

테스트 결과 보고서(BE)

★ contents ★

- 1 코드 컨벤션
 - [API](#)
 - [코드](#)
- 2 코드 모듈화 & 재사용성
 - [폴더 구조 설계](#)
 - [ERD](#)
- 3 보안
 - [유저별 기능 접근 제한](#)
 - [Access, Refresh Token](#)
 - [이메일 검증](#)
 - [토스 페이먼트와 연결](#)
 - [CORS 설정](#)
 - [SQLDelete](#)
 - [환경 변수 세팅](#)
 - [패스워드 인코딩](#)
- 4 성능
 - [N번의 Insert Query → JDBC Batch Update](#)
 - [인덱스](#)



안녕하세요! 투명한 가격으로 웨딩 플래너와 예비 부부를 연결하는 매칭 플랫폼, **순수웨딩** 팀의 백엔드입니다.

해당 문서에서는 작성한 **코드와 테스트 결과를 중심으로 컨벤션, 모듈화와 재사용성, 보안, 성능** 등을 어떻게 개선하였는지와 테스트 결과에 대해 설명합니다.

회색 기울임 글씨는 관련 코드 위치와 내용입니다.



1 코드 컨벤션

API

? API 구성은 어떻게 되어있나요?

저희 순수웨딩 팀은 기능에 따라 **회원(/user)**, **이메일(/email)**, **결제(/payments)**, **포트폴리오(/portfolios)**, **채팅(/chat)**, **매칭(/match)**, **견적서(/quotations)**, **리뷰(/reviews)**, **찜하기(/favorites)** 의 9개의 카테고리별로 분류하여 API를 구성하였습니다.

! Restful API를 사용했어요

GET, POST, DELETE, PUT 등 적재적소에 알맞는 메서드를 사용하여 서비스의 확장성, 유연성, 독립성을 높였습니다.

? 견적서의 엔드포인트에 chatId는 왜 쿼리문이지?

견적서의 경우 **quotationId**와 **chatId**가 모두 필요하기 때문에 직관성을 위해 quotationId는 path로, chatId는 쿼리로 넘겨주도록 설정하였습니다. ex) /quotations/{quotationId}?chatId={chatId}

코드

4명이 함께 프로젝트를 진행하다보니 **코드 컨벤션**을 맞추는 일이 중요했습니다. 순수웨딩-백엔드 팀의 코드 컨벤션을 정해두고 점점 확장시켰습니다.

📌 순수웨딩 - BE 컨벤션

1. 폴더 이름은 소문자로, 파일명, 변수명, 메서드명은 CamelCase로, 엔티티의 필드네임은 snake_case로, 상수는 UPPER_CASE 로, 테이블 이름은 snake_case_tb 로 통일
2. 클래스 대신 record 사용하기, RequestDTO와 ResponseDTO는 따로 묶고 DTOConverter에서 변환하기
3. CRUD 네이밍 통일 (생성 → save / 조회 → find / 수정 → update / 삭제 → delete)
4. Annotation 순서 설정: 클래스 → 필드 → 생성자 → 메서드
5. Class 전체에 @Transactional(readOnly = true)를 사용하고 필요한 메서드에 @Transactional 사용하기
6. 상태, 등급 등 constant한 값을 가지고 있는 내용은 enum 타입으로 분리하기
7. DB를 잘못 삭제하는 걸 방지하기 위해서 각 table마다 is_active 필드를 통해 SQLDelete활용하기
8. 테스트할 때 Assertions.assertThat, assertThrows는 static import하기
9. 테스트할 때 RepositoryTest는 DummyEntity에서 생성해서, ControllerTest는 teardown.sql에서 가져와서 테스트 하기
10. 테스트할 때 logResult() 함수를 따로 빼서 중복 코드 줄이기

2 코드 모듈화 & 재사용성

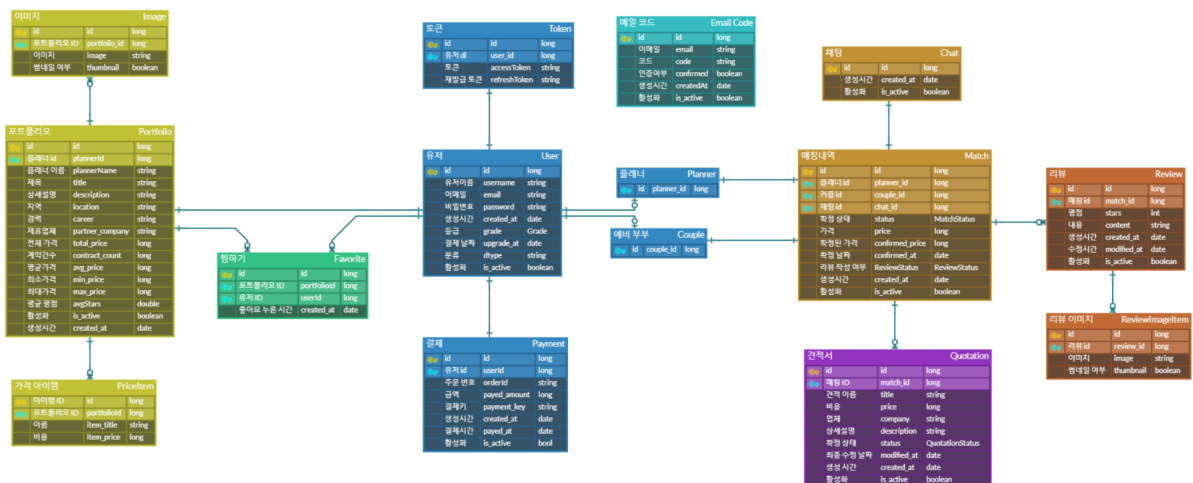
폴더 구조 설계

```
- Entity.java
- DTOConverter.java
- JPARepository.java
- Request.java
- Response.java
- RestController.java
- Service.java
- ServiceImpl.java
```



ERD

플래너와 예비 부부를 매칭해야 되기에 User에서 상속받아 분리된 엔티티를 사용하였습니다. 필드 내용은 거의 동일하기에 Single Table 상속 전략을 사용하였습니다.



3 보안

유저별 기능 접근 제한

▼ 코드: `_core > config > SecurityConfig.java`

```
// 11. 인증, 권한 필터 설정
http.authorizeHttpRequests((authorizeHttpRequests) ->
    authorizeHttpRequests
        .requestMatchers(
            new AntPathRequestMatcher("/api/portfolio/self", "GET")
        ).hasAuthority("planner")
        .requestMatchers(
            new AntPathRequestMatcher("/api/user/signup"),
            new AntPathRequestMatcher("/api/user/login"),
            new AntPathRequestMatcher("/api/portfolio/**", "GET"),
            new AntPathRequestMatcher("/api/mail/**")
        ).permitAll()
        .requestMatchers(
            new AntPathRequestMatcher("/api/chat"),
            new AntPathRequestMatcher("/api/match/**"),
            new AntPathRequestMatcher("/api/review/all", "GET"),
            new AntPathRequestMatcher("/api/review/{reviewId}", "GET"),
            new AntPathRequestMatcher("/api/review/**", "POST"),
            new AntPathRequestMatcher("/api/review/**", "PUT"),
            new AntPathRequestMatcher("/api/review/**", "DELETE")
        ).hasAuthority("couple")
        .requestMatchers(
            new AntPathRequestMatcher("/api/portfolio", "POST"),
            new AntPathRequestMatcher("/api/portfolio", "PUT"),
            new AntPathRequestMatcher("/api/portfolio", "DELETE"),
            new AntPathRequestMatcher("/api/quotation/**", "PUT"),
            new AntPathRequestMatcher("/api/quotation/**", "POST"),
            new AntPathRequestMatcher("/api/quotation/**", "DELETE")
        ).hasAuthority("planner")
        .requestMatchers(
            new AntPathRequestMatcher("/api/user/**"),
            new AntPathRequestMatcher("/api/portfolio/**"),
            new AntPathRequestMatcher("/api/chat/**"),
            new AntPathRequestMatcher("/api/match/**"),
            new AntPathRequestMatcher("/api/quotation/**"),
            new AntPathRequestMatcher("/api/payment/**"),
            new AntPathRequestMatcher("/api/review/all", "GET"),
            new AntPathRequestMatcher("/api/review/{reviewId}", "GET"),
            new AntPathRequestMatcher("/api/review/**", "POST"),
            new AntPathRequestMatcher("/api/review/**", "PUT"),
            new AntPathRequestMatcher("/api/review/**", "DELETE"),
            new AntPathRequestMatcher("/api/favorite/**")
        ).authenticated()
        .anyRequest().permitAll()
);
```

로그인 유무에 따른 기능 접근 제한 - 401 Unauthorized

로그인 필요 X	 회원	회원가입, 로그인
	 메일	인증 코드 전송, 검증
	 포트폴리오	리스트 조회, 상세 조회
로그인 필요 O	 회원	회원탈퇴, 유저 정보 조회, 토큰 갱신
	 결제	결제 데이터 저장, 결제 승인 및 업그레이드
	 포트폴리오	본인 게시글 조회, 등록, 수정, 삭제
	 채팅	채팅방 생성
	 매칭	매칭 확정, 리뷰 작성 가능한 매칭 조회
	 견적서	전체 리스트 조회, 매칭별 리스트 조회, 견적서 등록, 수정, 삭제, 1개 확정
	 리뷰	전체 리스트 조회, 플래너별 리뷰 조회, 상세 조회, 등록, 수정, 삭제
	 찜하기	찜리스트 조회, 등록, 삭제

사용자 role에 따른 기능 접근 제한 - 403 Forbidden

공통	 회원	회원가입, 로그인, 유저 정보 조회, 토큰 갱신, 회원 탈퇴
	 메일	인증 코드 전송, 검증
	 결제	결제 데이터 저장, 결제 승인 및 업그레이드
	 포트폴리오	리스트 조회, 상세 조회
	 견적서	전체 리스트 조회, 매칭별 리스트 조회
	 리뷰	플래너별 리뷰 조회
	 찜하기	찜리스트 조회, 등록, 삭제
플래너	 포트폴리오	본인 게시글 조회, 등록, 수정, 삭제
	 견적서	견적서 등록, 수정, 삭제, 1개 확정
	 채팅	채팅방 생성
	 매칭	매칭 확정, 리뷰 작성 가능한 매칭 조회
예비 부부	 채팅	채팅방 생성
	 매칭	매칭 확정, 리뷰 작성 가능한 매칭 조회
	 리뷰	전체 리스트 조회, 상세 조회, 등록, 수정, 삭제

Access, Refresh Token

▼ 코드: `_core > security`

```
@Component
public class JWTProvider {
    // access-token expire time = 15 min
    public final Long ACCESS_TOKEN_EXP = 1000L * 60 * 15;
    // refresh-token expire time = 3 days
    public final Long REFRESH_TOKEN_EXP = 1000L * 60 * 60 * 24 * 3;
    public final String TOKEN_PREFIX = "Bearer ";
    public final String AUTHORIZATION_HEADER = "Authorization";
    public final String REFRESH_HEADER = "Refresh";

    @Value("${security.jwt-config.secret.access}")
    public String ACCESS_TOKEN_SECRET;

    @Value("${security.jwt-config.secret.refresh}")
    private String REFRESH_TOKEN_SECRET;

    public String createAccessToken(User user) {
        String jwt = create(user, ACCESS_TOKEN_EXP, ACCESS_TOKEN_SECRET);
        return TOKEN_PREFIX + jwt;
    }

    public String createRefreshToken(User user) {
        String jwt = create(user, REFRESH_TOKEN_EXP, REFRESH_TOKEN_SECRET);
        return TOKEN_PREFIX + jwt;
    }

    private String create(User user, Long expire, String secret) {
        return JWT.create()
            .withSubject(user.getEmail())
            .withExpiresAt(new Date(System.currentTimeMillis() + expire))
            .withClaim("id", user.getId())
            .withClaim("role", user.getDtype())
            .sign(Algorithm.HMAC512(secret));
    }

    public DecodedJWT verifyAccessToken(String token) throws SignatureVerificationException, TokenExpiredException {
        return verify(token, ACCESS_TOKEN_SECRET);
    }

    public DecodedJWT verifyRefreshToken(String token) throws SignatureVerificationException, TokenExpiredException {
        return verify(token, REFRESH_TOKEN_SECRET);
    }

    private DecodedJWT verify(String token, String secret) {
        token = token.replace(TOKEN_PREFIX, "");
        return JWT
            .require(Algorithm.HMAC512(secret))
```

```

        .build()
        .verify(token);
    }

    public boolean isValidAccessToken(String token) {
        try {
            verify(token, ACCESS_TOKEN_SECRET);
            return true;
        } catch (JWTVerificationException exception) {
            return false;
        }
    }
}

```



JWT 도입

유저 정보 인증을 위해 JWT를 도입하였습니다. 따라서 유저는 access token을 가지고 인증을 진행하게 됩니다. 이때, access token은 API를 요청할 때마다 헤더에 추가하여 전송되기 때문에 네트워크상에 많이 노출되고, 탈취될 확률도 높습니다.

그리고 유저가 로그인을 자주하는 불편함 없이 오래 사용하기 위해서는 토큰의 유효기간이 길어야 합니다. 하지만 길게 설정하면, 탈취당한 토큰으로 해커는 오랜 기간동안 유저의 정보를 이용할 수 있게 됩니다.

따라서 access token의 유효기간을 짧게 가져가면서 상대적으로 기간이 긴 refresh token을 도입하게 되었습니다.



Refresh Token 도입

1. 유저는 최초 로그인을 하면 access token, refresh token 두가지를 발급받습니다.
2. 유저는 access token만을 가지고 인증을 요청합니다.
3. access token이 만료되면, 서버에서는 토큰 만료 에러를 전송합니다.
4. 이를 받은 클라이언트는 access token과 refresh token을 가지고 토큰 재발급 요청을 합니다.
5. 서버는 두 토큰을 모두 새로 발급하고 유저는 새로 발급받은 두 토큰을 저장 후, 다시 access token을 통해 인증 과정을 거치게 됩니다.

보안을 높이기 위해 탈취당하는 경우를 고민해 보았습니다.

1. 리프레시 토큰만 탈취당한 경우

- 토큰 재발급을 할 때, 이전에 발급했던 access token, refresh token 두개를 모두 사용해야 재발급이 가능하도록 설계했습니다. 따라서 둘 중 하나의 토큰만을 탈취했다면, 재발급이 불가능합니다.
- 리프레시 토큰만 존재하면 인증도 불가능하도록 설계했습니다. 따라서 리프레시 토큰만을 가지고 있다면, 해커는 인증을 진행할 수 없을 것입니다.

2. 모든 토큰이 탈취당한 경우

- 먼저 access token의 유효시간이 남아있다면, 재발급을 받을 수 없도록 설계했습니다.
- 만약 만료 시간에 맞추어 해커가 재발급을 했다면, 서버에 저장되어 있는 토큰 쌍과 정상 유저가 재발급을 요청한 토큰 쌍이 다를 것이기 때문에 정상 유저는 재로그인을 통해 새로운 토큰 쌍을 발급받습니다. 재발급을 받게 되면, 해커의 토큰이 폐기되고 유저는 정상적으로 다시 이용이 가능해집니다. 해커는 유효하지 않은 토큰쌍을 가지고 있으므로 재인증 및 재발급이 불가능합니다.

3. 로그인 시 새로운 JWT 발급

데이터베이스에 저장된 토큰 정보를 다시 준다면, 해커는 타이밍만 맞춘다면 무한정 해킹이 가능해집니다. 따라서 로그인을 할 경우, 기존 토큰 정보를 삭제 후 새로운 토큰 정보를 데이터베이스에 저장하도록 설계했습니다.

4. 재발급 시 모든 토큰을 새로 발급

리프레시 토큰 또한 재발급을 자주 받게 된다면 네트워크에 자연스럽게 많이 노출됩니다. 따라서 토큰을 재발급할 때, access token 및 refresh token 둘 다 새로 발급하도록 설계했습니다.

이메일 검증

▼ 코드: user > mail

```
@Service
@RequiredArgsConstructor
public class EmailServiceImpl implements EmailService {
    private final UserDataChecker userDataChecker;
    private final JavaMailSenderImpl javaMailSender;
    private final EmailCodeJpaRepository emailCodeJpaRepository;

    private final static int CODE_LENGTH = 6;

    private final static int CODE_EXP = 60 * 10;

    String EMAIL_CONTENT = ""
        <body style="margin: 0 0;">
            <div style="vertical-align: middle; text-align: center; font-size: 14px; color: black; margin: 0 0">
                <img style="padding: 100px 0; width: 60vw; max-width: 350px; display: block; margin: 0 auto;"
                <p style="margin: 0 0;">순수웨딩 회원가입 이메일 인증코드입니다.</p>
                <h2 style="margin: 50px 0;">인증코드: <span style="color: #0073C2;">%s</span></h2>
                <p>해당 인증코드는 10분 이내 1회만 사용 가능합니다.<br/>이후에는 인증코드를 다시 요청하셔야 합니다.</p>
                <p>감사합니다.</p>
                <p style="margin-bottom: 0;">- 순수웨딩 -</p>
            </div>
        </body>
    """;

    @Value("${email.username}")
    private String sender;

    @Value("${email.test-code}")
    private Long TEST_CODE;

    @Transactional
    public void send(EmailRequest.SendCode request) {
        userDataChecker.checkEmailAlreadyExist(request.email());

        // 크래폴린 내부 정책으로 인한 smtp 프로토콜 사용 불가로
        // 이메일 인증 코드를 테스트 코드로 대체 및 이메일 발송 제외
        // String code = generateCode();
        String code = TEST_CODE.toString();
        // MimeMessage email = createMail(request.email(), code);
        // javaMailSender.send(email);

        EmailCode emailCode = findMailCodeByRequest(request);

        checkEmailAlreadyAuthenticated(emailCode);

        emailCode.setCode(code);
        emailCode.setEmail(request.email());
        emailCode.setConfirmed(false);
        emailCode.setCreatedAt(LocalDateTime.now());

        emailCodeJpaRepository.save(emailCode);
    }

    public void verify(EmailRequest.CheckCode request) {
        EmailCode emailCode = findMailCodeByRequest(request);

        checkEmailAlreadyAuthenticated(emailCode);
        checkCodeExpiration(emailCode);
        matchCode(request, emailCode);

        emailCode.setConfirmed(true);
        emailCodeJpaRepository.save(emailCode);
    }

    private EmailCode findMailCodeByRequest(EmailRequest.SendCode request) {
        return emailCodeJpaRepository.findByEmail(request.email())
            .orElseGet(() -> EmailCode.builder().build());
    }

    private EmailCode findMailCodeByRequest(EmailRequest.CheckCode request) {
        return emailCodeJpaRepository.findByEmail(request.email())
            .orElseThrow(() -> new NotFoundException(BaseException.CODE_NOT_FOUND));
    }

    private static void matchCode(EmailRequest.CheckCode request, EmailCode emailCode) {
        if (!Objects.equals(emailCode.getCode(), request.code())) {
            throw new BadRequestException(BaseException.CODE_NOT_MATCHED);
        }
    }
}
```

```

private void checkCodeExpiration(EmailCode emailCode) {
    Duration duration = Duration.between(emailCode.getCreatedAt(), LocalDateTime.now());
    if (duration.getSeconds() > CODE_EXP) {
        emailCodeJpaRepository.delete(emailCode);
        throw new BadRequestException(BaseException.CODE_EXPIRED);
    }
}

private static void checkEmailAlreadyAuthenticated(EmailCode emailCode) {
    if (emailCode.getConfirmed() != null && emailCode.getConfirmed().equals(true)) {
        throw new BadRequestException(BaseException.EMAIL_ALREADY_AUTHENTICATED);
    }
}

private String generateCode() {
    try {
        Random random = SecureRandom.getInstanceStrong();
        StringBuilder builder = new StringBuilder();
        IntStream.range(0, CODE_LENGTH)
            .forEach(i -> builder.append(random.nextInt(10)));

        return builder.toString();
    } catch (NoSuchAlgorithmException exception) {
        throw new ServerException(BaseException.CODE_GENERATE_ERROR);
    }
}

private MimeMessage createMail(String receiver, String code) {
    MimeMessage email = javaMailSender.createMimeMessage();

    try {
        email.setFrom(sender);
        email.setRecipients(MimeMessage.RecipientType.TO, receiver);
        email.setSubject("순수웨딩 회원가입 이메일 인증 코드");

        String body = String.format(EMAIL_CONTENT, code);
        email.setText(body, "UTF-8", "html");
    } catch (MessagingException e) {
        throw new ServerException(BaseException.EMAIL_GENERATE_ERROR);
    }

    return email;
}
}

```



유저 이메일이 유효한지 검증을 할 필요가 있습니다. 하지 않을 경우, 유저는 존재하지 않는 가상의 이메일들을 무한정 만들어 낼 수 있습니다. 이를 방지하기 위하여 회원가입을 할 이메일로 인증 코드를 발송, 해당 인증코드를 입력해야만 가입할 수 있도록 설계했습니다.

하지만 크래프린 내부 정책으로 인한 SMTP 프로토콜 사용 불가로 이메일 발송은 제외하고 테스트 코드로 검증하도록 대체하였습니다.

토스 페이먼트와 연결

▼ 코드: *payment > PaymentServiceImpl.java*

```

@Transactional
public void save(Long userId, PaymentRequest.SaveDTO requestDTO){
    User user = findUserId(userId);
    Optional<Payment> paymentOptional = paymentJpaRepository.findById(userId);

    // 사용자의 결제 정보가 존재하면 업데이트
    if (paymentOptional.isPresent()){
        Payment payment = paymentOptional.get();
        payment.updatePaymentInfo(requestDTO.orderId(), requestDTO.amount());
    }
    else {
        // 결제 정보 저장
        Payment payment = Payment.builder()
            .user(user)
            .orderId(requestDTO.orderId())
            .payedAmount(requestDTO.amount())

```



```

        .build();
        paymentJpaRepository.save(payment);
    }
}

```

```

@Transactional
public void approve(Long userId, PaymentRequest.ApprovedDTO requestDTO) {
    User user = findUserById(userId);
    Payment payment = findPaymentByUserId(user.getId());

    // 1. 검증: 프론트 정보와 백엔드 정보 비교
    Boolean isOK = isCorrectData(payment, requestDTO.orderId(), requestDTO.amount());

    if (!isOK) {
        throw new BadRequestException(BaseException.PAYMENT_WRONG_INFORMATION);
    }

    payment.updatePaymentKey(requestDTO.paymentKey());
    // 2. 토스페이먼츠 승인 요청
    tossPayApprove(requestDTO);
    // 3. 유저 업그레이드
    user.upgrade();
    // 4. 결제시간 업데이트
    payment.updatePaidAt();
}

private void tossPayApprove(PaymentRequest.ApprovedDTO requestDTO){
    // 토스페이먼츠 승인 api 요청
    String basicToken = "Basic " + Base64.getEncoder().encodeToString((secretKey + ":").getBytes());

    WebClient webClient =
        WebClient
            .builder()
            .baseUrl("https://api.tosspayments.com")
            .build();

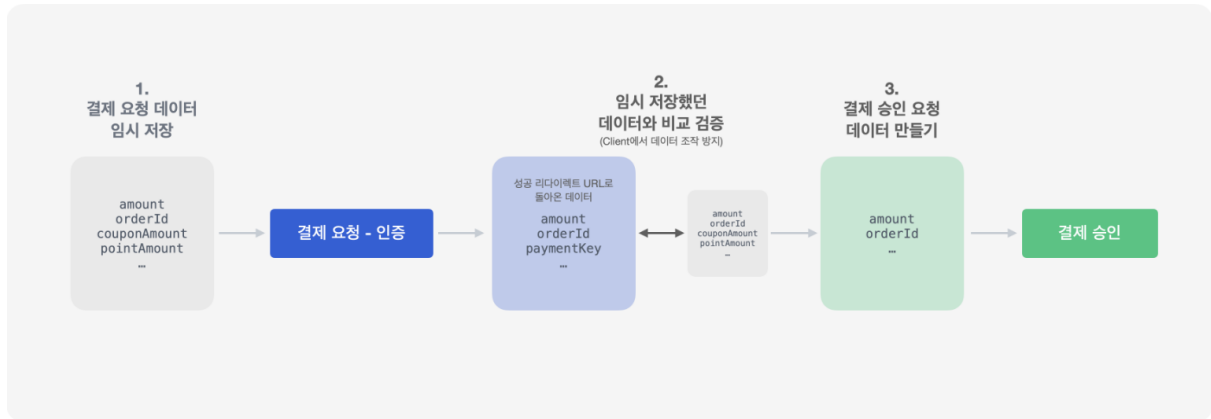
    TossPaymentResponse.TosspayDTO result =
        webClient
            .post()
            .uri("/v1/payments/confirm")
            .headers(headers -> {
                headers.add(HttpHeaders.AUTHORIZATION, basicToken);
                headers.add(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE);
            })
            .bodyValue(requestDTO)
            .retrieve()
            .bodyToMono(TossPaymentResponse.TosspayDTO.class)
            .onErrorResume(e -> {
                throw new ServerException(BaseException.PAYMENT_FAIL);
            })
            .block();


    // 받은 payment와 관련된 데이터(orderId, amount)가 정확한지 확인
    private Boolean isCorrectData(Payment payment, String orderId, Long amount){
        return payment.getOrderId().equals(orderId)
            && Objects.equals(payment.getPaidAmount(), amount);
    }

    private Payment findPaymentByUserId(Long userId){
        return paymentJpaRepository.findById(userId).orElseThrow(
            () -> new NotFoundException(BaseException.PAYMENT_NOT_FOUND)
        );
    }

    private User findUserById(Long userId){
        User user = userJpaRepository.findById(userId).orElseThrow(
            () -> new NotFoundException(BaseException.USER_NOT_FOUND)
        );
        // 이미 프리미엄 등급인 경우 결제하면 안되므로 예외 던짐
        if (user.getGrade() == Grade.PREMIUM){
            throw new BadRequestException(BaseException.USER_ALREADY_PREMIUM);
        }
        return user;
    }
}

```



 순수 멤버십 결제 후 유저 등급 업그레이드를 하기 위해 토스페이먼츠와 연동을 진행했습니다. (결제 + 업그레이드)를 atomic 하게 처리하여 **데이터 무결성**을 보장하였습니다. 전체 로직은 다음과 같습니다.

- 결제 요청을 위한 데이터 저장
 - orderId와 amount를 Payment 테이블에 저장합니다.
- 결제 승인 요청 및 업그레이드
 1. 검증: 이전에 저장한 데이터와 현재 request로 받은 데이터 비교
 2. 토스페이먼츠 승인요청: paymentKey, orderId, amount 데이터를 담아 토스페이먼츠로 승인 요청을 보냅니다.
 3. 승인 요청에 성공하면 유저 등급을 NORMAL → PREMIUM으로 업그레이드하고 결제 시간을 저장합니다.

CORS 설정

▼ 코드: `_core > config > SecurityConfig`

```
public CorsConfigurationSource configurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.addAllowedHeader("*");

    configuration.addAllowedMethod(HttpMethod.GET);
    configuration.addAllowedMethod(HttpMethod.POST);
    configuration.addAllowedMethod(HttpMethod.PUT);
    configuration.addAllowedMethod(HttpMethod.DELETE);

    configuration.addAllowedOriginPattern("http://localhost:3000");
    configuration.addAllowedOriginPattern("https://k6f3d3b1a0696a.user-app.krampoline.com");
    configuration.addAllowedOriginPattern("https://k5c1813d97f50a.user-app.krampoline.com"); // 프론트 테스트용

    configuration.setAllowCredentials(true);

    configuration.addExposedHeader("Authorization");
    configuration.addExposedHeader("Refresh");

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/*", configuration);
    return source;
}
```



저희 프로젝트의 CORS 설정은 다음과 같습니다.

- GET, POST, PUT, DELETE 메서드를 사용하고 있습니다. 따라서 위 4가지 HTTP methods를 허용하였습니다.
- 배포 환경에서 프론트 엔드와의 통신은 localhost의 3000번 포트를 통해 이루어지기 때문에 localhost:3000 포트만의 통신만 허용하였습니다.
- JWT 토큰을 사용하고 있습니다. 액세스 토큰을 위한 Authorization 헤더, 리프레시 토큰을 위한 Refresh 헤더를 사용하고 있습니다. 따라서 두 헤더를 클라이언트에서 열람 가능하도록 수정했습니다.
- Authorization 헤더를 통해 유저 인증을 진행하고 있기 때문에 setAllowCredentials 옵션을 true로 활성화했습니다.

그리고 백엔드 서버로 들어오는 모든 요청에 대해 위 설정을 적용하였습니다.

SQLDelete

▼ 코드: 엔티티마다 설정

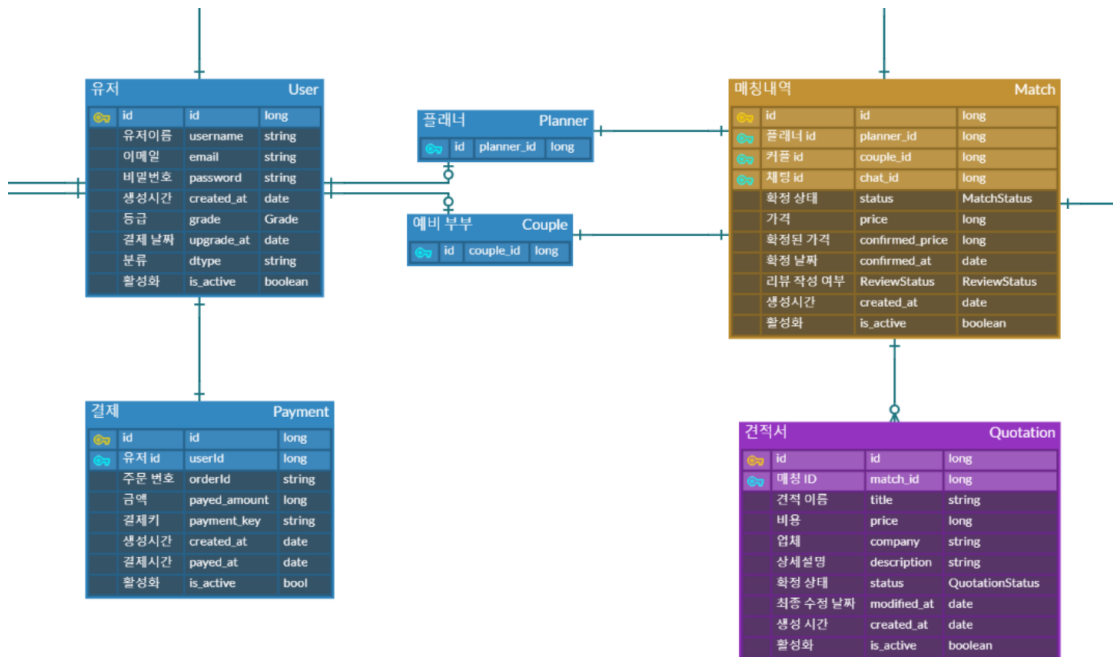
```
@SQLDelete(sql = "UPDATE user_tb SET is_active = false WHERE id = ?")
@Where(clause = "is_active = true")
```



저희 테이블 구조는 매칭내역에 플래너와 예비 부부가 함께 들어가 있는 형태입니다. 이 경우 예비 부부가 탈퇴하면 웨딩 플래너가 견적서 등 연관된 다른 데이터를 조회할 때 문제가 발생합니다. 저희는 포트폴리오 조회에서도 플래너의 이전 거래 내역을 조회하기 위해 견적서 내용을 가져와야 하기 때문에 Hard Delete를 하게되면 조회가 불가능합니다.

실무에선 is_active와 같은 상태값을 확인할 수 있는 필드를 만들어 SQL Delete를 사용한다는 멘토님의 조언을 듣고 유저, 매칭, 견적서 등에서 활성화 유무 등을 관리하도록 하였습니다.

또한 포트폴리오를 플래너가 실제로 삭제하거나 견적서 등 중요 데이터를 삭제할 경우 내용을 다시 복구할 수 있도록 하였습니다.



매칭 관련 ERD (플래너 id, 커플 id를 모두 참조함)

환경 변수 세팅



저희 github에 올라간 코드를 더 안전하게 관리하기 위해서 다음 내용을 환경변수로 분리해서 관리하고 있습니다.

- `EMAIL_TEST_CODE` (이메일 인증 코드)
- `SENDER` (이메일 전송 계정 이메일)
- `GMAIL_PASSWORD` (이메일 전송 계정 비밀번호)
- `TOSS_PAYMENT_SECRET` (토스 페이먼트 시크릿키)
- `JWT_ACCESS_SECRET` (access토큰 시크릿 키)
- `JWT_REFRESH_SECRET` (refresh토큰 시크릿키)

패스워드 인코딩

▼ 코드: `user > UserServiceImpl.java`

```
String encodedPassword = passwordEncoder.encode(requestDTO.password());
```



사용자의 개인정보를 보호하기 위해 PasswordEncoder를 사용하여 패스워드 정보를 인코딩해서 저장합니다.

4 성능

N번의 Insert Query → JDBC Batch Update

▼ 코드: `portfolio > image > PortfolioImageItemJDBCRepositoryImpl.java`
`> price > PriceItemJDBCRepositoryImpl.java`
`review > image > ReviewImageItemJDBCRepositoryImpl.java`

```
public void batchInsertPriceItems(List<PriceItem> priceItems) {
    String sql = String.format("""
        INSERT INTO %s (portfolio_id, item_title, item_price)
        VALUES (?, ?, ?)
        """, TABLE);

    jdbcTemplate.batchUpdate(sql, priceItems, priceItems.size(),
        (ps, priceItem) -> {
            ps.setLong(1, priceItem.getPortfolio().getId());
            ps.setString(2, priceItem.getItemTitle());
            ps.setLong(3, priceItem.getItemPrice());
        });
}
```

JDBC batchUpdate 도입

웨딩 플래너가 작성한 포트폴리오와 포트폴리오에 포함될 이미지, 가격 항목 & 예비 부부가 작성한 리뷰와 리뷰에 포함될 이미지 항목에는 서로 1대 N의 관계가 성립합니다.

그렇기 때문에 포트폴리오 등록 상황이 발생하면 서버에서는 DB의 '포트폴리오' 테이블과 '포트폴리오 이미지' 테이블, '포트폴리오 가격' 테이블에 접근해야 합니다.

예를 들어, 사용자가 포트폴리오를 등록할 때 이미지 4개, 가격 항목 3개를 추가하면 '포트폴리오' 테이블에 1개의 레코드, '포트폴리오 이미지' 테이블에 4개의 레코드, '포트폴리오 가격' 테이블에 3개의 레코드를 추가해야 합니다.

(1) JPA

하지만 이 동작을 JPA로 수행하면 총 8개의 INSERT Query가 발생합니다.

```
INSERT INTO portfolio_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_imageitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_imageitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_imageitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_imageitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_imageitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_priceitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_priceitem_tb VALUES (?, ?, ?, default);
INSERT INTO portfolio_priceitem_tb VALUES (?, ?, ?, default);
```

▼ 상세 설명

왜냐하면 JPA는 기본적으로 Batch Insert가 비활성화되어 있기 때문입니다.

```
spring.jpa.properties.hibernate.order_inserts=true
spring.jpa.properties.hibernate.order_updates=true
spring.jpa.properties.hibernate.jdbc.batch_size=50
```

하지만 위와 같이 Spring 설정 값들을 변경해주어도 여전히 동작하지 않습니다.

그 이유는 다음 Hibernate의 공식 문서에도 나와있는 것처럼



Hibernate disables insert batching at the JDBC level transparently if you use an identity identifier generator.

JPA Entity에서 Primary Key의 GenerationType을 IDENTITY로 설정할 경우에 Hibernate가 JDBC batch를 비활성화시켜 버리기 때문입니다.

그 이유는 새로 할당할 Key 값을 미리 알 수 없는 IDENTITY 방식을 사용할 때 Batch Support를 지원하면 Hibernate가 채택한 flush 방식인 'Transactional Write Behind'와 충돌이 발생하기 때문이라고 합니다.

따라서 가장 널리 사용되는 IDENTITY 방식을 사용하면 Batch Insert는 동작하지 않습니다.

(2) JDBC Template

그렇다고 다양한 이점이 있는 IDENTITY 방식을 변경할 수는 없기 때문에 JdbcTemplate에서 Batch를 지원하는 `batchUpdate` 메서드를 사용하기로 했습니다.

```
@Repository
public class PortfolioImageItemJDBCRepositoryImpl implements PortfolioImageItemJDBCRepository {
```

```
private final JdbcTemplate jdbcTemplate;

...
    jdbcTemplate.batchUpdate(sql, portfolioImageItems, portfolioImageItems.size(), pss);
...

```

위와 같이 `batchUpdate` 를 사용하면 Multi Value 형태로 작성된 SQL Batch Insert문을 실행시킬 수 있습니다.

```
o.s.jdbc.core.JdbcTemplate : Executing SQL batch update [INSERT INTO portfolio_imageitem_tb ... ]
o.s.jdbc.core.JdbcTemplate : Executing prepared SQL statement [INSERT INTO portfolio_imageitem_tb ... ]
...
o.s.jdbc.core.JdbcTemplate : Executing SQL batch update [INSERT INTO portfolio_priceitem_tb ... ]
o.s.jdbc.core.JdbcTemplate : Executing prepared SQL statement [INSERT INTO portfolio_priceitem_tb ... ]

```

결과적으로 Batch Insert문을 사용하게 되면 테이블마다 하나씩 총 3개의 쿼리가 실행됩니다.

결론

구현 후 테스트를 해보았을 때,

Batch Insert를 사용하지 않고 JPA로 **1만 건의 데이터**를 삽입했을 때 약 30초(31263ms)의 시간이 소요되었지만,

Batch Insert를 사용하니 935ms로 수행 시간이 불과 1초도 되지 않는 퍼포먼스를 보여주었습니다. **약 30배 이상의 엄청난 성능 차이**를 보이는 것을 확인할 수 있습니다.

JDBC batchUpdate는 수정(Update) 요청 시에도 마찬가지로 적용하였으며,

이를 통해, 서비스 과정에서 한꺼번에 요청 받는 데이터의 항목 수가 늘어나도 안정적인 Response 시간을 확보하며 효과적으로 잘 대처할 수 있을 것으로 기대합니다.

인덱스

▼ 코드: 엔티티마다 설정

```
@Table(
    name = "portfolio_tb",
    indexes = {
        @Index(name = "planner_index", columnList = "planner_id")
    })

```



저희 프로젝트의 궁극적인 목적은 정보 공유에 있습니다. 다른 사람들의 데이터를 이용해서 합리적인 의사결정을 내리는 것에 초점이 있기 때문에, 삽입 및 수정에 비해 조회 요청의 비율이 압도적으로 높습니다. 따라서 조회 성능을 높이기 위해 테이블에 인덱스를 추가했습니다.

- `user_tb` : 유저 테이블의 경우, 가장 빈번하게 발생하는 조회 쿼리는 유저 인증 시에 발생하는 email을 통한 엔티티 조회입니다. 따라서 email 인덱스를 추가했습니다.
- `token_tb` : 토큰은 유저 아이디에 해당하는 토큰을 조회하는 경우가 전부입니다. 따라서 인덱스로 `user_id`를 추가했습니다.
- `email_code_tb` : 이메일 인증코드의 경우, 이메일로 조회하는 경우 뿐입니다. 따라서 email을 인덱스로 추가했습니다.
- `quotation_tb` : `match_id` 혹은 유저 아이디로 조회하는 경우가 대부분입니다. 따라서 `match_id`를 인덱스로 추가했습니다. 유저 아이디로 조회는 Match 테이블이므로 추가하지 않았습니다.
- `portfolio_tb` : 대부분의 경우가 플래너를 이용한 조회입니다. 따라서 `planner_id`를 인덱스로 추가했습니다.
- `portfolio_price_item_tb` , `portfolio_image_item_tb` : 포트폴리오를 통한 조회가 전부이기 때문에, `portfolio_id`를 인덱스로 추가했습니다.
- `payment_tb` : 유저 아이디를 이용한 조회가 전부이기 때문에 `user_id`를 인덱스로 추가했습니다.
- `match_tb` : 플래너, 커플, 플래너와 커플 3가지 경우로 조회를 시도합니다. 하지만 `couple`을 이용한 조회가 대다수 이고, 매칭내역 생성을 위한 플래너와 커플 모두를 사용한 조회가 다음입니다. 플래너만을 이용한 조회는 빈번하게 발생하지 않습니다. 따라서 `planner_id`와 `couple_id`를 복합키로 인덱스 설정을 하고, `couple_id`를 앞에 배치하였습니다.
- `favorite_tb` : 유저 아이디를 이용한 조회가 전부입니다. 포트폴리오도 필요한 경우가 있지만, 유저 아이디만 이용하는 경우가 압도적이라고 판단되어 `user_id`만 인덱스로 추가했습니다.
- `chat_tb` : 채팅 테이블은 다른 테이블에서 join하여 사용하는 경우가 대부분입니다. 그렇기에 id만으로 충분하기 때문에, 추가적인 인덱스를 설정하지 않았습니다.