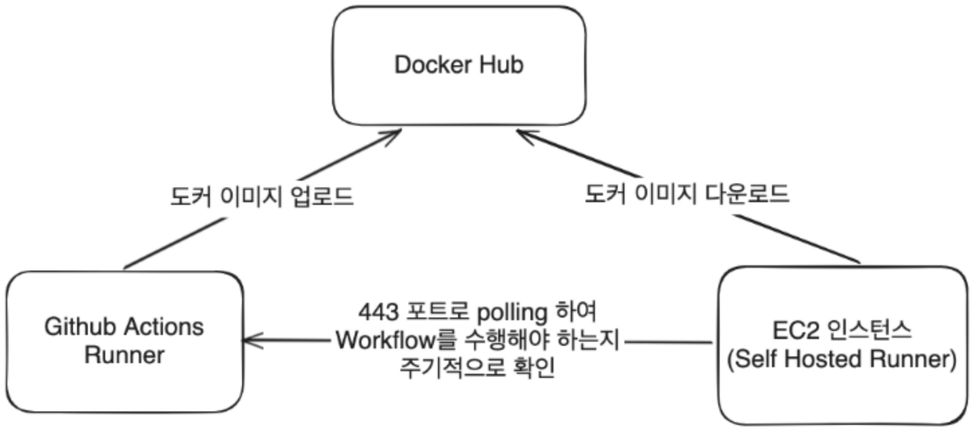
스타카토 - 간편 기록 서비스

Self Hosted Runner를 활용한 CI/CD

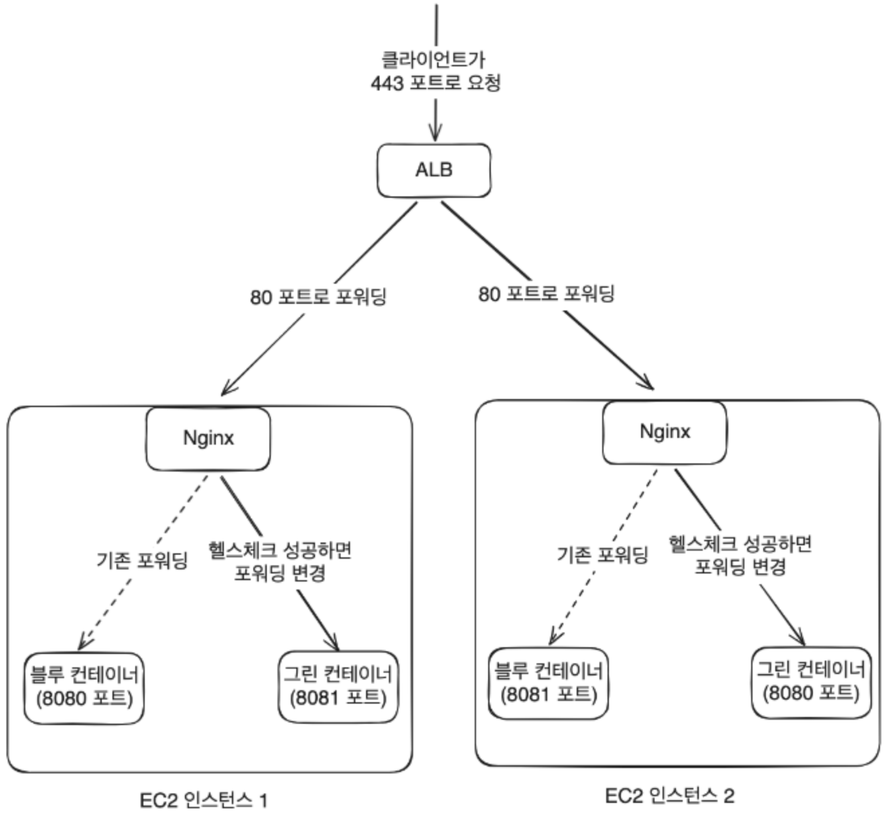
서버 애플리케이션 코드가 변경되어 배포가 필요할 때마다 git clone, build, jar 파일 실행을 위한 스크립트를 run 시키기 위해 매번 EC2에 접속해야 했고, 이는 반복적이면서 매우 번거로운 작업이었기에 자동화의 필요성을 느꼈습니다. 따라서 Github Actions를 활용해 자동화를 시도했고 Github Actions에서 제공하는 Runner를 활용해 빌드와 테스트를 자동화하는 CI Workflow 적용에 성공했습니다. 하지만 문제는 CD에 있었습니다. EC2 인스턴스의 22번 포트에 대한 인바운드 규칙이 우아한테크코스 교육장 내부 IP로만 허용되어 있었고, 따라서 Github Actions에서 제공하는 Runner가 EC2 인스턴스에 SSH 접속이 불가능했기 때문에 배포 작업을 위한 Workflow 작성에서 난항을 겪었습니다. 제공받은 IAM 사용자 계정으로는 인바운드 규칙에 대한 수정 권한이 없었기에 다른 해결책을 강구해야 했습니다.



Github Actions가 EC2 인스턴스로 접속이 불가능하다면, 반대로 EC2 인스턴스가 Github Actions에 접근하는 방식은 가능하지 않을까 고민했습니다. 아웃바운드 규칙을 확인해보니 443 포트를 통하면 외부로 요청하는게 가능했고, Github Actions에서도 Self Hosted Runner라는 polling 방식의 CD를 지원한다는 사실을 알게 되었습니다. 이 때 Self Hosted Runner가 명령어를 수행하면 서버의 자원을 사용하게 되므로 가능한 Github Actions Runner에서 작업을 수행하고 Self Hosted Runner는 최소한의 작업을 수행하도록 만들고 싶었습니다. 따라서 Github Actions Runner가 레포지토리의 코드를 다운 받아 스프링애플리케이션을 빌드하고, 도커 이미지로 만들어 도커 허브에 업로드하는 작업까지 수행하고, Self Hosted Runner는 도커 허브에 업로드된 이미지를 다운받아 컨테이너를 실행시키는 작업만 수행하도록 Workflow를 작성함으로써 서버의 자원을 최소한만 사용하면서 배포 작업을 자동화할 수 있었습니다.

스타카토 - 간편 기록 서비스

도커 컨테이너를 활용한 무중단 배포



서버 단의 코드가 변경되어 CD가 수행될 때마다, 배포가 완료될 때까지 사용자는 서비스를 이용할 수 없었기에 무중단배포의 필요성을 느꼈습니다. 당시 SPOF 문제 방지와 부하 분산을 위해 두 개의 서버 인스턴스와 로드밸런서(AWS ALB)를 사용하고 있었는데, 두 인스턴스가 항상 같은 버전의 서버를 실행할 수 있도록 블루그린 방식을 사용하려 했으나 우아한테크코스 측으로부터 전달받은 IAM 사용자 계정으로는 ALB 설정을 변경할 권한이 없었습니다.

도커 컨테이너를 사용해 서버 애플리케이션을 실행하고 있었기에, 이를 왼쪽 그림과 같이 응용하면 블루그린 방식과 비슷하게 무중단배포가 가능하지 않을까 고민했습니다.
각 인스턴스에서 내부적으로 수행되어야 하는 쉘 스크립트는 다음과 같습니다.

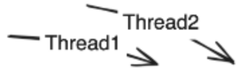
1. 현재 사용 중인 포트가 8080인지 8081인지 확인한다.
2. 현재 사용 중인 포트와 연결된 컨테이너를 블루 컨테이너라고 생각하고 8080, 8081 중 사용 중이지 않은 포트에 그린 컨테이너를 띄운다.
3. 인스턴스 자체적으로 그린 컨테이너에 대한 헬스 체크를 주기적으로 수행한다.
4. 그린 컨테이너에 헬스 체크가 완료됐다면, Nginx의 포트포워딩을 블루 컨테이너에서 그린 컨테이너로 교체한다.

이를 통해 블루그린과 유사한 무중단배포를 구현할 수 있었습니다.

방탈출 예약

동시성과 페이징을 함께 고려한 테이블 스키마 결정 (1)

예약 서비스를 구현하면서 중복 예약을 방지하기 위해 예약을 DB에 insert하기 전 특정 date, room_id에 예약이 존재하는지 조회 쿼리로 검증했습니다. 하지만 동시 요청으로 인한 스레드 경합이 발생하면 조회 쿼리를 활용한 검증이 여러 스레드에서 통과해 중복 예약이 발생할 위험이 있으므로 date, room_id를 묶어 Unique 제약조건을 설정했습니다.




```
private void validateIsDuplicated(LocalDate date, long roomId) {
    if (reservationRepository.existsByDateAndRoomId(date, roomId)) {
        throw new DuplicationException("이미 예약이 존재합니다.");
    }
}
```

이 때, 예약에 대기를 걸 수 있는 기능을 추가하면서 문제가 발생했습니다. 하나의 date, room_id에 여러 개의 대기가 생성될 수 있어 Unique 제약조건을 풀어야 했기 때문입니다. 당시 깊게 고민해 3가지 해결법을 떠올렸고, 각 방법의 장단점을 비교했습니다.

1. reservation 테이블에서 date, room_id에 대한 Unique 제약조건을 유지하고 member_id, reservation_id를 갖는 waiting 테이블을 추가하는 방법입니다. 둘을 함께 페이징 조회하려면 쿼리가 굉장히 복잡해지고, join이 강제돼 성능 저하가 발생합니다. 100만건의 데이터를 넣고 실험해보니, 이후 설명드릴 3번 테이블 구조에서의 페이징 쿼리에 비해 40배 이상의 시간이 걸렸습니다.

```
CREATE TABLE reservation
(
    id        BIGINT       NOT NULL AUTO_INCREMENT,
    member_id BIGINT       NOT NULL,
    date      timestamp    NOT NULL,
    room_id   BIGINT       NOT NULL,
    UNIQUE (date, room_id)
);

CREATE TABLE waiting
(
    id             BIGINT NOT NULL AUTO_INCREMENT,
    member_id      BIGINT NOT NULL,
    reservation_id BIGINT NOT NULL,
);
```

```
SELECT *
FROM reservation r
LEFT JOIN waiting w
    ON r.id = w.reservation_id
    AND w.member_id = 2
WHERE r.member_id = 2
   OR w.id IS NOT NULL
ORDER BY r.date
LIMIT 10, 20;
```

2번 회원의 모든 예약, 대기 페이징 조회

방탈출 예약

동시성과 페이징을 함께 고려한 테이블 스키마 결정 (2)

2. 하나의 date, room_id에 가장 빨리 생성된 예약을 예약으로, 나머지는 대기로 취급하는 방법입니다. 테이블 분리가 필요 없지만, 스레드 경합 중 먼저 시작한 트랜잭션이 늦게 커밋돼도 id가 빨라서 예약자가 되므로, 예약 성공 여부를 곧바로 판단하여 응답할 수 없습니다. 또한 예약 엔티티만으로 예약인지 대기인지 판단할 수 없으므로 비즈니스 로직이 불필요하게 복잡해집니다.

```
CREATE TABLE reservation
(
    id         BIGINT       NOT NULL AUTO_INCREMENT,
    member_id  BIGINT       NOT NULL,
    date       timestamp    NOT NULL,
    room_id    BIGINT       NOT NULL
);
```

3. date와 room_id를 slot 테이블로 분리하여 Unique 제약조건을 유지하고, reservation 테이블에서 slot_id를 참조하며 예약과 대기를 구분하기 위한 status 컬럼을 추가하는 방법입니다. slot에 존재하는 Unique 제약조건 덕분에 중복 예약이 발생할 위험이 없고, 예약 엔티티의 정보만으로 예약과 대기를 구분할 수 있으며, 예약 성공 여부도 곧바로 판단하여 응답할 수 있습니다. 또한 예약과 대기가 하나의 테이블에 존재하므로 "특정 회원의 모든 예약과 대기 조회" 같은 조회 쿼리에서 둘을 함께 페이징 조회하는 것이 간단해집니다

```
CREATE TABLE reservation
(
    id         BIGINT       NOT NULL AUTO_INCREMENT,
    member_id  BIGINT       NOT NULL,
    slot_id    BIGINT       NOT NULL,
    status     VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE reservation_slot
(
    id      BIGINT    NOT NULL AUTO_INCREMENT,
    date    timestamp NOT NULL,
    room_id BIGINT    NOT NULL,
    UNIQUE (date, room_id)
);
```

나의 예약 조회시, 예약과 대기를 함께 다루도록 요구사항을 받아 3번 해결법을 선택했지만, 둘을 함께 조회해야 하는 요구사항이 없으면서 따로 다루는 상황이 오히려 더 많았다면 1번 해결법을 선택했을 것 같습니다. 위 경험을 통해 설계에 정답은 없지만 근거를 갖춘다면 그 상황에서 최선에 가까워질 수 있다는 것을 배웠습니다.

방탈출 예약

예약 트랜잭션 실패 시 대기를 걸어주는 복구로직 구현(1)

여러 스레드가 동시에 하나의 슬롯에 대해 예약을 시도하면, reservation_slot 테이블의 date, room_id에 대한 복합 Unique 제약조건에 의해 예약에 실패하게 됩니다. 예약 대기 기능까지 존재할 정도로 예약 경쟁이 치열한 상황이므로, 먼저 요청한 사람이 빠른 대기 순번을 받는 로직이 중요하다고 생각했습니다. 따라서 예약에 실패하더라도 예약 대기를 자동으로 걸어주는 기능을 오른쪽과 같이 간단히 구현하려 시도했습니다. ReservationSlotRepository는 JpaRepository 인터페이스를 상속하고, 실제 구현체로는 SimpleJpaRepository를 주입받고 있었습니다.

```
@Transactional
public ReservationResponse createReservation(LocalDateTime date, long roomId, Member member) {
    try {
        ReservationSlot slot = createSlot(date, roomId);
        Reservation reservation = new Reservation(member, slot, ReservationStatus.RESERVED);

        return new ReservationResponse(reservationRepository.save(reservation));
    } catch (DuplicatedSlotException e) {
                슬롯이 이미 존재해서 예외가 발생하면, 기존 슬롯을 활용해 예약 대기 생성
        ReservationSlot slot = reservationSlotRepository.findByDateAndRoomId(date, roomId);
        Reservation waiting = new Reservation(member, slot, ReservationStatus.WAITING);

        return new ReservationResponse(reservationRepository.save(waiting));
    }
}

1 usage
private ReservationSlot createSlot(LocalDateTime date, long roomId) {
    Room room = roomRepository.getById(roomId);
    ReservationSlot slot = new ReservationSlot(date, room);

    return reservationSlotRepository.save(slot);   <- unique 제약 조건에 의해 예외 발생 가능
}
```

이 때, SimpleJpaRepository.save() 메서드에 달린 @Transcational 로 인해 서비스 트랜잭션의 내부 트랜잭션으로 편입되면서 slot의 Unique 제약 조건에 의해 예외가 발생하면 rollback-Only가 마킹되어 외부 트랜잭션도 커밋되지 않는 문제가 발생했습니다.

방탈출 예약

예약 트랜잭션 실패 시 대기를 걸어주는 복구로직 구현(2)

```
public ReservationResponse createReservation(ReservationRequest request, Member member) {
    if (ReservationStatus.findByViewName(request.status()) == ReservationStatus.RESERVED) {
        return createReservedReservation(request, member);
    }
    // 처음부터 대기 요청이 온 경우 아래 메서드가 수행됩니다.
    return reservationService.createWaitingReservation(request, member);
}

1 usage  devhoya97
private ReservationResponse createReservedReservation(ReservationRequest request, Member member) {
    try {
        return reservationService.createReservedReservation(request, member);
    } catch (DuplicatedSlotException e) {
        return reservationService.createWaitingReservation(request, member);
    }
}
```

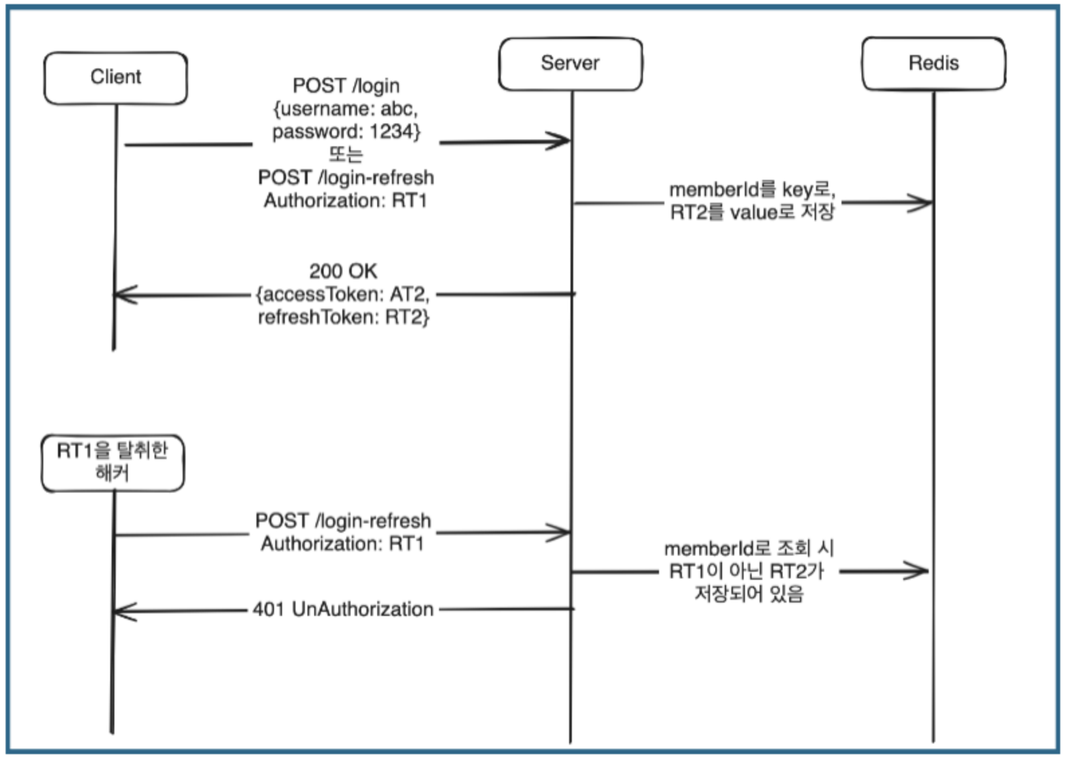
문제를 해결하기 위해, 가장 간단한 해결책으로 대기를 생성하는 메서드를 분리하고 @Transactional(propagation = Propagation.REQUIRES_NEW) 를 사용하는 방법을 떠올렸습니다.

하지만 createReservation() 메서드 내에서 같은 클래스의 메서드를 내부호출하면 프록시를 거치지 않기 때문에 @Transcational 이 정상적으로 동작하지 않았습니다.
따라서 새로운 Service 클래스를 만들어서 이를 호출해야 하는데, REQUIRED_NEW는 두 개의 커넥션을 물고 사용하므로, 굳이 새로운 클래스가 필요하다면 기존의 커넥션을 반납하고 완전히 새로운 트랜잭션을 생성하는 것이 더 좋은 해결법이라고 생각했습니다.

따라서 ReservationService를 외부에서 호출해줄 수 있는 새로운 클래스를 만들어, 예약 생성 로직을 수행하는 트랜잭션과 해당 트랜잭션이 실패했을 때 대기 생성 로직을 수행하는 트랜잭션을 완전히 분리하여 문제를 해결할 수 있었습니다.

리그오브풋볼

JWT와 Redis를 활용하여 보안과 편리성 동시에 챙기기



JWT의 만료기간은 발급시 정해지고 이후에 변경할 수 없습니다. 따라서 보안과 편리성을 고려해 로그인 시 만료기간이 짧은 accessToken과 만료 기간이 긴 refreshToken을 함께 발급하여 refreshToken의 만료기간 동안 재로그인이 필요없도록 만들었습니다.

refreshToken의 만료기간마다 재로그인이 필요한 것도 불편했기에,
accessToken을 재발급할 때마다 refreshToken도 함께 재발급했습니다.
하지만 refreshToken으로 새로운 refreshToken까지 재발급 받을 수 있기에
refreshToken이 탈취되면, accessToken을 무한정 재발급 가능한 문제가 생겼습니다.

이를 극복하기 위해 refreshToken이 새로 발급될 때마다 memberId를 key로,
refreshToken을 value로 Redis에 저장했습니다. 따라서 refreshToken이
탈취되더라도 사용자가 먼저 refreshToken을 재발급 받거나, 재로그인하거나,
Redis에 저장된 refreshToken을 직접 삭제하면 해커의 공격을 막을 수 있습니다.

accessToken의 만료주기마다 특정 회원의 refreshToken이 재발급되어
그 수가 많아질 수 있지만, 이 중 유효한 refreshToken 하나만을 Redis에서
관리하고 있으므로 보안과 편리성을 모두 잡을 수 있었습니다.