14. 스케줄링과 작업 처리

`@Scheduled`, `@EnableScheduling`

Spring Framework는 **스케줄링 기능**을 기본 제공한다.
`@Scheduled`와 `@EnableScheduling` 어노테이션을 활용하면 간단하게 **주기적인 작업(배치, 모니터링, 알림 등)을 자동으로 실행**할 수 있다.

구현이 매우 간단하며, 별도 외부 스케줄러 없이 **Spring Context** 내에서 관리된다.

1️⃣ 기본 구성

`@EnableScheduling`

- 스케줄링 기능을 활성화하는 어노테이션.
- 일반적으로 **Configuration 클래스**에 선언.

```
@Configuration
@EnableScheduling
public class SchedulingConfig {
    // 빈 등록이나 별도 설정 가능
}
```

이 어노테이션이 선언되어 있어야 `@Scheduled`가 정상 동작한다.
→ 스프링이 내부적으로 **TaskScheduler**를 등록한다.

`@Scheduled`

- 주기적으로 실행할 **메서드에 적용**하는 어노테이션.

```
@Component
public class ScheduledTasks {

    @Scheduled(fixedRate = 5000)  // 5초마다 실행
    public void reportCurrentTime() {
        System.out.println("현재 시간: " + LocalDateTime.now());
    }
}
```

📝 주의: `@Scheduled`가 붙은 메서드는 **반드시 void 리턴**, 인자 없어야 함.
(Spring이 프록시로 감싸서 스케줄링하기 때문)

2️⃣ 주요 속성 정리 🗂️

속성명	설명	예시
`fixedRate`	이전 실행 시작 시점 기준 **고정 주기**로 실행	`fixedRate = 5000` → 5초마다 실행
`fixedDelay`	이전 실행 종료 시점 기준 **고정 지연 후** 실행	`fixedDelay = 5000` → 실행 끝난 후 5초 후 재실행
`initialDelay`	처음 실행까지 대기 시간	`initialDelay = 10000` → 10초 후 첫 실행
`cron`	cron 표현식 기반으로 정교하게 스케줄링	`cron = "0 0 * * * *"` → 매 정시마다 실행

3️⃣ cron 표현식 활용 🌟

형식: `second minute hour day-of-month month day-of-week [year]`

예시

cron 표현식	의미
`"0 * * * * *"`	매 분 0초마다
`"0 0 * * * *"`	매 정시마다
`"0 0 9 * * *"`	매일 오전 9시
`"0 0 9 * * MON-FRI"`	월~금 오전 9시에 실행
`"0 0/5 * * * *"`	매 5분마다

구성요소 설명:

- `0` 초 단위 (0초부터 시작)
- `*` 전체 범위
- `0/5` 5분 간격

참고 사이트 → https://crontab.guru/
(실시간 cron 표현식 검증 가능 🔍)

4️⃣ 실전 예제

고정 주기 실행

```
@Scheduled(fixedRate = 10000)
public void sendHeartbeat() {
    System.out.println("Heartbeat 전송: " + LocalDateTime.now());
}
```

초기 지연 + 고정 딜레이

```
@Scheduled(initialDelay = 5000, fixedDelay = 10000)
public void performBatchJob() {
    System.out.println("배치 작업 실행 완료: " + LocalDateTime.now());
}
```

cron 기반 스케줄링

```
@Scheduled(cron = "0 0 2 * * *")  // 매일 새벽 2시 정각 실행
public void backupDatabase() {
    System.out.println("데이터베이스 백업 실행: " + LocalDateTime.now());
}
```

5️⃣ 주의 사항 ⚠️

- 기본적으로 `@Scheduled` 메서드는 **싱글 쓰레드(TaskScheduler)** 에 의해 실행됨 → 병렬 처리 필요 시 별도 설정 필요.
- 메서드 실행 시간이 고정 주기보다 길 경우 → overlapping(겹침) 가능성 있음 → 필요 시 `@Async` + ThreadPoolExecutor 사용 고려.
- `@Scheduled`는 프로덕션 환경에서 너무 과도하게 남발 시 **애플리케이션 부하 유발** → 신중하게 적용할 것.

6️⃣ ThreadPool 기반 스케줄링 🚀

기본은 싱글 쓰레드 → 병렬 실행하려면 ThreadPoolTaskScheduler 사용.

```java
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
        taskScheduler.setPoolSize(5);
        taskScheduler.setThreadNamePrefix("Scheduler-");
        taskScheduler.initialize();
        taskRegistrar.setTaskScheduler(taskScheduler);
    }
}
```

효과:

- 스케줄링 작업을 **멀티 쓰레드** 기반으로 처리 가능.
- long-running 작업 겹침 방지.

7️⃣ 활용 시나리오 🎯

- 주기적 **배치 작업 실행** (ex. 통계 집계)
- 주기적 **데이터 동기화**
- 시스템 **상태 점검 / 모니터링**
- 외부 API 데이터 주기적 수집
- **로그 정리 작업** 주기적 실행

결론

어노테이션	역할
`@EnableScheduling`	스케줄링 기능 활성화
`@Scheduled`	메서드 단위 주기적 실행 등록

- **간단한 주기적 작업 구현 시 매우 유용**한 기능.
- 고정 주기(fixedRate), 고정 지연(fixedDelay), cron 표현식 모두 지원.
- 복잡한 병렬 처리 시 별도 ThreadPool 구성 가능.

Quartz 스케줄러 연동

Quartz는 강력하고 유연한 **스케줄링 라이브러리**로, **정교한 스케줄 관리**가 필요한 엔터프라이즈급 애플리케이션에서 널리 사용된다.
Spring Boot에서는 Quartz를 손쉽게 통합할 수 있으며, `@Scheduled` 보다 고급 기능을 제공한다.

Quartz는 다음과 같은 기능을 제공한다:

- Cron 표현식 기반 스케줄링
- Job 상태 관리 및 영속화 (DB 연동)
- 분산 환경 지원 (클러스터링)
- Job 중복 방지(동시 실행 방지)
- Job 실행 기록 관리
- 재시도, Misfire 처리 지원

1️⃣ Quartz 기본 개념

Quartz는 다음 구성요소로 이루어진다:

구성요소	설명
Job	실행할 작업 단위
Trigger	Job 실행 조건 및 시점 설정
Scheduler	Job과 Trigger를 관리하며 실행 주관
JobStore	Trigger/Job 상태 저장소 (메모리/DB)

Quartz는 기본적으로 **RAMJobStore**(메모리 기반)를 사용하지만, 운영 환경에서는 **JDBCJobStore**(DB 기반 영속화)를 사용하는 경우가 많다.

2️⃣ 의존성 추가

Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-quartz</artifactId>
</dependency>
```

Gradle

```
implementation 'org.springframework.boot:spring-boot-starter-quartz'
```

👉 이로써 Spring Boot에서 Quartz Scheduler가 자동으로 구성된다.

3️⃣ Quartz Job 작성

Quartz에서 실행할 작업은 **Job 인터페이스**를 구현하여 작성한다.

```java
@Component
public class SampleJob implements Job {

    @Override
    public void execute(JobExecutionContext context) throws JobExecutionException {
        System.out.println("SampleJob 실행 시간: " + LocalDateTime.now());
    }
}
```

- `execute` 메서드가 실행 시 호출된다.
- 필요한 경우 **JobDataMap**을 통해 파라미터 전달 가능.

4️⃣ JobDetail과 Trigger 설정

Quartz에서 Job 실행을 위해서는 **JobDetail**과 **Trigger**를 등록해야 한다.

```java
@Configuration
public class QuartzConfig {

    @Bean
    public JobDetail sampleJobDetail() {
        return JobBuilder.newJob(SampleJob.class)
            .withIdentity("sampleJob")
            .storeDurably()
            .build();
    }

    @Bean
    public Trigger sampleJobTrigger() {
        SimpleScheduleBuilder scheduleBuilder = SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInSeconds(10)  // 10초마다 실행
            .repeatForever();

        return TriggerBuilder.newTrigger()
            .forJob(sampleJobDetail())
            .withIdentity("sampleTrigger")
            .withSchedule(scheduleBuilder)
            .build();
    }
}
```

주요 구성 요소

- `JobDetail` → 실행할 Job 정의
- `Trigger` → 실행 주기/시점 정의

👉 위 예시는 10초마다 `SampleJob`을 반복 실행하는 구성이다.

5️⃣ CronTrigger 사용 (고급 스케줄링) 🌟

보다 정교한 스케줄링을 위해 **CronTrigger**를 사용할 수 있다.

```
@Bean
public Trigger sampleJobCronTrigger() {
    return TriggerBuilder.newTrigger()
        .forJob(sampleJobDetail())
        .withIdentity("sampleCronTrigger")
        .withSchedule(CronScheduleBuilder.cronSchedule("0 0/1 * * * ?"))  // 매 1분마다 실행
        .build();
}
```

Cron 표현식은 Quartz에서 `?` 문법 등을 사용하므로 일반 Cron과 약간 차이가 있음.

예시 참고 → https://www.freeformatter.com/cron-expression-generator-quartz.html 🔍

6️⃣ Quartz 영속화 구성 (JDBCJobStore) 🗂️

운영 환경에서 Job 상태(Trigger 상태, 실행 이력 등)를 DB에 영속화하려면 다음 설정이 필요하다.

1. Quartz 테이블 생성

Quartz 공식 문서에서 제공하는 **DDL 스크립트**를 실행하여 DB에 테이블 구성.

```
org/quartz/impl/jdbcjobstore/tables_mysql_innodb.sql
```

(※ DBMS별로 제공됨 → MySQL, PostgreSQL, Oracle 등)

2. application.yml 설정

```
spring:
  quartz:
    job-store-type: jdbc
    jdbc:
      initialize-schema: always  # 초기 테이블 자동 생성 (운영 환경에서는 off 추천)
    properties:
      org:
        quartz:
          scheduler:
            instanceName: QuartzScheduler
          threadPool:
            threadCount: 10
```

- `job-store-type: jdbc` 설정 시 JDBCJobStore 사용
- 영속화된 Trigger 상태는 DB에 저장 → 서버 재시작 후에도 유지됨 🔁

7️⃣ Quartz vs @Scheduled 비교

항목	Quartz	@Scheduled
스케줄 관리	정교한 Trigger 설정	단순 주기 실행
상태 영속화	지원 (JDBCJobStore 사용 시)	미지원 (메모리 기반)
분산 환경 지원	클러스터링 지원	미지원
Job 관리 기능	다양한 Trigger, 재시도, Misfire 처리	단순 Timer 기반
사용 용도	엔터프라이즈 배치/스케줄링	간단한 주기적 작업

👉 Quartz는 **운영 환경에서 안정적이고 정교한 배치/스케줄링**이 필요한 경우 적합.
👉 `@Scheduled`는 **간단한 내부 주기 작업** 처리에 적합.

8️⃣ 종합 구성 흐름

```
Application Startup → Quartz Scheduler 초기화
   ↓
JobDetail 등록 → Trigger 등록
   ↓
Scheduler가 Trigger 시간에 따라 Job 실행
   ↓
Job ExecutionContext → execute() 호출
   ↓
Job 상태 및 실행 이력 관리 (JDBCJobStore 사용 시 DB 반영)
```

결론

✅ Quartz는 강력한 스케줄링 기능을 제공하는 외부 라이브러리.
✅ Spring Boot Starter로 손쉽게 통합 가능.
✅ JDBCJobStore 구성 시 분산 환경에서도 안정적인 Job 관리 가능.
✅ `@Scheduled` 보다 유연하고 확장성 높은 스케줄링 기능 제공.

TaskExecutor, ThreadPool 설정

Spring에서는 **비동기 처리** 또는 **병렬 처리**가 필요한 경우, `TaskExecutor`와 `ThreadPool`을 구성하여 **효율적이고 안정적인 스레드 관리**를 구현할 수 있다.
기본적으로 `TaskExecutor`는 **Spring의 비동기 작업을 추상화**한 인터페이스이며, 내부적으로는 `java.util.concurrent.Executor` 기반으로 구현된다.

1️⃣ 주요 용도

- `@Async` 메서드 실행 시 TaskExecutor 사용
- `@Scheduled` 스케줄링 시 ThreadPool 구성
- 비동기 이벤트 처리
- 병렬 처리 기반 배치 작업
- 대용량 API 호출 시 비동기 병렬 처리

2️⃣ 기본 개념

구성요소	설명
TaskExecutor	스레드 풀을 추상화한 Spring 인터페이스
ThreadPoolTaskExecutor	기본 제공되는 ThreadPool 기반 TaskExecutor 구현체
ThreadPoolTaskScheduler	스케줄링 작업용 ThreadPool 기반 Scheduler

Spring Boot에서는 `ThreadPoolTaskExecutor`를 주로 사용하여 비동기 작업을 수행한다.

3️⃣ ThreadPoolTaskExecutor 설정 🚀

1. Bean 정의

```
@Configuration
public class AsyncConfig {

    @Bean(name = "taskExecutor")
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(10);            // 기본 스레드 수
        executor.setMaxPoolSize(20);             // 최대 스레드 수
        executor.setQueueCapacity(500);          // 큐 크기 (대기 작업 수)
        executor.setThreadNamePrefix("Async-");  // 스레드 이름 접두사
        executor.setWaitForTasksToCompleteOnShutdown(true);  // 종료 시 대기 여부
        executor.setAwaitTerminationSeconds(30);  // 종료 대기 시간
        executor.initialize();
        return executor;
    }
}
```

주요 설정 값

설정값	설명
`corePoolSize`	기본 유지 스레드 수

설정값	설명
`maxPoolSize`	최대 생성 가능 스레드 수
`queueCapacity`	작업 큐 용량 (큐가 꽉 차면 maxPoolSize까지 스레드 생성)
`threadNamePrefix`	스레드 이름 접두사 지정 (디버깅 시 유용)
`waitForTasksToCompleteOnShutdown`	애플리케이션 종료 시 대기 여부
`awaitTerminationSeconds`	종료 시 최대 대기 시간 (초)

---

2. @Async 적용

비동기 실행 메서드는 `@Async` 어노테이션으로 선언한다.

```
@Service
public class AsyncService {

    @Async("taskExecutor")
    public void performAsyncTask() {
        System.out.println("Async 작업 시작: " + Thread.currentThread().getName());
        // 비동기 작업 로직
    }
}
```

- `@Async` 어노테이션으로 TaskExecutor를 명시적으로 지정 가능.
- 메서드는 반드시 `void` 또는 `Future<T>`, `CompletableFuture<T>` 형태로 정의.

4️⃣ ThreadPoolTaskScheduler 설정 🕒

스케줄링 작업(`@Scheduled`)에 사용할 별도 ThreadPool도 구성 가능하다.

```
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
        taskScheduler.setPoolSize(5);  // 스케줄링용 스레드 풀 크기
        taskScheduler.setThreadNamePrefix("Scheduler-");
        taskScheduler.initialize();
        taskRegistrar.setTaskScheduler(taskScheduler);
    }
}
```

활용 효과

- 기본 `@Scheduled`는 단일 쓰레드로 동작 → **병렬 스케줄링 시 ThreadPoolTaskScheduler 사용 필요**.

5️⃣ ThreadPool tuning 전략 ⚙️

튜닝 시 고려 사항

항목	고려 요소
corePoolSize	서버 CPU 코어 수 기반 설정 (ex. CPU 코어 수 * 2 등)
maxPoolSize	과부하 상황 대응 → 비즈니스 특성에 맞게 설정
queueCapacity	작업량 특성 분석 후 적정 크기 설정 (너무 크면 응답 지연 발생 가능)
threadNamePrefix	운영 디버깅 편의성 확보
graceful shutdown	shutdown 시 작업 정상 종료 보장 필요

일반적인 추천 값

- CPU 바운드 작업 → corePoolSize = CPU 코어 수
- I/O 바운드 작업 → corePoolSize = CPU 코어 수 * 2~4
- queueCapacity는 트래픽 패턴과 latency 특성 고려하여 설정

6️⃣ 종합 구성 흐름

```
Application Startup → ThreadPoolTaskExecutor Bean 등록
   ↓
Service Layer → @Async 메서드 호출
   ↓
TaskExecutor → 별도 스레드 풀에서 비동기 실행
   ↓
ThreadPoolTaskScheduler → 다중 스케줄링 작업 관리
```

7️⃣ TaskExecutor vs ExecutorService 차이

항목	TaskExecutor	ExecutorService
정의	Spring 추상화	Java 표준 라이브러리
구성 방식	Bean으로 관리 → DI 가능	직접 코드로 구성 필요
스프링 통합	쉽게 통합	수동 설정 필요
AOP, @Async 지원	지원	직접 구현 필요

👉 일반적인 Spring Boot 환경에서는 **TaskExecutor 사용 권장**.
👉 Spring Batch, Quartz 등과도 쉽게 통합 가능.

결론 🗂️

✅ TaskExecutor는 Spring에서 비동기 및 병렬 처리 구현 시 표준적으로 사용하는 스레드 풀 인터페이스.
✅ ThreadPoolTaskExecutor를 통해 유연하고 강력한 ThreadPool 구성 가능.
✅ 스케줄링 작업에는 ThreadPoolTaskScheduler 구성으로 병렬 실행 지원.
✅ 적절한 ThreadPool 튜닝을 통해 안정적이고 효율적인 서비스 운영 가능.