15. CSS 모듈화 및 구조화

BEM 방식(Block Element Modifier)

BEM은 **CSS 클래스 네이밍 규칙** 중 하나로,
재사용성, **가독성**, **유지보수성**을 높이기 위해 고안된 방식이다.

BEM은 다음과 같은 구조를 따른다:

```
Block__Element--Modifier
```

---

1. BEM이란?

구성 요소	의미	예시
Block	독립적인 기능 단위	`menu`, `button`, `card`
Element	블록의 구성 요소	`menu__item`, `card__title`
Modifier	상태나 변형	`button--large`, `menu__item--selected`

---

2. 예제

HTML

```
<div class="card card--featured">
  <h2 class="card__title">제목</h2>
  <p class="card__description">내용 설명</p>
</div>
```

CSS

```
.card {
  border: 1px solid #ccc;
  padding: 1rem;
}

.card--featured {
  background-color: #f0f0f0;
}

.card__title {
  font-weight: bold;
}

.card__description {
  color: #666;
```

```
}
```

3. 구조 규칙 요약

구분	표기법	예시
Block	`.block`	`.header`, `.form`
Element	`.block__element`	`.form__input`, `.nav__link`
Modifier	`.block--modifier` or `.block__element--modifier`	`.button--disabled`, `.card__title--highlighted`

4. 네이밍 예시

```
/* Block */
.button {}

/* Block with modifier */
.button--primary {}

/* Element */
.button__icon {}

/* Element with modifier */
.button__icon--small {}
```

5. BEM의 장점

항목	설명
명확성	클래스만 보고 어떤 역할인지 쉽게 알 수 있음
중첩 회피	클래스만으로 스타일 지정 → CSS 구조 단순화
재사용성	컴포넌트 단위 개발에 최적
확장성	모듈 방식의 개발에 적합 (React, Vue 등)

6. 주의 사항

- `__`, `--`는 각각 element와 modifier를 나타냄
- 무조건 이 표기법을 강요하는 건 아님 (선택적 도입 가능)
- 너무 길어질 수 있으므로 간결한 이름 선택 필요

7. VS 다른 방식

방식	특징
BEM	구조적 명확성, 클래스 기반
OOCSS	재사용성과 성능 중심
SMACSS	범주(category) 기반 구조화
Tailwind	유틸리티 클래스 기반, BEM 불필요

---

8. 실제 프로젝트에서 적용

프론트엔드 프레임워크와 함께 쓸 경우에도 적합:

```
<div class="profile-card profile-card--premium">
  <div class="profile-card__avatar"></div>
  <div class="profile-card__name">홍길동</div>
</div>
```

→ 모든 스타일이 **의미론적이고 컴포넌트 단위로 분리됨**

결론

- BEM은 클래스 네이밍을 구조화하여, **스타일의 명확성, 유지보수성, 재사용성**을 향상시킨다.
- 대규모 프로젝트나 팀 개발에서 **CSS 충돌 방지와 협업 효율**을 높이는 데 매우 유리하다.
- Vue, React와 같은 컴포넌트 기반 아키텍처와도 잘 어울린다.

OOCSS, SMACSS, Atomic CSS

CSS 설계 방식에는 여러 가지 철학과 전략이 있다. 그중에서도 **OOCSS**, **SMACSS**, **Atomic CSS**는 대표적인 CSS 아키텍처 설계 방식이다. 각각의 방식은 목적과 구현 방식이 다르며, 프로젝트 규모나 팀의 스타일에 따라 적합한 전략이 달라질 수 있다.

---

1. OOCSS (Object-Oriented CSS)

개요

- **"객체 지향" 개념을 CSS에 적용**한 설계 방식
- **구조(Structure)**와 **스킨(Skin)**을 분리
- **재사용 가능한 모듈**을 목표로 함

핵심 원칙

원칙	설명
구조와 스킨의 분리	레이아웃 구조와 시각적 표현을 분리
재사용 가능한 객체	공통 스타일을 묶어 클래스로 재사용

예시

```
/* 구조 */
.media {
  display: flex;
  align-items: flex-start;
}

/* 스킨 */
.avatar {
  width: 50px;
  height: 50px;
  border-radius: 50%;
}
```

```
<div class="media">
  <img class="avatar" src="profile.jpg">
  <div class="media-body">...</div>
</div>
```

장점

- 중복 제거
- 모듈화된 구조로 유지보수 쉬움

단점

- 시멘틱 HTML을 희생할 수 있음
- 너무 많은 클래스가 생길 수 있음

---

2. SMACSS (Scalable and Modular Architecture for CSS)

개요

- CSS를 **범주(category)**에 따라 분류
- 점진적인 채택이 가능함 (부분 적용 가능)
- **유지보수성과 확장성**에 중점

다섯 가지 스타일 범주

범주	설명
Base	HTML 요소에 대한 기본 스타일
Layout	레이아웃을 위한 구조적 요소
Module	재사용 가능한 UI 구성요소
State	요소의 상태 (활성, 비활성 등)
Theme	색상이나 스킨 관련 스타일

예시 구조

```
/* Base */
body {
  font-family: sans-serif;
}

/* Layout */
.l-header {
  display: flex;
  justify-content: space-between;
}

/* Module */
.card {
  border: 1px solid #ddd;
  padding: 1rem;
}

/* State */
.is-active {
  display: block;
}
```

장점

- 범주화된 구조 → 유지보수 용이
- 팀 협업 시 가독성 좋음

단점

- 설계 기준이 사람마다 달라질 수 있음
- 일관성 유지가 필요함

---

3. Atomic CSS

개요

- **하나의 클래스가 하나의 역할만 수행**하는 스타일링 방식
- 매우 작은 유틸리티 클래스들을 조합하여 UI 구성
- Tailwind CSS와 유사한 접근 방식

예시

```
<div class="w-1/2 p-4 text-center bg-gray-200">Hello</div>
```

→ 각각 `width`, `padding`, `text-align`, `background`를 담당하는 유틸리티 클래스

특징

항목	설명
유틸리티 기반	재사용 가능한 단일 책임 클래스
HTML 중심	CSS는 거의 작성하지 않고 HTML 클래스 조합에 집중
빠른 개발	반복 스타일 작성 최소화

장점

- 빠른 프로토타이핑
- 중복 제거 및 성능 최적화 가능
- CSS 파일 사이즈 최소화 (Purging 가능)

단점

- HTML이 난잡해질 수 있음
- 시멘틱 마크업과 충돌 가능
- 스타일 일관성 유지를 위한 전략 필요

---

4. 비교 요약

항목	OOCSS	SMACSS	Atomic CSS
기반 철학	객체 지향	범주화	유틸리티 클래스
클래스 수	중간	중간~많음	매우 많음
재사용성	높음	높음	극단적
HTML 가독성	보통	좋음	낮음
CSS 관리	구조화 필요	범주 기반 구조	최소

항목	OOCSS	SMACSS	Atomic CSS
대표 프레임워크	없음	없음	Tailwind CSS

결론

상황	추천 방식
대규모 팀 협업, 명확한 구조 필요	SMACSS
반복되는 모듈 많고 재사용 중요	OOCSS
빠른 UI 제작, 퍼포먼스 중시	Atomic CSS (Tailwind 등)
컴포넌트 기반 SPA (React/Vue)	BEM 또는 Atomic CSS 혼합

SCSS, SASS와 같은 CSS 전처리기

CSS 전처리기는 **CSS의 한계를 보완**하기 위해 고안된 확장 언어이다.
변수, 중첩, 연산, 믹스인, 반복문 등의 프로그래밍 기능을 제공하며,
최종적으로 순수 CSS로 **컴파일**되어 웹 브라우저에 적용된다.

대표적인 전처리기:

- **Sass (Syntactically Awesome Stylesheets)**
  - `.scss` 문법 (CSS와 유사, 가장 널리 사용됨)
  - `.sass` 문법 (들여쓰기 기반, 세미콜론과 중괄호 없음)
- Less
- Stylus

---

1. Sass & SCSS의 차이

항목	SCSS	SASS
파일 확장자	`.scss`	`.sass`
문법	CSS와 거의 동일	들여쓰기 기반
세미콜론 `;`	사용	생략
중괄호 `{}`	사용	생략

예시

SCSS:

```
$main-color: #333;

.navbar {
  background: $main-color;
  ul {
    margin: 0;
    li {
      display: inline-block;
    }
  }
}
```

SASS:

```
$main-color: #333

.navbar
  background: $main-color
  ul
    margin: 0
    li
      display: inline-block
```

---

2. 핵심 기능

2.1 변수

```
$primary-color: #3498db;

body {
  color: $primary-color;
}
```

2.2 중첩 (Nesting)

```
.card {
  padding: 1rem;

  &__title {
    font-weight: bold;
  }

  &:hover {
    background-color: #f0f0f0;
  }
}
```

2.3 믹스인 (`@mixin`, `@include`)

```
@mixin flex-center {
  display: flex;
  justify-content: center;
  align-items: center;
}

.container {
  @include flex-center;
}
```

2.4 상속 (`@extend`)

```
.button {
  padding: 0.5rem;
  border: none;
}

.button--primary {
  @extend .button;
  background-color: blue;
}
```

2.5 연산

```
.container {
  width: 100% - 2rem;
}
```

2.6 조건문 및 반복문

```
@for $i from 1 through 3 {
  .col-#{$i} {
    width: 100% / 3 * $i;
  }
}
```

---

3. SCSS의 장점

항목	설명
가독성 향상	중첩 구조, 변수명 등으로 코드 구조 명확
재사용성 증가	믹스인, 상속, 반복문 등을 통한 중복 제거
유지보수성 향상	테마 변경, 구조 리팩토링이 용이
모듈화	`@import` 또는 `@use`로 분할 관리 가능

4. SCSS vs CSS

항목	CSS	SCSS
변수 지원	❌	✅
중첩 지원	❌	✅
조건/반복	❌	✅
모듈화	일부 지원 (@import 제한적)	✅
복잡한 UI 스타일링	반복적인 코드	추상화 가능

5. 컴파일 과정

- 작성한 `.scss` 파일은 **컴파일러**를 통해 `.css` 파일로 변환됨
- 대표 도구: `sass`, Webpack + `sass-loader`, Vite + `sass` 플러그인

```
sass input.scss output.css
```

또는

```
npm install -D sass
```

6. SCSS 구조화 예시

```
styles/
├── base/
|   ├── _reset.scss
|   ├── _typography.scss
├── components/
|   ├── _button.scss
|   ├── _card.scss
├── layout/
|   ├── _header.scss
|   ├── _footer.scss
├── pages/
|   ├── _home.scss
|   ├── _about.scss
├── utils/
|   ├── _variables.scss
|   ├── _mixins.scss
|   ├── _functions.scss
├── main.scss
```

→ `main.scss`에서 나머지를 `@use` 또는 `@import`로 불러옴

7. 결론

항목	요약
대상	중대형 프로젝트, 팀 단위 스타일 관리
장점	모듈화, 유지보수, 재사용성 탁월
사용 기술	SCSS 문법이 가장 많이 쓰임
주의점	빌드 환경 필요, 초심자에게 진입장벽 있을 수 있음

CSS-in-JS (Styled Components, Emotion 등)

CSS-in-JS는 JavaScript 코드 안에서 CSS를 정의하고 사용하는 방법을 의미한다.
이는 전통적인 CSS 파일 사용 방식과 달리, **스타일을 컴포넌트화**하고,
동적 스타일링, **범위 캡슐화**, **런타임 스타일 조작** 등을 쉽게 만들기 위해 고안된 접근 방식이다.

대표 라이브러리:

- **Styled Components**
- **Emotion**
- (기타: Linaria, Stitches, JSS, etc.)

1. CSS-in-JS의 특징

항목	설명
컴포넌트 단위 스타일	React 등의 컴포넌트 구조와 맞물려 스타일도 분리하지 않음
동적 스타일링 지원	JavaScript 변수, props, 상태값 등을 기반으로 스타일 조절 가능
자동 범위 지정	스타일 충돌 방지 (scoped className 자동 생성)
런타임/정적 지원	런타임에 적용하거나, 빌드 시 정적으로 추출 가능
생산성 향상	JS 로직과 스타일을 한곳에서 관리 가능

2. Styled Components 예시

```
npm install styled-components
```

기본 사용

```
import styled from 'styled-components';

const Button = styled.button`
  background: ${(props) => (props.primary ? 'blue' : 'gray')};
  color: white;
  padding: 0.5rem 1rem;
  border: none;
  border-radius: 4px;
`;

function App() {
  return <Button primary>Primary Button</Button>;
}
```

특징

- props를 통해 스타일 동적으로 조절 가능
- 컴포넌트가 곧 스타일 정의
- className이 자동 생성되어 중복 방지

---

3. Emotion 예시

```
npm install @emotion/react @emotion/styled
```

`@emotion/styled` 방식 (Styled Components 유사)

```
/** @jsxImportSource @emotion/react */
import styled from '@emotion/styled';

const Container = styled.div`
  background: #eee;
  padding: 1rem;
`;

const Title = styled.h1`
  color: ${(props) => props.color || 'black'};
`;

function App() {
  return (
    <Container>
      <Title color="tomato">Emotion Example</Title>
    </Container>
  );
}
```

`css` 방식 (클래스명 할당 방식)

```
import { css } from '@emotion/react';

const style = css`
  font-size: 20px;
  color: hotpink;
`;

function App() {
  return <div css={style}>Emotion 스타일</div>;
}
```

---

4. 주요 비교: Styled Components vs Emotion

항목	Styled Components	Emotion
사용 방식	`styled` 전용	`styled`, `css`, `Global` 등 다양한 방식
성능	런타임 스타일	런타임 + 정적 스타일 추출 지원
문법	태그드 템플릿 리터럴	템플릿, 객체 리터럴 모두 가능
확장성	비교적 직관적	더 많은 기능과 유연성 제공
코드 추적성	개발자 도구 친화적	이름 지정 가능 (`label`)

---

5. 장점

- **스타일 충돌 없음**: 자동 className 생성
- **컴포넌트 기반 설계**와 자연스럽게 통합
- **상태/props 기반의 동적 스타일링** 쉬움
- **JavaScript 로직 활용 가능** (조건, 반복 등)
- **SSR (서버사이드 렌더링) 지원** 가능

---

6. 단점

- **런타임 성능 비용** (특히 Styled Components)
- **빌드 시간 증가** 가능성
- **초기 학습 비용**: 문법과 빌드 설정 이해 필요
- **스타일만 보는 것이 어려움** (JS 내부에 스타일 포함)

---

7. 사용 시 고려할 점

항목	설명
프로젝트 크기	중~대형 SPA에 유리함
프레임워크	React 기반 프로젝트에 최적화
SEO	Emotion은 SSR 친화적, Styled Components는 설정 필요
디자인 시스템	디자인 토큰과 Theme 적용이 용이함 (`ThemeProvider`)

---

8. Theme 적용 예시 (Styled Components)

```
import { ThemeProvider } from 'styled-components';

const theme = {
  primary: 'tomato',
  padding: '1rem',
};

const Box = styled.div`
  color: ${(props) => props.theme.primary};
  padding: ${(props) => props.theme.padding};
`;

function App() {
  return (
    <ThemeProvider theme={theme}>
      <Box>테마 기반 스타일</Box>
    </ThemeProvider>
  );
}
```

결론

CSS-in-JS가 적합한 경우	비추천되는 경우
컴포넌트 중심의 개발 (React)	정적 사이트, 단순 HTML+CSS
동적 스타일 조작 필요	고성능이 필요한 대형 사이트
디자인 시스템 도입 시	낮은 런타임 오버헤드가 중요한 프로젝트

Tailwind CSS 소개

Tailwind CSS는 현대적인 **유틸리티 퍼스트(utility-first)** CSS 프레임워크이다.
클래스를 통해 스타일을 적용하는 방식으로, **CSS 파일을 거의 작성하지 않고도**
빠르게 UI를 구성할 수 있도록 설계되었다.

> “클래스를 조합하여 스타일링하는 방식”으로, Atomic CSS의 대표 구현체이다.

---

1. 특징 요약

항목	설명
유틸리티 퍼스트	미리 정의된 클래스 이름으로 스타일 지정 (예: `p-4`, `bg-blue-500`)
빠른 프로토타이핑	복잡한 CSS 작성 없이 빠르게 UI 구성 가능
클래스 기반 구성	모든 스타일은 HTML의 class 속성에 작성
커스터마이징	`tailwind.config.js`로 테마, 색상, 브레이크포인트, 폰트 등을 설정
JIT 엔진	Just-In-Time 빌드로 사용한 클래스만 CSS에 포함 → 퍼포먼스 향상
반응형, 다크모드 내장	별도 설정 없이 `sm:`, `md:`, `dark:` 등으로 사용 가능

---

2. 기본 예시

HTML + Tailwind 예시

```
<button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
  클릭
</button>
```

→ 배경색, 호버 스타일, 텍스트 색상, 폰트, 패딩, 테두리 둥글기 모두 클래스로 적용됨

---

3. 설치 방법 (Tailwind CLI 기준)

```
npm install -D tailwindcss
npx tailwindcss init
```

`tailwind.config.js` 생성 후, CSS에 다음을 포함:

```
/* ./src/input.css */
@tailwind base;
@tailwind components;
@tailwind utilities;
```

빌드:

```
npx tailwindcss -i ./src/input.css -o ./dist/output.css --watch
```

---

4. 주요 유틸리티 클래스

범주	예시	설명
색상	`bg-red-500`, `text-gray-700`	색상 단계별 지정
여백	`m-4`, `mt-2`, `p-8`, `px-6`	마진/패딩
정렬	`flex`, `justify-center`, `items-start`	Flexbox 정렬
타이포그래피	`text-lg`, `font-bold`, `leading-relaxed`	폰트 관련
레이아웃	`w-full`, `h-screen`, `max-w-md`	사이즈, 레이아웃 제어
반응형	`sm:text-sm`, `lg:px-8`	브레이크포인트 대응
상태	`hover:bg-blue-700`, `focus:outline-none`	상태 기반 스타일
다크 모드	`dark:bg-gray-800`, `dark:text-white`	다크모드 지원

---

5. 반응형 / 상태 클래스 접두어

접두어	설명
`sm:`	640px 이상
`md:`	768px 이상
`lg:`	1024px 이상
`xl:`	1280px 이상
`hover:`	호버 시 적용
`focus:`	포커스 시 적용
`dark:`	다크 모드일 때 적용

예시:

```
<div class="bg-white dark:bg-black sm:text-sm md:text-lg hover:text-red-500">
```

---

6. 커스터마이징

`tailwind.config.js`에서 테마 확장 가능:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        brand: '#1abc9c',
      },
      spacing: {
        '72': '18rem',
        '84': '21rem',
      },
    },
  },
};
```

---

7. 장점

항목	설명
빠른 개발	별도 CSS 작성 없이 클래스만 조합
일관성	디자인 시스템 없이도 규칙적인 스타일 적용
퍼포먼스	JIT 빌드로 사용된 CSS만 추출하여 최소화
반응형/상태/다크모드 등 쉽게 적용	

---

8. 단점

항목	설명
HTML이 복잡해짐	클래스가 많아짐 (`class="..."`)
재사용 어려움	추상화된 CSS 작성이 어려움 (복잡한 스타일은 `@apply` 필요)
기존 CSS 방식과 달라 학습 필요	전통적인 `SCSS`, `BEM`과 개념 다름

---

9. `@apply` 지시어 (CSS에서 클래스 적용)

```
.btn {
  @apply px-4 py-2 bg-blue-500 text-white rounded;
}
```

→ 반복되는 클래스를 CSS처럼 재사용 가능

---

10. 실제 사용 예시

```
<div class="grid grid-cols-3 gap-4 p-6 bg-gray-100">
  <div class="bg-white shadow rounded p-4">카드 1</div>
  <div class="bg-white shadow rounded p-4">카드 2</div>
  <div class="bg-white shadow rounded p-4">카드 3</div>
</div>
```

---

11. 결론

Tailwind CSS가 적합한 경우	부적합한 경우
빠른 UI 조립, 프로토타이핑	클래스를 최소화하고자 할 때
컴포넌트 기반 프레임워크 사용	전통적인 HTML+CSS 작업
디자인 일관성이 중요한 팀	시맨틱 마크업을 우선할 때