Tak, można to zrealizować przy użyciu **Quarza**. Quartz jest bardzo elastyczny i pozwala precyzyjnie kontrolować mechanizm wyzwalania zadań w środowisku rozproszonym.

Rozwiązanie z Quarzem

1. Zastosowanie mechanizmu `DisallowConcurrentExecution`

- Quartz umożliwia oznaczenie zadania jako nieuruchamialnego równocześnie przez wiele wątków lub instancji. Wystarczy dodać adnotację `@DisallowConcurrentExecution` do klasy zadania.
- Jeśli jedna instancja zadania wciąż działa, kolejna nie zostanie uruchomiona, dopóki poprzednia się nie zakończy.

2. Synchronizacja w klastrze

- Quartz posiada wbudowany mechanizm synchronizacji w klastrze przy użyciu bazy danych jako wspólnego repozytorium.
- W konfiguracji klastra tylko jedna instancja węzła uruchomi dane zadanie.

3. Dynamiczny interwał startu (co 5 minut po zakończeniu zadania)

- Zamiast używać stałego harmonogramu (np. CRON), możesz ustawić, aby następne wywołanie zadania było rejestrowane dynamicznie, zaraz po zakończeniu bieżącego zadania.

Przykładowa implementacja

Konfiguracja Quarza

1. Dodaj zależności:

```
<dependency>
   <groupId>org.quartz-scheduler</groupId>
   <artifactId>quartz</artifactId>
   <version>2.3.2</version>
</dependency>
<dependency>
   <groupId>org.quartz-scheduler</groupId>
   <artifactId>quartz-jdbc-jobstore</artifactId>
   <version>2.3.2</version>
</dependency>
```

Skonfiguruj Quarza w `application.yml` lub `application.properties`:

```
spring.quartz.job-store-type=jdbc
spring.quartz.properties.org.quartz.jobStore.isClustered=true
spring.quartz.properties.org.quartz.jobStore.clusterCheckinInterval=20000
spring.quartz.properties.org.quartz.scheduler.instanceId=AUTO
```

spring.quartz.properties.org.quartz.scheduler.instanceName=ClusteredScheduler

Zadanie z dynamicznym planowaniem

1. Utwórz klasę zadania:

```
@DisallowConcurrentExecution
public class MyJob implements Job {
    @Override
    public void execute(JobExecutionContext context) {
        System.out.println("Executing job at: " + LocalDateTime.now());
        // Kod zadania (może być czasochłonny)
        try {
            Thread.sleep(120_000); // Symulacja długiego działania (2 min)
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Dynamiczne planowanie następnego uruchomienia
        JobDetail jobDetail = context.getJobDetail();
        Trigger trigger = TriggerBuilder.newTrigger()
            .forJob(jobDetail)
            .startAt(DateBuilder.futureDate(5, DateBuilder.IntervalUnit.MINUTE))
            .build();
        try {
            context.getScheduler().scheduleJob(trigger);
        } catch (SchedulerException e) {
            e.printStackTrace();
        }
    }
}
```

Zarejestruj zadanie w schedulerze:

```
@Configuration
public class QuartzConfig {
    @Bean
    public SchedulerFactoryBean schedulerFactoryBean() {
        return new SchedulerFactoryBean();
    }

    @Bean
    public JobDetail jobDetail() {
        return JobBuilder.newJob(MyJob.class)
            .withIdentity("myJob")
            .storeDurably()
            .build();
```

```
    }

    @Bean
    public Trigger trigger(JobDetail jobDetail) {
        return TriggerBuilder.newTrigger()
            .forJob(jobDetail)
            .startNow()
            .build();
    }
}

@Configuration
public class QuartzConfig {
    @Bean
    public SchedulerFactoryBean schedulerFactoryBean() {
        return new SchedulerFactoryBean();
    }

    @Bean
    public JobDetail jobDetail() {
        return JobBuilder.newJob(MyJob.class)
            .withIdentity("myJob")
            .storeDurably()
            .build();
    }

    @Bean
    public Trigger trigger(JobDetail jobDetail) {
        return TriggerBuilder.newTrigger()
            .forJob(jobDetail)
            .startNow()
            .build();
    }
}
```

Jak to działa?

1. **Jedna instancja uruchamia zadanie:** Quartz w trybie klastrowym gwarantuje, że zadanie uruchomi się tylko na jednym z węzłów.
2. **Dynamiczne wyzwalanie kolejnego uruchomienia:** Zadanie samo planuje swoje kolejne uruchomienie 5 minut po zakończeniu.
3. **Bez ograniczenia czasowego:** Nie ma sztucznego limitu czasu, w którym zadanie musi się zakończyć – harmonogram jest oparty na rzeczywistym zakończeniu.

Alternatywne podejście

Jeśli nie chcesz używać Quarza, możesz zrealizować podobny mechanizm przy użyciu:

- **Redis**: Użyj atomowych operacji na kluczach TTL (Time-To-Live).
- **Zewnętrznej kolejki zadań**: np. Kafka lub RabbitMQ do zarządzania kolejnymi wykonaniami.

1. Zależności do testów

Upewnij się, że masz dodane odpowiednie zależności w `pom.xml`:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-core</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.quartz-scheduler</groupId>
   <artifactId>quartz</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-inline</artifactId>
   <scope>test</scope>
</dependency>
```

2. Test zadania (`MyJob`)

Klasa testowa

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.quartz.*;
import static org.mockito.Mockito.*;

class MyJobTest {

   @Test
   void testJobExecutionAndReschedule() throws SchedulerException {
      // Mock Scheduler and JobExecutionContext
      Scheduler mockScheduler = mock(Scheduler.class);
      JobExecutionContext mockContext = mock(JobExecutionContext.class);
```

```
        JobDetail mockJobDetail = mock(JobDetail.class);
        when(mockContext.getJobDetail()).thenReturn(mockJobDetail);
        when(mockContext.getScheduler()).thenReturn(mockScheduler);

        // Execute the job
        MyJob job = new MyJob();
        job.execute(mockContext);

        // Verify rescheduling
        verify(mockScheduler, times(1)).scheduleJob(any(Trigger.class));
    }
}
```

Wyjaśnienie:

1. **Mockowanie Quarza:**
    - `Scheduler` i `JobExecutionContext` są mockowane, aby uniknąć faktycznego uruchamiania schedulerów i operacji na bazie danych.
2. **Sprawdzanie reschedule:**
    - Po wykonaniu zadania test weryfikuje, że scheduler wywołuje `scheduleJob()`.

3. Test konfiguracji Quarza

Klasa testowa

```
import org.junit.jupiter.api.Test;
import org.quartz.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
class QuartzConfigTest {

    @Autowired
    private Scheduler scheduler;

    @Autowired
    private JobDetail jobDetail;

    @Test
    void testSchedulerConfiguration() throws SchedulerException {
        // Verify that the scheduler is not null
        assertThat(scheduler).isNotNull();
```

```
        // Verify that the job is registered
        JobDetail registeredJob = scheduler.getJobDetail(JobKey.jobKey("myJob"));
        assertThat(registeredJob).isNotNull();
        assertThat(registeredJob.getJobClass()).isEqualTo(MyJob.class);
    }

    @Test
    void testInitialTrigger() throws SchedulerException {
        // Verify that the initial trigger exists
        Trigger trigger = scheduler.getTrigger(TriggerKey.triggerKey("trigger", "DEFAULT"));
        assertThat(trigger).isNotNull();
        assertThat(trigger.getJobKey()).isEqualTo(JobKey.jobKey("myJob"));
    }
}
```

Wyjaśnienie:

1. **Test integracyjny konfiguracji:**
    - Sprawdza, czy scheduler został poprawnie skonfigurowany i czy zadanie (`myJob`) jest zarejestrowane.
2. **Weryfikacja triggery:**
    - Upewnia się, że początkowy trigger został poprawnie utworzony.

4. Testy integracyjne z uruchamianiem zadania

Jeśli chcesz przetestować pełne działanie harmonogramu w środowisku testowym:

Klasa testowa

```
import org.junit.jupiter.api.Test;
import org.quartz.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.concurrent.TimeUnit;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
class QuartzIntegrationTest {

    @Autowired
    private Scheduler scheduler;

    @Test
    void testJobRunsAndReschedules() throws Exception {
        // Start the scheduler
        scheduler.start();
```

```
        // Wait for job execution (symulacja czasu pracy zadania)
        TimeUnit.SECONDS.sleep(10);

        // Verify job execution (sprawdzanie stanu w schedulerze)
        assertThat(scheduler.checkExists(JobKey.jobKey("myJob"))).isTrue();

        // Verify next trigger (sprawdzamy czy nowy trigger został zaplanowany)
        Trigger nextTrigger = scheduler.getTrigger(TriggerKey.triggerKey("trigger",
"DEFAULT"));
        assertThat(nextTrigger.getNextFireTime()).isNotNull();
    }
}
```

Wyniki i wnioski

- Testy sprawdzają zarówno konfigurację, jak i dynamiczne planowanie uruchomień.
- W przypadku testów integracyjnych Quarza należy uwzględnić **opóźnienia czasowe**, ponieważ zadania są uruchamiane w czasie rzeczywistym.
- Mockowanie sprawdza, czy logika działa bez faktycznego uruchamiania harmonogramu. Integracyjne testy weryfikują rzeczywiste działanie.

Czy chciałbyś, abym dostosował te testy do konkretnego przypadku? 😊

Test integracyjny z weryfikacją odstępu czasu
Klasa testowa

```
import org.junit.jupiter.api.Test;
import org.quartz.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.time.Instant;
import java.util.Date;
import java.util.concurrent.TimeUnit;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
class JobExecutionIntervalTest {

    @Autowired
    private Scheduler scheduler;

    @Test
    void testJobIntervalIsAtLeast30Seconds() throws Exception {
        // Start the scheduler
        scheduler.start();
```

```
        // Pierwszy czas uruchomienia
        Trigger firstTrigger = scheduler.getTrigger(TriggerKey.triggerKey("trigger", "DEFAULT"));
        Date firstFireTime = firstTrigger.getNextFireTime();

        // Poczekaj na uruchomienie pierwszego zadania
        TimeUnit.SECONDS.sleep(40); // Zakładamy, że pierwsze wykonanie trwa krócej niż 40 sekund

        // Pobranie kolejnego uruchomienia
        Trigger nextTrigger = scheduler.getTrigger(TriggerKey.triggerKey("trigger", "DEFAULT"));
        Date nextFireTime = nextTrigger.getNextFireTime();

        // Weryfikacja odstępu między uruchomieniami
        assertThat(nextFireTime).isNotNull();
        assertThat(nextFireTime.getTime() - firstFireTime.getTime())
            .isGreaterThanOrEqualTo(TimeUnit.SECONDS.toMillis(30));
    }
}
```

Wyjaśnienie kodu:

1. **Pobranie czasu pierwszego uruchomienia:**
    - `scheduler.getTrigger(TriggerKey.triggerKey("trigger", "DEFAULT"))` pozwala uzyskać szczegóły o kolejnym uruchomieniu.
    - `getNextFireTime()` zwraca czas planowanego uruchomienia zadania.
2. **Czekanie na uruchomienie:**
    - `TimeUnit.SECONDS.sleep(40)` pozwala symulować czas pracy zadania i upewnić się, że scheduler miał czas na zaplanowanie kolejnego uruchomienia.
3. **Weryfikacja czasu:**
    - Sprawdzamy różnicę między czasem kolejnego uruchomienia (`nextFireTime`) a czasem poprzedniego (`firstFireTime`).
    - Oczekujemy, że różnica wynosi co najmniej 30 sekund.

Uwagi:

- **Dynamiczne zaplanowanie następnego uruchomienia:** Kod zadania musi faktycznie ustawić odstęp 30 sekund lub więcej. W naszym przykładzie dynamiczne planowanie jest realizowane w klasie `MyJob` przez:

```
Trigger trigger = TriggerBuilder.newTrigger()
    .forJob(jobDetail)
    .startAt(DateBuilder.futureDate(5, DateBuilder.IntervalUnit.MINUTE)) // Tu zmień na 30 sekund, jeśli testujesz minimalny odstęp
```

```
    .build();


Trigger trigger = TriggerBuilder.newTrigger()
    .forJob(jobDetail)
    .startAt(DateBuilder.futureDate(5, DateBuilder.IntervalUnit.MINUTE)) // Tu zmień na 30 sekund, jeśli testujesz minimalny odstęp
    .build();
```

Czas trwania zadania: Jeśli zadanie trwa bardzo długo, trzeba odpowiednio dostosować czas oczekiwania w teście.

Rozszerzenie: Mockowanie czasu dla precyzyjnych testów

Jeśli czas jest krytyczny, a nie chcesz polegać na realnym zegarze, możesz użyć narzędzi takich jak **Mockito** lub **SystemClock** w Quarzu, aby symulować czas wykonania.

Jak to działa?

- Jeśli zadanie oznaczone tą adnotacją jest już w trakcie wykonywania, kolejne wywołanie (np. z powodu harmonogramu) zostanie **wstrzymane** do momentu zakończenia bieżącego uruchomienia.
- Ograniczenie to dotyczy tej samej instancji zadania (`JobDetail`), identyfikowanej po kluczu (`JobKey`).

Quartz osiąga to dzięki **JobStore**, gdzie przechowuje informacje o aktywnych instancjach i sprawdza ich status przed kolejnym uruchomieniem.

Przykład użycia

```
@DisallowConcurrentExecution
public class MyJob implements Job {
    @Override
    public void execute(JobExecutionContext context) {
        System.out.println("Executing job: " + context.getJobDetail().getKey());
        try {
            Thread.sleep(10000); // Symulacja długiego działania
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("Job finished: " + context.getJobDetail().getKey());
    }
}
```

Jeśli harmonogram wywoła to zadanie w odstępie mniejszym niż 10 sekund, kolejne uruchomienie zostanie opóźnione do zakończenia bieżącego.

Przykład bez `@DisallowConcurrentExecution`

Jeśli `@DisallowConcurrentExecution` NIE jest obecne, Quartz może uruchomić wiele instancji tego samego zadania równocześnie, co może prowadzić do problemów, takich jak:

- **konflikty dostępu do współdzielonych zasobów (np. plików lub baz danych),**
- **nieprzewidywalne zachowanie aplikacji.**

Przykład bez tej adnotacji:

Przykład bez `@DisallowConcurrentExecution`

Jeśli `@DisallowConcurrentExecution` NIE jest obecne, Quartz może uruchomić wiele instancji tego samego zadania równocześnie, co może prowadzić do problemów, takich jak:

- **konflikty dostępu do współdzielonych zasobów (np. plików lub baz danych),**
- **nieprzewidywalne zachowanie aplikacji.**

Przykład bez tej adnotacji:

public class MyJob implements Job { @Override public void execute(JobExecutionContext context) { System.out.println("Concurrent job execution: " + context.getFireTime()); try { Thread.sleep(10000); // Symulacja długiego działania } catch (InterruptedException e) { Thread.currentThread().interrupt(); } } }

Jeśli harmonogram uruchamia zadanie co 5 sekund, będziesz mieć równoczesne instancje wykonujące kod.

Kiedy używać `@DisallowConcurrentExecution`?

- **Gdy zadanie:**
  - **modyfikuje współdzielone zasoby (np. pliki, bazy danych),**
  - **działa na długich procesach wymagających synchronizacji,**
  - **powinno działać sekwencyjnie.**
- **Przykłady:**
  - **Generowanie raportów,**
  - **Przetwarzanie dużych plików.**

Ograniczenia

- **Nie dotyczy wielu `JobDetail`: `@DisallowConcurrentExecution` dotyczy tylko tej samej instancji zadania (klucza `JobKey`). Jeśli utworzysz dwa różne `JobDetail` dla tej samej klasy zadania, mogą działać równocześnie.**
- **Nie zatrzymuje nowych instancji: Jeśli harmonogram wywołuje zadanie często, instancje będą się kumulować w kolejce, jeśli czas ich wykonywania przekracza interwał.**

Podsumowanie

- **`@DisallowConcurrentExecution` gwarantuje, że tylko jedna instancja zadania będzie działać w danym momencie.**
- **Jest szczególnie przydatna w środowiskach wymagających synchronizacji i sekwencyjnego przetwarzania.**

Jeśli masz pytania lub potrzebujesz dodatkowego przykładu, daj znać! 😊