

개인 프로젝트

순수웨딩 2.0

2025.03 ~ 2025.05

1. 프로젝트 개요

가격 정보가 불투명하고 발품이 필수였던 기존 웨딩 시장의 문제를 해결하고자, 투명한 가격과 1:1 실시간 상담을 온라인으로 제공하는 소비자 중심의 웨딩 플랫폼입니다. 기존 팀 프로젝트(카카오테크캠퍼스)를 **1인 풀스택 프로젝트**로 리빌딩하여, **확장성과 안정성을 갖춘 실제 운영 수준의 시스템 구축**을 목표로 삼았습니다.

2. 주요 역할 및 성과

Frontend(React) 개선부터 Backend 전면 재설계, Infrastructure 구축까지 전 과정을 담당했습니다.

- **분산 채팅 시스템 구축**: 안정적인 메시지 전달을 보장하고, 수평 확장을 통해 대규모 트래픽에 대응할 수 있는 **Kafka 기반의 Event-Driven 아키텍처**를 설계했습니다.
- **결제 데이터 정합성 확보**: 외부 PG사(Toss Payments)와 내부 시스템 간 장애 발생 시에도 데이터 불일치가 발생하지 않도록 **다중 복구 메커니즘**을 적용한 Fault-Tolerant 시스템을 구축했습니다.
- **API 성능 최적화**: DTO 프로젝션, 커서 기반 페이징, 캐시 및 리소스 튜닝을 통해 핵심 API(포트폴리오 목록 조회)의 처리량을 **2.4배 향상(626→1,500 TPS)**시키고 응답 시간을 **47% 단축**했습니다.

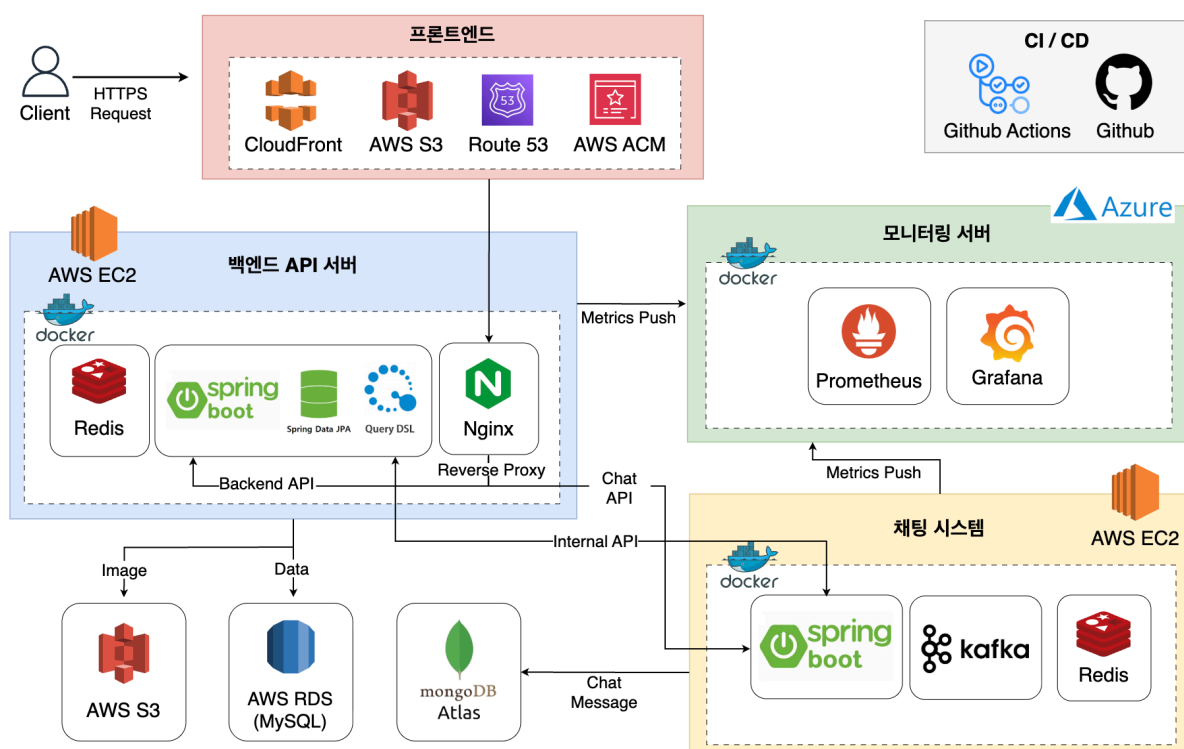
3. 사용 기술

Java, Spring Boot, JPA, JDBC, QueryDSL, Kafka, Redis, MongoDB, MySQL, React, JavaScript, Docker, Nginx, AWS (EC2, RDS, S3, ACM, CloudFront, Route 53), GitHub Actions, Prometheus, Grafana, k6

4. 관련 링크

- GitHub: [Backend Repo](#) | [Chat Server Repo](#) | [Frontend Repo](#)
- 문서: [프로젝트 상세 문서\(Notion\)](#).

5. 시스템 구성도



1. 대규모 트래픽을 위한 실시간 채팅 시스템 아키텍처 설계

배경 및 목표

이전 Firebase Realtime Database 기반 채팅 시스템이 가진 **프론트엔드 중심 아키텍처의 확장성 한계**를 극복하고자 했습니다. 이를 위해 두 가지 기술적 목표를 설정했습니다. 첫째, **안정적인 메시지 전달을 보장**하는 것, 둘째, 그 안정성을 기반으로 **수평 확장을 통해 대규모 트래픽에 대응**할 수 있는 시스템을 구축하는 것이었습니다. 나아가 이미지 전송, 접속 및 읽음 상태 표시 등 **Discord 수준의 고품질 사용자 경험**을 제공하는 것을 목표로 삼았습니다.

1-1. 실시간 채팅 아키텍처 설계: 안정성과 확장성을 향한 진화 과정

초기 아키텍처: Redis Pub/Sub의 비효율

초기 아키텍처는 수평 확장을 전제로 Redis Pub/Sub의 브로드캐스트 방식을 도입했습니다. 서버 증설은 편리했지만, 서버 수가 늘어날수록 모든 서버에 불필요한 메시지를 전송하여 **심각한 네트워크 자원 낭비**를 유발하는 명확한 한계가 있었습니다.

1차 개선: 중앙 집중형 아키텍처로의 진화

초기 아키텍처의 네트워크 비효율 문제를 해결하고자 Redis에 사용자 위치 정보를 기록해 **필요한 서버에만 메시지를 보내는 유니캐스트 방식**으로 1차 개선했습니다. 당장의 네트워크 문제는 해소했지만, 이내 **각 서버가 다른 모든 서버와 직접 통신하는 Mesh 구조**가 서버 간 인증 설정의 **복잡성 문제**와, 두 역할(WebSocket 연결과 전송)의 자원 경합이 서비스 마비로 이어지는 **연쇄 장애 문제**를 야기한다는 것을 깨달았습니다. 결국 운영 가능한 안정성과 확장성을 확보하기 위해, 모든 메시지 라우팅을 중앙에서 전담하는 **'메시지 전송 서버'를 분리한 중앙 집중형(Star) 아키텍처**로 최종 개선했습니다.

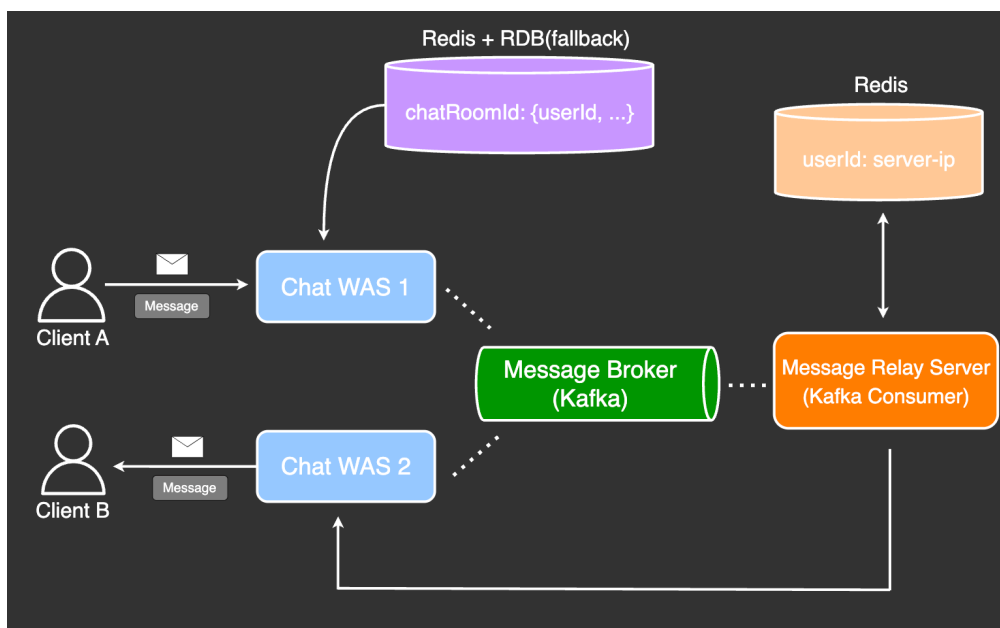
새로운 문제와 해결 방안: 메시지 브로커 도입

중앙 집중형 구조로 운영 복잡성은 해결했지만, **서버 간 직접 통신으로 인한 데이터 유실**이라는 문제를 낳았습니다. 네트워크 장애나 서버 다운 시 데이터가 영구적으로 유실될 위험이 있었고, 특히 직접 통신 구조에서는 한 서버의 장애가 응답을 대기하는 다른 서버의 부하를 연쇄적으로 높여 시스템 전체를 불안정하게 만드는 **장애 전파 문제**도 있었습니다. 궁극적인 신뢰성과 유연성 확보를 위해, 두 시스템을 Decoupling 해 줄 **메시지 브로커 도입**을 결정했습니다.

기술 선택: 왜 Kafka였는가?

메시지 브로커 선정의 최우선 기준은 **'At-least-once' 전달 보장**이었고, 이에 따라 RabbitMQ와 Kafka를 고려했습니다. 가장 큰 고민은 디스크 기반의 Kafka가 채팅 시스템에 필요한 빠른 응답 속도를 보장할 수 있는지였습니다. 검토 결과, Kafka는 순차 I/O, Zero-Copy 등의 최적화를 통해 디스크 기반임에도 높은 처리량을 확보하도록 설계되었음을 확인하며 성능 우려를 해소했습니다. 성능 이슈가 해결되자, 단순 메시지 큐 역할의 RabbitMQ와 달리, 모든 데이터를 영속적인 이벤트 로그로 보관해 향후 데이터 분석, 모니터링 등 기능 확장에 높은 유연성을 제공하는 **'이벤트 스트림 플랫폼'으로서의 장기적 확장성** 때문에 Kafka를 최종 선택했습니다.

그 결과, **데이터 유실에 대한 높은 내구성**과 새로운 기능 확장의 유연성을 모두 갖추고, 트래픽 증가에 대응하기 위한 **수평 확장이 용이한 아키텍처**를 완성했습니다.



Kafka 기반 채팅 시스템 아키텍처

1-2. 메시지 전달 신뢰성 보장: At-least-once와 Exactly-once

안정적인 데이터 파이프라인을 위해, 메시지의 비즈니스 중요도에 따라 전달 보장 수준을 다르게 적용했습니다.

기본 전략: At-least-once

데이터 파이프라인의 기본 신뢰도는 At-least-once를 기반으로 설계했습니다. 프로듀서는 멱등성 설정을 통해 메시지가 브로커에 중복 저장되는 것을 방지했습니다. 또한, 모든 컨슈머는 오프셋 커밋 실패로 인한 재전달에 대비해 **멱등성을 보장**하도록 설계했으며, 특히 외부 의존성이 있는 MongoDB의 경우 **upsert** 연산을 활용해 이를 구현했습니다.

핵심 전략: Exactly-once

기본 신뢰도는 그대로 두되, 모든 후속 기능의 기준점이 되는 '**저장 완료**' 이벤트는 **Exactly-once** 전달을 보장했습니다. 이는 파이프라인의 시작점을 '**Source of Truth**'로 만들어, 다운스트림 시스템의 디버깅을 용이하게 하고 예측 가능성을 높이기 위함이었습니다. 또한, 파이프라인 초입에서 논리적 중복을 제거함으로써 연쇄적인 데이터 오염을 방지하고 시스템 전체의 부하를 줄이는 실질적인 이점도 있었습니다.

이를 위해 **Kafka의 Transactional Producer**를 활용하여, '메시지 소비(오프셋 커밋)'와 '후속 메시지 발행'을 **하나의 원자적 단위로 처리**했습니다. 다운스트림 컨슈머 또한 **isolation.level=read_committed** 설정을 통해 트랜잭션이 완료된 메시지만 읽어가도록 구성하여 데이터 정합성을 확보했습니다.

1-3. Polyglot Persistence: 대용량 트래픽을 위한 저장소 설계

문제 정의: RDB의 쓰기 처리량 한계와 'Write Amplification'

채팅 서비스는 메시지 하나가 생성될 때마다 마지막 메시지, 채팅방 순서, 읽음 상태 등 다수의 쓰기 작업이 연쇄적으로 발생하는 '**Write Amplification**' 특성을 가집니다. 단일 RDB 환경에서는 이러한 부하가 심각한 성능 병목을 유발할 것이라 판단하여, 데이터 특성에 맞춰 저장소를 분리하는 **Polyglot Persistence 아키텍처**를 설계했습니다.

해결 전략: 역할 중심의 저장소 분리

- **MongoDB (메시지 본문 저장)**

채팅 메시지의 빈번한 대용량 쓰기 특성에 대응하기 위해 NoSQL인 MongoDB를 채택했습니다. **유연한 스키마** 덕분에, 텍스트, 이미지(URL) 등 구조가 각기 다른 메시지 데이터를 하나의 컬렉션에서 효율적으로 관리할 수 있었습니다. 또한 '**채팅방ID-시퀀스ID**' **복합 인덱스**를 활용하여 높은 조회 성능을 확보했습니다.

- **Redis (실시간 메타데이터)**

읽음 상태, 정렬 순위처럼 즉각적인 피드백이 필수적인 데이터에 대해서는 Redis가 '**Primary Store**' 역할을 담당했습니다. 단순 캐시를 넘어, 모든 실시간 쓰기 작업이 Redis에 먼저 일어나도록 하여 빠른 상호작용을 보장했습니다.

- **MySQL (신뢰도 높은 기록 시스템)**

채팅방 정보, 참여자 관계처럼 데이터 정합성이 최우선인 핵심 데이터는 MySQL에 저장하여 서비스의 영구적인 **System of Record** 역할을 하도록 했습니다.

최종 일관성 확보 전략

모든 실시간 데이터 변경은 Redis에 먼저 반영하여 사용자 경험을 최적화하고, 스케줄러 기반의 비동기 파이프라인을 통해 MySQL과 최종 일관성을 보장했습니다.

1. **변경 데이터 기록:** Redis 데이터 변경 시, 즉시 DB에 쓰지 않고 변경된 데이터의 Key를 별도의 Set에 기록하여 변경분을 추적합니다.
2. **주기적 일괄 처리(Batch Update):** 스케줄러가 주기적으로 이 Set의 Key들을 수집하여 변경분을 모아 MySQL에 JDBC Batch Update를 수행함으로써 DB 부하를 최소화합니다.
3. **안정적인 복구 및 재시도:** 동기화 실패 시 Set에서 Key를 제거하지 않아 다음 주기에 자동으로 재시도되며, Redis 장애 시에는 MySQL의 데이터를 통해 복구합니다.

1-4. 대규모 동시 접속 대응을 위한 채팅 시스템 성능 최적화

Kafka 기반 분산 채팅 시스템이 안정적인 메시지 전달을 유지하며 수용할 수 있는 **최대 동시 커넥션 수를 측정**하고, 병목 지점을 찾아 최적화하는 테스트를 진행했습니다. 이 과정의 목표는 단순히 높은 TPS를 달성하는 것이 아니라, 설정된 **SLO(p95 응답 시간 1초 이내, 유실 0%)**를 만족하는 **시스템의 최대 부하 지점(임계점)**을 찾는 것이었습니다.

테스트 전략 및 환경

- **아키텍처**: 채팅 서버(Spring Boot), 전송 서버(Kafka Consumer), Kafka, Redis, MongoDB
- **인프라 제약**: 모든 컴포넌트는 AWS EC2 t2.micro(1 vCPU, 1GB RAM) 환경에 배포하여 최소 사양에서의 성능 한계를 명확히 파악하고자 했습니다.
- **부하 시나리오**: k6를 사용하여 로컬(M1 Mac, 16GB RAM)에서 WebSocket 연결을 생성하고, 각 사용자는 5초마다 메시지를 보내도록 설정했습니다. (전체 TPS $\approx$ 동시 커넥션 수 / 5)

테스트 방법론 및 모니터링 환경

이번 테스트는 Kafka로 서버를 분리한 아키텍처가 높은 부하에서도 메시지 유실 없이 안정성을 유지하면서, 수평 확장을 통해 선형적으로 성능이 증가하는지를 정량적으로 증명하는 것이 목표였습니다. 이를 위해 Prometheus와 Grafana 기반의 통합 모니터링 환경을 구축하여, **Kafka Consumer Lag**로 파이프라인 병목을, **CPU/Heap 사용률**로 수평 확장의 필요 시점을, 그리고 **응답 시간과 메시지 유실률**로 전체 시스템의 안정성을 실시간으로 추적했습니다.

단계별 성능 테스트 결과 및 분석

단계	구성 (채팅/전송)	동시 커넥션 (VU)	처리량 (TPS)	핵심 변화 및 분석
1. 베이스라인	1 / 1	580	~110	MongoDB Atlas 쓰기 성능 병목
2. 단일 서버 튜닝	1 / 1	1,800	~360	OS·Nginx의 커넥션 한계 확장, JVM Heap 확장 및 Kafka Producer/Consumer 기본 튜닝 적용
3. 파티션 확장	1 / 1	2,750	~540	Kafka 파티션 및 Consumer 스레드 수를 늘려 병렬 처리량 극대화
4. 수평 확장	3 / 3	5,200	~1,050	Kafka Broker 자체가 병목 지점으로 작용

성능과 안정성의 트레이드오프: Exactly-once 제거

초기 테스트에서 MongoDB Atlas M0 티어의 쓰기 제한(약 100 TPS)이 첫 병목으로 확인되었습니다. 이후 테스트의 정확성을 위해, '메시지가 Kafka 토픽에 성공적으로 적재되면, DB에도 반드시 저장된다'는 가정하에 DB 저장 로직을 임시로 제거하고 **Kafka 오프셋**을 기준으로 유실 여부를 측정했습니다.

하지만 DB 병목이 사라지자, 오히려 Consumer Lag이 급증하며 시스템 전체 성능이 급격히 저하되는 예상 밖의 현상이 발생했습니다. 원인은 **Kafka의 Exactly-once 트랜잭션 오버헤드**였습니다. 이는 데이터 레코드 외에 트랜잭션의 성공 여부를 기록하는 **'마커 레코드'를 추가로 발행**하는 방식 때문으로, 실제 모니터링에서도 메시지 1건당 마커 1건이 추가 발행되어, 토픽의 레코드 수가 2배로 증가하는 것을 확인했습니다. 이 실증적 결과를 바탕으로 해당 기능이 실시간 채팅의 성능 요구사항과 상충됨을 확인하고, **At-least-once 기반으로 전환**하여 파이프라인 성능을 확보했습니다.

최종 결론 및 성과

- **최대 성능 지표 달성**: SLO(p95 응답 시간 1초 이내, 유실 0%)를 만족하는 **최대 동시 커넥션 5,200개, 메시지 처리량 1,050 TPS**를 달성했으며, 최종 병목은 Kafka Broker의 t2.micro 스펙 자체임을 확인했습니다.
- **확장 가능한 아키텍처 검증**: 최종 부하 테스트에서 애플리케이션 서버의 CPU/Heap 사용률은 50% 내외로 안정적이었으며, SLO 한계점을 넘긴 구간에서도 커넥션은 계속 확보됐습니다. 이는 병목 지점이 애플리케이션 서버가 아닌 **Kafka Broker의 한계**임을 뒷받침하며, 향후 **Broker만 스케일 업**하면 시스템 전체가 더 많은 트래픽을 처리할 수 있음을 의미합니다.

2. 결제 시스템: 신뢰성 있는 PG사 연동 구축

배경 및 목표

'순수웨딩'의 핵심 유료 기능인 '순수 멤버십'은 사용자가 결제를 통해 웨딩 플래너의 실제 계약 건적서를 열람할 수 있는 기능입니다. 이를 구현하고자 PG사인 Toss Payments를 연동하여 결제 시스템을 구축했으며, 돈을 직접 다루는 만큼 단순 기능 구현을 넘어 어떤 예외 상황에서도 **데이터 정합성과 보안이 보장되는 견고한 시스템을 만드는 것**을 최우선 목표로 삼았습니다.

이를 위해 시스템 신뢰도를 구성하는 세 가지 핵심 요소를 **1) 보안 설계, 2) 장애 허용, 3) 검증된 안정성**으로 정의하고, 이 순서에 따라 보안 취약점 해결부터 데이터 정합성 보장, 최종적으로는 단위 테스트를 통해 이를 증명하는 과정까지 단계적으로 개발을 진행했습니다.

2-1. 결제 데이터 위변조 차단: 책임 분리를 통한 보안 강화

문제: 클라이언트 주도 방식의 근본적인 보안 취약점

초기 결제 시스템은 클라이언트가 결제 데이터 생성부터 승인 요청까지 모두 처리하여, **금액 위변조에 취약**했습니다. 팀 프로젝트 당시, 승인 책임만 서버로 옮겨 검증하는 1차 개선을 진행했지만, 이는 근본적인 해결책이 아니었습니다.

핵심 문제는 '**결제 데이터의 생성 주체**'가 여전히 클라이언트에 있었다는 점입니다. Toss Payments는 결제창이 호출된 '이후'의 무결성만 보장하며, 결제창 호출 이전 단계의 데이터 보안은 전적으로 **가맹점 서버의 책임 영역**이기 때문입니다. 결국 클라이언트에서 생성된 조작된 금액으로 결제가 시작될 위험은 그대로였습니다.

해결: 서버 주도의 End-to-End 보안 플로우 설계

결제 데이터의 생성 주체를 클라이언트에서 서버로 이전하여, 금액 위변조가 원천적으로 불가능하도록 설계했습니다. 서버는 직접 원본 데이터를 생성 및 저장한 뒤, 최종 승인 시점에 DB의 원본 데이터와 클라이언트 요청 값을 **교차 검증**합니다. 이 서버 측 검증을 통해, PG사의 검증에 더해 최종적인 데이터 무결성을 서버가 직접 보장하도록 만들었습니다.

최종 End-to-End 보안 플로우

1. 클라이언트 → 서버: 상품 정보 전달 및 결제 준비 요청
2. 서버: 결제 데이터(`amount` , `orderId`) 생성 및 DB 저장
3. 서버 → 클라이언트: 생성된 결제 데이터 전달
4. 클라이언트 → 토스: 서버 제공 데이터로 결제창 호출
5. 클라이언트 → 서버: 사용자 인증 완료 후 결제 승인 요청
6. 서버: DB 저장 원본과 토스 데이터 교차 검증
7. 서버 → 토스: 검증 완료 시 최종 결제 승인

결과: 서버가 원본을 소유함으로써 클라이언트의 **악의적인 데이터 조작**을 원천 차단하고 프로세스 전반의 데이터 무결성을 확보했습니다.

2-2. 테스트 가능한 설계: 외부 API 추상화를 통한 품질 향상

Toss 결제 API와 같이 외부 시스템에 직접 의존하여 **강하게 결합된 코드**는 단위 테스트 작성을 어렵게 하고 유지보수를 저해했습니다.

이를 해결하기 위해 `PaymentApprovalClient` 인터페이스를 도입하여 **외부 API 호출 로직을 추상화**했습니다. 의존성 역전 원칙(DIP)을 적용하여, 테스트 코드에서는 Mockito를 활용한 Mocking으로 외부 API를 쉽게 대체할 수 있게 되었습니다.

그 결과, 외부 환경의 변화와 무관하게 비즈니스 로직의 단위 테스트가 가능해져 코드의 신뢰성을 확보했으며, 새로운 PG사 추가 시에도 기존 코드를 수정할 필요 없는 **개방-폐쇄 원칙(OCP)**을 준수하는 **유연한 구조**를 완성할 수 있었습니다.

2-3. 결제 데이터 정합성: 분산 환경에서의 일관성 보장 전략

문제 정의: 깨진 원자성과 데이터 불일치

결제 승인 로직에서 외부 PG사 API 호출과 내부 DB 업데이트를 단일 트랜잭션(**@Transactional**)으로 처리하는 것은 '깨진 원자성' 문제를 야기했습니다. 외부 API 응답이 지연되면 **DB 커넥션을 불필요하게 점유**해 성능을 저해했고, 더 치명적으로는 PG사 결제는 성공했으나 네트워크 장애로 응답이 유실되거나 내부 DB 커밋이 실패할 경우, 사용자는 돈을 지불했지만 서비스를 받지 못하는 **데이터 불일치 상황**이 발생할 수 있었습니다.

초기 대안으로 **Saga 패턴(보상 트랜잭션)**을 검토했습니다. 하지만 이는 사용자가 은행 앱에서 '출금' 알림을 받은 뒤, 잠시 후 웹사이트에서는 '결제 실패'를 보고 '입금(취소)' 알림을 받는 상황을 만듭니다. **상충하는 정보**를 여러 채널에서 마주하게 되어 **서비스 신뢰도를 근본적으로 훼손**할 수 있었습니다. 여기에 보상 트랜잭션 자체의 실패 처리까지 고려해야 해 구현 복잡성이 높다고 판단했습니다.

따라서, **외부 PG사의 결제 성공 상태를 'Source of Truth'로 삼고**, 우리 시스템의 데이터를 그에 맞춰 동기화하는 **'최종 일관성'**을 목표로 설정했습니다.

해결 과정: 다층 복구 아키텍처로의 진화

최종 일관성을 확보하고 사용자 경험까지 고려하기 위해, 복구 메커니즘을 단계적으로 발전시킨 다층 아키텍처를 구축했습니다.

1단계: 비동기 복구 채널 구축 (데이터 정합성 확보)

우선 데이터 유실을 방지하기 위해 **Toss Webhook** 수신과 **실패 로그 기반의 스케줄러**라는 두 가지 비동기 복구 채널을 구축했습니다. 이를 통해 어떤 장애 상황에서도 데이터를 반드시 복구하는 견고한 1차 안전망을 확보했습니다.

2단계: 동기 재시도 추가 (사용자 경험 개선)

하지만 위 방식은 장애 발생 시 **최대 3분(스케줄러 주기)**의 대기 시간을 유발하는 UX 문제를 안고 있었습니다. 근본 원인 분석 결과, 문제의 대부분이 '일시적 네트워크 단절'처럼 **재시도만으로 해결 가능한 '회복 가능한 장애'**라는 인사이트를 얻었습니다. 이에 **백오프 전략을 적용한 동기적 재시도(@Retryable)** 로직을 추가하여, 대부분의 일시적 장애를 사용자가 인지하지 못하는 사이 즉시 처리하도록 개선했습니다. 다만 타임아웃은 승인의 성공 여부조차 알 수 없어 재시도만으로는 대응할 수 없으므로, 조회 API로 최종 상태를 확인하는 **'실시간 재조회'**를 추가했습니다.

성과: 실용성과 안정성을 모두 갖춘 결제 시스템

동기 재시도와 비동기 복구 채널을 결합한 **다층 복구 아키텍처**를 통해, 대부분의 일시적 장애는 동기 계층에서 즉시 처리하여 **사용자 경험을 보장**하고, 비동기 복구 채널은 **최종 안전망**으로 기능하는 실용적인 시스템을 완성했습니다.

- **효율적인 리소스 관리:** 외부 API 호출을 트랜잭션과 분리하여 불필요한 DB 커넥션 점유 문제를 해결했습니다.
- **견고한 데이터 정합성:** [동기 재시도] → [실시간 재조회/Webhook] → [배치 스케줄러] 순으로 이어지는 다층 복구 채널을 통해 데이터 유실 위험을 최소화하는 장애 허용 시스템을 구축했습니다.

```
public void approvePayment() {
    // 1. 데이터 유효성 검증
    validateForApproval();
    // 2. 외부 PG사에 결제 승인 요청
    response = approvalClient.approve();
    // 3. 네트워크 타임아웃 발생 시, 조회 API로 최종 상태 재확인(Fallback)
    if (response is TIMEOUT) {
        response = getPaymentResponseById();
    }
    // 4. 최종 승인 결과 확인 후 분기
    if (response is not SUCCESS) {
        throw paymentFailed();
    }
    // 5. DB 저장(@Retryable 재시도) 및 실패 시 후처리
    if (!applyApproval()) {
        recordDbWriteFailure(); // 비동기 복구를 위해 로그 남기기
    }
}
```

3. 서비스 성능 및 안정성 개선

배경 및 목표

'순수웨딩'의 포트폴리오 조회 API는 서비스 내 최대 트래픽이 집중되는 핵심 기능입니다. 한편 안정적인 서비스 운영은 효율적인 배포 환경에서 시작됩니다. 저는 서비스의 확장성을 고려하여, 트래픽 급증에 선제적으로 대응하고 어떤 상황에서도 안정적인 운영이 가능한 시스템을 구축하는 것을 목표로 삼았습니다.

이를 위해 목표를 크게 '**성능 최적화**'와 '**운영 안정성 확보**' 두 가지로 나누어 접근했습니다. 성능 측면에서는 쿼리, 캐시, 리소스 튜닝을 통해 API의 처리량을 극대화하고, 안정성 측면에서는 CI/CD 파이프라인을 고도화하여 배포 과정의 불안정성을 제거하는 데 집중했습니다.

3-1. 메인 API 성능 고도화: 쿼리, 캐시, 리소스 전방위 개선

- **문제:** 가장 **호출 빈도가 높은 포트폴리오 목록 조회 API**는 응답 시간이 길고 CPU 사용률이 높아, 트래픽 증가 시 서비스 전체의 안정성을 저해할 수 있는 병목 지점이었습니다.
- **해결:**
 - **쿼리/캐시 최적화:** DTO 프로젝트와 커서 기반 페이징으로 데이터베이스 조회 효율을 극대화했습니다. 또한, 공통 데이터는 Redis에 캐싱하고 사용자별 개인화 데이터(찜 여부 등)는 동적으로 병합하여 캐시 효율과 데이터 최신성을 모두 확보했습니다.
 - **리소스 튜닝:** k6 기반의 부하 테스트를 통해 시스템의 한계점을 파악하고, 최적의 Tomcat 스레드 및 HikariCP 커넥션 풀 사이즈를 도출하여 서버 리소스를 효율적으로 사용하도록 설정했습니다.
- **결과:** 처리량 2.4배 증가 (626→1,500 TPS), 응답 시간 47% 단축, CPU 사용률 30% 감소

3-2. 커버링 인덱스 도입: 반복 호출되는 '찜 목록' 조회 쿼리 최적화

- **문제:** 로그인한 사용자는 포트폴리오를 조회하는 여러 API에서 공통적으로 자신의 '찜 목록' 정보를 확인해야 했습니다. 기존 쿼리도 충분히 빨랐지만, **API 호출 시마다 반복 실행되어 부하가 누적될 가능성**을 줄이고, 더 나은 응답성을 확보할 여지가 있다고 판단했습니다.
- **해결:** '찜 목록' 확인에는 포트폴리오의 상세 데이터 없이 ID만 필요하다는 점에 착안하여, (`user_id`, `portfolio_id`) 조합의 **커버링 인덱스**를 적용했습니다. 이를 통해 실제 테이블 데이터 블록에 접근하는 추가적인 Disk I/O 없이, 인덱스 스캔만으로 빠르게 조회를 완료하도록 쿼리를 개선했습니다.
- **결과:** 해당 쿼리의 **응답 시간을 65% 단축(2.215ms → 0.775ms)**시켰으며, 이를 통해 누적 호출에 따른 부하를 선제적으로 줄이고 서비스 전반의 응답성에 기여했습니다.

3-3. GitHub Actions와 Docker 기반의 CI/CD 파이프라인 구축

- **문제:** 초기 CI/CD는 GitHub Actions에서 EC2에 SSH 접속하여 **서버에서 직접 빌드**하는 방식이었습니다. 이 방식은 프리티어 인스턴스(t2.micro)의 제한된 메모리와 CPU 자원으로는 무거운 Spring Boot 빌드 과정을 감당할 수 없어, **빌드 중 서버가 다운되는 등 서비스 불안정 문제**를 야기했습니다.
- **해결:** 빌드와 배포의 책임을 **명확히 분리**하는 **아키텍처**로 개선했습니다. 리소스 소모가 큰 빌드 및 Docker 이미지 생성은 GitHub Actions가 전담하고, EC2 서버는 Docker Hub에 등록된 이미지를 내려받아 실행하는 역할만 담당하도록 하여 서버 부담을 최소화했습니다.
- **결과:** 서버 리소스를 Runtime에만 집중시켜 **배포 중 서버 다운 문제를 제거**했습니다.