BootMe | 개발자 커뮤니티 & 소프트웨어 커리큘럼 Aggregator 웹 서비스

개발 기간 2022.09 ~ 진행 중

개발 인원 1명 (개인 프로젝트)

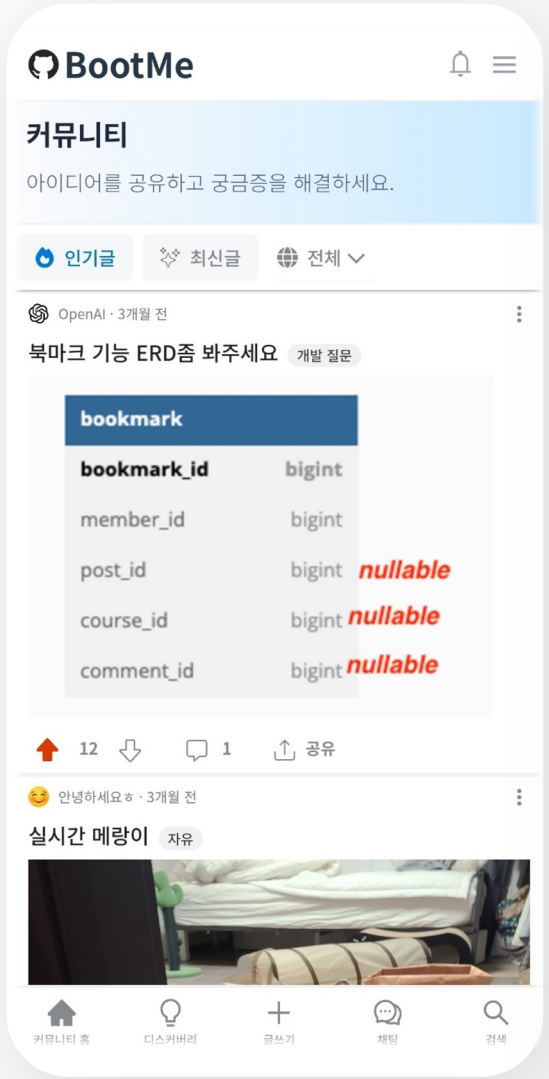
기술 스택

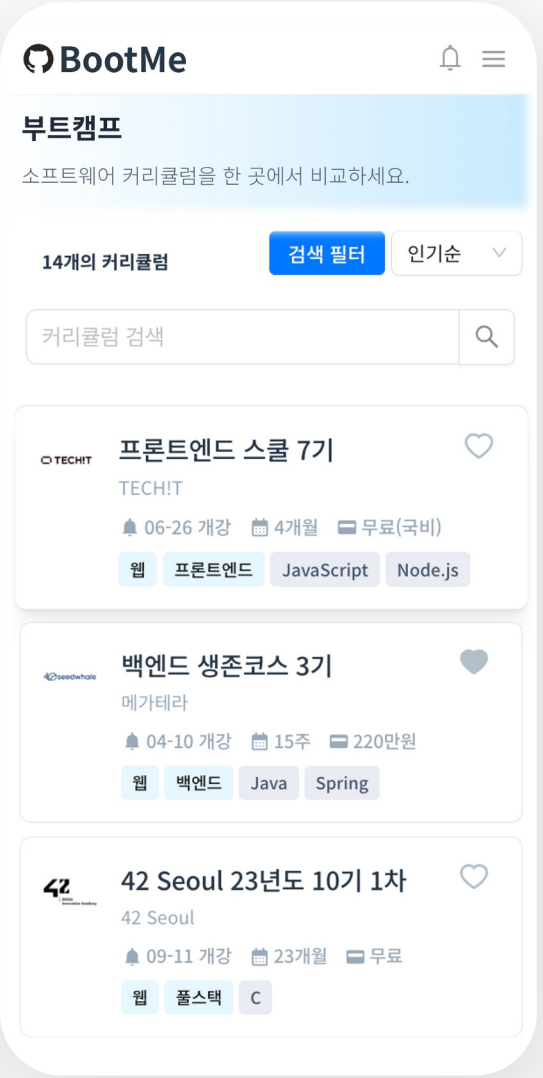
- **Java**, TypeScript
- **Spring Boot**, React
- MariaDB, Redis, Elasticsearch
- AWS CodeDeploy, CloudFront, EC2, S3

기타 도구

결과물 URL

GitHub	https://github.com/Jinwook94/bootme
서비스 (운영)	https://bootme.co.kr/
서비스 (스테이징)	https://staging.bootme.co.kr/
개발 문서 (Notion)	https://url.kr/x8fmg2
API 문서 (Swagger)	https://api.bootme.co.kr/docs/swagger/index.html
API 문서 (Rest Docs)	https://api.bootme.co.kr/docs/rest/index.html
ERD	https://shorturl.at/amnGY
디자인 (Figma)	https://url.kr/fdnj4u

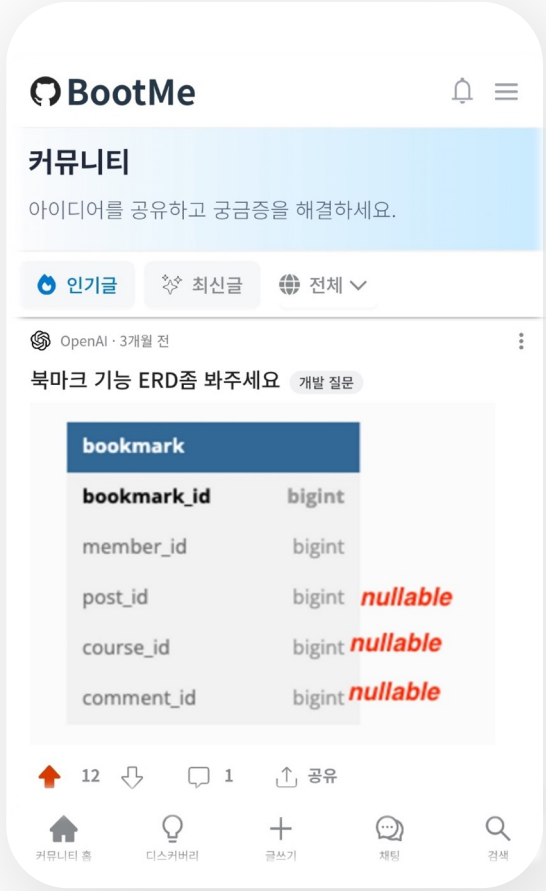




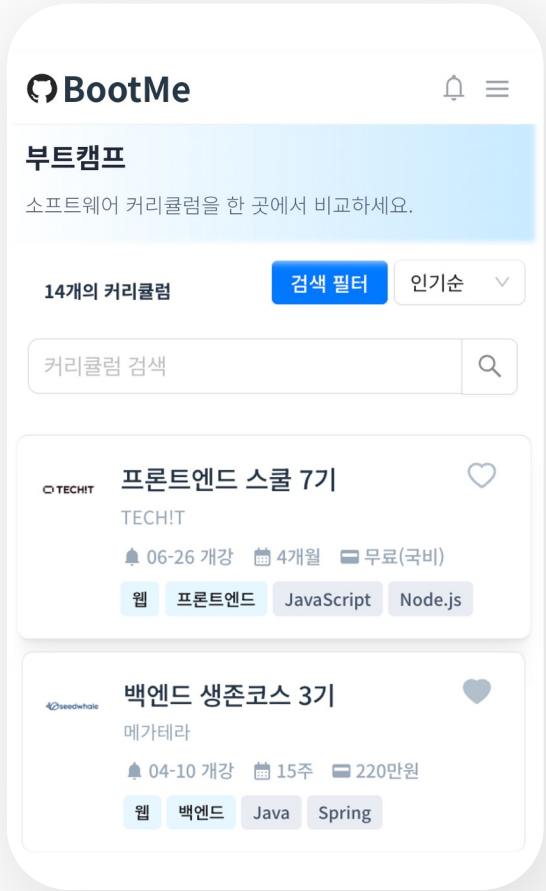
목차

BootMe | 주요 기능 요약 (1)

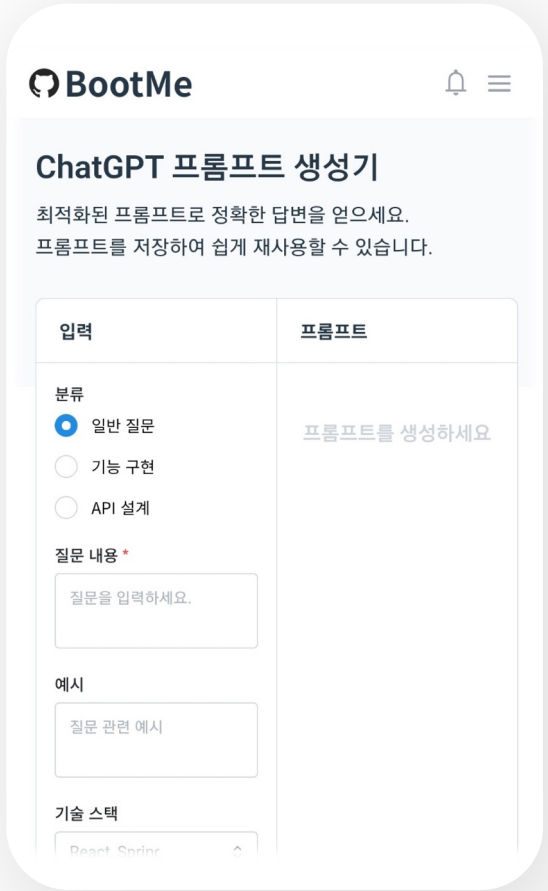
개발자 커뮤니티



부트캠프 리스트

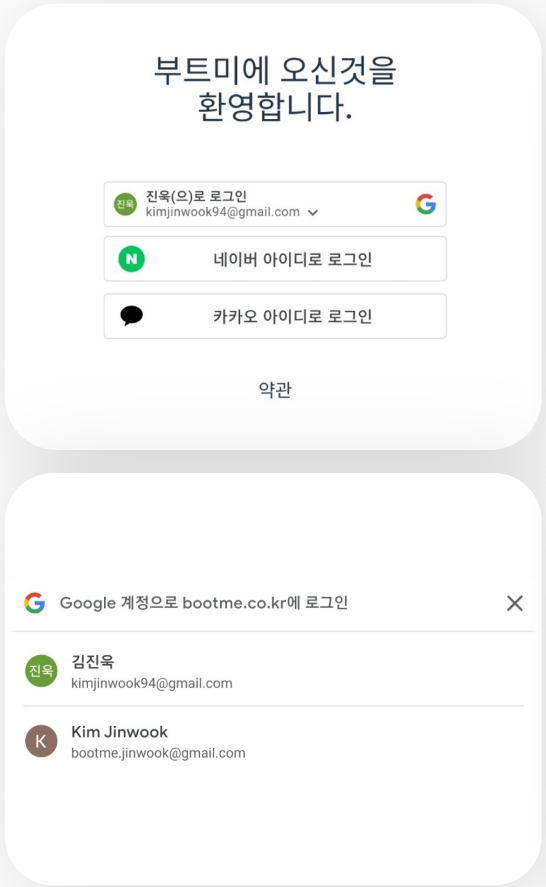


프롬프트 생성

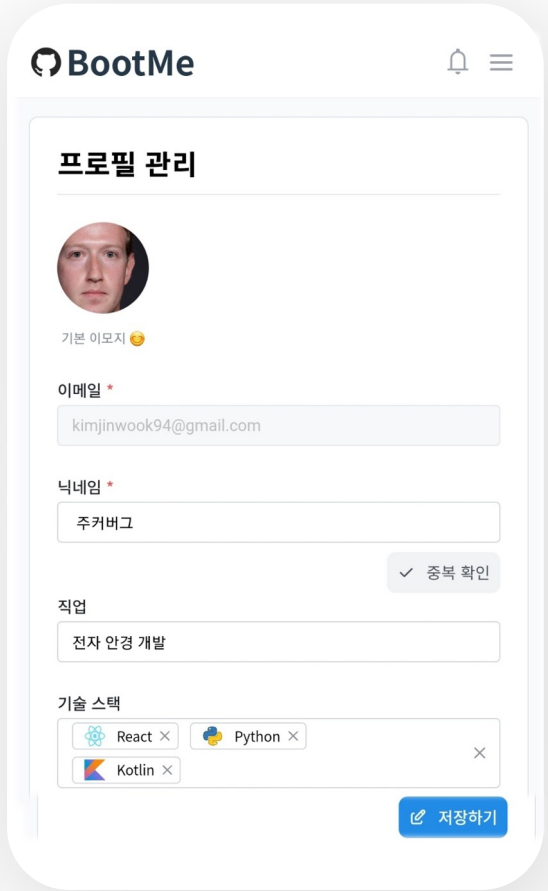


BootMe | 주요 기능 요약 (2)

소셜 로그인

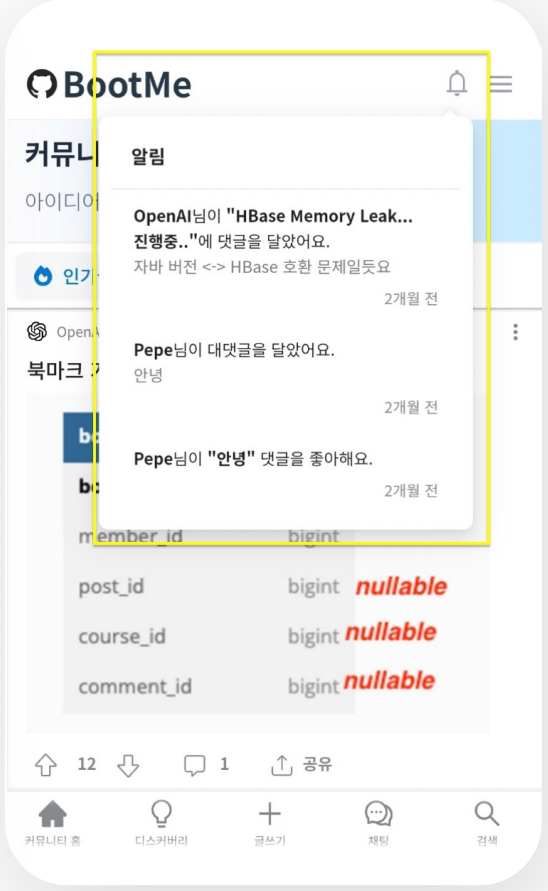


프로필 관리



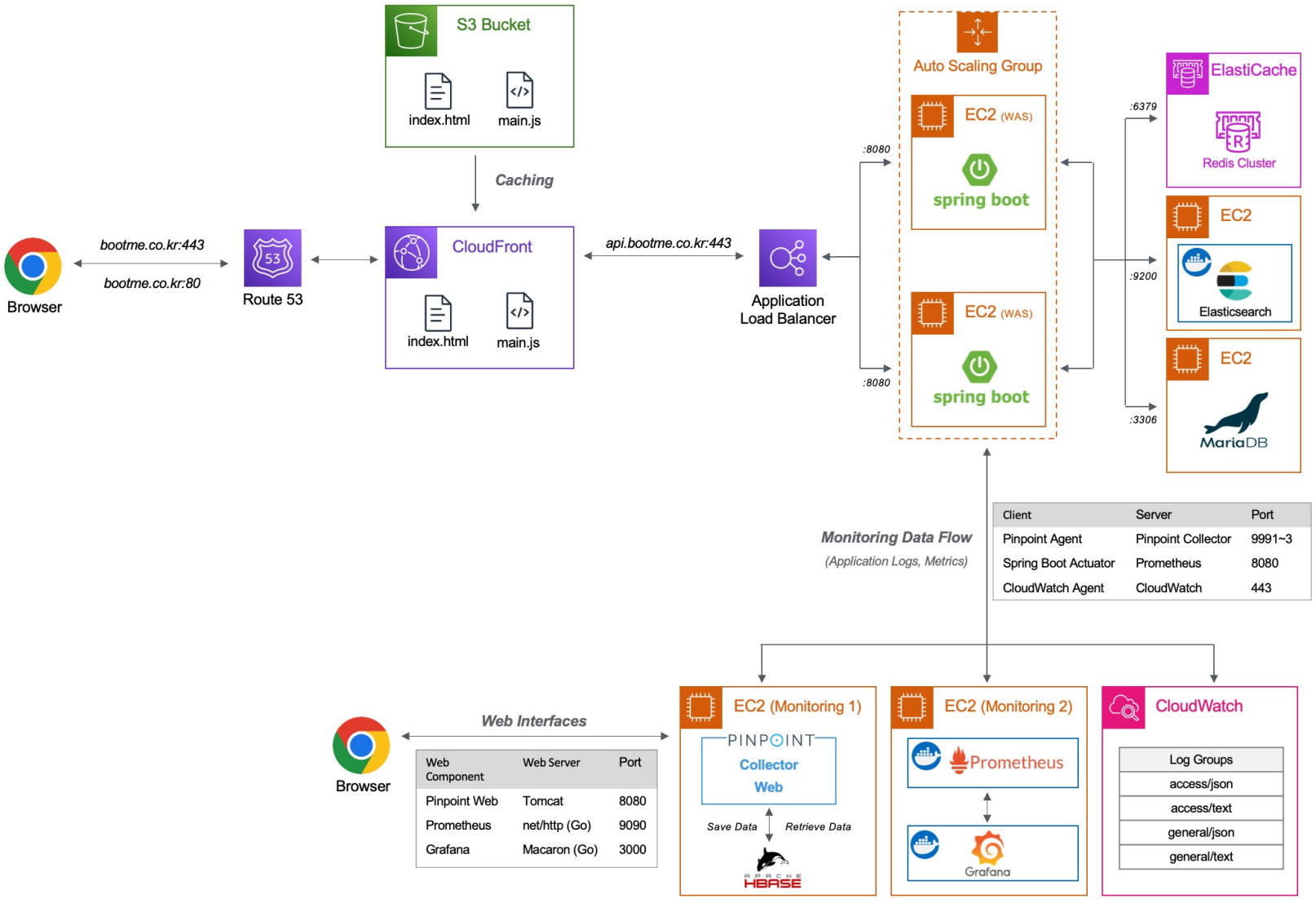
북마크 저장

실시간 알림



목차

배포 | 운영환경 아키텍처



관련 의사결정

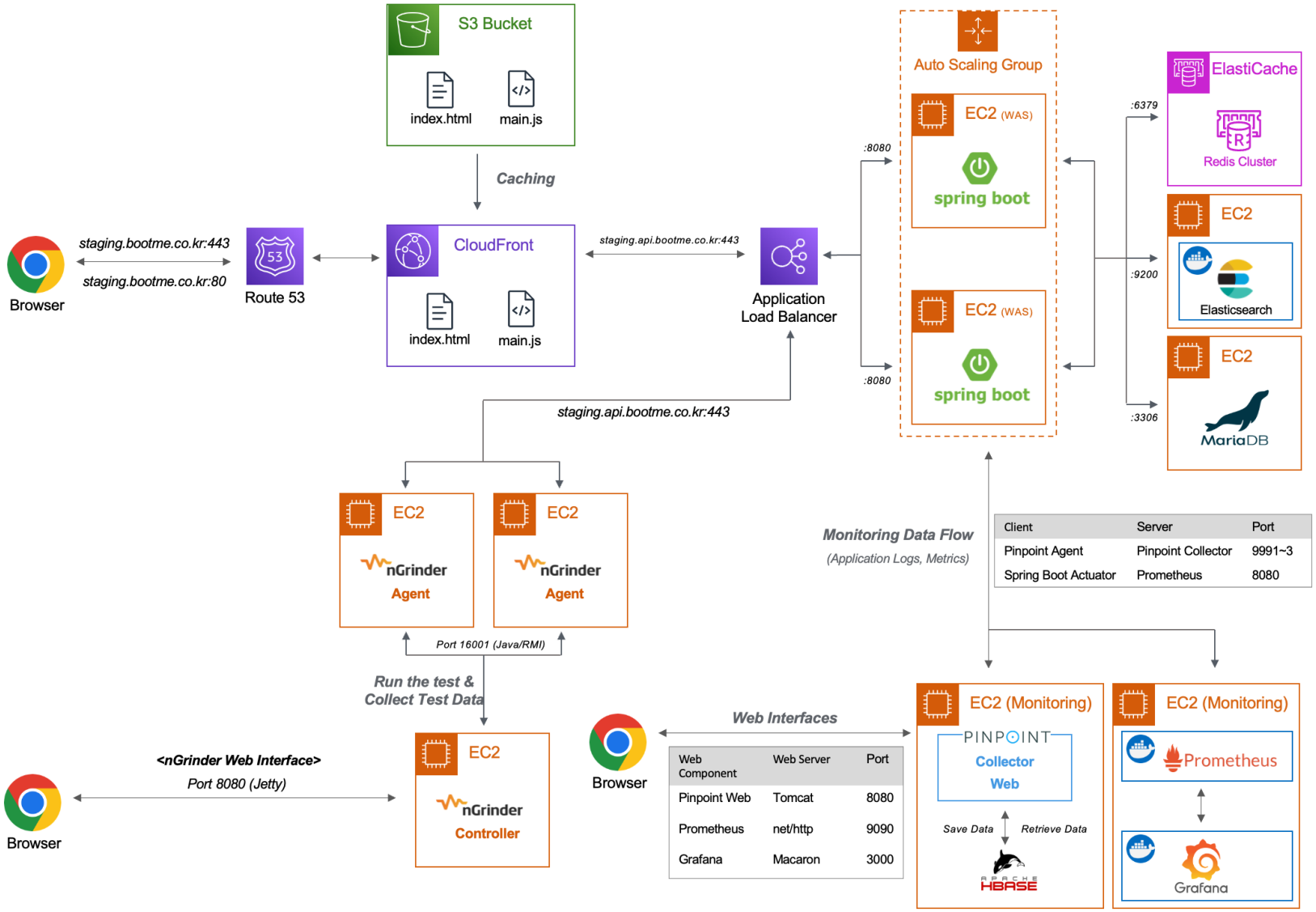
- EC2 Instance 유형 (WAS) 결정
- 로드 밸런서 도구 결정
- WAS 배포 컴퓨팅 서비스
- AWS RDS vs EC2에 DB 설치

관련 트러블 슈팅

- ASG 스케일 아웃시 AppSpec 실행 문제
- 배포 자동화 중 환경변수 사용 문제
- 정적 호스팅시 URL 입력으로 페이지 이동 문제

관련 개발 문서: 1.0 배포

배포 | 스테이징 환경 아키텍처



스테이징 환경 구성 설명

- API 성능 테스트 진행 위해 구성
- ‘staging’ 접두어를 추가한 서브 도메인 사용
- nGrinder 작동 위한 Controller, Agent 서버 운영 (작업 문서)
- DB 설정을 일관되게 관리하기 위해 Flyway 사용 (작업 문서)
- 서버 재시작 시 프로그램 실행 자동화 (작업 문서)
- 프로비저닝 반복적인 서버에 AMI 활용 (nGrinder Agent 서버)

스테이징 환경 인스턴스 현황

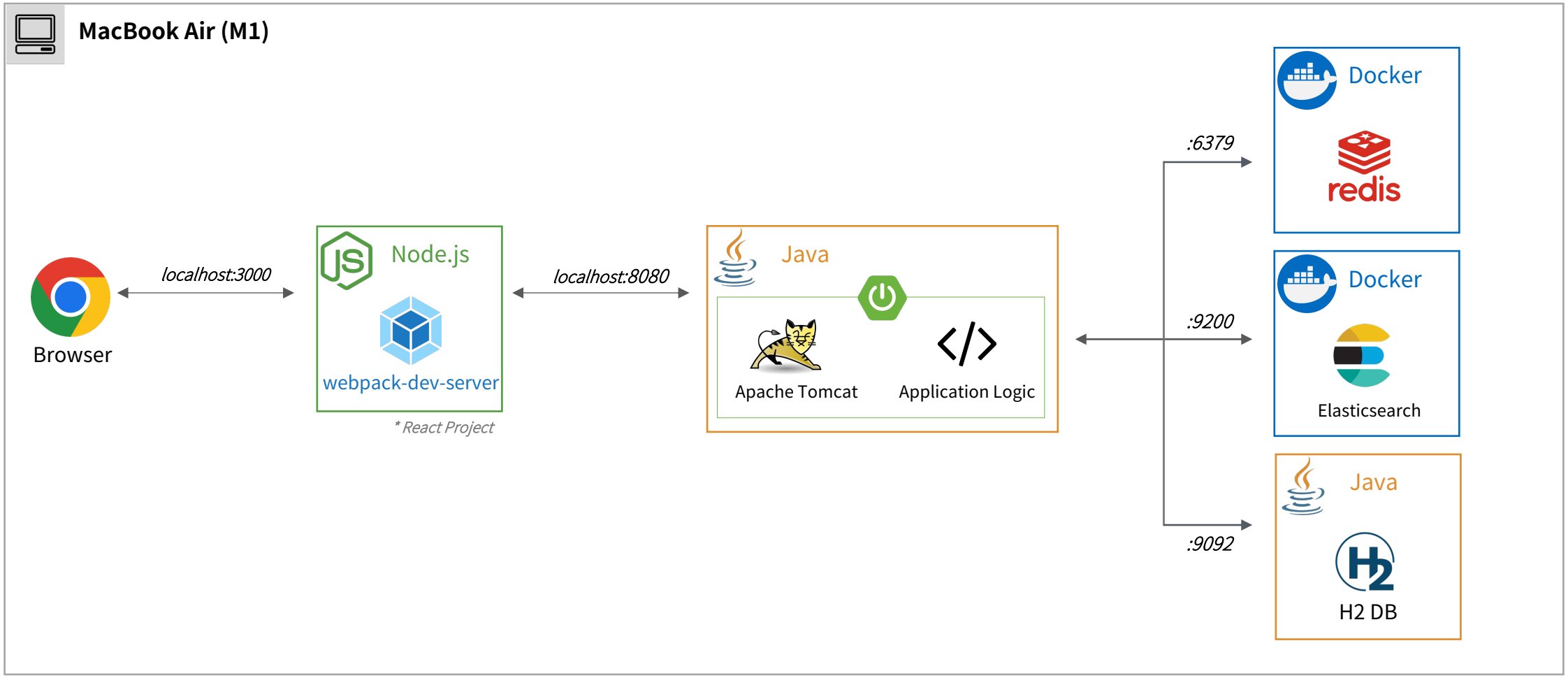
Name	인.	인스턴스 상태	인스턴스 유형
STG_WAS_AS_231022_233143	i...	실행 중	t4g.micro
STG_nGrinder_Controller	i...	실행 중	t4g.micro
STG_nGrinder_Agent	i...	실행 중	t4g.small
STG_Monitoring (Prometheus, Kibana)	i...	실행 중	t4g.nano
STG_Monitoring (Pinpoint)	i...	실행 중	t3a.medium
~~STG_MariaDB_Linux 2023 (x86)~~	i...	종지됨	t2.small
STG_MariaDB_Linux 2023 (Arm)	i...	실행 중	t4g.small

관련 개발 문서: 스테이징 환경 구성, Flyway 적용, nGrinder 사용, 프로그램 시작 자동화

배포 | 로컬 개발 환경 아키텍처

개발 중 테스트 용이성을 고려하여 설정이 간단하고 인메모리로 동작하는 H2 DB를 사용함

ElastiCache는 로컬에서 접근할 수 없기 때문에, 개발 중에는 Redis 실행하여 테스트함

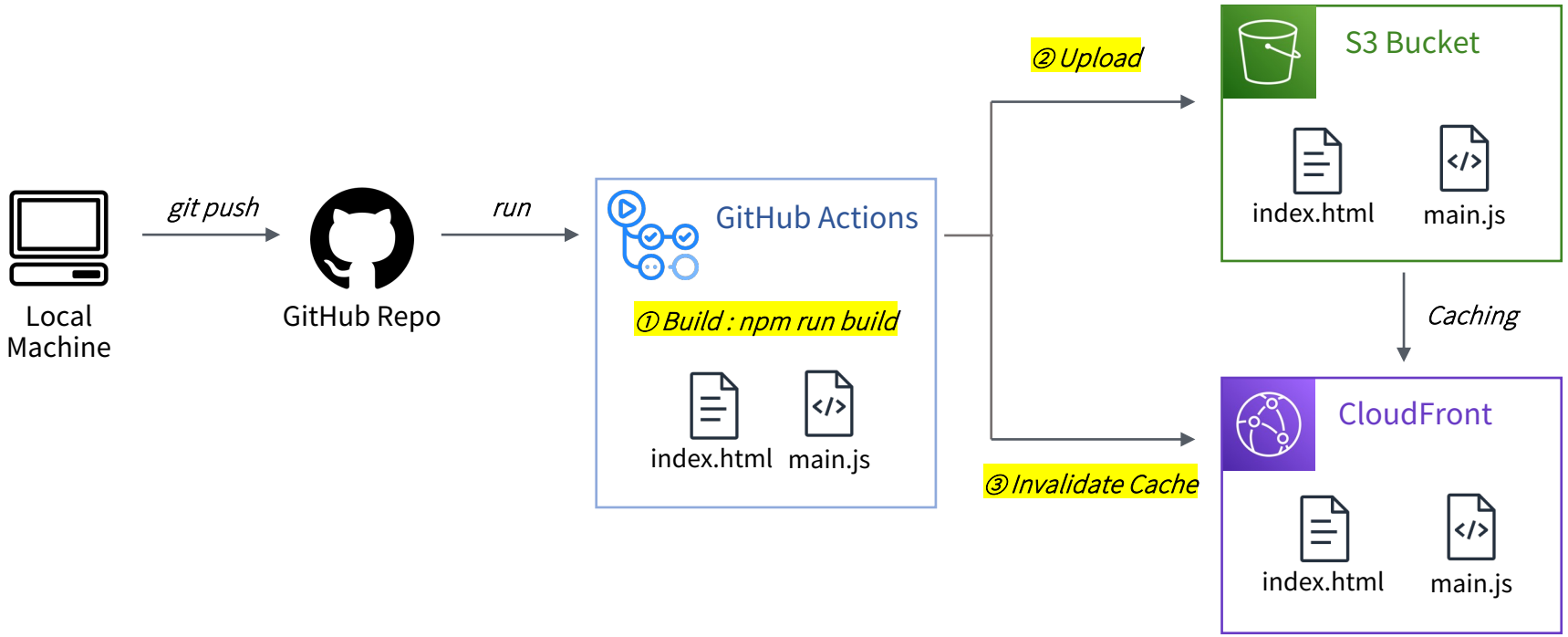


관련 개발 문서: [Webpack 개발 서버 node 프로세스 확인]

배포 | 프론트엔드 CI/CD

프론트엔드 코드 변경 후 main 브랜치 push 하면 GitHub Actions 실행 되며 HTML, JS 파일을 빌드하고 이를 S3에 업로드 함

이후 CloudFront 캐시를 무효화 하여 S3에 새로 업로드된 파일들이 캐싱되도록 함

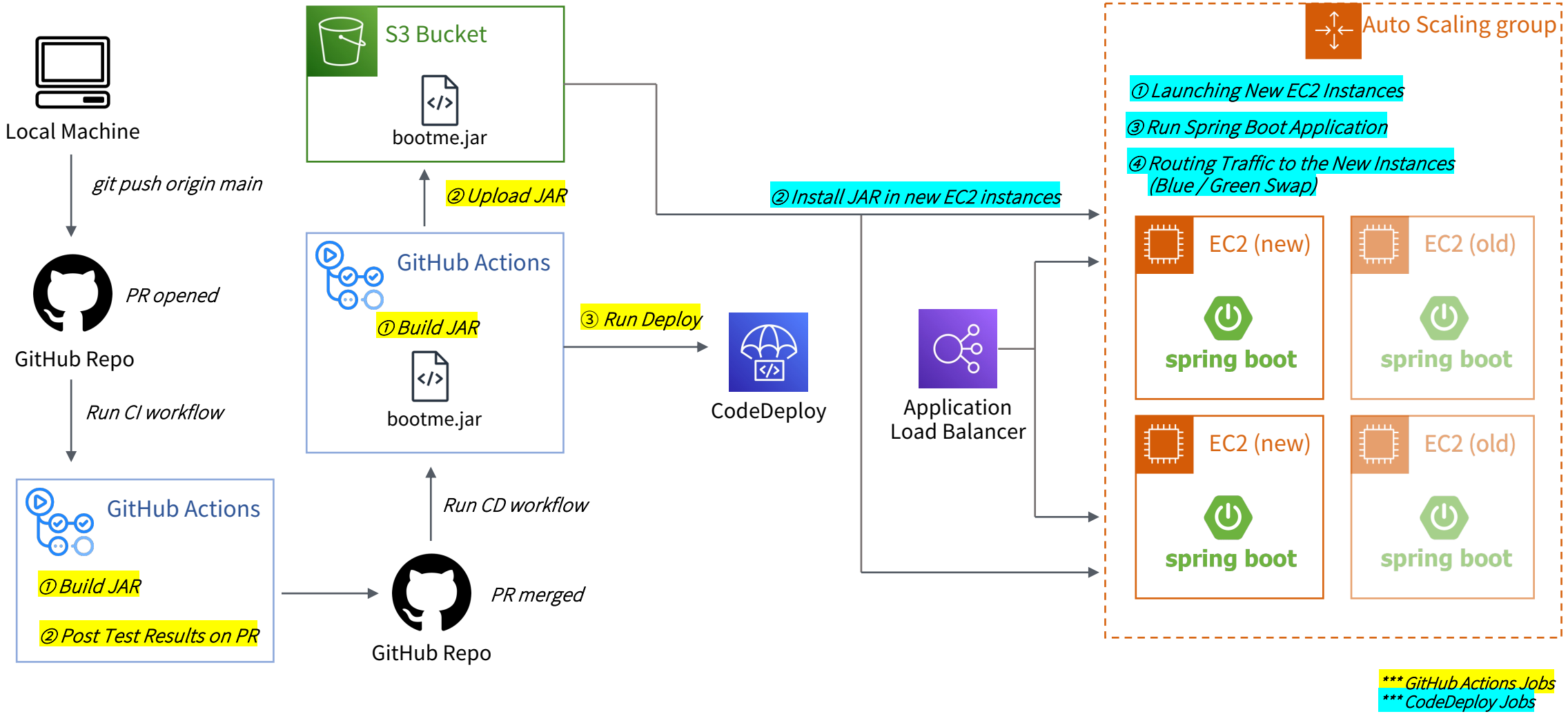


관련 개발 문서: 배포 자동화 중 환경변수 사용 문제

배포 | 백엔드 CI/CD

CI: git push 후 PR을 열면 GitHub Actions 실행 되며 자동으로 JAR 파일을 빌드하고, 생성된 빌드 파일에 포함된 테스트 결과가 PR 코멘트에 첨부됨

CD: PR merge 되면 GitHub Actions 실행 되며 JAR 파일 빌드 후 빌드된 파일을 S3에 업로드하고 CodeDeploy 배포를 실행함



목차

주요 작업 | 1.1. 게시글 조회 API 성능 개선_검색어 포함 되지 않은 요청

인덱스를 활용해 게시글 조회 API의 응답 시간을 1,210ms 에서 360ms 까지 단축 했으며(게시글 100만개 기준), 해당 API에서 발생하던 N+1 문제를 해결함
이후 Elasticsearch 사용하여 해당 API의 응답시간을 58ms 까지 단축함

게시글 조회 API 기본 정보 (검색어 미포함)

<게시글 조회 요청 URL>

GET /posts?sort=likes&page=1&size=25

<post 테이블>

<실행 쿼리 1>

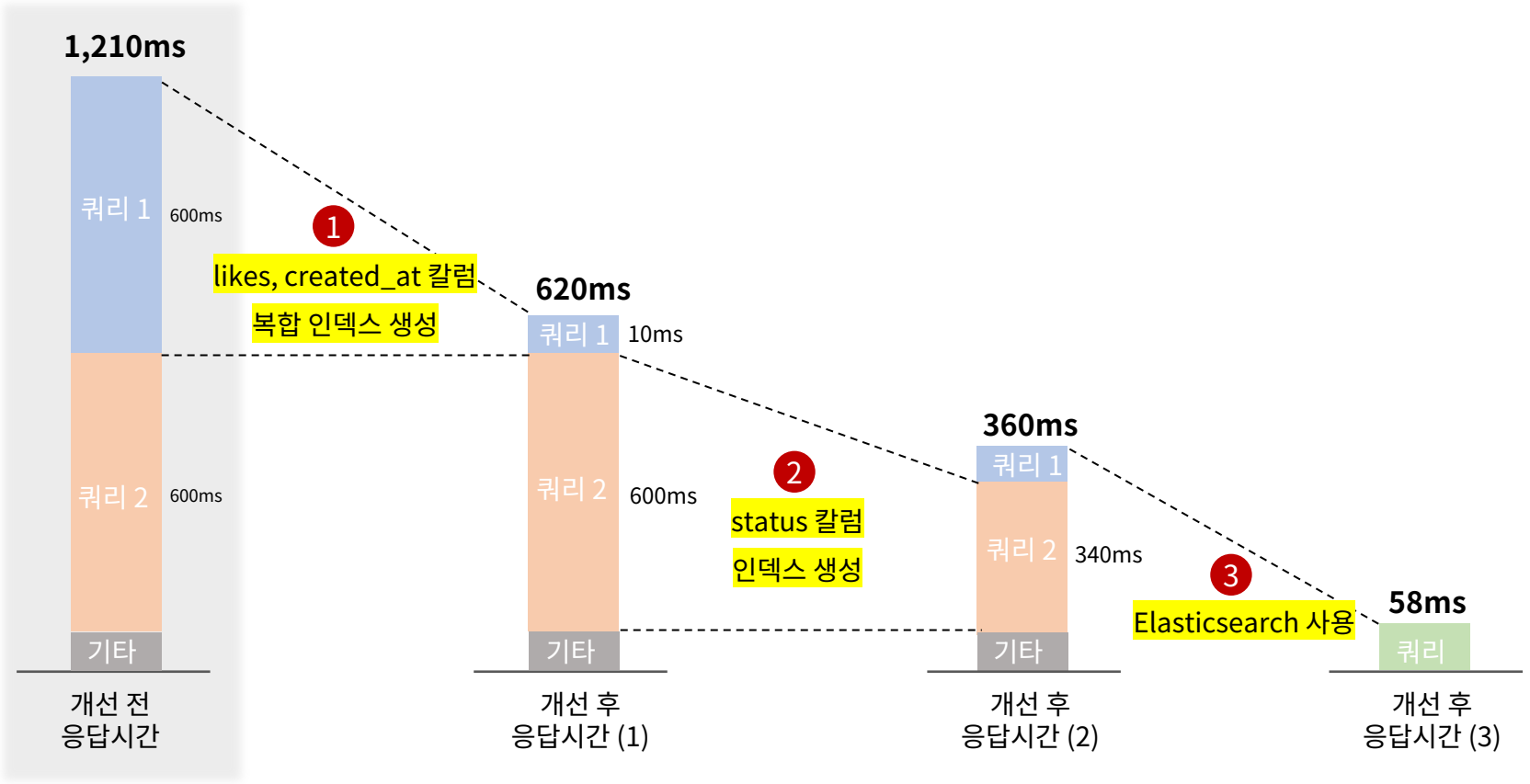
```
SELECT
  post_id, ···
FROM
  post
WHERE
  status = 'DISPLAY'
ORDER BY
  likes DESC,
  created_at ASC
LIMIT 0, 25;
```

<실행 쿼리 2>

```
SELECT
  COUNT(post_id)
FROM
  post
WHERE
  status = 'DISPLAY';
```

게시글 조회 API 응답 시간 개선 과정 (검색어 미포함)

** 게시글 100만개 기준*



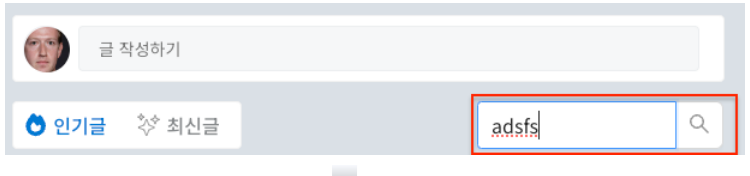
관련 개발 문서: 게시글 조회 성능 최적화 (1), (2), (3), (4), (5), (6)

주요 작업 | 1.2. 게시글 조회 API 성능 개선_검색어 포함된 요청_FULLTEXT 인덱스

인덱스 사용 이후에도 검색어가 포함된 요청은 검색어에 따라 응답 시간이 크게 길어질 수 있었음

FULLTEXT 인덱스 사용하여 검색어가 포함된 요청의 응답 시간을 약 10ms 수준으로 단축함 (응답 데이터 수가 매우 많지는 않은 경우)

검색어 포함시 지연 문제 (LIKE 연산자의 한계)



GET /posts?**search=adsfs**&sort=likes&page=1&size=25

```
[statement] | 12124 ms | com.bootme.post.repository.PostRepositoryProxy
    select
        p1_0.post_id,
        중략...
    where
        (
            p1_0.topic like '%adsfsdfafadsf%' escape '!'
            or p1_0.title like '%adsfsdfafadsf%' escape '!'
            or p1_0.content like '%adsfsdfafadsf%' escape '!'
        )
        and p1_0.status='DISPLAY'
```

DB 저장된 게시글 100만개 기준

- 데이터를 빠르게 찾기 어려운 검색어는 스캔 범위가 넓어지며 쿼리 실행 시간이 크게 길어짐
- LIKE 연산자 사용하여 문자열 검색 시, 인덱스의 활용이 제한되어 일반 인덱스 사용한 성능 개선은 어려움

FULLTEXT 인덱스 사용하여 응답 시간 개선 (작업 문서)

<FULLTEXT 인덱스 사용 쿼리>

```
SELECT
  post_id, ···
FROM
  post
WHERE
  MATCH (topic, title, content) AGAINST ('테스트')
  and status = 'DISPLAY'
ORDER BY
  likes DESC,
  created_at ASC
LIMIT 0, 25;
```

- 게시글 조회 로직을 분기하여 검색 쿼리가 포함된 요청의 경우 FULLTEXT 인덱스를 사용하는 MATCH AGAINST 문이 사용되도록 함 (쿼리 메서드)

<쿼리 실행시간 비교>

검색 쿼리	응답 데이터 수	쿼리 실행 시간 (LIKE 연산자 사용)	쿼리 실행 시간 (FULLTEXT 인덱스 사용)
"게시글"	100만	10 ms	5,948 ms
"important"	21,940	19 ms	210 ms
"테스트"	4	10,732 ms	9 ms
"abcde"	0	13,048 ms	8 ms

DB 저장된 게시글 100만개 기준

- 기존의 LIKE 연산자 사용하는 쿼리에 비해 쿼리 실행 시간이 크게 줄어 들었음

** 응답 데이터 수가 많은 경우 제외*

관련 개발 문서: 게시글 조회 성능 최적화 (5): Full-text Search 적용, Hibernate 6에서 커스텀 함수 추가, count 저장용 별도 테이블 사용

주요 작업 | 1.3. 게시글 조회 API 성능 개선_검색어 포함된 요청_Elasticsearch

Elasticsearch 사용하여 FULLTEXT 인덱스 사용시 단점을 극복하고 일관된 검색 성능 제공할 수 있도록 함

FULLTEXT 인덱스 사용시 단점

쿼리 실행시간 비교 (LIKE vs. MATCH AGAINST)			
검색 쿼리	응답 데이터 수	쿼리 실행 시간 (LIKE 연산자 사용)	쿼리 실행 시간 (MATCH AGAINST 사용)
“게시글”	100만	10 ms	5,948 ms
“important”	21,940	19 ms	210 ms
“테스트”	4	10,732 ms	9 ms
“abcde”	0	13,048 ms	8 ms

- LIKE 연산자 사용하는 쿼리는 LIMIT 0, 25 조건 있기 때문에 처음으로 일치하는 25개의 결과를 찾으면 즉시 쿼리 실행을 중단하여 검색어에 따라 빠른 응답도 가능
- 그러나 FULLTEXT 인덱스를 사용하는 쿼리는 LIMIT 조건 있더라도 **일치하는 결과를 모두 찾은 다음, 결과들을 관련성 점수별로 정렬하는 작업이 우선적으로 수행**됨
- 그렇기 때문에 **응답 데이터의 양 많은 경우 매우 느림**

DB 저장된 게시글 100만개 기준

Elasticsearch 사용하여 응답 시간 개선

검색 쿼리	검색된 데이터 수	응답시간		
		1. LIKE 연산자 사용 (MariaDB)	2. FULLTEXT 인덱스 사용 (MariaDB)	3. Elasticsearch 사용
“게시글 "	100만	1,207 ms	10,916 ms	**213 ms**
“important”	21,940	2,699 ms	418 ms	**74 ms**
“important school”	476		976 ms	**133 ms**
“테스트”	4	10,304 ms	35 ms	**125 ms**
“abcde”	0	10,054 ms	26 ms	**90 ms**

DB 저장된 게시글 100만개 기준

- Elasticsearch 사용시 검색어에 따라 응답 시간 편차가 크지 않아 사용자에게 일관된 검색 성능 제공할 수 있음
- 다양한 언어의 텍스트 분석 기능을 내장하고 있어 복잡한 검색 요구사항에도 유연하게 대응 가능함
- RDB에서 검색 부하를 가져가 처리함으로써 RDB의 부하를 줄일 수 있음

관련 개발 문서: 게시글 조회 성능 최적화 (6): Elasticsearch 적용, MariaDB ↔ Elasticsearch 동기화 문제

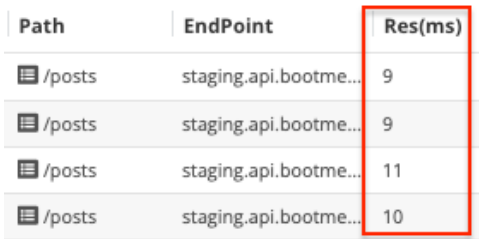
주요 작업 | 1.4. 게시글 조회 API 성능 개선_캐싱

게시글 조회 API에 Redis 캐싱 적용하여 응답 시간 약 10ms 수준으로 단축함 (캐시 히트시)

캐싱 관련 의사 결정

결정 대상	선택한 옵션	주요 결정 요인
로컬 vs. 글로벌	글로벌 캐시	분산 환경에서 서버간 데이터 일관성을 보장할 수 있고, 수평 확장 쉽게 가능하므로 대규모 데이터 처리에 보다 적합
Managed 정도	높음 (Amazon ElastiCache)	EC2에 직접 캐싱 솔루션 설치하여 운영시 관리 복잡도 높음 (Failover, Replica, 샤딩, 보안, 모니터링 등)
캐싱 솔루션	Redis	다양한 데이터 타입 등 Memcached가 지원하지 않는 여러 기능 지원 및 더 좋은 스프링 부트 호환성
캐싱 전략	Look-Aside + Write Around	Read-heavy 작업에 적합하며 데이터 정합성 보장 가능
자료 구조 (게시글 저장시)	Key: String, Value: String (JSON)	게시글 값 직렬화에 용이하고, 게시글의 구조나 속성이 변경되더라도 JSON 문자열 형태로 저장하면 유연하게 대응 가능
Java Redis Client	Lettuce	비동기 요청 처리로 Jedis에 비해 압도적인 성능

캐시 히트시 응답시간



Path	EndPoint	Res(ms)
/posts	staging.api.bootme...	9
/posts	staging.api.bootme...	9
/posts	staging.api.bootme...	11
/posts	staging.api.bootme...	10

Pinpoint APM

캐시 키 형태

```
127.0.0.1:6379> keys *
 1) "posts::none:none:likes:2:25"
 2) "posts::\xec\x9e\x90\xec\x9c\xa0:none:createdAt:1:25"
 3) "posts::none:none:createdAt:2:25"
 4) "posts::none:none:likes:1:25"
```

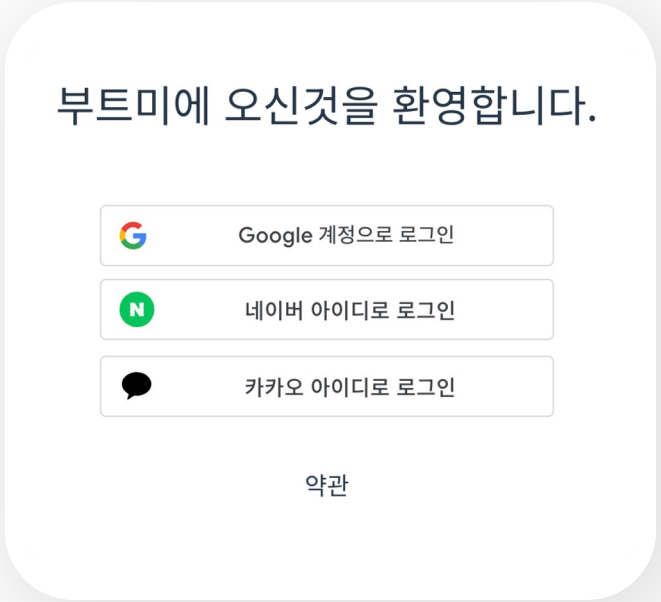
캐싱 개선 필요 사항

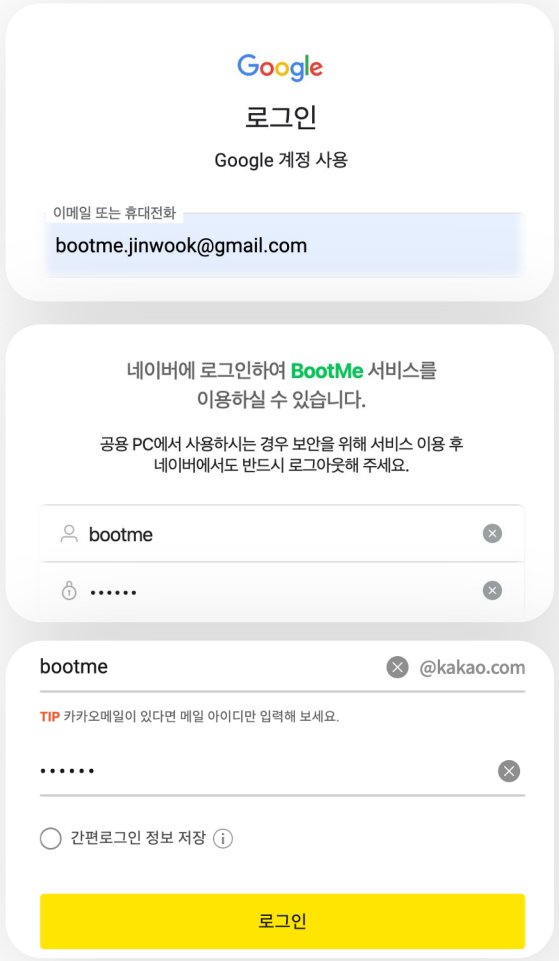
- **Cache Warming 적용**
  - 서비스 초기에 트래픽 급증시 대량의 Cache Miss 발생 방지 위해 적용 필요
  - 게시글 조회 조건별 첫 3페이지 미리 캐싱 고려
- **캐시 삭제 고도화**
  - 현재 게시글 CUD 발생시 전체 게시글 캐시를 삭제함
  - 게시글 CUD 발생시 삭제가 필수적인 캐시만 삭제되도록 구현해야함
- **캐시 키 가독성 개선**
  - 캐시 키 구조에서 한글 문자열이 인코딩된 형태로 표시되어 가독성 낮은 문제 개선 필요

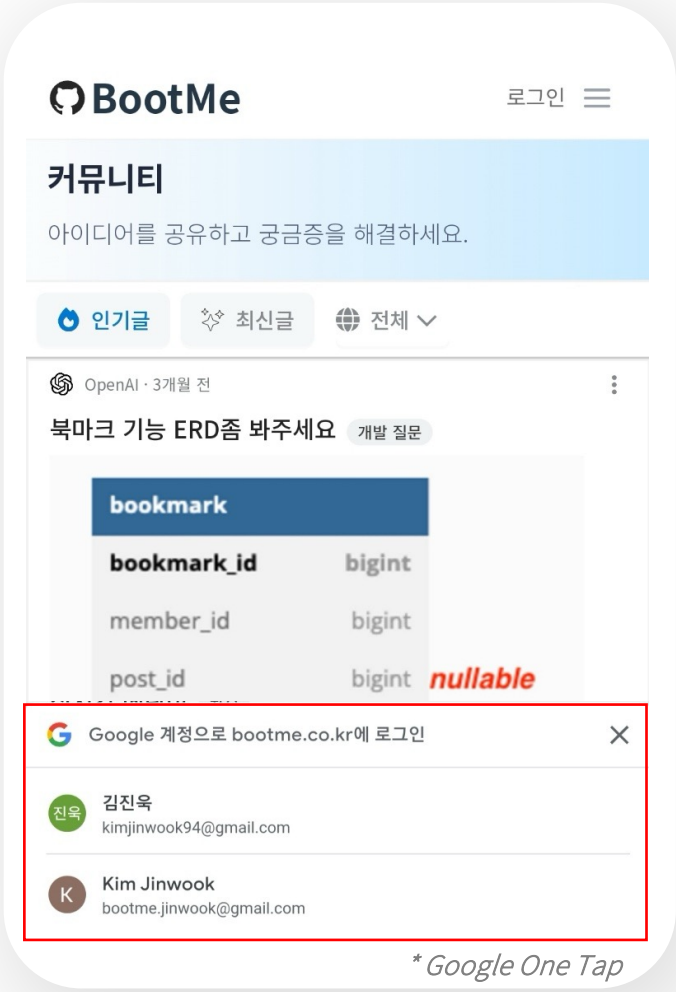
관련 개발 문서: 게시글 조회 성능 최적화 (4): Redis 적용, 로컬 캐시 vs. 글로벌 캐시

주요 작업 | 2.1. 인증 구현_요약

구글, 네이버, 카카오 로그인 등 소셜 로그인을 구현했으며, JWT를 사용하여 유저의 로그인 상태를 검증하고 유지하도록 구현함
Google One Tap 적용하여 보다 편리한 소셜 로그인 가능하도록 함



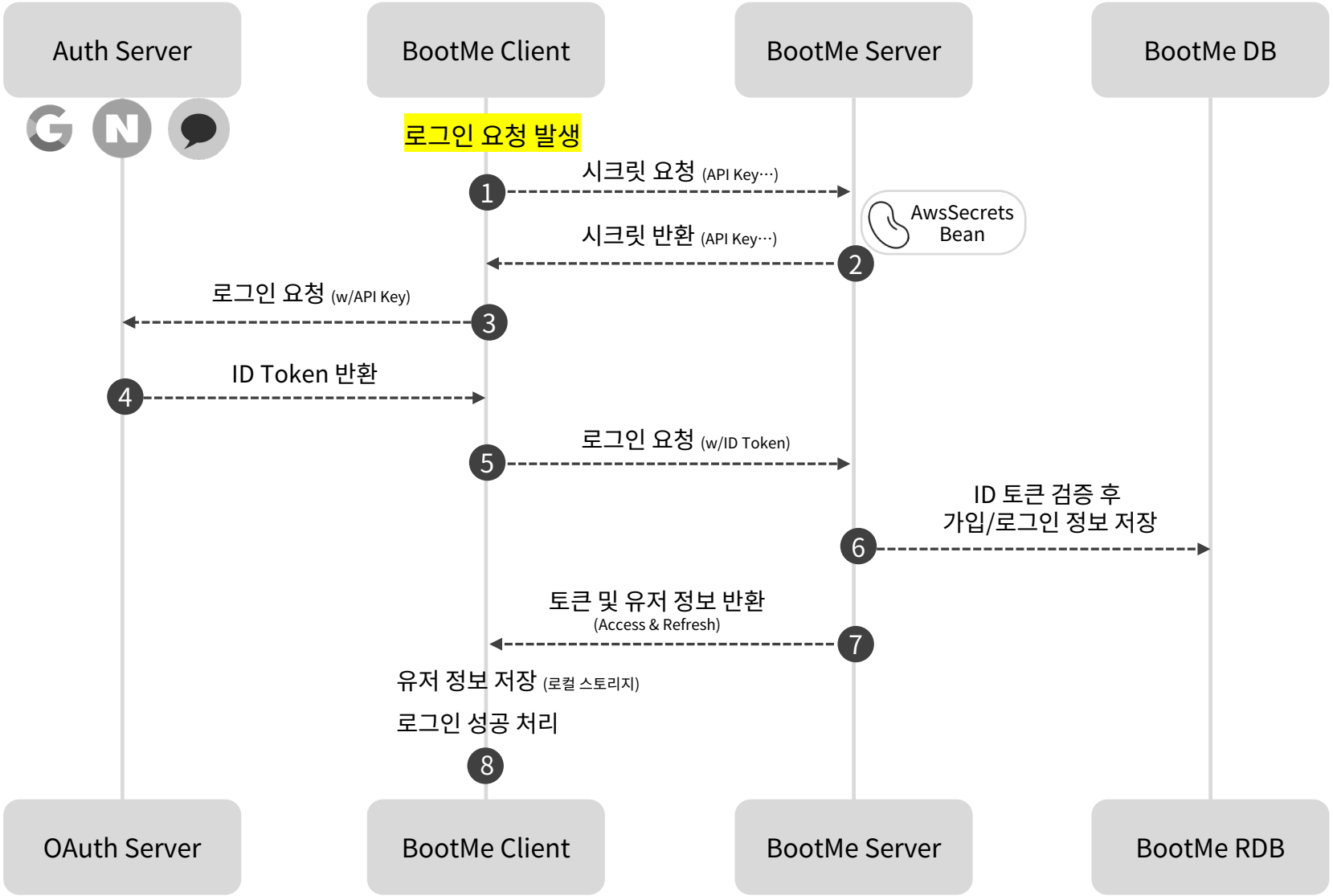




관련 개발 문서: 로그인, 회원가입 1차 구현, 토큰 재발급 구현, 토큰 검증

관련 코드: AuthService, TokenValidationInterceptor

주요 작업 | 2.2. 인증 구현_로그인 성공 흐름



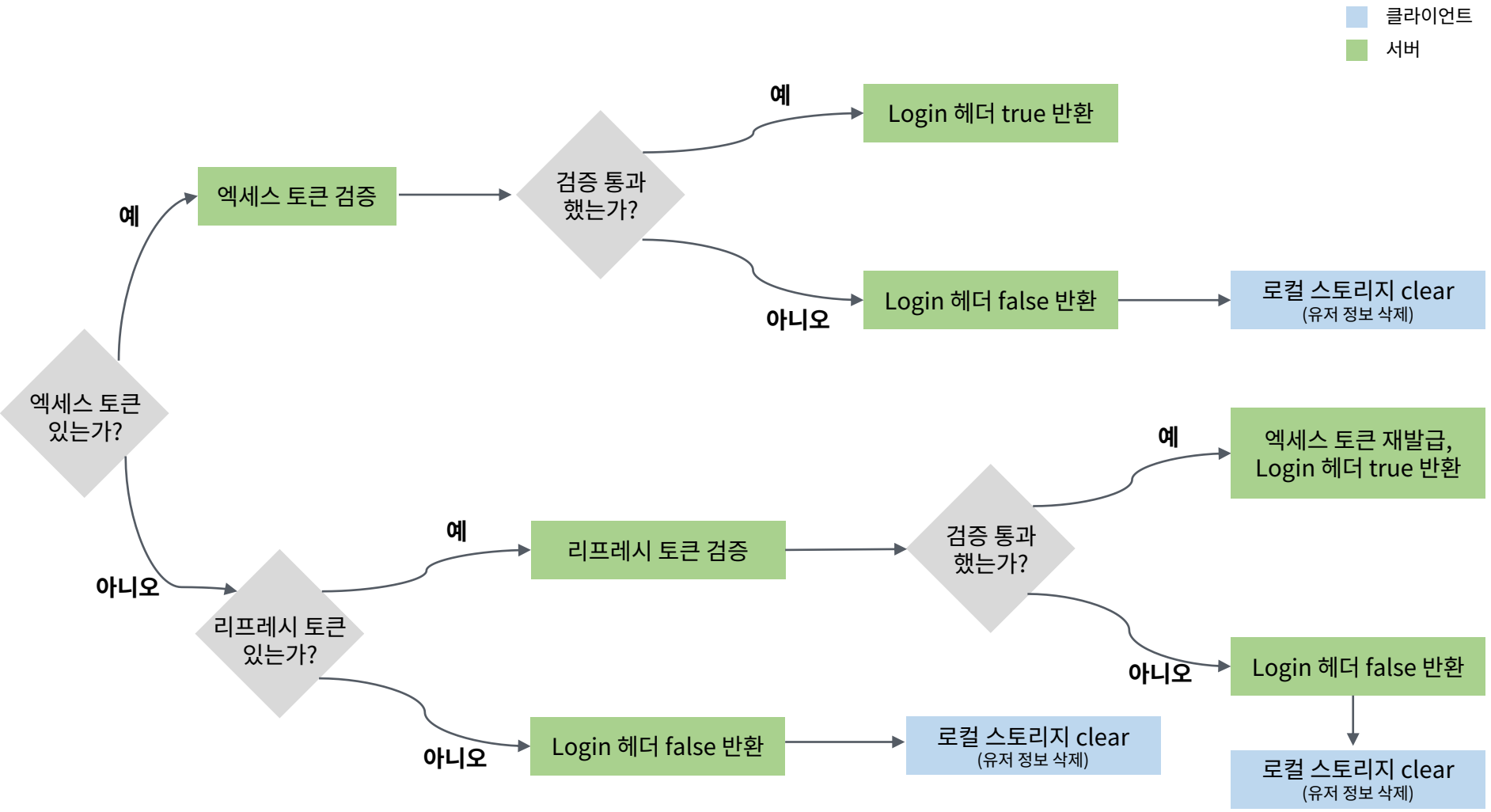
로그인 성공 흐름 부가 설명

- ❶~❷: 시크릿 값은 백엔드에서 AWS Secrets Manager 호출하여 받아와서 Bean으로 관리함
- ❻: ID 토큰의 발급자, Audience, 발행 시간, 만료 시간, 서명을 검증함 (해당 코드)
  ** 네이버 로그인은 OICD가 아닌 OAuth2.0 기반이기 때문에 ID 토큰을 반환하지 않고 유저 정보를 직접 반환하므로 토큰 검증 과정 없음*
- ❼: 엑세스 토큰과 리프레시 토큰을 생성해 쿠키로 반환하고 헤더 UI에서 사용해야 하는 유저 정보(닉네임, 이메일 등)를 응답 바디에 반환함

관련 개발 문서: 로그인, 회원가입 1차 구현, 토큰 재발급 구현, 토큰 검증

관련 코드: AuthService, TokenValidationInterceptor

주요 작업 | 2.3. 인증 구현_토큰 재발급 흐름



토큰 재발급 흐름 부가 설명

- 엑세스 토큰과 리프레시 토큰을 사용하여 유저가 로그인 상태를 일정 기간 유지할 수 있도록 구현함
- 이 로직은 스프링 MVC의 핸들러 인터셉터에 구현되어 모든 HTTP 요청 처리에 앞서 호출됨
- 프론트엔드 인터셉터에서는 모든 응답의 'Login' 헤더를 확인하여 'Login' 헤더 값이 'false'인 경우 로컬스토리지에 저장된 유저 정보를 삭제함
- 토큰 만료 시간
  - 엑세스 토큰: 1시간
  - 리프레시 토큰: 30일

** 리프레시 토큰이 탈취될 경우 큰 보안 위험 요소가 있으므로, 리프레시 토큰을 서버 측에서 무효화 할 수 있도록 추가 구현 필요 (참고 문서)*

관련 개발 문서: 로그인, 회원가입 1차 구현, 토큰 재발급 구현, 토큰 검증, 리프레시 토큰 보안 개선

관련 코드: AuthService, TokenValidationInterceptor

주요 작업 | 2.4. 인증 구현_인증 정보 사용 방식 및 데이터 저장 구조

@Login 커스텀 어노테이션을 활용해서 메서드에서 사용자 인증 정보를 쉽게 사용할 수 있도록 함
데이터 정합성을 보장하기 위해 사용자 인증 정보를 두 개의 테이블이 분리하여 저장함

사용자 인증정보 사용 방식: @Login 커스텀 어노테이션 활용

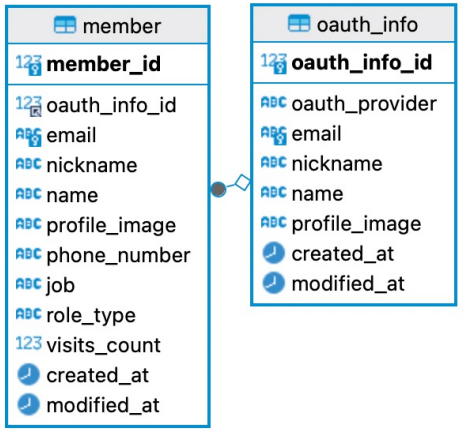
```java
@GetMapping("/comments/{id}")
public ResponseEntity<CommentResponse> findComment(@Login AuthInfo authInfo,
                                                   @PathVariable Long id) {
    CommentResponse commentResponse = commentService.findComment(authInfo, id);
    return ResponseEntity.ok(commentResponse);
}

@Transactional(readOnly = true)
public CommentResponse findComment(AuthInfo authInfo, Long id) {
    authService.validateLogin(authInfo);
    Comment comment = getCommentById(id);
    comment.assertAuthor(authInfo.getMemberId());

    return CommentResponse.of(comment);
}
```

- @Login 커스텀 어노테이션과 ArgumentResolver 사용해서 메서드에서 사용자 인증정보를 쉽게 사용할 수 있도록 함
- AuthenticationArgumentResolver는 @Login 어노테이션이 붙은 메서드 파라미터를 인식하고, 요청에 포함된 'accessToken' 쿠키를 통해 사용자의 인증 정보를 추출함
- 추출된 사용자 인증 정보는 AuthInfo 객체에 담겨 메서드로 전달

데이터 정합성 위해 인증 정보를 두 개 테이블로 분리 저장



- 변경 가능성이 있는 데이터는 member 테이블에 저장하고, 고정적인 데이터인 최초 로그인시의 정보는 oauth_info 테이블에 저장함
- 회원은 프로필 변경 페이지에서 닉네임, 직업 등의 개인 정보를 변경할 수 있음
- 개인 정보를 변경한 후에도 최초 로그인시의 정보를 보존하기 위해 변경 내용은 member 테이블에만 적용됨

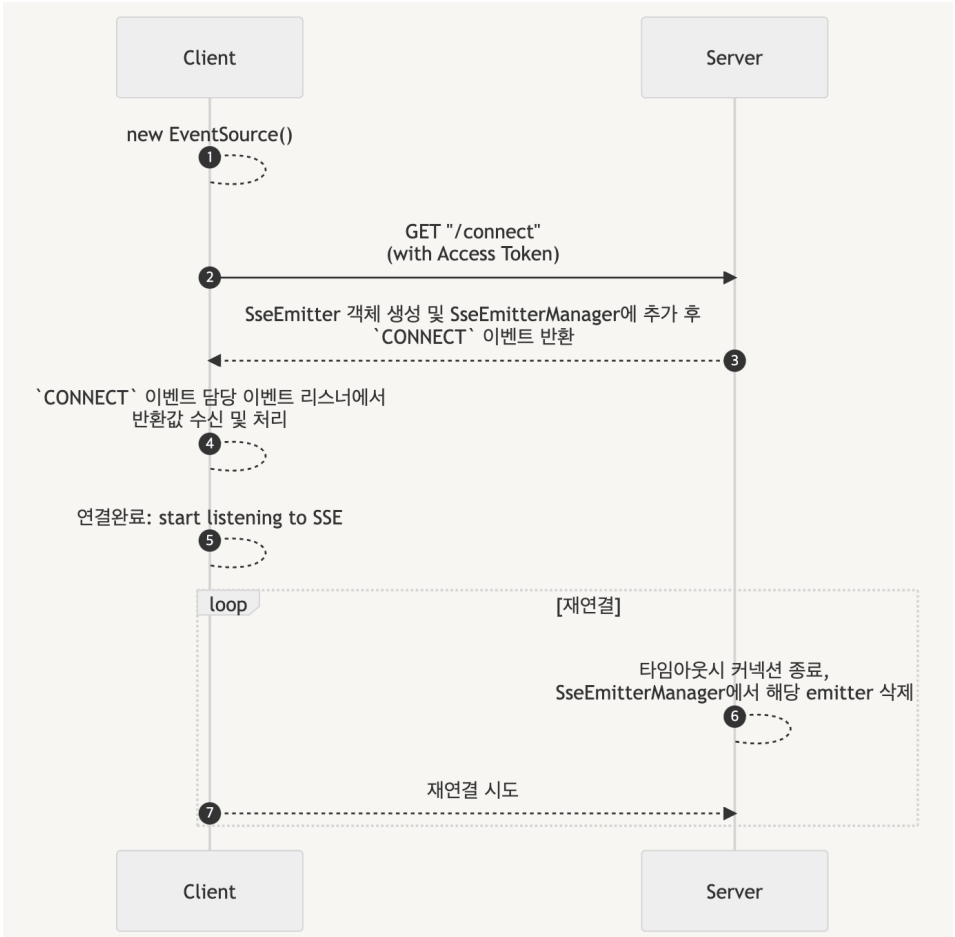
관련 코드: AuthenticationArgumentResolver, Member, Oauth_info

주요 작업 | 3.1. 실시간 알림 구현_요약 및 처리 과정

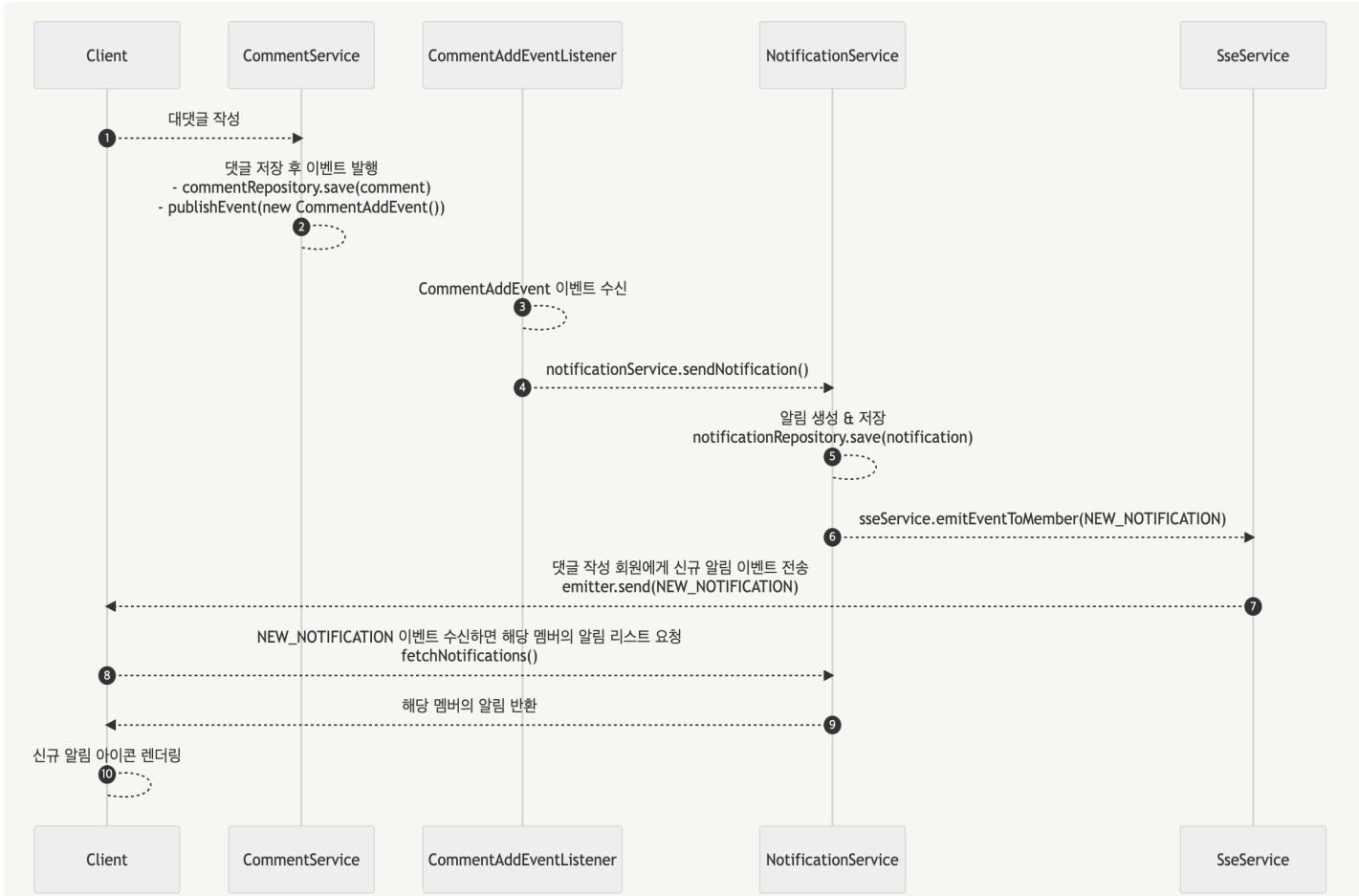
SSE 활용하여 실시간 알림을 구현하여, 새로운 알림이 생기면 페이지를 새로고침 하지 않아도 신규 알림 아이콘이 랜더링 됨

네트워크 부하를 최소화 하면서 단방향인 실시간 통신 구현하기 위해 다양한 실시간 통신 방법 중 SSE 선택함

SSE 커넥션 연결 과정



실시간 알림 처리 과정 (ex. 댓글 작성 상황)



관련 개발 문서: 실시간 알림 구현, 실시간 통신 방법, Connection Leak, SSE 커넥션 브라우저당 6개로 제한, 분산 환경에서 SSE 사용: sticky session

주요 작업 | 3.2. 실시간 알림 구현_Spring Event

알림을 발생시키는 작업(댓글 작성 등)과 알림 발송 작업은 본질적으로 다른 생명주기와 책임을 가지기 때문에 이벤트 기반으로 구현함

이벤트 기반으로 구현하지 않은 경우

```java
@Service
@RequiredArgsConstructor
public class CommentService {

    private final NotificationService notificationService;

    @Transactional
    public Long addComment(AuthInfo authInfo, Long postId,
                           CommentRequest commentRequest) {
        ...
        Comment savedComment = commentRepository.save(comment);
        ...
        notificationService.sendNotification(savedComment);

        return savedComment.getId();
    }

}
```

강한 결합 (높은 의존성)

- 코드가 간단하고 직관적이라는 장점 있음
- 그러나 서비스 클래스간 직접적인 의존성이 생겨 시스템이 확장될 때 유지 보수에 어려움을 겪을 수 있음
- 또한 CommentService.addComment()의 단위 테스트를 작성할 때 NotificationService의 로직도 함께 실행되어 독립적인 테스트 작성도 어려움

이벤트 기반 구현 (현재 구현)

```java
@Service
@RequiredArgsConstructor
public class CommentService {

    private final ApplicationEventPublisher eventPublisher;

    @Transactional
    public Long addComment(AuthInfo authInfo, Long postId,
                           CommentRequest commentRequest) {

        Comment savedComment = commentRepository.save(comment);
        ...
        eventPublisher.publishEvent(new CommentAddEvent(this, comment.getId()));

        return savedComment.getId();
    }
```

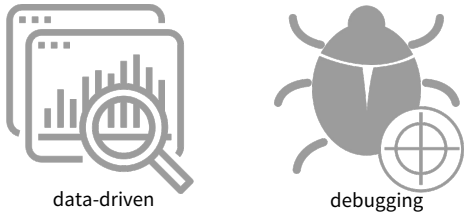
- 서비스 간 결합도 낮아 변경에 보다 유연함
  ex) NotificationService 변경 사항은 CommenService에 영향을 주지 않음
- 특정 이벤트(ex.댓글 작성)에 대한 추가적인 로직이 필요할 때 해당 이벤트의 리스너만 수정하면 되므로 기능 확장이 유연하게 이루어질 수 있음
- 낮은 의존성으로 독립적인 테스트 작성에 보다 용이함

관련 코드: CommentAddEventListener, NotificationService, SseService

주요 작업 | 4.1. 모니터링 시스템 구축_요약

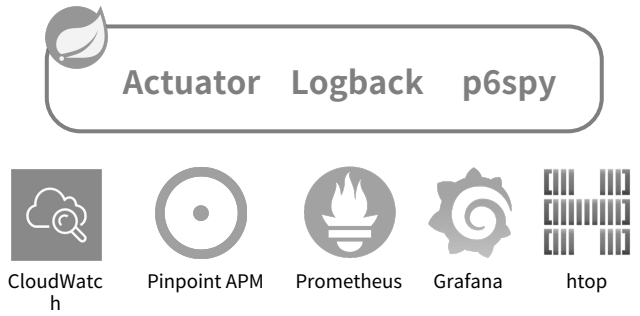
효과적인 디버깅을 위해 모니터링 시스템을 구축하였으며, 데이터 수집 및 시각화에 대한 흥미도 로깅 및 모니터링 시스템 구축의 동기가 됨

모니터링 시스템 구축 동기



- 다양한 환경에서 에러를 마주하며 효과적인 **디버깅을 위해** 각 상황에 맞는 에러 감지 및 분석 시스템의 필요성 인지함
  (ex. 배포 환경 스프링 부트 앱에서 에러 발생시
  EC2 콘솔에 접속하여 로그 확인 vs. CloudWatch, Pinpoint APM으로 로그 확인)
- **데이터 기반 의사결정**에 관심을 가지며, **데이터 수집과 시각화**의 중요성을 인지하게 됨. 그 과정에서 **로깅, 모니터링에 대한 흥미**를 가지게 됨

다양한 로깅/모니터링 도구 사용



사용 도구	목적 및 활용 방식
Logback	어플리케이션 로깅을 위해 사용하며, 로그 파일을 CloudWatch로 전송하여 조회
p6spy	JPA 사용 중 실행 되는 쿼리를 정확히 확인하기 위해 개발 환경에서 사용
Spring Actuator	1. 로드 밸런서 헬스 체크 엔드포인트 제공 2. JVM 메트릭 엔드포인트 제공 (⟹ Prometheus & Grafana 시각화)
Pinpoint APM	실시간 디버깅 및 응답 시간 모니터링 위해 사용
htop	Java 메모리 사용량 모니터링 ⟹ JVM 힙메모리 설정, WAS 인스턴스 유형 선택

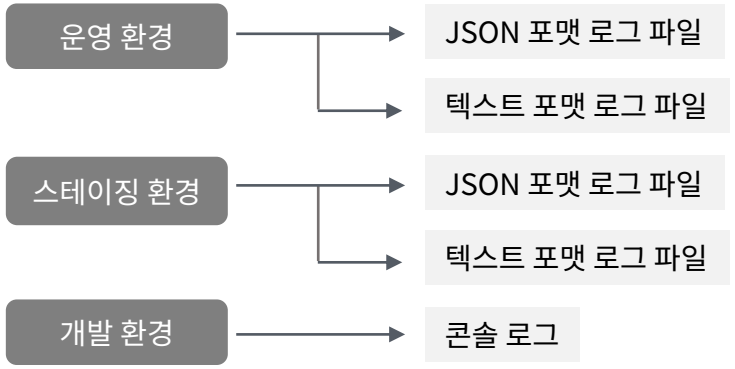
관련 개발 문서: 모니터링 환경 구성, 로깅 적용 (CloudWatch), SLF4J 구현체, 로그 출력 형식, 쿼리 성능 측정 방법

주요 작업 | 4.2. 모니터링 시스템 구축_어플리케이션 로깅

스프링 부트 어플리케이션의 로그는 Logback을 이용해 환경에 따라 로그 파일이 저장되거나 콘솔로 로그가 출력되도록 설정함

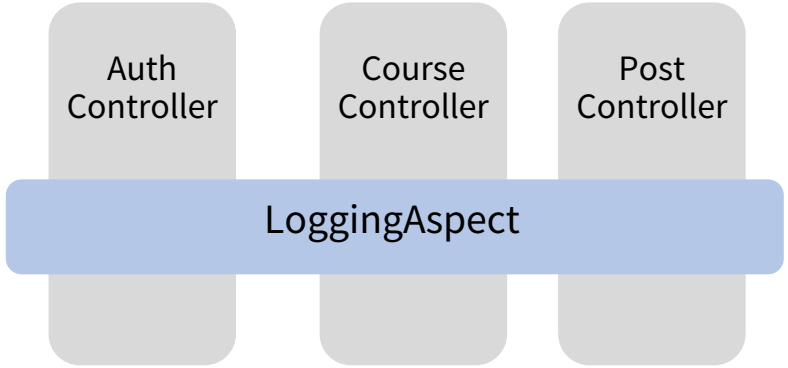
Spring AOP을 활용하여 HTTP 요청 및 응답 로그가 일관된 방식으로 기록되도록 했으며, 디버깅 활용도를 위해 로그에 최대한 많은 세부 정보를 포함했음

Logback 사용한 로깅 설정



- SLF4J 구현체 중 가장 범용적으로 사용되는 **Logback 사용**해 어플리케이션 로그 설정함
- **개발 환경은 콘솔 로그**만 출력하도록 하고, **운영/스테이징은 로그 파일**로 저장함
- 운영/스테이징 환경의 로그 파일은 **일반 텍스트 형식과 JSON 형식 두 가지로 분류**하여 저장
  - 일반 텍스트 로그: 간단한 확인 목적
  - JSON 형식 로그: 디버깅 등 자세한 분석 필요한 상황에서 가독성 높이고, 파싱과 쿼리가 용이하도록 함

Spring AOP 활용한 일관된 로깅 정책



- **Spring AOP를 활용**하여 컨트롤러의 모든 HTTP 요청과 응답에 대한 세부 정보를 **일관된 방식으로 로깅**함 (LoggingAspect.java)
- 디버깅을 위해 로그에는 HTTP 요청의 URL, 파라미터, 바디 및 HTTP 응답의 상태 코드와 바디등 많은 정보가 포함됨

  ** 응답 바디의 데이터 양이 많아서 로그 파일의 용량과 로그 전송 비용이 크게 증가하는 문제가 있음 → 추후 변경 고려*

관련 개발 문서: 모니터링 환경 구성, 로깅 적용 (CloudWatch), SLF4J 구현체, 로그 출력 형식, 쿼리 성능 측정 방법

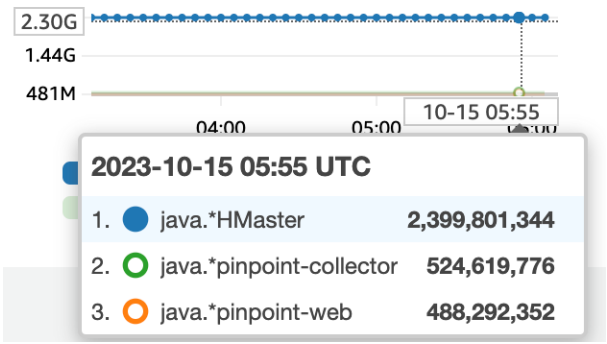
관련 코드: LoggingAspect

주요 작업 | 4.3. 모니터링 시스템 구축_CloudWatch, Pinpoint APM

어플리케이션 로그와 시스템 메트릭을 편리하게 조회하기 위해 CloudWatch를 사용함

Pinpoint APM을 통해 트랜잭션의 흐름을 명확하게 파악하고 오류가 발생한 위치를 신속하게 확인함

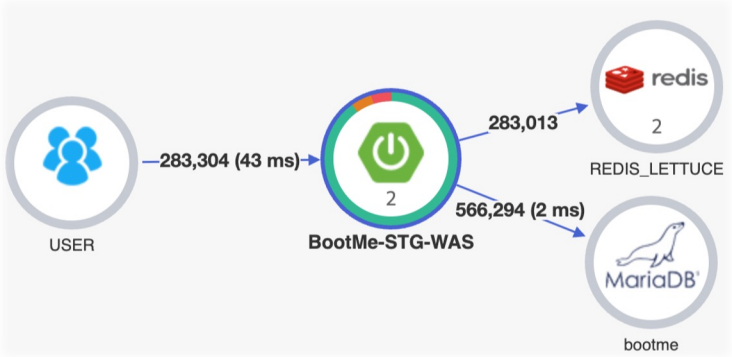
CloudWatch (어플리케이션 로그, 시스템 메트릭 조회)



** 프로세스별 메모리 사용량*

- 어플리케이션 로그나 EC2 인스턴스의 시스템 메트릭을 조회할 때 CloudWatch 사용함
- EC2 인스턴스에 설치된 CloudWatch Agent를 통해 로그 파일이나 시스템 메트릭을 CloudWatch에 전송함
- 배포 환경에서 어플리케이션 에러 발생시 EC2 콘솔에 접속해서 로그 파일을 확인하지 않고 CloudWatch에서 쉽게 조회할 수 있음
- 모니터링 서버에서 실행되는 다양한 프로세스의 실시간 메모리 사용량을 모니터링하여 메모리 배분 및 인스턴스 유형 선택에 활용함

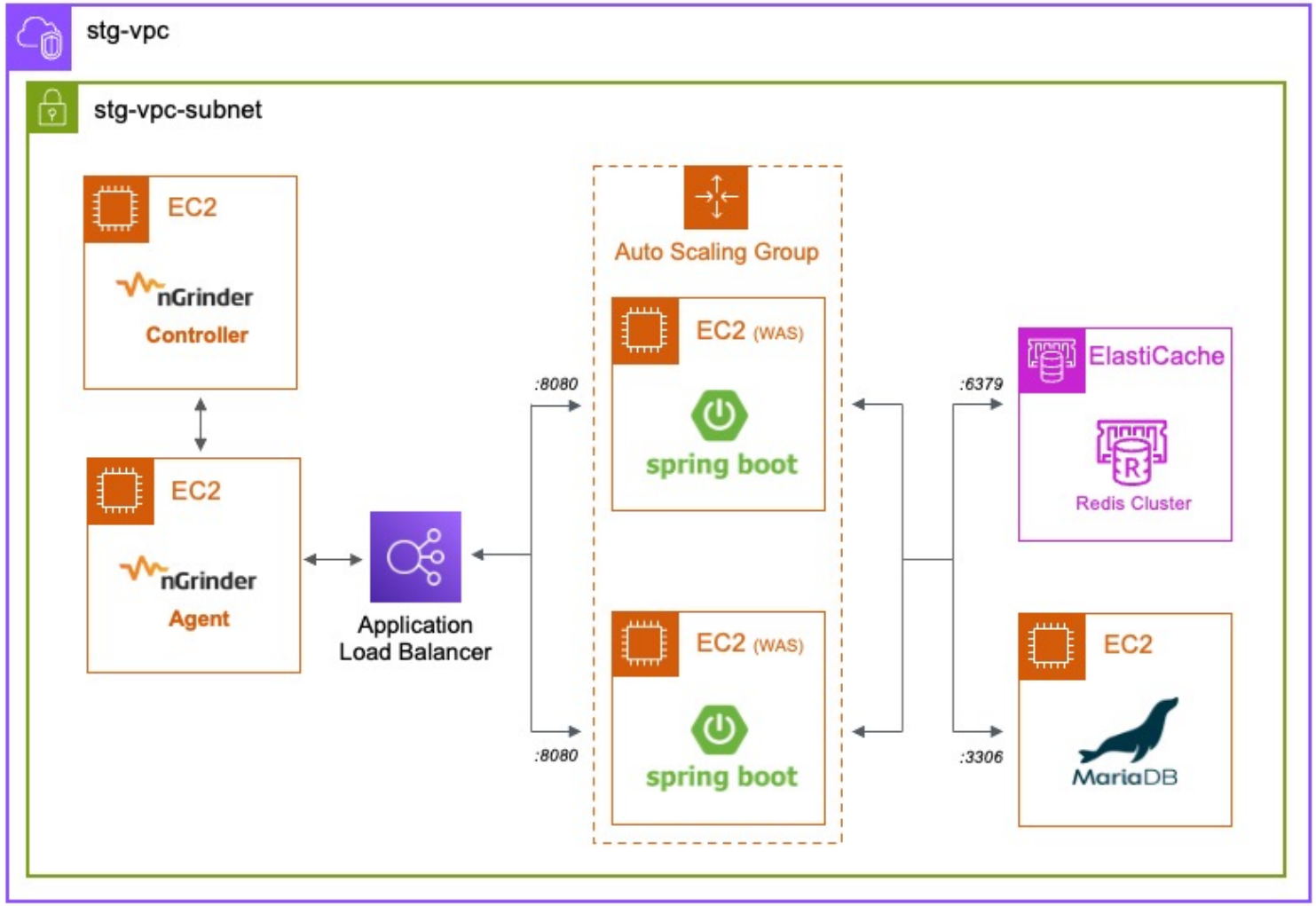
Pinpoint APM (어플리케이션 모니터링)



- 트랜잭션의 전체 흐름을 한 눈에 파악하고, 오류가 발생했을 때 해당 오류의 정확한 위치를 빠르기 확인하기 위해 사용함
- 트랜잭션의 응답 시간 분포를 시간 경과에 따른 그래프로 제시해주기 때문에 성능 테스트 중 응답 시간 분포를 보다 직관적으로 파악하기 용이함
- 각 트랜잭션 처리 과정에서 각 메서드 단위의 실행 시간을 직관적으로 보여주기 때문에 병목 지점을 파악하기 용이함

관련 개발 문서: 모니터링 환경 구성, 로깅 적용 (CloudWatch), 모니터링 서버에 Pinpoint 설치 & 실행, 모니터링 서버 실시간 메모리 사용량 모니터링 (CloudWatch)

주요 작업 | 5. 성능 테스트 환경 구축_요약



** 프론트엔드 배포 관련 서비스나 모니터링 서버는 이미지에서 제외함*

성능 테스트 환경 (스테이징 환경) 부가 설명

- 개발한 API의 성능을 측정하기 위해 별도 환경 구성
- 기본적으로 다음 조건 고려하여 테스트 환성 구성
  1. DB에 대량의 데이터 쉽게 적재
  2. 대규모 부하 발생
  3. 어플리케이션 및 시스템 실시간 모니터링
  4. 테스트 비용 발생 최소화 (동일 AZ 위치)
- DB에 성능 테스트를 위한 mock data 편리하게 입력하기 위해 파이썬 스크립트 작성하여 사용함 (작업 문서)
- 성능 테스트 진행 영상

성능 테스트 결과 예시 (게시글 조회 테스트)		
테스트 설정	테스트 엔드포인트	https://staging.api.bootme.co.kr/posts?topic=&sort=likes&page=1&size=25
	동시 접속자(Vuser)	50 (Ramp-up 적용)
	인프라	WAS 2대 (t4g.small) DB 1대 (t4g.small) ElastiCache 샤드1 노드2 (cache.t2.micro)
	테이블 row	100만 (post 테이블)
테스트 결과	TPS	389
	MTT (ms)	67.24ms

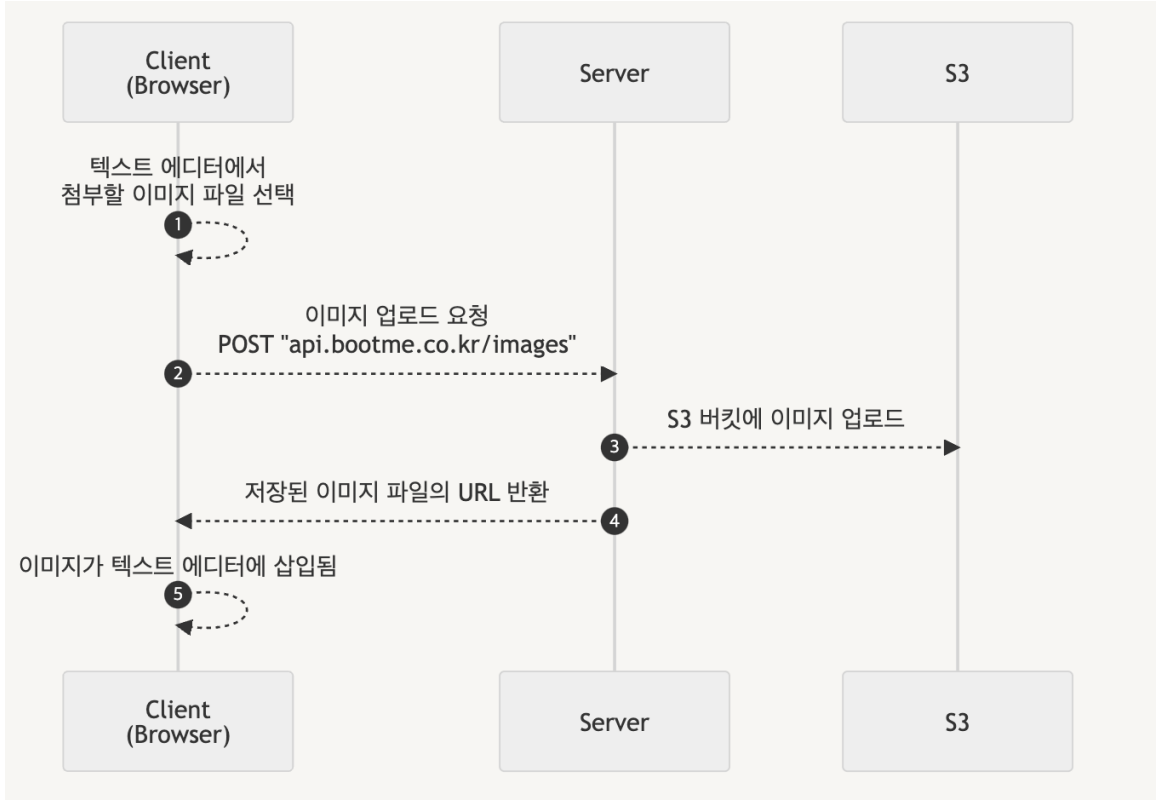
관련 개발 문서: API 성능 테스트 계획, 스테이징 환경 구성, 성능 테스트 결과, 스테이징 DB에 Mock Data 입력

주요 작업 | 6. 이미지 첨부 구현_처리 과정

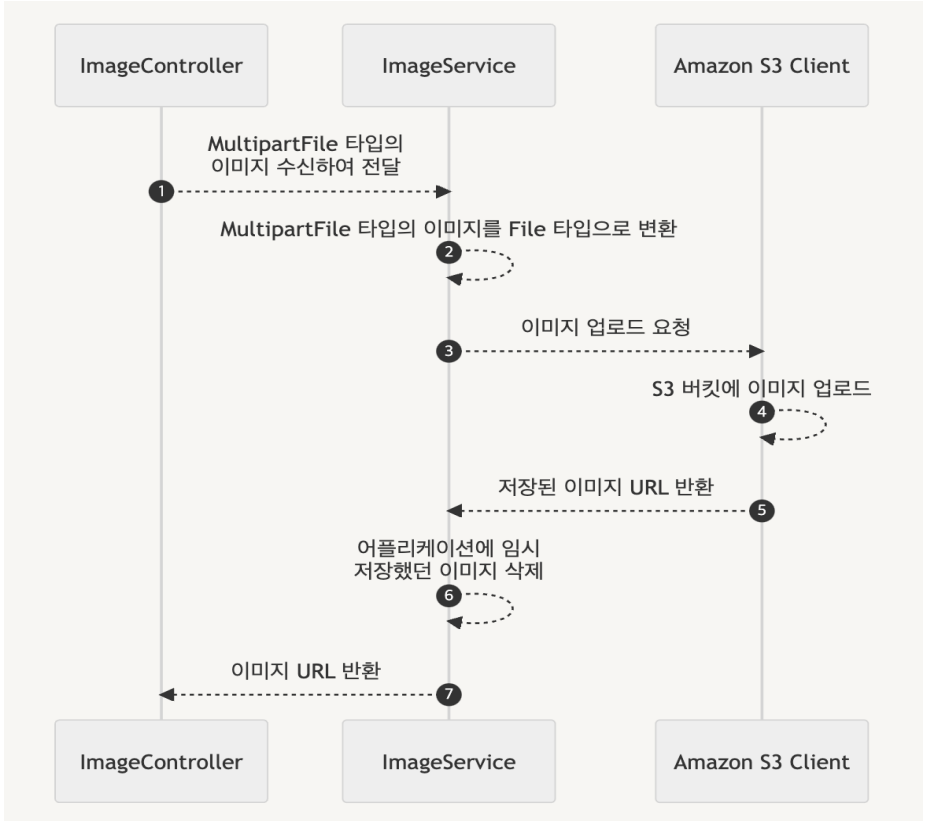
텍스트 에디터에 이미지를 첨부 할 수 있도록 구현함

에러 처리의 일관성과 보안 문제를 고려하여, 클라이언트에서 직접 S3에 업로드하지 않고 서버를 통해 S3에 업로드하는 방식으로 구현함

텍스트 에디터에서 이미지 첨부시 처리 과정



서버 內 이미지 업로드 처리 과정



관련 개발 문서: 코스 등록 페이지 구현

관련 코드: ImageController, ImageService

APPENDIX | 문서화

체계적인 문서화를 통해 분절적인 작업의 한계를 보완하고자 함

전체 개발 문서 (링크)

문서	설명	문서 종류
Sprints	기간별 테스크 관리	DB
Tasks	구체적으로 정리된 테스크 ★	DB
Daily Notes	일간 테스크 관리	DB
성능 테스트 결과	API 성능 테스트	DB
To do	간단하게 정리된 테스크	Sticky Note
개발 원칙	반복적인 결정 사항을 원칙으로 설정	DB
Decisions	작은 단위 의사결정	DB
Diagram	다이어그램 재사용 위해 관리	DB
Troubleshooting	개발 중 에러 해결	DB
환경변수, SQL	반복 사용하는 SQL 쿼리, 환경변수 정리	Single Page
참고 API	타 서비스 API 구조 참고	Single Page
참고 프로젝트	타 프로젝트 소스코드 참고	DB
참고 서비스	타 서비스 UI 참고	DB
부트캠프 리스트	부트캠프 주관사 정리	Single Page
To study	학습 주제 간단하게 정리	Sticky Note
Concepts	이론 & 개념 정리	DB

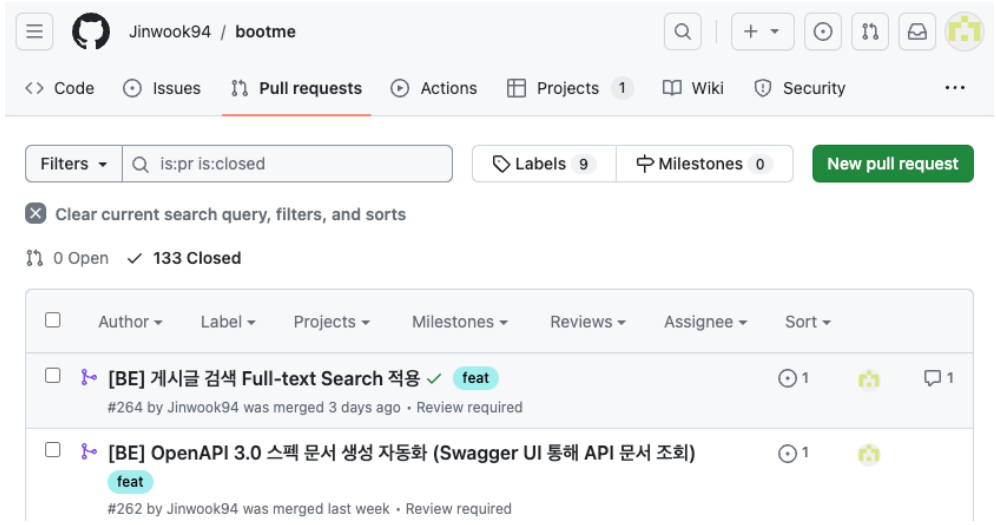
주요 문서 설명

문서 이름	설명	예시
Sprints	• 기간 별 작업 목표를 구체화 하여 계획적인 프로젝트 진행	• Sprint 5 • Sprint 6
- Tasks	• To do 문서에 기록한 간단한 형태의 작업 내용을 구체화	• 게시글 조회 성능 최적화 • JVM 힙 메모리 설정 • 1.0 배포 • 커뮤니티 페이지 구현
- Daily Notes	• 일간 테스크 관리	
To do	• 간단한 형태로 필요 작업을 기록	• 회원정보 저장시 암호화
개발 원칙	• 큰 틀에서 원칙을 정하여 비슷한 유형의 의사결정이 반복될 때 빠르게 의사 결정을 할 수 있도록 함	• API 설계 원칙 • REST 원칙 준수 범위 • JSON 설계 원칙
Decisions	• 작은 단위 결정 사항의 결정 과정을 간단히 기록	• EC2 Instance (WAS) • Map vs. DTO • 비즈니스 로직 위치
Troubleshooting	• 개발 중 에러 해결 과정 정리	• AfterInstall Hook 실행안됨 • API 응답 값 일부 캐싱
To study	• 개념적인 학습이 필요한 주제를 간단하게 정리	

APPENDIX | Git, GitHub 활용

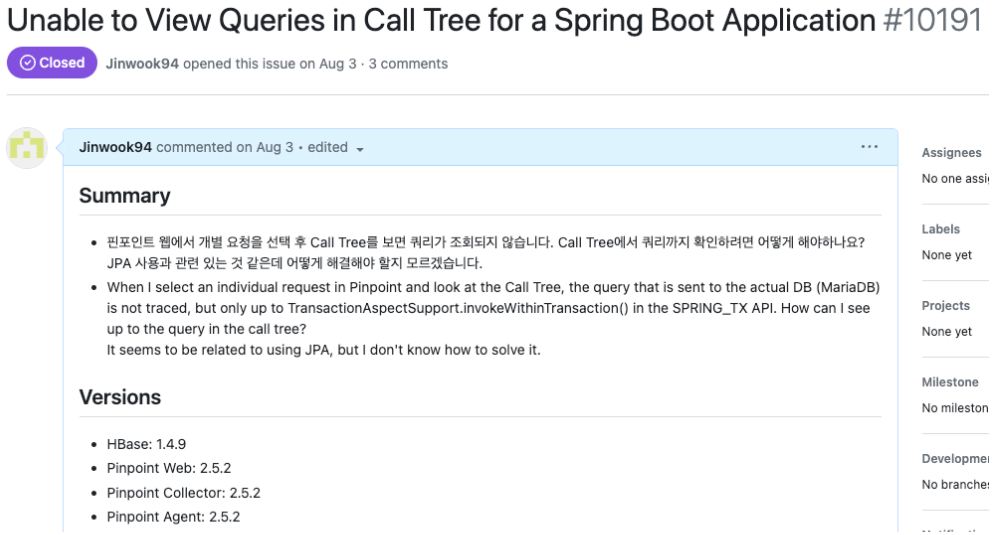
협업 경험이 부족할 수 있는 개인 프로젝트의 한계를 보완하기 위해 Git, GitHub 등 협업 상황에서 필요한 도구를 최대한 활용함

Issue, PR 작성



- 주요 기능 개발 시 Issue 를 생성하고 해당 Issue와 연결된 브랜치에서 작업 진행
- 작업 완료 후 코드를 원격 저장소에 푸쉬하고, PR 생성 후 리뷰를 한 뒤 병합
- 260개 이상의 Issue, PR 작성

오픈소스 프로젝트 Issue 작성

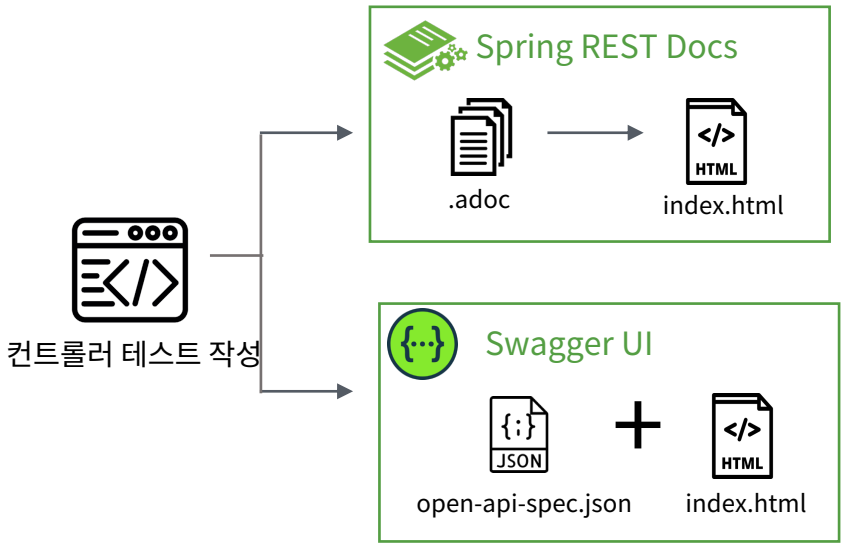


- Pinpoint APM, GitHub Actions, restdocs-api-spec 등의 도구를 사용하며 발생한 에러를 해결하기 위해 해당 프로젝트 리포지토리에 이슈를 작성하고 도움을 받아 해결

APPENDIX | API 문서

테스트 코드를 통해 자동으로 OpenAPI Spec 문서를 생성하며, 해당 문서는 Swagger UI에서 조회 가능하도록 구현함

작성된 테스트로부터 API 문서 자동 생성

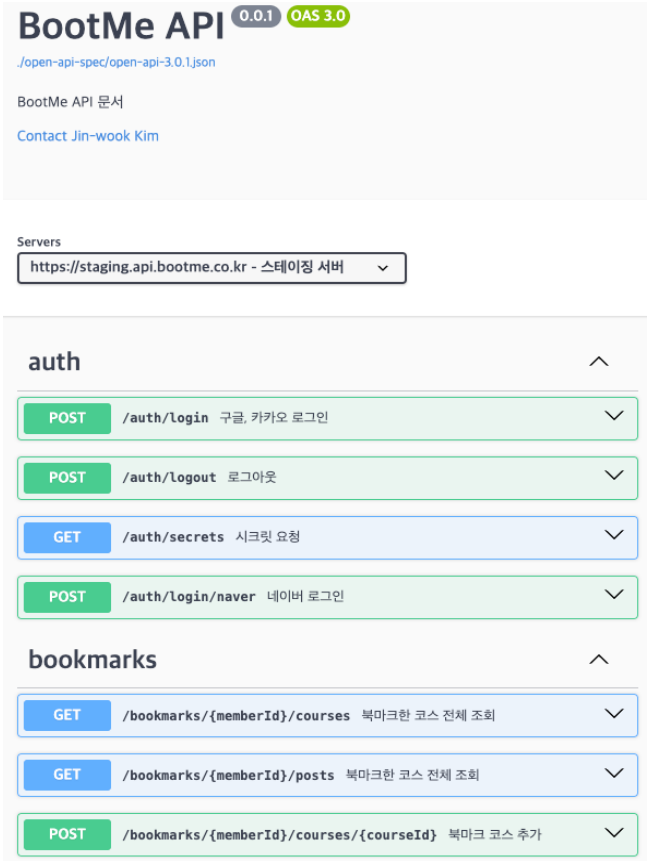


- 프로덕션 코드에 API 문서 관련 코드를 작성하지 않아도 되고, 테스트 작성이 강제된다는 장점 때문에 초기에는 Spring REST Docs 선택
- 추후 Swagger UI의 장점 고려해 restdocs-api-spec 라이브러리 사용한 OpenAPI Spec 문서 자동 생성 추가함

** Swagger UI 장점: 직관적인 인터페이스, API 테스트 편리하게 진행*

Swagger API 문서

https://api.bootme.co.kr/docs/swagger/index.html



Spring REST Docs API 문서

https://api.bootme.co.kr/docs/rest/index.html

관련 개발 문서: Spring Rest Docs 적용, OpenAPI 3.0 스펙 문서 생성 자동화, 컨트롤러 테스트 작성,

APPENDIX | 반복 작업 자동화

개발 학습 중이나 프로젝트 진행 중에 반복적인 작업을 자동화 한 경험이 있습니다.

Java API 사용 빈도 추출 스크립트 작성

- Java를 처음 학습하며 새로운 Java API를 접할 때마다 해당 API가 실무에서 실제로 많이 사용되는지를 찾아보며 중요도를 파악했습니다.
- 선별한 깃허브 리포지토리에 해당 API(클래스명)를 검색하여 사용 빈도와 사용 형태를 파악하는 방식이었습니다.
- 이 과정이 반복적이어서 다음과 같은 인풋과 아웃풋을 가지는 간단한 스크립트를 작성하여 사용 했습니다.

Input	1. N개의 Java API 클래스명
	2. N개의 깃허브 리포지토리명
Output	입력된 Java API들이 입력된 깃허브 리포지토리들에서 사용된 횟수가 정리된 엑셀 파일

서버 시작시 프로세스 실행 자동화

- 스테이징 환경에서 EC2 인스턴스들은 자주 중지되었다가 필요에 따라 재시작 되는 경우가 많았습니다.
- 인스턴스를 재시작 할 때마다 수동으로 접속하여 프로세스를 시작 해야 하는 번거로움이 있었습니다.
- 예를 들어 Pinpoint 모니터링 서버 재시작시 해당 서버에 AWS 콘솔이나 ssh로 접속하여 Java 프로세스 실행 커맨드를 직접 입력해야 했습니다.
- 이러한 문제를 해결하기 위해 **systemd 서비스 파일을 활용하여 프로세스의 자동 실행을 구현**했습니다.
- 인스턴스 시작만 하면 자동 실행 설정한 프로세스가 실행되도록 하여 단순 반복적인 작업을 줄일 수 있었습니다.

작업 문서: EC2 시작시 프로그램 시작 자동화

EC2 인스턴스 이름 자동 변경

- 인스턴스를 구분하기 위해 인스턴스의 이름을 해당 인스턴스가 시작된 시간으로 변경하는 작업이 필요했습니다.
- 새로운 배포 시마다 기존 WAS 인스턴스는 종료되고 새 WAS 인스턴스가 시작되므로 이름 변경 작업이 반복적으로 진행 되었습니다.
- EC2 시작 템플릿의 **사용자 데이터에 인스턴스 이름을 현재 시각으로 변경하는 커맨드를 추가**하여 위 작업을 자동화 했습니다.
- IMDSv2를 사용하여 인스턴스 ID를 가져온 뒤, aws-cli에서 해당 인스턴스 ID를 사용하여 인스턴스 이름을 현재 시각으로 변경하는 방식이었습니다.

Name	인스턴스 ID
STG_WAS_AS_231022_233143	i-00ef7aae700ff4811
PROD_WAS_AS_231027_004009	i-0769ea841c8f7f2e9