JQM

Release

March 15, 2015

Contents

1 JQM features 3

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

3 Quickstart 9
- 3.1 Windows . . . 9
- 3.2 Linux / Unix . . . 10
- 3.3 Next steps... . . . 10

4 Payload development 11
- 4.1 Payload basics . . . 11
- 4.2 Logging . . . 17
- 4.3 Using resources . . . 17
- 4.4 Engine API . . . 19
- 4.5 Packaging . . . 21
- 4.6 Testing payloads . . . 25

5 Client development 27
- 5.1 Introduction . . . 27
- 5.2 Simple web API . . . 27
- 5.3 Full client API . . . 28
- 5.4 CLI API . . . 41
- 5.5 Engine API . . . 42

6 Administration 43
- 6.1 Installation . . . 43
- 6.2 Command Line Interface (CLI) . . . 47
- 6.3 JMX monitoring . . . 47
- 6.4 Web administration console . . . 51
- 6.5 Logs . . . 52
- 6.6 Operations . . . 53
- 6.7 Parameters . . . 54
- 6.8 Managing queues . . . 56
- 6.9 Administrating resources . . . 58
- 6.10 Managing history . . . 61

6.11 Data security 61
6.12 Database and reliability 65
6.13 Administration web services 66

7 In case of trouble **69**
7.1 Troubleshooting 69
7.2 Reporting bugs and requests 69
7.3 Bug report 69

8 Developement **71**
8.1 Contributing to JQM 71
8.2 Release process 71
8.3 Release notes 73
8.4 Classloading 78
8.5 Testing 79

9 Glossary **81**

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 suitable 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 time-outs and runaway threads. JQM will host these externalized jobs in an asynchronous way, not requiring the application server to wait for completion. Execution can happen on another server/VM, freeing resources (and potentially licence costs, as JQM is free contrary to many application servers).
- job execution request frequency adaptation. Often a job is requested to run multiple times at the same moment (either by a human request, or an automated system reacting to frequent events, ...) while the job should actually run only one at a time (e.g. the job handles all available data at the time of its launch - so there is really no need for multiple instances in parallel). JQM will throttle these requests.

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 optionally offers a rich API that allows running code to ask for another execution, to retrieve structured parameters, to send messages and other advancement notices... Interacting with JQM is also easy: an API, with two different implementations (including REST-style web services, which can be used from a non-Java world) for different needs, is offered to do every imaginable operation easily (new execution request, querying the state of a request, retrieving files created by a job instance, ...).

CHAPTER 1

JQM features

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

Batch code

- Possible but not required to use a specific framework (Spring batch, etc.)
- Runs existing Java 1.6 code, without need for programming specifically for JQM
- Many samples for all features (inside JQM's integration tests)
- Specific API to handle file creation and easy retrieval (a file can be created on any server and retrieved from another in a single call)

Batch interactions

- Query API enabling to easily create client applications (with two full samples included in the distribution), such as web pages listing all the jobs for given user, for a given module, etc.
- Feature rich API

Batch packaging

- Full Maven 3 support: as a Maven-created jar contains its pom.xml, JQM is able to retrieve all the dependencies, simplifying packaging libraries.
- More classic packaging also supported

Administration

- Both command line and web-based graphic user interface for administration
- Can run as a Windows service or a Linux /etc/init.d script
- Fully ready to run out of the box without complicated configuration
- supported on most OSes and databases
- log files can be accessed easily through a central web GUI
- easy definition of class of service through queues
- easy integration with schedulers and CLI

CHAPTER 2

How JQM works

2.1 Basic JQM concepts

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

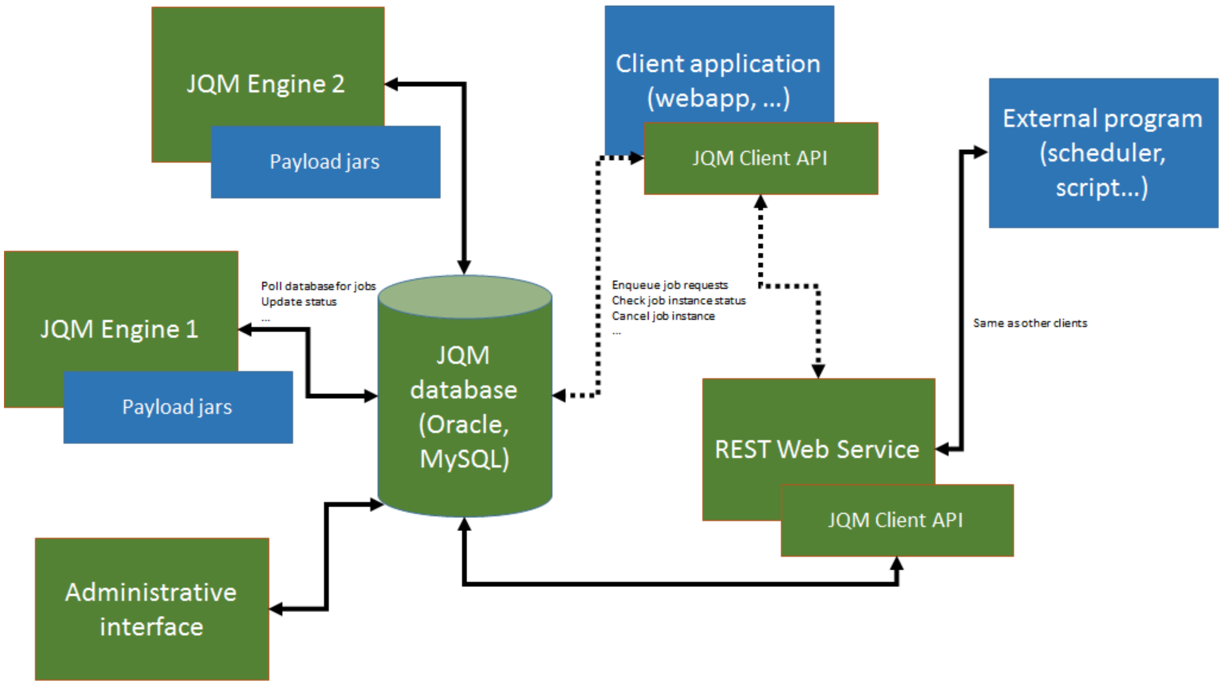
The payload is described inside a *job definition* - so that JQM knows things like the class to load, the path of the jar file, etc. It is usually contained within an XML file. The job definition is actually a deployment descriptor - the batch equivalent for a web.xml or an ejb-jar.xml.

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

Job instances are run by one or many engines called *JQM nodes*. These are simply Java processes that poll the different queues in which job instances are waiting. Runs take place within threads.

Full definitions are given inside the *Glossary*.

2.2 General architecture



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

JQM works like this:

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

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

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

2.3 Nodes, queues and polling

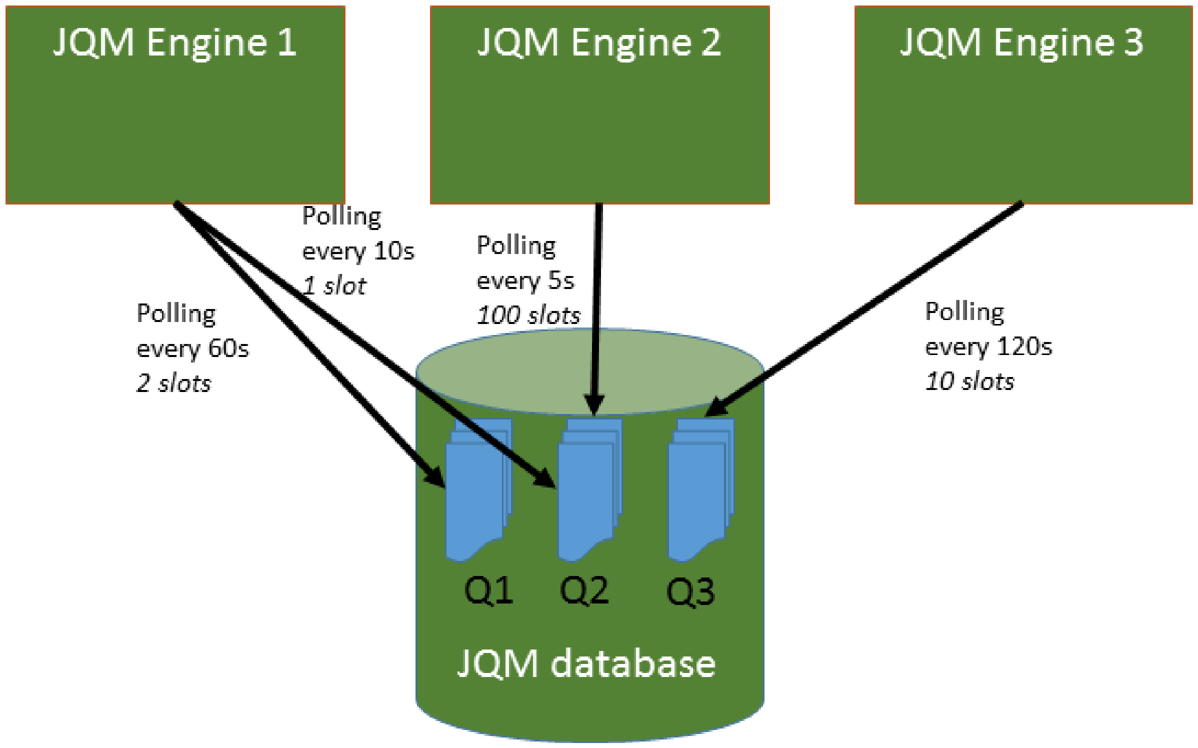
As it names entails, JQM is actually a queue manager. As many queues as needed can be created. A queue contains job instances waiting to be executed.

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 is 1s to avoid killing the database). 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* has a default queue: all job requests pertaining to a job definition are created (unless otherwise specified) inside this queue. It is possible at job request submission, or later once the job instance waits inside its queue, to move a job instance from one queue to another *as long as it has not already began to run*.

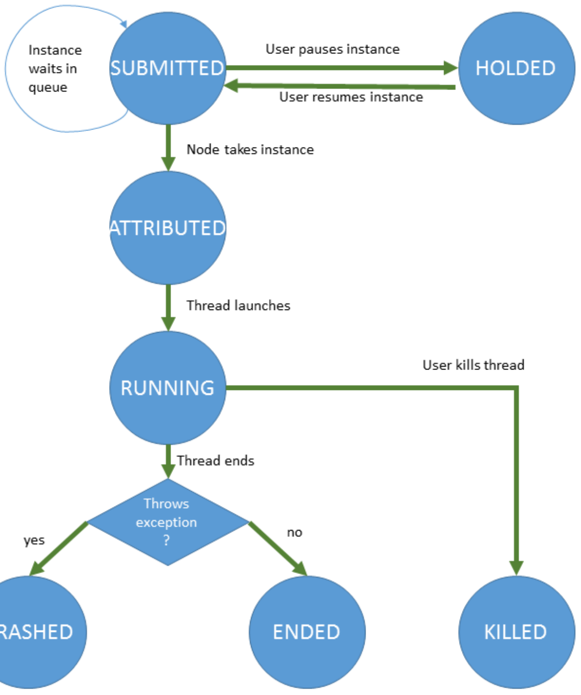
By default, when creating the first engine, one queue is created and is tagged as the default queue (meaning all jobdef that do not have a specific queue will end on that one).

An example:


JQM Engine 1
JQM Engine 2
JQM Engine 3
Polling every 10s 1 slot
Polling every 5s 100 slots
Polling every 60s 2 slots
Polling every 120s 10 slots
Q1
Q2
Q3
JQM database

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

2.4 Job Instance life-cycle



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

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

CHAPTER 3

Quickstart

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

3.1 Windows

Prerequisites:

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

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

```
$JQM_ROOT = "C:\TEMP\jqm" ## Change this
$JQM_VERSION = "1.2.2"  ## Change this
mkdir -Force $JQM_ROOT; cd $JQM_ROOT
Invoke-RestMethod https://github.com/enioka/jqm/releases/jqm-all-$JQM_VERSION/jqm-$JQM_VERSION.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 enable the web console with account root/test (do not use this in production!):

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

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

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

Finally this will start an engine inside the console.:

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

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

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

3.2 Linux / Unix

Prerequisites:

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

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

```
wget  https://github.com/enioka/jqm/releases/jqm-all-1.2.2/jqm-1.2.2.tar.gz # For 1.2.2 release. Adap
tar xvf jqm-1.2.2.tar.gz
```

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

```
cd jqm-1.2.2
./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.

3.3 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.

CHAPTER 4

Payload development

4.1 Payload basics

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

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

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

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

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

4.1.1 Payloads types

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

Main

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

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

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

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

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

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

Runnable

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

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

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

Explicit JQM job

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

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

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

```java
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):

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

4.1.2 Accessing the JQM engine API

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

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

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

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

The dependency is:

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

For more details, please read *Engine API*.

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

4.1.3 Creating files

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

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

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

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

Example:

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

public class App implements Runnable
{
    private JobManager jm;

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

4.1.4 Going to the culling

Payloads are run inside a thread by the JQM engine. Alas, Java threads have one caveat: they cannot be cleanly killed. Therefore, there is no obvious way to allow a user to kill a job instance that has gone haywire. To provide some measure of relief, the *Engine API* provides a method called `JobManager.yield()` that, when called, will do nothing but give briefly control of the job's thread to the engine. This allows the engine to check if the job should be killed (it throws an exception as well as sets the thread's interruption status to do so). Now, if the job instance really has entered an infinite loop where yield is not called nor is the interruption status read, it won't help much. It is more to allow killing instances that run well (user has changed his mind, etc.).

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

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

4.1.5 Full example

This fully commented payload uses nearly all the API.

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

public class App
```

```java
{
    // 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(), jm.module(),
            System.out.println("New request is number " + i);

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

4.1.6 Limitations

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

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

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

4.1.7 Staying reasonable

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

4.2 Logging

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

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

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

4.3 Using resources

4.3.1 Introduction

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

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

JQM JNDI directory can be used for:

- JDBC connections
- JMS resources
- Files
- URLs
- Simple Strings
- Mail session (outgoing SMTP only)
- every ObjectFactory provided by the payloads

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

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

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

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

To define resources, see *Administrating resources*.

Below are some samples & details for various cases.

4.3.2 JDBC

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

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

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

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

4.3.3 JMS

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

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

4.3.4 Files

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

4.3.5 URL

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

4.4 Engine API

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

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

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

class `JobManager`

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

Use is very straightforward:

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

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

4.4.1 Current job metadata

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

`JobManager.`**`parentId`**`()` → int

`JobManager.`**`jobApplicationId`**`()` → int

`JobManager.`**`jobInstanceID`**`()` → int

`JobManager.`**`canBeRestarted`**`()` → boolean

`JobManager.`**`applicationName`**`()` → String

`JobManager.`**`sessionID`**`()` → String

`JobManager.`**`application`**`()` → String

`JobManager.`**`module`**`()` → String

`JobManager.`**`keyword1`**`()` → String

`JobManager.`**`keyword2`**`()` → String

`JobManager.`**`keyword3`**`()` → String

`JobManager.`**`userName`**`()` → String

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

4.4.2 Enqueue & retrieve jobs

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

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

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

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

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

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

`JobManager.`**`waitChildren`**`()` → void

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

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

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

4.4.3 Communications

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

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

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

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

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

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

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

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

4.4.4 Misc.

`JobManager.`**`defaultConnect`**`()` → String

The default connection JNDI alias. To retrieve a default connection, simply use:

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

See *JDBC* for more details.

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

`JobManager.`**`getDefaultConnection`**`()` → Connection

A connection as described by the default JNDI alias. See *JDBC* for more details.

`JobManager.`**`getWorkDir`**`()` → File

If temp files are necessary, use this directory. The directory already exists. It is used by a single instance. It is purged at the end of the run.

`JobManager.`**`yield`**`()` → void

This simply notifies the engine that it can briefly take over the thread, mostly to check if the thread should commit suicide. See *Going to the culling* for more details.

4.5 Packaging

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

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

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

4.5.1 Libraries handling

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

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

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

Maven POM

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

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

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

Conclusion: in that case, no packaging to do.

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

lib directory

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

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

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

Conclusion: in that case, libraries must be packaged.

4.5.2 Shared libraries

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

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

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

4.5.3 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>

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

Importing

The XML can be imported through the command line.

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

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

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

4.6 Testing payloads

4.6.1 Unit testing

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

JQM provides a library named jqm-tst which allows tests that will run a **single job instance** in a stripped-down version of an embedded JQM engine requiring no configuration. The engine is destroyed immediately after the run.

An example taken from JQM owns unit tests:

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

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

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

4.6.2 Integration tests

If you have to test interactions between jobs (for example, one job instance queueing another), it may be necessary to use a full JQM engine. This gives the basics on how to do it (there are no embedded way to do it yet).

Prepare package

Following the previous chapters, you should have:

- a JAR file containing the payload (potentially with libs)
- a descriptor XML file containing all the metadata

Morevoer, if you do not have a working engine at your disposal, 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 are libraries to copy (pom.xml is not used), they must be placed inside a directory named "lib": JQM_DIR/jobs/xxxxx/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.

Client development

5.1 Introduction

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

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

5.2 Simple web API

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

It allows:

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

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

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

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

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

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

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

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

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

# Same, but with parameters
PS> Invoke-RestMethod http://localhost:61260/ws/simple/ji -Method Post -Body @{applicationname="DemoA
1047
```

5.3 Full client API

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

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

5.3.1 Basics

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

Then it is simply a matter of calling:

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

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

- queueing new execution requests

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

For example, to list all executions known to JQM:

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

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

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

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

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

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

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

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

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

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

5.3.2 Client API details

The JqmClient interface

class **`JqmClient`**

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

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

New execution requests

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

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

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

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

A simplified version of the method above.

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

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

Job request deleting

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

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

Synchronous method

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

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

Synchronous method

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

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

Asynchronous method

Pausing and restarting jobs

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

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

`JqmClient.`**`resumeJob`**(*int id*) → void

Will re insert a paused execution request into the queue. The place inside the queue may change from what it used to be before the pause.

`JqmClient.`**`restartCrachedJob`**(*int id*) → int

Will create an execution request from a crashed history element and *remove all traces of the failed execution**.

Queries on Job instances

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

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

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

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

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

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

Quick access helpers

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

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

Files & logs retrieval

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

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

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

  The responsibility to close the stream lies on the API user

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

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

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

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

  The responsibility to close the stream lies on the API user

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

Referential queries

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

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

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

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

API objects

JobRequest

class `JobRequest`

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

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

Queue

class `Queue`

All the metadata describing a *queue*. Read only element.

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

JobDef

class `JobDef`

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

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

Example

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

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

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

5.3.3 Query API

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

Basics, running & filtering

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

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

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

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

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

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

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

Querying live data

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

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

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

Note: When looking for instances of a desired state (ENDED, RUNNING, ...), it is highly recommended to query only the queue or only the history. Indeed, states are specific either to the queue or to history: an ended instance is always in the history, a running instance always in the queue, etc. This is far quicker than querying both history and queues while filtering on state.

Pagination

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

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

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

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

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

Sorting

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

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

5.3.4 JPA Client API

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

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

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

Parameters

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

It is possible to overload persistence unit properties either:

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

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

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

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

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

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

Libraries

In Maven terms, only one library is needed:

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

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

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

Logs

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

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

For example, this may be used as an implementation:

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

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

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

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

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

Container integration

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

In a JNDI-enabled container without other JPA use

Hypothesis:

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

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

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

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

With other JPA use

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

Hypothesis:

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

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

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

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

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

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

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

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

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

Making it work with both Tomcat and Glassfish/WebSphere

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

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

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

5.3.5 Web Service Client API

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

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

Client side

There are two ways to use the WS Client API:

- Using the Java client
- Directly using the web service

Using the Java client

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

For Maven users:

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

and then using the API:

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

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

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

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

The specific parameters are:

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

and can be set:

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

  ```
  Properties p = new Properties();
  p.put("com.enioka.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 example, in an EE6 server that provides JAX-RS 1 and just don't want to work with version 2.

Server side

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

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

Service reference

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

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

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

Used HTTP error codes are:

- 400 (bad request) when responsibility for the failure hangs on the user (trying to delete an already running instance, instance does not exist, etc)

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

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

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

```
404     GET on absent object
500
404     DELETE on absent object
400     Update user with absent role
500     Could not create certificate
400     Invalid enqueue parameters
400     Simple API  is only available when the application runs on top of JQM itself and not a we
400     File does not exist
500     Generic internal exception
400     Generic bad request
```

Script sample

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

```
# Note: we use JSON as a demonstration of how to use it over the default XML. Obviously, PowerShell c

# Authentication?
$cred = Get-Credential root

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

$request = @{applicationName="DemoApi"; user=$env:USERNAME} | ConvertTo-Json
$jobInstance = Invoke-RestMethod http://localhost:62948/ws/client/ji -Method Post -Body $request -Cre
$jobInstance.id


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

$query = @{applicationName="DemoApi"} | ConvertTo-Json
$res = Invoke-RestMethod http://localhost:62948/ws/client/ji/query -Method Post -Body $query -Credent
$res.instances | Format-Table -AutoSize
```

5.4 CLI API

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

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

5.5 Engine API

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

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

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

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

CHAPTER 6

Administration

6.1 Installation

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

6.1.1 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 $JQM_ROOT
Invoke-RestMethod https://github.com/enioka/jqm/releases/download/jqm-all-$JQM_VERSION/jqm-$JQM_VERSI
# Unzipping is finally coming in POSH 5... so not yet.
$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, password and desired node name):

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

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

Linux / Unix

Prerequisites:

- A directory where JQM will be installed, named JQM_ROOT afterwards

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

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

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

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

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

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

Testing

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

Windows:

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

Linux / Unix:

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

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

6.1.2 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 -JobDef DemoEcho
```

Linux / Unix:

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

Database support

Oracle

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

PostgreSQL

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

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

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

MySQL

MySQL 5.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 outf
\. /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.

6.1.3 Global configuration

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

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

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

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

6.1.4 JNDI configuration

See *Using resources*.

6.2 Command Line Interface (CLI)

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

```
java -jar jqm-engine.jar -createnode <nodeName> | -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

6.3 JMX monitoring

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

6.3.1 Monitoring JQM through JMX

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

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

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

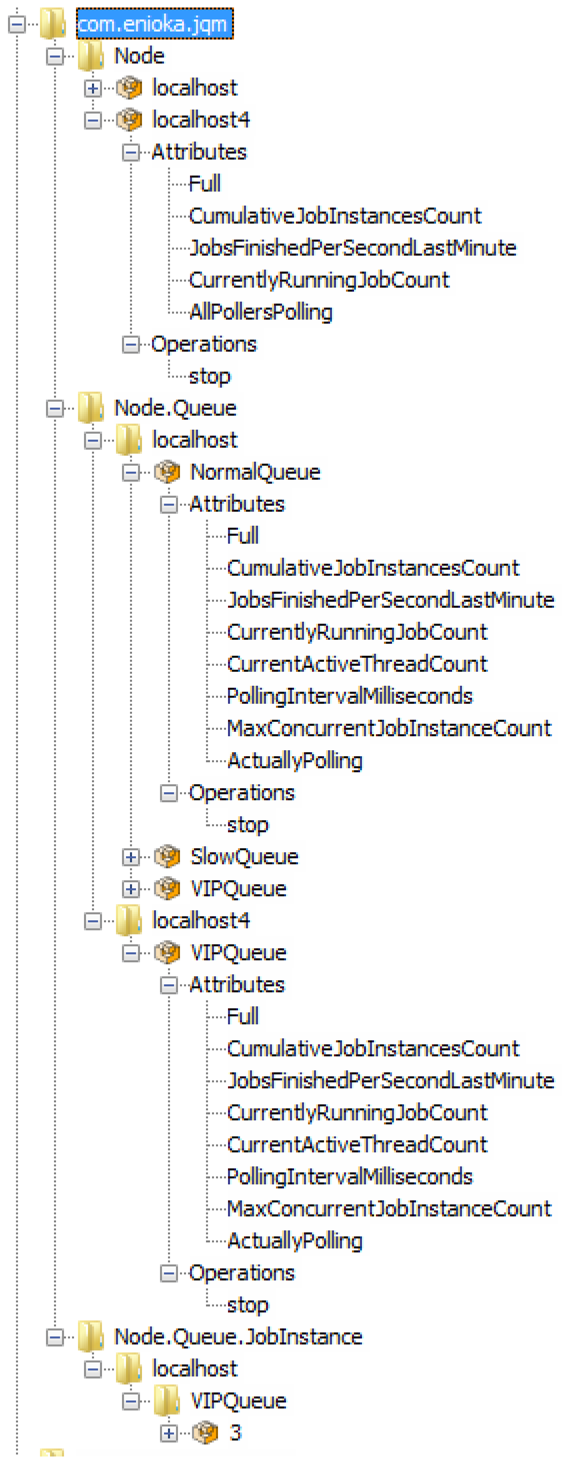
Finally, it is possible to monitor each job individually. This should not be needed very often, the main use being killing a running job.

The JMX tree is as follow:

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

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

In JConsole, this shows as:



```
com.enioka.jqm
    Node
        localhost
        localhost4
            Attributes
                Full
                CumulativeJobInstancesCount
                JobsFinishedPerSecondLastMinute
                CurrentlyRunningJobCount
                AllPollersPolling
            Operations
                stop
    Node.Queue
        localhost
            NormalQueue
                Attributes
                    Full
                    CumulativeJobInstancesCount
                    JobsFinishedPerSecondLastMinute
                    CurrentlyRunningJobCount
                    CurrentActiveThreadCount
                    PollingIntervalMilliseconds
                    MaxConcurrentJobInstanceCount
                    ActuallyPolling
                Operations
                    stop
            SlowQueue
            VIPQueue
        localhost4
            VIPQueue
                Attributes
                    Full
                    CumulativeJobInstancesCount
                    JobsFinishedPerSecondLastMinute
                    CurrentlyRunningJobCount
                    CurrentActiveThreadCount
                    PollingIntervalMilliseconds
                    MaxConcurrentJobInstanceCount
                    ActuallyPolling
                Operations
                    stop
    Node.Queue.JobInstance
        localhost
            VIPQueue
                3
```

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

6.3.2 Remote JMX access

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

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

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

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

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

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

6.3.3 Beans detail

class `JqmEngineMBean`
This bean tracks a JQM engine.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

getEnqueueDate();
: Start time (Calendar)

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

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

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

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

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

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

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

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

6.4 Web administration console

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

It is **disabled by default**.

6.4.1 Enabling the console

The console must be enabled node by node. It should only be enabled on a single node, as one console is able to administer every node referenced inside the central database.

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

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

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

6.4.2 First connection

Using either Internet Explorer (>= 10), Chrome (>= 25), Firefox (>= 28), connect to:

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

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

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

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

6.4.3 If SSL is enabled

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

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

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

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

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

6.5 Logs

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

6.5.1 Log levels

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

The default log-level is INFO.

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

6.5.2 Engine log

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

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

6.5.3 Java Virtual Machine Log

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

6.5.4 Payload logs

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

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

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

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

- the engine log level, which will determine the verbosity of the traces concerning the luanch of the job itself.

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

6.6 Operations

6.6.1 Starting

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

Windows

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

```
Start-Service JQM*
```

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

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

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

Unix/Linux

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

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

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

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

6.6.2 Stopping

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

6.6.3 Restarting

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

Windows:

```
Restart-Service JQM*
```

*X:

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

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

6.6.4 Backup

Elements to backup are:

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

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

6.6.5 Purges

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

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

6.7 Parameters

6.7.1 Engine parameters

These parameters gouvern the behaviour of the JQM engines.

There are three sets of engine parameters:

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

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

Bootstrap

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

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

A second file exists, named JQM_ROOT/conf/jqm.properties. It is 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	Nullable
mavenRepo	A Maven repository to use for dependency resolution	Maven Central	No	At least one
mavenSettingsCL	an alternate Maven settings.xml to use. If absent, the usual file inside ~/.m2 is used.	NULL	No	Yes
defaultConnection	the JNDI alias returned by the engine API getDefaultConnection method.	jdbc/jqm	No	No
logFilePerLaunch	if true, one log file will be created per launch. Otherwise, everything ends in the main log.	true	Yes	No
internalPollingPeriodMs	Period in ms for checking stop orders	10000	Yes	No
aliveSignalMs	Must be a multiple of internalPollingPeriodMs. Perdiod at which the “I’m a alive” signal is sent	60000	Yes	No
disableWsApi	Disable all HTTP interfaces on all nodes. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	false	Yes	Yes
enableWsApiSsl	All HTTP communications will be HTTPS and not HTTP.	false	Yes	No
enableWsApiAuth	Use HTTP basic authentication plus RBAC backend for all WS APIs	true	Yes	No
disableWsApiSimple	Forbids the simple API from loading on any node. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	NULL	Yes	Yes
disableWsApiClient	Forbids the client API from loading on any node. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	NULL	Yes	Yes
disableWsApiAdmin	Forbids the admin API from loading on any node. This takes precedence over node per node settings. Absent means false, i.e. not forbidden.	NULL	Yes	Yes
enableInternalPki	Use the internal (database-backed) PKI for issuing certificates and trusting presented certificates	true	Yes	No

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 explicitly specified (which is the default).

Also, as a side note, mail notifications use the JNDI resource named mail/default, which is created on node startup if it does not exist. See resource documentation to set it up.

6.8 Managing queues

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

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

6.8.1 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:

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

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

The timeToLive parameter is not used any more.

6.8.2 Defining pollers

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

The association between a node and a queue is done 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.**

6.9 Administrating resources

6.9.1 Defining a resource

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

The main JNDI directory table is named `JndiObjectResource` and the object parameters belong to the table `JndiObjectResourceParameter`.

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

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

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

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

6.9.2 Singletons

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

- Many resources are actually to be shared between payloads, such as a connection pool
- Very often, the payload will expect to be returned the same resource when making multiple JNDI lookups, not a different one on each call. Once again, one would expect to be returned the same connection pool on each call, and definitely not to have a new pool created on each call!
- Some resources are dangerous to create inside the payload's context. As stated in *Payload basics*, loading a JDBC driver creates memory leaks (actually, class loader leaks). By delegating this to the engine, the issue disappears.

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

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

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

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

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

6.9.3 Examples

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

JDBC

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

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

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

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

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

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

JMS

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

Parameters for MQ Series QueueConnectionFactory:

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

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

Parameters for MQ Series Queue:

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

Parameter name	Value
QU	queue name

Parameters for ActiveMQ QueueConnexionFactory:

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

Parameter name	Value
brokerURL	broker URL (see ActiveMQ site)

Parameters for ActiveMQ Queue:

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

Parameter name	Value
physicalName	queue name

Files

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

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

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

URL

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

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

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

Mail session

Outgoing SMTP mail session.

Note: the recommended naming pattern is mail/name

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

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

6.10 Managing history

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

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

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

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

- archiving
- partitioning
- indexing
- purges

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

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

Purges should always be considered.

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

6.11 Data security

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

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

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

6.11.1 Data & data flows

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

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

Central Configuration

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

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

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

Stored in: central database

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

Node-specific configuration

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

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

Confidentiality need: high (exposes the central database)

Stored in: local file system

Exposed by: file system (R).

Job referential

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

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

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

Stored in: central database

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

Tracking data

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

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

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

Stored in: central database

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

Logs & batch created files

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

Integrity need: depends

Confidentiality need: depends

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

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

Binaries

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

Integrity need: high

Confidentiality need: low (on GitHub!)

Stored in: file system

Exposed by: file system

6.11.2 Summary of elements to protect

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

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

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

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

Binaries: obvious

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

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

6.11.3 Security mechanisms

Web services

SSL

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

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

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

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

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

Authentication

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

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

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

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

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

Clients use of SSL and authentication

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

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

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

File retrieval specific protection

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

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

Switch off 'protection'

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

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

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

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

Database

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

Binaries

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

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

6.11.4 Monitoring access security

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

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

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

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

6.11.5 Tracing

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

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

6.12 Database and reliability

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

In details:

- pollers stop on first failure. Therefore, no new job instance will run until database connectivity is restored. Failed pollers are restarted on database coming back on line.
- **running job instances continue to run as long as they are not concerned with database connectivity.**
  - They will be impacted if they use some JQM API methods that call the database behind the scenes, such as when they themselves enqueue new job execution requests
  - They will not be impacted otherwise
  - Impacted instances will classically crash with a JDBC exception.
- **ending job instances are stored in memory and wait for the database to come back to be reported. This is referred to as “d**
  - Therefore, these instances will be stored as “running” inside the database even if they are actually done. Not that it matters if the database is fully down, but if only the connectivity is down and the db is up.

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

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

6.13 Administration web services

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

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

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

Verb	Action on container URLs	Action on item URLs
GET	obtain a list of instances	get the instance
POST	add an instance or update it if there is already an instance with the same ID (null ID means create an object)	Not used
PUT	replace the whole collection. Objects that are not in the POST part of the request are dropped, other are created or updated (null ID means create an object)	create or update the instance (null ID means create an object)
DELETE	Not used	removes the object for ever

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

URL	GET	POST	PUT	DELETE	Description
/q	X	X	X		Container for queues. Example of return JSON for a GET: `[{"defaultQueue":`
/q/{id}	X		X	X	A queue. For creating one, you may PUT this (note the absence of ID): `{"defaultQueue":f`
/qmapping	X	X	X		Container for the deployments of queues on nodes. nodeName & queueName cannot be set - they are only GUI helpers. Example of return JSON for a GET: `[{"id":9,"nbThrea` `{"id":154,"nbThr`
/qmapping/{id}	X		X	X	The deployment of a queue on a node. For creating one, you may PUT this (note the absence of ID. nodeName, queueName would be ignored if set): `{"nbThread":5,"no`
/jndi	X	X	X		Container for resource definitions with JNDI aliases. Example of return JSON for a GET: `[{"parameters":[{` `"factory":"com.en`
/jndi/{id}	X		X	X	Resource definitions with JNDI alias
/prm	X	X	X		Cluster parameters container. Example of return JSON for a GET: `[{"id":3,"key":"m` `"key":"deadline",` `{"id":8,"key":"al`
/prm/{id}	X		X	X	Cluster parameter
/node	X	X	X		Cluster nodes

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

CHAPTER 7

In case of trouble

7.1 Troubleshooting

- When starting JQM, “address already in use” error appears. Change the ports of your nodes (by default a node is on a random free port, but you may have started multiple nodes on the same port)
- When starting JQM, “Unable to build EntityManager factory” error appears. It means JQM cannot connect to 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.

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

7.2 Reporting bugs and requests

Please direct all bug reports or feature requests at our tracker on GitHub.

In case you are wondering if your feature request or bug report is well formatted, justified or any other question, feel free to mail the maintainer at mag@enioka.com.

The minimum to join to a bug report:

- the logs in TRACE mode
- if possible, your jobdef XMLs
- if possible, a database dump
- if concerning a payload API issue, the code in question

7.3 Bug report

Please direct all bug reports or feature requests at GitHub.

CHAPTER 8

Developement

This chapter is only useful for JQM developers. For would-be contributors, it is a must-read. Otherwise, it can be skipepd without remorse.

8.1 Contributing to JQM

JQM is an Open Source project under the Apache v2 license. We welcome every contribution through GitHub pull requests.

If you wonder if you should modify something, do not hesitate to mail the maintainer at mag@enioka.com with [JQM] inside the subject. It also works before opening feature requests.

JQM dev environment:

- Eclipse
- 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.

8.2 Release process

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

8.2.1 Environment

The release environment must have:

- PGP & the release private key
- The Selenium setup (see *Testing*)
- Internet access
- Login & password to Sonatype OSSRH.

8.2.2 Update release notes

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

8.2.3 Checkout

Check out the branch master with git.

8.2.4 Full build & tests

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

```
mvn clean install -Pselenium
```

8.2.5 Sonar snapshot

This will run all tests once again.

```
mvn sonar:sonar
```

Once done, take a snaphot in Sonar.

8.2.6 Release test

The release plug-in is (inside the pom.xml) parametrized to use a local git repository, so as to allow mistakes. During that step, all packages are bumped in version number, even if they were not modified.

```
mvn release:prepare -Darguments='-DskipTests'
mvn package
```

Then the test package must be test-deployed in a two-node configuration.

8.2.7 Release

This will upload the packages to the OSSRH staging repository.:

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

OSSRH validation

Go to https://oss.sonatype.org/ and unstage the release. This will in time allow synchronization with Maven Central.

8.2.8 Git push

At this step, the release is done and the local git modifications can be pushed to the central git repository on GitHub.

Warning: when using GitHub for Windows, tags are not pushed during sync. Using the command line is compulsory.

```
git push origin --tags
```

8.2.9 GitHub upload

Create a release inside GitHub and upload the zip and tar.gz produced by the jqm-engine project.

8.3 Release notes

8.3.1 1.2.2

Release goal

This is a maintenance release, containing mostly bugfixes and very few new features that could not be included in the previous version (mostly administration GUI tweaks).

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. 1.2.1 apis will work with 1.2.2 engines. However, as 1.2.2 contains fixes, everyone is strongly encouraged to upgrade.

Database must be rebuilt for version 1.2.2, this means History purge.

Major

- Engine: can now resist a temporary database failure

Minor

- Engine: access log now logs failed authentications
- Engine: various minor bugfix in extreme performance scenarios
- Engine: there is now one log file per node
- Client API: various fixes
- Client API: now support retrieval of running job instance logs
- GUI: various minor improvements
- CLI: jobdef reimport fixes
- Tests: major refactoring with 3x less Maven artifacts

8.3.2 1.2.1

Release goal

The main goal of this release was to simplify the use of JQM. First for people who dislike command line interfaces, by adding a graphical user interface both for administration and for daily use (enqueue, check job status, etc). Second, for payload developers by adding a few improvements concerning testing and reporting.

Upgrade notes

All APIs have been upgraded and **do not contain any breaking change**. Please note that the only version that will work with engine and database in version 1.2.1 is API version 1.2.1: upgrade is compulsory.

Database must be rebuilt for version 1.2.1, this means History purge.

Major

- Client API: Added a fluid version of the JobRequest API
- GUI: Added an administration web console (present in the standard package but disabled by default)
- All APIs: Added an authentication system for all web services, with an RBAC back-end and compatible with HTTP authentication as well as SSL certificate authentication
- Tests: Added a payload unit tester
- General: Added mail session JNDI resource type

Minor

- Client API: Client APIs file retrieval will now set a file name hint inside an attachment header
- Client API: Added an IN option for applicationName in Query API
- Client API: Query API optimization
- Engine: Unix/Linux launch script is now more complete and robust (restart works!)
- Engine: JAVA_OPTS environment variable is now used by the engine launch script
- Engine: Added special "serverName" JNDI String resource
- Engine: All automatic messages (was enqueued, has begun...) were removed as they provided no information that wasn't already available
- Engine: In case of crash, a job instance now creates a message containing "Status changed: CRASHED due to " + first characters of the stacktrace
- Engine: Log levels and content were slightly reviewed (e.g.: stacktrace of a failing payload is now INFO instead of DEBUG)
- Engine API: Added more methods to the engine API (JobManager)
- Tests: Refactored all engine tests
- Documentation: clarified class loading structure
- Documentation: general update. Please read the doc. Thanks!
- General: Jobs can now easily be disabled

8.3.3 1.1.6

Release goal

This release was aimed at making JQM easier to integrate in production environments, with new features like JMX monitoring, better log file handling, JDBC connection pooling, etc.

A very few developer features slipped inside the release.

Upgrade notes

No breaking changes.

Compatibility matrix:

Version 1.1.6 / Other version	Engine	Client API	Engine API
Engine		>= 1.1.4	>= 1.1.4
Client API	== 1.1.6		
Engine API	>= 1.1.5		

How to read the compatibility matrix: each line corresponds to one JQM element in version 1.1.6. The different versions given correspond to the minimal version of other components for version 1.1.6 to work. A void cell means there is no constraint between these components.

For exemple : a payload using engine API 1.1.6 requires at least an engine 1.1.5 to work.

Major

- Documentation: now in human readable form and on https://jqm.readthedocs.org
- Distribution: releases now published on Maven Central, snapshots on Sonatype OSSRH.
- Engine: added JDBC connection pooling
- Engine: added JMX monitoring (local & remote on fixed ports). See http://jqm.readthedocs.org/en/latest/admin/jmx.html for details
- Engine: each job instance now has its own logfile
- Engine: it is now impossible to launch two engines with the same node name (prevent startup cleanup issues creating data loss)
- Engine: failed job requests due to engine kill are now reported as crashed jobs on next engine startup
- Engine: added UrlFactory to create URL JNDI resources
- Engine: dependencies/libs are now reloaded when the payload jar file is modified or lib folder is modified. No JQM restart needed anymore.

Minor

- Engine API: legacy JobBase class can now be inherited through multiple levels
- Engine: incomplete payload classes (missing parent class or lib) are now correctly reported instead of failing silently
- Engine: refactor of main engine classes
- Engine: races condition fixes in stop sequence (issue happening only in JUnit tests)
- Engine: no longer any permanent database connection
- Engine: Oracle db connections now report V$SESSION program, module and user info
- Engine: logs are less verbose, default log level is now INFO, log line formatting is now cleaner and more readable
- General: Hibernate minor version upgrade due to major Hibernate bugfixes
- General: cleaned test build order and artifact names

8.3.4 1.1.5

Release goal

Bugfix release.

Upgrade notes

No breaking changes.

Major

Nothing

Minor

- Engine API: engine API enqueue works again
- Engine API: added get ID method
- Db: index name shortened to please Oracle

8.3.5 1.1.4

Release goal

This release aimed at fulfilling all the accepted enhancement requests that involved breaking changes, so as to clear up the path for future evolutions.

Upgrade notes

Many breaking changes in this release in all components. Upgrade of engine, upgrade of all libraries are required plus rebuild of database. *There is no compatibiliy whatsoever between version 1.1.4 of the libraries and previous versions of the engine and database.*

Please read the rest of the release notes and check the updated documentation at https://github.com/enioka/jqm/blob/master/doc/index.md

Major

- Documentation: now fully on Github
- **Client API: - breaking - is no longer static. This allows:**
  - to pass it parameters at runtime
  - to use it on Tomcat as well as full EE6 containers without configuration changes
  - to program against an interface instead of a fully implemented class and therefore to have multiple implementations and less breaking changes in the times to come
- Client API: - **breaking** - job instance status is now an enum instead of a String
- Client API: added a generic query method

- Client API: added a web service implementation in addition to the Hibernate implementation
- Client API: no longer uses log4j. Choice of logger is given to the user through the slf4j API (and still works without any logger).
- Client API: in scenarios where the client API is the sole Hibernate user, configuration was greatly simplified without any need for a custom persistence.xml
- Engine: can now run as a service in Windows.
- Engine: - **breaking** - the engine command line, which was purely a debug feature up to now, is officialized and was made usable and documented.
- Engine API: now offers a File resource through the JNDI API
- Engine API: payloads no longer need to use the client or engine API. A simple static main is enough, or implementing Runnable. Access to the API is done through injection with a provided interface.
- Engine API: added a method to provide a temporary work directory

Minor

- Engine: various code refactoring, including cleanup according to Sonar rules.
- Engine: performance enhancements (History is now insert only, classpaths are truly cached, no more unzipping at every launch)
- Engine: can now display engine version (CLI option or at startup time)
- Engine: web service now uses a random free port at node creation (or during tests)
- Engine: node name and web service listeing DNS name are now separate notions
- Engine: fixed race condition in a rare high frequency scenario
- Engine: engine will now properly crash when Jetty fails to start
- Engine: clarified CLI error messages when objects do not exist or when database connection cannot be established
- Engine: - **breaking** - when resolving the dependencies of a jar, a lib directory (if present) now has priority over pom.xml
- Engine tests: test fixes on non-Windows platforms
- Engine tests: test optimization with tests no longer waiting an arbitrary amount of time
- Client API: full javadoc added
- Engine API: calling System.exit() inside payloads will now throw a security ecveption (not marked as breaking as it was already forbidden)
- General: - **breaking** - tags fields (other1, other2, ...) were renamed “keyword” to make their purpose clearer
- General: packaging now done with Maven

8.3.6 1.1.3

Release goal

Fix release for the client API.

Major

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

Minor

Nothing

8.4 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):

In the engine:

System class loader (JVM provided - type AppClassLoader). Used by the engine itself (except JNDI calls).
Extension class loader (JVM provided - no need in JQM)
Bootstrap class loader (JVM provided)

For payloads:

Payload class loader (JQM provided - type JarClassLoader). Loads the libs of payloads from .m2 or from the payload's "lib" directory
JNDI class loader (JQM provided - type URLClassloader) Loads everything inside JQM_ROOT/ext
Extension class loader (JVM provided - no need in JQM)
Bootstrap class loader (JVM provided)

The general idea is:

- The engine uses the classic JVM-provided AppClassLoader for everything concerning its internal business
- Every payload launch has its own classloader, **totally independent from the engine classloader**. This classloader is garbage collected at the end of the run.
- JNDI resources in singleton mode (see *Using resources*) must be kept cached by the engine, so they cannot be loaded through the transcient payload classloader itself. Also, the loaded resources must be understandable by the payloads - and therefore there must be a shared classloader here. The JNDI classloader is therefore the parent of the payload classloaders. The JNDI classloader itself has no parent (save obviously the bootstrap CL), so the payloads won't be polluted by anything foreign.

Advantages:

- The engine is totally transparent to payloads, as the engine libraries are inside a classloader which is not accessible to payloads.
- It allows to have multiple incompatible versions of the same library running simultaneously in different payloads.
- It still allows for the exposition to the payload of an API implemented inside the engine through the use of a proxy class, a pattern designed explicitely for that use case.
- Easily allows for hot swap of libs and payloads.
- Avoids having to administer complex classloader hierarchies and keeps payloads 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. If there are costly shared resources, they can however be put inside ext - but the same version will be used by all payloads on the engine.
- In case the payload does something stupid which prevents the garbage collection of at least one of its objects, the classloader will not be able to be garbage collected. This a huge memory leak (usually called a classloader leak). The best known example: registering a JDBC driver inside the static bootstrap-loaded DriverManager. This keeps a reference to the payload-context driver inside the bootstrap-context, and prevents collection. This special case is the reason why singleton mode should always be used for JDBC resources.
- There is a bug inside the Sun JVM 6: even if garbage collected, a classloader will leave behind an open file descriptor. This will effectively prevent hot swap of libs on Windows.

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.

The result is also robust, as payloads have virtually no access to the engine and can't set it off tracks.

8.5 Testing

JQM is tested through an series of automated JUnit tests. These tests are usage-oriented (*integration tests*) rather than unit oriented. This means: every single functionality of JQM must have (at least) one automated test that involves running a job inside a full engine.

8.5.1 Automated builds

Travis

The project has a public CI server on http://travis-ci.org/enioka/jqm.

Selenium

The project has a public Selenium server at https://saucelabs.com/u/marcanpilami

8.5.2 Tests

Standard tests

These are the tests that **should always be run before any commit**. Any failure fails the build.

They are run though Maven (mvn clean install test) and should be able to run without any specific configuration. They are always run by Travis.

Selenium tests

The UI also has a few dedicated tests that run inside Selenium. To avoid configuration and ease test reproducibility, we use Sauce Labs' cloud Selenium. The Travis build uses the maintainer's account.

As this account is personal, its credentials are not included inside the build descriptor and these tests are disabled by default (they are inside a specific Maven profile). In order to use them, a free account on Sauce Labs is required, as well as putting this inside Maven's settings.xml:

```
<profile>
    <id>selenium</id>
    <activation>
        <activeByDefault>false</activeByDefault>
    </activation>
    <properties>
        <SAUCE_USERNAME>YOUR_USER_NAME</SAUCE_USERNAME>
        <SAUCE_ACCESS_KEY>YOUR_ACCESS_KEY</SAUCE_ACCESS_KEY>
        <SAUCE_URL>localhost:4445/wd/hub</SAUCE_URL>
    </properties>
</profile>
```

Moreover, as the web application actually runs on the developer's computer and not on the Selenium server, a tunnel must be activated, using Sauce Connect. The URL above reflects this.

Note: the Sauce Connect Maven plugin was not included in the pom, because it implies starting and stopping the tunnel on each test run - and this is a very long process. It's easier on the nerves to simply start the tunnel and forget it.

Finally, running the tests is simply done by going inside the jqm-wstst project and running the classic "mvn test -Pselenium" command. Obviously, if in the settings.xml file the profile was marked as active by default, the -P option can be omitted.

Web-services dev and tests

The admin GUI as well as all the web services are inside the jqm-ws project.

To develop and test this project in Eclipse, one needs a fully working JQM database. The easiest way to get it is to install a local node following the documentation. Then enable the admin GUI & create the root account with the command line. Do not enable SSL.

The node can be stopped - it won't be needed anymore.

Then, inside Eclipse, install a Tomcat 7. (not 8 - this would require Java 7).

The project contains a context.xml file inside src/test/webapp/META-INF that must be updated with the connection string to your database. Please do not commit these modifications.

Warning: you must ensure the src/test/webapp/META-INF directory is inside the "deployment assembly" inside Eclipse's project properties.

Then the database driver to the the lib directory of Tomcat

Everything is ready - the project can now be "run on server". The URL will be http://localhost:8080/jqm-ws

CHAPTER 9

Glossary

Payload the actual Java code that runs inside the JQM engine, containing business logics. This is must be provided by the application using JQM.

Job Definition, JobDef the metadata describing the payload. Also called JobDef. Entirely described inside the JobDef XML file. Identified by a name called "Application Name"

Job Request the action of asking politely the execution of a *JobDef* (which in turn means running the payload)

Job Instance the result of of a Job Request. It obeys the Job Instance lifecycle (enqueued, running, endded, ...). It is archived at the end of its run (be it successful or not) into the history.

JQM Node, JQM Engine an instance of the JQM service (as in 'Windows service' or 'Unix init.d service') that can run payloads

Job queue, Queue a virtual FIFO queue where *job requests* are lined up. These queues are polled by some *nodes*.

Enqueue the action of putting a new *Job Request* inside a *Queue*. The queue is usually determined by the *JobDef* which holds a default queue.

Symbols

-createnode <nodeName>
    command line option, 47
-enqueue <applicationname>
    command line option, 47
-exportallqueues <xmlpath>
    command line option, 47
-h, –help
    command line option, 47
-importjobdef <xmlpath>
    command line option, 47
-importqueuefile <xmlpath>
    command line option, 47
-startnode <nodeName>
    command line option, 47
-v, –version
    command line option, 47

A

addDeliverable() (JobManager method), 21
application() (JobManager method), 20
applicationName() (JobManager method), 20

C

canBeRestarted() (JobManager method), 20
cancelJob() (JqmClient method), 30
command line option
    -createnode <nodeName>, 47
    -enqueue <applicationname>, 47
    -exportallqueues <xmlpath>, 47
    -h, –help, 47
    -importjobdef <xmlpath>, 47
    -importqueuefile <xmlpath>, 47
    -startnode <nodeName>, 47
    -v, –version, 47

D

defaultConnect() (JobManager method), 21
deleteJob() (JqmClient method), 30

E

Enqueue, **81**
enqueue() (JobManager method), 20
enqueue() (JqmClient method), 30
enqueueFromHistory() (JqmClient method), 30
enqueueSync() (JobManager method), 20

G

getActiveJobs() (JqmClient method), 31
getCumulativeJobInstancesCount() (JqmEngineMBean method), 49
getCumulativeJobInstancesCount() (PollingMBean method), 50
getCurrentActiveThreadCount() (PollingMBean method), 49
getCurrentlyRunningJobCount() (JqmEngineMBean method), 49
getCurrentlyRunningJobCount() (PollingMBean method), 50
getDefaultConnection() (JobManager method), 21
getDeliverableContent() (JqmClient method), 31
getJob() (JqmClient method), 31
getJobDefinition() (JqmClient method), 32
getJobDefinitions() (JqmClient method), 32
getJobDeliverables() (JqmClient method), 31
getJobDeliverablesContent() (JqmClient method), 31
getJobLogStdErr() (JqmClient method), 31
getJobLogStdOut() (JqmClient method), 31
getJobMessages() (JqmClient method), 31
getJobProgress() (JqmClient method), 31
getJobs() (JqmClient method), 31
getJobsFinishedPerSecondLastMinute() (JqmEngineMBean method), 49
getJobsFinishedPerSecondLastMinute() (PollingMBean method), 50
getMaxConcurrentJobInstanceCount() (PollingMBean method), 50
getPollingIntervalMilliseconds() (PollingMBean method), 50
getQueues() (JqmClient method), 32

getUptime() (JqmEngineMBean method), 49
getUserActiveJobs() (JqmClient method), 31
getVersion() (JqmEngineMBean method), 49
getWorkDir() (JobManager method), 21

H

hasEnded() (JobManager method), 20
hasFailed() (JobManager method), 20
hasSucceeded() (JobManager method), 20

I

isActuallyPolling() (PollingMBean method), 50
isAllPollersPolling() (JqmEngineMBean method), 49
isFull() (JqmEngineMBean method), 49
isFull() (PollingMBean method), 50

J

Job Definition, **81**
Job Instance, **81**
Job queue, **81**
Job Request, **81**
jobApplicationId() (JobManager method), 20
JobDef, **81**
JobDef (built-in class), 32
jobInstanceID() (JobManager method), 20
JobManager (built-in class), 19
JobRequest (built-in class), 32
JQM Engine, **81**
JQM Node, **81**
JqmClient (built-in class), 29
JqmEngineMBean (built-in class), 49

K

keyword1() (JobManager method), 20
keyword2() (JobManager method), 20
keyword3() (JobManager method), 20
kill() (LoaderMBean method), 50
killJob() (JqmClient method), 30

L

LoaderMBean (built-in class), 50

M

module() (JobManager method), 20

P

parameters() (JobManager method), 20
parentId() (JobManager method), 20
pauseQueuedJob() (JqmClient method), 30
Payload, **81**
PollingMBean (built-in class), 49

Q

Queue, **81**
Queue (built-in class), 32

R

restartCrachedJob() (JqmClient method), 30
resumeJob() (JqmClient method), 30

S

sendMsg() (JobManager method), 21
sendProgress() (JobManager method), 21
sessionID() (JobManager method), 20
stop() (JqmEngineMBean method), 49
stop() (PollingMBean method), 50

U

userName() (JobManager method), 20

W

waitChild() (JobManager method), 20
waitChildren() (JobManager method), 20

Y

yield() (JobManager method), 21