JQM

Oct 19, 2018

Contents

1 JQM features **3**

2 How JQM works **5**
- 2.1 Basic JQM concepts . . . 5
- 2.2 General architecture . . . 6
- 2.3 Nodes, queues and polling . . . 7
- 2.4 Job Instance life-cycle . . . 8

3 Quickstart **9**
- 3.1 Docker on Windows or Linux/Mac . . . 9
- 3.2 Windows without Docker . . . 9
- 3.3 Linux / Unix without Docker . . . 10
- 3.4 Next steps... . . . 11

4 Payload development **13**
- 4.1 Payload basics . . . 13
  - 4.1.1 Payloads types . . . 13
  - 4.1.2 Accessing the JQM engine API . . . 15
  - 4.1.3 Creating files . . . 16
  - 4.1.4 Going to the culling . . . 16
  - 4.1.5 Full example . . . 17
  - 4.1.6 Limitations . . . 18
  - 4.1.7 Staying reasonable . . . 19
- 4.2 Logging . . . 19
- 4.3 Using resources . . . 19
  - 4.3.1 Introduction . . . 19
  - 4.3.2 JDBC . . . 20
  - 4.3.3 JMS . . . 20
  - 4.3.4 Files . . . 22
  - 4.3.5 URL . . . 22
- 4.4 Engine API . . . 22
  - 4.4.1 Current job metadata . . . 22
  - 4.4.2 Enqueue & retrieve jobs . . . 23
  - 4.4.3 Communications . . . 23
  - 4.4.4 Misc. . . . 24
- 4.5 Packaging . . . 24
  - 4.5.1 Libraries handling . . . 24

4.5.2 Shared libraries . . . 25
4.5.3 Pure Maven package . . . 25
4.5.4 Creating a JobDef . . . 26
4.6 Testing payloads . . . 29
4.6.1 Unit testing . . . 29
4.6.2 Integration tests . . . 29
4.7 Understanding the execution context . . . 31
4.7.1 The default mode: isolation . . . 31
4.7.2 Changing the default mode . . . 31
4.7.3 Advanced mode: context definition . . . 32
4.8 Using Spring . . . 34
4.8.1 By doing nothing special . . . 35
4.8.2 By having JQM set the context . . . 36

5 Client development **39**
5.1 Introduction . . . 39
5.2 Simple web API . . . 39
5.3 Full client API . . . 40
5.3.1 Basics . . . 41
5.3.2 Client API details . . . 42
5.3.3 Query API . . . 45
5.3.4 JPA Client API . . . 47
5.3.5 Web Service Client API . . . 50
5.4 CLI API . . . 54
5.5 Engine API . . . 55

6 Administration **57**
6.1 Installation . . . 57
6.1.1 Docker install . . . 57
6.1.2 Binary install . . . 57
6.1.3 Enabling the web interface . . . 59
6.1.4 Database configuration . . . 59
6.1.5 Global configuration . . . 61
6.1.6 JNDI configuration . . . 61
6.2 Command Line Interface (CLI) . . . 61
6.3 JMX monitoring . . . 62
6.3.1 Monitoring JQM through JMX . . . 62
6.3.2 Remote JMX access . . . 65
6.3.3 Beans detail . . . 65
6.4 Web administration console . . . 67
6.4.1 Enabling the console . . . 67
6.4.2 First connection . . . 67
6.4.3 If TLS is enabled . . . 67
6.5 Logs . . . 68
6.5.1 Log levels . . . 68
6.5.2 Engine log . . . 68
6.5.3 Java Virtual Machine Log . . . 68
6.5.4 Payload logs . . . 68
6.6 Operations . . . 69
6.6.1 Starting . . . 69
6.6.2 Stopping . . . 69
6.6.3 Restarting . . . 70
6.6.4 Pausing and resuming . . . 70
6.6.5 Backup . . . 70

6.6.6 Purges . . . 71
6.7 Parameters . . . 71
6.7.1 Engine parameters . . . 71
6.8 Managing queues . . . 74
6.8.1 Defining queues . . . 74
6.8.2 Defining pollers . . . 74
6.9 Administrating resources . . . 75
6.9.1 Defining a resource . . . 75
6.9.2 Singletons . . . 75
6.9.3 Examples . . . 76
6.10 Managing history . . . 78
6.11 Data security . . . 79
6.11.1 Data & data flows . . . 79
6.11.2 Summary of elements to protect . . . 81
6.11.3 Security mechanisms . . . 81
6.11.4 Monitoring access security . . . 83
6.11.5 Tracing . . . 83
6.12 Database and reliability . . . 83
6.13 Administration web services . . . 84
6.14 Scheduling jobs . . . 86
6.14.1 Recurrence rules . . . 86
6.14.2 Starting later . . . 87
6.14.3 Manual scheduling . . . 87

7 In case of trouble **89**
7.1 Troubleshooting . . . 89
7.2 Reporting bugs and requests . . . 89
7.3 Bug report . . . 89

8 Developement **91**
8.1 Contributing to JQM . . . 91
8.2 Conventions and style . . . 92
8.2.1 Java . . . 92
8.2.2 Database . . . 92
8.3 Release process . . . 93
8.3.1 Environment . . . 93
8.3.2 Update release notes . . . 93
8.3.3 Checkout . . . 93
8.3.4 Full build & tests . . . 93
8.3.5 Sonar snapshot . . . 94
8.3.6 Release test . . . 94
8.3.7 Release . . . 94
8.3.8 Git push . . . 94
8.3.9 Documentation . . . 94
8.3.10 GitHub upload . . . 95
8.3.11 Docker Hub upload . . . 95
8.4 Release notes . . . 95
8.4.1 2.1.0 . . . 95
8.4.2 2.0.0 . . . 96
8.4.3 1.4.1 . . . 98
8.4.4 1.3.6 . . . 99
8.4.5 1.3.5 . . . 100
8.4.6 1.3.4 . . . 100
8.4.7 1.3.3 . . . 101

8.4.8 1.3.2 . . . 102
8.4.9 1.3.1 . . . 102
8.4.10 1.2.2 . . . 103
8.4.11 1.2.1 . . . 104
8.4.12 1.1.6 . . . 105
8.4.13 1.1.5 . . . 106
8.4.14 1.1.4 . . . 106
8.4.15 1.1.3 . . . 108
8.5 Classloading . . . 108
8.6 Testing . . . 109
8.6.1 Automated builds . . . 109
8.6.2 Tests . . . 110

9 Glossary **113**

JQM (short for Job Queue Manager) is a Java batch job manager. It takes standard Java code (which does not need to be specifically tailored for JQM) and, when receiving an execution request, will run it asynchronously, taking care of everything that would otherwise be boilerplate code with no added value whatsoever: configuring libraries, logs, throttling processes, handling priorities between different classes of jobs, distributing the load over multiple servers, distributing the files created by the jobs, and much more… Basically, it is a very lightweight application server specifically tailored for making it easier to run Java batch jobs.

The rational behind JQM is that there are too many jobs that fall inside that uncomfortable middle ground between "a few seconds" (this could be done synchronously inside a web application server) and "a few hours" (in which case forking a new dedicated JVM is often the most suitable way). A traditional servlet or J2EE application server should not house this kind of jobs: they are designed to deal with very short locally-running synchronous user requests, not asynchronous distributed long running jobs. For example, creating a thread in such a server is dangerous as they offer little control over it, and they do not offer job queuing and remote execution (which is a basic tenet of asynchronous batch execution).

JQM should also be considered just for its ability to untangle the execution itself from the program that requires it. Some of the most obvious applications are:

- relieving the application server, which often costs money - the front end stays on the licensed application server on an expensive server, while the resource consuming jobs go inside JQM on low-end servers.
- job execution request frequency adaptation. Often a job is requested to run multiple times at the same moment (either by a human request, or an automated system reacting to frequent events, …) while the job should actually run only one at a time (e.g. the job handles all available data at the time of its launch - so there is really no need for multiple instances in parallel). JQM will throttle these requests.
- adding a queueing & distributed execution capacity to job schedulers.

Most of the time, the code that will be run by JQM will be a direct reuse of existing code without any modifications (for jars including a classic main function, or Runnable threads). But JQM also optionally offers a rich API that allows running code to ask for another execution, to retrieve structured parameters, to send messages and other advancement notices… Also of note, JQM is pure Java Standard Edition 6 (JSE 1.6) to enable not only code but binary reuse. Standard JSE code also means it is possible to use any framework within JQM, like Spring batch or Hibernate.

Interacting with JQM is also easy: an API, with two different implementations (SQL & REST web service, which can be used from a non-Java world) for different needs, is offered to do every imaginable operation (new execution request, querying the state of a request, retrieving files created by a job instance, …).

It is also of note that JQM was created with compatibility in mind:

- uses either PostgreSQL, Oracle, MySQL, DB2 or HSQLDB
- the client API is usable in all application servers and JSE code (tested with WebsSphere 8.x, Glassfish 3.x, Tomcat 7.x, JBoss 7+…)
- one of the client API implementations is a REST-like API, callable from everywhere, not only Java but also .NET or shell scripts (which by the way allows very easy scheduler integration).
- under an Apache 2 license, which basically allows you to do anything you want with the product and its code in any environment

Finally, JQM is a free (as beer) and open source product backed by the IT consulting firm [Enioka](http://www.enioka.com) which first developed it for a multinational conglomerate. Enquiries about support, development of extensions, integration with other products, consulting and other commercial questions are more than welcome at contact@enioka.com. Community support is of course freely offered on GitHub using the bug-tracker.

CHAPTER 1

JQM features

- The only dedicated Java batch server
- Open Source under the Apache 2 licence, a business-friendly licence securing your investments in JQM
- No-cost ready to use solution. Paying support can if needed be purchased from the original authors at contact@enioka.com or at any other firm open to doing maintenance on the tool.
- Fully documented

Batch code is easy to create:

- Runs existing Java 1.6 to 1.10 code, without need for programming specifically for JQM
- Possible (but not required) to use a specific framework of your choice (Spring batch, etc.)
- Possible (but not required) to easily tweak class loading to enable advance scenarios
- Many samples for all features (inside JQM's own integration tests)
- Specific API to handle file creation and easy retrieval (a file can be created on any server and retrieved from another in a single call)
- Embedded standard JNDI directory with JDBC connection pooling for jobs needing database connectivity
- Jobs can be tested as if they were running inside a JQM node thanks to a test library which can be used in jQuery tests.
- Can easily report an advancement status to users or administrators
- All JQM artefacts (optional libraries developers may want to use in some cases) are published on Maven Central and therefore easily integrate with most build systems

Interacting with batch jobs is simple:

- Query API enabling to easily create client applications (with two full samples included in the distribution), such as web pages listing all the jobs for given user, for a given module, etc.
- Feature rich API with provided Java clients, which can be used out of the box for launching jobs, cancelling them, changing their priorities

Batch packaging: just use your own

- Full Maven 3 support: as a Maven-created jar contains its pom.xml, JQM is able to retrieve all the dependencies, simplifying packaging libraries.
- It is even possible to directly run Maven coordinates without providing any jar file!
- More classic packaging also supported (library directory, or putting all libraries inside the jar)

Administration is a breathe:

- Both command line and web-based graphic user interface for administration
- Can run in Docker environments with provided images optimized for development usage as well as scale-out production scenarios (Swarm, Kubernetes. . .)
- Can run as a Windows service or a Linux /etc/init.d script
- Fully ready to run out of the box without complicated configuration
- Supported on most OSes and databases
- Log files can be accessed easily through a distributed web GUI
- Easy definition of service classes (VIP jobs, standard jobs, . . .) through queues
- Easy integration with schedulers and CLI
- Most configuration changes are hot-applied, with little to no need for server restarts
- Resists most environment failures (database failure, network failure, . . .)
- Maintenance jobs are integrated
- Can be fully monitored through JMX - which make it compatible with most monitoring systems out of the box (a Zabbix template is provided)
- Authentication and permissions handling is fully in-box, including an optional PKI to create client certificates.

CHAPTER 2

How JQM works

2.1 Basic JQM concepts

The goal of JQM is to launch *payloads*, i.e. Java code doing something useful, asynchronously. This code can be anything - a shell program launcher, a Spring batch, really anything that works with Java SE and libraries provided with the code.

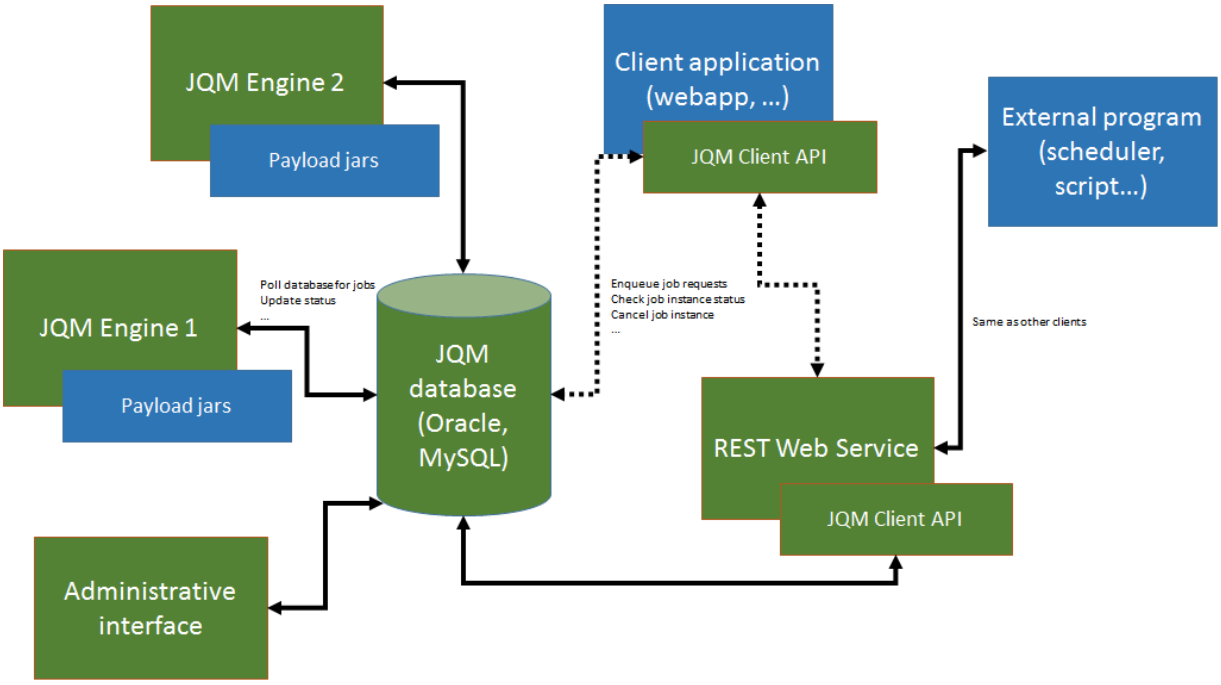
The payload is described inside a *job definition* - so that JQM knows things like the class to load, the path of the jar file if any, etc. It is usually contained within an XML file. The job definition is actually a deployment descriptor - the batch equivalent for a web.xml or an ejb-jar.xml.

A running payload is called a *job instance* (the "instance of a job definition"). These instances wait in *queues* to be run, then are run and finally archived. To create a job instance, a *job request* is posted by a client. It contains things such as optional parameters values, but most importantly it specifies a job definition so that JQM will know what to run.

Job instances are run by one or many engines called *JQM nodes*. These are simply Java processes that poll the different queues in which job instances are waiting. Runs take place within threads, each with a dedicated class loader so as to fully isolate them from each others (this is the default behaviour - class loader sharing is also possible).

Full definitions are given inside the *Glossary*.

2.2 General architecture



On this picture, JQM elements are in green while non-JQM elements are in blue.

JQM works like this:

- an application (for example, a J2EE web application but it could be anything as long as it can use a Java SE library) needs to launch an asynchronous job
- it imports the JQM client (one of the two - web service or direct-to-database. There are two dotted lines representing this choice on the diagram)
- it uses the 'enqueue' method of the client, passing it a job request with the name of the job definition to launch (and potentially parameters, tags, . . .)
- a job instance is created inside the database
- engines are polling the database (see below). One of them with enough free resources takes the job instance
- it creates a dedicated class loader for this job instance, imports the correct libraries with it, launches the payload inside a thread
- during the run, the application that was at the origin of the request can use other methods of the client API to retrieve the status, the advancement, etc. of the job instance
- at the end of the run, the JQM engine updates the database and is ready to accept new jobs. The client can still query the history of executions.

It should be noted that clients never speak directly to a JQM engine - it all goes through the database.

Note: There is one exception to this: when job instances create files that should be retrieved by the requester, the 'direct to database' client will download the files through a direct HTTP GET call to the engine. This avoids creating and maintaining a central file repository. The 'web service' client does not have this issue as it always uses web services for all methods.

2.3 Nodes, queues and polling

As it names entails, JQM is actually a queue manager. As many queues as needed can be created. A queue contains job instances waiting to be executed.

To get a JQM node to poll a queue, it is necessary to associate the node with the queue. The association specifies:

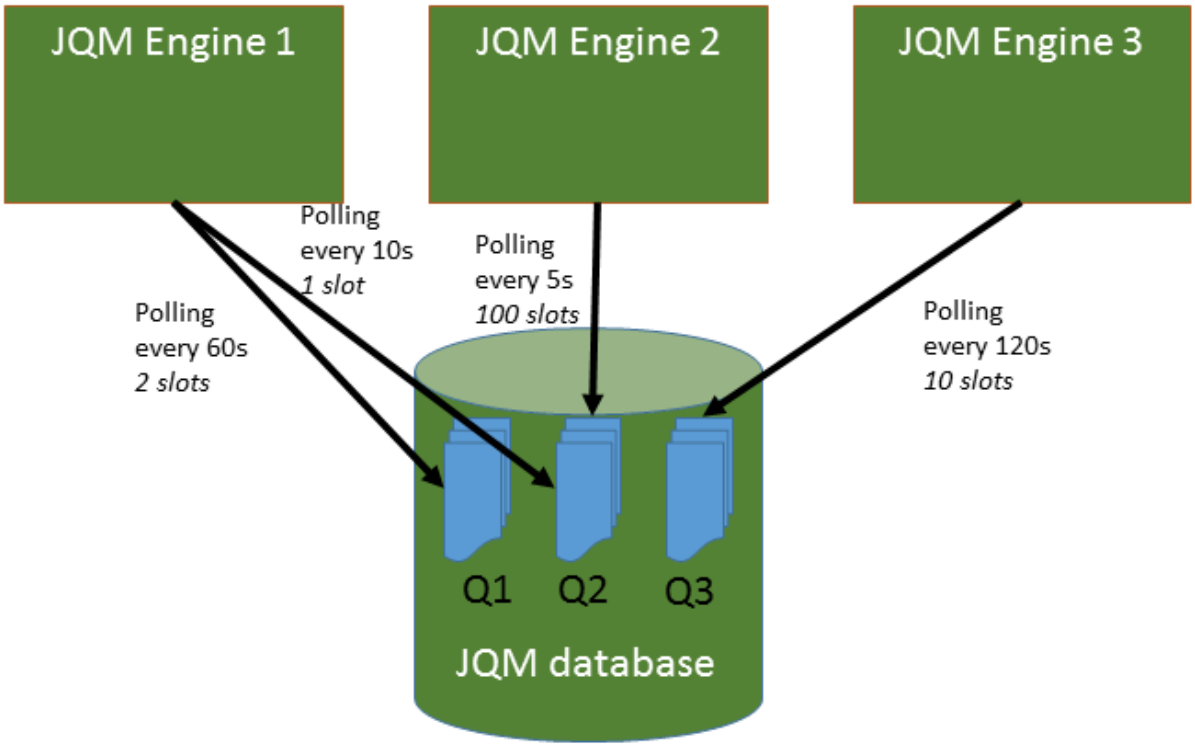
- The node which should poll
- The queue which should be polled
- The maximum number of job instances the node should be able to run at the same time for this queue (a maximum thread count)
- The polling interval (minimum is 1 second to spare the poor database). This is the time between two requests to the queue. On each request, the node will try to fill up its maximum number of job instances (if the maximum is three concurrent job instances and nothing is running yet, the node will ask for three jobs on each loop).

By default, when creating the first engine, one queue is created and is tagged as the default queue (meaning all jobdef that do not have a specific queue will end on that one). All further new nodes will have one association created to this default queue.

With this way to associate queue to nodes, it is very easy to specialise nodes and usages. It is for example possible to have one node polling a queue very fast, while another polls it more slowly. More interestingly, it makes it easy to adapt to the size of the underlying servers: one engine may be able to run 10 jobs for a queue in parallel while another, on a more powerful server, may run 50 for the same queue. It also makes it possible to specialise servers: some will poll a set of queues, others will poll a completly different set of queues.

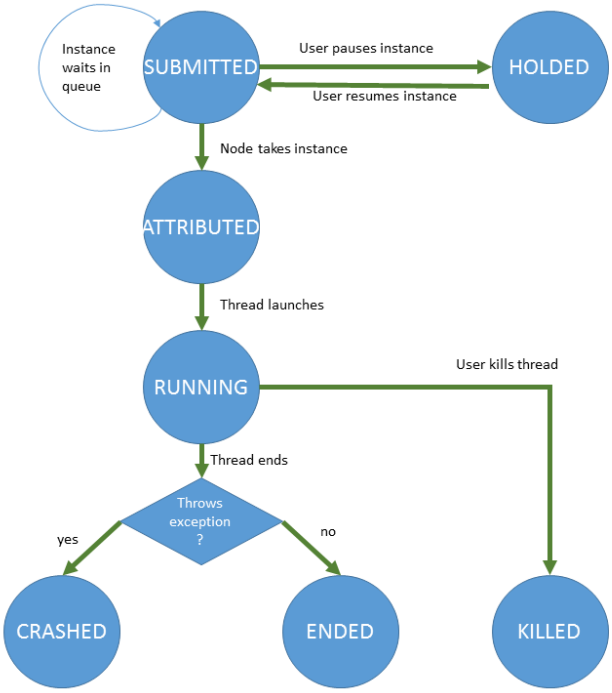
A *Job Definition* has a default queue: all job requests pertaining to a job definition are created (unless otherwise specified) inside this queue. It is possible at job request submission, or later once the job instance waits inside its queue, to move a job instance from one queue to another *as long as it has not already began to run*.

An example:



Here, there are three queues and three engine nodes inside the JQM cluster. Queue 1 is only polled by engine 1. Queue 3 is only polled by engine 3. But queue 2 is polled both by engine 1 and engine 2 at different frequencies. Engine 2 may have been added because there was too much wait time on queue 2 (indeed, engine 1 only will never run more than one job instance at the same time for queue 2 as it has only one slot. Engine 2 has 100 so with both engines at most 101 instances will run for queue 2).

2.4 Job Instance life-cycle



This represents all the states a *job instance* goes through. The diagram is self explanatory, but here are a few comments:

- The first state, SUBMITTED, happens when a *job request* is submitted hence its name. It basically is a "waiting in queue" state.
- The ATTRIBUTED state is transient since immediately afterwards the engine will launch the thread representing the running job (and the instance will take the RUNNING state). Engines never take in instances if they are unable to run it (i.e. they don't have free slots for this queue) so instances cannot stay in this state for long. It exists to signal all engines that a specific engine has promised to launch the instance and that no one else should try to launch it while it prepares the launch (which takes a few milliseconds).

CHAPTER 3

Quickstart

This guide will show how to run a job inside JQM with the strict minimum of operations. The resulting installation is not suitable for production at all, but perfect for development environments. It also gives pointers to the general documentation.

3.1 Docker on Windows or Linux/Mac

Prerequisites:

- Docker is configured and can access public images
- A recent Windows (greater or equal to 1709) or Linux (this includes Macs which run containers inside a hidden Linux VM)

Just run the very classic:

```
docker run -it --rm -p 1789:1789 enioka/jqm
```

The log inside the console should give you an indication "Jetty has started on port 1789". You can now use your preferred browser to go to http://localhost:1789 and browse the administration console.

Go to the last tab, click on "new launch" on the bottom, select the "DemoEcho" job and validate. The job should run and appear in the history list when clicking on "refresh". Congratulations, that was a first JQM job instance launch!

Use Ctrl+C inside the console to stop the engine with the container.

3.2 Windows without Docker

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterwards.
- An account with full permissions in JQM_ROOT. Not need for admin or special rights - it just needs to be able to open a PowerShell session.

The following script will download and copy the binaries (adapt the first two lines).

```
$JQM_ROOT = "C:\TEMP\jqm" ## Change this
$JQM_VERSION = "1.3.3"  ## Change this
mkdir -Force $JQM_ROOT; cd $JQM_ROOT
Invoke-RestMethod https://github.com/enioka/jqm/releases/download/jqm-all-$JQM_
↪VERSION/jqm-$JQM_VERSION.zip -OutFile jqm.zip
$shell = new-object -com shell.application
$zip = $shell.NameSpace((Resolve-Path .\jqm.zip).Path)
foreach($item in $zip.items()) { $shell.Namespace($JQM_ROOT).copyhere($item) }
rm jqm.zip; mv jqm*/* .
```

The following script will create a database and reference the test jobs (i.e. *payloads*) inside a test database:

```
./jqm.ps1 createnode # This will create a new node named after the computer name
./jqm.ps1 allxml  # This will import all the test job definitions
```

The following script will enable the web console with account root/test (do not use this in production!):

```
./jqm.ps1 enablegui -RootPassword test
```

The following script will *enqueue* an execution request for one of the test jobs:

```
./jqm.ps1 -Enqueue DemoEcho
```

Finally this will start an engine inside the console.:

```
./jqm.ps1 startconsole
```

Just check the JQM_ROOT/logs directory - a numbered log file should have appeared, containing the log of the test job.

The log inside the console should give you an indication "Jetty has started on port <PORT>". You can now use your preferred browser to go to localhost:port and browse the administration console. Use Ctrl+C inside the PowerShell console to stop the engine.

3.3 Linux / Unix without Docker

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterwards.
- An account with full permissions in JQM_ROOT. Not need for administrative or special permissions.

The following script will download and install the binaries (adapt the first two lines).

```
wget  https://github.com/enioka/jqm/releases/download/jqm-all-1.3.3/jqm-1.3.3.tar.gz
↪# For 1.3.3 release. Adapt it to the one you want.
tar xvf jqm-1.3.3.tar.gz
```

The following script will create a database and reference the test jobs (i.e. *payloads*) inside a test database:

```
cd jqm-1.3.3
./jqm.sh createnode
./jqm.sh allxml  # This will import all the test job definitions
```

The following script will enable the web console with account root/test (do not use this in production!):

```
./jqm.sh enablegui test
```

The following script will *enqueue* an execution request for one of the test jobs:

```
./jqm.sh enqueue DemoEcho
```

Finally this will start an engine inside the console.:

```
./jqm.sh startconsole
```

Just check the JQM_ROOT/logs directory - a numbered log file should have appeared, containing the log of the test job.

3.4 Next steps...

Note: Congratulations, you've just run your first JQM batch! This batch is simply a jar with a main function doing an echo - a totally usual Java JSE program with no extensions whatsoever. If using standard JSE is not enough, just read the *Payload development* chapter.

To exit the engine, simply do Ctrl+C or close your console.

To go further: engines under Windows should be installed as services. This is easily done and explained in the *full install documentation*. Moreover, this test install is using a very limited (and limiting) database - the full doc also explains how to use fully fledged databases.

CHAPTER 4

Payload development

4.1 Payload basics

JQM is a specialized application server dedicated to ease the management of Java batch jobs. Application servers usually have two main aspects: on one hand they bring in frameworks to help writing the business programs, on the other they try to ease daily operations. For example, JBoss or Glassfish provide an implementation of the EE6 framework for building web applications, and provide many administration utilities to deploy applications, monitor them, load balance them, etc.

JQM's philosophy is that **all existing Java programs should be reusable as is**, and that programmers should be free to use whatever frameworks they want (if any at all). Therefore, JQM nearly totally forgoes the "framework" part and concentrates on the management part. For great frameworks created for making batch jobs easier to write, look at Spring batch, a part of Spring, or JSR 352, a part of EE7. As long as the required libraries are provided, JQM can run *payloads* based on all these frameworks.

This section aims at giving all the keys to developers in order to create great batch jobs for JQM. This may seem in contradiction with what was just said: why have a "develop for JQM" chapter if JQM runs any Java code?

- First, as in all application server containers, there a a few guidelines to respect, such as packaging rules.
- Then, as an option, JQM provides a few APIs that can be of help to batch jobs, such as getting the ID of the run or the caller name.

But this document must insist: unless there is a need to use the APIs, there is no need to develop specifically for JQM. **JQM runs standard JSE code**.

4.1.1 Payloads types

There are three *payload* types: programs with a good old main (the preferred method for newly written jobs), and two types designed to allow reuse of even more existing binaries: Runnable implementers & JobBase extenders.

Main

This a classic class containing a "static void main(String[] args)" function.

In that case, JQM will simply launch the main function. If there are some arguments defined (default arguments in the *JobDef* or arguments given at enqueue time) their value will be put inside the String[] parameter *ordered by key name*.

There is no need for any dependencies towards any JQM libraries in that case - direct reuse of existing code is possible.

This would run perfectly, without any specific dependencies or imports:

```
public class App
{
    public static void main(String[] args)
    {
        System.out.println("main function of payload");
    }
}
```

Note: It is not necessary to make jars executable. The jar manifest is ignored by JQM.

Runnable

Some existing code is already written to be run as a thread, implementing the Runnable interface. If these classes have a no-argument constructor (this is not imposed by the Runnable interface as interfaces cannot impose a constructor), JQM can instantiate and launch them. In that case, the run() method from the interface is executed. As it takes no arguments, it is not possible to access parameters without using JQM specific methods as described later in this chapter.

This would run perfectly, without any specific dependencies or imports:

```
public class App implements Runnable
{
    @Override
    public void run()
    {
        System.out.println("run method of runnable payload");
    }
}
```

Explicit JQM job

Warning: This is deprecated and should not be used for new payloads

This type of job only exists for ascending compatibility with a former limited JQM version. It consisted in subclassing class JobBase, overloading method start() and keeping a no-arg constructor. Parameters were accessible through a number of accessors of the base class.

For example (note the import and the use of an accessor from the base class):

```
import com.enioka.jqm.api.JobBase;

public class App extends JobBase
```

(continues on next page)

(continued from previous page)

```
{
    @Override
    public void start()
    {
        System.out.println("Date: " + new Date());
        System.out.println("Job application name: " + this.getApplicationName());
    }
}
```

It requires the following dependency (Maven):

```
<dependency>
    <groupId>com.enioka.jqm</groupId>
    <artifactId>jqm-api</artifactId>
    <version>${jqm.version}</version>
</dependency>
```

4.1.2 Accessing the JQM engine API

Sometimes, a job will need to directly interact with JQM, for operations such as:

- *enqueue* a new *Job Request*
- get the different IDs that identify a *Job Instance* (i.e. a run)
- get a resource (see *Using resources*)
- get the optional data that was given at *enqueue* time
- report progress to an end user
- ...

For this, an interface exists called `JobManager` inside jar jqm-api.jar. Using it is trivial: just create a field (static or not) inside your job class (whatever type - Main, Runnable or JQM) and the engine will **inject an implementation ready for use**.

Note: the 'explicit JQM jobs' payload type already has one `JobManager` field named jm defined in the base class JobBase - it would have been stupid not to define it as the API must be imported anyway for that payload type.

The dependency is:

```
<dependency>
    <groupId>com.enioka.jqm</groupId>
    <artifactId>jqm-api</artifactId>
    <version>${jqm.version}</version>
    <scope>provided</scope>
</dependency>
```

For more details, please read *Engine API*.

Note: the scope given here is provided. It means it will be present for compilation but not at runtime. Indeed, JQM always provides the jqm-api.jar to its payloads without them needing to package it. That being said, packaging it (default 'compile' scope) is harmless as it will be ignored at runtime in favour of the engine-provided one.

4.1.3 Creating files

An important use case for JQM is the generation of files (such as reports) at the direct request of an end-user through a web interface (or other interfaces). It happens when generating the file is too long or resource intensive for a web application server (these are not made to handle 'long' processes), or blocking a thread for a user is unacceptable: the generation must be deported elsewhere. JQM has methods to do just that.

In this case, the *payload* simply has to be the file generation code. However, JQM is a distributed system, so unless it is forced into a single node deployment, the end user has no idea where the file was generated and cannot directly retrieve it. The idea is to notify JQM of a file creation, so that JQM will take it (remove it from the work directory) and reference it. It is then be made available to clients through a small HTTP GET that is leveraged by the engine itself (and can be proxied).

The method to do so is *`JobManager.addDeliverable()`* from the *Engine API*.

Note: Work/temp directories are obtained through *`JobManager.getWorkDir()`*. These are purged after execution. Use of temporary Java files is strongly discouraged - these are purged only on JVM exit, which on the whole never happens inside an application server.

Example:

```
import java.io.FileWriter;
import java.io.PrintWriter;

public class App implements Runnable
{
    private JobManager jm;

    @Override
    public void run()
    {
        String dir = jm.getWorkDir();
        String fileName = dir + "/temp.txt";
        try
        {
            PrintWriter out = new PrintWriter(fileName);
            out.println("Hello World!");
            out.close();
            addDeliverable(fileName, "ThisIsATag");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

4.1.4 Going to the culling

Payloads are run inside a thread by the JQM engine. Alas, Java threads have one caveat: they cannot be cleanly killed. Therefore, there is no obvious way to allow a user to kill a job instance that has gone haywire. To provide some measure of relief, the *Engine API* provides a method called *`JobManager.yield()`* that, when called, will do nothing but give briefly control of the job's thread to the engine. This allows the engine to check if the job should be killed (it throws an exception as well as sets the thread's interruption status to do so). Now, if the job instance really

has entered an infinite loop where yield is not called nor is the interruption status read, it won't help much. It is more to allow killing instances that run well (user has changed his mind, etc.).

To ease the use of the kill function, all other engine API methods actually call yield before doing their own work.

Finally, for voluntarily killing a running payload, it is possible to do much of the same: throwing a runtime exception. Note that System.exit is forbidden by the Java security manager inside payloads - it would stop the whole JQM engine, which would be rather impolite towards other running job instances.

4.1.5 Full example

This fully commented payload uses nearly all the API.

```
import com.enioka.jqm.api.JobManager;

public class App
{
    // This will be injected by the JQM engine - it could be named anything
    private static JobManager jm;

    public static void main(String[] args)
    {
        System.out.println("main function of payload");

        // Using JQM variables
        System.out.println("run method of runnable payload with API");
        System.out.println("JobDefID: " + jm.jobApplicationId());
        System.out.println("Application: " + jm.application());
        System.out.println("JobName: " + jm.applicationName());
        System.out.println("Default JDBC: " + jm.defaultConnect());
        System.out.println("Keyword1: " + jm.keyword1());
        System.out.println("Keyword2: " + jm.keyword2());
        System.out.println("Keyword3: " + jm.keyword3());
        System.out.println("Module: " + jm.module());
        System.out.println("Session ID: " + jm.sessionID());
        System.out.println("Restart enabled: " + jm.canBeRestarted());
        System.out.println("JI ID: " + jm.jobInstanceID());
        System.out.println("Parent JI ID: " + jm.parentID());
        System.out.println("Nb of parameters: " + jm.parameters().size());

        // Sending info to the user
        jm.sendProgress(10);
        jm.sendMsg("houba hop");

        // Working with a temp directory
        File workDir = jm.getWorkDir();
        System.out.println("Work dir is " + workDir.getAbsolutePath());

        // Creating a file made available to the end user (PDF, XLS, ...)
        PrintWriter writer;
        File dest = new File(workDir, "marsu.txt");
        try
        {
            writer = new PrintWriter(dest, "UTF-8");
        }
        catch (FileNotFoundException e)
        {
```

(continues on next page)

(continued from previous page)

```
            e.printStackTrace();
            return;
        }
        catch (UnsupportedEncodingException e)
        {
            e.printStackTrace();
            return;
        }
        writer.println("The first line");
        writer.println("The second line");
        writer.close();
        try
        {
            jm.addDeliverable(dest.getAbsolutePath(), "TEST");
        }
        catch (IOException e)
        {
            e.printStackTrace();
            return;
        }

        // Using parameters & enqueue (both sync and async)
        if (jm.parameters().size() == 0)
        {
            jm.sendProgress(33);
            Map<String, String> prms = new HashMap<String, String>();
            prms.put("rr", "2nd run");
            System.out.println("creating a new async job instance request");
            int i = jm.enqueue(jm.applicationName(), null, null, null, jm.
↪application(), jm.module(), null, null, null, prms);
            System.out.println("New request is number " + i);

            jm.sendProgress(66);
            prms.put("rrr", "3rd run");
            System.out.println("creating a new sync job instance request");
            jm.enqueueSync(jm.applicationName(), null, null, null, jm.application(),␣
↪jm.module(), null, null, null, prms);
            System.out.println("New request is number " + i + " and should be done now
↪");
            jm.sendProgress(100);
        }
    }
}
```

4.1.6 Limitations

Nearly all JSE Java code can run inside JQM, with the following limitations:

- no system.exit allowed - calling this will trigger a security exeption.
- ... This list will be updated when limits are discovered. For now this is it!

Changed in version 1.2.1: JQM used to use a thread pool for running its job instances before version 1.2.1. This had the consequence of making thread local variables very dangerous to use. It does not any more - the performance gain was far too low to justify the impact.

4.1.7 Staying reasonable

JQM is some sort of light application server - therefore the same type of guidelines apply.

- Don't play (too much) with class loaders. Creating and swapping them is allowed because some frameworks require them (such as Hibernate) and we wouldn't want existing code using these frameworks to fail just because we are being too strict.
- Don't create threads. A thread is an unmanageable object in Java - if it blocks for whatever reason, the whole application server has to be restarted, impacting other jobs/users. They are only allowed for the same reason as for creating class loaders.
- Be wary of bootstrap static contexts. Using static elements is all-right as long as the static context is from your class loader (in our case, it means classes from your own code or dependencies). Messing with static elements from the bootstrap class loader is opening the door to weird interactions between jobs running in parallel. For example, loading a JDBC driver does store such static elements, and should be frowned upon (use a shared JNDI JDBC resource for this).
- Don't redefine System.setOut and System.setErr - if you do so, you will loose the log created by JQM from your console output. See *Logging*.

4.2 Logging

Once again, running Java code inside JQM is exactly as running the same code inside a bare JVM. Therefore, there is nothing specific concerning logging: if some code was using log4j, logback or whatever, it will work. However, for more efficient logging, it may be useful to take some extra care in setting the parameters of the loggers:

- the "current directory" is not defined (or rather, it is defined but is guaranteed to be the same each time), so absolute paths are better
- JQM captures the console output of a job to create a log file that can be retrieved later through APIs.

Therefore, **the recommended approach for logging in a JQM payload is to use a Console Appender and no explicit log file**.

4.3 Using resources

4.3.1 Introduction

Most programs use some sort of resource - some read files, other write to a relational database, etc. In this document, we will refer to a "resource" as the description containing all the necessary data to use it (a file path, a database connection string + password, ...)

There are many approaches to define these resources (directly in the code, in a configuration file...) but they all have caveats (mostly: they are not easy to use in a multi environment context, where resource descriptions change from one environment to another). All these approaches can be used with JQM since JQM runs all JSE code. Yet, Java has standardized JNDI as a way to retrieve these resources, and JQM provides a limited JNDI directory implementation that can be used by the *payloads*.

JQM JNDI directory can be used for:

- JDBC connections
- JMS resources
- Files

- URLs
- Simple Strings
- Mail session (outgoing SMTP only)
- every ObjectFactory provided by the payloads

Warning: JNDI is actually part of JEE, not JSE, but it is so useful in the context of JQM use cases that it was implemented. The fact that it is present does **not** mean that JQM is a JEE container. Notably, there is no injection mechanism and JNDI resources have to be manualy looked up.

Note: An object returned by a JNDI lookup (in JQM or elsewhere) is just a description. The JNDI system has not checked if the object existed, if all parameters are present, etc. It also means that it is the client's respsonsbility to open files, database connections... and **close them in the end**.

The JNDI system is totally independent from the JQM API described in *Accessing the JQM engine API*. It is always present, whatever type your payload is and even if the jqm-api jar is not present.

By 'limited', we mean the directory only provides a single root JNDI context. Basically, all JNDI lookups are given to the same JNDI context and are looked up inside the JQM database by name exactly as they are given (case-sensitive).

This chapter focusses on using them inside payloads. To define resources, see *Administrating resources*.

Below are some samples & details for various cases.

4.3.2 JDBC

```
DataSource ds = InitialContext.doLookup("jdbc/superalias");
```

Please note the use of InitialContext for the context lookup: as noted above, JQM only uses the root context.

It is interesting to note that the JQM NamingManager is standard - it can be used from wherever is needed, such as a JPA provider configuration: in a persistence.xml, it is perfectly valid to use `<non-jta-datasource>jdbc/superalias</non-jta-datasource>`.

If all programs running inside a JQM cluster always use the same database, it is possible to define a JDBC alias as the "default connection" (cf. *Parameters*). It can then be retrieved directly through the *`JobManager.getDefaultConnection()`* method of the JQM engine API. (this is the only JNDI-related element that requires the API).

4.3.3 JMS

Connecting to a JMS broker to send or receive messages, such as ActiveMQ or MQSeries, requires first a QueueConnectionFactory, then a Queue object. The implementation of these interfaces changes with brokers, and are not provided by JQM - they must be provided with the payload or put inside the ext directory.

```
import javax.jms.Connection;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnectionFactory;
import javax.jms.Session;
import javax.jms.TextMessage;
```

(continues on next page)

(continued from previous page)

```
import javax.naming.spi.NamingManager;
import com.enioka.jqm.api.JobBase;

public class SuperTestPayload extends JobBase
{
        @Override
        public void start()
        {
                int nb = 0;
                try
                {
                        // Get the QCF
                        Object o = NamingManager.getInitialContext(null).lookup("jms/
↪qcf");
                        System.out.println("Received a " + o.getClass());

                        // Do as cast & see if no errors
                        QueueConnectionFactory qcf = (QueueConnectionFactory) o;

                        // Get the Queue
                        Object p = NamingManager.getInitialContext(null).lookup("jms/
↪testqueue");
                        System.out.println("Received a " + p.getClass());
                        Queue q = (Queue) p;

                        // Now that we are sure that JNDI works, let's write a message
                        System.out.println("Opening connection & session to the broker
↪");
                        Connection connection = qcf.createConnection();
                        connection.start();
                        Session session = connection.createSession(true, Session.AUTO_
↪ACKNOWLEDGE);

                        System.out.println("Creating producer");
                        MessageProducer producer = session.createProducer(q);
                        TextMessage message = session.createTextMessage("HOUBA HOP.␣
↪SIGNED: MARSUPILAMI");

                        System.out.println("Sending message");
                        producer.send(message);
                        producer.close();
                        session.commit();
                        connection.close();
                        System.out.println("A message was sent to the broker");
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
        }
}
```

4.3.4 Files

```
File f = InitialContext.doLookup("fs/superalias");
```

4.3.5 URL

```
URL f = InitialContext.doLookup("url/testurl");
```

4.4 Engine API

The engine API is an interface offered optionally to running job instances allowing them to interact with JQM.

It allows them to do some operations only available to running jobs (such as specifying that a file they have just created should be made available to end users) as well as a subset of operations coming directly from the *Full client API*. The latter is mostly for convenience - that way, clients do not have to import, set parameters and initialize the full API - everything is readied by the engine (and very quickly because the engine reuses some of its own already-initialized objects).

Using the API is easy: one just has to declare, inside the job main class, a Field of *JobManager* type. It can be static. Then, the JQM engine will inject an instance inside that field at runtime and it can be used without further ado.

class JobManager

This interface gives access to JQM engine variables and methods. It allows to retrieve the characteristics of the currently running job instances, as well as creating new instances and other useful methods. It should never be instantiated but injected by the JQM engine. For the injection to take place, the payload main class should have a field of type JobManager (directly or through inheritance, as well as public or private).

Use is very straightforward:

```
public class App implements Runnable
{
    private JobManager jm;

    @Override
    public void run()
    {
        // JM can be used immediately.
        jm.enqueue("otherjob", "me");
    }
}
```

4.4.1 Current job metadata

For the description of these items, please see the job instance description. Please note that these are methods, not fields - this is only because Java does not allow to specify fields inside an interface.

JobManager.**parentId**() → int

JobManager.**jobApplicationId**() → int

JobManager.**jobInstanceID**() → int

JobManager.**canBeRestarted**() → boolean

JobManager.**applicationName**() → String

JobManager.**sessionID**() → String

JobManager.**application**() → String

JobManager.**module**() → String

JobManager.**keyword1**() → String

JobManager.**keyword2**() → String

JobManager.**keyword3**() → String

JobManager.**userName**() → String

JobManager.**parameters**() → Map<String, String>

4.4.2 Enqueue & retrieve jobs

JobManager.**enqueue**(*String applicationName*, *String user*, *String mail*, *String sessionId*, *String application*, *String module*, *String keyword1*, *String keyword2*, *String keyword3*, *Map<String, String> parameters*) → int

Enqueues a new execution request. This is asynchronous - it returns as soon as the request was posted.

Equivalent to *JqmClient.enqueue()*, but where the parameters are given directly instead of using a *JobRequest* instance. This is a little ugly but necessary due to the underlying class loader proxying magic.

JobManager.**enqueueSync**(*String applicationName*, *String user*, *String mail*, *String sessionId*, *String application*, *String module*, *String keyword1*, *String keyword2*, *String keyword3*, *Map<String, String> parameters*) → int

Calls *enqueue()* and waits for the end of the execution.

JobManager.**waitChild**(*int jobInstanceId*) → void

JobManager.**waitChildren**() → void

JobManager.**hasEnded**(*int jobInstanceId*) → Boolean

JobManager.**hasSucceeded**(*int jobInstanceId*) → Boolean

JobManager.**hasFailed**(*int jobInstanceId*) → Boolean

4.4.3 Communications

JobManager.**sendMsg**(*String message*) → void

Messages are strings that can be retrieved during run by other applications, so that interactive human users may have a measure of a job instance progress. (typical messages highlight the job's internal steps)

JobManager.**sendProgress**(*Integer progress*) → void

Progress is an integer that can be retrieved during run by other applications, so that interactive human users may have a measure of a job instance progress. (typically used for percent of completion)

JobManager.**addDeliverable**(*String path*, *String fileLabel*) → int

When a file is created and should be retrievable from the client API, the file must be referenced with this method.

The file is moved by this method! Only call when you don't need the file any more.

It is strongly advised to use *getWorkDir()* to get a directory where to first create your files.

4.4.4 Misc.

`JobManager.`**`defaultConnect`**`()` → String
: The default connection JNDI alias. To retrieve a default connection, simply use:

```
((DataSource)InitialContext.doLookup(jm.defaultConnect)).getConnection();
```

See *JDBC* for more details.

Preferably use directly *`JobManager.getDefaultConnection()`* to directly retrieve a connection.

`JobManager.`**`getDefaultConnection`**`()` → Connection
: A connection as described by the default JNDI alias. See *JDBC* for more details.

`JobManager.`**`getWorkDir`**`()` → File
: If temp files are necessary, use this directory. The directory already exists. It is used by a single instance. It is purged at the end of the run.

`JobManager.`**`yield`**`()` → void
: This simply notifies the engine that it can briefly take over the thread, mostly to check if the thread should commit suicide. See *Going to the culling* for more details.

4.5 Packaging

JQM is able to load *payloads* from jar files (in case your code is actually inside a war, it is possible to simply rename the file), which gives a clear guidance as to how the code should be packaged. However, there are also other elements that JQM needs to run the code.

For example, when a client requests the *payload* to run, it must be able to refer to the code unambiguously, therefore JQM must know an "application name" corresponding to the code. This name, with other data, is to be put inside an XML file that will be imported by JQM - it's a deployment descriptor, akin to a web.xml or an ejb-jar.xml. A code can only run if its XML has been imported (or the corresponding values manually entered though the web administration console, or manually written inside the database, which by the way is a fully unsupported way to do it).

Should some terms prove to be obscure, please refer to the *Glossary*.

4.5.1 Libraries handling

JQM itself is hidden from the payloads - payloads cannot see any of its internal classes and resources. So JQM itself does not provide anything to payloads in terms of libraries (with the exception of libraries explicitly added to the ext directory, see below).

But there are two ways, each with two variants, to make sure that required libraries are present at runtime.

Note: All the four variants are exclusive. **Only one library source it used at the same time**.

Maven POM

A jar created with Maven always contains the pom.xml hidden inside META-INF. JQM will extract it, read it and download the dependencies, putting them on the payload's class path.

It is also possible to put a pom.xml file in the same directory as the jar, in which case it will have priority over the one inside the jar.

JQM uses the Maven 3 engine internally, so the pom resolution should be exactly similar to one done with the command line. It includes using your settings.xml. There a few *Parameters* that can tweak that behaviour.

Conclusion: in that case, no packaging to do.

Warning: using this means the pom is fully resolvable from the engine server (repository access, etc). This includes every parent pom used.

Warning: if you use a non-Maven system such as Gradle to create "uber jars" that include files from jars created with Maven, you may end up with a pom.xml inside the META-INF even if you are not using Maven! This would result in a Maven library resolution for a non-Maven jar and fail. To avoid this, exclude pom.xml files from your uber-ification. With Gradle:

```
jar { from {configurations.compile.collect { it.isDirectory() ? it : zipTree(it)}}
↪{ exclude "META-INF/maven/**" }}
```

lib directory

If using Maven is not an option (not the build system, no access to a Nexus/Maven central, etc), it is possible to simply put a directory named "lib" in the same directory as the jar file.

POM files are ignored if a lib directory is present. An empty lib directory is valid (allows to ignore a pom).

The lib directory may also be situated at the root of the jar file (lower priority than external lib directory).

Conclusion: in that case, libraries must be packaged.

4.5.2 Shared libraries

It is possible to copy jars inside the JQM_ROOT/ext directory. In that case, these resources will be loaded by a classloader common to all libraries and will be available to all payloads.

This should only be used very rarely, and is not to be considered in packaging. This exists mostly for shared JNDI resources such as JDBC connection pools. Note that a library in ext has priority over one provided by the payload (through Maven or lib directory).

Note: JQM actually makes use of this priority to always provide the latest version of the jqm-api to payloads. The API can therefore be referenced as a "provided" dependency if using Maven.

4.5.3 Pure Maven package

JQM also supports fetching your jobs through Maven. Of course, in that case, it means your artifacts must be published to a repository accessible to JQM (Maven Central, a local Nexus...).

To do this, inside your deployment descriptor, use `<path>groupid:artifactid:version</path>` and `<pathType>MAVEN</pathType>`.

JQM will do a standard Maven3 resolution. That includes using a local cache and the standard refresh policy on SNAPSHOT artifacts. See the Maven *Parameters* to change the standard behaviour.

4.5.4 Creating a JobDef

Structure

The full XSD is given inside the lib directory of the JQM distribution.

An XML can contain as many Job Definitions as needed. Moreover, a single jar file can contain as many payloads as needed, therefore there can be multiple job definitions with the same referenced jar file.

The general XML structure is this:

```
<jqm>
        <jar>
                <path>jqm-test-fibo/jqm-test-fibo.jar</path>
    <pathType>FS</pathType>

                <jobdefinitions>
                        <jobDefinition>
                                ...
                        </jobDefinition>
                        ... other job definitions ...
                </jobdefinitions>
        </jar>
        <jar>... as many jars as needed ...</jar>
</jqm>
```

Jar attributes

name	description
path	the path to the jar. It must be relative to the "repo" attribute of the nodes. (default is installdir/jobs) If pathType is 'MAVEN', it contains the Maven coordinates of the artifact containgin your payload
pathType	The meaning of the "path" attribute. If abse,nt, defaults to FS. Can be FS or MAVEN.

New in version 1.1.6: There used to be a field named "filePath" that was redundant. It is no longer used and should not be specified in new xmls. For existing files, the field is simply ignored so there is no need to modify the files.

JobDef attributes

All JobDefinition attributes are mandatory, yet the tag fields (keyword, module, . . .) can be empty.

All attributes are case sensitive.

name	description
name	the name that will be used everywhere else to designate the payload. (can be seen as the primary key).
description	a short description that can be reused inside GUIs
canBeRestarted	some payloads should never be almlowed to restarted after a crash
javaClassName	the fully qualified name of the main class of the payload (this is how JQM can launch a payload even without any jar manifest)
maxTimeRunning	currently ignored
application	An open classification. Not used by the engine, only offered to ease querying and GUI creation.
module	see above
keyword1	see above
keyword2	see above
keyword3	see above
highlander	if true, there can only be one running instance at the same time (and queued instances are consolidated)

It is also possible to define parameters, as key/value pairs. Note that it is also possible to give parameters inside the *Job Request* (i.e. at runtime). If a parameter specified inside the request has the same name as one from the *JobDef*, the runtime value wins.

There is an optional parameter named "queue" in which it is possible ot specify the name of the queue to use for all instances created from this job definition. If not specified (the default), JQM will use the default queue.

There are also two parameters to configure CL behavior: - childFirstClassLoader which offers an option to have a child first class loader for this job definition - hiddenJavaClasses which offer the possibility to hide Java classes from jobs using one or more regex to define classes to hide from the parent class loader

There is also an optional parameter named "specificIsolationContext", if set all job definitions using the same value (case sensitive) will share the same CL. If this parameter is set you need to ensure all job definitions with the same specificIsolationContext share the same configuration for childFirstClassLoader and hiddenJavaClasses. SHould you fail to ensure this only job instances with the same configuration as the first one executed (with same specific context) will be executed.

XML example

Other examples are inside the jobs/xml directory of the JQM distribution.

This shows a single jar containing two payloads.

```
<jqm>
        <jar>
                <path>jqm-test-fibo/jqm-test-fibo.jar</path>

                <jobdefinitions>
                        <jobDefinition>
                                <name>Fibo</name>
                                <description>Test based on the Fibonachi suite</
↪description>
                                <canBeRestarted>true</canBeRestarted>
                                <javaClassName>com.enioka.jqm.tests.App</
↪javaClassName>
                                <application>CrmBatchs</application>
```

(continues on next page)

(continued from previous page)

```
                                <module>Consolidation</module>
                                <keyword1>nightly</keyword1>
                                <keyword2>buggy</keyword2>
                                <keyword3></keyword3>
                                <highlander>false</highlander>
                                <parameters>
                                        <parameter>
                                                <key>p1</key>
                                                <value>1</value>
                                        </parameter>
                                        <parameter>
                                                <key>p2</key>
                                                <value>2</value>
                                        </parameter>
                                </parameters>
                        </jobDefinition>
                        <jobDefinition>
                                <name>Fibo2</name>
                                <description>Test to check the xml implementation</
↪description>
                                <canBeRestarted>true</canBeRestarted>
                                <javaClassName>com.enioka.jqm.tests.App</
↪javaClassName>
                                <application>ApplicationTest</application>
                                <module>TestModule</module>
                                <keyword1></keyword1>
                                <keyword2></keyword2>
                                <keyword3></keyword3>
                                <highlander>false</highlander>
                                <parameters>
                                        <parameter>
                                                <key>p1</key>
                                                <value>1</value>
                                        </parameter>
                                        <parameter>
                                                <key>p2</key>
                                                <value>2</value>
                                        </parameter>
                                </parameters>
                        </jobDefinition>
                </jobdefinitions>
        </jar>
</jqm>
```

Importing

The XML can be imported through the command line.

```
java -jar jqm.jar -importjobdef /path/to/xml.file
```

Please note that if your JQM deployment has multiple engines, it is not necessary to import the file on each node - only once is enough (all nodes share the same configuration). However, the jar file must obviously still be present on the nodes that will run it.

Also, jmq.ps1 or jqm.sh scripts have an "allxml" option that will reimport all xml found under JQM_ROOT/jobs and subdirectories.

4.6 Testing payloads

4.6.1 Unit testing

By unit testing, we mean here running a single payload inside a JUnit test or any other form of test (including a 'main' Java program) without needing a full JQM engine.

JQM provides a library named jqm-tst which allows tests that will run a **single job instance** in a stripped-down synchronous version of an embedded JQM engine requiring no configuration. The engine is destroyed immediately after the run. Single job instance also has for consequence that if your job enqueues new execution requests these will be ignored.

An example taken from JQM's own unit tests:

```
@Test
public void testOne()
{
    JobInstance res = com.enioka.jqm.test.JqmTester.create("com.enioka.jqm.test.
↪Payload1").addParameter("arg1", "testvalue").run();
    Assert.assertEquals(State.ENDED, res.getState());
}
```

Here, we just have to give the class of the payload, optionally parameters and that's it. The result returned is from the client API.

Refer to JqmTester javadoc for further details, including how to specify JNDI resource if needed.

4.6.2 Integration tests

If you have to test interactions between jobs (for example, one job instance queueing another), it may be necessary to use a full JQM engine. JQM provides another embedded tester class to do so. It is inside the same jqm-tst library.

These are the steps to follow to launch an integration test:

- create the tester object
- add at least on node (engine)
- add at least one queue
- deploy a queue on a node (i.e. set a node to poll a queue)
- start the engines
- create a job definition (the equivalent of the deployment descriptor XML file, which describes where the class to launch is, its parameters…)
- launch a new job instance and other normal JQM client interactions using the client API.
- stop the engines.

When using test frameworks like JUnit, all the node creation stuff is usually inside @BeforeClass methods, like in the following example.:

```
public class MyIntegrationTest
{
    public static JqmAsyncTester tester;

    @BeforeClass
```

(continues on next page)

(continued from previous page)

```
    public static void beforeClass()
    {
        // This creates a cluster with two JQM nodes, three queues (queue1 polled by node1, queue2 polled by node2, queue3 polled by both nodes).
        // The nodes are started at the end of this line.
        tester = JqmAsyncTester.create().addNode("node1").addNode("node2").addQueue("queue1").addQueue("queue2").addQueue("queue3")
                .deployQueueToNode("queue1", 10, 100, "node1").deployQueueToNode("queue2", 10, 100, "node2")
                .deployQueueToNode("queue3", 10, 100, "node1", "node2").start();
    }

    @AfterClass
    public static void afterClass()
    {
        // Only stop the cluster when all tests are done. This means there is no reboot or cleanup between tests if tester.cleanupAllJobDefinitions() is not explicitely called.
        tester.stop();
    }

    @Before
    public void before()
    {
        // A helper method to ensure there is no traces left of previous executions and job definitions from other tests
         tester.cleanupAllJobDefinitions();
    }

    @Test
    public void testOne()
    {
        // Quickly create a job definition from a class present in the test class path.
        tester.addSimpleJobDefinitionFromClasspath(Payload1.class);

        // Request a launch of this new job definition. Note we could simply use the JqmClient API.
        tester.enqueue("Payload1");

        // Launches are asynchronous, so wait for results (with a timeout).
        tester.waitForResults(1, 10000);

        // Actual tests
        Assert.assertEquals(1, tester.getOkCount());
        Assert.assertEquals(1, tester.getHistoryAllCount());
    }

    @Test
    public void testTwo()
    {
        // Quickly create a job definition from a class present in a jar. This is the way production JQM nodes really work - they load jar stored on the local file system.
        tester.addSimpleJobDefinitionFromLibrary("payload1", "App", "../jqm-tests/jqm-test-datetimemaven/target/test.jar")
```

(continues on next page)

(continued from previous page)

```
        tester.enqueue("payload1");
        tester.waitForResults(1, 10000, 0);

        Assert.assertTrue(tester.testCounts(1, 0));
    }
}
```

Refer to JqmAsyncTester javadoc for further details, including how to specify JNDI resource and retrieving files created by the job instances.

Note: the tester outputs logs on stdout using log4j. You can set set log level through a tester method. If you use other loggers, this may result in a mix of different logger outputs.

Warning: the nodes run inside the current JVM. So if you start too many nodes, or allow too many concurrent jobs to run, you may run out of memory and need to set higher JVM -Xmx parameters. If using Maven, you may for example set the environment variable *export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"*

4.7 Understanding the execution context

JQM has a basic promise: *your code runs as if it were running inside a standalone JVM*. That's all. If your code runs fine with java -jar my.jar (or My.class...), you are all set. Your code will never see anything from the engine (like the libraries the engine itself use - everything is fully hidden), nor from other jobs which may run at the same time. It really behaves as if a brand new JVM had been created just for your job instance (and a new one is created for each different launch).

This chapter is an advanced topic useful if you want to go beyond that and weaken the isolation.

4.7.1 The default mode: isolation

As written above, the default mode is "every launch is fully isolated". It works this way: a different class loader is created for each launch. It only has access to the classes inside the job (the job jar file, its optional "lib" directory and its optional Maven dependencies) and the "ext" directory, which contains libraries shared by all job definitions.

At the end of each launch, the class loader is garbage collected and never reused.

This means all libraries and classes are loaded on each and every launch. There is no static context which is kept between launches. This is exactly what happens when a user launches a program inside a JVM, and the JVM stops at the end of the execution.

Also, only the classes which are one of the supported job types (with a static main method, implementing Runnable or implementing JobBase) can run out of the box.

4.7.2 Changing the default mode

A few parameters can be set to change the default behaviour - i.e. the execution context of all jobs which do not request a specific execution context. Two modes are possible:

- a single shared execution context for all job definitions inside all jars
- one execution context for all jobs inside the same jar (therefore one execution context per jar file)

See the global parameters documentation for more details.

4.7.3 Advanced mode: context definition

It is actually possible to specify options defining the execution context inside the deployment descriptor (see *Packaging*).

Here is a full example, explained below:

```
<context>
        <name>MyPrettyContext</name>
        <childFirst>true</childFirst>
        <hiddenJavaClasses>java.maths.*</hiddenJavaClasses>
        <tracingEnabled>false</tracingEnabled>
        <persistent>true</persistent>

        <runners>com.enioka.jqm.tools.LegacyRunner,com.enioka.jqm.tools.MainRunner,
↪com.enioka.jqm.tools.RunnableRunner</runners>

        <eventHandlers>
                <handler>
                        <className>com.enioka.handlers.filterOne</className>
                        <event>JI_STARTING</event>
                        <parameters>
                                <parameter>
                                        <key>keyname</key>
                                        <value>value</value>
                                </parameter>
                                <parameter>
                                        <key>keyname2</key>
                                        <value>value2</value>
                                </parameter>
                        </parameters>
                </handler>
        </eventHandlers>
</context>
```

Inside the "context" tag the only compulsory information is "name". Everything else is optional and has default values.

A context is defined at the root level of the deployment descriptor. It can be used by any number of job definitions, by using the name of the context:

```
<jobDefinition>
        <name>Fibo</name>
        <description>Test based on the Fibonachi suite</description>
        <canBeRestarted>true</canBeRestarted>
        <javaClassName>com.enioka.jqm.tests.App</javaClassName>
        <application>CrmBatchs</application>
        <module>Consolidation</module>
        <keyword1>nightly</keyword1>
        <highlander>false</highlander>
        <executionContext>MyPrettyContext</executionContext>
        <parameters>
                <parameter>
                        <key>p1</key>
                        <value>1</value>
                </parameter>
```

(continues on next page)

(continued from previous page)

```
                <parameter>
                        <key>p2</key>
                        <value>2</value>
                </parameter>
        </parameters>
</jobDefinition>
```

(note the "executionContext" tag).

Class loading order

A normal JSE class loader is parent first - that is, if a class exists in a lower layer of the class loading hierarchy, it will be loaded even if your own jar provides a class of the same package + name.

For example, if you jar contains a java.util.String class, it will never be loaded as it defined in the JDK itself, the lowest level and therefore the highest priority.

Sometimes, you will want to give priority to your own classes. This is done by setting "childFirst" to "true". In that case, a class will be loaded from the lower levels only if not defined in your job (and its libraries).

A similar effect can be obtained by simply hiding classes, see next paragraph.

Default is "false" - meaning parent first.

Hiding Java classes

Changing the class loading loading priority is radical, sometimes you just want to override a small set of classes. To do that, just put a comma-separated list of regular expressions inside the "hiddenJavaClasses" tag. Classes which match at least one of the regular expressions will never ever be loaded from a source outside your own jar and libraries.

Default is no exclusions.

Class loading tracing

To debug "why isn't my library loaded" issues, you can enable a trace by setting the "tracingEnabled" parameter to "true". The trace is written in the log (and stdout if active).

Default is "false" - meaning disabled.

Context persistence

By default, the context is destroyed at the end of a run. This means there is no possibility to set anything static in a first run and retrieve it in a further job. While this is most often an excellent programming principle (no side effects possible!), it may be detrimental to some programs. For example, initializing a JPA provider such as Hibernate has a huge cost be it in memory or CPU cycles, which is why the JPA context (the EntityManagerFactory - EMF) is usually a shared static singleton. But as the context is thrown out at the end of each execution, with it goes the static context too, and the EMF has to be re-created on each run.

To avoid this, a context can be set as persistent. Just set "persistent" to "true". In that case the context will be created the first time it is needed, and kept forever afterwards.

Warning: enabling context persistence also means side effects become possible once again, as well as many other issues like some memory leaks which otherwise would just disappear with the context. To be enabled only by users who fully understand the implications!

Note: if a same context is referenced by multiple job definitions, and this context is persistent, it means that at runtime the same context is used by multiple job instances coming from different job definitions! This is often what is desired - sharing a static context between multiple job types. But it of course also increases the risk of unforeseen side effects.

The default is "true" when a context is specified. If a job definition is not associated with a specific context, the default is false.

Runners

The runners are the agents responsible for actually launching the job instances. The example above actually give the default value, which is a comma-separated list of the three runners corresponding to the three different types of supported job definitions:

- com.enioka.jqm.tools.LegacyRunner runs any class which implements the "JobBase" interface
- com.enioka.jqm.tools.MainRunner runs any class with a "static main" method
- com.enioka.jqm.tools.RunnableRunner runs any class which implements the Runnable interface and has a default no arguments constructor.

This list allows to restrict the job types available inside the context.

Note that the runners only exist to define "how to start" a job instance. They cannot do more, and they actually run in a very limited bubble with only access to themselves and the JDK.

Event handlers

A common requirement is to be able to run code at different times in the life cycle of a job instance. JQM allows this for one type of event, when a job instance is about to start.

The handlers run in the same context as the job instance itself. It means the class of the handler is inside the class path of the job instance itself. It is the responsibility of the developer to check there are no conflicts between his own code and the handler code.

The handler parameters are key/value pairs, with unique keys.

Warning: handlers are provided by the job definition itself, not by the engine. They MUST be present inside the available libraries (be it from a Maven dependency, a jar inside the "lib" directory, inside the über-jar...)

For an example of the use of an interpretor in the context of a Spring application, see *Using Spring* where one is used to bootstrap the Spring context (much like when a listener is often used when dealing with Spring in a servlet container).

4.8 Using Spring

This gives a rundown on how to efficiently use Spring inside JQM. This can of course be an inspiration for other big "container" frameworks.

There are multiple possibilities, and this page shows how to use two of them. They are presented here in order of increasing complexity, which is also the order of decreasing recommendation.

4.8.1 By doing nothing special

It has been said before, by default launching a new job instance in a JQM server is like launching a new JVM: if a Spring job already works from the command line, it will work in JQM without adaptation.

There are different ways to create Spring programs, but they all boil down to: create a Spring context, load configuration inside the context, create the job bean from the context and launch it.

A most common example is by using Spring Boot, which hides most boilerplate code. The main method is simply:

```
import org.springframework.boot.SpringApplication;

public class Application
{
        public static void main(String[] args)
        {
                SpringApplication.run(MyJob.class, args);
        }
}
```

And the job implements CommandLineRunner, which will automatically instantiate the bean and run it on context creation:

```
@Import(ContextConfig.class)
@SpringBootApplication
public class MyJobClass implements CommandLineRunner
{
        @Autowired
        private MyService myServiceToInject;

        @Override
        public void run(String... args) throws Exception
        {
                myServiceToInject.doSomething();
                System.out.println("Job is done!");
        }
}
```

In terms of job definition, the Application class is the JQM entry point. JQM knows nothing about Spring, it is just another main method to run.

Advantages:

- direct code reuse from CLI batch jobs
- just another job definition - nothing special to do
- free to initialize and configure Spring in any way: annotations, XML, packages to scan or ignore...

Cons:

- the Spring context is recreated on each launch, which is costly.

This is the recommended way of using Spring inside JQM, in the "keep it simple" philosophy.

Note: a full working sample is included inside the JQM integration tests. It is named "jqm-test-spring-1". (it also uses JPA with a JNDI resource handled by the JQM JNDI directory)

4.8.2 By having JQM set the context

In this option, there is only one Spring context for all job definitions using Spring. The jobs themselves (payload code) do no Spring context initialization - they just use Spring features (injection...) and do not care where they do come from.

This option is the direct equivalent of what happens inside a servlet container (Tomcat...) when using Spring: the context is actually initialized by a servlet initialization listener, and the application code just uses Spring, never creating a SpringContext itself.

JQM uses the same method, with an event handler. It also has a specialized runner which retrieves the job bean from the Spring context and runs it (it must implement Runnable).

The payload can be defined like this:

```
package com.compagny.project;

@Component
public class MyJobClass implements Runnable
{
        @Autowired
        private MyService myServiceToInject;

        @Resource(name = "runtimeParameters")
        private Map<String, String> parameters;

@Resource
private JobManagerProvider jmp;

        @Override
        public void run()
        {
                myServiceToInject.doSomething();
                System.out.println("Job " + jmp.getObject().jobInstanceID() + " is␣
↪done!");
        }
}
```

and there is no need for an encapsulation class like the Application class of the previous methods: JQM directly runs the job bean.

It is necessary to add the handler and runner to the execution context inside the deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<jqm>
        <jar>
                <path>directory/springjobs.jar</path>
                <jobdefinitions>
                        <jobDefinition>
                                <name>FirstJob</name>
                                <description>Does something</description>
                                <canBeRestarted>true</canBeRestarted>
```

(continues on next page)

(continued from previous page)

```
                                <javaClassName>com.compagny.project.MyJobClass</
↪javaClassName>
                                <module>BatchJobs</module>
                                <highlander>false</highlander>
                                <executionContext>MainSharedSpringContext</
↪executionContext>
                        </jobDefinition>
                </jobdefinitions>
        </jar>

        <context>
                <name>MainSharedSpringContext</name>
                <childFirst>false</childFirst>
                <hiddenJavaClasses></hiddenJavaClasses>
                <tracingEnabled>false</tracingEnabled>
                <persistent>true</persistent>

                <runners>com.enioka.jqm.runner.spring.AnnotationSpringRunner</runners>
                <eventHandlers>
                        <handler>
                                <className>com.enioka.jqm.handler.
↪AnnotationSpringContextBootstrapHandler</className>
                                <event>JI_STARTING</event>
                                <parameters>
                                        <parameter>
                                                <key>additionalScan</key>
                                                <value>com.compagny.project</value>
                                        </parameter>
                                </parameters>
                        </handler>
                </eventHandlers>
        </context>
</jqm>
```

The handler will intercept the "job instance is starting" event and initialize if needed an AnnotationConfigApplicationContext. All parameters are optional:

- additionalScan: a set of base packages to scan for annotations. Example: com.compagny.project,com.compagny.otherpackage
- beanNameGenerator: a fully qualified class implementing the BeanNameGenerator interface with a no-args constructor to use for creating the names of the beans
- contextDisplayName: name of the context in the logs
- contextId: id of the context bean
- allowCircularReferences: if "true", the context will allow circular references.

If no parameters are given, the job class (the first one to run) itself will be added to the Spring context, so if is a @Configuration it will be enabled.

The handler must be present in the job dependencies. In this case, it is provided with JQM, and the artifact is "com.enioka.jqm:jqm-runner-spring:${jqmversion}"

Warning: it is not possible to extend a Spring context after it has been initialized ("refreshed" in Spring-talk). So you must take care to put all your jobs inside the same class path so they are present during the initial scan. Basically, it means packaging all jobs in a single jar (or a single jar + set of libs). As this is the most common

packaging method in the JQM ecosystem (since it is the simplest), and as the Spring world very often uses über-jars, this should not be seen as a huge limitation.

Note we have only selected a single runner, which is provided by JQM: com.enioka.jqm.runner.spring.Runner. Depending on your needs, you may want to add the other runners (if you do not launch only Spring jobs in the same execution context for example).

Finally you may have noted in the sample that we had a @Resource(name = "runtimeParameters") Map: the runner actually registers a named bean to allow access to the job instance parameters through the Spring APIs. This bean is scoped on the thread, so you'll obviously get different values in different job instances even if they run at the same time in the same runtime context. If you need the full engine API, inject a JobManagerProvider as in the sample. This is a factory/provider, not a direct injection because a Spring context creates all non-lazy beans during context creation - and obviously the different parameters of a job contained by the JobManager can be set after that. So the provider is just a means to force lazy initialization.

Note: a full working sample is included inside the JQM integration tests. It is named "jqm-test-spring-2". It's deployment descriptor is named "xmlspring.xml".

CHAPTER 5

Client development

5.1 Introduction

A "client" is an external agent (Java program, shell script, ...) that needs to interact with the root function of JQM: job queueing and execution[1]. JQM offers multiple ways to do so, each being tailored to a specific type of client.

- a minimal web service API with very simple signatures, designed for scripts and the like, called the **simple API**
- **a full client API designed for more evolved programs. It is a superset of the minimal API (and actually directly reuses som**
  - a set of (language agnostic) REST web-services
  - a direct-to-database JPA2 implementation
- a minimal command line utility (**CLI**)
- for payloads running inside a JQM engine only, it is also possible to access a subset of the full client API as exposed through an object injected by the engine. it is called the **engine API**.

5.2 Simple web API

This is the strict minimum to allow easy wget/curl integration. Most notably, this is what will be usually used for integration with a job scheduler like Orsyp $Universe or BMC ControlM.

It allows:

- Submitting a job execution request (returns the query ID)
- Checking the status of a request (identified by its ID)
- Retrieving the logs created by an ended request
- Retrieving the files (PDF reports, etc) created by an ended request

[1] Therefore, all administrative functions (restart a JQM engine, modify a job parameter, ...) are fully excluded from this section. They are detailed inside a dedicated section.

Note: this API should never be used directly from a Java program. The more complete client APIs actually encapsulate the simple API in a Java-friendly manner.

Please refer to the following examples in PowerShell for the URLs to use (adapt DNS and port according to your environment. If you don't know these parameters, they are inside the NODE definitions and written at startup inside the server log). Also, these examples assume authentication is disabled (option -Credential should be used otherwise).

```
# Get the status of job instance 1035
PS> Invoke-RestMethod http://localhost:61260/ws/simple/status?id=1035
ENDED

# Get stdout of jobinstance 1035
PS> Invoke-RestMethod http://localhost:61260/ws/simple/stdout?id=1035
LOG LINE

# Get stderr of job instance 1035 (void log as a result in this example)
PS> Invoke-RestMethod http://localhost:61260/ws/simple/stderr?id=1035

# Get file which ID is 77a73e85-e2b6-4e89-bb07-f7097b17e532
# This ID cannot be guessed or retrieved through the simple API - this method mostly␣
↪exist for the full API to call.
# It is documented here for completion sake only.
PS> Invoke-RestMethod http://localhost:61260/ws/simple/file?id=77a73e85-e2b6-4e89-
↪bb07-f7097b17e532
The first line
The second line

# Enqueue a new execution request. Returned number is the ID of the request (the ID␣
↪to use with the other methods)
PS> Invoke-RestMethod http://localhost:61260/ws/simple/ji -Method Post -Body @
↪{applicationname="DemoApi"}
1039

# Same, but with parameters
PS> Invoke-RestMethod http://localhost:61260/ws/simple/ji -Method Post -Body @
↪{applicationname="DemoApi";parameterNames="p1";parameterValues="eee"}
1047
```

5.3 Full client API

The **client API** enables any program (in Java, as well as in any other languages for some implementations of the API) to interact with the very core function of JQM: asynchronous executions. This API exposes every common method pertaining to this goal: new execution requests, checking if an execution is finished, listing finished executions, retrieving files created by an execution...

It is named "client API" because it contains the methods that are often directly exposed to human end-users. They may, for example, have a web-based GUI with buttons such as "I want to run that report", "let's synchronize invoices with accounting", ... which will in turn submit a job execution request to JQM. This client API contains such a submission method, as well as all the others the end user may need, such as "is my job done", "cancel this job", and so on. And obviously, what is true for human clients is also true for automated systems - for example, a scheduler may use this API (even if the *Simple web API* may be better suited for this).

5.3.1 Basics

The client API is defined an a Java interface, and has two implementations. Therefore, to use the client API, one of its two implementations must be imported: either *the Hibernate JPA 2.0 one* with jqm-api-client-hibernate.jar or *the web service client* with jqm-api-client-jersey.jar.

Then it is simply a matter of calling:

```
JqmClientFactory.getClient();
```

The client returned implements an interface named *JqmClient*, which is profusely documented in JavaDoc form, as well as in the *next section*. Suffice to say that it contains many methods related to:

- queueing new execution requests
- removing requests, killing jobs, pausing waiting jobs
- modifying waiting jobs
- querying job instances along many axis (is running, user, ...)
- get messages & advancement notices
- retrieve files created by jobs executions
- some metadata retrieval methods to ease creating a GUI front to the API

For example, to list all executions known to JQM:

```
List<JobInstance> jobs = JqmClientFactory.getClient().getJobs();
```

Now, each implementation has different needs as far as configuration is concerned. Basically, Hibernate needs to know how to connect to the database, and the web service must know the web service server. To allow easy configuration, the following principles apply:

1. Each client provider can have one (always optional) configuration file inside the classpath. It is specific for each provider, see their doc
2. It is possible to overload these values through the API **before the first call to getClient**:

   ```
   Properties p = new Properties();
   p.put("com.enioka.jqm.ws.url", "http://localhost:9999/marsu/ws");
   JqmClientFactory.setProperties(p);
   List<JobInstance> jobs = JqmClientFactory.getClient().getJobs();
   ```

3. An implementation can use obvious other means. E.g. Hibernate will try JNDI to retrieve a database connection.

The name of the properties depends on the implementation, refer to their respective documentations.

Please note that all implementations are supposed to cache the *JqmClient* object. Therefore, it is customary to simply use JqmClientFactory.getClient() each time a client is needed, rather than storing it inside a local variable.

For non-Java clients, use the *web service API* which can be called from anywhere.

Finally, JQM uses unchecked exception as most APIs should (see this article). As much as possible (when called from Java) the API will throw:

- `JqmInvalidRequestException` when the source of the error comes from the caller (inconsistent arguments, null arguments, ...)
- `JqmClientException` when it comes from the API's internals - usually due to a misconfiguration or an environment issue (network down, etc).

5.3.2 Client API details

The JqmClient interface

class JqmClient

This interface contains all the necessary methods to interact with JQM functions.

All methods have detailed Javadoc. The Javadoc is available on Maven Central (as are the binaries and the source code). This paragraph gives the methods prototypes as well as how they should be used. For details on exceptions thrown, etc. please refer to the javadoc.

New execution requests

JqmClient.**enqueue**(*JobRequest executionRequest*) → integer

The core method of the Job Queue Manager: it enqueues a new job execution request, as described in the object parameter. It returns the ID of the request. This ID will be kept throughout the life cycle of the request until it becomes the ID of the history item after the execution ends. This ID is reused in many other methods of the API.

It consumes a *JobRequest* item, which is a "form" object in which all ncessary parameters can be specified.

JqmClient.**enqueue**(*String applicationName*, *String user*) → integer

A simplified version of the method above.

JqmClient.**enqueueFromHistory**(*Integer jobIdToCopy*) → integer

This method copies an ended request. (this creates a new request - it has no impact whatsoever on the copied request)

Job request deleting

JqmClient.**cancelJob**(*Integer id*) → void

When called on a waiting execution request, removes it from the queue and moves it to history with CANCELLED status. This is the standard way of cancelling a request.

Synchronous method

JqmClient.**deleteJob**(*int id*) → void

This method should not usually be called. It completely removes a job execution request from the database. Please use cancelJob instead.

Synchronous method

JqmClient.**killJob**(*int id*) → void

Attempts to kill a running job instance. As Java thread are quite hard to kill, this may well have no effect.

Asynchronous method

Pausing and restarting jobs

JqmClient.**pauseQueuedJob**(*int id*) → void

When called on a job execution request it is ignored by engines and status forever in queue.

`JqmClient.`**`resumeJob`**`(`*int id*`)` → void
: Will re insert a paused execution request into the queue. The place inside the queue may change from what it used to be before the pause.

`JqmClient.`**`restartCrachedJob`**`(`*int id*`)` → int
: Will create an execution request from a crashed history element and *remove all traces of the failed execution**.

Queries on Job instances

The API offers many methods to query either ended jobs or waiting/running ones. When there is a choice, please use the method which is the mst specific to your needs, as it may have optimizations not present in the more general ones.

`JqmClient.`**`getJob`**`(`*int id*`)` → JobInstance
: Returns either a running or an ended job instance.

`JqmClient.`**`getJobs`**`()` → List<JobInstance>
: Returns all job instances.

`JqmClient.`**`getActiveJobs`**`()` → List<JobInstance>
: Lists all waiting or running job instances.

`JqmClient.`**`getUserActiveJobs`**`(`*String username*`)` → List<JobInstance>
: Lists all waiting or running job instances which have the given "username" tag.

`JqmClient.`**`getJobs`**`(`*Query q*`)` → List<JobInstance>
: please see *Query API*.

Quick access helpers

`JqmClient.`**`getJobMessages`**`(`*int id*`)` → List<String>
: Retrieves all the messages created by a job instance (ended or not)

`JqmClient.`**`getJobProgress`**`(`*int id*`)` → int
: Get the progress indication that may have been given by a job instance (running or done).

Files & logs retrieval

`JqmClient.`**`getJobDeliverables`**`(`*int id*`)` → List<Deliverable>
: Return all metadata concerning the (potential) files created by the job instance: Excel files, PDFs, ... These are the files explicitly referenced by the job instance through the *`JobManager.addDeliverable()`* method.

`JqmClient.`**`getDeliverableContent`**`(`*Deliverable d*`)` → InputStream
: The actual content of the file described by the `Deliverable` object.

  This method, in all implementations, uses a direct HTTP(S) connection to the engine that has run the job instance.

  The responsibility to close the stream lies on the API user

`JqmClient.`**`getDeliverableContent`**`(`*int deliverableId*`)` → InputStream
: Same a above.

`JqmClient.`**`getJobDeliverablesContent`**`(`*int jobId*`)` → List<InputStream>
: Helper method. A loop on *`getDeliverableContent()`* for all files created by a single job instance.

`JqmClient.`**`getJobLogStdOut`**`(`*int jobId*`)` → InputStream
Returns the standard output flow of of an ended job instance.

This method, in all implementations, uses a direct HTTP(S) connection to the engine that has run the job instance.

The responsibility to close the returned stream lies on the API user

`JqmClient.`**`getJobLogStdErr`**`(`*int jobId*`)` → InputStream
Same as *`getJobLogStdOut()`* but for standard error flow.

Referential queries

These methods allow to retrieve all the referential data that may be needed to use the other methods: queue names, application names, etc.

`JqmClient.`**`getQueues`**`()` → List<Queue>

`JqmClient.`**`getJobDefinitions`**`()` → List<JobDef>

`JqmClient.`**`getJobDefinition`**`(`*String applicationName*`)` → JobDef

API objects

JobRequest

`class JobRequest`
Job execution request. It contains all the data needed to enqueue a request (the application name), as well as non-mandatory data. It is consumed by *`JqmClient.enqueue()`*.

Basically, this is the form one has to fill in order to submit an execution request.

Queue

`class Queue`
All the metadata describing a *queue*. Read only element.

Please note there is another queue class that exists within JQM, inside the com.enioka.jqm.jpa packages. The JPA one is an internal JQM class and should not be confused with the API one, which is a stable interface.

JobDef

`class JobDef`
All the metadata describing a *job definition*. Read-only element.

Please note there is another class with this name that exists within JQM, inside the com.enioka.jqm.jpa packages. The JPA one is an internal JQM class and should not be confused with the API one, which is a stable interface.

Example

```
# Enqueue a job
int i = JqmClientFactory.getClient().enqueue("superbatchjob");

# Get its status
Status s = JqmClientFactory.getClient().getStatus(i);

# If still waiting, cancel it
if (s.equals(State.WAITING))
    JqmClientFactory.getClient().cancel(i);
```

5.3.3 Query API

The query API is the only part of the client API that goes beyond a simple method call, and hence deserves a dedicated chapter. This API allows to easily make queries among the past and current *job instance* s, using a fluent style.

Basics, running & filtering

To create a `Query`, simply do *Query.create()*. This will create a query without any predicates - if run, it will return the whole execution history.

To add predicates, use the different `Query` methods. For example, this will return every past instance for the *job definition* named JD:

```
Query.create().setApplicationName("JD");
```

To create predicates with wildcards, simply use "%" (the percent sign) as the wildcard. This will return at least the results of the previous example and potentially more:

```
Query.create().setApplicationName("J%");
```

To run a query, simply call run() on it. This is equivalent to calling *JqmClientFactory.getClient().getJobs(Query q)*. Running the previous example would be:

```
List<JobInstance> results = Query.create().setApplicationName("J%").run();
```

Querying live data

By default, a Query only returns instances that have ended, not instances that are inside the different *queues*. This is for performance reasons - the queues are the most sensitive part of the JQM database, and live in different tables than the History.

But it is totally supported to query the queues, and this behaviour is controlled through two methods: Query.setQueryLiveInstances (default is false) and Query.setQueryHistoryInstances (default is true). For example, the following will query only the queues and won't touch the history:

```
Query.create().setQueryLiveInstances(true).setQueryHistoryInstances(false).setUser(
↪"test").run();
```

Note: When looking for instances of a desired state (ENDED, RUNNING, ...), it is highly recommended to query only the queue or only the history. Indeed, states are specific either to the queue or to history: an ended instance is

always in the history, a running instance always in the queue, etc. This is far quicker than querying both history and queues while filtering on state.

Pagination

The history can grow very large - it depends on the activity inside your cluster. Therefore, doing a query that returns the full history dataset would be quite a catastrophy as far as performance is concerned (and would probably fail miserably out of memory).

The API implements pagination for this case, with the usual first row and page size.

```
Query.create().setApplicationName("JD").setFirstRow(100000).setPageSize(100).run();
```

Warning: failing to use pagination on huge datasets will simply crash your application.

Pagination cannot be used on live data queries - it is supposed there are never more than a few rows inside the queues. Trying to use it nevertheless will trigger an JqmInvalidRequestException.

Sorting

The most efficient way to sort data is to have the datastore do it, especially if it is an RDBMS like in our case. The Query API therefore allows to specify sort clauses:

```
Query.create().setApplicationName("J%").addSortAsc(Sort.APPLICATIONNAME).
↪addSortDesc(Sort.DATEATTRIBUTION);
```

The two addSortxxx methods must be called in order of sorting priority - in the example above, the sorting is first by ascending application name (i.e. batch name) then by descending attribution date. The number of sort clauses is not limited.

Please note that sorting is obviously respected when pagination is used.

Shortcuts

A few methods exist in the client API for the most usual queries: running instances, waiting instances, etc. These should always be used when possible instead of doing a full Query, as the shortcuts often have optimizations specific to their data subset.

Sample

JQM source code contains one sample web application that uses the Query API. It is a J2EE JSF2/Primefaces form that exposes the full history with all the capabilities detailed above: filtering, sorting, pagination, etc.

It lives inside jqm-all/jqm-webui/jqm-webui-war.

Note: this application is but **a sample**. It is not a production ready UI, it is not supported, etc.

5.3.4 JPA Client API

Client API is the name of the API offered to the end users of JQM: it allows to interact with running jobs, offering operations such as creating a new execution request, cancelling a request, viewing all currently running jobs, etc. Read *client API* before this chapter, as it gives the definition of many terms used here as well as the general way to use clients.

JQM is very database-centric, with (nearly) all communications going through the database. It was therefore logical for the first client implementation to be a direct to database API, using the same ORM named Hibernate as in the engine.

Note: actually, even if this API uses a direct connection to the database for nearly everything, there is one API method which does not work that way: file retrieval. Files produced by job instances (business files or simply logs) are stored locally on each node - therefore retrieving these files requires connecting directly (HTTP GET) to the nodes. Therefore, talk of HTTP connection parameters should not come as a surprise.

Parameters

The API uses a JPA persistence unit to interact to the database. Long story short, it needs a JNDI resource named jdbc/jqm to connect to the database. This name can be overloaded.

It is possible to overload persistence unit properties either:

- (specific to this client) with a jqm.properties file inside the META-INF directory
- (as for every other client)by using Java code, before creating any client:

```
Properties p = new Properties();
p.put("javax.persistence.nonJtaDataSource", "jdbc/houbahop");
JqmClientFactory.setProperties(p);
```

The different properties possible are JPA2 properties (http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-eval-oth-JSpec/) and Hibernate properties (http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch03.html#configuration-optional). The preceding example changed (or set in the first place) the <non-jta-datasource> to some JNDI alias. Dafault is jdbc/jqm.

If the file retrieval abilities are used, some connection data may also be provided through the same systems when SSL is used:

- com.enioka.jqm.ws.truststoreFile: in case SSL is used, this will be the trustStore to use. Default is: system trust store (inside Java installation).
- com.enioka.jqm.ws.truststoreType: same as above - type of the store. Default is JKS.
- com.enioka.jqm.ws.truststorePass: same as above. Default is empty.

There is no need to specify user/passwords/certificate even if API authentication is enabled as the API will grant itself permissions inside the database. (see *Data security*)

Libraries

In Maven terms, only one library is needed:

```
<dependency>
        <groupId>com.enioka.jqm</groupId>
        <artifactId>jqm-api-client-hibernate</artifactId>
        <version>${jqm.version}</version>
</dependency>
```

If the file retrieval APIs are not used, it is possible to remove one library with itself a lot of dependencies from the API. In Maven terms:

```
<dependency>
        <groupId>com.enioka.jqm</groupId>
        <artifactId>jqm-api-client-hibernate</artifactId>
        <version>${jqm.version}</version>
        <exclusions>
                <exclusion>
                        <groupId>org.apache.httpcomponents</groupId>
                        <artifactId>httpclient<artifactId>
                <exclusion>
        </exclusions>
</dependency>
```

Logs

The API uses slf4j to log information. It only provides slf4j-api, without any implementation to avoid polluting the user's class path. Therefore, out of the box, the only log it will ever create is a warning on startup that an implementation is required in order to view log messages.

If logs are needed, an implementation must be provided (such as slf4j-log4j12) and configured to retrieve data from classes in the *com.enioka.jqm* namespace.

For example, this may be used as an implementation:

```
<dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
                <version>${slf4j.version}</version>
</dependency>
```

and then the following log4j configuration file will set reasonable log levels on the console standard output:

```
# define the console appender
log4j.appender.consoleAppender = org.apache.log4j.ConsoleAppender

# now define the layout for the appender
log4j.appender.consoleAppender.layout = org.apache.log4j.PatternLayout
log4j.appender.consoleAppender.layout.ConversionPattern=%d{dd/MM HH:mm:ss.SSS}|%-5p|%-
↪40.40t|%-17.17c{1}|%x%m%n

# now map our console appender as a root logger, means all log messages will go to␣
↪this appender
log4j.rootLogger = INFO, consoleAppender
log4j.logger.com.enioka.jqm = INFO
```

Container integration

There may be nefast interactions between the persistence unit contained inside the API and the rest of the environment.

In a JNDI-enabled container without other JPA use

Hypothesis:

- deployment inside an EE6 container such as WebSphere, JBoss, Glassfish, or deployment inside a JSE container with JNDI abilities (Tomcat, **JQM itself**, ...)
- There is no use of any JPA provider in the application (no persistence.xml)

In this case, using the API is just a matter of providing the API as a dependency, plus the Hibernate implementation of your choice (compatible with 3.5.6-Final onwards to 4.2.x).

Please note that if your container provides a JPA2 provider by itself, there is obviously no need for providing a JPA2 implementation but beware: this client is **only compatible with Hibernate**, not OpenJPA, EclipseLink/TopLink or others. So if you are provided another provider, you may need to play with the options of your application server to replace it with Hibernate. This has been tested with WebSphere 8.x and Glassfish 3.x. JBoss comes with Hibernate. If changing this provider is not possible or desirable, use the *Web Service Client API* instead.

Then it is just a matter of declaring the JNDI alias "jdbc/jqm" pointing to the JQM database (refer to your container's documentation) and the API is ready to use. There is no need for parameters in this case (everything is already declared inside the persistence.xml of the API).

With other JPA use

> **Warning:** this paragraph is not needed for recent versions of Hibernate (4.x) as they extend the JPA specification by allowing multiple persistence units. Therefore, only the previous paragraph applies.

Hypothesis:

- deployment inside an EE6 container such as WebSphere, JBoss, Glassfish, or deployment inside a JSE container with JNDI abilities (Tomcat, **JQM itself**, ...), or no JNDI abilities (plain Sun JVM)
- There is already a persistence.xml in the project that will use the client API

This case is a sub-case of the previous paragraph - so first thing first, everything stated in the previous paragraph should be applied.

Then, an issue must be solved: there can only be (as per JPA2 specification) one persistence.xml used. The API needs its persistence unit, and the project using the client needs its own. So we have two! The classpath mechanisms of containers (servlet or EE6) guarantee that the persistence.xml that will be used is the one from the caller, not the API. Therefore, it is necesseray to redeclare the JQM persistence unit inside the final persistence.xml like this:

```
<persistence-unit name="jobqueue-api-pu">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <non-jta-data-source>jdbc/jqm2</non-jta-data-source>

        <jar-file>../jqm-model/target/jqm-model-VERSION.jar</jar-file>

        <properties>
                <property name="javax.persistence.validation.mode" value="none" />
        </properties>
</persistence-unit>

<persistence-unit name="whatever-pu-needed-by-your-application">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <non-jta-data-source>jdbc/test</non-jta-data-source>
```

(continues on next page)

(continued from previous page)

```
        <class>jpa.Entity</class>
</persistence-unit>
```

Note the use of "jar-file" to reference a jar containing a declared persistence unit. The name of the persistence unit must always be "jobqueue-api-pu". The **file path inside the jar tag must be adapted to your context and packaging, as well as JQM version**. The non-jta-datasource alias can be named anything you want (you may even want to redefine completely the datasource here, not using JNDI - see the Hibernate reference for the properties to set to do so).

Warning: the use of the <jar-file> tag is only allowed if the application package is an ear file, not a war.

Making it work with both Tomcat and Glassfish/WebSphere

Servlet containers such as Tomcat have a different way of handling JNDI alias contexts than full JEE containers. Basically, a developper would use java:/comp/env/jdbc/datasource inside Tomcat and simply jdbc/datasource in Glassfish. JQM implements a hack to make it work anyway in both cases. To enable it, it is compulsory to specify the JNDI alias inside the configuration file or inside the Properrty object, just like above.

TL;DR: to make it work in both cases, don't write anything specific inside your web.xml and use this in your code before making any API call:

```
Properties p = new Properties();
p.put("javax.persistence.nonJtaDataSource", "jdbc/jqm");
JqmClientFactory.setProperties(p);
```

5.3.5 Web Service Client API

Client API is the name of the API offered to the end users of JQM: it allows to interact with running jobs, offering operations such as creating a new execution request, cancelling a request, viewing all currently running jobs, etc. Read *client API* before this chapter, as it gives the definition of many terms used here as well as the general way to use clients.

The main client API is the Hibernate Client API, which runs directly against the JQM central database. As JQM is database centric, finding jobs to run by polling the database, this is most efficient. However, this API has a serious drawback: it forces the user of the API to use Hibernate. This can be a huge problem in EE6 applications, as most containers (WebSphere, Glassfish, JBoss...) offer their own implementation of the JPA standard which is not compatible with Hibernate and cannot coexist with it (there must be only one JPA implementation at the same time, and a database created for Hibernate is very difficult to reuse in another JPA provider). Moreover, even outside the EE6 field, the client may already have chosen a JPA implementation that is not Hibernate. This is why JQM also offers an optional **REST Web Service Client API**.

Client side

There are two ways to use the WS Client API:

- Using the Java client
- Directly using the web service

Using the Java client

This is a standard client API implementing the JqmClient interface. Like all clients, it is used by putting its jar on your classpath. The client uses the JAX-RS 2 API, so it needs an implementation such as Jersey (obviously not provided, the one provided by the container will be used).

For Maven users:

```
<dependency>
        <groupId>com.enioka.jqm</groupId>
        <artifactId>jqm-api-client-jersey</artifactId>
        <version>${jqm.version}</version>
</dependency>
```

and then using the API:

```
JqmClient jqm = JqmClientFactory.getClient();
```

As with any client (see the JavaDoc) clients are cached by the API, so it is not necessary to cache them yourself.

Interrogating the API is then also exactly the same as with any other client. For example, to list all running jobs:

```
JqmClientFactory.getClient().getJobs()
```

The specific parameters are:

Name	Compulsory	Description	Example
com.enioka.jqm.ws.url	YES	The base URL of the web service	http://localhost:1789/api/ws
com.enioka.jqm.ws.login		If auth is used only.	mylogin
com.enioka.jqm.ws.password	if login		password
com.enioka.jqm.ws.keystoreFile	if CSA	Store for client certificates authentication	./conf/client.pfx
com.enioka.jqm.ws.keystoreType		Type of the previous store	PKCS12
com.enioka.jqm.ws.keystorePass		Password of the store	MyPassword
com.enioka.jqm.ws.truststoreFile	if SSL	Trust roots for server certificates	./conf/client.pfx
com.enioka.jqm.ws.truststorePass		Password of the store	NoPassword

and can be set:

- (specific to this client) with a jqm.properties file inside the META-INF directory
- (as for every other client) using Java code, before creating any client:

```
Properties p = new Properties();
p.put("com.enioka.jqm.ws.url", "http://localhost:9999/marsu/ws");
JqmClientFactory.setProperties(p);
```

- through a system parameter (-Dcom.enioka.jqm.ws.url=http://...)

Interrogating the service directly

The previous client is only a use of the JAX-RS 2 API. You can also create your own web service proxy by interrogating the web service with the library of your choice (including the simple commons-http). See the *Service reference* for that.

Should that specific implementation need the interface objects, they are present in the jqm-api-client jar (the pure API jar without any implementation nor dependencies).

```
<dependency>
        <groupId>com.enioka.jqm</groupId>
        <artifactId>jqm-api-client</artifactId>
        <version>${jqm.version}</version>
</dependency>
```

Choosing between the two approaches

When using Java, the recommended approach is to **use the provided client**. This will allow you to:

- ignore completely all the plumbing needed to interrogate a web service
- change your client type at will, as all clients implement the same interface
- go faster with less code to write!

The only situations when it is recommended to build your own WS client are:

- when using another language
- when you don't want or can't place the WS client library Jersey on your classpath. For example, in an EE6 server that provides JAX-RS 1 and just don't want to work with version 2.

Server side

The web service is not active on any engine by default. To activate it, see the administration guide.

It is not necessary to enable the service on all JQM nodes. It is actually recommended to dedicate a node that will not host jobs (or few) to the WS. Moreover, it is a standard web application with purely stateless sessions, so the standard mechanisms for load balancing or high availability apply if you want them.

Service reference

All objects are serialized to XML by default or to JSON if required. The service is a REST-style web service, so no need for SOAP and other bubbly things.

URL pattern	Method	Non-URL arguments	Return type	Return MIME	Interface equivalent	Description
/ji	GET		List<JobInstance>	application/xml	getJobs	List all known job instances
/ji	POST	JobRequest	JobInstance	application/xml	enqueue	New execution request
/ji/query	POST	Query	Query	application/xml	getJobs(Query)	Returns the executed query
/ji/{jobId}	GET		JobInstance	application/xml	getJob(int)	Details of a Job instance
/ji/{jobId}/messages	GET		List<String>	application/xml	getJobMessages(int)	Retrieve messages created by a Job Instance
/ji/{jobId}/files	GET		List<Deliverable>	application/xml	getJobDeliverables	Retrieve the description of all files created by a JI
/ji/{jobId}/stdout	GET		InputStream	application/os	getJobLogStdOut	Retrieve the stdout log file of the (ended) instance
/ji/{jobId}/stderr	GET		InputStream	application/os	getJobLogStdErr	Retrieve the stderr log file of the (ended) instance
/ji/{jobId}/position/{pos}	POST		void		setJobQueuePosition	Change the position of a waiting job instance inside a queue.
/ji/active	GET		List<JobInstance>	application/xml	getActiveJobs	List all waiting or running job instances
/ji/cancelled/{jobId}	POST		void		cancelJob(int)	Cancel a waiting Job Instance (leaves history)
/ji/killed/{jobId}	POST		void		killJob(int)	Stop (crashes) a running job instance if possible
/ji/paused/{jobId}	POST		void		pauseQueuedJob(int)	Pause a waiting job instance
/ji/paused/{jobId}	DELETE		void		resumeJob(int)	Resume a paused job instance
/ji/waiting/{jobId}	DELETE		void		deleteJob(int)	Completely cancel/remove a waiting Job Instance (even history)
/ji/crashed/{jobId}	DELETE		JobInstance	application/xml	restartCrashedJob	Restarts a crashed job instance (deletes failed history)
/q	GET		List<Queue>	application/xml	getQueues	List all queues defined in the JQM instance
/q/{qId}/{jobId}	POST		void		setJobQueue	Puts an existing waiting JI into a given queue.
/user/{uname}/ji	GET		List<JobInstance>	application/xml	getActiveJobs	List all waiting or running job instances for a user
/jd	GET		List<JobDef>	application/xml	getActiveJobs	List all job definitions
/jd/{appName}	GET		List<JobInstance>	application/xml	getActiveJobs	List all job definitions for a given application
/jr	GET		JobRequest	application/xml	N/A	Returns an empty JobRequest. Usefull for scripts.

Note: application/os = application/output-stream.

Used HTTP error codes are:

- 400 (bad request) when responsibility for the failure hangs on the user (trying to delete an already running

instance, instance does not exist, etc)

- 500 when it hangs on the server (unexpected error)

On the full Java client side, these are respectively translated to `JqmInvalidRequestException` and `JqmClientException`.

The body of the response contains an XML or JSON item giving details on the error.:

```
1   404     GET on absent object
2   500
3   404     DELETE on absent object
4   400     Update user with absent role
5   500     Could not create certificate
6   400     Invalid enqueue parameters
7   400     Simple API  is only available when the application runs on top of JQM␣
↪itself and not a web application server.
8   400     File does not exist
9   500     Generic internal exception
10  400     Generic bad request
```

Script sample

PowerShell script. Logics is the same in any language, script or compiled.:

```
# Note: we use JSON as a demonstration of how to use it over the default XML.␣
↪Obviously, PowerShell deals with XML very well and does not need this.

# Authentication?
$cred = Get-Credential root

#################################
## Enqueue demonstration
#################################

$request = @{applicationName="DemoApi"; user=$env:USERNAME} | ConvertTo-Json
$jobInstance = Invoke-RestMethod http://localhost:62948/ws/client/ji -Method Post -
↪Body $request -Credential $cred -ContentType "application/json" -Headers @{Accept=
↪"application/json"}
$jobInstance.id


#################################
## Query demonstration
#################################

$query = @{applicationName="DemoApi"} | ConvertTo-Json
$res = Invoke-RestMethod http://localhost:62948/ws/client/ji/query -Method Post -Body
↪$query -Credential $cred -ContentType "application/json" -Headers @{Accept=
↪"application/json"}
$res.instances | Format-Table -AutoSize
```

5.4 CLI API

The Command Line Interface has a few options that allow any program to launch a job instance and run a few other commands. The CLI is actually described in the *Command Line Interface (CLI)* chapter of the administration section.

Warning: the CLI creates a JVM with a full JDBC pool on each call call. This is horribly inefficient. A new CLI based on the web services is being considered.

5.5 Engine API

This is a subset of the Client API designed to be usable from payload running inside a JQM engine without any required libraries besides an interface named JobManager.

Its main purpose is to avoid needing the full client library plus Hibernate (a full 20MB of perm gen space, plus a very long initialization time...) just for doing client operations - why not simply use the already initialized client API used by the engine itself? As there is a bit of classloading proxying magic involved, the signatures are not strictly the same to the ones of the client API but near enough so as not be lost when going from one to the other.

TL;DR: inside a JQM payload, use the engine API, not the full client API (unless needing a method not exposed by the engine API).

This engine API is detailed in a chapter of the "creating payloads" section: *Engine API*.

CHAPTER 6

Administration

6.1 Installation

Please follow the paragraph specific to your OS and then go through the common chapter.

6.1.1 Docker install

JQM is available for Docker for Linux/MacOS and Docker for Windows on the Docker Hub. The image is optimized both for development usage and for scale-out production deployment (Kubernetes, Swarm).

Full image documentation is available on the Docker Hub.

6.1.2 Binary install

Windows

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterwards
- An admin account (for installation only)
- A service account with minimal permissions: LOGON AS SERVICE + full permissions on JQM_ROOT.

The following script (PowerShell 5+) will download and copy the binaries (adapt the first two lines). Run it with admin rights.

```
$JQM_ROOT = "C:\TEMP\jqm"
$JQM_VERSION = "${project.version}"
mkdir -Force $JQM_ROOT; cd $JQM_ROOT
Invoke-RestMethod https://github.com/enioka/jqm/releases/download/jqm-all-$JQM_
↪VERSION/jqm-$JQM_VERSION.zip -OutFile jqm.zip
```

(continues on next page)

(continued from previous page)

```
Expand-Archive ./jqm.zip . -Force
rm jqm.zip; mv jqm*/* .
```

Then create a service (change user, password and desired node name):

```
./jqm.ps1 installservice -ServiceUser marsu -ServicePassword marsu -NodeName
↪$env:COMPUTERNAME
./jqm.ps1 start
```

And it's done, a JQM service node is now running.

Note: it is **really** not a good idea to run JQM, an application server with a broad attack surface, as SYSTEM or any priviledged account. Always use a service account as stated above.

Linux / Unix

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterwards
- A group for containing the user which will actually be allowed to run jqm. Default name is jqm. (Debian-like: *sudo groupadd jqm*)
- A user account owning JQM_ROOT (not necessarily root). Default name is jqmadm. (Debian-like: *sudo useradd -m -g jqm jqmadm*)
- A user for running the engine, no specific permissions (and certainly NOT root). Inside the group above. Default name is jqm. (Debian-like: *sudo useradd -m -g jqm jqm*)

The following script will download and copy the binaries (adapt the first two lines). Run as jqmadm

```
JQM_ROOT="/opt/jqm"
JQM_VERSION="1.2.2"
cd $JQM_ROOT
wget https://github.com/enioka/jqm/releases/download/jqm-all-$JQM_VERSION/jqm-$JQM_
↪VERSION.tar.gz
tar xvf jqm-*.tar.gz
rm jqm-*.tar.gz
mv jqm-*/* .
rmdir jqm-*
./bin/permissions.sh
./jqm.sh createnode
./jqm.sh start
```

And it's done, a JQM node is now running.

As root (optional, only if run as a service):

```
## Ensure JQM is not running
ln -s $JQM_ROOT/jqm.sh /etc/init.d/jqm
chmod 700 /etc/init.d/jqm
vi /etc/init.d/jqm
## Change line 5 to the value of JQM_ROOT (cd /opt/...)
## Purge the directory JQM_ROOT/logs
/etc/init.d/jqm start
```

Testing

The following will import the definition of some test jobs included in the distribution, then launch one. (no admin rights necessary nor variables).

Windows:

```
./jqm.ps1 stop  ## Default database is a single file... that is locked by the engine␣
↪if started
./jqm.ps1 allxml  # This will import all the test job definitions
./jqm.ps1 enqueue -JobDef DemoEcho
./jqm.ps1 start
```

Linux / Unix:

```
./jqm.sh stop  ## Default database is a single file... that is locked by the engine␣
↪if started
./jqm.sh allxml  # This will import all the test job definitions
./jqm.sh enqueue DemoEcho
./jqm.sh start
```

Check the JQM_ROOT/logs directory: two log files (stdout, stderr) should have been created (and contain no errors). Success!

6.1.3 Enabling the web interface

By default the web interface is disabled. This will enable it (on all network interfaces) and create a user named "root".

The server listens to a random free port shown in the main log. It can be changed later.

Windows

./jqm.ps1 enablegui -RootPassword mypassword

Linux

./jqm.sh enablegui mypassword

6.1.4 Database configuration

The node created in the previous step has serious drawbacks:

- it uses an HSQLDB database with a local file that can be only used by a single process
- it cannot be used in a network as nodes communicate through the database
- General low performances and persistence issues inherent to HSQLDB

Just edit JQM_ROOT/conf/resources.xml file to reference your own database. It contains by default sample configuration for Oracle, PostgreSQL, HSQLDB, DB2 and MySQL which are the supported databases. (HSQLDB is not supported in production environments)

Note: the database is intended to be shared by all JQM nodes - you should not create a schema/database per node.

Afterwards, place your JDBC driver inside the "ext" directory.

Then stop the service.

Windows:

```
./jqm.ps1 stop
./jqm.ps1 createnode
./jqm.ps1 start
```

Linux / Unix:

```
./jqm.sh stop
./jqm.sh createnode
./jqm.sh start
```

Then, test again (assuming this is not HSQLDB in file mode anymore, and therefore that there is no need to stop the engine).

Windows:

```
./jqm.ps1 allxml
./jqm.ps1 enqueue -JobDef DemoEcho
```

Linux / Unix:

```
./jqm.sh allxml
./jqm.sh enqueue DemoEcho
```

Database support

Oracle

Oracle 10gR2 & 11gR2 are supported. No specific configuration is required in JQM: no options inside jqm.properties (or absent file). No specific database configuration is required.

PostgreSQL

PostgreSQL 9 & 10 are supported (tested with PostgreSQL 9.3). It is the recommended open source database to work with JQM. No specific configuration is required in JQM: no options inside jqm.properties (or absent file). No specific database configuration is required.

Here's a quickstart to setup a test database. As postgres user:

```
$ psql
postgres=# create database jqm template template1;
CREATE DATABASE
postgres=# create user jqm with password 'jqm';
CREATE ROLE
postgres=# grant all privileges on database jqm to jqm;
GRANT
```

MySQL

MySQL 5.6+ is supported with InnoDB (the default). No specific configuration is required in JQM: no options inside jqm.properties (or absent file).

These commands can be used to setup a database.:

```
$ mysql -u root -p
mysql> create database jqm;
mysql> grant all privileges on jqm.* to jqm@'%' identified by 'jqm';
mysql> flush privileges;
```

Note: before version 1.4, a startup script was needed to align sequences between tables on database startup. This is no longer needed and if present, this script should be removed.

HSQLDB

HSQLDB 2.3.x is supported in test environments only.

No specific HSQLDB configuration is required. Please note that if using a file database, HSQLDB prevents multiple processes from accessing it so it will cause issues for creating multi node environments.

Note: prior to version 2.0, there was a bug in a library which required specific options inside the jqm.properties file. This is no longer needed, and this file (now useless but harmless) can be removed.

6.1.5 Global configuration

When the first node is created inside a database, some parameters are automatically created. You may want to change them using your preferred database editing tool or the web console. See *Parameters* for this.

Many users will immediately enable the web administration console in order to easily change this configuration:

```
./jqm.sh enablegui <rootpassword>
./jqm.sh restart
```

The console is then available at http://localhost:xxxxx (where the port is a free port chosen randomly. It is written inside the main log at startup).

6.1.6 JNDI configuration

See *Using resources*.

6.2 Command Line Interface (CLI)

New in version 1.1.3: Once a purely debug feature, JQM now offers a standard CLI for basic operations.

```
java -jar jqm-engine.jar -createnode <nodeName> [-port <portNumber>] | -enqueue
↪<applicationname> | -exportallqueues <xmlpath> | -h | -importjobdef <xmlpath> | -
↪importqueuefile <xmlpath> | -startnode <nodeName> | -v
```

(continues on next page)

(continued from previous page)

-createnode <nodeName>
: create a JQM node of this name (init the database if needed)

-port <portNumber>
: port to use for the newly created node

-enqueue <applicationname>
: name of the application to launch

-exportallqueues <xmlpath>
: export all queue definitions into an XML file

-h, --help
: display help

-importjobdef <xmlpath>
: path of the XML configuration file to import

-importqueuefile <xmlpath>
: import all queue definitions from an XML file

-startnode <nodeName>
: name of the JQM node to start

-v, --version
: display JQM engine version

-p, --resources
: use specified resource.xml file (the file containing the database connection string). Default is JQM_HOME/config/resources.xml

Note: Common options like start, createnode, importxml etc. can be used with convenience script jqm.sh / jqm.ps1

Warning: a few options exist which are not documented here. They are internal and are not part the API (may change without notice, etc).

6.3 JMX monitoring

JQM fully embraces JMX as its main way of being monitored.

6.3.1 Monitoring JQM through JMX

JQM exposes threee different level of details through JMX: the engine, the pollers inside the engine, and the job instances currently running inside the pollers.

The first level will most often be enough: it has checks for seeing if the engine process is alive and if the pollers are polling. Basically, the two elements needed to ensure the engine is doing its job. It also has a few interesting statistics.

The poller level offers the same checks but at its level.

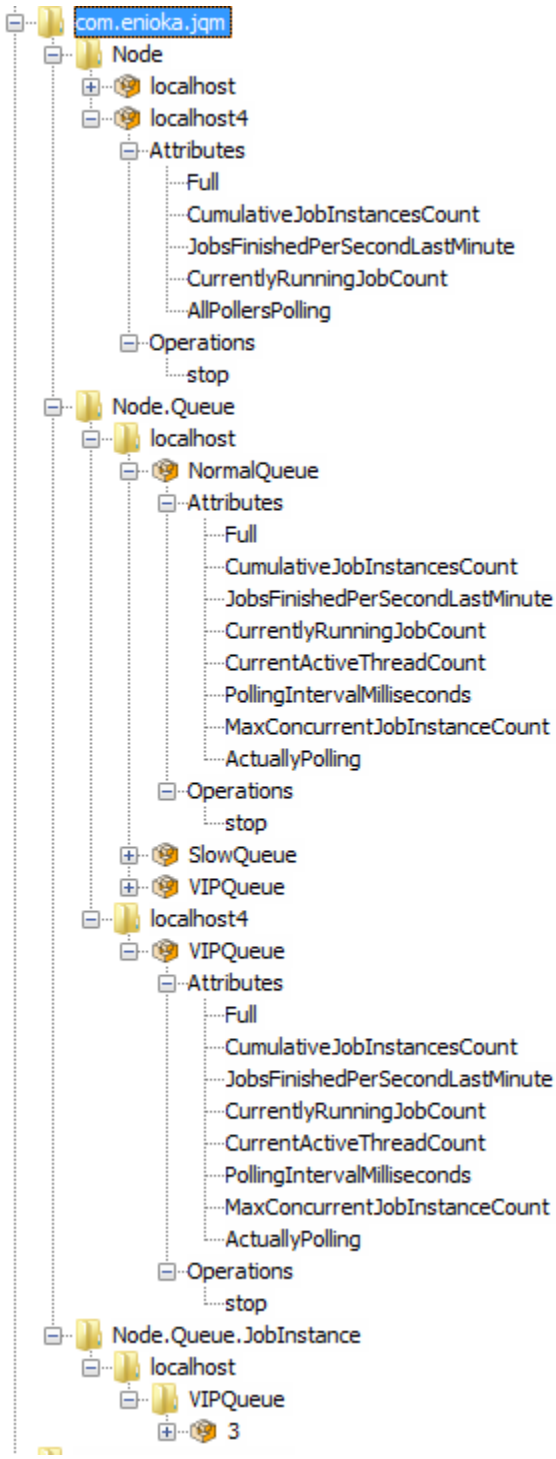
Finally, it is possible to monitor each job individually. This should not be needed very often, the main use being killing a running job.

The JMX tree is as follow:

- com.enioka.jqm:type=Node,name=XXXX
- com.enioka.jqm:type=Node.Queue,Node=XXXX,name=YYYY
- com.enioka.jqm:type=Node.Queue.JobInstance,Node=XXXX,Queue=YYYY,name=ZZZZ

where XXXX is a node name (as given in configuration), YYYY is a queue name (same), and ZZZZ is an ID (the same ID as in History).

In JConsole, this shows as:



Note: there is another type of object which is exposed by JQM: the JDBC pools. Actually, the pool JMX beans come from tomcat-jdbc, and for more details please use their documentation at https://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html. Suffice to say it is very complete, and exposes methods to recycle, free connections, etc.

6.3.2 Remote JMX access

By default, JQM does not start the remote JMX server and the JMX beans can only be accessed locally. To start the JMX remote server, two `Node` (i.e. the parameters of a *JQM engine*) parameters must be set: jmxRegistryPort (the connection port) and jmxServerPort (the port on which the real communicate will occur). If one of these two parameters is null or less than one, the JMX remote server is disabled.

The connection string is displayed (INFO level) inside the main engine log at startup. It is in the style of

```
service:jmx:rmi://dnsname:jmxServerPort/jndi/rmi://dnsname:jmxRegistryPort/jmxrmi
```

When using jConsole, it is possible to simply specify dnsname:jmxRegistryPort.

Remark: JMX usually uses a random port instead of a fixed jmxServerPort. As this is a hassle in an environment with firewalls, JQM includes a JMX server that uses a fixed port, and specifying jmxServerPort in the configuration is therefore mandatory.

Warning: JQM does not implement any JMX authentication nor encryption. This is a huge security risk, as JMX allows to run arbitrary code remotely. **Only enable this in production within a secure network**. Making JQM secure is already an open enhancement request.

6.3.3 Beans detail

class JqmEngineMBean
This bean tracks a JQM engine.

getCumulativeJobInstancesCount()
The total number of job instances that were run on this node since the last history purge. (long)

getJobsFinishedPerSecondLastMinute()
On all queues, the number of job requests that ended last minute. (float)

getCurrentlyRunningJobCount()
The number of currently running job instances on all queues (long)

getUptime()
The number of seconds since engine start. (long)

isAllPollersPolling()
A must-be-monitored element: True if, for all pollers, the last time the poller looped was less than a polling period ago. Said the other way: will be false if at least one queue is late on evaluating job requests. (boolean)

isFull()
Will usually be a warning element inside monitoring. True if at least one queue is full. (boolean)

getVersion()
The engine version, in x.x.x form. (string)

stop()
Stops the engine, exactly as if stopping the service (see stop procedure for details).

class PollingMBean
This bean tracks a local poller. A poller is basicaly a thread that polls a *queue* inside the database at a given interval (defined in a `DeploymentParameter`).

getCurrentActiveThreadCount()
The number of currently running job instances inside this queue.

stop()
Stops the poller. This means the queue won't be polled anympore by the engine, even if configuration says otherwise, until engine restart.

getPollingIntervalMilliseconds()
Number of seconds between two database checks for new job instance to run. Purely configuration - it is present to help computations inside the monitoring system.

getMaxConcurrentJobInstanceCount()
Max number of simultaneously running job instances on this queue on this engine. Purely configuration - it is present to help computations inside the monitoring system.

getCumulativeJobInstancesCount()
The total number of job instances that were run on this node/queue since the last history purge.

getJobsFinishedPerSecondLastMinute()
The number of job requests that ended last minute. (integer)

getCurrentlyRunningJobCount()
The number of currently running job instances inside this queue.

isActuallyPolling()
True if the last time the poller looped was less than a period ago. (the period can be retrived through *getPollingIntervalMilliseconds()*)

isFull()
True if running count equals max job number. (the max count number can be retrieved through *getMaxConcurrentJobInstanceCount()*)

class LoaderMBean
This bean tracks a running job, allowing to query its properties and (try to) stop it. It is created just before the start of the *payload* and destroyed when it ends.

kill()
Tries to kill the job. As Java is not very good at killing threads, it will often fail to achieve anything. See *the job documentation* for more details.

getApplicationName();
The name of the job. (String)

getEnqueueDate();
Start time (Calendar)

getKeyword1();
A fully customizable and optional tag to help sorting job requests. (String)

getKeyword2();
A fully customizable and optional tag to help sorting job requests. (String)

getKeyword3();
A fully customizable and optional tag to help sorting job requests. (String)

getModule();
A fully customizable and optional tag to help sorting job requests. (String)

getUser();
A fully customizable and optional tag to help sorting job requests. (String)

getSessionId();
A fully customizable and optional tag to help sorting job requests. (int)

getId();
The unique ID attributed by JQM to the execution request. (int)

getRunTimeSeconds();
Time elapsed between startup and current time. (int)

6.4 Web administration console

As all serious server-oriented middlewares, JQM is first and foremost a CLI (and configuration files) administered tool. However, for a more Windows administrator-friendly experience, a web console is also offered. It allows every parameter modification (alter the definition of jobs, remove an engine node from the cluster, etc) and every client operation (new launch, consult launch history...) available.

It is **disabled by default** (except in single node Docker distributions).

6.4.1 Enabling the console

The console must be enabled node by node. It should only be enabled on a single node (or a couple of nodes behind a load balancer), as one console is able to administer every node referenced inside the central database.

For the GUI basic functions to work, the admin web service API must be enabled. See *Data security* for this. For full functionality the three WS APIs must be enabled.

A CLI shortcut is offered to enable all what is needed to use the GUI:

```
./jqm.sh enablegui <rootpassword>
./jqm.sh restart
```

6.4.2 First connection

Using a supported browser, connect to:

http://servername:httpport (or, if SSL is enabled, https://...)

The port is written during node startup inside the JQM log.

Then click on "login", and submit authentication data for user "root" (its password can be reset through the CLI if needed).

Then head to the "users" tab, and create your own user with your own password and associate it with a suitable role.

6.4.3 If TLS is enabled

In that case, the recommended approach is to use a certificate to connect.

Head to the "users" tab and select your own user. In the lower right part of the page, click on "new certificate" and save the proposed file.

Unzip the file on your computer. For Linux and Unixes, then import the unzipped PFX file into your usual browser following its specific instructions.

For Windows, just double click on the PFX file, click next, next, enter the password (do NOT select make the key exportable), then next and accept everything. Restart IE or Chrome and farewell password prompts. (Won't work for Firefox, which has chosen to have its own certificate store so either import it inside Firefox using the method on their website or use another browser)

Note: password for the PFX file is always "SuperPassword" without quotes.

6.5 Logs

There are two kinds of logs in JQM: the engine log, and the job instances logs.

6.5.1 Log levels

Level	Meaning in JQM
TRACE	Fine grained data only useful for debugging, exposing the innards of the steps of JMQ's state-machine
DEBUG	For debugging. Gives the succession of the steps of the state-machine, but not what happens inside a step.
INFO	Important changes in the state of the engine (engine startup, poller shutdown, ...)
WARN	An alert: something has gone wrong, analysis is needed but not urgent. Basically, the way to raise the admins' attention
ERROR	The engine may continue, but likely something really bad has happened. Immediate attention is required
FATAL	The engine is dead. Immediate attention is required

The default log-level is INFO.

In case of a classic two-level monitoring system ('something weird' & 'run for your life'), WARN whould be mapped to the first level while ERROR and FATAL should be mapped to the second one.

6.5.2 Engine log

It is named jqm.log. It is rotated as soon as it reaches 10MB. The five most recent files are kept.

It contains everything related to the engine - job instance launches leave no traces here.

6.5.3 Java Virtual Machine Log

Named jqm_<nodename>_std.log and jqm_<nodename>_err.log for respectively standard ouput and error output. It contains every log that the engine did not manage to catch. For instance low level JVM error statement such as OutOfMemoryException. It is rotated at startup when it reaches 10MB. 30 days of such logs are kept.

6.5.4 Payload logs

One file named after the ID of the job instance is created per payload launch. It contains:

- the engine traces concerning this log (classloader creation, start, stop, ...)
- the stdout/stderr of the job instance. This means that if payloads use a ConsoleAppender for their logs (as is recommended) it will be fully here.

These files are **not purged** automatically. This is the admin's responsability.

Also of note, there are two log levels involved here:

- the engine log level, which will determine the verbosity of the traces concerning the luanch of the job itself.
- the payload log level: if the payload uses a logger (log4j, logback, whatever), it has its own log level. This log level is not related in any way to the engine log level. (remember: running a payload inside JQM is the same as running it inside a standard JVM. The engine has no more influence on the behaviour of the payload than a JVM would have)

6.6 Operations

6.6.1 Starting

Note: there is a safeguard mechanism which prevents two engines (JQM java processes) to run with the same node name. In case of engine crash (kill -9) the engine will ask you to wait (max. 2 minutes) to restart so as to be sure there is no other engine running with the same name. On the other hand, cleanly stopping the engine is totally transparent without ever any need to wait.

Windows

The regular installation is as a service. Just do, inside a PowerShell prompt with elevated (admin) rights:

```
Start-Service JQM*
```

It is also possible to start an engine inside a command prompt. In that case, the engine stops when the prompt is closed. This is mainly useful for debugging purposes.

```
java -jar jqm.jar -startnode $env:COMPUTERNAME
```

(change the node name at will - by default, the computer name is used for the node name).

Unix/Linux

A provided script will launch the engine in "nohup &" and store the pid inside a file.

```
./jqm.sh start
```

Under *x systems, the default node name is the username.

The script respects the general conventions of init.d scripts.

6.6.2 Stopping

A stop operation will wait for all running jobs to complete, with a two minutes (parameter) timeout. No new job instances are started on the node as soon as the stop order is thrown.

Windows

The regular installation is as a service. Just do, inside a PowerShell prompt with elevated (admin) rights:

```
Stop-Service JQM*
```

For console nodes, just do Ctrl+C or close the console.

Unix

```
./jqm.sh stop
```

The clean stop sequence is actually triggered by a SIGTERM (normal kill) - the jqm.sh script simply stores the PID at startup and does a kill to shutdown.

6.6.3 Restarting

There should never be any need for restarting an engine, save for the few configuration changes that are listed in *Parameters*.

Windows:

```
Restart-Service JQM*
```

*X:

```
./jqm.sh restart
```

In both cases, it is strictly equivalent to stopping and then starting again manually (including the two-minutes timeout).

6.6.4 Pausing and resuming

An paused engine runs normally but does not take new job instances anymore. Job instances already running at the time of pause go on normally.

Pausing and resuming methods are available at two levels:

- per node (pause all queues for an engine)
- per binding (pause only one queue for an engine)

These methods are available through JMX and though the database (in which case modifications are only applied after at most parameter internalPollingPeriodMs). The database method is also exposed by the admin GUI.

Also, a client through the client API can pause all bindings on a designated queue.

6.6.5 Backup

Elements to backup are:

- the database (unless the history is not precious)
- files created

Standard tools can be used, there is nothing JQM specific here.

6.6.6 Purges

The logs of the engine are automatically purged. Job instance logs and created files, however, are not.

The History table should be purged too - see *Managing history*.

6.7 Parameters

6.7.1 Engine parameters

These parameters gouvern the behaviour of the JQM engines.

There are three sets of engine parameters:

- node parameters, for parameters that are specific to a single engine (for example, the TCP ports to use). These are stored inside the database.
- global parameters, for parameters concerning all engines (for example, a list of Nexus repositories). These are stored inside the database.
- bootstrap parameters: as all the previous elements are stored inside the database, an engine needs a minimal set of parameters to access the database and start.

Note: also of interest in regard to engine configuration is the *queues configuration*.

Bootstrap

This is a file named JQM_ROOT/conf/resource.xml. It contains the definition of the connection pool that is used by JQM to access its own database. See *Administrating resources* for more details on the different parameters - it is exactly the same as a resource defined inside the JQM database, save it is inside a file read before trying to connect to the JQM database.

Actually, resources.xml can contain any resource, not just the connection to the JQM database. However, it is not recommended - the resource would only be available to the local node, while resources defined in the database are available to any node.

A second file exists, named JQM_ROOT/conf/jqm.properties. It is used specifically for database-specific bootstrap parameters, and is only useful in very specific use cases. It can be - and should be - safely deleted otherwise. The parameters are:

- com.enioka.jqm.jdbc.tablePrefix: a prefix to add to all table names (if value is "**MARSU_**", tables will be named MARSU_HISTORY, MARSU_NODE...). Default is empty.
- com.enioka.jqm.jdbc.datasource: JNDI name of the datasource from resource.xml to use as the main JQM database connection. Default is jdbc/jqm.
- com.enioka.jqm.jdbc.allowSchemaUpdate: should not be used in normal operations.

Changes to bootstrap files require an engine restart.

Node parameters

These parameters are set inside the JQM database table named NODE. They have to be altered through the GUI, through the CLI options or directly inside the database with your tool of choice.

Name	Description	Default	Nullable	Restart
DNS	The interface name on which JQM will listen for its network-related functions	first hostname	No	No
PORT	Port for the basic servlet API	Random free	No	No
dlRepo	Storage directory for files created by payloads	JQM_ROOT\outputfiles	No	Yes
REPO	Storage directory for all payloads jars and libs	JQM_ROOT\jobs	No	Yes
ROOT-LOGLEVEL	The log level for this engine (TRACE, DEBUG, INFO, WARN, ERROR, FATAL)	INFO	No	Yes
EX-PORTREPO	Not used			
JMXREG-ISTRYPORT	TCP port on which the JMX registry will listen. Remote JMX disabled if NULL or <1.	NULL	Yes	Yes
JMXSERVER-PORT	Same with server port	NULL	Yes	Yes

('restart' means: restarting the engine in question is needed to take the new value into account)

Global parameters

These parameters are set inside the JQM database table named GLOBALPARAMETER. There is no CLI to modify these, therefore they have to be altered directly inside the database with your tool of choice or through the GUI.

Name	Description	Default	Restart	Nullable
mavenRepo	A comma-separated list of Maven repositories to use for dependency resolution	Maven Central	No	No
mavenSettingsCL	an alternate Maven settings.xml to use. If absent, the usual file inside ~/.m2 is used.	NULL	No	Yes
defaultConnection	the JNDI alias returned by the engine API getDefaultConnection method.	jdbc/jqm	No	No
logFilePerLaunch	if 'true', one log file will be created per launch. If 'false', job stdout/stderr is lost. if 'both', one log file will be created per launch PLUS one common file concatening all these files	true	Yes	No
internalPollingPeriodMs	Period in ms for checking stop orders. Also period at which the "I'm a alive" signal is sent. Also used for checking and applying parameter modifications (new queues, global prm changes...)	60000	Yes	No
disableWsApi	Disable all HTTP interfaces on all nodes. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	false	No	Yes
enableWsApiSsl	All HTTP communications will be HTTPS and not HTTP.	false	No	No
enableWsApiAuth	Use HTTP basic authentication plus RBAC backend for all WS APIs	true	No	No
disableWsApiSimple	Forbids the simple API from loading on any node. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	NULL	Yes	Yes
disableWsApiClient	Forbids the client API from loading on any node. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	NULL	Yes	Yes
disableWsApiAdmin	Forbids the admin API from loading on any node. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	NULL	Yes	Yes
enableInternalPki	Use the internal (database-backed) PKI for issuing certificates and trusting presented certificates	true	No	No
pfxPassword	Password of the private key file (if not using internal PKI).	SuperPassword	No	Yes

Here, nullable means the parameter can be absent from the table.

Parameter name is case-sensitive.

Note: There must be at least one Maven repository specified. If using Maven central, please specify 'http://repo1.maven.org/maven2/' and not one the numerous other aliases that exist. Maven Central is only used if explicitly specified (which is the default).

Also, as a side note, mail notifications use the JNDI resource named mail/default, which is created on node startup if

it does not exist. See resource documentation to set it up.

6.8 Managing queues

JQM is a Queue Manager (the enqueued objects being payload execution requests). There can be as many queues as needed, and each JMQ node can be set to poll a given set of queues, each with different parameters.

By default, JQM creates one queue named "default" and every new node will poll that queue every ten seconds. This is obviously very limited - this chapter details how to create new queues and set nodes to poll it.

Please read the *concepts overview* chapter before dealing with this paragraph, as it defines the underlying concepts and gives examples.

6.8.1 Defining queues

Queues are defined inside the JQM database table QUEUE. It can be directly modified though the web administration console, or an XML export/import system can be used. Basically, a queue only has an internal technical ID, a name and a description. All fields are compulsory.

The XML is in the form:

```
<jqm>
        <queues>
                <queue>
                        <name>XmlQueue</name>
                        <description>Queue to test the xml import</description>
                        <timeToLive>10</timeToLive>
                        <jobs>
                                <applicationName>Fibo</applicationName>
                                <applicationName>Geo</applicationName>
                        </jobs>
                </queue>
                <queue>
                        <name>XmlQueue2</name>
                        <description>Queue 2 to test the xml import</description>
                        <timeToLive>42</timeToLive>
                        <jobs>
                                <applicationName>DateTime</applicationName>
                        </jobs>
                </queue>
        </queues>
</jqm>
```

The XML does more than simply specify a queue: it also specify which job definitions should use the queue by default. The XML can be created manually or exported from a JQM node. (See the *CLI reference* for import and export commands)

The timeToLive parameter is not used any more.

6.8.2 Defining pollers

Having a queue is enough to enqueue job requests in it but nothing will happen to these requests if no node polls the queue to retrieve the requests...

The association between a node and a queue is done with the GUI.

It defines the following elements:

- ID: A technical unique ID
- CLASSID: unused (and nullable)
- NODE: the technical ID of the Node
- QUEUE: the technical ID of the Queue
- NBTHREAD: the maximum number of requests that can be treaded at the same time
- POLLINGINTERVAL: the number of milliseconds between two peeks on the queue. **Never go below 1000ms.**

6.9 Administrating resources

6.9.1 Defining a resource

Resources are defined inside the JQM database, and are therefore accessible from all JQM nodes. By 'resource' JNDI means an object that can be created through a (provided) ObjectFactory. There are multiple factories provided with JQM, concerning databases, files & URLs which are detailed below. Moreover, the *payload* may provide whatever factories it needs, such as a JMS driver (example also below).

The main JNDI directory table is named `JndiObjectResource` and the object parameters belong to the table `JndiObjectResourceParameter`. Resources can be edited through the administration console.

The following elements are needed for every resource, and are defined in the main table:

Name	Description	Example
name	The JNDI alias - the string used to refer to the resource in the *payload* code	jdbc/mydatasource
description	a short string giving the admin every info he needs	connection to main db
type	the class name of the desired resource	com.ibm.mq.jms.MQQueueConnectionFactory
factory	the class name of the ObjectFactory able to create the desired resource	com.ibm.mq.jms.MQQueueConnectionFactoryFactory
singleton	see below	false

For every resource type (and therefore, every ObjectFactory), there may be different parameters: connection strings, paths, ports, ... These parameters are to be put inside the table JndiObjectResourceParameter.

The JNDI alias is free to choose - even if conventions exist. Please note that JQM only provides a root context, and no subcontexts. Therefore, in all lookups, the given alias will searched 'as provided' (including case) inside the database.

6.9.2 Singletons

One parameter is special: it is named "singleton". Default is 'false'. If 'true', the creation and caching of the resource is made by the engine itself in its own class context, and not inside the payload's context (i.e. classloader). It is useful for the following reasons:

- Many resources are actually to be shared between payloads, such as a connection pool
- Very often, the payload will expect to be returned the same resource when making multiple JNDI lookups, not a different one on each call. Once again, one would expect to be returned the same connection pool on each call, and definitely not to have a new pool created on each call!

- Some resources are dangerous to create inside the payload's context. As stated in *Payload basics*, loading a JDBC driver creates memory leaks (actually, class loader leaks). By delegating this to the engine, the issue disappears.

Singleton resources are created the first time they are looked up, and kept forever afterwards.

As singleton resources are created by the engine, the jar files containing resource & resource factory must be available to the engine class loader. For this reason, the jar files must be placed manually inside the $JQM_ROOT/ext directory (and they do not need to be placed inside the dependencies of the payload, even if it does not hurt to have them there when the default parent-first class loading is used). For a resource which provider is within the payload, being a singleton is impossible - the engine class context has no access to the payload class context.

By default, the $JQM_ROOT/ext directory contains the following providers, ready to be used as singleton (or not) resources:

- the File provider and URl provider inside a single jar named jqm-provider
- the JDBC pool, inside two jars (tomcat-jdbc and tomcat-juli)
- the HSQLDB driver

Besides the HSQLDB driver, which can be removed if another database is used, the provided jars should never be removed. Jars added later (custom resources, other JDBC drivers, ...) can of course be removed. Also of note: it is not because a jar is inside 'ext' that the corresponding resources can only be singletons. They can be standard as well.

6.9.3 Examples

Below, some examples of resources definition. To see how to actually use them in your code, look at *Using resources*.

JDBC

Note: the recommended naming pattern for JDBC aliases is jdbc/name

Connection pools to databases through JDBC is provided by an ObjectFactory embedded with JQM named tomcat-jdbc.

As noted above, JDBC pool resources should always be singletons: it is stupid to create a new pool on each call AND it would create class loader leaks otherwise.

Classname	Factory class name
javax.sql.DataSource	org.apache.tomcat.jdbc.pool.DataSourceFactory

Parameter name	Value
maxActive	max number of pooled connections
driverClassName	class of the db JDBC driver
url	database url (see db documentation)
singleton	always true (since engine provider)
username	database account name
password	password for the database account

There are many other options, detailed in the Tomcat JDBC documentation.

JMS

Note: the recommended naming pattern for JMS aliases is jms/name

Parameters for MQ Series QueueConnectionFactory:

Classname	Factory class name
com.ibm.mq.jms.MQQueueConnectionFactory	com.ibm.mq.jms.MQQueueConnectionFactoryFactory

Parameter name	Value
HOST	broker host name
PORT	mq broker listener port
CHAN	name of the channel to connect to
QMGR	name of the queue manager to connect to
TRAN	always 1 (means CLIENT transmission)

Parameters for MQ Series Queue:

Classname	Factory class name
com.ibm.mq.jms.MQQueue	com.ibm.mq.jms.MQQueueFactory

Parameter name	Value
QU	queue name

Parameters for ActiveMQ QueueConnexionFactory:

Classname	Factory class name
org.apache.activemq.ActiveMQConnectionFactory	org.apache.activemq.jndi.JNDIReferenceFactory

Parameter name	Value
brokerURL	broker URL (see ActiveMQ site)

Parameters for ActiveMQ Queue:

Classname	Factory class name
org.apache.activemq.command.ActiveMQQueue	org.apache.activemq.jndi.JNDIReferenceFactory

Parameter name	Value
physicalName	queue name

Files

Note: the recommended naming pattern for files is fs/name

Classname	Factory class name
java.io.File.File	com.enioka.jqm.providers.FileFactory

Parameter name	Value
PATH	path that will be used to initialize the File object

URL

Note: the recommended naming pattern for URL is url/name

Classname	Factory class name
java.io.URL	com.enioka.jqm.providers.UrlFactory

Parameter name	Value
URL	url that will be used to initialize the URL object

Mail session

Outgoing SMTP mail session.

Note: the recommended naming pattern is mail/name

Classname	Factory class name
javax.mail.Session	com.enioka.jqm.providers.MailSessionFactory

Parameter name	Value
smtpServerHost	Name or IP of the SMTP server. The only compulsory parameter
smtpServerPort	Optional, default is 25
useTls	Default is false
fromAddress	Can be overloaded when sending a mail. Default is noreply@jobs.org
smtpUser	If SMTP server requires authentication.
smtpPassword	

6.10 Managing history

For each completed execution request, one entry is created inside the database table named History. This table is very special in JQM, for it is the sole table that is actually never read by the engine. It is purely write-only (no deletes, no updates) and it only exists for the needs of reporting.

Therefore, this is also the only table which is not really under the control of the engine. It is impossible to guess how this table will be used - perhaps all queries will be by field 'user', or by job definition + date, etc. **Hence, it is the only table that needs special DBA care** and on which it is allowed to do structural changes.

The table comes without any indexes, and without any purges.

When deciding the deployment options the following items should be discussed:

- archiving
- partitioning
- indexing
- purges

The main parameters to take into account during the discussion are:

- number of requests per day
- how far the queries may go back
- the possibilities of your database backend

Purges should always be considered.

Purging this table also means purging related rows from tables 'Message' and 'RuntimeParameter'.

6.11 Data security

JQM tries to be as simple as possible and to "just work". Therefore, things (like many security mechanisms) that require compulsory configuration or which always fail on the first tries are disabled by default.

Therefore, out of the box, JQM is not as secure as it can be (but still reasonably secure).

This does not mean that nothing can be done. The rest of this chapter discusses the attack surface and remediation options.

6.11.1 Data & data flows

Security needs are rated on a three ladder scale: low, medium, high.

Note: the administration GUI is not listed in this section, as it is simply a client based on the different web services.

Central Configuration

This is the defintion of the JQM network: which JVM runs where, with which parameters. These parameters include the main security options. Most data in here is easy to guess by simply doing a network traffic analysis (without having to know the actual content of the traffic - just routes will tell the structure of the network).

Integrity need: high (as security mechanisms can be disabled here)

Confidentiality need: low (as data is public anyway. Exception: if the internal PKI isued, the root certificate is here. But this certificate actually protects… the central configuration so its not a real issue)

Stored in: central database

Exposed by: admin web service (R/W), direct db connection (R/W).

Node-specific configuration

Every node has a configuration file containing the connection information to the central database.

Integrity need: medium (rerouting the node on another db will allow to run arbitrary commands, but such a case means the server is compromised anyway)

Confidentiality need: high (exposes the central database)

Stored in: local file system

Exposed by: file system (R).

Job referential

The definition of the different batch jobs that are run - basically shell command lines.

Integrity need: high (as a modification of this data allows for arbitrary command line execution)

Confidentiality need: high (as people often store password and other critical data inside their command lines)

Stored in: central database

Exposed by: admin web service (R/W), client web service (R), direct db connection (R/W).

Tracking data

Every queued, running or ended job instance has tracking objects inside the central database.

Integrity need: medium (a modification on an ended instance will simply make history wrong, but altering a yet to run instance will allow to modify its command line)

Confidentiality need: high (as people often store password and other critical data inside their command lines, which are stored in this data)

Stored in: central database

Exposed by: client web service (R/W), simple web service (enqueue execution request & get status), direct db connection (R/W).

Logs & batch created files

Every job instance creates a log file. It may, depending on the jobs, contain sensitive data. There is however no sensitive data inside JQM's own logs. Moreover, batch jobs can create file (reports, invoices, ...) that may be critical and are stored alongside logs.

Integrity need: depends

Confidentiality need: depends

Stored in: file system (on the node it was created)

Exposed by: simple web service (R), file system

Binaries

Obviously, the JQM binaries are rather critical to its good operation.

Integrity need: high

Confidentiality need: low (on GitHub!)

Stored in: file system

Exposed by: file system

6.11.2 Summary of elements to protect

Central database: it contains elements that are both confidential and which integrity is crucial.

Admin web service: it exposes (R/W) the same elements

Client web service: it exposes (R/W) all operational data

Simple web service: it exposes log files, business files, and allows for job execution submission.

Binaries: obvious

Database connection configuration on each node: exposes the central database.

> **Warning:** in any way, compromising the central database means compromising the whole JQM cluster. Therefore protecting it should be the first step in any JQM hardening! This will not be detailed here, as this is database specific - ask your DBA for help.

6.11.3 Security mechanisms

Web services

SSL

All communications can be forced inside a SSL channel that will guarantee both confidentiality and integrity, provided certificate chains are correctly set.

JQM provided its own Private Key Infrastructure (PKI), which allows it to start without need for any certificate configuration. Its root certificate is stored inside the central database. The root key is created randomly at first startup. It also allows for easy issuing of client certificates for authentication through a web service of the admin API (and the admin GUI).

However, using the internal PKI is not compulsory. Indeed, it the limitation of not having a revocation mechanism. If you don't want to use your own:

- put the private key and public certificate of each node inside JQM_ROOT/conf/keystore.pfx (PKCS12 store, password SuperPassword)
- put the public certificate chain of the CA inside JQM_ROOT/conf/trusted.jks (JKS store, password SuperPassword)
- set the global parameter enableInternalPki to 'false'.

SSL is **disabled** by default, as in most cases JQM is run inside a secure perimeter network where data flows are at acceptable risk. It can be enabled by setting the global parameter enableWsApiSsl to 'true'. Once enabled, all nodes will switch to SSL-only mode on their next reboot.

Authentication

JQM uses a Role Based Access Control (RBAC) system to control access to its web services, coupled with either basic HTTP authentication or client certificate authentication (both being offered at the same time).

JQM comes with predefined roles that can be modified with the exception of the "administrator" role which is compulsory.

Passwords are stored inside the central database in hash+salt form. Accounts can have a validity limit or be disabled.

Authentication is **enabled** by default. The rational behind this is not really to protect data from evil minds, but to prevent accidents in multi user environments. It can be disabled by setting the global parameter enableWsApiAuth to 'false'.

Note: as the web GUI is based on the admin web service, it also uses these. In particular, it can use SSL certificates for authentication.

Clients use of SSL and authentication

JQM comes with two "ready to use" client libraries - one directly connecting to the central database, the other to the client web service API.

The web service client has a straightforward use of SSL and authentication - it must be provided a trust store, and either a user/password or a client certificate store.

The direct to database client does not use authentication - it has after all access to the whole database, so it would be rather ridiculous. It has however a gotcha: file retrieval (log files as well as business files created by jobs) can only be done through the simple web service API. Therefore, the client also needs auth data. As it has access to the database, it will create a temporary user with 24 hours validity for this use on its own. As far as SSL is concerned, it must be provided a trust store too (or else will use system default stores). **This is only necessary if the file retrieval abilities are to be used inside a SSL environment** - otherwise, this client library does not use the web services API at all.

File retrieval specific protection

The simple API exposes the only API able to fetch a business file (report, invoice, etc - all files created by the jobs). To prevent predictability, the ID given as the API parameter is not the sequential ID of the file as referenced inside the central database but a random 128 bits GUID.

Therefore, it will be hard for an intruder to retrieve the files created by a job instance even without SSL or authentication.

Switch off 'protection'

If the web services are not needed, they can be suppressed by setting the disableWsApi global parameter to 'true'. This will simply prevent the web server from starting at all on every node.

Web services can also be selectively disabled on all nodes by using the following global parameters: disableWsApiClient, disableWsApiAdmin, disableWsApiSimple. These parameters are not set by default.

Finally, each node has three parameters allowing to choose which APIs should be active on it. By **default, simple API is enabled, client & admin APIs are disabled**.

Warning: disabling the simple API means file retrieval won't work.

Database

Please see your DBA. Once again, the database is the cornerstone of the JQM cluster and its compromission is the compromisson of every server/OS account on which a JQM node runs.

Binaries

A script will soon be provided to set minimal permissions on files.

Warning: a useful reminder: JQM should never run as root/local system/administrator/etc. No special permissions under Unix-like systems, logon as service under windows. That's all. Thanks!

6.11.4 Monitoring access security

Local JMX is always active (it's a low level Java feature) and Unix admins can connect to it.

Remote JMX is disabled by default. Once enabled, it is accessible without authentication nor encryption. Tickets #68 an #69 are feature requests for this.

This is a huge security risk, as JMX allows to run arbitrary code. Firewalling is necessary in this case.

Remediation: using local JMX (through SSH for example) or using firewall rules.

6.11.5 Tracing

To come. Feature request tickets already open. The goal will be to trace in a simple form all configuration modification and access to client APIs.

Currently, an access log lists all calls to the web services, but there is no equivalent for the JPA API (and logs are not centralized in any way).

6.12 Database and reliability

As shown in *How JQM works* JQM nodes need the database to run, since they are basically polling the database for job instances to run. So when the database goes down, nodes are unable to work correctly. However, they won't go down, for this would be an administration nightmare (for example, a database cluster switch over will briefly cut connectivity but should not require to restart a hundred JQM processes as it is a standard operation in many environments). They will just wait for the database to come back online.

In details:

- pollers stop on first failure. Therefore, no new job instance will run until database connectivity is restored. Failed pollers are restarted on database coming back on line.
- **running job instances continue to run as long as they are not concerned with database connectivity.**
  - They will be impacted if they use a JQM API method, as they all call the database behind the scenes
  - They will not be impacted otherwise

  - Impacted instances will classically crash with a JDBC exception.

- **ending job instances are stored in memory and wait for the database to come back to be reported. This is referred to as "d**

  - Therefore, these instances will be stored as "running" inside the database even if they are actually already done. Not that it matters if the database is fully down, but if only the connectivity is down and the db is up it may seem surprising.

Pollers failing and the need for delayed finalization are reported as errors inside the main log. Coming back online operations are reported as warnings.

Finally, please note that delayed finalization is purely an in-memory process. That is, if the node is stopped, the state of the ended job instance is lost. On next node startup, the node will realize it does not know what has happened to a job instance it was running before being killed and will report it as crashed, even if it had ended correctly. This is to avoid false OK that would cause havoc inside scheduled production plans. So the rule of thumb is: *do not restart JQM nodes as long as the database is unavailable*.

6.13 Administration web services

Warning: the admin REST web service is a **private** JQM API. It should never be accessed directly. Either use the web administration console or the future CLI. The service is only described here for reference and as a private specification.

The web console is actually only an HTML5 client built on top of some generic administration web services - it has no priviledged access to any resources that could not be accessed to in any other ways.

These services are REST-style services. It is deployed along with the client web services and the console (in the same war file). In accordance to the most used REST convention, the HTTP verbs are used this way:

Note: the API never returns anything on POST/PUT/DELETE operations. On GET, it will output JSON (application/json). By setting the "accept" header in the request, it is also possible to obtain application/xml.

URL	GET	POST	PUT	DELETE	Description
/q	X	X	X		Container for queues. Example of return JSON for a GET: [{"defaultQueue":true,"description":"default queue","id":2,"name":"DEFAULT"},{"defaultQueue":false,"description":"meuh","id":3,"name":"MEUH"}]
/q/{id}	X		X	X	A queue. For creating one, you may PUT this (note the absence of ID): {"defaultQueue":false,"description":"my new queue","name":"SUPERQUEUE"}
/qmapping	X	X	X		Container for the deployments of queues on nodes. nodeName & queueName cannot be set - they are only GUI helpers. Example of return JSON for a GET: [{"id":9,"nbThread":5,"nodeId":1,"nodeName":"JEUX","pollingInterval":1000,"queueId":2,

Note: queues and job definitions are also available through the client API. However, the client version is different with less data exposed and no possibility to update anything.

6.14 Scheduling jobs

JQM is at its core a passive system: it receives execution requests from an external system and processes these requests according to simple queuing mechanisms. The requests can come from any type of systems, the two prime and common examples being an interactive system (a user has pushed a button which has in turn triggered a new job request) and a scheduler (Control^M, Automic, $Universe... or even the Unix crontab/Windows task scheduler).

However sometimes the need is simply to schedule launches on simple recurrence rules - without any need for the advanced functions offered by full-fledged job scheduler, such as inter-launch dependencies, misfire control... In this case, the user had before JQM version 2 only two sub-optimal solutions: use a full job scheduler (costly and complicated) or use a crontab script (no logs, no history, ...). JQM v2 therefore adds support for simple recurrence rules, which are detailed in this chapter.

6.14.1 Recurrence rules

What they do

Any job definition can be associated with zero to any number of schedules. The "schedule" object defines:

- a recurrence (see below), like * * * * * for "every minute"
- optionally, a queue which will override the default queue from the job definition
- optionally, a set of parameters which will override the ones inside the job definition.

Rule syntax

JQM uses standard crontab expressions. Here is a primer, taken from the documentation of the library used for parsing cron expressions:

There are five parts inside a cron pattern.

- Minutes: 0 to 59
- Hours: 0 to 23
- Days of month: 1 to 31 (L means last day of month)
- Month: 1 to 12 (or jan to dec)
- Day of week (0 sunday to 6 saturday, or sun-sat)

The star * means "any". The comma is used to specify a list of values for a part. The dash is used to specify a range.

The scheduler will trigger when all five parts of the pattern are verified true.

Examples:

- * * * * *: every minute
- 4 * * * *: every hour at minute 4.
- */5 * * * *: every five minute (00:00, 00:05, 00:10...)

- 3-18/5 * * * *: every 5 minutes between the third and the eighteenth minute of every hour. (00:03, 00:08, 00:13, 00:18, 01:03, . . .)
- */15 9-17 * * *: every 15 minutes during business hours (from 09:00 until 17:45).

Time zones

JQM always uses the system time zone to interpret the cron patterns. So if you have a cluster spanning multiple zones, this may lead to weird behaviour.

How to create them

The administrator will use the web administration "job definitions" page. It is a button enabled when a job definition is selected.

The programmer will use the JobRequest API, which has been extended.

class JobRequest

setRecurrence(*String cronExpression*)
: This actually creates a Scheduled Job for the given applicationName and optionally queue and parameters. (all other JobRequest elements are ignored). Note that when using this, there is no request immediately added to the queues - the actual requests will be created by the schedule.

setScheduleId(*Integer id*)
: This creates an occurrence from an existing schedule instead of simply from the job definition (this does not affect the recurrence itself which continues to run normally)

Also note that a JqmClient.removeRecurrence exists to remove schedules created by this method.

6.14.2 Starting later

Another type of scheduling is "one off" runs. This is done through the method:

JobRequest.**setRunAfter**(*Calendar after*)
: Put the request inside the queue but do not run it when it reaches the top of the queue. It will only be eligible for run when the given date is reached. When the given date is reached, standard queuing resumes. The resolution of this function is the minute: seconds and lower are ignored (truncated).

6.14.3 Manual scheduling

If you want to control the release time at will (the "run after" date is not known at the time of enqueuing) use this:

JobRequest.**startHeld**()
: Put the request inside the queue but do not run it until the JqmClient.resumeJob(int) method is called on the newly created job instance.

CHAPTER 7

In case of trouble

7.1 Troubleshooting

- When starting JQM, "address already in use" error appears. Change the ports of your nodes (by default a node is on a random free port, but you may have started multiple nodes on the same port)
- Problem with the download of the dependencies during the execution: Your nexus or repositories configuration must be wrong. Check your pom.xml or the JQM global parameters.

If your problem does not appear above and the rest of the documentation has no answer for your issue, please *open a ticket*.

7.2 Reporting bugs and requests

Please direct all bug reports or feature requests at our tracker on GitHub.

In case you are wondering if your feature request or bug report is well formatted, justified or any other question, feel free to mail the maintainer at mag@enioka.com.

The minimum to join to a bug report:

- the logs in TRACE mode
- if possible, your jobdef XMLs
- if possible, a database dump
- if concerning a payload API issue, the code in question

7.3 Bug report

Please direct all bug reports or feature requests at GitHub.

CHAPTER 8

Developement

This chapter is only useful for JQM developers. For would-be contributors, it is a must-read. Otherwise, it can be skipepd without remorse.

8.1 Contributing to JQM

JQM is an Open Source project under the Apache v2 license. We welcome every contribution through GitHub pull requests.

If you wonder if you should modify something, do not hesitate to mail the maintainer at mag@enioka.com with [JQM] inside the subject. It also works before opening feature requests.

JQM dev environment:

- Eclipse, latest version (standard for Enioka developers, it is of course possible to successfully use other IDEs or editors - use provided style xml) * Visual Studio Code configuration is also provided with the code, as it directly re-uses Eclipse formatting (and language server!).
- Maven 3.x (CLI, no Eclipse plugin) for dependencies, build, tests, packaging
- Sonar (through Maven. No public server provided, rules are `here`)
- Git

For the web application, a WebPack build is triggered by Maven. Developpers would simply use the "exec:exec" goal in the WS project to start a dev server with both Java and JS on port 8080.

Finally, please respect our coding style and conventions: they are described in the *Conventions and style* page.

8.2 Conventions and style

8.2.1 Java

Formatting

Our code style is c++ inspired, since the maintainers just hate having opening braces on the same line. This is a fully assumed personal bias, so please do not argue with it.

To ease the pain for the users of what we may call the "default Eclipse formatting", an Eclipse formatter is provided `here`). If you are an Eclipse user, please import it and associate it with the JQM project. (The Sonar rules we use are also included inside that directory.)

8.2.2 Database

General:

- always use upper-case letters for object names (whatever the database, even if case insensitive).
- in case multiple words are needed in a name, they are separated by an underscore.
- spaces and special characters are forbidden. Only A-Z and _ and 0-9.
- Always prefer choices compatible with ANSI SQL.
- Shorter names are better!

Main tables:

- the name of the table is the name of the notion contained by the table. E.g. QUEUE for the table containing the description of the different queues.
- always use singular: QUEUE, and not QUEUES.
- never more than 25 characters. If needed, use a shorter version of the notion (JOB_DEF instead of JOB_DEFINITION for example).

Constraints:

- constraints must all be named. Never use auto generated names as they make the schema harder to update. (exception for NOT NULL)
- PK: PK_TABLE_NAME
- FK: FK_TABLE_NAME_n where n is an integer incremented on each new FK. We choose not to mention the target table in the name, as it would make the name really long and unwieldy.
- UK: UK_TABLE_NAME_n
- All FK must be indexed. Always, even on small tables, to avoid some locks (abusive generalization, but this allows to avoid some nasty bugs)
- As much as possible, constraints are expressed inside the database rather than in code.

ID generation:

- use sequences if available. There should be one unique sequence for all needs.
- otherwise do the best you can. Beware MySQL auto ID, which are reset on startup.

Table columns:

- Name follows same guides as for table names.

- FK columns do not need to add _ID: an FK to a queue is named QUEUE.
- max length is 20 characters.
- for columns in the style of KEYWORD1, KEYWORD2, ... : no underscore between the final figure and the rest of the name.
- for strings, always use VARCHAR (or equivalent for your database) variable length data type. The length must be checked before insertion from the Java code.
- defaults should be expressed in Java code, not in the column definitions
- sometimes, the "good name" is a reserved SQL keyword: key, value... in that case just put another word next to it, without underscore (KEYNAME instead of KEY...)

Indices:

- For indices on constraints, IDX_CONSTRAINT_NAME
- Otherwise, IDX_TABLE_NAME_n

8.3 Release process

This is the procedure that should be followed for making an official JQM release.

8.3.1 Environment

The release environment must have:

- PGP & the release private key
- Access to a Sonar server with a correctly configured Maven settings.xml
- The Selenium setup (see *Testing*) - this has been deprecated. It may come back later.
- Internet access
- Login & password to Sonatype OSSRH with permissions on com.enioka.jqm.
- Login & password to Read the Docs with permissions on com.enioka.jqm.
- Docker client 18.06+ and access to the multi-arch build environment.

8.3.2 Update release notes

Add a chapter to the release notes & commit the file.

8.3.3 Checkout

Check out the branch master with git.

8.3.4 Full build & tests

There is no distinction between tests & integration tests in JQM so this will run all tests.

```
mvn clean install
```

8.3.5 Sonar snapshot

This will create a new Sonar analysis.

```
mvn sonar:sonar
```

Once done, take a snaphot in Sonar.

8.3.6 Release test

The release plug-in is (inside the pom.xml) parametrized to use a local git repository, so as to allow mistakes. During that step, all packages are bumped in version number, even if they were not modified.

```
mvn release:prepare -Darguments='-DskipTests'
mvn package
```

Then the test package must be test-deployed in a two-node configuration.

8.3.7 Release

This will upload the packages to the OSSRH staging repository.:

```
mvn release:perform -Darguments='-DskipTests'
```

OSSRH validation

Go to https://oss.sonatype.org/ and unstage (which means: close, then release the staged repository) the release. This will in time allow synchronization with Maven Central.

8.3.8 Git push

At this step, the release is done and the local git modifications can be pushed to the central git repository on GitHub.

> **Warning:** when using GitHub for Windows, tags are not pushed during sync. Using the command line is compulsory.

```
git push origin --tags
git push origin
```

(push tags before code to help RTD synchronization)

8.3.9 Documentation

Go to jqm.rtfd.org and change the default branch to the newly created tag.

8.3.10 GitHub upload

Create a release inside GitHub and upload the zip and tar.gz produced by the jqm-engine project. Add a link to the release notes inside.

Note: only do this **after** the documentation is up on ReadTheDocs. Creating a release sends a mail to followers, so any link to the doc would be dead otherwise.

8.3.11 Docker Hub upload

This updates the following tags:

- release tag
- major tag
- latest is updated to this release
- nightly is updated to the next upcoming version.

For maintenance releases of past majors, care must be taken to change the updated tags.

Changing version, run *$jqmVer="2.1.0"; $majorVer=$jqmVer.Split('.')[0]; $newTag="jqm-all-$jqmVer"; docker/Update-AllBranches.ps1 -Branches @{$jqmVer = $newTag; $majorVer = $newTag; "latest" = $newTag; "nightly" = "master"}*

You also may rebuild older branches - this updates OS and middlewares.

8.4 Release notes

8.4.1 2.1.0

Release goal

This release aimed at increasing compatibility with various development ecosystems, chief of which Docker and newer Java versions. Just run *docker run -it –rm -p 1789:1789 enioka/jqm* and go to http://localhost:1789 !

It is a simple upgrade with no breaking change.

Major changes

- Docker compatibility. Official images (Linux Alpine & Windows Nano) are released on the Docker Hub at https://hub.docker.com/r/enioka/jqm/ and are usable for many development and production scenarios. Read the documentation on the Docker Hub for more details - this is the pièce de résistance of the release.
- Java 9 and 10 compatibility. Note that Java 6 & 7 are still supported, but also still deprecated and will be removed in the next version. * Note that using the WS client will require to change the Jersey dependencies to newer one on Java 9+, as the older Java 6 compatible libraries used by default are not compatible with 9+.
- Oracle compatibility is back.
- Engine: on Java >= 7, the job instance class loader are now closed. On Windows, this means no more file locks remaining after run and therefore job jars are now hot swap-able.

Minor changes

- Engine: better db failure handling on MySQL and Oracle.
- Engine: will now wait for the database to be available on startup, allowing easier startup sequences.
- Engine: drivers and other libraries can now be placed in sub-folders of the "ext" directory (used to be: only at the root of ext).
- Client API: can now switch scheduled job instances from one queue to another, and cancel them.
- Simple API: new easier health check by an HTTP GET (equivalent to calling JMX bean AreAllPollersPolling).
- CLI: added possibility to apply a node template to a given node, allowing it to poll specific queues and other parameters.

Deprecated

No new entries - same list as for 2.0.x.

- The Maven artifact named "jqm-api-client-hibernate" has been removed, and replaced by a redirection to the jqm-api-cient-jdbc" artifact. The redirection will be removed in a future release.
- JqmClient.resumeJob is deprecated in favor of the strictly equivalent resumeQueuedJob (to avoid confusion between the different pause/resume verbs)
- Java 6 & 7, which are no longer supported, are considered deprecated in this release. Support for these versions will be removed in the next major version. The 2.x release is the last JQM version to fully support Java 6 & 7.

8.4.2 2.0.0

Release goal

We are excited to announce the release of JQM 2.0.0. This release is the first of the 2.x series. It is at core a major refactoring of the 1.4 code, which has enabled a few big new features and will allow many more in future versions.

Important note: Oracle support is not present in the initial release. It will be added again in the next release.

Major changes

Better integration with big frameworks:

- More class loading options: it is now possible to specify CL options on transient CL.
- New "starting job instance" event which can be used in user-provided handlers.
- New Spring context management, using the aforementioned event. JQM can now be a fully-fledged Spring container!

Client APIs:

- Many new client APIs to modify job instances.
- Running job instances can now be paused (in addition to being killed).
- New client APIs on queues : pause a queue, resume it...
- New client API to enqueue an instance in a frozen state (and unfreeze it).

- Queues, which used to be purely FIFO, can now use an optional priority parameter. This priority is also translated in Thread priority (the CPU quota for the job instance).

Performances:

- All but one explicit database locks have been eliminated. This means greater JQM cluster scalability and performance.
- Less memory usage. JQM 1.4 was about 40MN idle, 2.0 is 25MB.
- Startup time is now below one second without web services
- Far less libraries used, including in the tester module. (this includes removing Hibernate - JQM does not need an ORM anymore).

Administration:

- New integrated cron-like scheduler - no need anymore for a scheduler in simple cases.
- Beginning with the next version, upgrade scripts are provided when the database schema changes.
- Support for DB2 databases (v 10.5+).

Minor additions

- All components: it is now possible to prefix the name of the database tables.
- All components: no more log4j in the different modules - purely slf4j-api.
- Engine: better external launch logs.
- JDBC client API: no need anymore to specify the datasource name to use the Tomcat hack.
- WS client API: lots of reliability fixes and better logging both on client and server side.

Breaking changes

As the semantic versioning designation entails, this version conatains a few breaking changes. However, it should be noted that the code API (the Java interfaces) themselves have no breaking changes from version 1.4, so impact should be minimal - most changes are behind the scenes, and have consequences for the administrators only.

The breaking changes are:

- The client API implementation named "jqm-api-hibernate" has been replaced by the "jqm-api-jdbc" implementation (with a Maven redirection). The parameters have changed. If you were not using specific parameter (like a specific datasource JNDI name) it should be transparent, as defaults are the same.
- When using the client API, note that validation of the parameters is now stricter (this means failures now occur earlier). It may mean that a JqmInvalidRequestException is now thrown instead of a JqmClientException. If you were catching JqmException, it has no impact as it is the mother class of the two other.
- The JSF sample has been dropped (it was a demonstration of using the full client API in the context of a JSF2/PrimeFaces web application). Users may still look at the sample in version 1.4, as the API used have not changed. This was done because we do not want anyone to believe we encourage to use JSF for creating user interfaces with JQM.
- Web API user login is now case sensitive, as it should always have been.
- Then "mavenRepo" global parameter cannot be specified multiple times anymore. It now takes a list (comma separated) instead. All global parameters keys are now unique.

- Class loading options are no more given per job definition, but have a declaration of their own. This allows for a more consistent configuration, and should reduce confusion over how to configure class loaders. This impacts the deployment descriptor XML (XSD change).
- For those using the client API Webservice implementation, note that the system properties com.enioka.ws.url has been renamed com.enioka.jqm.ws.url, making it consistent with all the other properties.
- Killed jobs now consistently report as CRASHED. KILLED is no longer a job status, as instructions to running jobs are now handled properly outside the status of the job instance.

Also, a few changes may be breaking for those who were doing explicitly forbidden things, as a lot of internals have changed.

- The database schema has changed a lot. This was never an official API (and likely won't ever be one), but we know a few users were directly making changes in the database so we are listing it here.
- As a consequence the Java classes used to map the database have changed (or disappeared altogether). Same remark: was not an API.
- If you were using an unsupported database, it is it will very likely not work anymore - JQM has dropped using an ORM and therefore does not benefit from the abstraction it provided anymore. Supported databases (HSQLDB, Oracle, MySQL, PostgreSQL, DB2) of course continue to work.

Deprecated

- The Maven artifact named "jqm-api-client-hibernate" has been removed, and replaced by a redirection to the jqm-api-cient-jdbc" artifact. The redirection will be removed in a future release.
- JqmClient.resumeJob is deprecated in favor of the strictly equivalent resumeQueuedJob (to avoid confusion between the different pause/resume verbs)
- Java 6 & 7, which are no longer supported, are considered deprecated in this release. Support for these versions will be removed in the next major version. The 2.x release is the last JQM version to fully support Java 6 & 7.

8.4.3 1.4.1

Release goal

This is a feature release aiming at giving more control over the class loaders used by the engine.

Many other features are also included, see details below.

Upgrade notes

All API changes are backward compatible: 1.3.x APIs will work with 1.4.1 engines. However, everyone is strongly encouraged to upgrade to the latest version.

There are database structure modifications in this release, so the standard upgrade path must be used (with database drop).

Major

- Engine: added possibility (at job definition level) to share non-transient class loader with other jobs instances (created from the same job definition or from other job definitions). Default behaviour is still to use one isolated transient class loader per launch.

- Engine: added possibility (at job definition level) to use a child first or parent first class loader.
- Engine: added possibility (at job definition level) to trace the classes loaded by a job instance.
- Engine: added possibility (at job definition level) to hide classes from a job.
- Engine: added new "Maven" type of job - this type is fetched directly from a Maven repository without any need for local deployment.
- Engine: MySQL is now fully supported without reserves, and do not need a startup script anymore.
- GUI: updated to expose the new CL options.
- GUI: major frameworks upgrade - it should be more reactive.
- CLI: added option to export job definition XML (the deployment descriptor). This should help developers to create and maintain it.
- Dev API: added a helper class to embed a full JQM node in the JUnit tests of payloads.

Minor

- Query API: better handling of pagination.
- Client API: on enqueue, the job instance creation date now comes from the DB to avoid issues with time differences between servers.
- CLI: can now specify a port when creating a node.
- CLI: fixed 'root' account creation which was not in the right profile.
- GUI: added favicon to prevent browser warnings.
- Documentation: clarified some notions.
- Test: the 'send mail on completion' function is now correctly tested.
- Test: added testing on OpenJDK 8.

8.4.4 1.3.6

Release goal

Maintenance release with a few optimizations concerning the client API.

Upgrade notes

All API changes are backward compatible: 1.2.x and 1.3.x APIs will work with 1.3.6 engines. However, everyone is strongly encouraged to upgrade to the latest version.

No database modification in this release - upgrade can be done by simply replacing engine files.

Major

- Engine: a new JMX counter has been added so as to detect jobs longer than desired (a parameter set in the job definition).
- Engine: added an option to create an additional log file containing all the logs of all jobs. This should ease job log parsing by monitoring tools.

- Client API: extended QUery API results so as to return all the keywords (those set in the job definition and those set at enqueue time).
- Client API & Engine API can now cohabit inside a payload for the rare cases when the engine API is not enough.

Minor

- Client API: the job definition XSD is now included inside the jqm-api artifact to ease validation by payload developers.
- Client API: enqueue method should now run faster with less memory consumed.
- Client API: fixed a very rare race condition in file retrieval methods when WS authentication is enabled.
- Test: migrated to SonarQube+Jacoco & added necessary variables.

8.4.5 1.3.5

Release goal

Maintenance release for the integration scripts (jqm.sh and jqm.ps1).

Upgrade notes

No API change (APIs version 1.3.5 are the same as version 1.3.3). 1.2.x and 1.3.x APIs will work with 1.3.4 engines. However, everyone is strongly encouraged to upgrade to the latest version.

No database modification in this release - upgrade can be done by simply replacing engine files.

Major

Nothing.

Minor

- Scripts: The automatic kill on OutOfMemoryError now works on more Linux variants and on Windows.
- Scripts: JAVA_OPTS is now used in the Linux script in all commands (used to be used only on startup commands).
- Engine: fixed a case that had jobs with end date < start date (now everything uses the time of the central DB).
- Engine: better error message on Job Definition XML import error.
- Simplified Travis builds.

8.4.6 1.3.4

Release goal

Maintenance release.

Upgrade notes

No API change (APIs version 1.3.4 are the same as version 1.3.3). 1.2.x and 1.3.x APIs will work with 1.3.4 engines. However, everyone is strongly encouraged to upgrade to the latest version.

No database modification in this release - upgrade can be done by simply replacing engine files.

Major

- Engine: in some situations, highlander job execution requests could clog a queue. This has been fixed.

Minor

- Engine: A nagging transaction bug that only showed up in automated Travis builds has finally been squashed.
- GUI: double-clicking on "next page" in history screen will no longer open a detail window.
- GUI: a regression from 1.3.3 has been fixed - pagination no longer worked in history screen. (the refresh button had to be pressed after clicking the next page button)
- Test: Selenium is no longer used in the automated build.

8.4.7 1.3.3

Release goal

Maintenance release.

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. 1.2.x and 1.3.x APIs will work with 1.3.3 engines. However, everyone is strongly encouraged to upgrade.

No database modification in this release - upgrade can be done by simply replacing engine files.

Major

- Admin UI: the history page was enhanced with more filters including date filters.
- Engine: the Unix/Linux startup script was modified so as to kill automatically the engine when an OutOfMemoryError occurs. This can be overridden with environment variables.

Minor

- CLI: XML schema of deployment descriptors is now validated on installations (was disabled previously due to issues on IBM J9 JVM).
- Client API: files downloaded are now briefly stored in the system temp directory instead of a subdirectory. This makes it easier to have multiple JQM engines running with different accounts on the same server.
- Client API: can now filter by node name.
- Engine: highlander status is now correctly archived in the history table (used to be always false).

8.4.8 1.3.2

Release goal

Maintenance release.

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. 1.2.1 & 1.2.2 and 1.3.1 apis will work with 1.3.2 engines. However, as 1.2.2 contains fixes and 1.3.1 new functionalities, everyone is strongly encouraged to upgrade.

No database modification in this release - upgrade can be done by simply replacing engine files.

Major

Nothing.

Minor

- Engine: added a JDBC connection leak hunter to prevent some leak cases
- CLI: added a CLI option to modify an administration JQM user
- GUI: fixed randomly hidden JNDI resource parameters
- Client API: fixed hedge case in which a job instance may not be found by getJob()
- Providers: fixed print job name and added option to specify requesting user name

8.4.9 1.3.1

Release goal

This release had one goal: reducing the need for engine restart. Other administration usability tweaks are also included.

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. 1.2.1 & 1.2.2 apis will work with 1.3.1 engines. However, as 1.2.2 contains fixes and 1.3.1 new functionalities, everyone is strongly encouraged to upgrade.

Database must be rebuilt for version 1.3.1, this means History purge.

Major

- Engine: will automatically reload some parameters when they change, reducing the need for engine restarts
- Engine: now resists better database failures
- Engine API: shouldKill method is now throttled, reducing the database hammering (as this method is called by all other methods)
- Admin API: added a method to retrieve the engine logs

- Client API & GUI: can now download files created by a job instance even if it has not finished yet

Minor

- Engine: added sample purge job
- GUI: added an online log viewer for job instance logs (no need to download log files anymore)
- GUI: added an online log viewer for engine logs (which were not retrievable through the GUI before)
- GUI: allowed column resize on History panel
- GUI: added an option to view only KO job instances
- Engine: small code refactor

8.4.10 1.2.2

Release goal

This is a maintenance release, containing mostly bugfixes and very few new features that could not be included in the previous version (mostly administration GUI tweaks).

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. 1.2.1 apis will work with 1.2.2 engines. However, as 1.2.2 contains fixes, everyone is strongly encouraged to upgrade.

Database must be rebuilt for version 1.2.2, this means History purge.

Major

- Engine: can now resist a temporary database failure

Minor

- Engine: access log now logs failed authentications
- Engine: various minor bugfix in extreme performance scenarios
- Engine: there is now one log file per node
- Client API: various fixes
- Client API: now support retrieval of running job instance logs
- GUI: various minor improvements
- CLI: jobdef reimport fixes
- Tests: major refactoring with 3x less Maven artifacts

8.4.11 1.2.1

Release goal

The main goal of this release was to simplify the use of JQM. First for people who dislike command line interfaces, by adding a graphical user interface both for administration and for daily use (enqueue, check job status, etc). Second, for payload developers by adding a few improvements concerning testing and reporting.

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. Please note that the only version that will work with engine and database in version 1.2.1 is API version 1.2.1: upgrade is compulsory.

Database must be rebuilt for version 1.2.1, this means History purge.

Major

- Client API: Added a fluid version of the JobRequest API
- GUI: Added an administration web console (present in the standard package but disabled by default)
- All APIs: Added an authentication system for all web services, with an RBAC back-end and compatible with HTTP authentication as well as SSL certificate authentication
- Tests: Added a payload unit tester
- General: Added mail session JNDI resource type

Minor

- Client API: Client APIs file retrieval will now set a file name hint inside an attachment header
- Client API: Added an IN option for applicationName in Query API
- Client API: Query API optimization
- Engine: Unix/Linux launch script is now more complete and robust (restart works!)
- Engine: JAVA_OPTS environment variable is now used by the engine launch script
- Engine: Added special "serverName" JNDI String resource
- Engine: All automatic messages (was enqueued, has begun…) were removed as they provided no information that wasn't already available
- Engine: In case of crash, a job instance now creates a message containing "Status changed: CRASHED due to " + first characters of the stacktrace
- Engine: Log levels and content were slightly reviewed (e.g.: stacktrace of a failing payload is now INFO instead of DEBUG)
- Engine API: Added more methods to the engine API (JobManager)
- Tests: Refactored all engine tests
- Documentation: clarified class loading structure
- Documentation: general update. Please read the doc. Thanks!
- General: Jobs can now easily be disabled

8.4.12 1.1.6

Release goal

This release was aimed at making JQM easier to integrate in production environments, with new features like JMX monitoring, better log file handling, JDBC connection pooling, etc.

A very few developer features slipped inside the release.

Upgrade notes

No breaking changes.

Compatibility matrix:

Version 1.1.6 / Other version	Engine	Client API	Engine API
Engine		>= 1.1.4	>= 1.1.4
Client API	== 1.1.6		
Engine API	>= 1.1.5		

How to read the compatibility matrix: each line corresponds to one JQM element in version 1.1.6. The different versions given correspond to the minimal version of other components for version 1.1.6 to work. A void cell means there is no constraint between these components.

For exemple : a payload using engine API 1.1.6 requires at least an engine 1.1.5 to work.

Major

- Documentation: now in human readable form and on https://jqm.readthedocs.org
- Distribution: releases now published on Maven Central, snapshots on Sonatype OSSRH.
- Engine: added JDBC connection pooling
- Engine: added JMX monitoring (local & remote on fixed ports). See http://jqm.readthedocs.org/en/latest/admin/jmx.html for details
- Engine: each job instance now has its own logfile
- Engine: it is now impossible to launch two engines with the same node name (prevent startup cleanup issues creating data loss)
- Engine: failed job requests due to engine kill are now reported as crashed jobs on next engine startup
- Engine: added UrlFactory to create URL JNDI resources
- Engine: dependencies/libs are now reloaded when the payload jar file is modified or lib folder is modified. No JQM restart needed anymore.

Minor

- Engine API: legacy JobBase class can now be inherited through multiple levels
- Engine: incomplete payload classes (missing parent class or lib) are now correctly reported instead of failing silently
- Engine: refactor of main engine classes

- Engine: races condition fixes in stop sequence (issue happening only in JUnit tests)
- Engine: no longer any permanent database connection
- Engine: Oracle db connections now report V$SESSION program, module and user info
- Engine: logs are less verbose, default log level is now INFO, log line formatting is now cleaner and more readable
- General: Hibernate minor version upgrade due to major Hibernate bugfixes
- General: cleaned test build order and artifact names

8.4.13 1.1.5

Release goal

Bugfix release.

Upgrade notes

No breaking changes.

Major

Nothing

Minor

- Engine API: engine API enqueue works again
- Engine API: added get ID method
- Db: index name shortened to please Oracle

8.4.14 1.1.4

Release goal

This release aimed at fulfilling all the accepted enhancement requests that involved breaking changes, so as to clear up the path for future evolutions.

Upgrade notes

Many breaking changes in this release in all components. Upgrade of engine, upgrade of all libraries are required plus rebuild of database. *There is no compatibiliy whatsoever between version 1.1.4 of the libraries and previous versions of the engine and database.*

Please read the rest of the release notes and check the updated documentation at https://github.com/enioka/jqm/blob/master/doc/index.md

Major

- Documentation: now fully on Github
- **Client API: - breaking - is no longer static. This allows:**
  - to pass it parameters at runtime
  - to use it on Tomcat as well as full EE6 containers without configuration changes
  - to program against an interface instead of a fully implemented class and therefore to have multiple implementations and less breaking changes in the times to come
- Client API: - **breaking** - job instance status is now an enum instead of a String
- Client API: added a generic query method
- Client API: added a web service implementation in addition to the Hibernate implementation
- Client API: no longer uses log4j. Choice of logger is given to the user through the slf4j API (and still works without any logger).
- Client API: in scenarios where the client API is the sole Hibernate user, configuration was greatly simplified without any need for a custom persistence.xml
- Engine: can now run as a service in Windows.
- Engine: - **breaking** - the engine command line, which was purely a debug feature up to now, is officialized and was made usable and documented.
- Engine API: now offers a File resource through the JNDI API
- Engine API: payloads no longer need to use the client or engine API. A simple static main is enough, or implementing Runnable. Access to the API is done through injection with a provided interface.
- Engine API: added a method to provide a temporary work directory

Minor

- Engine: various code refactoring, including cleanup according to Sonar rules.
- Engine: performance enhancements (History is now insert only, classpaths are truly cached, no more unzipping at every launch)
- Engine: can now display engine version (CLI option or at startup time)
- Engine: web service now uses a random free port at node creation (or during tests)
- Engine: node name and web service listeing DNS name are now separate notions
- Engine: fixed race condition in a rare high frequency scenario
- Engine: engine will now properly crash when Jetty fails to start
- Engine: clarified CLI error messages when objects do not exist or when database connection cannot be established
- Engine: - **breaking** - when resolving the dependencies of a jar, a lib directory (if present) now has priority over pom.xml
- Engine tests: test fixes on non-Windows platforms
- Engine tests: test optimization with tests no longer waiting an arbitrary amount of time
- Client API: full javadoc added

- Engine API: calling System.exit() inside payloads will now throw a security ecveption (not marked as breaking as it was already forbidden)
- General: - **breaking** - tags fields (other1, other2, . . .) were renamed "keyword" to make their purpose clearer
- General: packaging now done with Maven

8.4.15 1.1.3

Release goal

Fix release for the client API.

Major

- No more System.exit() inside the client API.

Minor

Nothing

8.5 Classloading

JQM obeys a very simple classloading architecture, respecting the design goal of simplicity and robustness (to the expense of PermGen/Metaspace size).

The engine classloader stack is as follows (bottom of the stack is at the bottom of the table):

In the engine:

System class loader (JVM provided - type AppClassLoader). Used by the engine itself (except JNDI calls).
Extension class loader (JVM provided - no need in JQM)
Bootstrap class loader (JVM provided)

For payloads:

Payload class loader (JQM provided - type JarClassLoader). Loads the libs of payloads from .m2 or from the payload's "lib" directory
JNDI class loader (JQM provided - type URLClassloader) Loads everything inside JQM_ROOT/ext
Extension class loader (JVM provided - no need in JQM)
Bootstrap class loader (JVM provided)

The general idea is:

- The engine uses the classic JVM-provided AppClassLoader for everything concerning its internal business
- Every payload launch has its own classloader, **totally independent from the engine classloader**. This classloader is garbage collected at the end of the run.

- JNDI resources in singleton mode (see *Using resources*) must be kept cached by the engine, so they cannot be loaded through the transient payload classloader itself. Also, the loaded resources must be understandable by the payloads - and therefore there must be a shared classloader here. The JNDI classloader is therefore the parent of the payload classloaders. The JNDI classloader itself has no parent (save obviously the bootstrap CL), so the payloads won't be polluted by anything foreign.

Advantages:

- The engine is totally transparent to payloads, as the engine libraries are inside a classloader which is not accessible to payloads.
- It allows to have multiple incompatible versions of the same library running simultaneously in different payloads.
- It still allows for the exposition to the payload of an API implemented inside the engine through the use of a proxy class, a pattern designed explicitly for that use case.
- Easily allows for hot swap of libs and payloads.
- Avoids having to administer complex classloader hierarchies and keeps payloads independent from one another.

Cons:

- It is costly in terms of PermGen/metaspace: if multiple payloads use the same library, it will be loaded once per payload, which is a waste of memory. If there are costly shared resources, they can however be put inside ext - but the same version will be used by all payloads on the engine.
- In case the payload does something stupid which prevents the garbage collection of at least one of its objects, the classloader will not be able to be garbage collected. This a huge memory leak (usually called a classloader leak). The best known example: registering a JDBC driver inside the static bootstrap-loaded DriverManager. This keeps a reference to the payload-context driver inside the bootstrap-context, and prevents collection. This special case is the reason why singleton mode should always be used for JDBC resources.
- There is a bug inside the Sun JVM 6: even if garbage collected, a classloader will leave behind an open file descriptor. This will effectively prevent hot swap of libs on Windows.

To alleviate some of the conses, JQM also provides options to share class loaders between different job instances and reuse them between runs, but this is at the cost of some of the pros. It is disabled by default.

All in all, this solution is not perfect (the classloader leak is a permanent threat) but has so many benefits in terms of simplicity that it was chosen. This way, there is no need to wonder if a payload can run alongside another - the answer is always yes. There is no need to deal with libraries - they are either in libs or in ext, and it just works. The engine is invisible - payloads can consider it as a pure JVM, so no specific development is required.

The result is also robust, as payloads have virtually no access to the engine and can't set it off tracks.

8.6 Testing

JQM is tested through an series of automated JUnit tests. These tests are usage-oriented (*integration tests*) rather than unit oriented. This means: every single functionality of JQM must have (at least) one automated test that involves running a job inside a full engine.

8.6.1 Automated builds

Travis

The project has a public CI server on http://travis-ci.org/enioka/jqm.

Selenium

The project has a public Selenium server at https://saucelabs.com/u/marcanpilami

8.6.2 Tests

Standard tests

These are the tests that **should always be run before any commit**. Any failure fails the build.

They are run though Maven (mvn clean install test) and should be able to run without any specific configuration. They are always run by Travis.

Selenium tests

The UI also has a few dedicated tests that run inside Selenium. To avoid configuration and ease test reproducibility, we use Sauce Labs' cloud Selenium. The Travis build uses the maintainer's account.

As this account is personal, its credentials are not included inside the build descriptor and these tests are disabled by default (they are inside a specific Maven profile). In order to use them, a free account on Sauce Labs is required, as well as putting this inside Maven's settings.xml:

```
<profile>
    <id>selenium</id>
    <activation>
        <activeByDefault>false</activeByDefault>
    </activation>
    <properties>
        <SAUCE_USERNAME>YOUR_USER_NAME</SAUCE_USERNAME>
        <SAUCE_ACCESS_KEY>YOUR_ACCESS_KEY</SAUCE_ACCESS_KEY>
        <SAUCE_URL>localhost:4445/wd/hub</SAUCE_URL>
    </properties>
</profile>
```

Moreover, as the web application actually runs on the developer's computer and not on the Selenium server, a tunnel must be activated, using Sauce Connect. The URL above reflects this.

Note: the Sauce Connect Maven plugin was not included in the pom, because it implies starting and stopping the tunnel on each test run - and this is a very long process. It's easier on the nerves to simply start the tunnel and forget it.

Finally, running the tests is simply done by going inside the jqm-wstst project and running the classic "mvn test -Pselenium" command. Obviously, if in the settings.xml file the profile was marked as active by default, the -P option can be omitted.

Web-services dev and tests

The admin GUI as well as all the web services are inside the jqm-ws project.

To develop and test this project in Eclipse, one needs a fully working JQM database. The easiest way to get it is to install a local node following the documentation. Then enable the admin GUI & create the root account with the command line. Do not enable SSL.

The node can be stopped - it won't be needed anymore.

Then, inside Eclipse, install a Tomcat 7. (not 8 - this would require Java 7).

The project contains a context.xml file inside src/test/webapp/META-INF that must be updated with the connection string to your database. Please do not commit these modifications.

Warning: you must ensure the src/test/webapp/META-INF directory is inside the "deployment assembly" inside Eclipse's project properties.

Then the database driver to the the lib directory of Tomcat

Everything is ready - the project can now be "run on server". The URL will be http://localhost:8080/jqm-ws

CHAPTER 9

Glossary

Payload the actual Java code that runs inside the JQM engine, containing business logics. This is must be provided by the application using JQM.

Job Definition

JobDef the metadata describing the payload. Also called JobDef. Entirely described inside the JobDef XML file. Identified by a name called "Application Name"

Job Request the action of asking politely the execution of a *JobDef* (which in turn means running the payload)

Job Instance the result of of a Job Request. It obeys the Job Instance lifecycle (enqueued, running, endded, . . .). It is archived at the end of its run (be it successful or not) into the history.

JQM Node

JQM Engine an instance of the JQM service (as in 'Windows service' or 'Unix init.d service') that can run payloads

Job queue

Queue a virtual FIFO queue where *job requests* are lined up. These queues are polled by some *nodes*.

Enqueue the action of putting a new *Job Request* inside a *Queue*. The queue is usually determined by the *JobDef* which holds a default queue.

Index

Symbols

-createnode <nodeName>
    command line option, 62
-enqueue <applicationname>
    command line option, 62
-exportallqueues <xmlpath>
    command line option, 62
-h, –help
    command line option, 62
-importjobdef <xmlpath>
    command line option, 62
-importqueuefile <xmlpath>
    command line option, 62
-p, –resources
    command line option, 62
-port <portNumber>
    command line option, 62
-startnode <nodeName>
    command line option, 62
-v, –version
    command line option, 62

A

addDeliverable() (JobManager method), 23
application() (JobManager method), 23
applicationName() (JobManager method), 22

C

canBeRestarted() (JobManager method), 22
cancelJob() (JqmClient method), 42
command line option
    -createnode <nodeName>, 62
    -enqueue <applicationname>, 62
    -exportallqueues <xmlpath>, 62
    -h, –help, 62
    -importjobdef <xmlpath>, 62
    -importqueuefile <xmlpath>, 62
    -p, –resources, 62
    -port <portNumber>, 62
    -startnode <nodeName>, 62
    -v, –version, 62

D

defaultConnect() (JobManager method), 24
deleteJob() (JqmClient method), 42

E

Enqueue, **113**
enqueue() (JobManager method), 23
enqueue() (JqmClient method), 42
enqueueFromHistory() (JqmClient method), 42
enqueueSync() (JobManager method), 23

G

getActiveJobs() (JqmClient method), 43
getCumulativeJobInstancesCount() (JqmEngineMBean method), 65
getCumulativeJobInstancesCount() (PollingMBean method), 66
getCurrentActiveThreadCount() (PollingMBean method), 65
getCurrentlyRunningJobCount() (JqmEngineMBean method), 65
getCurrentlyRunningJobCount() (PollingMBean method), 66
getDefaultConnection() (JobManager method), 24
getDeliverableContent() (JqmClient method), 43
getJob() (JqmClient method), 43
getJobDefinition() (JqmClient method), 44
getJobDefinitions() (JqmClient method), 44
getJobDeliverables() (JqmClient method), 43
getJobDeliverablesContent() (JqmClient method), 43
getJobLogStdErr() (JqmClient method), 44
getJobLogStdOut() (JqmClient method), 43
getJobMessages() (JqmClient method), 43
getJobProgress() (JqmClient method), 43
getJobs() (JqmClient method), 43
getJobsFinishedPerSecondLastMinute() (JqmEngineMBean method), 65

getJobsFinishedPerSecondLastMinute() (PollingMBean method), 66
getMaxConcurrentJobInstanceCount() (PollingMBean method), 66
getPollingIntervalMilliseconds() (PollingMBean method), 66
getQueues() (JqmClient method), 44
getUptime() (JqmEngineMBean method), 65
getUserActiveJobs() (JqmClient method), 43
getVersion() (JqmEngineMBean method), 65
getWorkDir() (JobManager method), 24

H

hasEnded() (JobManager method), 23
hasFailed() (JobManager method), 23
hasSucceeded() (JobManager method), 23

I

isActuallyPolling() (PollingMBean method), 66
isAllPollersPolling() (JqmEngineMBean method), 65
isFull() (JqmEngineMBean method), 65
isFull() (PollingMBean method), 66

J

Job Definition, **113**
Job Instance, **113**
Job queue, **113**
Job Request, **113**
jobApplicationId() (JobManager method), 22
JobDef, **113**
JobDef (built-in class), 44
jobInstanceID() (JobManager method), 22
JobManager (built-in class), 22
JobRequest (built-in class), 44, 87
JQM Engine, **113**
JQM Node, **113**
JqmClient (built-in class), 42
JqmEngineMBean (built-in class), 65

K

keyword1() (JobManager method), 23
keyword2() (JobManager method), 23
keyword3() (JobManager method), 23
kill() (LoaderMBean method), 66
killJob() (JqmClient method), 42

L

LoaderMBean (built-in class), 66

M

module() (JobManager method), 23

P

parameters() (JobManager method), 23
parentId() (JobManager method), 22
pauseQueuedJob() (JqmClient method), 42
Payload, **113**
PollingMBean (built-in class), 65

Q

Queue, **113**
Queue (built-in class), 44

R

restartCrachedJob() (JqmClient method), 43
resumeJob() (JqmClient method), 42

S

sendMsg() (JobManager method), 23
sendProgress() (JobManager method), 23
sessionID() (JobManager method), 23
setRecurrence() (JobRequest method), 87
setRunAfter() (JobRequest method), 87
setScheduleId() (JobRequest method), 87
startHeld() (JobRequest method), 87
stop() (JqmEngineMBean method), 65
stop() (PollingMBean method), 65

U

userName() (JobManager method), 23

W

waitChild() (JobManager method), 23
waitChildren() (JobManager method), 23

Y

yield() (JobManager method), 24