Adds Quartz job scheduling features

Quartz plugin for Grails - Reference Documentation

Authors: Sergey Nebolsin, Graeme Rocher, Ryan Vanderwerf, Vitalii Samolovskikh
Version: 1.0.2

Table of Contents

1 Introduction

Quartz plugin allows your Grails application to schedule jobs to be executed using a specified interval or cron expression. The underlying system uses the Quartz Enterprise Job Scheduler configured via Spring, but is made simpler by the coding by convention paradigm. Since 1.0-RC3 this plugin requires Quartz 2.1.x and no longer supports Quartz 1.8.x.

2 Scheduling Basics

Scheduling Jobs

To create a new job run the "grails create-job" command and enter the name of the job. Grails will create a new job and place it in the "grails-app/jobs" directory:

```
class MyJob {
  static triggers = {
    simple name: 'mySimpleTrigger', startDelay: 60000, repeatInterval: 1000
  }
  def group = "MyGroup"
  def description = "Example job with Simple Trigger"

def execute(){
    print "Job run!"
  }
}
```

The above example will wait for 1 minute and after that will call the 'execute' method every second. The 'repeatInterval' and 'startDelay' properties are specified in milliseconds and must have Integer or Long type. If these properties are not specified default values are applied (1 minute for 'repeatInterval' property and 30 seconds for 'startDelay' property). Jobs can optionally be placed in different groups. The triggers name property must be unique across all triggers in the application.

By default, jobs will not be executed when running under the test environment.

Scheduling a Cron Job

Jobs can be scheduled using a cron expression. For those unfamiliar with "cron", this means being able to create a firing schedule such as: "At 8:00am every Monday through Friday" or "At 1:30am every last Friday of the month". (See the API docs for the CronTrigger class in Quartz for more info on cron expressions):

```
class MyJob  {
  static triggers = {
    cron name: 'myTrigger', cronExpression: "0 0 6 * * ?"
  }
 def group = "MyGroup"
 def description = "Example job with Cron Trigger"

def execute(){
   print "Job run!"
 }
}
```

The fields in the cronExpression are: (summarizing the Quartz CronTrigger Tutorial)

```
cronExpression: "s m h D M W Y"
                 | | | | | | `- Year [optional]
                 | | | | | `- Day of Week, 1-7 or SUN-SAT, ?
                 | | | | `- Month, 1-12 or JAN-DEC
                 | | | `- Day of Month, 1-31, ?
                 | | `- Hour, 0-23
                 | `- Minute, 0-59
                 `- Second, 0-59
```

- Year is the only optional field and may be omitted, the rest are mandatory
- Day-of-Week and Month are case insensitive, so "DEC" = "dec" = "Dec"
- Either Day-of-Week or Day-of-Month must be "?", or you will get an error since support by the underlying library is not complete. So you can't specify both fields, nor leave both as the all values wildcard "*"; this is a departure from the unix crontab specification.
- See the CronTrigger Tutorial for an explanation of all the special characters you may use.

3 Understanding Triggers

Scheduling configuration syntax

Currently plugin supports three types of triggers:

- **simple** — executes once per defined interval (ex. “every 10 seconds”);
- **cron** — executes job with cron expression (ex. “at 8:00am every Monday through Friday”);
- **custom** — your implementation of Trigger interface.

Multiple triggers per job are allowed.

```
class MyJob {
  static triggers = {
    simple name:'simpleTrigger', startDelay:10000, repeatInterval: 30000,
repeatCount: 10
    cron name:'cronTrigger', startDelay:10000, cronExpression: '0/6 * 15 * *
?'
    custom name:'customTrigger', triggerClass:MyTriggerClass,
myParam:myValue, myAnotherParam:myAnotherValue
  }

def execute() {
    println "Job run!"
  }
}
```

With this configuration job will be executed 11 times with 30 seconds interval with first run in 10 seconds after scheduler startup (simple trigger), also it’ll be executed each 6 second during 15th hour (15:00:00, 15:00:06, 15:00:12, … — this configured by cron trigger) and also it’ll be executed each time your custom trigger will fire.

Three kinds of triggers are supported with the following parameters:

- `simple`:
  - `name` — the name that identifies the trigger;
  - `startDelay` — delay (in milliseconds) between scheduler startup and first job’s execution;
  - `repeatInterval` — timeout (in milliseconds) between consecutive job’s executions;
  - `repeatCount` — trigger will fire job execution (1 + repeatCount) times and stop after that (specify 0 here to have one-shot job or -1 to repeat job executions indefinitely);
- `cron`:
  - `name` — the name that identifies the trigger;
  - `startDelay` — delay (in milliseconds) between scheduler startup and first job’s execution;
  - `cronExpression` — cron expression
- `custom`:
  - `triggerClass` — your class which implements Trigger interface;

any params needed by your trigger.

Dynamic Jobs Scheduling

Starting from 0.4.1 version you have the ability to schedule job executions dynamically.

These methods are available:

```
// creates cron trigger;
MyJob.schedule(String cronExpression, Map params?)

//  creates simple trigger: repeats job repeatCount+1 times with delay of
repeatInterval milliseconds;
MyJob.schedule(Long repeatInterval, Integer repeatCount?, Map params?) )

// schedules one job execution to the specific date;
MyJob.schedule(Date scheduleDate, Map params?)

//schedules job's execution with a custom trigger;
MyJob.schedule(Trigger trigger)

// force immediate execution of the job.
MyJob.triggerNow(Map params?)

// Each method (except the one for custom trigger) takes optional 'params'
argument.
// You can use it to pass some data to your job and then access it from the
job:
class MyJob {
  def execute(context) {
    println context.mergedJobDataMap.get('foo')
  }
}
// now in your controller (or service, or something else):

MyJob.triggerNow([foo:"It Works!"])
```

4 Plugin Configuration

Configuring the plugin

Since 0.3 version plugin supports configuration file which is stored in grails-app/conf/QuartzConfig.groovy. The syntax is the same as default Grails configuration file Config.groovy . You could also use per-environment configuration feature (more info).

To have an initial Quartz config file generated for you, type the following in the command line: 'grails install-quartz-config' . This will generate a file that looks like this:

```
quartz {
    autoStartup = true
    jdbcStore = false
}
environments {
    test {
        quartz {
            autoStartup = false
        }
    }
}
```

Currently supported options:

- `autoStartup` controls automatic startup of the Quartz scheduler during application bootstrap (default: true)
- `jdbcStore` set to true if you want Quartz to persist jobs in your DB (default: false), you'll also need to provide quartz.properties file and make sure that required tables exist in your db (see Clustering section below for the sample config and automatic tables creation using Hibernate)

You could also create grails-app/conf/quartz.properties file and provide different options to the Quartz scheduler (see Quartz configuration reference for details).

Logging

A log is auto-injected into your task Job class without having to enable it. To set the logging level, just add something like this to your grails-app/conf/Config.groovy log4j configuration.

```
debug 'grails.app.jobs'
```

Hibernate Sessions and Jobs

Jobs are configured by default to have Hibernate Session bounded to thread each time job is executed. This is required if you are using Hibernate code which requires open session (such as lazy loading of collections) or working with domain objects with unique persistent constraint (it uses Hibernate Session behind the scene). If you want to override this behavior (rarely useful) you can use 'sessionRequired' property:

```
def sessionRequired = false
```

Configuring concurrent execution

By default Jobs are executed in concurrent fashion, so new Job execution can start even if previous execution of the same Job is still running. If you want to override this behavior you can use 'concurrent' property, in this case Quartz's StatefulJob will be used (you can find more info about it here):

```
def concurrent = false
```

Configuring description

Quartz allows for each job to have a short description. This may be configured by adding a description field to your Job. The description can be accessed at runtime using the JobManagerService and inspecting the JobDetail object.

```
def description = "Example Job Description"
```

Clustering

Quartz plugin doesn't support clustering out-of-the-box now. However, you could use standard Quartz clustering configuration. Take a look at the example provided by Burt Beckwith. You'll also need to set jdbcStore configuration option to true .

There are also two parameters for configuring store/clustering on jobs (volatility and durability , both are true by default) and one for triggers (volatility , also true by default). Volatile job and trigger will not persist between Quartz runs, and durable job will live even when there is no triggers referring to it.

Read Quartz documentation for more information on clustering and job stores as well as volatility and durability.

Now that the plugin supports Quartz 2.1.x, you can now use current versions of open source Terracotta see https://github.com/rvanderwerf/terracotta-grails-demo for an example app.

Recovering

Since 0.4.2 recovering from 'recovery' or 'fail-over' situation is supported with requestsRecovery job-level flag (false by default).

If a job "requests recovery", and it is executing during the time of a 'hard shutdown' of the scheduler (i.e. the process it is running within crashes, or the machine is shut off), then it is re-executed when the scheduler is started again. In this case, the JobExecutionContext.isRecovering() method will return true.