3-3. 백엔드 시작 - Part 3: 개발 시작

들어가며

Part 1에서는 개발 시작 전에 반드시 정해야 할 것들을 다뤘다. Part 2에서는 개발하면서 점진적으로 개선해도 되는 것들을 다뤘다.

이제 실제로 개발을 시작할 차례다.

"어디서부터 시작하지?"
"첫 번째 API는 뭘 만들지?"
"데이터베이스는 어떻게 세팅하지?"

Part 3에서는 프로젝트 생성부터 첫 API 배포까지, 실제 개발 과정을 단계별로 다룬다.

---

1. 프로젝트 생성

1.1 Spring Initializr로 시작하기

가장 쉬운 방법은 Spring Initializr를 사용하는 것이다.

start.spring.io에서 선택할 것들

```
Project: Gradle - Kotlin
Language: Java
Spring Boot: 3.2.x (최신 stable 버전)
Java: 17

Group: com.example
Artifact: gagbattle
Name: GagBattle
Package name: com.example.gagbattle
Packaging: Jar
```

의존성 선택

처음부터 모든 의존성을 추가할 필요는 없다. 필요한 것만 추가하고, 나중에 필요하면 추가한다.

```
Spring Web          - REST API
Spring Data JPA     - 데이터베이스
PostgreSQL Driver   - PostgreSQL 연결
Lombok              - 보일러플레이트 코드 감소
Validation          - 입력값 검증
Spring Security     - 인증/인가 (나중에 추가해도 됨)
```

Generate를 누르면 zip 파일이 다운로드된다. 압축을 풀고 IDE에서 연다.

1.2 왜 이렇게 선택했나?

왜 Gradle인가?

Maven보다 빌드 속도가 빠르다. 설정 파일도 더 간결하다. Spring Boot 3.x부터는 Gradle을 권장한다.

왜 Java 17인가?

LTS(Long Term Support) 버전이다. 안정적이고, 3년 이상 지원된다. Java 21도 LTS지만, 라이브러리 호환성이 아직 완벽하지 않다.

왜 Jar 패키징인가?

War는 톰캣 같은 외부 서버가 필요하다. Jar는 내장 톰캣이 포함되어 있어서 `java -jar`로 바로 실행할 수 있다. 배포가 간단하다.

---

2. 로컬 개발 환경 세팅

2.1 Docker Compose로 DB 실행

로컬에 PostgreSQL을 직접 설치하면 팀원마다 버전이 다를 수 있다. Docker Compose를 사용하면 모두가 동일한 환경을 사용한다.

docker-compose.yml 작성

```
version: '3.8'

services:
  postgres:
    image: postgres:15-alpine
    container_name: gagbattle-postgres
    environment:
      POSTGRES_DB: gagbattle
      POSTGRES_USER: gagbattle
      POSTGRES_PASSWORD: password123
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    container_name: gagbattle-redis
    ports:
      - "6379:6379"

volumes:
  postgres-data:
```

실행

```
docker-compose up -d
```

이제 `localhost:5432`로 PostgreSQL에 접속할 수 있다.

2.2 application.yml 설정

Spring Boot는 `src/main/resources/application.yml`에서 설정을 읽는다.

```
spring:
  application:
    name: gagbattle

  datasource:
    url: jdbc:postgresql://localhost:5432/gagbattle
    username: gagbattle
```

```
    password: password123
    driver-class-name: org.postgresql.Driver

  jpa:
    hibernate:
      ddl-auto: update  # 개발: update, 운영: validate
    show-sql: true
    properties:
      hibernate:
        format_sql: true

server:
  port: 8080

logging:
  level:
    com.example.gagbattle: DEBUG
    org.springframework: INFO
```

2.3 왜 이렇게 설정했나?

`ddl-auto: update`는 위험하지 않나?

개발 환경에서는 편리하다. 엔티티를 수정하면 자동으로 테이블이 변경된다. 하지만 운영 환경에서는 절대 사용하면 안 된다. 데이터가 날아갈 수 있다.

운영 환경에서는 `validate`를 사용하고, 스키마 변경은 Flyway나 Liquibase 같은 마이그레이션 도구로 관리한다.

`show-sql: true`는 성능에 영향이 없나?

개발 환경에서는 SQL을 보는 게 유용하다. 하지만 운영 환경에서는 끈다. 로그가 너무 많아진다.

---

3. 디렉토리 구조 만들기

3.1 기본 구조 생성

Part 1에서 도메인형 구조로 결정했다. 이제 실제로 만들어보자.

```
src/main/java/com/example/gagbattle/
├── domain/
│   ├── user/
│   │   ├── User.java
│   │   ├── UserRepository.java
│   │   ├── UserService.java
│   │   ├── UserController.java
│   │   └── dto/
│   │       ├── UserCreateRequest.java
│   │       └── UserResponse.java
│   ├── auth/
│   └── match/
├── common/
```

```
│   ├── config/
│   ├── exception/
│   │   ├── BusinessException.java
│   │   ├── ErrorCode.java
│   │   └── GlobalExceptionHandler.java
│   └── response/
│       ├── ApiResponse.java
│       └── ErrorResponse.java
└── GagBattleApplication.java
```

처음부터 모든 도메인을 만들 필요는 없다. User부터 시작한다.

3.2 왜 이런 구조인가?

왜 dto 패키지를 따로 만드나?

Request와 Response를 Entity와 분리하기 위해서다. Entity를 직접 API 응답으로 주면, 나중에 Entity를 수정할 때 API 스펙이 깨진다.

왜 common 패키지가 필요한가?

모든 도메인에서 공통으로 사용하는 코드를 모아둔다. 예외 처리, 응답 포맷, 설정 같은 것들이다.

---

4. Hello World API 만들기

4.1 간단한 Health Check부터

가장 간단한 API부터 만들어서 서버가 정상 작동하는지 확인한다.

```
package com.example.gagbattle.common.config;

@RestController
@RequestMapping("/api/v1")
public class HealthController {

    @GetMapping("/health")
    public ResponseEntity<Map<String, String>> health() {
        Map<String, String> response = new HashMap<>();
        response.put("status", "UP");
        response.put("timestamp", LocalDateTime.now().toString());
        return ResponseEntity.ok(response);
    }
}
```

서버를 실행하고 `http://localhost:8080/api/v1/health`를 호출하면 응답이 온다.

```
{
  "status": "UP",
  "timestamp": "2026-01-13T15:30:00"
}
```

4.2 왜 Health Check가 필요한가?

배포 후에 서버가 정상적으로 실행됐는지 확인하려면 Health Check 엔드포인트가 필요하다. 로드밸런서(ALB)도 이 엔드포인트를 주기적으로 호출해서 서버가 살아있는지 확인한다.

---

5. 첫 번째 도메인: User

5.1 Entity 작성

```
package com.example.gagbattle.domain.user;

@Entity
@Table(name = "users")
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false, unique = true)
    private String nickname;

    private String profileImageUrl;

    private Integer totalWins = 0;

    private Integer totalGames = 0;

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;

    @Builder
    public User(String email, String password, String nickname) {
        this.email = email;
        this.password = password;
        this.nickname = nickname;
```

```
    }
}
```

5.2 왜 이렇게 작성했나?

왜 `@NoArgsConstructor(access = AccessLevel.PROTECTED)` 인가?

JPA는 엔티티 생성 시 기본 생성자가 필요하다. 하지만 외부에서 `new User()`로 막 만들면 안 되니까 PROTECTED로 제한한다. Builder 패턴으로만 생성하도록 강제한다.

왜 ID가 UUID인가?

Auto Increment는 예측 가능하다. `GET /users/1`, `GET /users/2`로 모든 사용자를 순회할 수 있다. UUID는 예측이 불가능해서 더 안전하다.

왜 totalWins를 User 엔티티에 넣나? 정규화 위반 아닌가?

맞다. 정규화 관점에서는 Match 테이블에서 계산해야 한다. 하지만 "사용자 전적 조회"가 매우 자주 발생한다면, 매번 계산하는 건 비효율적이다. 이건 의도적인 비정규화다. 성능을 위한 trade-off다.

5.3 Repository 작성

```
package com.example.gagbattle.domain.user;

public interface UserRepository extends JpaRepository<User, UUID> {

    Optional<User> findByEmail(String email);

    boolean existsByEmail(String email);

    boolean existsByNickname(String nickname);
}
```

Spring Data JPA는 메서드 이름만으로 쿼리를 자동 생성한다. `findByEmail`은 `SELECT * FROM users WHERE email = ?`로 변환된다.

5.4 Service 작성

```
package com.example.gagbattle.domain.user;

@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class UserService {

    private final UserRepository userRepository;

    public User getUser(UUID userId) {
        return userRepository.findById(userId)
            .orElseThrow(() -> new BusinessException(ErrorCode.USER_NOT_FOUND));
    }
```

```
    @Transactional
    public User createUser(UserCreateRequest request) {
        // 이메일 중복 체크
        if (userRepository.existsByEmail(request.getEmail())) {
            throw new BusinessException(ErrorCode.DUPLICATE_EMAIL);
        }

        // 닉네임 중복 체크
        if (userRepository.existsByNickname(request.getNickname())) {
            throw new BusinessException(ErrorCode.DUPLICATE_NICKNAME);
        }

        // 비밀번호 해싱은 나중에 추가
        User user = User.builder()
            .email(request.getEmail())
            .password(request.getPassword())  // TODO: bcrypt 해싱
            .nickname(request.getNickname())
            .build();

        return userRepository.save(user);
    }
}
```

5.5 왜 이렇게 작성했나?

왜 `@Transactional(readOnly = true)`를 클래스 레벨에?

대부분의 메서드는 조회다. 기본을 readOnly로 하고, 수정이 필요한 메서드에만 `@Transactional`을 붙인다. readOnly는 성능 최적화에 도움이 된다.

왜 비밀번호를 평문으로 저장하나?

지금은 일단 작동하게 만드는 게 목표다. 비밀번호 해싱은 나중에 추가한다. "TODO"로 남겨두면 잊지 않는다.

---

6. DTO와 응답 포맷

6.1 Request DTO

```
package com.example.gagbattle.domain.user.dto;

@Getter
@NoArgsConstructor
public class UserCreateRequest {

    @NotBlank(message = "이메일은 필수입니다")
    @Email(message = "올바른 이메일 형식이 아닙니다")
    private String email;

    @NotBlank(message = "비밀번호는 필수입니다")
    @Size(min = 8, max = 20, message = "비밀번호는 8-20자여야 합니다")
```

```
    private String password;

    @NotBlank(message = "닉네임은 필수입니다")
    @Size(min = 2, max = 10, message = "닉네임은 2-10자여야 합니다")
    private String nickname;
}
```

Validation 어노테이션으로 입력값을 검증한다. 이렇게 하면 Controller에서 if문 지옥을 피할 수 있다.

6.2 Response DTO

```
package com.example.gagbattle.domain.user.dto;

@Getter
@Builder
public class UserResponse {

    private UUID id;
    private String email;
    private String nickname;
    private String profileImageUrl;
    private Integer totalWins;
    private Integer totalGames;
    private LocalDateTime createdAt;

    public static UserResponse from(User user) {
        return UserResponse.builder()
            .id(user.getId())
            .email(user.getEmail())
            .nickname(user.getNickname())
            .profileImageUrl(user.getProfileImageUrl())
            .totalWins(user.getTotalWins())
            .totalGames(user.getTotalGames())
            .createdAt(user.getCreatedAt())
            .build();
    }
}
```

Entity를 Response로 변환하는 `from` 메서드를 만든다. 비밀번호 같은 민감한 정보는 포함하지 않는다.

6.3 공통 응답 포맷

```
package com.example.gagbattle.common.response;

@Getter
@Builder
public class ApiResponse<T> {
```

```
    private boolean success;
    private T data;
    private ErrorResponse error;

    public static <T> ApiResponse<T> success(T data) {
        return ApiResponse.<T>builder()
            .success(true)
            .data(data)
            .build();
    }

    public static ApiResponse<Void> error(ErrorCode errorCode) {
        return ApiResponse.<Void>builder()
            .success(false)
            .error(ErrorResponse.of(errorCode))
            .build();
    }
}
```

모든 API가 같은 형태로 응답한다. 프론트엔드는 항상 `response.success`만 확인하면 된다.

7. Controller 작성

```
package com.example.gagbattle.domain.user;

@RestController
@RequestMapping("/api/v1/users")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;

    @GetMapping("/{id}")
    public ResponseEntity<ApiResponse<UserResponse>> getUser(
        @PathVariable UUID id
    ) {
        User user = userService.getUser(id);
        return ResponseEntity.ok(
            ApiResponse.success(UserResponse.from(user))
        );
    }

    @PostMapping
    public ResponseEntity<ApiResponse<UserResponse>> createUser(
        @Valid @RequestBody UserCreateRequest request
    ) {
        User user = userService.createUser(request);
```

```
        return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(ApiResponse.success(UserResponse.from(user)));
    }
}
```

`@Valid`를 붙이면 Request DTO의 Validation이 자동으로 실행된다.

---

8. 예외 처리

8.1 커스텀 Exception

```
package com.example.gagbattle.common.exception;

@Getter
public class BusinessException extends RuntimeException {

    private final ErrorCode errorCode;

    public BusinessException(ErrorCode errorCode) {
        super(errorCode.getMessage());
        this.errorCode = errorCode;
    }
}
```

8.2 ErrorCode Enum

```
package com.example.gagbattle.common.exception;

@Getter
@RequiredArgsConstructor
public enum ErrorCode {

    // User (2xxx)
    USER_NOT_FOUND("USER-2001", "사용자를 찾을 수 없습니다", HttpStatus.NOT_FOUND),
    DUPLICATE_EMAIL("USER-2002", "이미 사용 중인 이메일입니다", HttpStatus.CONFLICT),
    DUPLICATE_NICKNAME("USER-2003", "이미 사용 중인 닉네임입니다", HttpStatus.CONFLICT),

    // Common
    INVALID_INPUT("COMMON-0001", "잘못된 입력값입니다", HttpStatus.BAD_REQUEST),
    INTERNAL_ERROR("COMMON-9999", "서버 오류가 발생했습니다", HttpStatus.INTERNAL_SERV
ER_ERROR);

    private final String code;
    private final String message;
```

```
    private final HttpStatus status;
}
```

8.3 GlobalExceptionHandler

```
package com.example.gagbattle.common.exception;

@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(BusinessException.class)
    public ResponseEntity<ApiResponse<Void>> handleBusinessException(
        BusinessException e
    ) {
        log.error("Business exception: {}", e.getMessage());
        return ResponseEntity
            .status(e.getErrorCode().getStatus())
            .body(ApiResponse.error(e.getErrorCode()));
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ApiResponse<Void>> handleValidationException(
        MethodArgumentNotValidException e
    ) {
        log.error("Validation exception: {}", e.getMessage());
        return ResponseEntity
            .status(HttpStatus.BAD_REQUEST)
            .body(ApiResponse.error(ErrorCode.INVALID_INPUT));
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiResponse<Void>> handleException(
        Exception e
    ) {
        log.error("Unexpected exception", e);
        return ResponseEntity
            .status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(ApiResponse.error(ErrorCode.INTERNAL_ERROR));
    }
}
```

모든 예외를 한 곳에서 처리한다. Controller에서는 예외를 던지기만 하면 된다.

---

9. 테스트

9.1 간단한 통합 테스트

```
package com.example.gagbattle.domain.user;

@SpringBootTest
@AutoConfigureMockMvc
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private UserRepository userRepository;

    @Test
    void 사용자_생성_성공() throws Exception {
        // given
        String requestBody = """
            {
                "email": "test@example.com",
                "password": "password123",
                "nickname": "테스터"
            }
            """;

        // when & then
        mockMvc.perform(post("/api/v1/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(requestBody))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.success").value(true))
            .andExpect(jsonPath("$.data.email").value("test@example.com"));
    }
}
```

처음에는 모든 케이스를 테스트하지 않아도 된다. "정상 케이스"만 테스트한다. 예외 케이스는 나중에 추가한다.

---

10. 첫 배포

10.1 빌드

```
./gradlew clean build
```

`build/libs/` 디렉토리에 jar 파일이 생성된다.

10.2 실행

```
java -jar build/libs/gagbattle-0.0.1-SNAPSHOT.jar
```

서버가 8080 포트에서 실행된다.

10.3 EC2에 배포 (간단한 방법)

```
# EC2에 접속
ssh -i key.pem ec2-user@ec2-ip

# 파일 업로드
scp -i key.pem build/libs/*.jar ec2-user@ec2-ip:~/

# 실행
java -jar gagbattle-0.0.1-SNAPSHOT.jar
```

nohup으로 백그라운드 실행:

```
nohup java -jar gagbattle-0.0.1-SNAPSHOT.jar > app.log 2>&1 &
```

---

11. 다음 단계

11.1 무엇을 먼저 할까?

인증 구현

로그인, 회원가입, JWT 발급. 이게 없으면 사용자를 구분할 수 없다.

두 번째 도메인 추가

Match 도메인을 추가한다. User와 Match가 있으면 핵심 기능의 골격이 완성된다.

WebSocket 연결

실시간 매칭 알림을 위해 WebSocket을 추가한다.

11.2 하지 말아야 할 것

완벽한 코드를 작성하려고 하지 말자

지금은 "작동하는 코드"가 목표다. 리팩토링은 나중에 한다.

모든 기능을 한 번에 만들려고 하지 말자

작은 기능부터 하나씩 완성한다. "회원가입"이 완성되면 "로그인"으로 넘어간다.

테스트를 너무 많이 작성하지 말자

핵심 로직만 테스트한다. 모든 코드를 테스트하려고 하면 개발 속도가 느려진다.

체크리스트

개발 시작 단계에서 확인할 것들:

환경 세팅

- ☐ Spring Initializr로 프로젝트 생성
- ☐ Docker Compose로 DB 실행
- ☐ application.yml 설정

- [ ] 디렉토리 구조 생성

첫 번째 기능

- [ ] Hello World API 작성
- [ ] User Entity 작성
- [ ] UserRepository 작성
- [ ] UserService 작성
- [ ] UserController 작성
- [ ] 예외 처리 구현
- [ ] 간단한 테스트 작성

배포

- [ ] 빌드 성공 확인
- [ ] 로컬에서 실행 확인
- [ ] 개발 서버에 배포

마치며

완벽하지 않아도 괜찮다.

비밀번호는 평문으로 저장되어 있고, 테스트는 몇 개 없고, 에러 처리도 부족하다. 하지만 **작동한다**.

이게 중요하다. 작동하는 코드가 있으면, 개선할 수 있다. 하지만 작동하지 않는 코드는 개선할 수도 없다.

지금부터는 점진적으로 개선하면 된다.

다음 단계:

1. 비밀번호 해싱 추가 (bcrypt)
2. JWT 인증 구현
3. Match 도메인 추가
4. WebSocket 연결
5. 프론트엔드와 통합