

Quarkus - Scheduler Reference Guide

Modern applications often need to run specific tasks periodically. There are two scheduler extensions in Quarkus. The `quarkus-scheduler` extension brings the API and a lightweight in-memory scheduler implementation. The `quarkus-quartz` extension implements the API from the `quarkus-scheduler` extension and contains a scheduler implementation based on the Quartz library. You will only need `quarkus-quartz` for more advanced scheduling use cases, such as persistent tasks, clustering and programmatic scheduling of jobs.



If you add the `quarkus-quartz` dependency to your project the lightweight scheduler implementation from the `quarkus-scheduler` extension is automatically disabled.

1. Scheduled Methods

If you annotate a method with `@io.quarkus.scheduler.Scheduled` it is automatically scheduled for invocation. In fact, such a method must be a non-private non-static method of a CDI bean. As a consequence of being a method of a CDI bean a scheduled method can be annotated with interceptor bindings, such as `@javax.transaction.Transactional` and `@org.eclipse.microprofile.metrics.annotation.Counted`.



If there is no CDI scope defined on the declaring class then `@Singleton` is used.

Furthermore, the annotated method must return `void` and either declare no parameters or one parameter of type `io.quarkus.scheduler.ScheduledExecution`.



The annotation is repeatable so a single method could be scheduled multiple times.

1.1. Triggers

A trigger is defined either by the `@Scheduled#cron()` or by the `@Scheduled#every()` attributes. If both are specified, the cron expression takes precedence. If none is specified, the build fails with an `IllegalStateException`.

1.1.1. CRON

A CRON trigger is defined by a cron-like expression. For example `"0 15 10 * * ?"` fires at 10:15am every day.

CRON Trigger Example

```
@Scheduled(cron = "0 15 10 * * ?")
void fireAt10AmEveryDay() { }
```

The syntax used in CRON expressions is controlled by the `quarkus.scheduler.cron-type` property. The values can be `cron4j`, `quartz`, `unix` and `spring.quartz` is used by default.

If a CRON expression starts with `{` and ends with `}` then the scheduler attempts to find a corresponding config property and use the configured value instead.

CRON Config Property Example

```
@Scheduled(cron = "{myMethod.cron.expr}")
void myMethod() { }
```

1.1.2. Intervals

An interval trigger defines a period between invocations. The period expression is based on the ISO-8601 duration format `PnDTnHnMn.nS` and the value of `@Scheduled#every()` is parsed with `java.time.Duration#parse(CharSequence)`. However, if an expression starts with a digit then the `PT` prefix is added automatically. So for example, `15m` can be used instead of `PT15M` and is parsed as "15 minutes".

Interval Trigger Example

```
@Scheduled(every = "15m")
void every15Mins() { }
```

If a value starts with `{` and ends with `}` then the scheduler attempts to find a corresponding config property and use the configured value instead.

Interval Config Property Example

```
@Scheduled(every = "{myMethod.every.expr}")
void myMethod() { }
```

1.2. Identity

By default, a unique id is generated for each scheduled method. This id is used in log messages and during debugging. Sometimes a possibility to specify an explicit id may come in handy.

Identity Example

```
@Scheduled(identity = "myScheduledMethod")
void myMethod() { }
```

1.3. Delayed Execution

`@Scheduled` provides two ways to delay the time a trigger should start firing at.

`@Scheduled#delay()` and `@Scheduled#delayUnit()` form the initial delay together.

```
@Scheduled(every = "2s", delay = 2, delayUnit = TimeUnit.HOUR) ①  
void everyTwoSeconds() { }
```

① The trigger fires for the first time two hours after the application start.



The final value is always rounded to full second.

`@Scheduled#delayed()` is a text alternative to the properties above. The period expression is based on the ISO-8601 duration format `PnDTnHnMn.nS` and the value is parsed with `java.time.Duration#parse(CharSequence)`. However, if an expression starts with a digit, the `PT` prefix is added automatically. So for example, `15s` can be used instead of `PT15S` and is parsed as "15 seconds".

```
@Scheduled(every = "2s", delayed = "2h")  
void everyTwoSeconds() { }
```



If `@Scheduled#delay()` is set to a value greater than zero the value of `@Scheduled#delayed()` is ignored.

The main advantage over `@Scheduled#delay()` is that the value is configurable. If the value starts with `{` and ends with `}` then the scheduler attempts to find a corresponding config property and use the configured value instead:

```
@Scheduled(every = "2s", delayed = "{myMethod.delay.expr}") ①  
void everyTwoSeconds() { }
```

① The config property `myMethod.delay.expr` is used to set the delay.

1.4. Concurrent Execution

By default, a scheduled method can be executed concurrently. Nevertheless, it is possible to specify the strategy to handle concurrent executions via `@Scheduled#concurrentExecution()`.

```
import static  
io.quarkus.scheduler.Scheduled.ConcurrentExecution.SKIP;  
  
@Scheduled(every = "1s", concurrentExecution = SKIP) ①  
void nonConcurrent() {  
    // we can be sure that this method is never executed concurrently  
}
```

① Concurrent executions are skipped.

2. Scheduler

Quarkus provides a built-in bean of type `io.quarkus.scheduler.Scheduler` that can be injected and used to pause/resume the scheduler.

Scheduler Injection Example

```
import io.quarkus.scheduler.Scheduler;

class MyService {

    @Inject
    Scheduler scheduler;

    void ping() {
        scheduler.pause(); ❶
        if (scheduler.isRunning()) {
            throw new IllegalStateException("This should never
happen!");
        }
        scheduler.resume(); ❷
    }
}
```

❶ Pause all triggers.

❷ Resume the scheduler.

3. Programmatic Scheduling

If you need to schedule a job programmatically you'll need to add the [Quartz extension](#) and use the Quartz API directly.

```
import org.quartz.Scheduler;

class MyJobs {

    void onStart(@Observes StartupEvent event, Scheduler quartz)
    throws SchedulerException {
        JobDetail job = JobBuilder.newJob(SomeJob.class)
            .withIdentity("myJob", "myGroup")
            .build();
        Trigger trigger = TriggerBuilder.newTrigger()
            .withIdentity("myTrigger", "myGroup")
            .startNow()

        .withSchedule(SimpleScheduleBuilder.simpleSchedule()
            .withIntervalInSeconds(1)
            .repeatForever())
            .build();
        quartz.scheduleJob(job, trigger);
    }
}
```



By default, the scheduler is not started unless a `@Scheduled` business method is found. You may need to force the start of the scheduler for "pure" programmatic scheduling. See also [Quartz Configuration Reference](#).

4. Scheduled Methods and Testing

It is often desirable to disable the scheduler when running the tests. The scheduler can be disabled through the runtime config property `quarkus.scheduler.enabled`. If set to `false` the scheduler is not started even though the application contains scheduled methods. You can even disable the scheduler for particular [Test Profiles](#).

5. Configuration Reference

Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
<code>quarkus.scheduler.cron-type</code> The syntax used in CRON expressions.	<code>cron4j</code> , <code>quartz</code> , <code>unix</code> , <code>spring</code>	<code>quartz</code>

<code>quarkus.scheduler.enabled</code> If schedulers are enabled.	boolean	<code>true</code>
--	---------	-------------------