.
git commit -m 메시지
변경 내용을 하나의 단위로 저장한다.
예시 메시지는 feat : 로그인 페이지 기본 UI 구현이다.
커밋은 작업 단위가 어느 정도 의미 있을 때마다 여러 번 나누어 하는 것이 좋다.

(5) 원격 브랜치로 올리기
git push origin 브랜치이름
로컬 브랜치의 커밋을 원격 저장소의 같은 이름 브랜치로 올린다.
옵션 u
이 브랜치를 앞으로 기본 push 대상과 연결한다.
이후에는 단순히 git push 만 입력해도 된다.
이 단계까지 완료되면, 내 컴퓨터의 feat/login-page와 GitHub 의 feat/login-page 가 동일한 내용이 된다.
Pull Request 와 코드 리뷰 GitHub 를 사용할 경우
1단계 GitHub 에서 feat/login-page 브랜치 기준으로 Pull Request 생성
2단계 변경된 파일과 내용을 설명으로 작성
3단계 팀원이 코드 리뷰
4단계 이상 없으면 main 에 merge

4-3. merge 흐름

단계	명령어 또는 작업 내용	설명
1	git checkout main	main 브랜치로 이동한다
2	git pull origin main	최신 main 코드로 맞춘다
3	git merge feat/login-page	해당 기능 브랜치를 main 에 병합한다

4	git push origin main	병합 내용 반영
5	git branch -d feat/login-page	로컬 브랜치 삭제
6	git push origin --delete feat/login-page	원격 브랜치 삭제

main 으로 돌아가기
현재 작업 위치를 main 브랜치로 이동한다.
merge 작업은 항상 main 에서 진행한다.

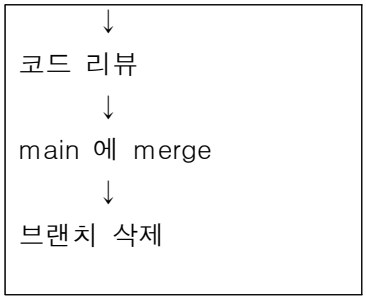
main 최신 코드 받기
원격 main 에 다른 사람이 먼저 올린 변경사항이 있을 수 있으므로
항상 병합 전에 최신 상태로 맞춘다.

브랜치 병합
git merge 브랜치이름
해당 브랜치에서 작업한 커밋들을 현재 브랜치에 합친다.
현재 브랜치가 main 이므로 feat/login-page 의 내용이 main 으로 들어간다.
충돌이 없다면 자동으로 병합이 완료된다.
충돌이 있다면 파일을 열어 수정한 뒤 다시 git add git commit 으로 병합을 마무리한다.

병합된 main 을 원격으로 푸시 로컬 main 에 병합된 변경사항을 원격 main 에도 반영한다.
이 작업 후에는 팀원이 다시 pull 해서 최신 main 을 받는다.
사용이 끝난 브랜치 삭제
-d 옵션
해당 브랜치가 이미 main 에 안전하게 병합된 경우에만 삭제한다.
브랜치를 삭제해도 main 에 병합된 내용은 그대로 남는다.
오래된 브랜치를 정리하면 브랜치 목록이 깔끔해져서 관리가 쉬워진다.
원격 브랜치도 함께 정리하고 싶으면 git push origin —delete feat/login-page 를 사용한다.

▶ main 에서는 작업하지 않는다.
▶ 언제나 main 에서 최신 코드부터 받는다.
▶ 기능마다 feat 브랜치를 만들어서 작업한다.
▶ 작업이 끝나면 반드시 main 으로 merge 한 뒤 push 한다.
▶ 병합이 끝난 브랜치는 삭제해서 목록을 정리한다.

```
main 최신 코드 pull
        ↓
새 브랜치 생성
        ↓
브랜치에서 개발
        ↓
commit
        ↓
push
        ↓
Pull Request 생성
```



5. 브랜치 전략에서 주의해야 할 사항

5-1. 브랜치를 너무 오래 끌지 말 것

2주, 3주 동안 feat/mega-feature 같은 브랜치에서만 작업하면 그 사이에 main 은 계속 바뀌고 나중에 merge 할 때 충돌지옥

큰 기능도 작은 단위로 나누어 feat/login-ui, feat/login-api 등으로 쪼개기
자주 main 의 변경사항을 받아와서 conflict 를 자주, 조금씩 해결
내 브랜치에 main 변경 사항 가져오기
git checkout feat/login-page
git pull origin main # 또는 git merge origin/main

5-2. main 에 직접 commit / push 하지 말 것

실수로 main 에서 작업하다가 실험 코드, 콘솔로그, 깨지는 코드가 바로 배포용에 들어감

main 은 직접 push 금지, PR 통해서만 merge
GitHub 보호 브랜치 설정 (실무)

5-3. 브랜치 이름을 의미 있게

나쁜 예 : test, abc, branch2
좋은 예 : feat/user-profile-edit, fix/dashboard-scroll
PR 목록만 봐도 어떤 기능이 어디서 작업 중인지 팀 전체가 한눈에 알 수 있음.

6. 자주 발생하는 오류 / 문제와 대책

가. main 에서 작업해 버린 경우

증상

원래는 feat 브랜치에서 작업해야 하는데
커밋을 몇 개 하고 나서 보니 현재 브랜치가 main 이다.
또는 깃 로그를 보니 main 에 실험 코드, 콘솔 출력, 미완성 코드가 섞여 있다.

원인

브랜치를 만들기 전에 바로 작업을 시작했다.
git branch, git status 로 현재 브랜치를 확인하지 않았다.

해결 방법

상황에 따라 두 가지로 나눈다.

a) 아직 원격에 push 하지 않았을 때

이 경우가 가장 안전하다.

1단계 현 상태에서 새 브랜치를 만든다

git branch feat/옮길-작업

2단계 새 브랜치로 이동한다

git checkout feat/옮길-작업

이렇게 하면 현재까지의 커밋들이 feat/옮길-작업 브랜치에도 그대로 존재한다.

main 은 아직 그 커밋들을 가리키고 있기 때문에 여기서 main 을 이전 커밋으로 되돌릴지 말지는 상황에 따라 결정한다.

처음부터 main 에 커밋이 없었다면 reset 이 필요할 수 있다.

3단계 앞으로는 main 에서는 작업하지 않고 항상 새 브랜치를 만들어서 작업한다

b) 이미 main 을 원격에 push 한 경우

이 경우에는 프로젝트 상황을 보고 결정한다.

ㄱ) 팀원과 교사에게 알린다

방금 main 에 잘못 올린 커밋이 있다는 점을 공유한다.

ㄴ) 커밋 내용이 괜찮고 빌드가 깨지지 않는다면

그대로 두고 이후부터 브랜치 전략을 잘 지키도록 한다.

ㄷ) 커밋 내용이 심각한 경우 예 빌드 완전 실패

교사 또는 경험 있는 사람이 reset, revert 로 복구한다.

이 부분은 잘못 다루면 더 큰 문제가 생길 수 있으므로 초보자가 혼자 처리하지 않도록 한다.

나. push 가 거절되는 경우 non fast forward

증상

git push origin main 실행 시

rejected

non fast forward

같은 메시지가 출력된다.

원인

원격 main 에 다른 사람이 먼저 커밋을 올렸는데 내 로컬 main 에는 그 커밋이 없는 상태에서 push 하려고 해서 생기는 문제이다.

즉원격 main 이 내 main 보다 더 앞서 있다.

해결 방법

항상 순서는 pull 먼저, push 나중이다.

1단계 main 으로 이동

git checkout main

2단계 원격 main 의 최신 내용을 가져와 합친다

git pull origin main

이 과정에서 자동 병합이 되거나 충돌이 발생할 수 있다.

충돌이 나면 관련 파일을 열어 직접 수정한 다음
git add
git commit
으로 병합을 마무리한다.
3단계 다시 push
git push origin main
앞으로 새 기능 개발 전에
git checkout main
git pull origin main
을 먼저 실행하는 습관을 들이면 이 오류가 많이 줄어든다.

다. merge conflict 충돌 이 발생하는 경우

증상

git merge 실행 후
충돌 표시가 있는 파일 목록이 뜬다.
파일을 열어보면 아래와 같은 표시가 보인다.
<<<<<<< HEAD
내 코드
=======
다른 브랜치 코드
>>>>>>> 브랜치이름
또는 commit 이 안 되고
충돌을 먼저 해결하라는 메시지가 나온다.

원인

같은 파일, 같은 위치를 두 브랜치에서 서로 다르게 수정했다.
Git 이 어느 쪽을 선택해야 할지 자동으로 결정할 수 없어서 개발자에게 직접 선택을 요구하는 상황이다.
보통 브랜치를 오랫동안 분리해서 작업하거나 역할 분담을 제대로 나누지 못했을 때 자주 생긴다.

해결 방법

1단계 충돌 난 파일 열기
git status 로 어떤 파일에 충돌이 났는지 확인하고 해당 파일을 에디터로 연다.
2단계 충돌 구간을 눈으로 확인한다
<<<<<<< HEAD
현재 브랜치 내용
=======
병합하려는 브랜치 내용
>>>>>>> 브랜치이름
위 구조에서
HEAD 부분은 현재 브랜치 코드
아래 부분은 병합 대상 브랜치 코드이다.
3단계 둘 중 하나를 선택하거나, 두 코드를 합쳐서 수정한다
현재 코드를 살리고 싶으면 HEAD 쪽 내용을 남기고 나머지 충돌 표시들을 삭제한다.
둘 다 필요한 경우 두 코드 내용을 적절히 섞어서 새로 작성한 뒤 충돌 표시 기호들을 모두 삭제한다.
4단계 수정 후 저장하고 add

git add 수정된파일들
5단계 병합 완료 커밋
git commit
이제 병합이 완료된다.
6단계 앞으로의 예방책
브랜치를 너무 오래 끌지 않는다.
같은 파일을 동시에 여러 명이 수정하지 않도록 역할을 페이지 단위, 기능 단위로 나눈다.

라. 잘못된 브랜치에서 작업한 경우

증상

원래는 feat/login 브랜치에서 작업해야 하는데 작업 후 확인해 보니 feat/chart 같은 다른 브랜치였다.
또는 브랜치 이름과 실제 작업 내용이 전혀 다르다.

원인

브랜치 전환을 제대로 확인하지 않았다.
작업 시작 전 git branch 혹은 git status 를 보지 않았다.

해결 방법

a) 아직 push 하지 않은 경우
현재 브랜치에서 작업한 커밋들을 새 브랜치로 옮기는 방법이다.
1단계 현재 브랜치에서 새 브랜치 생성
git branch feat/올바른-이름
2단계 새 브랜치로 이동
git checkout feat/올바른-이름
이제 앞으로는
feat/올바른-이름 브랜치에서 계속 작업하면 된다.
원래 브랜치는 필요에 따라 그대로 두거나
정리할 수도 있다.

b) 이미 원격에 push 한 경우
1 팀원과 상의한다.
2 현재 브랜치 이름을 그대로 유지하고
PR 제목과 설명에 실제 작업 내용을 명확히 적는 방법이 있다.
3 또는 선생님과 상의하여
이후부터는 규칙대로 새 브랜치를 만들어 사용한다.
교육용 프로젝트에서는 굳이 복잡한 history 수정보다 규칙을 다시 정리하고 앞으로 잘 지키는 방향을 추천한다.

마. 작업 중인 변경 사항 때문에 브랜치 이동이 안 되는 경우

증상

git checkout 다른브랜치를 실행했더니 변경 사항이 있어서 이동할 수 없다는 메시지가 나온다.

원인

현재 브랜치에서 수정 중인 파일이 있는데 아직 commit 이나 stash 를 하지 않았다.
다른 브랜치로 이동하면 그 수정 사항이 섞이거나 잃어버릴 수 있기 때문에 Git 이 막은 것이다.

해결 방법

세 가지 방법이 있다.
1 지금 한 작업을 유지하면서 이동하고 싶다 commit 사용
2 잠깐만 숨겨두고 나중에 다시 꺼내고 싶다 stash 사용
3 수정 내용이 필요 없다 되돌리기

바. 브랜치를 삭제했는데 다시 필요해진 경우

증상

git branch -d 브랜치
브랜치를 지웠는데 나중에 그 브랜치의 작업 내용이 필요해졌다.

원인

병합이 끝난 브랜치를 정리하는 과정에서필요한 내용을 아직 다른 곳에 백업하지 않았다.

해결 방법

브랜치를 삭제해도 그 브랜치의 커밋 자체는 바로 사라지지 않는다.
다만 접근하기 어려울 뿐이다.
조금 더 깊은 Git 명령어인 reflog 를 사용하면 최근에 이동했던 커밋 위치들을 확인할 수 있다.

1단계 최근 커밋 위치 기록 확인
    git reflog
2단계 되돌아가고 싶은 커밋 해시를 찾는다
    reflog 출력에서
HEAD 이동 기록과 커밋 해시를 확인한다.
3단계 해당 커밋에서 새 브랜치 만들기
    git checkout -b 복구브랜치 커밋해시
이렇게 하면 그 시점의 내용을 복구할 수 있다.
이 명령은 잘못 사용하면 더 복잡해질 수 있다.

팀에서 사용할 브랜치 종류 가급적 간단한 전략으로 가는 것이 좋음

기본 브랜치

main
배포용, 발표용, 시연용
항상 빌드가 통과되고 실행 가능한 코드만 존재
절대 직접 작업하지 않고, 다른 브랜치를 merge 해서만 변경

feat/*
새 기능 개발용 브랜치
예 : feat/login-page, feat/chart-dashboard, feat/firebase-auth

fix/*
버그 수정 브랜치
예 : fix/navbar-layout, fix/chart-legend-error

hotfix/*
정말 급한 버그 수정 (배포 후 심각한 문제)
예 : hotfix/login-crash

브랜치 이름 규칙

팀 전체가 이해하기 쉬운 규칙이 중요
기능 개발 : feat/기능-설명
예 : feat/user-signup, feat/todo-crud, feat/chart-monthly-expense
버그 수정 : fix/버그-내용
예 : fix/footer-overflow, fix/login-validation
긴급 수정 : hotfix/문제-내용
예 : hotfix/prod-build-fail

의미 없는 이름 사용 금지 : test, temp, aa 같은 브랜치는 쓰지 않는다.
팀 규칙으로 브랜치 이름만 봐도 무슨 일을 하는지 알 수 있게 만들기

하루 작업 루틴 (아침에 할 일)

모든 팀원이 아침에 공통으로 해야 하는 절차 - 새브랜치 만들꺼면
1. main 으로 이동
    git checkout main
2. 원격 main 최신 코드 가져오기
    git pull origin main
이렇게 해야 어제 다른 팀원이 merge 한 내용을 모두 가져온 상태에서 새 브랜치를 만들 수 있다.

새 기능 맡을 때의 흐름

팀원 A : 로그인 페이지
팀원 B : 회원가입 페이지

팀원 A (로그인 페이지 담당)

1. main 에서 브랜치 생성
   git checkout main
   git pull origin main
   git checkout -b feat/login-page
2. 코드 작업 (화면, 로직 등)
3. 변경 확인
   git status
   git diff
4. 커밋
   git add .
   git commit -m "feat : 로그인 페이지 UI 및 기본 검증 추가"
5. 원격 브랜치 푸시
   git push -u origin feat/login-page

팀원 B (회원가입 페이지 담당)

   git checkout main
   git pull origin main
   git checkout -b feat/signup-page

작업 후

   git add .
   git commit -m "feat : 회원가입 페이지 UI 및 폼 검증"
   git push -u origin feat/signup-page

merge 와 PR 흐름

1) 작업 완료 후 팀장/리뷰 요청
   각자 GitHub 에서 Pull Request(PR) 생성
   base : main
   compare : feat/login-page 같은 브랜치

2) 코드 리뷰
   팀원 1~2명이 코드 확인
   변수명 의미 있는지
   콘솔로그 그대로 남아 있는지
   중복 코드 있는지
   문제가 있으면 PR에 코멘트
   수정 후 커밋을 PR 브랜치에 다시 push

3) merge
   팀장 또는 담당 리뷰어가 OK 하면
   GitHub 에서 merge 클릭 → main 에 합쳐짐

4) 브랜치 정리
merge 가 끝난 feat/login-page 는 정리
로컬에서 브랜치 삭제

git branch -d feat/login-page

원격에서 브랜치 삭제
git push origin --delete feat/login-page
merge 된 브랜치는 반드시 삭제해서 목록을 깔끔하게 유지한다.

팀에서의 역할 분담 예시

역할	예시 담당	브랜치 예시
팀장 / 리드	전체 구조, 라우터, 배포	feat/app-layoutfeat/router-setup
프론트 1	메인 대시보드, 차트 화면	feat/dashboard-pagefeat/chart-income
프론트 2	로그인 / 회원가입 / 마이페이지	feat/login-pagefeat/signup-pagefeat/profile
데이터 / 백엔드 연동	API 호출, Firebase 연동	feat/firebase-authfeat/fb-firestore-todos
공통 UI	공통 버튼, 카드, 테이블 컴포넌트	feat/ui-buttonfeat/ui-table

같은 파일을 동시에 만지지 않도록 페이지 단위 / 기능 단위 역할 분리 를 미리 정해두면 충돌이 확 줄어듭니다.