3-2. 백엔드 시작 - Part 2: 개발하면서 정해도 되는 것들

들어가며

Part 1에서는 개발 시작 전에 반드시 정해야 할 것들을 다뤘다. 기술 스택, API 설계 원칙, 인증 방식 같은 것들이다. 이것들을 정하지 않고 개발을 시작하면 나중에 재앙이 찾아온다.

하지만 모든 것을 처음부터 완벽하게 정할 필요는 없다.

"N+1 쿼리는 어떻게 해결하지?"
"테스트 커버리지는 몇 퍼센트를 목표로 하지?"
"로그는 얼마나 상세하게 남기지?"

이런 것들은 개발하면서 정해도 된다. 아니, 오히려 **개발하면서 정해야** 한다. 실제로 문제가 보이기 전에는 무엇이 문제인지 알 수 없기 때문이다.

Part 2에서는 개발하면서 점진적으로 개선해도 되는 것들을 다룬다. 이것들을 처음부터 완벽하게 하려고 하면, 개발 속도만 느려지고 출시는 늦어진다.

---

1. 데이터베이스 인덱스 설계

1.1 왜 나중에 해도 되나?

인덱스는 "느린 쿼리"가 보일 때 추가하는 것이 효율적이다.

처음부터 모든 컬럼에 인덱스를 만들면? INSERT와 UPDATE가 느려진다. 인덱스도 저장 공간을 차지한다. 그런데 그 인덱스가 실제로 사용되는지는 모른다.

실제 사례

어떤 팀에서는 "혹시 나중에 필요할까봐" 모든 컬럼에 인덱스를 걸었다. 데이터가 10만 건을 넘어가자 INSERT가 점점 느려졌다. 알고 보니 사용하지 않는 인덱스가 10개나 있었다. 인덱스를 제거하자 INSERT 속도가 2배 빨라졌다.

1.2 언제 추가하나?

1) Slow Query Log를 확인했을 때

실제로 운영하다 보면 "이 쿼리가 3초나 걸리네?"라는 순간이 온다. 그때 EXPLAIN으로 분석하고, 필요한 컬럼에 인덱스를 추가한다.

2) 사용자가 느리다고 느낄 때

사용자가 "검색이 너무 느려요"라고 불평하면, 그때 검색 쿼리를 분석하고 인덱스를 추가한다.

1.3 처음부터 만들어도 되는 인덱스

완전히 안 만들면 안 된다. 최소한의 인덱스는 처음부터 만든다.

Primary Key와 Foreign Key는 자동 생성

PostgreSQL은 PK에 자동으로 인덱스를 만든다. FK는 명시적으로 만들어야 하지만, JPA를 쓴다면 자동으로 만들어준다.

자주 조회되는 것이 명확한 컬럼

"사용자를 이메일로 조회"는 당연히 필요하다. 이런 건 처음부터 만든다.

```
-- 처음부터 만들어도 되는 인덱스
CREATE INDEX idx_user_email ON users(email);
```

```
CREATE INDEX idx_match_status ON matches(status);
```

하지만 "사용자를 닉네임 + 생성일 + 활성 상태로 동시 조회"는 실제로 그런 쿼리가 있는지 확인한 후에 만든다.

1.4 왜 미리 최적화는 악인가?

"Premature optimization is the root of all evil" - Donald Knuth

처음부터 완벽하게 최적화하려고 하면 개발 속도가 느려진다. 일단 작동하게 만들고, 병목이 보이면 그때 최적화한다.

---

2. 성능 최적화 전략

2.1 왜 나중에 해도 되나?

처음부터 "초당 10만 요청을 처리해야 해!"라고 최적화하면? MVP는 영원히 출시되지 않는다.

사용자가 10명일 때와 10만 명일 때 필요한 최적화는 다르다. 사용자가 10명일 때는 단순한 코드가 최선이다.

2.2 언제 최적화하나?

1) 성능 문제가 실제로 발생했을 때

"API 응답이 2초나 걸려요"
"매칭 대기 시간이 너무 길어요"

이런 불평이 나올 때 최적화한다.

2) 모니터링에서 병목이 보일 때

Grafana를 보니 특정 API의 P95 응답 시간이 1초를 넘는다. 그때 해당 API를 최적화한다.

2.3 최적화 우선순위

모든 걸 최적화할 수는 없다. 우선순위를 정해야 한다.

1순위: 사용자가 자주 사용하는 기능

로그인, 매칭, 게임 시작. 이런 핵심 기능이 느리면 사용자가 떠난다.

2순위: 서버 부하가 높은 기능

한 번 호출에 DB 쿼리가 10개씩 나가는 API가 있다면, 이걸 최적화하면 서버 비용을 줄일 수 있다.

3순위: 가끔 사용하는 관리 기능

어드민 페이지에서 전체 사용자 통계를 보는 기능. 하루에 10번 사용된다면 굳이 최적화할 필요 없다.

2.4 흔한 최적화 포인트

N+1 쿼리 해결

처음에는 신경 안 써도 된다. 나중에 Slow Query Log에서 발견되면 그때 `@EntityGraph`나 `fetch join`으로 해결한다.

캐싱 추가

자주 조회되지만 잘 변하지 않는 데이터. 예를 들어 사용자 프로필. Redis에 캐싱하면 DB 부하를 줄일 수 있다. 하지만 처음부터 할 필요는 없다.

비동기 처리

이메일 발송, 이미지 리사이징 같은 오래 걸리는 작업. 처음에는 동기로 처리하다가, 사용자가 "너무 느려요"라고 하면 그때 비동기로 바꾼다.

---

3. 테스트 전략과 커버리지

3.1 왜 나중에 정해도 되나?

"테스트 커버리지 100%"를 목표로 하면? MVP는 영원히 출시되지 않는다.

테스트도 코드다. 테스트 코드를 작성하는 데도 시간이 든다. 처음부터 모든 코드에 테스트를 작성하면 개발 속도가 절반으로 느려진다.

3.2 MVP 단계의 테스트 전략

핵심 비즈니스 로직만 테스트

매칭 알고리즘, 웃음 게이지 계산, 승패 판정. 이런 핵심 로직에 버그가 있으면 서비스 자체가 무너진다. 이것들은 단위 테스트를 작성한다.

CRUD는 나중에

단순 CRUD는 테스트를 나중에 작성해도 된다. "사용자 조회"에 버그가 있으면? 금방 발견되고 금방 고칠 수 있다.

통합 테스트는 주요 시나리오만

"회원가입 → 로그인 → 매칭 → 게임" 같은 주요 흐름만 통합 테스트로 작성한다. 모든 엣지 케이스를 테스트하려고 하지 않는다.

3.3 테스트 커버리지 목표

처음부터 80%를 목표로 하지 말자

MVP 단계에서는 40-50%면 충분하다. 핵심 로직 위주로 테스트를 작성한다.

점진적으로 올린다

서비스가 안정화되면 커버리지를 60%, 70%로 올린다. 하지만 100%는 목표로 하지 않는다.

왜 100%가 아닌가?

Getter/Setter, 간단한 DTO, Configuration 클래스. 이런 것들을 테스트하는 건 시간 낭비다. 의미 있는 로직만 테스트한다.

3.4 언제 테스트를 추가하나?

버그가 발견됐을 때

운영 중에 버그가 발견되면, 먼저 그 버그를 재현하는 테스트를 작성한다. 그다음 버그를 고친다. 이렇게 하면 같은 버그가 다시 발생하지 않는다.

리팩토링하기 전에

코드를 대대적으로 수정하기 전에, 기존 동작을 보장하는 테스트를 먼저 작성한다. 그래야 리팩토링 후에도 "기능이 정상 작동하는가?"를 확인할 수 있다.

---

4. 로깅 전략

4.1 왜 나중에 정해도 되나?

처음에는 "일단 많이 남기자"로 시작해도 된다.

개발 초기에는 DEBUG 레벨로 모든 걸 로깅한다. 어떤 정보가 필요한지 아직 모르기 때문이다. 실제로 운영하다 보면 "아, 이 로그는 필요 없구나" "이 로그는 더 상세해야 하는구나"를 알게 된다.

4.2 개발 단계별 로깅 레벨

개발 환경 (Local, Dev)

```
root: DEBUG
com.example.gagbattle: DEBUG
org.springframework: INFO
```

모든 것을 자세히 본다. SQL 쿼리도 보고, HTTP 요청/응답도 본다.

운영 환경 (Production)

```
root: WARN
com.example.gagbattle: INFO
org.springframework: WARN
```

필요한 것만 본다. 모든 DEBUG 로그를 남기면 디스크가 금방 찬다.

4.3 무엇을 로깅하나?

반드시 로깅해야 하는 것

- 에러: 모든 Exception은 스택 트레이스와 함께
- 중요한 비즈니스 이벤트: 회원가입, 로그인, 매칭 성공, 게임 종료
- 외부 API 호출: 카카오 로그인, S3 업로드

로깅하면 안 되는 것

- 비밀번호: 절대 로그에 남기면 안 됨
- JWT 토큰: 탈취 위험
- 개인정보: 이메일, 전화번호 (필요하면 마스킹)

4.4 로그 구조화

나중에는 JSON 형태로 구조화된 로그를 남기는 게 좋다. 하지만 처음에는 단순한 텍스트 로그로 시작해도 된다.

왜 구조화가 필요한가?

나중에 "어제 몇 시에 매칭 타임아웃이 발생했지?"를 찾고 싶을 때, 구조화된 로그는 검색이 쉽다. 하지만 MVP 단계에서는 과도하다.

---

5. 모니터링 시스템

5.1 왜 나중에 해도 되나?

사용자가 10명일 때는 모니터링이 필요 없다. 문제가 생기면 사용자가 직접 알려준다.

하지만 사용자가 1,000명이 되면? 모든 사용자의 불평을 직접 듣기 어렵다. 그때 모니터링이 필요하다.

5.2 단계별 모니터링

Phase 1 (MVP): 최소한의 모니터링

- Health Check 엔드포인트: 서버가 살아있는가?
- 에러 로그: 에러가 발생하면 Slack 알림

이 정도면 충분하다.

Phase 2 (성장기): 기본 메트릭

- CPU, 메모리, 디스크 사용률
- API 응답 시간 (평균, P95, P99)
- 에러율

Prometheus + Grafana로 수집하고 시각화한다.

Phase 3 (확장기): 상세 모니터링

- 비즈니스 메트릭: DAU, 매칭 성공률, 게임 완료율
- 분산 추적: 한 요청이 여러 서비스를 거칠 때 추적
- APM: 코드 레벨에서 병목 지점 찾기

5.3 언제 추가하나?

사용자 불평이 패턴으로 보일 때

"요즘 서버가 자주 느려지는 것 같아요"라는 불평이 3번 이상 들리면, 모니터링을 추가한다.

장애가 발생했을 때

장애가 발생했는데 "무엇이 문제였는지" 알 수 없다면, 모니터링이 부족한 것이다. 장애 복구 후에 관련 메트릭을 추가한다.

---

6. 에러 처리 전략

6.1 왜 나중에 개선해도 되나?

처음에는 단순하게 시작한다.

```
try {
    // 비즈니스 로직
} catch (Exception e) {
    log.error("Error occurred", e);
    return error("서버 오류가 발생했습니다");
}
```

이 정도면 충분하다. 나중에 "이 에러는 어떻게 처리해야 하지?"가 명확해지면 개선한다.

6.2 점진적으로 개선하기

처음: 모든 에러를 같게 처리

모든 에러에 500을 반환한다. 사용자는 "서버 오류"라고만 본다.

개선 1: 에러 종류 구분

4xx는 클라이언트 잘못, 5xx는 서버 잘못으로 구분한다.

개선 2: 커스텀 Exception

`UserNotFoundException`, `MatchNotFoundException` 같은 의미 있는 예외를 만든다.

개선 3: 에러 코드 체계

Part 1에서 정한 에러 코드 체계를 실제로 적용한다.

6.3 언제 개선하나?

프론트엔드 개발자가 불편해할 때

"이 에러가 무슨 에러인지 모르겠어요"
"404와 400을 구분할 수 없어요"

이런 불평이 나오면 개선한다.

운영 중에 디버깅이 어려울 때

"사용자가 에러가 났다고 하는데, 무슨 에러인지 모르겠어요"

이럴 때 더 상세한 에러 정보를 추가한다.

---

7. 보안 강화

7.1 왜 점진적으로 강화해도 되나?

보안은 중요하다. 하지만 처음부터 완벽한 보안을 구현하면 출시가 늦어진다.

최소한의 보안만 처음부터

- HTTPS (Let's Encrypt 무료)
- 비밀번호 해싱 (bcrypt)
- SQL Injection 방지 (Prepared Statement)

이 정도는 처음부터 해야 한다. 하지만 "2단계 인증", "IP 화이트리스트", "보안 감사 로그" 같은 건 나중에 추가해도 된다.

7.2 언제 강화하나?

사용자가 증가할 때

사용자가 1,000명을 넘으면 악의적인 공격자가 나타날 가능성이 높아진다. 그때 Rate Limiting, CAPTCHA 같은 걸 추가한다.

보안 사고가 발생했을 때

누군가 무차별 대입 공격(Brute Force)으로 계정을 탈취했다면, 로그인 실패 횟수 제한을 추가한다.

규정 준수가 필요할 때

개인정보보호법, GDPR 같은 규정을 준수해야 한다면, 관련 보안 조치를 추가한다.

---

8. 트랜잭션 세분화

8.1 왜 나중에 해도 되나?

처음에는 Service 메서드 전체에 `@Transactional`을 붙인다.

```
@Transactional
public Match createMatch(MatchRequest request) {
    // 모든 로직
}
```

나중에 성능 문제가 보이면, 트랜잭션 범위를 줄인다.

8.2 언제 세분화하나?

트랜잭션이 너무 길 때

한 트랜잭션이 3초 이상 걸리면 DB 커넥션을 오래 점유한다. 이럴 때 꼭 필요한 부분만 트랜잭션으로 묶는다.

Deadlock이 발생할 때

트랜잭션이 길면 Deadlock 가능성이 높아진다. 트랜잭션 범위를 줄이면 해결되는 경우가 많다.

---

9. 캐싱 전략

9.1 왜 나중에 추가해도 되나?

캐싱은 복잡도를 높인다.

"캐시 무효화는 언제 하지?"
"캐시와 DB가 불일치하면 어떡하지?"

이런 문제를 처음부터 고민하면 개발 속도가 느려진다. 일단 캐시 없이 만들고, 성능 문제가 보이면 그때 추가한다.

9.2 언제 추가하나?

같은 데이터를 반복 조회할 때

사용자 프로필을 한 요청에서 10번 조회한다면, 캐싱을 고려한다.

DB 부하가 높을 때

CloudWatch를 보니 RDS CPU가 80%를 넘는다. 조회 쿼리가 많다면 Redis 캐싱으로 부하를 줄일 수 있다.

9.3 무엇을 캐싱하나?

자주 읽히지만 잘 변하지 않는 데이터

- 사용자 프로필
- 설정 값
- 정적 컨텐츠

캐싱하면 안 되는 데이터

- 실시간성이 중요한 데이터 (매칭 상태, 게임 진행)
- 자주 변경되는 데이터 (웃음 게이지)

---

10. 배포 전략 고도화

10.1 처음에는 단순하게

MVP 단계에서는 단순한 배포로 충분하다.

1. 서버 SSH 접속
2. `git pull`
3. `./gradlew build`
4. `java -jar app.jar`

이 정도면 된다. 2분 정도 서비스가 멈추지만, 사용자가 10명이라면 큰 문제가 아니다.

10.2 점진적으로 개선

사용자 100명: GitHub Actions 자동 배포

SSH로 접속해서 수동 배포하는 건 번거롭다. GitHub Actions로 자동화한다.

사용자 1,000명: Blue-Green 배포

2분 다운타임이 문제가 된다. Blue-Green 배포로 무중단 배포를 구현한다.

사용자 10,000명: Canary 배포

새 버전에 버그가 있으면 전체 사용자에게 영향을 준다. Canary 배포로 일부 사용자에게만 먼저 배포한다.

Part 2 체크리스트

개발하면서 점진적으로 개선해야 할 것들:

당장은 아니지만 곧 필요한 것들

- [ ] 주요 쿼리에 인덱스 추가
- [ ] 핵심 로직 단위 테스트 작성
- [ ] 에러 로그 Slack 알림 설정
- [ ] Health Check 엔드포인트 추가

사용자가 늘어나면 필요한 것들

- [ ] Redis 캐싱 추가
- [ ] N+1 쿼리 해결
- [ ] Rate Limiting 추가
- [ ] 모니터링 시스템 (Prometheus + Grafana)

성능 문제가 보이면 필요한 것들

- [ ] Slow Query 분석 및 최적화
- [ ] 비동기 처리 추가
- [ ] DB Read Replica 추가
- [ ] CDN 추가

마치며

> "완벽함은 적의 적이다" - Voltaire

처음부터 완벽하게 만들려고 하지 말자.

일단 작동하게 만든다. 그다음 사용자 피드백을 받는다. 문제가 보이면 그때 개선한다.

"나중에 성능 문제가 생기면 어떡하지?"라고 걱정하지 말자. 성능 문제는 **생겼을 때** 해결하면 된다. 지금은 **출시**가 목표다.

MVP를 출시하지 못하면, 최적화는 의미가 없다. 아무도 사용하지 않는 완벽한 시스템보다, 많은 사람이 사용하는 개선 여지가 있는 시스템이 낫다.

Part 3에서는 실제로 개발을 시작하는 방법을 다룬다.