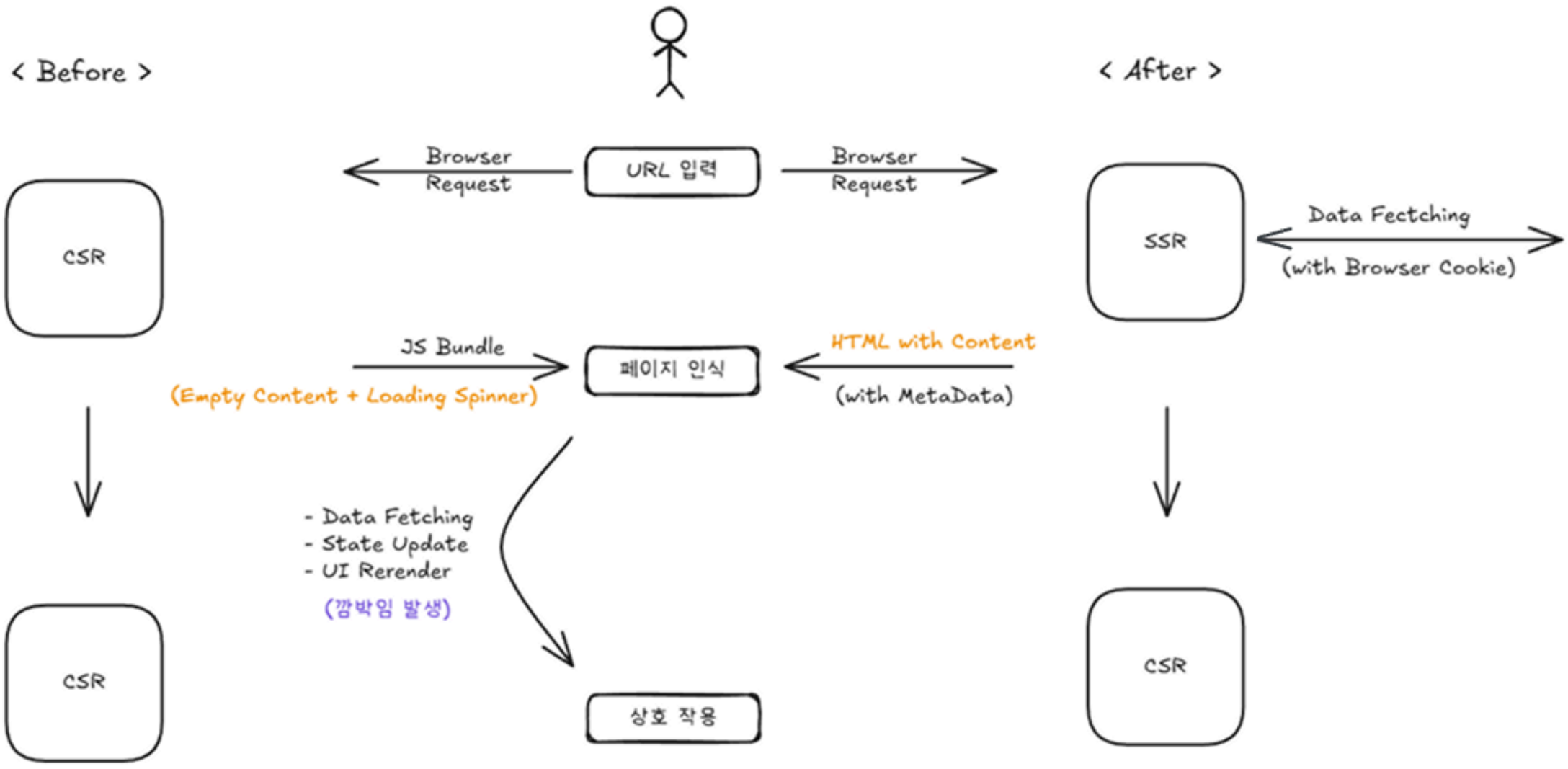
사용자 초기 경험과 검색 노출 문제를 해결하기 위한 Next.js 하이브리드 렌더링 설계



[문제 상황]

- 페이지 초기 진입 시 빈 화면이 보이다가 데이터가 렌더링되는 UI 깜박임 발생으로 사용자 경험 저하
- 클라이언트 데이터 페칭으로 인해 초기 렌더링 시 로딩 스피너가 필수적으로 노출되어 Layout Shift 발생 및 First Contentful Paint(FCP) 지표 악화
- 초기 HTML에 핵심 콘텐츠가 부족해 검색 엔진 봇이 콘텐츠를 수집하지 못하는 SEO 취약점이 존재하여 서비스 검색어 유입에 제한

[해결 과정]

- JS 파일을 내려받은 뒤 상태를 업데이트하는 구조로 인해 데이터 의존 UI가 비동기로 채워지며 초기 HTML에 빈 영역이 발생한다고 판단
- 초기 HTML 단계에서 콘텐츠를 포함할 수 있도록 클라이언트 컴포넌트 대신 서버 컴포넌트를 사용하는 방식을 선택
- Next.js 서버 컴포넌트에서 초기 렌더에 필요한 데이터를 먼저 로드한 뒤 클라이언트에 초기 상태를 주입하여 첫 HTML 단계에서 핵심 콘텐츠가 포함되도록 하이브리드 렌더링 구조를 설계
- SSR로 초기 로딩 시간 단축 후 CSR로 인터랙션(무한 스크롤, 필터링) 수행
- 사전 조회된 데이터와 Next.js Metadata API를 활용해 각 페이지별 동적 메타데이터 생성, robots.txt 및 sitemap.xml 생성으로 검색 엔진 크롤링 효율화

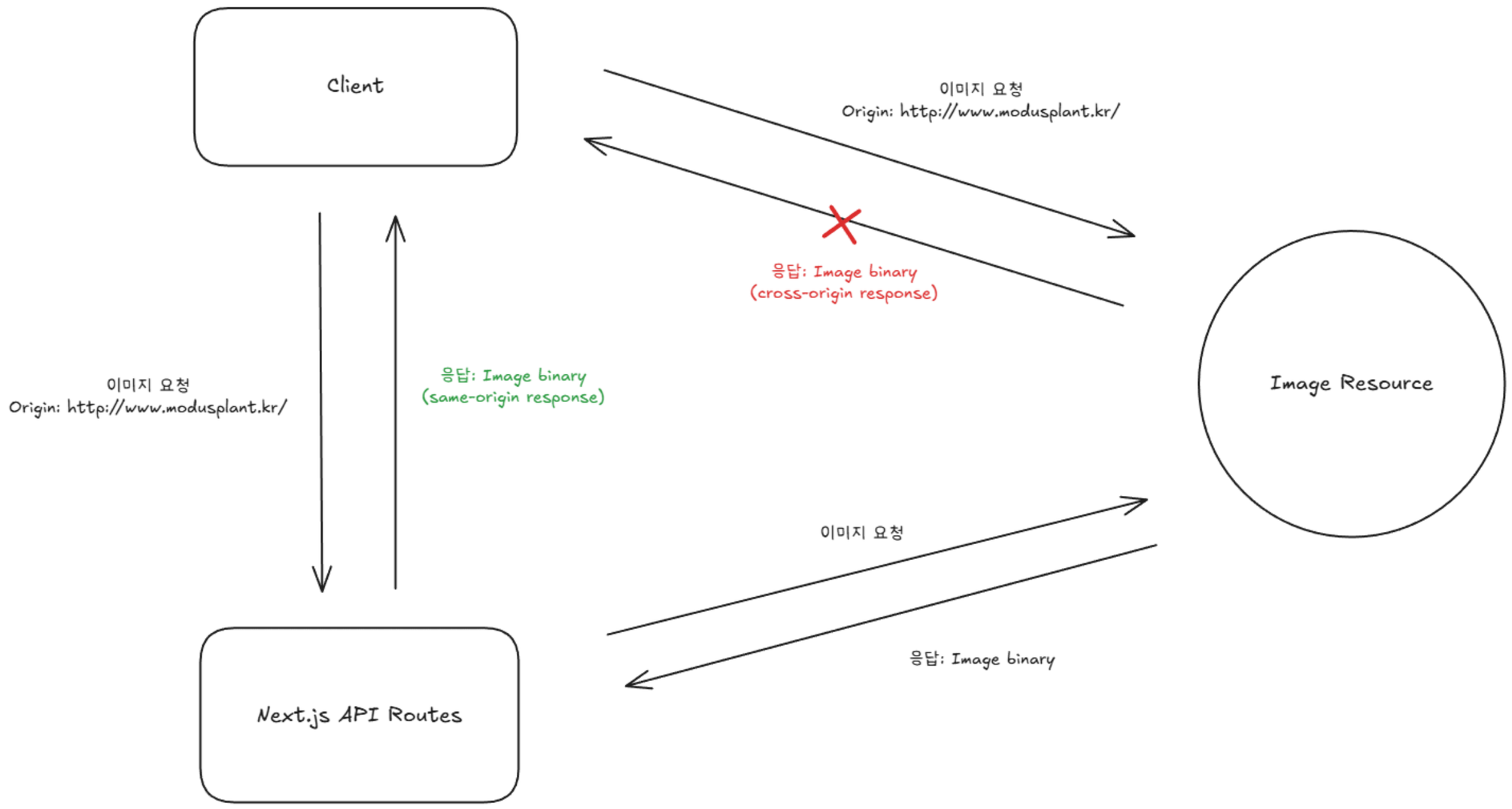
[결과]

- 초기 렌더링 시 콘텐츠가 즉시 표시, 시각적 흔들림을 개선하여 사용자가 첫 화면을 더 '완성된 상태'로 인지하도록 UX 개선
- 첫 렌더 HTML에 핵심 텍스트 콘텐츠가 포함되면서 검색엔진이 페이지 주제를 더 명확히 해석할 수 있는 구조 확보
- 구글 검색엔진에 "모두의 식물" 키워드로 1페이지 상위 노출 달성, 검색 유입 기반의 트래픽 확보에 기여 🔗

[SSR 도입 과정에서 마주한 인증 상태 문제 해결]

- 서버 컴포넌트가 브라우저 메모리에 접근 불가한 제약으로 인해 인증 토큰을 쿠키 기반으로 재설계해 SSR 환경에 인증 상태 공유
- 서버/클라이언트의 쿠키 조작 방식 차이로 환경별 쿠키 유틸 및 API 인스턴스 분리 구현
- 서버 컴포넌트 호환을 위해 fetch 기반 공통 코어를 구축하고 Axios를 벤치마킹한 인스턴스 형태로 래핑
- `(server|client)ApiInstance.method<ResponseType>(endPoint,body,options)` 형태의 표준화된 인터페이스로 유지보수성과 생산성 향상

CORS 정책 대응을 위한 서버 사이드 프록시 구현



[문제 상황]

- 이미지 비중이 큰 화면에서 외부 도메인의 이미지 리소스를 직접 호출하면서 이미지가 깨지거나 로딩에 실패하는 현상이 발생했고 이로 인해 UI 신뢰도와 첫 인상이 크게 저하됨
- 게시글 수정 시 이미지를 가공하는 과정(URL 리소스 → fetch → Blob 변환)에서 CORS 에러로 인한 이미지 재전송 불가능 문제 발생

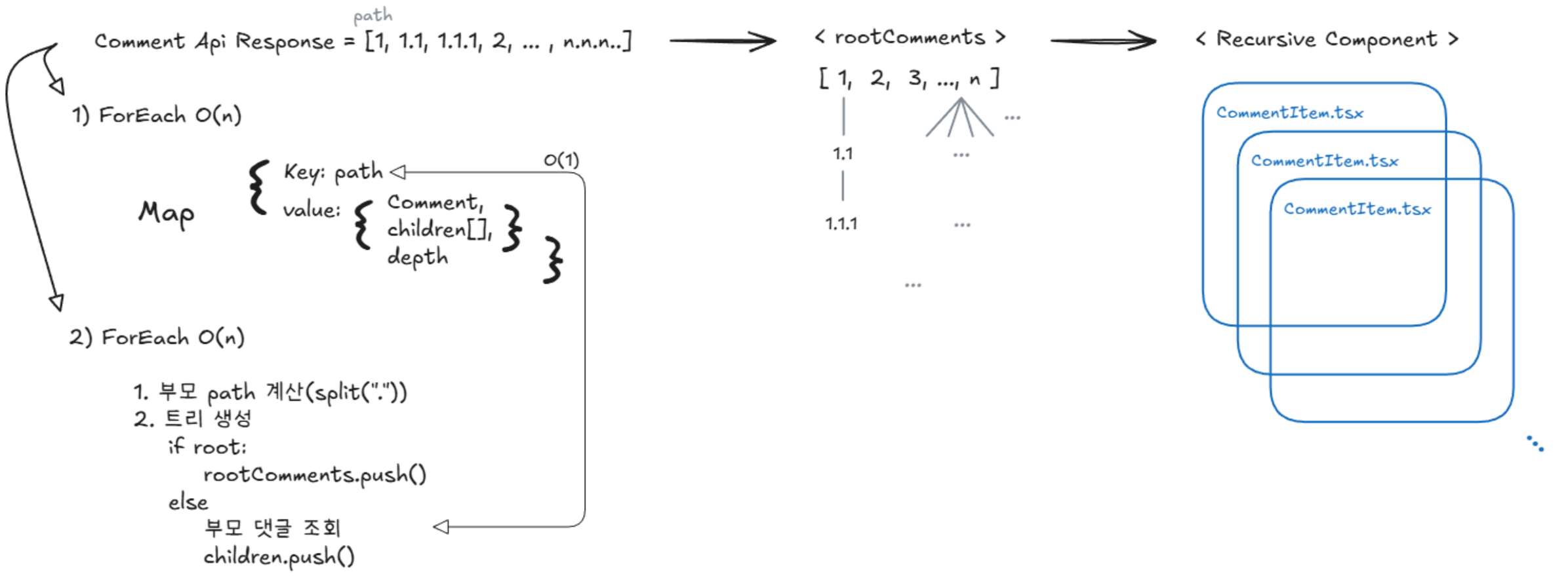
[해결 과정]

- 별도의 프록시 서버를 구축하는 것은 인프라 비용 발생 및 배포/관리 복잡도가 증가한다고 판단
- 추가 인프라 없이 서버리스처럼 동작하는 Next.js API Routes를 활용하기로 결정
- `/api/image-proxy?url={ImageURL}` 엔드포인트를 구현하여 클라이언트 요청을 서버 사이드에서 중계
- 클라이언트가 외부 도메인을 직접 호출하지 않고 이미지 프록시 API를 통해 동일 출처의 이미지를 요청하는 구조로 전환
- 외부 이미지 URL을 프록시 경로로 변환하는 헬퍼 함수를 구현하여 게시글 수정 로직간 일관된 방식으로 이미지 처리

[결과]

- CORS로 인한 외부 이미지 로딩 실패 해소, 이미지가 필요한 화면에서 깨짐/빈칸 현상을 줄여 UX 안정화
- 게시글 수정 로직이 정상 작동하여 사용자가 이미지를 재업로드하지 않아도 기존 이미지를 유지하며 수정할 수 있는 UX 완성

Path 기반 플랫 댓글 데이터를 트리 구조로 변환한 무한 Depth 댓글 설계



[문제 상황]

- 백엔드 API가 댓글을 path 필드("1", "1.1", "1.1.1") 기반의 플랫 배열로 응답하지만, UI는 들여쓰기와 답글의 답글을 표현하는 계층 구조(트리)로 렌더링해야 하는 데이터 구조 불일치 문제
- Depth마다 별도의 하위 리스트 컴포넌트를 하드코딩하는 방식은 Depth가 깊어질수록 코드가 복잡해지고 데이터 정합성을 맞추기 어려워짐
- Depth 제한 방식은 기획 요구사항(무한 답글)과 불일치
- API 응답 데이터를 그대로 렌더링하며 각 댓글마다 부모를 선형 탐색하는 구조로 구현할 경우 전체 시간 복잡도가 $O(n^2)$로 증가해 댓글 수가 증가할수록 성능 저하가 발생하는 구조적 한계

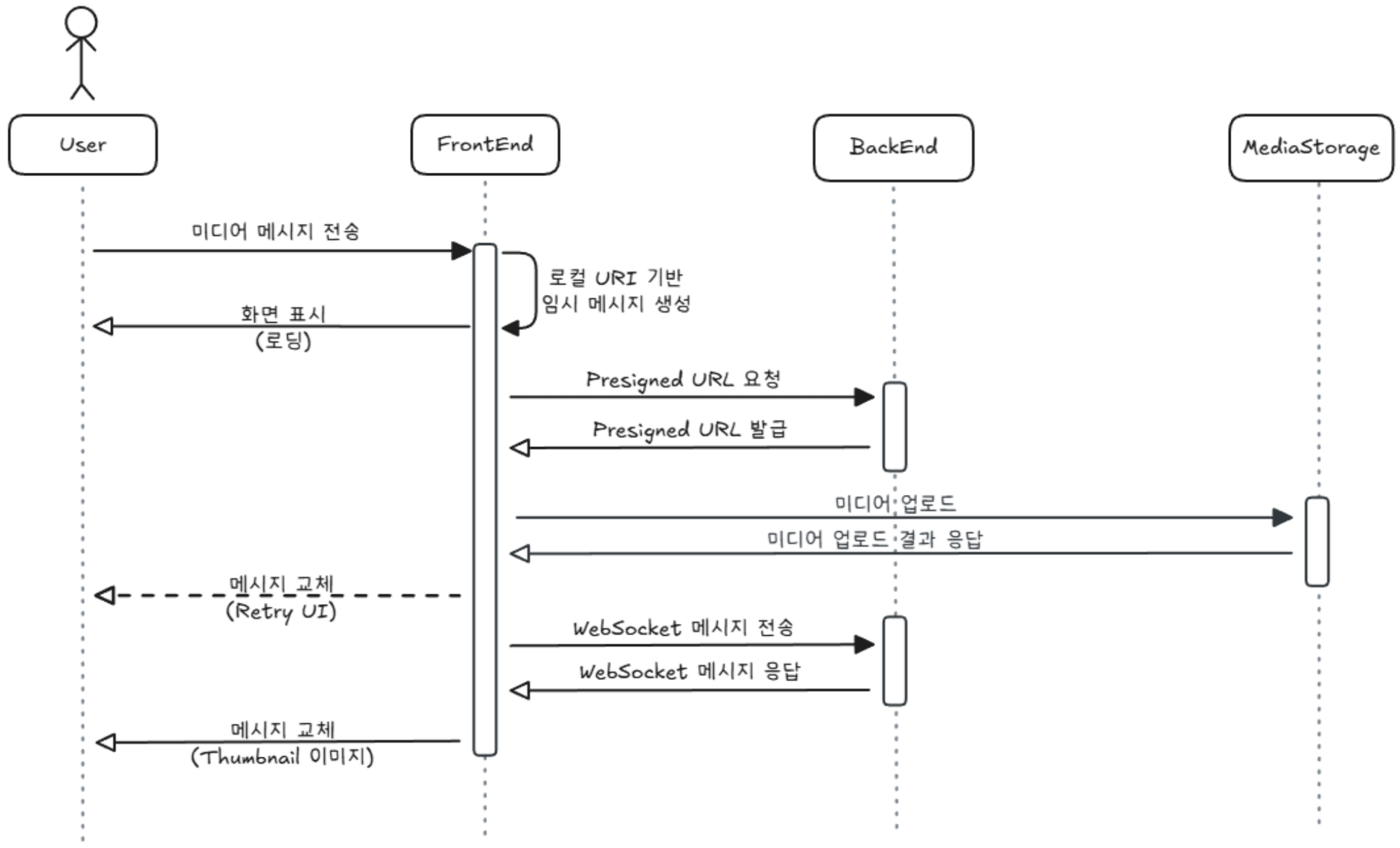
[해결 과정]

- 백엔드에서 트리 구조로 변환하여 전달하는 경우 API 응답 크기 증가(중첩 JSON) 및 서버 리소스 부담으로 상대적으로 리소스 여유가 있는 클라이언트 측에서 데이터를 가공하기로 결정
- 첫 번째 순회에서 path를 key로 하는 Map을 구성하여 부모 댓글을 O(1)로 조회할 수 있는 기반을 마련
  → 두 번째 순회에서 path를 기준으로 부모 path를 계산한 뒤 Map을 통해 부모 노드를 즉시 조회하여 children에 연결
  → Depth가 1인 rootComments 반환
- 렌더링 시 rootComments를 순회, children이 있는 경우 자기 자신을 호출하는 재귀 컴포넌트(Recursive Component) 패턴을 도입

[결과]

- 깊이 제한 없는 댓글 시스템의 시간 복잡도를 $O(n^2)$이 아닌 O(n)으로 구현
- 재귀 컴포넌트 패턴을 통해 단 하나의 컴포넌트로 모든 Depth를 커버하여 Depth 증가에 따른 UI 로직 복잡도를 최소화하고 유지보수성을 확보

즉각적 피드백을 제공하는 미디어 메시지 업로드 흐름 재설계



[문제 상황]

- 미디어 전송 시 업로드 완료까지 화면에 아무것도 표시되지 않아 “눌렀는데 아무 일도 안 일어나는” 지연 경험 발생
- 특히 네트워크 상태가 나쁘거나 파일 크기가 커질수록 입력 → 반응까지의 시간이 길어져 전송 성공/실패 여부 판단의 어려움으로 UX 저하
- 미디어 파일이 서버를 경유하여 스토리지로 업로드되면서 서버 네트워크 대역폭 및 메모리 과다 소비

[해결 과정]

- 사용자 니즈는 즉각 업로드가 아닌 즉각 피드백이라 판단하고 UI 응답성 개선을 위해 미디어 메시지 전송에 Optimistic Update 패턴을 적용
- 서버 부하 감소를 위해 Presigned URL 기반 직접 업로드 방식을 도입하여 클라이언트가 스토리지에 직접 업로드
- 사용자가 미디어를 선택하는 순간 UUID 기반의 tempId를 생성하고 로컬 URI를 참조하는 임시 메시지를 즉시 채팅창에 추가하여 화면에 표시하고 실제 전송은 비동기로 수행
- uploadStatus 필드를 통해 'pending → uploading → success/failed' 순으로 상태를 관리하며, 각 단계마다 사용자에게 시각적 피드백을 제공
- 업로드 완료 이벤트를 실시간으로 반영하기 위해 WebSocket 기반 메시지 수신 시 tempId를 기준으로 임시 메시지를 실제 메시지로 교체
- 실패 시에는 에러 메시지와 재시도 버튼을 표시하여 사용자가 직접 재전송할 수 있도록 설계

[결과]

- 미디어 메시지 전송 직후 화면에 UI가 나타나는 “즉시성”을 제공하여 체감 지연을 제거
- 미디어 파일 업로드 중에도 다른 메시지 입력과 스크롤이 가능한 논블로킹 인터랙션 구현
- 백엔드 서버의 대용량 파일 중계 부담을 제거하여 미디어 트래픽이 많은 상황에서의 서버 리소스 사용과 응답성 악화 가능성 개선