17. HTML 성능 최적화

17.1 이미지 지연 로딩 (`loading="lazy"`)

— 페이지 속도 향상을 위한 HTML5 표준 속성

`loading="lazy"` 속성은 이미지나 iframe 같은 리소스를 **화면에 보일 때까지 지연해서 로딩**하는 방식이다.
✅ **초기 로딩 속도 향상**, ✅ **트래픽 절약**, ✅ **UX 개선**에 매우 효과적이다.

✅ 기본 문법

```
<img src="photo.jpg" alt="사진 설명" loading="lazy">
```

속성값	의미
`lazy`	지연 로딩 (화면에 가까워질 때 로드됨)
`eager`	즉시 로딩 (기본 동작)
`auto`	브라우저가 판단 (사실상 eager처럼 작동)

✅ 어떤 요소에 사용 가능한가?

태그	지원 여부
`<img>`	✅ 사용 가능
`<iframe>`	✅ 사용 가능
`<video>` `<source>`	❌ (JavaScript 방식 필요)
`<picture>` 내부 `<img>`	✅ `<img>`에 적용

예:

```
<iframe src="map.html" loading="lazy"></iframe>
```

✅ 왜 필요한가?

🔸 예시: 한 페이지에 이미지 50개가 있는 블로그

- `loading="eager"` 일 경우 → 페이지 로딩 시 이미지 50개 **모두 불러옴**
- `loading="lazy"` 일 경우 → **스크롤해서 보일 때** 개별 이미지 **하나씩** 불러옴

📌 → **최초 페이지 로딩 속도가 훨씬 빨라지고, 네트워크 사용량이 감소함**

✅ 시각화 이해

```
코드 복사사용자 뷰포트
──────────────
|             |
|  🖼️ 이미지 1 |  ← 즉시 로딩
|  🖼️ 이미지 2 |  ← 즉시 로딩
|             |
|   화면 밖    |
|             |
|  🖼️ 이미지 3 |  ← 나중에 로딩됨
|  🖼️ 이미지 4 |  ← 나중에 로딩됨
──────────────
```

✅ 브라우저 지원

브라우저	지원 여부
Chrome 76+	✅ 완벽 지원
Firefox 75+	✅ 지원
Edge (Chromium)	✅ 지원
Safari (16+)	✅ 제한적 지원 (점차 개선 중)
구형 브라우저	❌ fallback 필요 (JS로 대체 가능)

🔸 **Safari 15 이하, IE**는 미지원 → IntersectionObserver API로 대체 가능

✅ Best Practice 예시

```
<img
  src="article-cover.jpg"
  alt="기사 대표 이미지"
  loading="lazy"
  width="800"
  height="600">
```

> `width`와 `height` 속성은 레이아웃 shift(레이아웃이 밀리는 현상)를 방지하므로 함께 쓰는 것이 SEO/UX에 좋음

✅ SEO & Core Web Vitals 영향

지표	영향
LCP (Largest Contentful Paint)	✅ 향상
CLS (Cumulative Layout Shift)	✅ 이미지 크기 명시 시 안정적
TTI (Time to Interactive)	✅ 긍정적 영향

구글은 `loading="lazy"` 사용을 **공식적으로 권장**함 → Google 웹.dev

✅ 요약 정리

항목	설명
속성	`loading="lazy"`
사용 대상	`<img>`, `<iframe>`
장점	초기 로딩 속도 향상, 리소스 절약, SEO 개선
단점	구형 브라우저 미지원 (fallback 고려)
팁	`width`, `height` 명시 필수 / JS 대체도 고려

17.2 리소스 미리 불러오기 (`<link rel="preload">`)

✅ 개요

`<link rel="preload">`는 **웹 페이지 렌더링 이전에 특정 리소스를 미리 불러오도록 브라우저에 지시**하는 HTML5 기능이다. 페이지에서 중요한 리소스(예: 폰트, 이미지, 스크립트 등)를 렌더링 전에 미리 다운로드해 두면 **렌더링 지연을 줄이고 성능을 개선**할 수 있다.

이는 단순한 `<link>`나 `<script>` 태그보다 **의도적이고 명시적인 성능 최적화 도구**로 사용된다.

🧩 문법

```
<link rel="preload" href="리소스경로" as="리소스타입">
```

- `rel="preload"`: preload의 핵심
- `href`: 미리 로드할 리소스의 URL
- `as`: 리소스의 유형을 명시 (필수 속성)
- `type`, `crossorigin`: 일부 리소스에 따라 필요

🔧 `as` 속성: 리소스의 타입

`as` 값	의미
`script`	JavaScript 파일
`style`	CSS 스타일시트
`image`	이미지 리소스 (`.jpg`, `.png`)
`font`	웹폰트 (`.woff2`, `.woff`)
`document`	iframe용 문서
`fetch`	fetch API로 불러올 데이터
`audio`	오디오 리소스
`video`	비디오 리소스
`track`	비디오/오디오 자막
`worker`	웹 워커 (`.js`)

🧪 예제 1: 폰트 미리 불러오기

```
<link rel="preload" href="/fonts/NotoSansKR.woff2" as="font" type="font/woff2"
crossorigin="anonymous">
```

- `type` 속성은 미디어 타입 명시
- `crossorigin`은 CORS 헤더 처리를 위해 필요

🧪 예제 2: 스크립트 미리 불러오기

```
<link rel="preload" href="/js/critical.js" as="script">
```

단, 이 경우 `preload`**는 스크립트를 "실행"하지는 않으며**, 이후 `<script src="...">`로 불러야 실행됨.

⚠️ 주의 사항

- `as` **속성은 반드시 지정**해야 preload가 유효하게 작동함.
- **중복 다운로드 방지**를 위해 preload한 리소스는 반드시 해당 리소스를 후속 HTML/CSS/JS에서 참조해야 함.
- preload만으로 실행이 되지 않음. 실행은 별도 태그 필요.
- `crossorigin` 속성 누락 시 폰트 등 일부 리소스는 캐시가 무효화될 수 있음.

🚀 preload vs prefetch vs prerender

유형	목적	사용 시점
`preload`	**현재 페이지에서 반드시 필요한 리소스**를 미리 로드	즉시
`prefetch`	다음 페이지 또는 가능성 있는 리소스를 미리 로드	유휴 시간
`prerender`	다음 페이지를 미리 렌더링까지 함	유휴 시간 (무겁고 제한적)

✅ 사용 예시 (최적화 시나리오)

- Web Font FOUT(Fallback of Unstyled Text) 방지
- 첫 화면 렌더링에 필요한 Hero 이미지 미리 불러오기
- 주요 JavaScript 번들 조각을 미리 내려받기

📌 실전 팁

- Lighthouse 성능 점검에서 "Preload key requests" 경고가 있으면 `<link rel="preload">`로 해결 가능
- CSS 파일에는 보통 `preload`보다 `prefetch`나 `media="print"` trick이 더 유효

17.3 스크립트 위치 최적화 (`defer`, `async`)

✅ 개요

HTML에서 JavaScript `<script>` 태그를 어디에, 어떻게 배치하는지에 따라 **페이지 로딩 속도와 렌더링 성능이 크게 달라진다**.
특히 HTML5에서는 `defer`와 `async` 속성을 통해 **스크립트의 로딩과 실행 시점을 제어**할 수 있다.

🧩 기본 `<script>` 동작 방식

```
<script src="script.js"></script>
```

- HTML은 위 코드를 만나면 **스크립트를 즉시 다운로드하고 실행한 뒤에야 다음 HTML을 파싱**한다.
- 즉, **파싱 중단(blocking)**이 발생하여 **페이지 로딩 지연**의 원인이 됨.

⚙️ `defer` 속성

```
<script src="script.js" defer></script>
```

- HTML 파싱 **동시에** 스크립트를 **비동기 다운로드**
- HTML 문서 파싱이 끝난 후, `DOMContentLoaded` 이벤트 직전에 **순서대로 실행**
- **문서 구조와 분리된 실행이 가능하므로**, 보통 `head`에 넣는 것을 권장

- 여러 스크립트 간 **실행 순서가 보장됨**

예시:

```
<head>
  <script src="lib1.js" defer></script>
  <script src="lib2.js" defer></script>
</head>
```

→ `lib1.js` → `lib2.js` 순서 보장됨.

⚙️ `async` 속성

```
<script src="script.js" async></script>
```

- HTML 파싱과 **동시에** 비동기로 스크립트를 다운로드
- 다운로드가 완료되는 즉시 **파싱을 멈추고 스크립트를 실행**
- **실행 순서 보장되지 않음**
- **독립적인 스크립트** (예: 광고, 통계 등)에 적합

예시:

```
<head>
  <script src="ad.js" async></script>
  <script src="analytics.js" async></script>
</head>
```

→ 다운로드 완료 순서에 따라 실행 순서가 달라질 수 있음.

📊 비교 정리

속성 없음 (`<script>`)	`defer`	`async`
HTML 파싱 중단	파싱과 병렬 다운로드	파싱과 병렬 다운로드
다운로드 후 즉시 실행	파싱 끝난 후 순차 실행	다운로드 후 즉시 실행
순서 보장 X	순서 보장 O	순서 보장 X
렌더링 차단 O	렌더링 차단 X	렌더링 차단 일부 가능
적합한 용도	중요 스크립트, 모듈 로딩	통계, 광고 등 독립적 스크립트

🧠 위치별 활용 전략

위치	속성	권장 여부 및 용도
`<head>`	`defer`	✅ (권장) HTML 파싱 차단 없음
`<head>`	`async`	⚠️ (비추천) 순서/종속성 주의
`<body>` 하단	없음	🆗 전통적 방식 (HTML 다 파싱 후 실행됨)
`<body>` 하단	`defer`, `async`	❌ 무의미 (이미 파싱 완료됨)

✅ 실전 팁

- Vue, React, Angular 같은 SPA 빌더는 자동으로 `defer` 처리함
- 여러 의존성이 얽힌 스크립트는 `defer`가 안전
- 독립 스크립트(예: Google Analytics)는 `async`가 적절

📌 최종 정리

```
<!-- 순서가 중요하고 페이지 전체에 영향 주는 스크립트 -->
<script src="main.js" defer></script>

<!-- 순서가 상관없는 외부 스크립트 (광고, 분석 등) -->
<script src="ads.js" async></script>
```

17.4 DOM 최소화 전략

✅ 개요

DOM(Document Object Model)은 브라우저가 HTML 문서를 객체 구조로 표현한 것으로, 웹페이지가 커질수록 DOM 트리도 커진다.
하지만 **불필요하게 복잡하거나 깊은 DOM 구조는 렌더링 성능 저하, 자바스크립트 처리 지연, 메모리 증가** 등의 문제를 유발한다.

> 핵심 목표:
> **DOM 요소의 개수와 깊이를 줄이고**, 렌더링·스크립트 처리 성능을 최적화하자.

🧩 왜 DOM이 많으면 느려질까?

- 브라우저는 DOM 요소 각각을 **레이아웃 → 스타일 계산 → 페인트 → 컴포지트** 단계로 처리함
- DOM 트리가 깊거나 복잡하면:
  - **스타일 계산 시간 증가**
  - **Reflow/Repaint 발생 빈도 증가**
  - **JS DOM 접근 성능 저하**

- 메모리 사용량 증가

🛠️ 최적화 전략

1. 불필요한 태그 제거

- 의미 없이 `<div>`나 `<span>`을 남용하지 말 것
- CSS로만 해결되는 구조에 굳이 중첩 구조를 만들지 말 것

```
<!-- ❌ 나쁜 예시 -->
<div class="wrapper">
  <div class="inner">
    <span class="text">Hello</span>
  </div>
</div>

<!-- ✅ 좋은 예시 -->
<p>Hello</p>
```

---

2. 중첩(depth) 최소화

- DOM의 **깊은 트리 구조**는 렌더링에 부담이 됨
- 레이아웃을 그리드/Flexbox로 간소화하고, 자식 요소를 덜어낼 수 있는 구조를 고민할 것

---

3. 컴포넌트화 및 템플릿 활용

- 반복되는 DOM을 **템플릿화**하고 재사용할 수 있도록 구성
- 불필요한 inline 구조는 `template`, `include`, JavaScript 템플릿 시스템으로 분리

---

4. 조건부 렌더링 (불필요한 DOM 렌더링 지연)

- 화면에 필요할 때만 DOM을 생성 (예: 탭, 모달, 토글 UI)
- JavaScript로 실제로 사용할 때 DOM을 동적으로 생성하거나 숨겨둠

```
<!-- ❌ 페이지 로딩 시 불필요한 DOM -->
<div class="modal">...</div>

<!-- ✅ 필요할 때만 DOM 생성 -->
<!-- 또는 display: none → visible 토글 -->
```

---

5. 리스트 렌더링 최적화

- 수백 개 이상의 리스트를 한 번에 DOM에 그리지 말고, **가상 스크롤(Virtual Scroll)** 도입
- 필요할 때만 보이는 부분만 그려주는 기법 사용 (예: `react-window`, `virtual list`)

---

6. 반복 DOM 요소는 ID 대신 `class` 사용

- 여러 개 있는 DOM을 접근할 때 `id`가 아닌 `class`로 묶고 `querySelectorAll`로 접근

📈 실전 도구

- **Chrome DevTools → Performance 패널**: DOM 트리 구조와 렌더링 비용 확인
- **Lighthouse**: “Avoid an excessive DOM size” 경고 제공

> 권장 기준 (Lighthouse 기준):
>
> - 총 노드 수 1,500개 미만
> - 최대 깊이 32 이하
> - 부모 요소의 자식 수 60 이하

🧠 요약

항목	전략 요약
태그 수 줄이기	불필요한 `<div>` 제거
중첩 구조 간소화	Flex/Grid로 레이아웃 단순화
조건부 렌더링	필요한 시점에만 DOM 추가/보임
반복 요소 최적화	Virtual Scroll, Lazy DOM
공통 템플릿화	`<template>` 또는 JS로 컴포넌트화
실시간 DOM 수정 최소화	Batch 업데이트, Debounce 적용

✅ 함께 사용하면 좋은 기술

- `requestAnimationFrame()`: DOM 업데이트 시점 최적화
- `IntersectionObserver`: 뷰포트 진입 시 Lazy DOM 처리
- **Shadow DOM**: 로컬 DOM 구성 (Web Components)