2. 기획 정리 그리고 개발 들어가기 앞서

마치며

들어가며

기획이 완료되었다. 이제 바로 코드를 작성하고 싶은 마음이 굴뚝같을 것이다. 하지만 잠깐, 서두르지 말자.

개발을 시작하기 전에 제대로 준비하지 않으면, 프로젝트 중반에 엄청난 혼란을 겪게 된다. "이 API는 어떤 스펙이었지?", "ERD는 어디에 있지?", "배포는 어떻게 하기로 했더라?" 같은 질문들이 쏟아지면서 개발 속도가 급격히 느려진다.

이 문서는 개발을 시작하기 전에 반드시 해야 할 일들을 정리한 것이다. 지루하게 느껴질 수 있지만, 이 과정을 제대로 거치면 개발 단계에서 훨씬 빠르고 효율적으로 움직일 수 있다.

---

0. 개발 이전에

브레인스토밍이 끝나고, 시장조사와 어느 정도의 기획, 최소 MVP 선정, 와이어프레임 및 어떤 기술들을 사용할지 대략적으로 윤곽이 나왔으면 한 페이지로 기획에 대해 설명하고 정리해보는 것을 추천한다.

0.1 왜 한 페이지 기획서가 필요할까?

회의록은 있다. 시장조사 자료도 있다. 와이어프레임도 그렸다. 하지만 이 모든 자료들이 흩어져 있으면, 팀원들이 전체 그림을 보기 어렵다.

"우리가 지금 뭘 만들고 있지?" "왜 이 기능을 넣기로 했더라?" 같은 질문이 계속 나온다면, 그건 기획이 명확하게 정리되지 않았다는 신호일 수도 있기에 한 페이지로 작성을 하면서 팀원 다같이 생각이 다져보는 시간을 가지는 편이 이때까지 프로젝트 하면서 도움이 됐던거같다.

~~사실 제가 잘 까먹습니다 허허~~

개발 중에 방향을 잃었을 때, 여기로 돌아와서 "우리가 왜 이걸 만들기로 했지?"를 확인할 수 있다.

0.2 한 페이지에 무엇을 담아야 할까?

서비스 핵심 가치 (Value Proposition)

한 문장으로 표현해보자.
"우리 서비스는 [누구]에게 [무엇]을 제공한다."

이 한 문장이 명확하지 않으면, 나중에 기능을 추가할 때마다 "이게 정말 우리 서비스에 필요한가?"를 판단하기 어려워진다.

타겟 유저

구체적일수록 좋다. "모든 사람"이 아니라 "10대 후반~20대 초반, 유튜브에서 웃음 참기 콘텐츠를 즐겨보는, 사람들"처럼.

타겟이 명확하면 디자인부터 기능까지 모든 결정이 쉬워진다. "이 기능은 우리 타겟 유저가 원할까?"라는 질문만 하면 된다. 근데 이게 쉽지 않긴 하다..

핵심 기능 3가지

MVP 단계에서 반드시 구현해야 할 기능 3가지만 적는다. 3가지보다 많으면 우선순위가 흐려진다.

딱 뽑아야 하는 3가지...!

제외하기로 한 것들

오히려 이게 더 중요할 수 있다. 무엇을 안 할지 명확히 하는 것.

우리 서비스라면:

- MVP에서는 랭킹 시스템 없음 (추후 추가)
- MVP에서는 다대다 배틀 없음 (1대1만)
- MVP에서는 도네이션 기능 없음

이렇게 적어두면, 나중에 까먹지도 않고 뭔가 갑자기 맞다 이거 해야지 하면서 MVP 만들다가 도중에 추가하는 불상사도 막을 수 있다. 타 프로젝트 진행했었을때 중간중간 기획이 바뀌게 되면서 디자인과 개발을 갈아엎은적도 꽤 있었기에 초기에 미연에 방지하고 열려있는 설계를 하면 오히려 확장성에도 좋다

성공 지표

어떤 숫자를 보고 "성공했다"고 판단할 것인가?

MVP 단계라면:

- 첫 달 가입자 1,000명
- DAU 100명

⇒ 근데 이건 SSAFY에서 진행하기에 굳이 싶긴 하지만 그래도 기획 설명이니..

기술 스택 개요

상세한 건 나중에 정리하겠지만, 큰 그림만 적어둔다

- 프론트: React + TypeScript
- 백엔드: Spring Boot (Java)
- 실시간 통신: WebRTC + Socket.io
- AI: **모릅니다.**
- DB: MySQL+ Redis
- 인프라: AWS + Docker

왜 이 기술들을 선택했는지 한 줄씩 이유를 적으면 더 좋다.

1. 기술 스택 선정

1.1 왜 기술 스택을 먼저 정해야 할까?

기술 스택은 집을 지을 때 사용할 재료를 고르는 것과 같다. 약간 뼈대 고르기..?

나무로 지을지, 벽돌로 지을지, 철골로 지을지 결정하는 단계다. 일단 짓기 시작하면 중간에 바꾸기가 매우 어렵다.

DB는 그래도 마이그레이션하면 되지만, 언어의 변경은 사실상 처음부터 재설계라 봐야할 정도의 대공사..

1.2 무엇을 고려해야 할까?

팀의 역량

가장 중요한 건 팀원들이 무엇을 할 수 있느냐다. 아무리 최신 기술이 좋다고 해도, 팀원 모두가 처음 접하는 기술이라면 학습 곡선 때문에 개발 속도가 크게 느려진다.

반대로 팀원 대부분이 이미 익숙한 기술이 있다면, 그걸 활용하는 게 현명하다. MVP 단계에서는 빠른 출시가 중요하기 때문이다.

프로젝트 특성

우리가 만들려는 서비스는 실시간 화상 통신과 AI 기반 웃음 감지가 핵심이다. 이런 특성을 고려해야 한다.

실시간 통신이 많다면 비동기 처리에 강한 언어나 프레임워크가 유리하다. AI 모델을 사용한다면 Python 생태계를 활용할 수 있는지도 중요하다. 웹소켓 연결이 많을 것이므로 이를 잘 지원하는 기술을 선택해야 한다.

예를 들어 파이썬을 이용한 AI 모델이 들어온다면? 모델 서버도 있어야 하고 모델을 컨테이너로 굴리냐, 별도의 인스턴스로 굴리냐, was 서버와의 통신은?, 대규모 트래픽 처리는? 등등

확장성과 유지보수

지금 당장은 작은 MVP지만, 나중에 사용자가 폭발적으로 늘어날 수 있다. 그때를 대비해 수평 확장(스케일 아웃)이 쉬운 아키텍처를 선택하는 게 좋다.

또한 3개월 후, 6개월 후에도 이 코드를 유지보수할 수 있어야 한다. 커뮤니티가 활발하고 문서가 잘 정리된 기술을 선택하면 나중에 문제가 생겼을 때 해결하기 쉽다.

1.3 기술 스택 문서화

기술 스택을 정했다면 반드시 문서로 남겨야 한다. 단순히 "Spring Boot를 쓴다"가 아니라, 왜 그 기술을 선택했는지 이유까지 적어두어야 한다.

기술 스택 문서 예시

```
# 기술 스택

## 백엔드
- 프레임워크: Spring Boot 3.2
- 언어: Java 21
- 선택 이유: 팀원 대부분이 Spring 경험이 있고,
  엔터프라이즈급 안정성과 풍부한 생태계 제공
```

```
## 프론트엔드
- 프레임워크: React 18
- 상태 관리: Zustand
- 선택 이유: 컴포넌트 기반 개발로 재사용성 높고,
  Zustand는 Redux보다 학습 곡선이 낮음

## 실시간 통신
- 라이브러리: WebRTC (simple-peer)
- 시그널링 서버: Socket.io
- 선택 이유: P2P 연결로 서버 부하 감소,
  Socket.io는 웹소켓 폴백 지원으로 안정성 높음

## AI/ML
- 프레임워크:
- 선택 이유:

## 데이터베이스
- 주 DB: MySQL
- 캐시: Redis 7

## 인프라
- 클라우드: AWS
- 컨테이너: Docker
- 오케스트레이션: Docker Compose (초기), 추후 ECS 고려
- 선택 이유: AWS는 SSAFY니까 꽁짜꽁짜 ㅎ
  Docker로 개발-운영 환경 일치
```

---

2. 기술 명세서 작성

2.1 기술 명세서가 필요한 이유

기술 명세서는 개발팀 전체가 같은 그림을 그리도록 만드는 청사진이다. 프론트엔드 개발자와 백엔드 개발자 들이 개발해야할 명시적인 방향성이라 꼭 있어야 한다. 버전부터 꼭..!!!

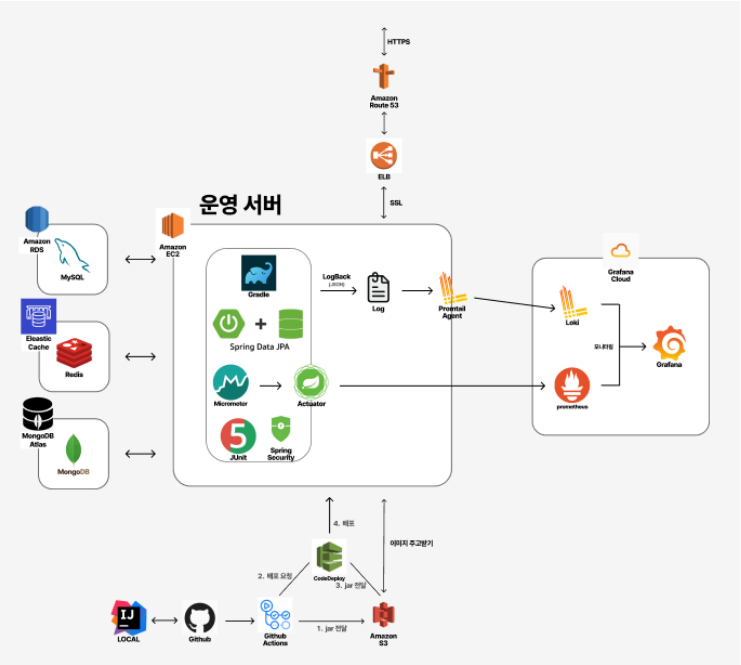
기술적인 부분은 개발자들끼리 논의를 거쳐 결정하면 좋고, 플로우의 경우는 PM이 작성하는게 전체적으로 이해가 잘 간다.

2.2 무엇을 포함해야 할까?

시스템 아키텍처 다이어그램

전체 시스템이 어떻게 구성되어 있는지 한눈에 보이는 그림이 필요하다. 사용자가 웹 브라우저에서 요청을 보내면, 그 요청이 어떤 경로를 거쳐 처리되는지 보여주어야 한다.

```
[사용자]
  ↓
[로드밸런서]
  ↓
[웹 서버 (Nginx)]
  ↓
[API 서버 (Spring Boot)] ←→ [Redis] ←→ [MySQL]
  ↓
[AI 서비스 (Python FastAPI)]
```



데이터 흐름

사용자가 서비스를 시작하면, 데이터가 어떻게 흘러가는지 설명한다.

1. 사용자가 "매칭 시작" 버튼 클릭
2. 프론트엔드에서 WebSocket 연결 요청
3. 백엔드에서 매칭 큐에 사용자 추가
4. 상대방 찾으면 양쪽에 WebRTC offer/answer 교환
5. P2P 연결 수립 후 게임 시작
6. AI가 실시간으로 표정 분석해서 웃음 게이지 업데이트

...

보안 고려사항

어떤 부분에서 보안에 신경 써야 하는지 미리 정리한다.

- JWT 토큰 만료 시간: 액세스 토큰 1시간, 리프레시 토큰 7일
- HTTPS 필수 적용
- 웹소켓 연결 시 토큰 검증
- CORS 설정: 특정 도메인만 허용

2.3 기술 명세서 템플릿 (feat. AI)

```
# 기술 명세서

## 1. 시스템 개요
- 서비스명: 개그 배틀
- 목적: 실시간 웃음 참기 대결 플랫폼
- 핵심 기술: WebRTC, AI 표정 인식, 실시간 매칭

## 2. 시스템 아키텍처
[다이어그램 첨부]

## 3. 기술 스택 상세
### 백엔드
- Spring Boot 3.2.0
- Java 21
- Spring Security + JWT
- Spring Data JPA

### 프론트엔드
- React 18.2.0
- TypeScript 5.0
- Vite (빌드 도구)
- Tailwind CSS

### 실시간 통신
- WebRTC (P2P)
- Socket.io 4.6 (시그널링)
- TURN 서버: Coturn (NAT 통과용)

## 4. 주요 플로우
### 매칭 플로우
[시퀀스 다이어그램]

### 게임 진행 플로우
[시퀀스 다이어그램]

## 5. 성능 목표
- 매칭 대기 시간: 10초 이내
- 웃음 감지 지연: 100ms 이내
- 동시 접속자: 10,000명 (Phase 1)
```

```
## 6. 보안 정책
- 인증: JWT 기반
- 통신: HTTPS, WSS 필수
- 개인정보: 암호화 저장 (AES-256)
```

---

3. API 명세서

3.1 API 명세서가 없으면 생기는 일

프론트엔드 개발자와 백엔드 개발자가 각자 상상하는 API 와 주고받는 정보 형태가 다를 수 있다.

프론트는 `userId`를 보냈는데 백엔드는 `user_id`를 기대한다거나, 프론트는 배열을 보냈는데 백엔드는 단일 객체를 기대한다거나. 이런 사소한 차이들이 쌓이면 통합 단계에서 엄청난 혼란이 생긴다.

글자 하나라도 잘못되면 파싱에 문제가 발생할 수도 있기에 그런 상황이 발생하지 않으려면

무엇보다 통일화 및 규격화가 잘 되어있어야 한다.

3.2 API 명세서에 포함되어야 할 것들

엔드포인트 정보

```
### POST /api/v1/auth/login
사용자 로그인

Request
Content-Type: application/json

{
  "email": "user@example.com",
  "password": "password123"
}

Response
Status: 200 OK

{
  "success": true,
  "data": {
    "accessToken": "eyJhbGciOiJIUzI1NiIs...",
    "refreshToken": "eyJhbGciOiJIUzI1NiIs...",
    "user": {
      "id": "uuid-1234",
```

```
    "email": "user@example.com",
    "nickname": "웃음참기왕"
   }
  }
}

Error Response
Status: 401 Unauthorized

{
  "success": false,
  "error": {
    "code": "INVALID_CREDENTIALS",
    "message": "이메일 또는 비밀번호가 올바르지 않습니다."
  }
}
```

상태 코드와 에러 코드

HTTP 상태 코드만으로는 부족하다. 더 구체적인 에러 코드가 필요하다.

```
## 에러 코드

| 코드 | 설명 | HTTP Status |
|------|------|-------------|
| INVALID_CREDENTIALS | 인증 정보 오류 | 401 |
| TOKEN_EXPIRED | 토큰 만료 | 401 |
| USER_NOT_FOUND | 사용자 없음 | 404 |
| ALREADY_IN_MATCH | 이미 매칭 중 | 409 |
| MATCH_FULL | 방이 가득참 | 409 |
| INTERNAL_ERROR | 서버 오류 | 500 |
```

3.3 API 문서화 도구 활용

Swagger/OpenAPI

Spring Boot라면 Springdoc-OpenAPI를 사용하면 코드에서 자동으로 API 문서를 생성할 수 있다. 개발자가 직접 문서를 업데이트하지 않아도 되고, 실제 코드와 문서가 항상 일치한다.

Postman Collection

팀원들이 실제로 API를 테스트해볼 수 있도록 Postman Collection을 만들어두면 좋다. 각 API의 예제 요청을 미리 만들어두면, 프론트엔드 개발자가 백엔드를 기다리지 않고 목(Mock) 데이터로 먼저 개발할 수 있다.

---

4. ERD (Entity Relationship Diagram)

4.1 ERD가 중요한 이유

데이터베이스 설계는 나중에 바꿀래야 바꿀순 있지만. 서비스가 운영 중일 때 테이블 구조를 변경하려면 데이터 마이그레이션이라는 고통스러운 작업을 거쳐야 한다.

처음부터 제대로 설계하는 게 훨씬 낫다.

4.2 ERD 설계 시 고려사항

정규화 vs 비정규화

이론적으로는 제3정규형까지 정규화하는 게 맞다. 하지만 실무에서는 성능을 위해 의도적으로 비정규화하는 경우도 있다.

예를 들어, 사용자의 총 승수를 매번 계산하지 않고 `users` 테이블에 `total_wins` 컬럼을 추가해서 저장할 수 있다. 이건 비정규화지만, 랭킹 조회 성능이 크게 개선된다.

인덱스 설계

자주 조회되는 컬럼에는 인덱스를 추가해야 한다. 하지만 인덱스가 많으면 쓰기 성능이 떨어진다. 균형을 잡아야 한다.

매칭 시스템에서 "현재 대기 중인 사용자 목록"을 자주 조회한다면, `status`와 `created_at` 컬럼에 복합 인덱스를 거는 게 좋다.

소프트 삭제 vs 하드 삭제

사용자가 계정을 삭제할 때, 실제로 데이터를 지울 것인가? 아니면 `deleted_at` 컬럼만 업데이트할 것인가?

소프트 삭제를 사용하면 나중에 데이터를 복구할 수 있고, 통계 분석도 가능하다. 하지만 모든 쿼리에 `WHERE deleted_at IS NULL` 조건을 붙여야 한다.

4.3 ERD 예시

```
# 주요 테이블

## users (사용자)
- id (PK, UUID)
- email (UK)
- password_hash
- nickname (UK)
- profile_image_url
- total_wins (비정규화)
- total_games (비정규화)
```

```
- created_at
- updated_at
- deleted_at (소프트 삭제)

## matches (매칭/게임)
- id (PK, UUID)
- match_type (ENUM: ONE_VS_ONE, TWO_VS_TWO, DEATH_MATCH)
- status (ENUM: WAITING, IN_PROGRESS, COMPLETED, CANCELLED)
- winner_id (FK → users.id)
- started_at
- ended_at
- created_at

## match_participants (매칭 참가자)
- id (PK, UUID)
- match_id (FK → matches.id)
- user_id (FK → users.id)
- team (ENUM: TEAM_A, TEAM_B)
- final_laugh_gauge (INT, 0-100)
- is_winner (BOOLEAN)

## laugh_events (웃음 이벤트)
- id (PK, UUID)
- match_id (FK → matches.id)
- user_id (FK → users.id)
- laugh_intensity (INT, 0-100)
- timestamp
- created_at

INDEX:
- matches(status, created_at) - 대기 중인 매칭 조회
- match_participants(user_id) - 사용자 전적 조회
- laugh_events(match_id, timestamp) - 게임 리플레이
```

4.4 ERD 도구

dbdiagram.io

간단한 DSL 문법으로 ERD를 그릴 수 있는 온라인 도구다. 팀원들과 공유하기도 쉽다.

ERDCloud

한국어를 지원하고, 직관적인 UI를 제공한다. 무료 플랜으로도 충분히 사용할 수 있다.

(전 이거 많이 씁니다 핳)

---

5. 코드 컨벤션

5.1 왜 코드 컨벤션이 필요할까?

코드 컨벤션이 없으면 각자 자기 스타일대로 코드를 작성한다. 어떤 사람은 스네이크 케이스를 쓰고, 어떤 사람은 카멜 케이스를 쓴다. 어떤 사람은 탭을 쓰고, 어떤 사람은 스페이스를 쓴다.

이런 차이들이 쌓이면 코드 리뷰가 불필요하게 길어진다. "이건 왜 이렇게 썼나요?"라는 질문에 "제가 평소에 이렇게 쓰는데요"라는 답변이 나온다.

코드 컨벤션이 있으면 이런 불필요한 논쟁을 줄일 수 있다.

5.2 코드 컨벤션에 포함할 내용

네이밍 규칙

```
## 네이밍 규칙 (자바)

### 변수명
- 카멜 케이스 사용: `userName`, `matchId`
- 불리언은 `is`, `has`, `can`으로 시작: `isActive`, `hasWon`
- 상수는 대문자 스네이크 케이스: `MAX_LAUGH_GAUGE`, `DEFAULT_TIMEOUT`

### 함수명
- 동사로 시작: `getUser()`, `createMatch()`, `validateToken()`
- 불리언 반환은 `is`, `has`, `can`으로 시작: `isValidEmail()`, `hasPermission()`

### 클래스명
- 파스칼 케이스: `UserService`, `MatchController`
- 인터페이스는 `I` 접두사 없이: `UserRepository` (not `IUserRepository`)
```

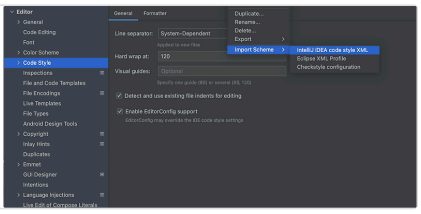
포맷팅 규칙

포맷팅 규칙은 사실 팀마다 너무 상이하고 사람마다 너무 상이하기에 하나의 포맷팅 컨벤션 결정하고, 플러그인 같은 도구를 통해 자동정렬 시키는 편이 훨씬 간편하다.

[Intellij] Naver Java 코딩 컨벤션 적용 방법

코딩 컨벤션 코딩 컨벤션이란 가독성이 좋고 관리하기 쉬운 코드를 작성하기 위한 코딩 스타일 규약을 말합니다. 각자 코딩 스타일이 다르겠지만 팀원 모두 코딩 컨벤션을 준수할 경우 가독성이 좋아지고, 성능에 영향을 주거나 오류를 발생시키는 잠재적 위험 요소를 줄여줍니다. 특히

https://jaimemin.tistory.com/2351



주석 작성 규칙

정말 잘 설명된 주석은 코드 이해가 매우매우매우 큰 도움을 주는건 사실이다. 하지만 무조건적인 주석 남발은 가독성에 있어 좋지 않다. 그래도 안쓰는거보단 나으니 조금이라도 써보자..!

```
## 주석

### 언제 주석을 쓸까?
- 복잡한 비즈니스 로직 설명
- 외부 API 연동 시 스펙 참조
- 임시 해결책(workaround)일 때 TODO와 함께

### 언제 주석을 쓰지 말아야 할까?
- 코드 자체로 명확한 경우
- 변수명/함수명이 의미를 잘 전달하는 경우

### 주석 예시
```javascript
// Bad: 불필요한 주석
// 사용자 이름을 가져온다
const userName = user.name;

// Good: 왜 이렇게 했는지 설명
// WebRTC 연결 시 iOS Safari 버그 회피를 위한 딜레이
// https://bugs.webkit.org/show_bug.cgi?id=179363
await delay(100);
```

6. Git 세팅

6.1 브랜치 전략

Git Flow vs GitHub Flow

Git Flow는 복잡하지만 체계적이다. `main`, `develop`, `feature`, `release`, `hotfix` 브랜치를 사용한다. 대규모 프로젝트나 정기적인 릴리스가 있는 경우 적합하다.

GitHub Flow는 단순하다. `main`브랜치와 `feature`브랜치만 사용한다. 작은 팀이나 빠른 배포 주기를 가진 경우 적합하다.

원브랜치 전략이 훨씬 빠를수도 있지만, 관리가 쉽진 않다..

브랜치 네이밍 규칙

```
markdown
## 브랜치 네이밍

### 기능 개발
feature/기능명
```
```

```
예: feature/user-login-23, feature/match-system-12

### 버그 수정
fix/버그명
예: fix/login-error-21, fix/websocket-disconnect-44

### 핫픽스 (긴급 수정)
hotfix/이슈명
예: hotfix/critical-crash-11

### 리팩토링
refactor/대상
예: refactor/user-service-22

### 문서
docs/내용
예: docs/api-spec, docs/readme-55
```

6.2 커밋 메시지 규칙

Conventional Commits

구조화된 커밋 메시지를 사용하면, 나중에 변경 이력을 추적하기 쉽다.

```
## 커밋 메시지 형식

(이모지)<타입>(<스코프>): <제목>

<본문>

<푸터>

### 타입
- feat: 새로운 기능
- fix: 버그 수정
- docs: 문서 변경
- style: 코드 포맷팅 (기능 변경 없음)
- refactor: 리팩토링
- test: 테스트 추가/수정
- chore: 빌드, 설정 변경

### 예시
✨ feat(auth): 카카오 소셜 로그인 구현

- OAuth 2.0 흐름 구현
- 카카오 API 연동
- JWT 토큰 발급
```

```
Closes #123
```

6.3 Pull Request 규칙

PR 템플릿

```
## Pull Request 템플릿

### 변경 사항
이 PR에서 무엇을 변경했나요?

### 변경 이유
왜 이 변경이 필요한가요?

### 테스트 방법
어떻게 테스트했나요?

### 스크린샷 (선택)
UI 변경이 있다면 스크린샷을 첨부하세요.

### 체크리스트
- [ ] 코드 컨벤션을 따랐습니다
- [ ] 테스트를 작성했습니다
- [ ] 문서를 업데이트했습니다
- [ ] Breaking Change가 있다면 명시했습니다

### 관련 이슈
Closes #이슈번호
```

코드 리뷰 규칙

최소 1명 이상의 승인을 받아야 머지할 수 있도록 설정해야한다. 코드 리뷰는 48시간 이내에 하는 것을 원칙으로 한다.

그리고 PR 검토에 있어 AI도 넣어두면 좋다..! (경험담)

6.4 .gitignore 설정

환경별 설정 파일, 빌드 결과물, IDE 설정 파일 등은 Git에 올리지 않는다.

.env 와 같은 파일은 절대절대절대 올리면 안된다.

```
# 리액트 기준
# 환경 변수
.env
.env.local
.env.production
```

```
# 빌드 결과물
/build
/dist
/target

# 의존성
node_modules/
.pnp
.pnp.js

# IDE
.idea/
.vscode/
*.swp
*.swo

# OS
.DS_Store
Thumbs.db

# 로그
*.log
npm-debug.log*
```

---

7. 패키지 구조

7.1 백엔드 패키지 구조

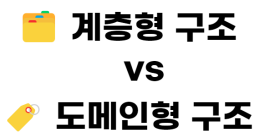
패키지 구조화 방식은 레이어드(계층형), 도메인형, 헥사고날 등등 여러개가 존재한다.

이중 프로젝트에 맞게 선택하는 편인데 개인적으로는 도메인형을 선호한다.

[Spring Boot] 패키지 구조: 계층형 vs 도메인형

규모가 있는 프로젝트를 진행하다 보면 직면하게 되는 문제가 있다. 바로 내가 맡은 기능의 코드를 찾기가 까다롭다는 것이다. 수많은 Service 중 UserService를 찾고, Domain 패키지에서 User를 찾는 것은 꽤 불편한 일이다. 이것들은 모두 계층형 구조

https://velog.io/@chanmi125/Spring-Boot-%ED%8C%A8%ED%82%A4%EC%A7%80-%EA%B5%AC%EC%A1%B0-%EA%B3%84%EC%B8%B5%ED%98%95-vs-%EB%8F%84%EB%A9%94%EC%9D%B8%ED%98%95



7.2 프론트엔드 디렉토리 구조 (AI)

(작성예정)

---

8. Docker Compose 설정

8.1 왜 Docker Compose를 사용할까?

개발 환경을 세팅하는 데 드는 시간을 줄일 수 있다. 새로운 팀원이 합류했을 때, 컴퓨터가 달라지거나 환경이 달라졌을때 "MySQL 설치하고, Redis 설치하고..." 같은 과정을 거치지 않아도 된다. 그냥 `docker-compose up` 명령 하나면 모든 서비스가 실행된다.

또한 개발 환경과 운영 환경을 최대한 비슷하게 만들 수 있다. "내 컴퓨터에서는 잘 되는데요?"라는 상황을 줄일 수 있다.

8.2 Docker Compose 예시

```
version: '3.8'

services:
  # Redis
  redis:
    image: redis:7-alpine
    container_name: gagbattle-redis
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Backend API
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile.dev
    container_name: gagbattle-backend
    environment:
      SPRING_PROFILES_ACTIVE: dev
      DB_HOST: postgres
      DB_PORT: 5432
      REDIS_HOST: redis
      REDIS_PORT: 6379
    ports:
      - "8080:8080"
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
    volumes:
      - ./backend:/app
      - /app/target  # 빌드 결과물은 제외
```

```
    command: mvn spring-boot:run

  # Frontend
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile.dev
    container_name: gagbattle-frontend
    environment:
      VITE_API_URL: http://localhost:8080
      VITE_WS_URL: ws://localhost:8080
    ports:
      - "3000:3000"
    volumes:
      - ./frontend:/app
      - /app/node_modules
    command: npm run dev

volumes:
  postgres-data:
  redis-data:
```

8.3 Docker 사용 시 주의사항

볼륨 마운트

소스 코드를 컨테이너에 볼륨으로 마운트하면, 코드를 수정할 때마다 컨테이너를 재시작하지 않아도 된다. 하지만 `node_modules`나 빌드 결과물은 제외해야 한다. 호스트와 컨테이너의 OS가 다를 수 있기 때문이다.

헬스 체크

`depends_on`만 사용하면, MySQL 컨테이너가 시작되자마자 백엔드가 연결을 시도한다. 하지만 MySQL이 완전히 준비되려면 몇 초가 걸린다. `healthcheck`를 추가하면 실제로 준비가 완료된 후에 연결할 수 있다.

---

9. 배포 전략

9.1 환경 분리

개발(Development), 스테이징/테스트(Staging), 프로덕션(Production)

최소한 세 개의 환경이 필요하다.

개발 환경은 개발자들이 자유롭게 실험하는 곳이다. 여기서는 뭘 해도 상관없다. 데이터베이스를 날려도, 서버를 다운시켜도 괜찮다.

스테이징 환경은 프로덕션과 최대한 비슷하게 만든 환경이다. 실제 배포 전에 여기서 테스트한다. QA팀이나 PM이 여기서 기능을 확인한다.

프로덕션 환경은 실제 사용자가 사용하는 환경이다. 여기서는 절대 실험하지 않는다.

```
## 환경별 설정

### Development
- 도메인: dev.gagbattle.com
- DB: 개발용 MySQL
- 로그 레벨: DEBUG
- 외부 API: Sandbox/Test 모드

### Staging
- 도메인: staging.gagbattle.com
- DB: 스테이징용 MySQL (프로덕션 데이터 복제)
- 로그 레벨: INFO
- 외부 API: Sandbox/Test 모드

### Production
- 도메인: gagbattle.com
- DB: 프로덕션 MySQL (백업 자동화)
- 로그 레벨: WARN
- 외부 API: Production 모드
```

9.2 배포 승인 흐름

Trunk-Based Development + Feature Flags

`main` 브랜치는 항상 배포 가능한 상태를 유지한다. 새로운 기능을 개발할 때는 Feature Flag를 사용해서, 코드는 머지하되 기능은 숨긴다.

```
## 배포 프로세스

1. 개발자가 feature 브랜치에서 작업
2. PR 생성 및 코드 리뷰
3. CI 통과 확인 (테스트, 린트, 빌드)
4. main 브랜치에 머지
5. 자동으로 Development 환경에 배포
6. QA 팀이 Staging 환경에서 테스트
7. 승인되면 Production 배포 (수동 승인)
```

배포 승인자

프로덕션 배포는 반드시 테크 리드나 PM의 승인을 받아야 한다. GitHub Actions에서 `environment` 기능을 사용하면 이를 자동화할 수 있다.

9.3 Blue-Green 배포

다운타임 없이 배포하려면 Blue-Green 배포를 고려해야 한다.

현재 운영 중인 서버를 Blue라고 하자. 새 버전을 Green 서버에 배포한다. Green 서버가 정상적으로 작동하는지 확인한 후, 로드밸런서를 Green으로 전환한다. 문제가 생기면 즉시 Blue로 롤백할 수 있다.

```
## Blue-Green 배포 과정

1. 현재 Blue 환경이 서비스 중
2. Green 환경에 새 버전 배포
3. Green 환경 헬스 체크 (자동화)
4. 로드밸런서를 Green으로 전환 (점진적 트래픽 전환)
5. Blue 환경 모니터링
6. 문제 없으면 Blue 환경 종료
7. 다음 배포 시 Blue와 Green 역할 교체
```

9.4 모니터링

무엇을 모니터링할까?

서비스가 운영되면 실시간으로 상태를 확인할 수 있어야 한다. 문제가 생겼을 때 사용자가 불평하기 전에 먼저 알아차려야 한다.

```
## 모니터링 항목

### 인프라 메트릭
- CPU 사용률 (80% 이상 알림)
- 메모리 사용률 (85% 이상 알림)
- 디스크 사용률 (90% 이상 알림)
- 네트워크 트래픽

### 애플리케이션 메트릭
- API 응답 시간 (평균, P95, P99)
- 에러율 (5% 이상 알림)
- 동시 접속자 수
- 매칭 대기 시간
- WebRTC 연결 성공률

### 비즈니스 메트릭
- 신규 가입자 수
- 일일 활성 사용자 (DAU)
- 게임 완료율
- 평균 게임 시간
```

도구 선택

초기 단계에서는 무료 도구들로 충분하다.

- **Prometheus + Grafana**: 메트릭 수집 및 시각화
- **Sentry**: 에러 추적 및 알림

- **ELK Stack** (Elasticsearch, Logstash, Kibana): 로그 수집 및 분석
- **UptimeRobot**: 서비스 가용성 모니터링 (무료 플랜 제공)

9.5 로그 관리

로그 레벨 전략

```
## 로그 레벨

### ERROR
- 즉각 조치가 필요한 심각한 오류
- 예: DB 연결 실패, 결제 실패, 웃음 감지 AI 크래시

### WARN
- 문제의 징후이지만 서비스는 동작
- 예: 매칭 대기 시간 초과, 외부 API 응답 지연

### INFO
- 중요한 비즈니스 이벤트
- 예: 사용자 로그인, 매칭 성공, 게임 완료

### DEBUG
- 개발 중 디버깅용 (프로덕션에서는 비활성화)
- 예: 상세한 요청/응답 데이터
```

구조화된 로그

단순한 문자열 로그보다는 JSON 형태로 구조화된 로그를 남기는 게 좋다. 나중에 로그를 분석하거나 검색할 때 훨씬 편하다.

```
{
  "timestamp": "2026-01-12T10:30:45.123Z",
  "level": "INFO",
  "service": "match-service",
  "traceId": "abc123def456",
  "userId": "user-uuid-1234",
  "event": "match_created",
  "matchId": "match-uuid-5678",
  "matchType": "ONE_VS_ONE",
  "message": "New match created successfully"
}
```

로그 보관 정책

로그는 무한정 쌓이면 저장 공간을 차지한다. 보관 기간을 정해야 한다.

```
## 로그 보관 정책

- ERROR 로그: 90일 보관
- WARN 로그: 30일 보관
- INFO 로그: 7일 보관 (핫 스토리지), 이후 아카이빙
- DEBUG 로그: 프로덕션에서는 수집하지 않음
```

9.6 알림 설정

알림 채널

Slack이나 Discord 같은 팀 메신저로 알림을 받는 게 좋다. 이메일은 확인이 늦을 수 있다.

```
## 알림 설정

### Critical (즉시 조치 필요)
- 서비스 다운
- DB 연결 실패
- 에러율 10% 이상
- 채널: Slack #alerts-critical, 전화/SMS

### High (빠른 조치 필요)
- API 응답 시간 2초 이상 (5분 지속)
- 메모리 사용률 85% 이상
- 웃음 감지 실패율 20% 이상
- 채널: Slack #alerts-high

### Medium (확인 필요)
- 매칭 대기 시간 30초 이상
- 외부 API 응답 지연
- 채널: Slack #alerts-medium

### Low (참고)
- 신규 버전 배포 완료
- 일일 통계 리포트
- 채널: Slack #general
```

알림 피로 방지

너무 많은 알림을 보내면 정작 중요한 알림을 놓칠 수 있다. 같은 알림이 5분 내에 10번 이상 발생하면 그룹핑해서 한 번만 보내는 게 좋다.

(저흰 MM을 쓰지만, 회사 들어가시면 Slack 알림은 꼭 다들 켜두셔야 함다.)

---

10. 문서화 체크리스트

마지막으로 문서화 체크리스트를 정리해보자.

```
## 문서화 체크리스트

### 필수 문서
- [ ] README.md (프로젝트 소개, 실행 방법)
- [ ] 아키텍처
- [ ] 기술 명세서 (PM)
- [ ] API 명세서 (+Swagger/Postman)
- [ ] ERD
- [ ] 코드 컨벤션
- [ ] Git 워크플로우
- [ ] 배포 가이드

### 권장 문서
- [ ] 아키텍처
- [ ] 트러블슈팅
- [ ] 회고록 (스프린트마다)
```

마치며

여기까지 읽었다면, "와, 이렇게 많은 걸 다 해야 하나?"라는 생각이 들 수 있다.

맞다. 많다. 하지만 이 모든 것을 한 번에 완벽하게 할 필요는 없다.

우선순위를 정하자

1단계로 반드시 해야 할 것: 기술 스택 선정, API 명세서, ERD, 코드 컨벤션, Git 세팅

2단계로 빠르게 해야 할 것: Docker Compose 설정, 기본 CI/CD, 환경 분리

3단계로 점진적으로 개선할 것: 로깅 및 모니터링, 여러 배포 방식, 좀 더 상세한 문서화 및 정리정돈 등

완벽함보다 실행

개인적으로 할 일을 처리하거나 프로젝트를 진행하면서 시도를 망설였던 경험이 많다.

뭔가 주어진 일에 대해 완벽하게 처리하고 싶은 마음에 여러 기술들을 알아보고, 너무 많이 학습해야해서 주눅들게 되고 시도조차 안하는 경험이 많았다..

이 모든 준비가 완벽해질 때까지 기다리지 말자. 기본적인 것들만 갖춰지면 개발을 시작해도 된다. 개발하면서 필요한 것들을 추가하고 개선하면 된다.

다만 명심하자. 지금 30분 투자해서 API 명세서를 작성하면, 나중에 3시간을 절약할 수 있다. 지금 1시간 투자해서 CI/CD를 설정하면, 나중에 수십 시간을 절약할 수 있다. ~~진짭니다.~~