3-1. 백엔드 시작 - Part 1: 개발 시작 전 반드시 정해야 할 것들

들어가며

프론트엔드 개발자에게 물어보면, 디자인 시안만 있으면 바로 코드를 작성할 수 있다고 한다. 하지만 백엔드는 다르다.

"어떤 데이터베이스를 쓸까?"
"API는 어떤 형태로 만들까?"
"사용자 인증은 어떻게 처리하지?"

이런 질문들에 답을 하지 않고 코드부터 작성하면, 나중에 엄청난 혼란이 찾아온다. 개발 2주 차에 "아, 이렇게 하면 안 되는데..."를 깨닫는 순간, 이미 작성한 코드를 모두 뜯어고쳐야 한다.

더 큰 문제는 팀원마다 다른 그림을 그리고 있다는 것이다. 한 사람은 RESTful API를 만들고 있고, 다른 사람은 GraphQL을 만들고 있다. 한 사람은 MongoDB를 쓰려고 하고, 다른 사람은 PostgreSQL을 쓰려고 한다.

이 가이드는 총 3부작으로 구성되어 있다.

- **Part 1: 개발 시작 전 반드시 정해야 할 것들** (이 문서)
- **Part 2: 개발하면서 정해도 되는 것들**
- **Part 3: 개발 시작**

Part 1에서는 코드를 한 줄도 작성하기 전에 팀 전체가 합의해야 하는 것들을 다룬다. 이것들을 정하지 않고 개발을 시작하면, 나중에 엄청난 시간 낭비와 혼란을 겪게 된다.

---

1. 기술 명세서 작성

1.1 왜 가장 먼저 해야 하나?

기술 명세서는 "우리가 무엇을 만들 것인가"가 아니라 **"어떻게 만들 것인가"**를 정의하는 문서다.

이게 없으면 팀원마다 다른 시스템을 상상한다. 어떤 사람은 "당연히 PostgreSQL을 쓸 거야"라고 생각하는데, 다른 사람은 "MongoDB가 더 나을 것 같은데?"라고 생각한다.

프론트엔드 개발자는 RESTful API를 기대하는데, 백엔드 개발자는 GraphQL로 만들고 있을 수 있다. 이런 불일치는 개발 시작 2주 후에 발견되는 경우가 많다. 그때는 이미 작성한 코드를 모두 폐기해야 한다.

1.2 무엇을 포함해야 하나?

1) 시스템 아키텍처 다이어그램

말로 설명하면 10분 걸릴 것을 다이어그램 하나면 10초 만에 이해시킬 수 있다.

```
[사용자 브라우저]
  ↓ HTTPS
[CloudFront (CDN)]
  ↓
[ALB (로드밸런서)]
  ↓
[API 서버 (Spring Boot)] ←→ [Redis]
  ↓
[RDS (PostgreSQL)]
```

다이어그램만으로는 부족하다. 각 컴포넌트가 **왜 필요한지** 설명을 추가한다.

"CloudFront는 왜 필요한가? → 정적 리소스 캐싱으로 서버 부하 감소, HTTPS 인증서 관리"
"Redis는 왜 필요한가? → 매칭 대기 큐, JWT 블랙리스트, 실시간 랭킹"

2) 기술 스택과 선택 이유

단순히 "Spring Boot를 쓴다"가 아니라, **왜 Spring Boot인가?**를 설명해야 한다.

```
## Spring Boot 3.2 (Java 17)

선택 이유:
- 팀 역량: 4명 중 3명이 Spring 경험 보유
- 생태계: Spring Security, Spring Data JPA로 빠른 개발
- 안정성: 엔터프라이즈급 검증된 프레임워크
- 커뮤니티: 문제 해결 자료가 풍부

고려했지만 선택하지 않은 것:
- Node.js: 비동기 처리 강력하지만 팀 역량 부족
- Django: Python은 AI 연동 유리하지만 대용량 트래픽 경험 부족
```

선택한 이유만큼 **선택하지 않은 이유**도 중요하다. 나중에 "왜 Node.js 안 썼어요?"라는 질문에 명확히 답할 수 있다.

3) 성능 요구사항을 구체적인 숫자로

"빨라야 한다"는 의미가 없다. "평균 200ms 이하"가 의미 있다.

```
API 응답 시간:
- 평균: 200ms 이하
- P95: 500ms 이하

동시 접속자:
- Phase 1 (MVP): 1,000명
- Phase 2 (성장기): 10,000명

매칭 대기 시간:
- 10초 이내 매칭 성공률 80% 이상
```

이런 숫자가 있어야 나중에 "우리가 목표를 달성했는가?"를 판단할 수 있다.

4) 보안 정책을 미리 정의

나중에 `"아차!"` 하지 않으려면 미리 정해둬야 한다.

```
통신: HTTPS/WSS 필수
인증: JWT (Access Token 15분, Refresh Token 7일)
비밀번호: bcrypt 해싱 (cost factor 12)
개인정보: AES-256 암호화
Rate Limiting: 사용자당 분당 100회
```

보안은 나중에 추가하기 어렵다. 처음부터 설계에 포함되어야 한다.

5) 핵심 기능의 데이터 흐름

"매칭은 어떻게 작동하나?"를 글로 설명하면 복잡하지만, 흐름도로 그리면 명확해진다.

```
[사용자 A] "매칭 시작" 버튼 클릭
   ↓
[API 서버] Redis 매칭 큐에 추가
   ↓
[스케줄러] 3초마다 큐 확인
   ↓
[사용자 B도 대기 중] 매칭 성립
   ↓
[PostgreSQL] Match 레코드 생성
   ↓
[WebSocket] 양쪽에 알림
```

이런 흐름이 있으면 개발할 때 "다음은 뭘 해야 하지?"를 고민할 필요가 없다.

1.3 기술 명세서가 없으면 생기는 일

실제로 어떤 팀에서 겪은 일이다.

기술 명세서 없이 개발을 시작했다. 프론트엔드 개발자는 RESTful API를 기대했는데, 백엔드 개발자는 GraphQL로 만들고 있었다. 2주 후에 통합 단계에서 이 사실을 발견했다.

"왜 GraphQL로 만들었어요?"
"더 유연하잖아요."
"하지만 우리 프론트는 REST로 설계했는데요..."

결국 백엔드를 처음부터 다시 만들어야 했다. 2주 치 작업이 날아갔다.

기술 명세서가 있었다면? "API는 RESTful로 한다"는 한 줄이 이 재앙을 막았을 것이다.

---

2. 코드 컨벤션 정하기

2.1 왜 코드 컨벤션이 필요한가?

코드 리뷰에서 "왜 이렇게 썼어요?"라는 질문이 계속 나오면, 개발 속도가 느려진다.

A 개발자는 `getUserById`로 함수를 만들었고, B 개발자는 `findUser`로 만들었다. 3개월 후 코드베이스에는 `get`, `find`, `retrieve`, `fetch`가 뒤섞여 있다. 신입 개발자는 "저는 뭘 써야 하나요?"라고 묻지만, 아무도 명확한 답을 주지 못한다.

코드 컨벤션이 있으면 이런 혼란을 막을 수 있다. "조회는 무조건 `get`을 쓴다"고 정하면 끝이다.

2.2 무엇을 정해야 하나?

1) 네이밍 규칙

변수명 규칙

- camelCase 사용
- boolean은 `is`, `has`, `can`으로 시작
- 상수는 UPPER_SNAKE_CASE

함수명 규칙

- 동사로 시작
- CRUD는 `get`, `create`, `update`, `delete`로 통일
- boolean 반환은 `is`, `has`, `can`으로 시작

클래스명 규칙

- PascalCase 사용
- 인터페이스에 `I` 접두사 붙이지 않음
- `Impl` 접미사도 피함

2) 주석 작성 규칙

언제 주석을 쓰는가?

- 복잡한 비즈니스 로직 설명
- 외부 API 스펙 참조
- 임시 해결책(workaround)일 때 TODO와 함께

언제 주석을 쓰지 않는가?

- 코드 자체로 명확한 경우
- 변수명/함수명이 의미를 잘 전달하는 경우

```
// Bad: 코드만 반복
// 사용자 이름을 가져온다
String userName = user.getName();

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

3) 포맷팅은 자동화 도구로

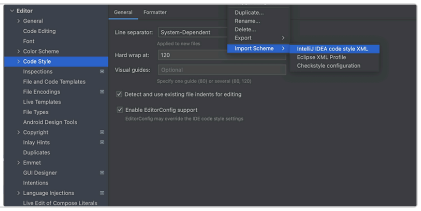
포맷팅으로 논쟁하지 말자. Prettier, Checkstyle 같은 도구로 자동화하면 끝이다.

들여쓰기 2칸인가 4칸인가? 중괄호는 같은 줄인가 다음 줄인가? 이런 걸 회의에서 논쟁하면 시간 낭비다. 도구 설정 파일 하나로 해결한다.

[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



---

3. 디렉토리 구조 선정

3.1 왜 미리 정해야 하나?

디렉토리 구조는 나중에 바꾸기 매우 어렵다. 파일이 100개일 때 구조를 바꾸는 건 괜찮다. 하지만 파일이 1,000개가 되면? 구조를 바꾸는 순간 모든 import 경로가 깨진다.

3.2 선택지와 의사결정

계층형 (Layered)

```
controller/
service/
repository/
```

- 장점: 직관적, 배우기 쉬움
- 단점: 도메인이 파편화됨, User 기능 수정할 때 여러 폴더 오가야 함

도메인형 (Domain-Driven)

```
user/
  - UserController
  - UserService
  - UserRepository
match/
  - MatchController
  - MatchService
```

- 장점: 도메인별로 코드가 모여있어서 찾기 쉬움
- 단점: 초기 러닝 커브

MVP 단계 권장: 도메인형 (하이브리드)

왜 도메인형인가?

- User 기능 수정할 때 `user/` 폴더만 보면 됨
- 나중에 마이크로서비스로 분리하기 쉬움
- 팀 분업도 쉬움 (A팀 User, B팀 Match)

왜 하이브리드인가?

- 도메인 내부는 단순하게 유지
- 처음부터 과도하게 복잡하게 만들지 않음
- 복잡해지면 그때 세분화

---

4. API 설계 원칙과 응답 포맷

4.1 왜 API 설계 원칙이 필요한가?

프론트엔드 개발자가 매번 "이 API는 어떻게 호출해야 하나요?"라고 물어보면 비효율적이다.

"사용자 조회는 GET /users/{id}인가요, POST /getUser인가요?"
"에러 응답은 어떤 형태인가요?"

"페이지네이션은 어떻게 처리하나요?"

이런 질문이 100번 반복되면, 답변하는 데만 하루가 간다.

4.2 RESTful API 원칙

URL은 리소스 중심

- Good: `GET /api/v1/users/{id}`
- Bad: `GET /api/v1/getUser?id=123`

HTTP 메서드로 행위 표현

- GET: 조회
- POST: 생성
- PUT: 전체 수정
- PATCH: 부분 수정
- DELETE: 삭제

상태 코드를 정확히

- 200: 성공
- 201: 생성 성공
- 400: 잘못된 요청
- 401: 인증 필요
- 404: 리소스 없음
- 500: 서버 오류

4.3 응답 포맷 표준화

왜 표준화가 필요한가?

프론트엔드 개발자가 API마다 다른 형태의 응답을 처리해야 한다면? 코드가 복잡해지고 버그가 생긴다.

어떤 API는 `{success: true, data: {...}}`를 반환하고, 어떤 API는 바로 `{id, name, ...}`을 반환하면, 프론트엔드는 API마다 다른 처리 로직을 작성해야 한다.

표준 응답 포맷

```
// 성공
{
  "success": true,
  "data": {...}
}

// 실패
{
  "success": false,
  "error": {
    "code": "USER_NOT_FOUND",
    "message": "사용자를 찾을 수 없습니다."
```

```
  }
}
```

이렇게 정하면 프론트엔드는 항상 `response.success`만 확인하면 된다.

4.4 에러 코드 체계

왜 에러 코드가 필요한가?

HTTP 상태 코드만으로는 부족하다. 404 Not Found라고 해도, "사용자가 없는 건가? 매칭이 없는 건가?"를 알 수 없다.

에러 코드를 체계적으로 만들면 디버깅이 쉬워진다.

```
AUTH-1001: 인증 정보 오류
AUTH-1002: 토큰 만료
USER-2001: 사용자 없음
MATCH-3001: 매칭 없음
```

프론트엔드 개발자가 "MATCH-3001 에러가 났어요"라고 하면, 백엔드 개발자는 바로 어떤 문제인지 안다.

---

5. 사용자 인증 방식 결정

5.1 왜 인증 방식을 미리 정해야 하나?

인증은 나중에 바꾸기 매우 어렵다. 세션 기반으로 만들었다가 JWT로 바꾸려면? 전체 API를 다시 만들어야 한다.

5.2 인증 정보를 어디에 담을 것인가?

선택지

1. Request Body
2. Query Parameter
3. Cookie 헤더
4. Authorization 헤더

왜 Authorization 헤더인가?

Request Body는 GET 요청에서 사용할 수 없다. Query Parameter는 URL에 노출되어 보안에 취약하다. Cookie는 모바일에서 구현이 번거롭다.

Authorization 헤더는 HTTP 표준이고, 모든 요청에 사용 가능하며, 모바일에서도 쉽게 구현할 수 있다.

5.3 JWT vs Session vs OAuth 2.0

Session

- 장점: 구현 간단
- 단점: Stateful (서버에 세션 저장), 확장성 낮음

JWT

- 장점: Stateless (서버에 저장 안 함), 확장성 높음
- 단점: 토큰 탈취 시 만료까지 무효화 어려움

OAuth 2.0

- 장점: 소셜 로그인, 표준
- 단점: 복잡함, HTTPS 필수

MVP 단계 권장: JWT

왜 JWT인가?

- Stateless라서 서버 확장 쉬움
- 모바일 앱에서 사용 편리
- 구현 사례와 라이브러리 풍부

왜 OAuth 2.0이 아닌가?

- HTTPS 비용 문제 (나중에 전환 가능)
- 초기 단계에서는 과도하게 복잡

5.4 Access Token + Refresh Token 전략

왜 두 개의 토큰이 필요한가?

Access Token만 사용하면? 만료 시간을 길게 하면 보안에 취약하고, 짧게 하면 사용자가 자주 로그인해야 한다.

두 개의 토큰을 사용하면?

- Access Token: 15분 (짧게, 보안)
- Refresh Token: 7일 (길게, 편의성)

Access Token이 만료되면 Refresh Token으로 재발급받는다. 사용자는 7일마다 한 번만 로그인하면 된다.

---

6. Git 브랜치 전략 및 커밋 규칙

6.1 왜 브랜치 전략이 필요한가?

브랜치 전략 없이 개발하면? 모두가 main 브랜치에서 작업하고, 충돌이 끊임없이 발생한다.

"누가 내 코드를 지웠어요?"
"이 버그는 누가 만든 거예요?"

Git 히스토리가 엉망이 되면, 문제가 생겼을 때 원인을 찾기 어렵다.

6.2 Git Flow vs GitHub Flow

Git Flow

- main, develop, feature, release, hotfix 브랜치
- 복잡하지만 체계적
- 대규모 프로젝트에 적합

GitHub Flow

- main, feature 브랜치만
- 단순하고 빠름
- 작은 팀, 빠른 배포 주기에 적합

MVP 단계 권장: GitHub Flow

왜 GitHub Flow인가?

- 단순해서 배우기 쉬움
- 빠른 배포 주기에 적합
- 4인 팀에게 충분

6.3 브랜치 네이밍 규칙

```
feature/user-auth      새 기능
fix/jwt-expiration     버그 수정
hotfix/db-connection   긴급 수정
```

이렇게 정하면 브랜치 이름만 봐도 무슨 작업인지 안다.

6.4 커밋 메시지 규칙 (Conventional Commits)

왜 규칙이 필요한가?

커밋 메시지가 제각각이면?

- "수정"
- "버그 고침"
- "ㅇㅇㅇ"

나중에 "어떤 커밋에서 버그가 생겼지?"를 찾을 수 없다.

규칙

```
feat(auth): 카카오 로그인 구현
fix(match): 매칭 타임아웃 버그 수정
docs(api): API 문서 업데이트
```

이렇게 하면 나중에 "로그인 관련 커밋만 보고 싶다"고 할 때 쉽게 찾을 수 있다.

---

7. API 문서화 도구 선택

7.1 왜 API 문서화가 필요한가?

프론트엔드 개발자가 매번 "이 API 어떻게 쓰나요?"라고 물어보면 비효율적이다.

API 문서가 있으면? 프론트엔드 개발자가 스스로 확인할 수 있다. 백엔드 개발자는 개발에만 집중할 수 있다.

7.2 Swagger vs Postman vs Notion

Swagger

- 장점: 코드에서 자동 생성, 실제 API와 문서 항상 일치
- 단점: 초기 설정 필요

Postman

- 장점: 테스트하기 편함

- 단점: 수동 업데이트 필요

Notion

- 장점: 협업 좋음
- 단점: 수동 업데이트, 실제 API와 불일치 가능

MVP 단계 권장: Swagger

왜 Swagger인가?

- 코드에서 자동 생성되어서 항상 최신 상태
- Swagger UI로 바로 테스트 가능
- 프론트엔드가 스스로 확인 가능

---

8. 환경 분리 전략

8.1 왜 환경을 분리해야 하나?

모두가 같은 데이터베이스에서 작업하면?

A 개발자가 테스트하다가 실수로 모든 사용자 데이터를 삭제했다. B 개발자는 "내 데이터가 왜 없어졌어요?"라고 묻는다.

환경을 분리하면 이런 사고를 막을 수 있다.

8.2 Local / Dev / Staging / Production

Local (로컬 개발)

- 개발자 컴퓨터
- 마음대로 실험 가능
- Docker로 DB 실행

Dev (개발 서버)

- 팀 공용 개발 환경
- main 브랜치 머지 시 자동 배포
- 간단한 통합 테스트

Staging (QA 환경)

- 프로덕션과 동일한 환경
- QA팀 테스트
- 운영 데이터 복제본 사용

Production (운영)

- 실제 사용자
- 수동 배포 (PM 승인 필요)
- 장애 시 즉시 대응

각 환경마다 독립적인 데이터베이스와 설정 파일을 사용한다.

Part 1 체크리스트

개발 시작 전에 이것들을 모두 정했는가?

- [ ] 기술 명세서 작성 완료
- [ ] 코드 컨벤션 문서화
- [ ] 디렉토리 구조 합의
- [ ] API 설계 원칙 정립
- [ ] 에러 코드 체계 정의
- [ ] 인증 방식 결정
- [ ] Git 브랜치 전략 합의
- [ ] 커밋 메시지 규칙 정의
- [ ] API 문서화 도구 선택
- [ ] 환경 분리 전략 수립

마치며

"계획에 1시간을 쓰면, 실행에 10시간을 절약할 수 있다."

처음엔 귀찮고 시간 낭비처럼 느껴질 수 있다. "이런 거 나중에 정하면 안 돼?"라는 생각이 들 수도 있다.

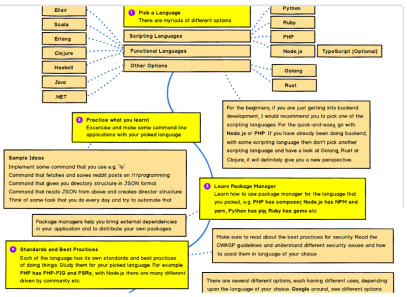
하지만 개발 2주 차에 "아, 이거 처음에 정해뒀어야 하는데..."라고 후회하는 순간, 이미 작성한 코드를 모두 뜯어고쳐야 한다. 그때 드는 시간과 스트레스는 처음 계획에 쓰는 시간의 10배가 넘는다.

지금 30분 투자해서 API 응답 포맷을 정하면, 나중에 "프론트엔드 개발자가 다른 형식을 예상했다"며 API를 전부 수정하는 상황을 막을 수 있다.

지금 1시간 투자해서 에러 코드 체계를 만들면, 나중에 "이 에러가 뭐였지?"라며 코드를 뒤지는 시간을 절약할 수 있다.

준비는 지루하지만, 그 지루함을 견디는 팀이 결국 더 빠르게 달린다.

Part 2에서는 개발하면서 정해도 되는 것들을 다룬다. 성능 최적화, 테스트 전략, 모니터링 같은 것들이다. 🚀

> 마지막으로 확인했으면 좋은 시리즈
>
> 백엔드가 이정도는 해줘야 함 - 1. 컨텐츠의 동기와 개요
>
> 필자가 고등학교 1학년 말에 처음으로 백엔드 포지션에서 프로젝트를 진행하며, 멍청했던 과거 이야기와 이 컨텐츠를 기획한 동기를 공유합니다.
>
> https://velog.io/@city7310/%EB%B0%B1%EC%97%94%EB%93%9C%EA%B0%80-%EC%9D%B4%EC%A0%95%EB%8F%84%EB%8A%94-%ED%95%B4%EC%A4%98%EC%95%BC-%ED%95%A8-1.-%EC%BB%A8%ED%85%90%EC%B8%A0%EC%9D%98-%EB%8F%99%EA%B8%B0%EC%99%80-%EA%B0%9C%EC%9A%94
>
> 
>