



[ANDROID](#)
[CORE JAVA](#)
[DESKTOP JAVA](#)
[ENTERPRISE JAVA](#)
[JAVA BASICS](#)
[JVM LANGUAGES](#)
[SOFTWARE DEVELOPMENT](#)
[DEVOPS](#)

[Home](#) » [Enterprise Java](#) » [Quartz](#) » [Java Quartz Architecture Example](#)

## ABOUT LEFTERIS KARAGEORGIOU



Lefteris is a Lead Software Engineer at ZuluTrade and has been responsible for re-architecting the backend of the main website from a monolith to event-driven microservices using Java, Spring Boot/Cloud, RabbitMQ, Redis. He has extensive work experience for over 10 years in Software Development, working mainly in the FinTech and Sports Betting industries. Prior to joining ZuluTrade, Lefteris worked as a Senior Java Developer at Inspired Gaming Group in London, building enterprise sports betting applications for William Hills and Paddy Power. He enjoys working with large-scalable, real-time and high-volume systems deployed into AWS and wants to combine his passion for technology and traveling by attending software conferences all over the world.



## Java Quartz Architecture Example

Publicado por: Lefteris Karageorgiou
 en Cuarzo
 4 de junio de 2019
 0
 578 Vistas

### 1. Introducción

En esta publicación, analizaremos más de cerca la arquitectura de Quartz, una biblioteca de programación de trabajos de código abierto muy popular que se puede usar en aplicaciones Java. Veremos un diagrama arquitectónico y aprenderemos todos los componentes principales y opcionales de Quartz proporcionando ejemplos de código.

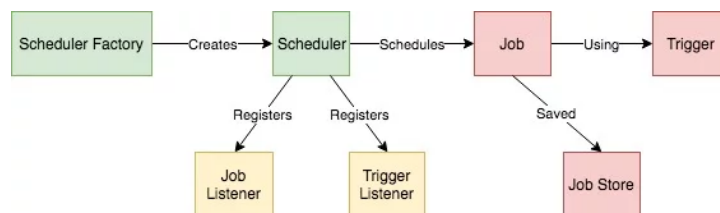
### 2. Configuración del proyecto

Para ejecutar los ejemplos de código de esta publicación, usaremos las siguientes tecnologías:

- Java 8
- Cuarzo 2.2.1
- SLF4J 1.7.26
- Logback 1.2.3
- Maven 3.3.3
- Eclipse 4.10.0

### 3. Arquitectura de cuarzo

En pocas palabras, el concepto principal de Quartz es que un **programador** mantiene una lista de **trabajos**, en un almacén de persistencia, que se **activan** en momentos específicos o repetidamente. También puede registrar **trabajos** o **activar detectores** en el programador, que realiza algunas operaciones antes y después de la finalización de trabajos o activadores. El siguiente diagrama muestra el proceso de programación de trabajos en una aplicación de cuarzo.



#### HOJA INFORMATIVA

**i169,222** insiders ya est  
 de actualizaciones semanales  
**blancos de** cortesía!  
**Únase a ellos ahor**  
**acceso exclusivo** a las  
 en el mundo de Java, así com  
 sobre Android, Scala, Groovy  
 tecnologías relacionadas.

#### Dirección de correo electrónico:

Your email address

☒ Reciba alertas de empleo  
 desarrolladores en su área

Regístrate

#### ÚNETE A NOSOTROS

Con **1,240,600** visitantes únic  
 más de **500** autores, estamos



## 4.1 Programador

El

```
org.quartz.Scheduler
```

es la interfaz principal de un *programador de cuarzo*. Un programador mantiene un registro de *JobDetails* y *Triggers*. Una vez registrado, el *programador* es responsable de ejecutar los *trabajos* cuando se *activan* los *disparadores* asociados cuando llega su hora programada.

## 4.2 Fábrica de programadores

La

```
org.quartz.SchedulerFactory
```

interfaz es la responsable de crear instancias de *Scheduler*. Cualquier clase que implemente esta interfaz debe implementar los siguientes métodos:

- `Scheduler getScheduler() throws SchedulerException`
- `Scheduler getScheduler(String schedName) throws SchedulerException`
- `Collection<Scheduler> getAllSchedulers() throws SchedulerException`

Los dos primeros métodos devuelven una instancia de *Scheduler* con el nombre predeterminado o un nombre dado. El tercer método devuelve todos los *programadores* conocidos.

Hay dos implementaciones de *SchedulerFactory*:

- **StdSchedulerFactory** -  
`org.quartz.impl.StdSchedulerFactory`
- **DirectSchedulerFactory** -  
`org.quartz.impl.DirectSchedulerFactory`

*StdSchedulerFactory* crea una instancia de *Scheduler* basada en el contenido de un archivo de propiedades que se llama por defecto **quartz.properties** y se carga desde el directorio de trabajo actual. Por otro lado, *DirectSchedulerFactory* es una implementación más simple de *SchedulerFactory* y también es un singleton.

A continuación, encontrará un ejemplo de cómo crear una instancia de *Scheduler*:

```
1 SchedulerFactory schedulerFactory = new StdSchedulerFactory();
2 Scheduler scheduler = schedulerFactory.getScheduler();
```

A partir del código anterior, se crea una nueva instancia de *StdSchedulerFactory* que devuelve una instancia de *Scheduler* llamando al

```
getScheduler()
```

método.

## Trabajo 4.3

El

```
org.quartz.Job
```

es la interfaz más crucial para ser implementado por las clases, ya que representa un trabajo a realizar. A continuación vemos un ejemplo de una clase que implementa esta interfaz:

```
1 public class SimpleJob implements Job {
2
3     private final Logger log = LoggerFactory.getLogger(SimpleJob.class);
4
5     public void execute(JobExecutionContext context) throws JobExecutionException {
6         log.info("SimpleJob executed!");
7     }
8 }
```

Del código anterior, vemos que la

```
SimpleJob
```

clase implementa el

El

```
org.quartz.JobDetail
```

transmite las propiedades de detalle de una instancia de trabajo dada. Quartz no almacena una instancia real de una clase de Trabajo, sino que le permite definir una instancia de una, mediante el uso de un *JobDetail*. Veamos cómo se hace esto:

```
1 | JobDetail job = JobBuilder.newJob(SimpleJob.class)
2 |     .withIdentity("myJob", "myGroup")
3 |     .build();
```

En el ejemplo anterior, definimos un nuevo trabajo y lo vinculamos a la

```
SimpleJob
```

clase que creamos anteriormente. Tenga en cuenta que *JobDetails* se crean utilizando la

```
org.quartz.JobBuilder
```

clase.

## 4.5 gatillo

El

```
org.quartz.Trigger
```

es el interfaz base con propiedades comunes a todos los disparadores. *Los disparadores* son el mecanismo por el cual se programan los *trabajos*. Muchos *disparadores* pueden apuntar al mismo trabajo, pero un solo *disparador* solo puede apuntar a un solo trabajo. El

```
org.quartz.TriggerBuilder
```

se utiliza para instanciar *Triggers*.

Hay varias implementaciones de *Trigger*. Los más utilizados son:

- Simple Trigger -

```
org.quartz.SimpleTrigger
```

- CronTrigger -

```
org.quartz.CronTrigger
```

El *SimpleTrigger* se usa para disparar un *trabajo* en un momento dado en el tiempo, y opcionalmente se repite en un intervalo específico. El *CronTrigger* se usa para disparar un *trabajo* en determinados momentos, definidos con definiciones de cronograma similares a Unix.

El siguiente ejemplo muestra cómo crear un *Disparador* utilizando *TriggerBuilder*:

```
1 | Trigger trigger = TriggerBuilder.newTrigger()
2 |     .withIdentity("myTrigger", "myGroup")
3 |     .withSchedule(SimpleScheduleBuilder.simpleSchedule()
4 |         .withIntervalInSeconds(3)
5 |         .repeatForever())
6 |     .build();
```

En el código anterior, usamos la clase de ayuda *TriggerBuilder* para crear un disparador que se ejecuta cada 3 segundos de forma indefinida. Hasta ahora no hemos atado ningún *trabajo* al *gatillo*. Esto es lo que hace el *Programador*.

## 4.6 Programador de cuarzo

El

```
org.quartz.core.QuartzScheduler
```

es el corazón de cuarzo, una aplicación indirecta del *programador* de interfaz, que contiene métodos para programar *los trabajos* usando *disparadores*. El código anterior programa un *trabajo* utilizando la instancia de *Scheduler* que creamos en una sección anterior:

```
1 | scheduler.scheduleJob(job, trigger);
```

Del código anterior, vemos que no pasamos un *Trabajo* al *Programador*, sino un *JobDetail*, en el que *vinculamos* el *Trabajo*. También pasamos un *Desencadenador* que programa el *Trabajo* para que se ejecute en momentos específicos. Finalmente para iniciar la llamada del *programador*:

```
1 | scheduler.start();
```

Y para apagar el *Programador*:

```
1 | scheduler.shutdown(boolean waitForJobsToComplete);
```

## 5.1 Job Store

El

```
org.quartz.spi.JobStore
```

es la interfaz a ser implementado por las clases que quieren ofrecer un *empleo* y *de activación* mecanismo de almacenamiento para el *QuartzScheduler* uso 's. Hay dos implementaciones de la interfaz *JobStore*:

- *RAMJobStore* -

```
org.quartz.simpl.RAMJobStore
```

- *JobStoreSupport* -

```
org.quartz.impl.jdbcjobstore.JobStoreSupport
```

El *RAMJobStore* es el *JobStore* predeterminado que utiliza RAM como dispositivo de almacenamiento. La ramificación de esto es que el acceso es extremadamente rápido, pero los datos son completamente volátiles; por lo tanto, este *JobStore* no debe usarse si se requiere una verdadera persistencia entre los cierres de programas. El *JobStoreSupport* contiene funcionalidad de base para basados en JDBC *JobStore* implementaciones.

Puede habilitar *JobStoreSupport* utilizando JDBC, a través del archivo de propiedades de Quartz:

```
1  org.quartz.jobStore.class=org.quartz.impl.jdbcjobstore.JobStoreTX
2  org.quartz.jobStore.driverDelegateClass=org.quartz.impl.jdbcjobstore.StdJDBCDelegate
3  org.quartz.jobStore.dataSource=quartzDataSource
4  org.quartz.jobStore.tablePrefix=QRTZ_
5  org.quartz.threadPool.class=org.quartz.simpl.SimpleThreadPool
6  org.quartz.dataSource.quartzDataSource.driver=com.mysql.jdbc.Driver
7  org.quartz.dataSource.quartzDataSource.URL=jdbc:mysql://localhost:3306/quartz_schema
8  org.quartz.dataSource.quartzDataSource.user=root
9  org.quartz.dataSource.quartzDataSource.password=change_me
```

De lo anterior, la clase *JobStoreTX* (que extiende la clase *JobStoreSupport*) se utiliza como *JobStore*. Para obtener una explicación más detallada de cómo usar Quartz con JDBC, consulte aquí.

## 5.2 Escucha de trabajo

El

```
org.quartz.JobListener
```

es la interfaz a ser implementado por las clases que desea ser informado cuando un *JobDetail* ejecuta. Los detectores de trabajos se adjuntan al programador y tienen métodos que se llaman antes y después de la ejecución de los trabajos. En el siguiente ejemplo creamos una nueva clase *JobListener*:

```
01  public class MyJobListener implements JobListener {
02
03      private final Logger log = LoggerFactory.getLogger(MyJobListener.class);
04
05      public String getName() {
06          return MyJobListener.class.getSimpleName();
07      }
08
09      public void jobToBeExecuted(JobExecutionContext context) {
10          log.info("{} is about to be executed", context.getJobDetail().getKey().toString());
11      }
12
13      public void jobWasExecuted(JobExecutionContext context, JobExecutionException jobException) {
14          log.info("{} finished execution", context.getJobDetail().getKey().toString());
15      }
dieciséis
17      public void jobExecutionVetoed(JobExecutionContext context) {
18          log.info("{} was about to be executed but a JobListener vetoed it's execution", context.getJobDetail().getKey().toString());
19      }
20  }
```

Desde el *JobListener* que creamos anteriormente, el orden de los métodos que se ejecutarán es:

```
MyJobListener.jobToBeExecuted()
```

```
-->
```

```
MyJob.execute()
```

```
-->
```

```
MyJobListener.jobWasExecuted()
```

Finalmente registramos el *MyJobListener* en el *Programador*:

creamos una nueva clase *TriggerListener*:

```

01 public class MyTriggerListener implements TriggerListener {
02
03     private final Logger log = LoggerFactory.getLogger(MyTriggerListener.class);
04
05     @Override
06     public String getName() {
07         return MyTriggerListener.class.getSimpleName();
08     }
09
10     @Override
11     public void triggerFired(Trigger trigger, JobExecutionContext context) {
12         log.info("{} trigger is fired", getName());
13     }
14
15     @Override
dieciséis     public boolean vetoJobExecution(Trigger trigger, JobExecutionContext context) {
17         log.info("{} was about to be executed but a TriggerListener vetoed it's execution", context.g
18         return false;
19     }
20
21     @Override
22     public void triggerMisfired(Trigger trigger) {
23         log.info("{} trigger was misfired", getName());
24     }
25
26     @Override
27     public void triggerComplete(Trigger trigger, JobExecutionContext context,
28         CompletedExecutionInstruction triggerInstructionCode) {
29         log.info("{} trigger is complete", getName());
30     }
31 }

```

Desde el *TriggerListener* que creamos anteriormente, el orden de los métodos que se ejecutarán es:

MyTriggerListener.triggerFired()

->

MyJob.execute()

->

MyJobListener.

triggerComplete

○

Finalmente registramos el *MyTriggerListener* en el *Programador*:

```
1 scheduler.getListenerManager().addTriggerListener(new MyTriggerListener());
```

## 6. Java Quartz Architecture - Conclusión




En este post, examinamos con más detalle la arquitectura de Cuarzo. Vimos cómo funciona la programación de trabajos al proporcionar un diagrama arquitectónico. También observamos de cerca los componentes principales y opcionales de Quartz, como el *Programador*, *Trabajo*, *Disparador*, etc.

## 7. Descarga el proyecto Eclipse.

### Descargar

Puede descargar el código fuente completo de los ejemplos anteriores aquí: [Ejemplo de Java Quartz Architecture](#)

Etiquetado con: CUARZO

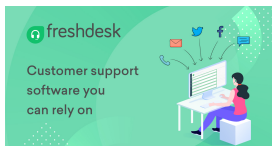
 (+3 rating, 3 votos)  Iniciar la discusión  578 Vistas  Tweet it!

¿Quieres saber cómo desarrollar tu conjunto de habilidades para convertirte en un **Java Rockstar**?

Suscríbete a nuestro boletín para comenzar Rocking ahora mismo!

Para comenzar, te regalamos nuestros libros electrónicos más vendidos

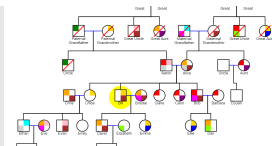
**GRATIS!**

**Dirección de correo electrónico:**☒ Reciba alertas de empleo de Java y desarrolladores en su área[Regístrate](#)**¿TE GUSTA ESTE ARTÍCULO? LEER MÁS DE JAVA CODE GEEKS****Customer Support Software**

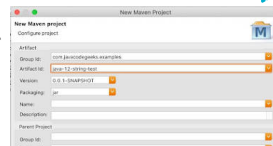
Ad Freshdesk

**Docker Install on Ubuntu Example**

javacodegeeks.com

**JavaScript Flowcharts**

Ad GoJS

**Java 12 String Methods Example**

javacodegeeks.com

**API Collaboration Simplified**

Ad Postman

**Java Nio Heartbeat Example**

javacodegeeks.com

**Build your First Android App using Android Studio**

javacodegeeks.com

**Comments in**

javacodegeeks.com

**Deja una respuesta**

Start the discussion...

☒ Suscribirse ▼

**Juega como un Founder.**  
**129 €**

**BASE DE CONOCIMIENTOS**[Los cursos](#)**HALL OF FAME**[Android Alert Dialog Example](#)**ABOUT JAVA CODE GEEKS**[JCGs \(Java Code Geeks\) is an independent online community focused on](#)



LA RED DE CODE GEEKS	solve File Not Found Exception	Oracle Corporation in the United States and other countries. Examples is not connected to Oracle Corporation and is not sponsored by Oracle (
Código .NET Geeks	java.lang.arrayindexoutofboundsexception – How to handle Array Index Out Of Bounds Exception	
Java Code Geeks	java.lang.NoClassDefFoundError – How to solve No Class Def Found Error	
System Code Geeks	JSON Example With Jersey + Jackson	
Web Code Geeks	Spring JdbcTemplate Example	

