

Quarkus - Scheduling Periodic Tasks with Quartz

Modern applications often need to run specific tasks periodically. In this guide, you learn how to schedule periodic clustered tasks using the [Quartz](#) extension.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).



If you only need to run in-memory scheduler use the [Scheduler](#) extension.

Prerequisites

To complete this guide, you need:

- less than 10 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- Docker and Docker Compose installed on your machine

Architecture

In this guide, we are going to expose one Rest API `tasks` to visualise the list of tasks created by a Quartz job running every 10 seconds.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `quartz-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.1.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=quartz-quickstart \
  -DclassName="org.acme.quartz.TaskResource" \
  -Dpath="/tasks" \
  -Dextensions="quartz, hibernate-orm-panache, flyway, resteasy-
jsonb, jdbc-postgresql"
cd quartz-quickstart
```

It generates:

- the Maven structure
- a landing page accessible on <http://localhost:8080>
- example **Dockerfile** files for both **native** and **jvm** modes
- the application configuration file
- an **org.acme.quartz.TaskResource** resource
- an associated test

The Maven project also imports the Quarkus Quartz extension.

If you already have your Quarkus project configured, you can add the **quartz** extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="quartz"
```

This will add the following to your **pom.xml**:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-quartz</artifactId>
</dependency>
```



To use a JDBC store, the **quarkus-agroal** extension, which provides the datasource support, is also required.

Creating the Task Entity

In the `org.acme.quartz` package, create the `Task` class, with the following content:

```
package org.acme.quartz;

import javax.persistence.Entity;
import java.time.Instant;
import javax.persistence.Table;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
@Table(name="TASKS")
public class Task extends PanacheEntity { ①
    public Instant createdAt;

    public Task() {
        createdAt = Instant.now();
    }

    public Task(Instant time) {
        this.createdAt = time;
    }
}
```

1. Declare the entity using [Panache](#)

Creating a scheduled job

In the `org.acme.quartz` package, create the `TaskBean` class, with the following content:

```

package org.acme.quartz;

import javax.enterprise.context.ApplicationScoped;
import javax.transaction.Transactional;
import io.quarkus.scheduler.Scheduled;

@ApplicationScoped ①
public class TaskBean {

    @Transactional
    @Scheduled(every = "10s") ②
    void schedule() {
        Task task = new Task(); ③
        task.persist(); ④
    }
}

```

1. Declare the bean in the *application* scope
2. Use the `@Scheduled` annotation to instruct Quarkus to run this method every 10 seconds.
3. Create a new `Task` with the current start time.
4. Persist the task in database using [Panache](#).

Scheduling Jobs Programmatically

It is also possible to leverage the Quartz API directly. You can inject the underlying `org.quartz.Scheduler` in any bean:

```

package org.acme.quartz;

@ApplicationScoped
public class TaskBean {

    @Inject
    org.quartz.Scheduler quartz; ①

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

SimpleScheduleBuilder.simpleSchedule()
                .withIntervalInSeconds(10)
                .repeatForever()
                .build();
        quartz.scheduleJob(job, trigger); ②
    }

    @Transactional
    void performTask() {
        Task task = new Task();
        task.persist();
    }

    // A new instance of MyJob is created by Quartz for every job
    execution
    public static class MyJob implements Job {

        @Inject
        TaskBean taskBean;

        public void execute(JobExecutionContext context) throws
JobExecutionException {
            taskBean.performTask(); ③
        }

    }
}

```

1. Inject the underlying `org.quartz.Scheduler` instance.

2. Schedule a new job using the Quartz API.
3. Invoke the `TaskBean#performTask()` method from the job. Jobs are also [container-managed](#) beans if they belong to a [bean archive](#).



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](#).

Updating the application configuration file

Edit the `application.properties` file and add the below configuration:

```
# Quartz configuration
quarkus.quartz.clustered=true ①
quarkus.quartz.store-type=jdbc-cmt ②

# Datasource configuration.
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=quarkus_test
quarkus.datasource.password=quarkus_test
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost/quarkus_test

# Hibernate configuration
quarkus.hibernate-orm.database.generation=none
quarkus.hibernate-orm.log.sql=true
quarkus.hibernate-orm.sql-load-script=no-file

# flyway configuration
quarkus.flyway.connect-retries=10
quarkus.flyway.table=flyway_quarkus_history
quarkus.flyway.migrate-at-start=true
quarkus.flyway.baseline-on-migrate=true
quarkus.flyway.baseline-version=1.0
quarkus.flyway.baseline-description=Quartz
```

1. Indicate that the scheduler will be run in clustered mode
2. Use the database store to persist job related information so that they can be shared between nodes

Updating the resource and the test

Edit the `TaskResource` class, and update the content to:

```
package org.acme.quartz;

import java.util.List;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/tasks")
@Produces(MediaType.APPLICATION_JSON)
public class TaskResource {

    @GET
    public List<Task> listAll() {
        return Task.listAll(); ①
    }
}
```

1. Retrieve the list of created tasks from the database

We also need to update the tests. Edit the `TaskResourceTest` class to match:

```

package org.acme.quartz;

import io.quarkus.test.junit.QuarkusTest;

import static org.hamcrest.Matchers.greaterThanOrEqualTo;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class TaskResourceTest {

    @Test
    public void tasks() throws InterruptedException {
        Thread.sleep(1000); // wait at least a second to have the
        first task created
        given()
            .when().get("/tasks")
            .then()
            .statusCode(200)
            .body("size()", is(greaterThanOrEqualTo(1))); ①
    }
}

```

1. Ensure that we have a **200** response and at least one task created

Creating Quartz Tables

Add a SQL migration file named `src/main/resources/db/migration/V2.0.0__QuarkusQuartzTasks.sql` with the content copied from file with the content from `V2.0.0__QuarkusQuartzTasks.sql`.

Configuring the load balancer

In the root directory, create a `nginx.conf` file with the following content:


```
user  nginx;

events {
    worker_connections  1000;
}

http {
    server {
        listen 8080;
        location / {
            proxy_pass http://tasks:8080; ①
        }
    }
}
```

1. Route all traffic to our tasks application

Setting Application Deployment

In the root directory, create a `docker-compose.yml` file with the following content:

```

version: '3'

services:
  tasks: ❶
    image: quarkus-quickstarts/quartz:1.0
    build:
      context: ./
      dockerfile: src/main/docker/Dockerfile.${QUARKUS_MODE:-jvm}
    environment:
      QUARKUS_DATASOURCE_URL:
jdbc:postgresql://postgres/quarkus_test
    networks:
      - tasks-network
    depends_on:
      - postgres

  nginx: ❷
    image: nginx:1.17.6
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - tasks
    ports:
      - 8080:8080
    networks:
      - tasks-network

  postgres: ❸
    image: postgres:11.3
    container_name: quarkus_test
    environment:
      - POSTGRES_USER=quarkus_test
      - POSTGRES_PASSWORD=quarkus_test
      - POSTGRES_DB=quarkus_test
    ports:
      - 5432:5432
    networks:
      - tasks-network

networks:
  tasks-network:
    driver: bridge

```

1. Define the tasks service
2. Define the nginx load balancer to route incoming traffic to an appropriate node
3. Define the configuration to run the database

Running the database

In a separate terminal, run the below command:

```
docker-compose up postgres ①
```

1. Start the database instance using the configuration options supplied in the `docker-compose.yml` file

Run the application in Dev Mode

Run the application with: `./mvnw quarkus:dev`. After a few seconds, open another terminal and run `curl localhost:8080/tasks` to verify that we have at least one task created.

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file. You can also generate the native executable with `./mvnw clean package -Pnative`.

Packaging the application and run several instances

The application can be packaged using `./mvnw clean package`. Once the build is successful, run the below command:

```
docker-compose up --scale tasks=2 --scale nginx=1 ①
```

1. Start two instances of the application and a load balancer

After a few seconds, in another terminal, run `curl localhost:8080/tasks` to verify that tasks were only created at different instants and in an interval of 10 seconds.

You can also generate the native executable with `./mvnw clean package -Pnative`.

Registering Plugin and Listeners

You can register `plugins`, `job-listeners` and `trigger-listeners` through Quarkus configuration.


The example below registers the plugin `org.quartz.plugins.history.LoggingJobHistoryPlugin` named as `jobHistory` with the property `jobSuccessMessage` defined as `Job [{1}]{0}` execution complete and `reports: {8}`





```
quarkus.quartz.plugins.jobHistory.class=org.quartz.plugins.history.  
LoggingJobHistoryPlugin  
quarkus.quartz.plugins.jobHistory.properties.jobSuccessMessage=Job  
[{1}]{0}] execution complete and reports: {8}
```


You can also register a listener programmatically with an injected `org.quartz.Scheduler`:

```
public class MyListenerManager {  
    void onStart(@Observes StartupEvent event, org.quartz.Scheduler  
scheduler) throws SchedulerException {  
        scheduler.getListenerManager().addJobListener(new  
MyJogListener());  
        scheduler.getListenerManager().addTriggerListener(new  
MyTriggerListener());  
    }  
}
```

Quartz Configuration Reference

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

Configuration property	Type	Default
 <code>quarkus.quartz.clustered</code> Enable cluster mode or not. If enabled make sure to set the appropriate cluster properties.	boolean	<code>false</code>
 <code>quarkus.quartz.store-type</code> The type of store to use. When using the <code>db</code> store type configuration value make sure that you have the datasource configured. See Configuring your datasource for more information. To create Quartz tables, you can perform a schema migration via the Flyway extension using a SQL script matching your database picked from Quartz repository .	<code>ram</code> , <code>jdbc-tx</code> , <code>jdbc-cmt,db</code>	<code>ram</code>
 <code>quarkus.quartz.datasource</code> The name of the datasource to use. Optionally needed when using the <code>db</code> store type. If not specified, defaults to using the default datasource.	string	
 <code>quarkus.quartz.table-prefix</code> The prefix for quartz job store tables. Ignored if using a <code>ram</code> store.	string	<code>QRTZ_</code>

<code>quarkus.quartz.instance-name</code>		
The name of the Quartz instance.	string	QuarkusQuartzScheduler
<code>quarkus.quartz.thread-count</code>		
The size of scheduler thread pool. This will initialize the number of worker threads in the pool.	int	25
<code>quarkus.quartz.thread-priority</code>		
Thread priority of worker threads in the pool.	int	5
<code>quarkus.quartz.force-start</code>		
By default, the scheduler is not started unless a <code>io.quarkus.scheduler.Scheduled</code> business method is found. If set to true the scheduler will be started even if no scheduled business methods are found. This is necessary for "pure" programmatic scheduling.	boolean	false
Trigger listeners	Type	Default
 <code>quarkus.quartz.trigger-listeners."listener-name".class</code>	string	required 
 <code>quarkus.quartz.trigger-listeners."listener-name".properties</code>	Map<String, String>	
Job listeners	Type	Default
 <code>quarkus.quartz.job-listeners."listener-name".class</code>	string	required 
 <code>quarkus.quartz.job-listeners."listener-name".properties</code>	Map<String, String>	
Plugins	Type	Default
 <code>quarkus.quartz.plugins."plugin-name".class</code>	string	required 
 <code>quarkus.quartz.plugins."plugin-name".properties</code>	Map<String, String>	