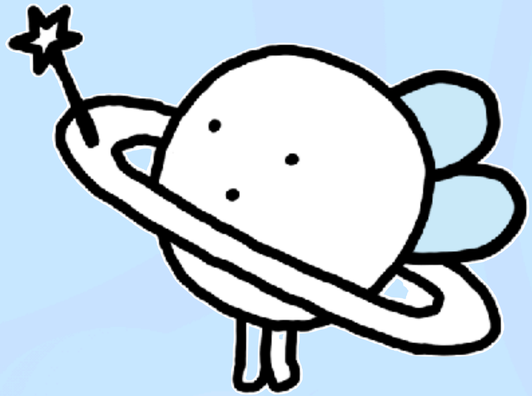
야구보구에서

경기 결과를 빠르고 효율적으로 가져오는 방법

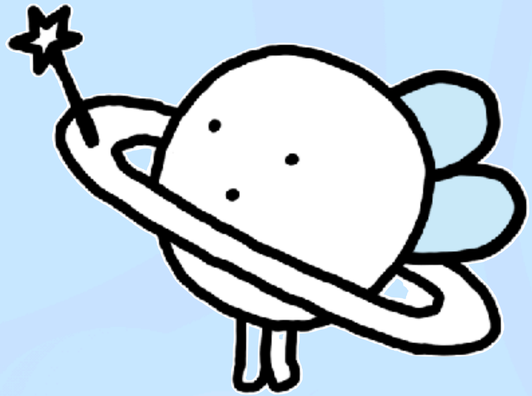
BE 7기 밍트

개요



1. 배경
2. Adaptive Polling
3. jitter & 지수 백오프
4. DB 최소 업데이트
5. 마무리

개요

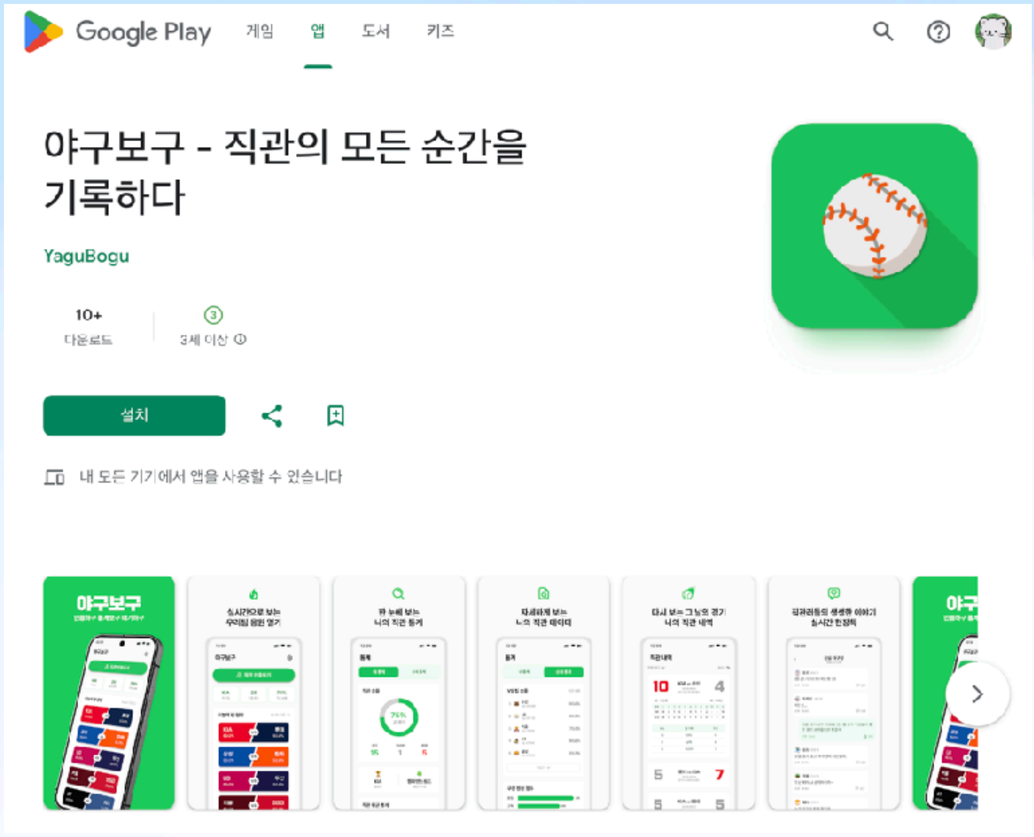


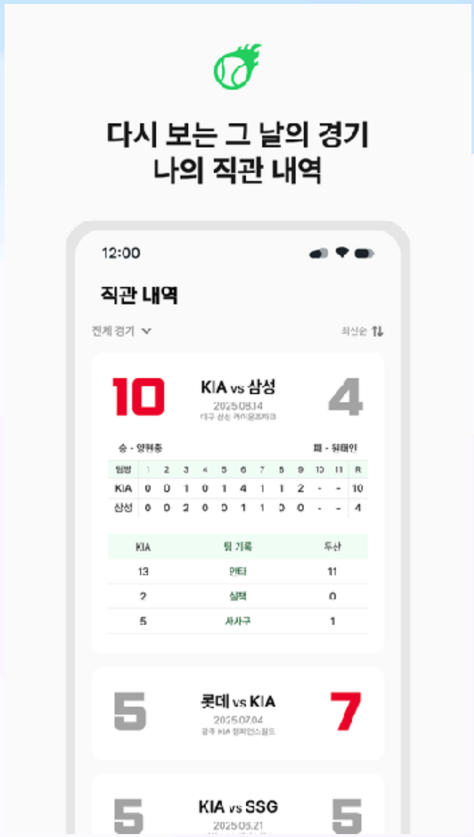
1. 배경
2. Adaptive Polling
3. jitter & 지수 백오프
4. DB 최소 업데이트
5. 마무리

1. 배경

야구보구 서비스 런칭 🥳

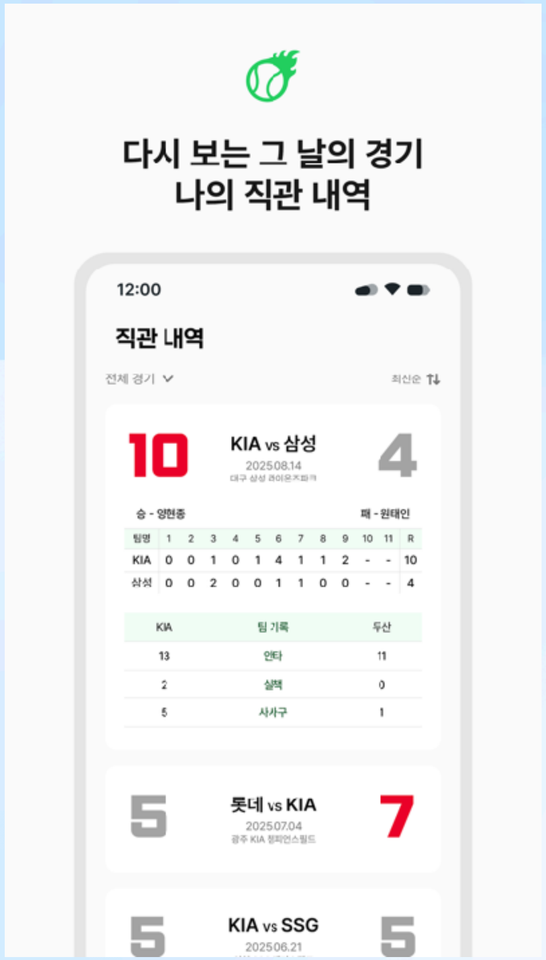
https://play.google.com/store/apps/details?id=com.yagubogu





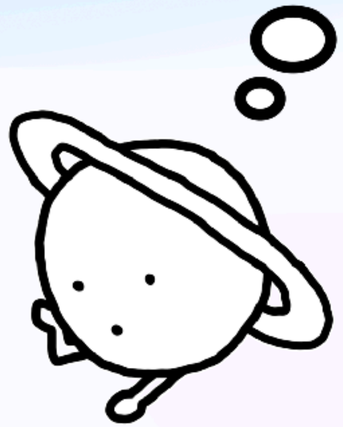
1. 배경

경기 결과 업데이트 주기



경기 결과가 최대한 빨리
반영되면 좋겠어요.

기존 방식은 단순한 cron인데..



1. 배경

기존 스케줄링 방식

```java
@Slf4j
@RequiredArgsConstructor
@Component
public class GameScheduler {

    private final GameScheduleSyncService gameScheduleSyncService;
    private final GameResultSyncService gameResultSyncService;

    @Scheduled(cron = "0 0 0 * * *")
    public void fetchDailyGameSchedule() {
        LocalDate today = LocalDate.now();
        try {
            gameScheduleSyncService.syncGameSchedule(today);
        } catch (GameSyncException e) {
            log.error("[GameSyncException]- {}", e.getMessage());
        }
    }
}
```

🔷 게임 결과 자정 업데이트

→ 경기 종료 직후 반영 불가

→ 사용자 경험 저하

09.19(금)	18:30	두산	vs	SSG
	18:30	롯데	vs	NC
	18:30	한화	vs	KT
09.20(토)	17:00	삼성	vs	LG
	17:00	두산	vs	SSG
	17:00	키움	vs	롯데
	17:00	한화	vs	KT
	17:00	NC	vs	KIA
09.21(일)	14:00	두산	vs	SSG
	14:00	삼성	vs	KT
	14:00	NC	vs	KIA

🔷 야구 시작 시각 변동

→ 날짜별, 구장별, 우천 지연...

1. 배경

스케줄링 주기를 짧게 가져가면?

🔷 5분 간격 스케줄링

```java
@Slf4j
@RequiredArgsConstructor
@Component
public class GameScheduler {

    private final GameScheduleSyncService gameScheduleSyncService;
    private final GameResultSyncService gameResultSyncService;

    @Scheduled(cron = "0 */5 * * * *")
    public void fetchDailyGameSchedule() {
        LocalDate today = LocalDate.now();
        try {
            gameScheduleSyncService.syncGameSchedule(today);
        } catch (GameSyncException e) {
            log.error("[GameSyncException]- {}", e.getMessage());
        }
    }
}
```

🔷 비효율적인 호출

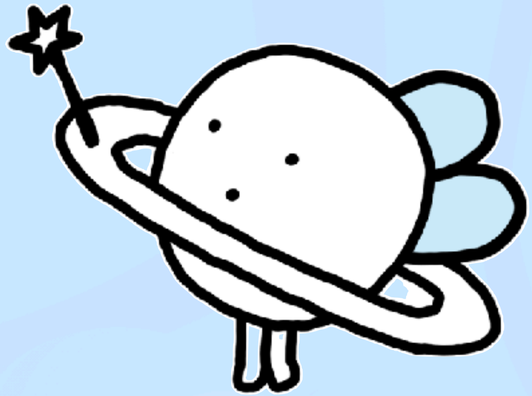
→ 경기가 진행되지 않은 시간에도 계속 API 호출

🔷 경기 상황별 최적화 불가

→ 경기 초반 5분 간격은 충분하다.

→ 경기 후반, 종료 직전에 더 짧게 돌려야
빠르게 결과 감지 가능

개요



1. 배경
2. Adaptive Polling
3. jitter & 지수 백오프
4. DB 최소 업데이트
5. 마무리

2. Adaptive Polling

Adaptive Polling

스케줄링 주기를 상황에 맞춰 동적으로 조절한다.

상태별 폴링 간격

상태	경과 시간	폴링 간격	비고
SCHEDULED	< 120분	10분	일반 대기 상태
SCHEDULED	≥ 120분	15분	장기 지연 시
LIVE	< 90분	5분	경기 초반
LIVE	90~170분	**4~5분 랜덤**	경기 중반
LIVE	≥ 170분	**2~3분 랜덤**	종료 임박, 촘촘

LIVE 구간에서는 랜덤 분산(4~5분, 2~3분)으로 API 호출 폭주 방지

```java
private static final int[] MID_INTERVAL_OPTIONS = new int[]{4, 5};
private static final int[] LATE_INTERVAL_OPTIONS = new int[]{2, 3};

private Duration computePollingInterval(LocalDate date, LocalTime startAt, Instant now, GameState state) {
      Instant plannedStart = toStartInstant(date, startAt);
      long elapsedMin = Duration.between(plannedStart, now).toMinutes();

      if (state == GameState.SCHEDULED) {
          if (elapsedMin < 120) {
              return Duration.ofMinutes(10);
          }
          return Duration.ofMinutes(15);
      }

      if (elapsedMin < 90) {
          return Duration.ofMinutes(5);
      }

      if (elapsedMin < 170) {
          int choice = MID_INTERVAL_OPTIONS[ThreadLocalRandom.current().nextInt(MID_INTERVAL_OPTIONS.length)];
          return Duration.ofMinutes(choice);
      }

      int choice = LATE_INTERVAL_OPTIONS[ThreadLocalRandom.current().nextInt(LATE_INTERVAL_OPTIONS.length)];
      return Duration.ofMinutes(choice);
  }
```

2. Adaptive Polling

Adaptive Polling

polling 주기를 상황에 맞춰 동적으로 조절한다.

상태별 폴링 간격

상태	경과 시간	폴링 간격	비고
SCHEDULED	< 120분	10분	일반 대기 상태
SCHEDULED	≥ 120분	15분	장기 지연 시
LIVE	< 90분	5분	경기 초반
LIVE	90~170분	**4~5분 랜덤**	경기 중반
LIVE	≥ 170분	**2~3분 랜덤**	종료 임박, 촘촘

LIVE 구간에서는 랜덤 분산(4~5분, 2~3분)으로 API 호출 폭주 방지

🔷 실행 스케줄 관리

```java
private final Map<Long, Instant> nextRunAt = new ConcurrentHashMap<>();
private volatile Instant globalWakeAt = Instant.EPOCH;

@Scheduled(fixedDelay = 60_000)
public void tick() {
    for (Game g : todayGames) {
        if (now.isBefore(nextRunAt.get(g))) continue;
        gameResultSyncService.updateGameDetails(...);

        if (g.isFinalized()) {
          nextRunAt.remove(g);
          continue;
        }
        nextRunAt.put(g, now.plus(interval).plusSeconds(jitter()));
    }
}
```

각 경기별 nextRunAt 추가 / 제거

2. Adaptive Polling

Adaptive Polling

polling 주기를 상황에 맞춰 동적으로 조절한다.

상태별 폴링 간격

상태	경과 시간	폴링 간격	비고
SCHEDULED	< 120분	10분	일반 대기 상태
SCHEDULED	≥ 120분	15분	장기 지연 시
LIVE	< 90분	5분	경기 초반
LIVE	90~170분	**4~5분 랜덤**	경기 중반
LIVE	≥ 170분	**2~3분 랜덤**	종료 임박, 촘촘

LIVE 구간에서는 랜덤 분산(4~5분, 2~3분)으로 API 호출 폭주 방지

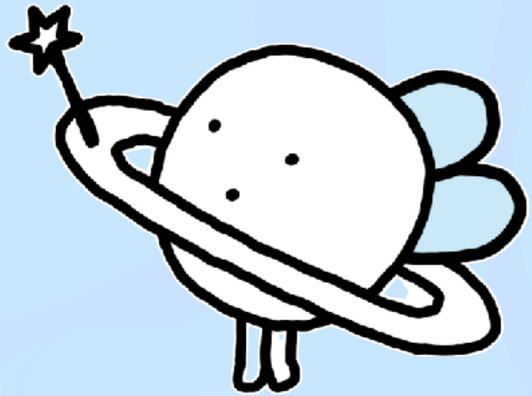
🔷 자정 + 부팅 시에도 로딩

```java
@Scheduled(cron = "0 0 0 * * *")
@EventListener(ApplicationReadyEvent.class)
public void fetchDailyGameSchedule() {
    LocalDate today = LocalDate.now(clock);
    try {
        gameScheduleSyncService.syncGameSchedule(today);
        adaptivePoller.loadMidnightOrStartup(today);
    } catch (GameSyncException e) {
        log.error("[GameSyncException]- {}", e.getMessage());
    }
}
```

자정(00:00) 배포 시에도 polling이 끊기지 않기 위함

→ 안전한 서비스 운영 보장하기 😀

개요



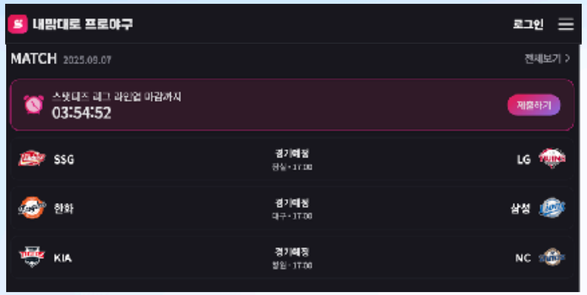
1. 배경
2. Adaptive Polling
3. jitter & 지수 백오프
4. DB 최소 업데이트
5. 마무리

3. Jitter & 지수 백오프

Jitter

polling 주기를 랜덤하게 분산시키기

🔷 필요성



KBO 경기는 동시에 여러 경기가 시작한다.

KBO 서버에 같은 시간에 API 요청이 몰릴 수 있다.

→ jitter를 통해 요청 분산시키기

🔷 작동 방식

```
Instant baseDue = computeBaseDue(planned, now, interval);
Instant due = baseDue.plusSeconds(generateJitter(preStart));
```

```
private long generateJitter(boolean preStart) {
  if (preStart) {
      return ThreadLocalRandom.current().nextLong(-10, 11);
  }
  return ThreadLocalRandom.current().nextLong(-30, 31);
}
```

- 경기 시작 전 기본 ±10s
- 경기 시작 후 ±30s

2. Adaptive Polling

Jitter

polling 주기를 랜덤하게 분산시키기

상태별 폴링 간격

상태	경과 시간	폴링 간격	비고
SCHEDULED	< 120분	10분	일반 대기 상태
SCHEDULED	≥ 120분	15분	장기 지연 시
LIVE	< 90분	5분	경기 초반
LIVE	90~170분	**4~5분 랜덤**	경기 중반
LIVE	≥ 170분	**2~3분 랜덤**	종료 임박, 촘촘

LIVE 구간에서는 랜덤 분산(4~5분, 2~3분)으로 API 호출 폭주 방지

🔷 랜덤 폴링 간격

```java
private static final int[] MID_INTERVAL_OPTIONS = new int[]{4, 5};
private static final int[] LATE_INTERVAL_OPTIONS = new int[]{2, 3};

private Duration computePollingInterval(LocalDate date, LocalTime startAt, Instant now, GameState state) {
      // etc

      if (elapsedMin < 170) {
          int choice = MID_INTERVAL_OPTIONS[ThreadLocalRandom.current().nextInt(MID_INTERVAL_OPTIONS.length)];
          return Duration.ofMinutes(choice);
      }

      int choice = LATE_INTERVAL_OPTIONS[ThreadLocalRandom.current().nextInt(LATE_INTERVAL_OPTIONS.length)];
      return Duration.ofMinutes(choice);
  }
```

폴링 주기를 항상 고정하지 않고, 랜덤으로 선택해서 분산

→ 트래픽이 특정 시각에 몰리는 현상 방지

3. Jitter & 지수 백오프

지수 백오프

실패할 수록 재시도 간격을 기하급수적으로 늘리는 전략

🔷 필요성

KBO 서버에 장애가 나거나, 느려질 경우

고정 주기로 재시도하면 장애 전파 + 제한 강화

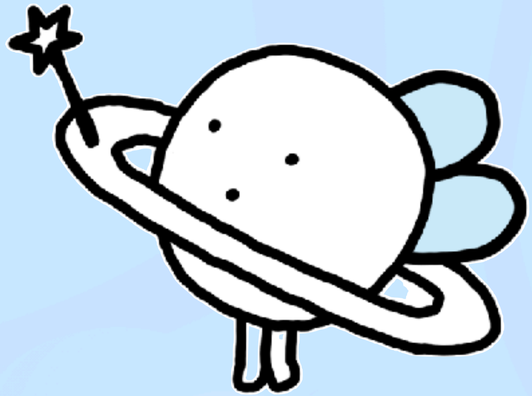
→ 간격을 점점 늘려 재시도하여 모두를 보호한다.

🔷 작동 방식

```java
private final Map<Long, Integer> failureCount = new ConcurrentHashMap<>();

private void onCallSuccess(Long gameId) {
    if (gameId != null) failureCount.remove(gameId);
}

private void onCallFailure(Long gameId) {
    int fail = failureCount.merge(gameId, 1, Integer::sum);
    // 1, 2, 4, 8, 16, 32분… (30분 초과는 안함)
    int minutes = Math.min(30, 1 << Math.min(fail, 5));
    Instant due = Instant.now(clock).plus(Duration.ofMinutes(minutes));
    nextRunAt.put(gameId, due);
    if (fail >= 5) {
        log.warn("Poll repeatedly failed: gameId={}, failCount={}", gameId, fail);
    }
}
```

- 실패시 1 → 2 → 4 → 8 → .. → 30분 뒤 재시도

개요



1. 배경
2. Adaptive Polling
3. jitter & 지수 백오프
4. **DB 최소 업데이트**
5. 마무리

4. DB 최소 업데이트

DB 부하를 줄이자

필요성

polling → 잦은 읽기 / 쓰기 작업

→ DB 병목 발생 가능

바뀐 부분만 업데이트

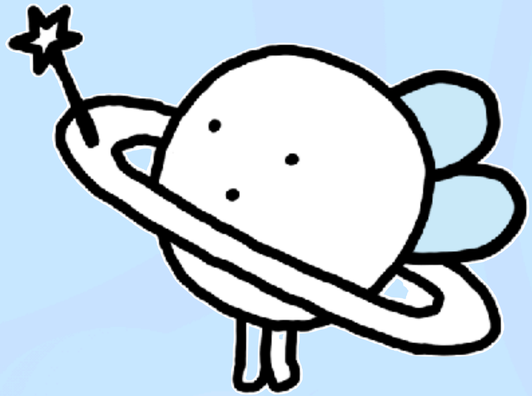
```
changed |= setIfDiff(this::getHomeScore, this::setHomeScore, newHome.getRuns());
changed |= setIfDiff(this::getAwayScore, this::setAwayScore, newAway.getRuns());

changed |= this.homeScoreBoard.updateFrom(newHome);
changed |= this.awayScoreBoard.updateFrom(newAway);
```

```
public boolean replaceInningsIfDiff(final List<String> newInnings) {
    final List<String> normalized = new ArrayList<>(newInnings);
    if (!Objects.equals(this.inningScores, normalized)) {
        this.inningScores = normalized; // 새로운 인스턴스로 교체(더티체킹 안전)
        return true;
    }
    return false;
}
```

값이 같으면 DB Update 발생 ❌

→ **Dirty Checking 최소화**

개요



1. 배경
2. Adaptive Polling
3. jitter & 지수 백오프
4. DB 최소 업데이트
5. 마무리

5. 마무리

결과 & 효과

🔷 사용자 경험 개선

→ 경기 실시간 / 종료 시 5 ~ 10분 내 빠른 결과 반영

🔷 DB 부하 절감

→ applyDiff로 불필요한 update 최소화

🔷 운영 안정성 확보

→ polling 호출량 절감 + 외부 API 장애 전파 차단

가치 + 안정성 + 효율성

꼿