JQM

Release

June 20, 2014

Contents

CHAPTER 1

Introduction

JQM (short for Job Queue Manager) is a middleware allowing to run arbitrary Java code asynchronously on a distributed network of servers. It was designed as an application server specifically tailored for making it easier to run (Java) batch jobs asynchronously, removing all the hassle of configuring libraries, throttling executions, handling logs, forking new processes & monitoring them, dealing with created files, and much more... It should be considered for batch 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 adapted way).

It should also be considered for its ability to untangle the execution itself from the program that requires it. Two of the most obvious cases are:

- getting long running jobs out of the application server. An application server is not supposed to handle these, which fill up its queues and often end in weird timeouts and runaway threads. JQM will host these externalized jobs in an async way, not requiring the application server to wait for completion. Execution can happen on another server/VM, freeing resources (and potentially licence costs).
- 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.

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 it also optionaly 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.

Interacting with JQM is also easy: an API, with two different implementations (JPA & 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, ...).

CHAPTER 2

Documentation

2.1 How JQM works

2.1.1 Concepts

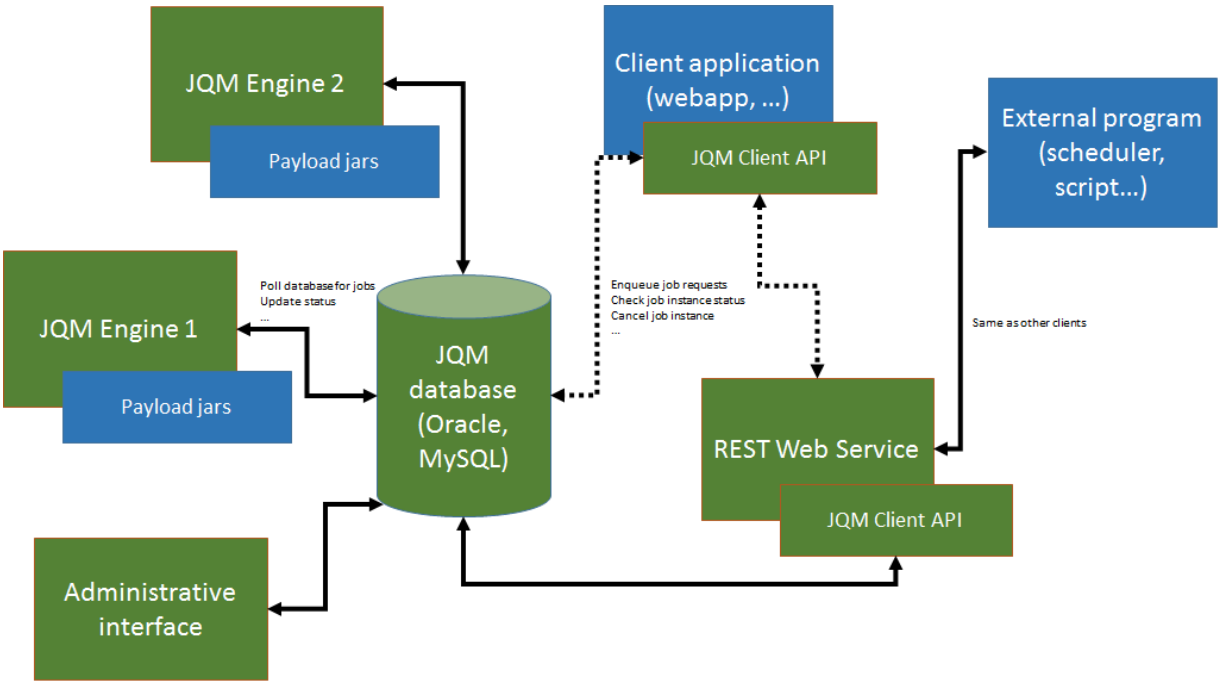
The goal of JQM is to launch *payloads*, i.e. Java code doing something useful for the end user, aysnchronously. The payload is described inside a *job definition* - so that JQM knows things like the class to load, the path of the jar file, etc. A running payload is called a *job instance* (the "instance of the job definition"). To create a job instance, a *job request* is posted by a client. It contains things such as parameter values, and references a job definition so that JQM will know what to run.

2.1.2 Definitions

Full definitions are given inside the *Glossary*.

Name	Definition
Payload	the actual Java code that runs inside the JQM engine, containing business logics
Job Definition	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, ...)
JQM Node	an instance of the JQM service that can run payloads
JQM Engine	synonym to JQM Node

2.1.3 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 app 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 option 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 free room takes the job instance
- it creates a classloader for this job instance, imports the correct libraries inside 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 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 expose this issue.

2.1.4 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.

An engine (a node) is associated to as many queues as needed. The engine will poll job instances that are posted on these queues. The polling frequency is defined per node/queue association: it is possible to have one engine polling very often a queue while another polls slowly the same queue (minimum period: 1s). Also, the number of slots is defined at the same level: 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. When all slots of all nodes polling a given queue are filled, job instances stay in the queue, waiting for a slot to be freed. Note that it also allows some queues to be defined only on some nodes and not others, therefore giving some control over where payloads are actually run.

A Job Definition (JobDef) is associated to a queue. It is the jobdef default queue: all job requests pertaining to a jobdef are created inside the jobdef default queue. It is possible, once created, to move a job instance from one queue to another as long as it has not already run.

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).

2.2 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 dev environments. It also gives pointers to the general documentation.

2.2.1 Windows

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterward.
- 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"
$JQM_VERSION = "${project.version}"  ## For example: "1.1.6"
mkdir -Force $JQM_ROOT; cd $JQM_ROOT
Invoke-RestMethod https://github.com/enioka/jqm/archive/jqm-$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
./jqm.ps1 allxml  # This will import all the test job definitions
```

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.

2.2.2 Linux / Unix

Prerequisites:

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

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

```
wget https://github.com/enioka/jqm/archive/jqm-1.1.6.tar.gz # For 1.1.6 release. Adapt it to the one
tar xvf jqm-1.1.6.tar.gz
```

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

```
cd jqm-1.1.6
./jqm.sh createnode
./jqm.sh allxml  # This will import all the test job definitions
```

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.

2.2.3 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.

2.3 Payload development

2.3.1 Payload basics

The goal of JQM is to run some Java code containing some business logic: the *payload*. This chapter describes how to write the Java code that will really be executed. The philosophy is: JQM is able to run most existing code without adaptations, and taking advantage of some JQM functionalities is easy to add to any code be it new or existing.

Payloads types

There are three payload types: just launching the good old main (the preferred method for newly written jobs), and two types designed to allow reuse of even more existing binaries: Runnable & JQM API.

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 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 - therefore 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

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
```

```
{
        @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>
```

Accessing the JQM engine API

Often, a job will need to 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 JQM payload type already has one JobManager field named jm defined in the base class JobBase - it would have been stupid not to define it while the API is always present for that payload type.

The dependency is:

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

Creating files

An important use case for JQM is the generation of reports at the direct request of an end-user through a web interface. This report is too long to generate on the application server (timeout), or blocking a thread for a user is unacceptable: the generation must be deported elsewhere. JQM has methods to do that.

The report generation is the payload - but how should the file be sent to the end user? 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 it is definitely not on the application server, so not easy to access from the web interface). 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.

TL;DR: when a file is created that should be accessible to remote clients, use JobManager.addDeliverable

---

Note: work directories are obtained through JobManager.getWorkDir. These are purged after execution. Never use a temporary java file - these are purged on JVM exit - which on the whole never happens inside an application server.

Example:

```
import java.io.FileWriter;
import java.io.PrintWriter;
import com.enioka.jqm.api.JobBase;

public class App extends JobBase
{
        @Override
        public void start()
        {
                String file = this.getParameters().get("filepath");
                String fileName = this.getParameters().get("fileName");
                try
                {
                        PrintWriter out = new PrintWriter(new FileWriter(file + fileName));
                        out.println("Hello World!");
                        out.close();
                        addDeliverable(file + fileName, "JobGenADeliverableFamily");
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
        }
}
```

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 *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 not 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.

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)
                {
                        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(),
                        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.
                        System.out.println("New request is number " + i + " and should be done now");
                        jm.sendProgress(100);
                }
        }
}
```

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!

Staying reasonable

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

- Don't play (too much) with classloaders. This 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 classloaders.
- Be wary of bootstrap static contexts. Using static elements is all-right as long as the static context is from your classloader (in our case, it means classes from your own code or dependencies). Messing with static elements from the bootstrap classloader 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.
- 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*.

2.3.2 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**.

2.3.3 Using resources

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 can be used for:

- JDBC connections
- JMS resources
- Files
- URLs
- 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.

JNDI resources

Using

This is vanilla JNDI inside the root JNDI context:

```
DataSource ds = (DataSource) NamingManager.getInitialContext(null).lookup("jdbc/superalias");
```

Please note the "null" for the context lookup: JQM only uses a root context. See below for details.

Defining

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`.

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.

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 definitiely 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, classloader 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 its classloader. 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). 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 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.

Examples

JDBC

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

Using

```
DataSource ds = (DataSource) NamingManager.getInitialContext(null).lookup("jdbc/superalias");
```

It could of interest 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 getDefaultConnection method of the JQM API. (this is the only JNDI-related element that requires the API).

Defining

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

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 options, detailed in the Tomcat JDBC documentation.

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 ext.

Using

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

```
import javax.jms.TextMessage;
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: MARSUPILA

                        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();
                }
        }
}
```

Defining

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

Exemple 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)

Exemple for MQ Series Queue:

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

Parameter name	Value
QU	queue name

Exemple 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)

Exemple for ActiveMQ Queue:

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

Parameter name	Value
physicalName	queue name

Files

Provided by the engine - these resources must therefore always be singletons.

Using

```
File f = (File) NamingManager.getInitialContext(null).lookup("fs/superalias");
```

Defining

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

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

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

URL

Provided by the engine - these resources must therefore always be singletons.

Using

```
URL f = (URL) NamingManager.getInitialContext(null).lookup("url/testurl");
```

Defining

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

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

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

2.3.4 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 exemple, 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. A code can only run if its XML has been imported (or the corresponding values manually entered inside the database, a fully unsupported alternative way to do it).

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

Libraries handling

JQM itself does not provide any libraries to the payloads - all its internal classes are hidden. But there are two ways, each with two variants, to make sure the required libraires are present at runtime.

Note: All the four variants are exclusive. **Only one libray 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. (the repositories used can be parameterized)

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.

Conclusion: in that case, no packaging to do.

Warning: using this means the pom is fully resolvable from the engine server. This includes every parent pom.xml used.

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.

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>

                <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)

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.

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>
                <filePath>jqm-test-fibo/</filePath>

                <jobdefinitions>
                        <jobDefinition>
                                <name>Fibo</name>
                                <description>Test based on the Fibonachi suite</description>
                                <canBeRestarted>true</canBeRestarted>
                                <javaClassName>com.enioka.jqm.tests.App</javaClassName>
                                <maxTimeRunning>42</maxTimeRunning>
                                <application>CrmBatchs</application>
                                <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>
                        <maxTimeRunning>42</maxTimeRunning>
                        <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.

2.3.5 Testing payloads

A this step the following artifacts exist:

- a JAR file containing the payload
- a descriptor XML file containing all the metadata

Moreover, this section makes the assumption that you have a working JQM engine at your disposal. If this is not the case, please read *Installation*.

Copy files

Place the two files inside JQM_DIR/jobs/xxxxx where xxxxx is a directory of your choice. Please note that the name of this directory must be the same as the one inside the "filePath" tag from the XML.

If there have libraries to copy (pom.xml is not used), they must be placed inside a directory named "lib".

Example (with explicit libraries):

```
$JQM_DIR\
$JQM_DIR\jobs\
$JQM_DIR\jobs\myjob\myjob.xml
$JQM_DIR\jobs\myjob\myjob.jar
$JQM_DIR\jobs\myjob\lib\
$JQM_DIR\jobs\myjob\lib\mylib1.jar
$JQM_DIR\jobs\myjob\lib\mylib2.jar
```

Note: there is no need to restart the engine on any import, jar modification or whatever.

Import the metadata

Note: this only has to be done the first time. Later, this is only necessary if the XML changes. Each time the XML is imported, it overwrites the previous values so it can also be done at will.

Open a command line (bash, powershell, ksh...) and run the following commands (adapt JQM_DIR and xxxx):

```
cd $JQM_DIR
java -jar jqm.jar -importjobdef ./jobs/xxxxx/xxxx.xml
```

Run the payload

This part can be run as many times as needed. (adapt the job name, it is the "name" attribute from the XML)

```
java -jar jqm.jar -enqueue JOBNAME
```

The logs are inside JQM_ROOT/logs. The user may want to do "tail -f" (or "cat -Wait" in PowerShell) on these files during tests. There are two files per launch: one containing the standard output flow, the other with the standard error flow.

2.4 Client development

2.4.1 Interacting with JQM

Client API

The **client API** enables any Java (and other languages for some implementations of the API) program 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...

To use it, one of the two implementations of the client API must be imported: either *an Hibernate JPA 2.0 client* with jqm-api-client-hibernate.jar or *a 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. Suffice to say that it contains many methods related to:

- enqueing 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 exemple, 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.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 provider, 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 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).

From scripts

A very rudimentary "web service" exists to allow for curl/wget-style interaction. (it has nothing to do with the full web service API, which is optional. The script API is always present). Basically, it allows to easily launch a JQM job from a job scheduler.

To enqueue a job, POST on /enqueue with the following POST compulsory parameters: application name, user and the following POST optional parameters: module, mail, other1, other2, other3, parentid, sessionid. Parameters are given as param_nn for the name and paramvalue_nn for the value. Signification of these is the same as in the client API. The server answers with the ID of the request.

To retrieve the status of a request, GET on /status?id=nnn Status is given as text.

Warning: there is no authentication. It will be implemented one day as an option. See ticket #9.

From the command line

A few options of the engine command line allow the same kind of limited interaction with JQM than the s"script API" described in the previous paragraph.

See *Command Line Interface (CLI)* for details.

2.4.2 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 to JQM servers 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.

Using the Hibernate client

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). In Maven terms:

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

Please note that if your container provides a JPA provider by itself, there is obviously no need for the second dependency 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. 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

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 must 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-1.1.4-SNAPSHOT.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>
        <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.

Optional parameters

In both cases, 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) 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 exemple changed (or set in the first place) the <non-jta-datasource> to some JNDI alias.

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);
```

2.4.3 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 exemple, to list all running jobs:

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

The specific parameters are:

Name	Compulsory	Description	Example
com.enioka.ws.url	YES	The base URL of the web service	http://localhost:1789/api/ws

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.ws.url", "http://localhost:9999/marsu/ws");
  JqmClientFactory.setProperties(p);
  ```

- through a system parameter (-Dcom.enioka.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 exemple, 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 by default. To activate it, you must drop the file jqm-ws.war inside a directory (that you must create) named "webapp". This directory should be inside the JQM engine root (alongside conf, lib, ...) and the OS account running the JQM service should have full permissions on it. JQM node must then be restarted.

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.

Warning: currently, there is no authentication mechanism implemented. See ticket #9 for the implementation of this function.

Service reference

All objects are serialized to XML. 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

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`.

2.4.4 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.DATEATTRIBU
```

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.

2.5 Administration

2.5.1 Installation

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

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 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 $JQM_ROOT
Invoke-RestMethod https://github.com/enioka/jqm/archive/jqm-$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*/* .
```

Then create a service (adapt user and password):

```
./jqm.ps1 createnode
./jqm.ps1 -ServiceUser marsu -ServicePassword marsu
./jqm.ps1 start
```

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

Linux / Unix

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterwards
- A user account with read/write rights on JQM_ROOT

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

```
JQM_ROOT = "/opt/jqm"
JQM_VERSION = "1.1.6"
mkdir -p $JQM_ROOT; cd $JQM_ROOT
wget https://github.com/enioka/jqm/archive/jqm-$JQM_VERSION.taz.gz
tar xvf jqm-$JQM_VERSION.taz.gz
rm jqm-$JQM_VERSION.taz.gz
mv jqm-*/* .
rmdir jqm-*
```

Then use the provided jqm.sh script:

```
jqm.sh createnode
jqm.sh start
```

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

Testing

The following will import the definition of three 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 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!

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 and delete or comment JQM_ROOT/conf/db.properties. It contains by default sample configuration for Oracle, PostgreSQL, HSQLDB and MySQL which are the three supported databases. (HSQLDB is not supported in production environments)

Note: the database is intended to be shared between 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 DemoEcho
```

Linux / Unix:

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

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 is supported (tested with PostgreSQL 9.3). It is the recommanded 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
postgres=# grant all privileges on database jqm to jqm;
GRANT
```

MySQL

MySQL 5.x is supported with InnoDB (the default). No specific configuration is required in JQM: no options inside jqm.properties (or absent file).

With InnoDB, a startup script must be used to reset an auto-increment inside the database (InnoDB behaviour messes up with JQM handling of keys, as it resets increment seeds with MAX(ID) on each startup even on empty tables). The idea is to initialize the auto increment for the JobInstance table at the same level as for the History table. An example of script is (adapt the db name & path):

```
select concat('ALTER TABLE jqm.JobInstance AUTO_INCREMENT = ',max(ID)+1,';') as alter_stmt into outfi
\. /tmp/alter_JI_auto_increment.sql
\! rm -f /tmp/alter_JI_auto_increment.sql
```

HSQLDB

HSQLDB 2.3.x is supported in test environments only.

As Hibernate support of HSQLDB has a bug, the jqm.properties file must contain the following line:

```
hibernate.dialect=com.enioka.jqm.tools.HSQLDialect7479
```

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.

Global configuration

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

JNDI configuration

See *Using resources*.

2.5.2 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> | -enqueue <applicationname> | -exportallqueues <xmlp
```

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

-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

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

2.5.3 JMX monitoring

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

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.

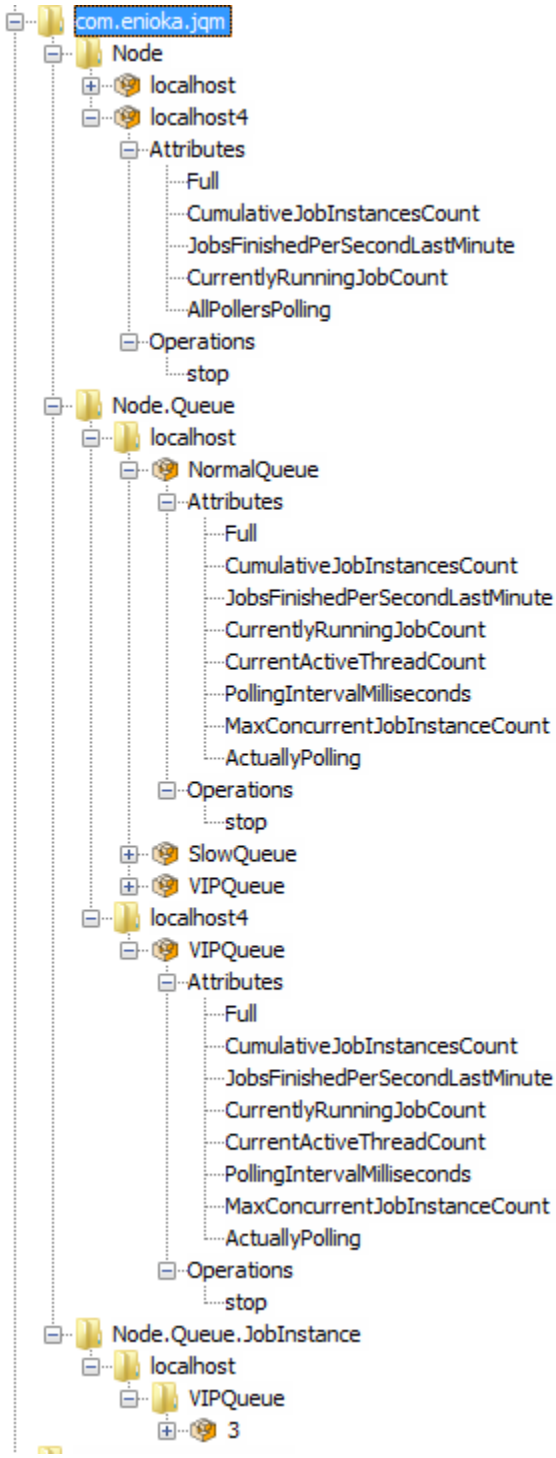
The JMX tree is as follow:

- 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.

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.

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)

`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)

2.5.4 Logs

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

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.

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.

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.

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)

2.5.5 Operations

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.

Stopping

A stop operation will wait for all running jobs to complete, with a two minutes (parameter) timeout. No new jobs are taken 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.

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).

2.5.6 Parameters

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 not currently used, except if you are using the (not production grade) database HSQLDB, in which case the line it contains must be uncommented. It can be safely deleted otherwise.

Changes to bootstrap files require an engine restart.

Node parameters

These parameters are set inside the JQM database table named NODE. There is no GUI or CLI to modify these, therefore they have to be altered 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	Yes
PORT	Port for the basic servlet API	Random free	No	Yes
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
ROOTLOGLEVEL	The log level for this engine (TRACE, DEBUG, INFO, WARN, ERROR, FATAL)	INFO	No	Yes
EXPORTREPO	Not used			
JMXREGISTRYPORT	TCP port on which the JMX registry will listen. Remote JMX disabled if NULL or <1.	NULL	Yes	Yes
JMXSERVERPORT	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 GUI or CLI to modify these, therefore they have to be altered directly inside the database with your tool of choice.

Name	Description	Default	Restart	Null
mavenRepo	A Maven repository to use for dependency resolution	Maven Central	No	At least one
mailSmtpServer	SMTP server to send end-of-job notifications	none	No	Yes
mailFrom	the "from" field of notification mails	jqm@noreply.com	No	?
mailSmtpUser	if SMTP with authentication	NULL	No	?
mailSmtpPassword	if SMTP with authentication	NULL	No	?
mailUseTls	if SMTP with authentication. true of false	NULL	No	No
defaultConnection	don't use this...	jdbc/jqm	No	No
deadline	???	?	?	?
logFilePerLaunch	if true, one log file will be created per launch. Otherwise, everything ends in the main log.	true	Yes	No
internalPollingPeriodMs	Period in ms for checking stop orders	?	?	?
aliveSignalMs	Must be a multiple of internalPollingPeriodMs. Perdiod at which the "I'm a alive" signal is sent	?	?	?

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

Parameter name is case-sensitive.

Note: the mavenRepo is the only parameter that can be specified multiple times. There must be at least one 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 explicitely specified (which is the default).

2.5.7 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.

Defining queues

Queues are defined inside the JQM database table QUEUE. It can be directly modified, 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.

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 inside the JQM database table DEPLOYMENTPARAMETER. 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.**

2.5.8 Administrating resources

See *Using resources*.

2.5.9 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 and always will be.

Out of the box, JQM is not secure.

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

The install files

Inside the install directory, the conf directory should only be read-available to admin/root OS accounts and the account which runs the engine. Indeed, it contains (clear text) the JQM database login and password.

The whole JMQ_ROOT directory and subdirectories should not be writable by anyone but the admin account and the service account (modifying these files would allow to replace engine or payload binaries and therefore run arbitrary code).

The log files

The JQM log does not contain any sensitive data. However, payloads may choose to display whatever thay want and their logs may need to be secured.

The servlet API

This is the simple HTTP GET/POST API that allows to enqueue an execution request, to get the status of a request, to retrieve a file created by a run request.

This API cannot be disabled and is accessible without authentication on a clear HTTP channel. However, it listens on the interface specified in the Node parameters only, so setting it to localhost will prevent remote interaction.

(it was made mostly for local schedulers, so this should not impact functionality much to do so).

Remediation: using localhost or using firewall rules.

Adding an optional certificate auth on HTTPS is an open feature request.

The REST API

This is full implementation of the client API. It is disabled by default. When enabled, it is accessible without authentication on a clear HTTP channel.

Remediation: not deploying the option or using a firewall.

Adding an optional certificate auth on HTTPS is an open feature request.

JMX

Local JMX is always active (Java feature) and admins can connect to it.

Remote JMX is disabled by default. Once enabled, it is accessible without authentication nor encryption.

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 exemple) or using firewall rules.

Database

The security of the data inside the database is subject to the database security systems, nothing there is JQM-specific.

Please note that it is critical to JQM security that the database be secure.

2.6 In case of trouble

2.6.1 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)
- When starting JQM, "Unable to build EntityManager factory" error appears. It means JQM cannot connect o its database. Check the information in the conf/resource.xml file.
- 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.
- "NoSuchMethodException": check that the versions of the APIs and engine is the same.

If your problem does not appear above and the rest of the documentation has no answer for your issue, please *open a ticket*.

2.6.2 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 concerning a payload API issue, the code in question

2.6.3 Bug report

Please direct all bug reports or feature requests at GitHub.

2.7 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.

2.7.1 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
- Maven (CLI, no eclipse plugin) for dependencies, build, tests, packaging
- Sonar (through Maven. No public server provided, rules are `here`)
- Git

Finally, please respect our coding style - it is C++ style, it's on purpose and we like it like that! An Eclipse formatter configuration file is provided `here`. The Sonar rules we use are also included inside that directory.

2.7.2 Release process

This is the procedure that should be followed for making an official JQM release.

Update release notes

Add a chapter to the release notes & commit the file.

Checkout

Check out the branch master with git.

Full build & tests

There is no distinction between tests & integration tests in JQM so this will run all tests.

```
mvn clean install
```

Sonar snapshot

This will run all tests once again.

```
mvn sonar:sonar
```

Once done, take a snaphot in Sonar.

Release test

The release plugin is (inside the pom.xml) parametered 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.

Release

This will upload the packages to the Nexus defined in the "internal" profile. Note that "soon" this should be replaced by a Central-synced repository.

```
mvn release:perform -Pinternal -Darguments='-DskipTests'
```

GitHub upload

Create a release inside GitHub and upload the following artifacts:

- jqm-engine
- jqm-webui
- jqm-ws

And, as long as JMQ is not on Central:

- jqm-api-client-core
- jqm-api-client-hibernate
- jqm-api-client-jersey

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
```

2.7.3 Release notes

1.1.6

Release goal

This release was aimed at making JQM easier to intagrate 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

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

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

1.1.3

Release goal

Fix release for the client API.

Major

- No more System.exit() inside the client API.

Minor

Nothing

2.7.4 Classloading

JQM obeys a very simple classloading architecture, respecting the design goal of simplicity and robustness (to the expense of PermGen space).

The engine classloader stack is as follows (bottom of the stack is at the bottom of the table):

<table>
<tr><td>JNDI class loader (JQM provided - type URLClassloader) Loads everything inside JQM_ROOT/ext</td><td>Payload class loader (JQM provided - type JarClassLoader). Loads the libs of payloads from .m2 or from the payload's "lib" directory</td></tr>
<tr><td colspan="2">System class loader (JVM provided - type AppClassLoader)</td></tr>
<tr><td colspan="2">Extension class loader (JVM provided - no need in JQM)</td></tr>
<tr><td colspan="2">Bootstrap class loader (JVM provided)</td></tr>
</table>

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** (it is created with a null parent - which means its parent is the bootstrap classloader). This classloader is garbage collected at the end of the run.
- JNDI resources in singleton mode (see *Using resources*) must be loaded by th engine from the jars inside JQM_ROOT/ext. This is impossible to do from the AppClassLoader (its class path is fixed once and for all - one cannot add elements to it), so an URLClassLoader is used.

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 explicitely for that use case.
- Easily allows for hot swap of libs and paylaods.
- Avoids having to administer classloader hierarchies and keeps paylaods independant from one another.

Cons:

- It is costly in terms of PermGen: if multiple payloads use the same library, it will be loaded once per payload, which is a waste of memory.
- 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 *one 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.

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 ir in ext, and it just works. The engine is invisible - payloads can consider it as a pure JVM, so no specific development is required. There result is also robust, as payloads have virtually no access to the engine and can't set it off tracks.

2.8 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.