Orchestra User Guide

This document contains an installation and user guide for Orchestra 4.6.2

Orchestra Team

- May 2010 -

Table of Contents

Introduction

This documentation is targeted to Orchestra users. It presents the installation procedure and a quick user guide of Orchestra features.

Chapter 1, *General information* describes the new version Orchestra v4

Chapter 2, *Prerequisites* describes the prerequisites to the installation of Orchestra

Chapter 3, *Installation guide* describes how to install the Orchestra engine

Chapter 4, *Configuration and Services* describes main configuration features and default services

Chapter 5, *User guide* This chapter will guide you through the discovery of the functionalities of Orchestra.

Chapter 6, *Developer's guide* guides you through APIs of Orchestra.

Chapter 1. General information

1.1. Orchestra Overview

The new version of Orchestra is based on the “Process Virtual Machine” conceptual model for processes. The Process Virtual Machine defines a generic process engine enabling support for multiple process languages (such BPEL, XPDL…).

On top of that, it leads to a pluggable and embeddable design of process engines that gives modelling freedom to the business analyst. Additionally, it enables the developer to leverage process technology embedded in a Java application.

For more information about the Process Virtual Machine, check Orchestra FAQs [http://orchestra.ow2.org/xwiki/bin/view/Main/FAQ] on the Orchestra web site [http://orchestra.ow2.org] .

1.2. Features list

Orchestra is a Web Service Orchestration solution that provides BPEL 2.0 support. Business Process Execution Language (BPEL) is an XML language created by the Oasis Consortium. More information and the specifications can be found on Oasis web site [www.oasis-open.org/committees/wsbpel/]

Orchestra provides full support of the BPEL 2.0 standard.

This version provides Web Service support using the Axis 1.4 framework or CXF 2.2.7.

Orchestra is shipped with a complete test suite and a few examples.

Orchestra is persistable. This means that all the data concerning your processes definition and instances execution is stored in a Database using a persistence framework (hibernate by default). The following database systems have been successfully tested :

- H2 Database (default)
- Postgres (8.3)
- MySQL (5.0)
- Oracle (10g)

1.3. Restrictions

Orchestra comes out with an innovative architecture based on a generic and extensible engine, called "The Process Virtual Machine" and a powerful injection technology allowing services pluggability.

This new version of Orchestra is aimed at showing the power of its very innovative architecture by providing support for all the basic activities defined in the BPEL standard. As stated in the previous section, this version provides the possibility to persist the processes definition and execution. The 4.2 release provides support for the last important BPEL statement named eventHandler. Orchestra now provides full support of BPEL 2.0. The next stage will be to extend Orchestra to provide the first Open Source Business Process Server to power your SOA infrastructure. Stay tuned ! Check the roadmap [http://wiki.orchestra.objectweb.org/xwiki/bin/view/Main/Roadmap] for more information.

This version has some restrictions on the following aspects :

- Some restrictions in assign statement :
  - no extensionAssignOperation
  - validate not supported
- Some restrictions in scope statement
  - isolated not supported
  - exitOnStandardFault not supported
- The following BPEL 2.0 statements are not supported :
  - validate
  - extensionActivity
  - import
  - extensions

1.4. Tooling

For the new version, Orchestra does not ship a graphical designer. Orchestra engine has been tested with processes created using the Netbeans BPEL designer [http://www.netbeans.org/kb/55/bpel_gsg.html]. It is also possible to use the Eclipse BPEL designer [www.eclipse.org/bpel/] . Download and installation instruction are available on the project web site. However we have encountered a few bugs in the eclipse designer. So we advise the use of NetBeans. There is a work in progress to provide a Web 2.0 designer that will be accessible directly from the console. A preview is already available.

This version of Orchestra provides an new Web 2.0 administration console. This console will be improved in following releases to add monitoring capabilities.

Chapter 2. Prerequisites

2.1. Hardware

A 1GHz processor is recommended, with a minimum of 512 Mb of RAM. Windows users can avoid swap file adjustments and get improved performance by using 1Gb or more of RAM

2.2. Software

- Orchestra requires Java Development Kit (JDK) 1.5 (also called JDK 5.0) but also runs with following releases.

  The JDK software can be downloaded from http://java.sun.com/j2se/1.5.0

- Orchestra requires Apache Ant 1.7.1 or higher

  It can be downloaded from http://ant.apache.org

Chapter 3. Installation guide

Orchestra comes in two kinds of distribution:

- Tomcat distribution: Orchestra is embedded in a web application deployed in tomcat container.
- Felix OSGI distribution: Orchestra is embedded in an OSGI bundle deployed in felix OSGI platform.

As explained in Chapter 4, *Configuration and Services*, Orchestra can use different Web Service frameworks. Apache Axis1 and CXF are supported. For each web service framework, a tomcat package and a felix package are provided.

The installation and configuration steps are independent of the web service framework.

3.1. Web Service Frameworks

3.1.1. Apache Axis

Orchestra web service implementation based on Axis 1.4 offers basic web service capabilities.

3.1.2. Apache CXF

Orchestra web service implementation based on CXF offers advanced web service capabilities.

CXF implementation adds support for:

- WS-addressing
- WS-RM
- Apache Camel

3.2. Orchestra Tomcat distribution

3.2.1. Installation

Unzip the orchestra-tomcat distribution package.

```
>unzip orchestra-tomcat-4.6.2.zip
```

A new directory `orchestra-tomcat-4.6.2` will be created. It contains an ant file to install and start Orchestra.

3.2.1.1. Basic installation

Remark : Orchestra runs in Apache Tomcat servlet container. Tomcat 6.0.26 is delivered with the Orchestra Package.

To install Orchestra, go to orchestra directory and launch the install by running ant:

```
>cd orchestra-tomcat-4.6.2
```

```
>ant install
```

The install script installs Tomcat and Orchestra. The default installation activates the persistence using the H2 Database.

> **Important**
> *if your network is based on a proxy, please specify the proxy settings in your JAVA_OPTS environment property. The system properties to specify are described in the java documentation [http://java.sun.com/j2se/1.5.0/docs/guide/net/properties.html].*

3.2.1.2. Advanced installation: Using another tomcat distribution.

Orchestra is shipped with a lightweight Apache Tomcat Servlet container. This section explains how to install Orchestra in an existing tomcat distribution.

The install.properties file in the conf directory contains the information used by Orchestra installation. The default content is:

```
catalina.home=${orchestra.dir}/tomcat
catalina.base=${catalina.home}
```

To use another tomcat installation, just update the *catalina.home* and *catalina.base* properties before calling:

```
>ant install
```

3.2.1.3. Advanced installation: into JOnAS

Orchestra is shipped with a ligthweight Apache Tomcat Servlet container. This section explains how to install Orchestra in a JOnAS Application Server.

3.2.1.3.1. JOnAS 4

- Delete or upgrade xml-apis.jar, xercesImpl.jar and xalan-xxx.jar from $JONAS_ROOT/lib/endorsed. These libraries are needed for J2EE compatibility, but JOnAS4 can run fine without them. JOnAS4 has old versions of these libraries, which creates incompatibility issues with Orchestra.
- Copy Orchestra conf into $JONAS_BASE/conf (copy all files from orchestra conf directory to $JONAS_BASE/conf).
- Copy jdbc jar driver into $JONAS_BASE/lib/ext
- In orchestra.properties, change orchestra.servlet.port value to your JOnAS's tomcat port configuration (cf $JONAS_BASE/conf/server.xml)
- Copy Orchestra war (available in orchestra lib/ directory) to $JONAS_BASE/webapps and start JOnAS

3.2.2. Database Management

The default configuration of Orchestra uses the Database persistence service and the H2 Database. Orchestra has also been tested with Oracle, MySQL and Postgres database system. To change to mysql, postgres or Oracle, you need to put the corresponding JDBC driver in the directory `$CATALINA_BASE/lib` and modify the `hibernate.properties` file (see Section 4.3.4.1, “Database Access Configuration”)

3.2.3. Orchestra directory structure

Hereafter is detailled the structure of Orchestra installation. The installation directory contains the following structure :

```
    README
build.xml
install.xml
   common.xml
Licence.txt
tomcat/
conf/
doc/
examples/
lib/
   resources/
```

Let's present those items :

- README

  This file gives the basic information related to Orchestra

- build.xml

  This file is an ant file that provides tasks to install and use Orchestra. Just typing `ant` will result giving you the usage.

- install.xml and common.xml

  These files are ant files that are used by build.xml

- License.txt

  The license of Orchestra. All of Orchestra is available under the LGPL license.

- conf/

  This directory contains all the configuration files of Orchestra.

- tomcat/

  This directory is the default Tomcat installation shipped with Orchestra.

- doc/

  This directory contains the documentation of Orchestra. It contains :

  - userGuide.pdf

    For PDF documentation

  - html/userGuide.html

    For HTML documentation in a single page

  - html/userGuide/userGuide.html

    For HTML documentation in different pages

- examples/

  This directory contains the examples provided with Orchestra package. See Section 5.4, “Running the examples”

- lib/

  This directory contains the libraries used in Orchestra.

- resources/

  This directory contains Orchestra database creation scripts for supported databases and Orchestra environment configuration examples.

3.3. Orchestra OSGI Felix distribution

3.3.1. Installation

Unzip the orchestra-felix distribution package.

```
>unzip orchestra-felix-4.6.2.zip
```

A new directory `orchestra-felix-4.6.2` will be created. It contains an ant file to install and start Orchestra.

Remark : Orchestra runs in Apache Felix OGSI platform. Felix 2.0.2 is delivered with the Orchestra Package.

There is no specific installation step for running Orchestra :

```
>cd orchestra-felix-4.6.2
```

The default configuration activates the persistence using the H2 Database.

> **Important**
> *if your network is based on a proxy, please specify the proxy settings in your JAVA_OPTS environment property. The system properties to specify are described in the java documentation [http://java.sun.com/j2se/1.5.0/docs/guide/net/properties.html].*

3.3.2. Database Management

The default configuration of Orchestra uses the Database persistence service and the H2 Database. Orchestra has also been tested with Oracle, MySQL and Postgres database system. To change to mysql, postgres or Oracle, you need to install the corresponding JDBC driver bundle in the OSGI platform and modify the `hibernate.properties` file (see Section 4.3.4.1, “Database Access Configuration”)

3.3.3. Orchestra directory structure

Hereafter is detailled the structure of Orchestra installation. The installation directory contains the following structure :

```
        README
    build.xml
    Licence.txt
```

```
   bundle/
      felix/
   conf/
   doc/
   examples/
   lib/
      resources/
```

Let's present those items :

- README

  This file gives the basic information related to Orchestra

- build.xml

  This file is an ant file that provides tasks to use Orchestra. Just typing `ant` will result giving you the usage.

- License.txt

  The license of Orchestra. All of Orchestra is available under the LGPL license.

- conf/

  This directory contains all the configuration files of Orchestra.

- bundle/

  This directory contains Orchestra OSGI bundles and dependencies.

- felix/

  This directory contains the Apache Felix bundle.

- doc/

  This directory contains the documentation of Orchestra. It contains :

  - userGuide.pdf

    For PDF documentation

  - html/userGuide.html

    For HTML documentation in a single page

  - html/userGuide/userGuide.html

    For HTML documentation in different pages

- examples/

  This directory contains the examples provided with Orchestra package. See Section 5.4, “Running the examples”

- lib/

  This directory contains the libraries used for tests.

- resources/

  This directory contains Orchestra database creation scripts for supported databases and Orchestra environment configuration examples.

Chapter 4. Configuration and Services

This chapter introduces the services configuration infrastructure provided by Orchestra as well as main services included in this version.

4.1. Simple configuration

The orchestra.properties file in the conf/ directory contains properties that can be easily changed. These properties are used by both orchestra client and orchestra server. Here is the default orchestra.properties file:

```
orchestra.servlet.host=localhost
orchestra.servlet.port=8080
orchestra.servlet.path=orchestra/services

orchestra.jmx.port=9999
orchestra.jmx.objectName=JMXAgent:name=orchestraRemoteDeployer
orchestra.jmx.serviceUrl=service:jmx:rmi:///jndi/rmi://localhost:9999/orchestraServer
```

- *orchestra.servlet.host* the host where orchestra server is installed.
- *orchestra.servlet.port* the port on which the web services will be exposed.
- *orchestra.servlet.path* the path on the server where the web services will be exposed. Orchestra web services will be available from http://${orchestra.servlet.host}:${orchestra.servlet.port}/$orchestra.servlet.path}/serviceName
- *orchestra.jmx.port* the port of the JMX server.
- *orchestra.jmx.serviceUrl* the JMX service url where the api mbeans will be available.
- *orchestra.jmx.objectName* the name of Orchestra mbean.

4.2. Services Container

The Process Virtual Machine technology includes a services container allowing the injection of services and objets that will be leveraged during the process definition and execution. Objects and services used by the Orchestra engine are defined through a XML file. A dedicated parser and a wiring framework are in charge of creating those objects. Service invoker, publisher, persistence and timers are examples of pluggable services.

This services container (aka IoC container) can be configured through a configuration file. A default configuration file is included in the package under the /conf directory (environment.xml).

This configuration is only used on the server side.

4.2.1. Environment.xml file

The default environment.xml file created during the installation of Orchestra is set to use the database implementation of the persistence service. This file also sets the configuration of hibernate. Here is the environment.xml file generated :

```
<environment-definition>
  <environment-factory>
    <properties name="orchestra-properties" resource="orchestra.properties"/>
    <hibernate-configuration name="hibernate-configuration:core">
      <properties resource="hibernate.properties"/>
      <mappings resource="hibernate/bpel.core.mappings.xml"/>
      <mappings resource="hibernate/bpel.monitoring.mappings.xml"/>
      <cache-configuration resource="hibernate/bpel.cache.definition.xml" usage="nonstrict-read-write"/>
    </hibernate-configuration>
    <hibernate-session-factory configuration="hibernate-configuration:core" init="eager" name="hibernate-session-fact
    <command-service>
      <orchestra-retry-interceptor delay-factor="2" retries="10"/>
      <environment-interceptor/>
      <standard-transaction-interceptor/>
    </command-service>
    <invoke-executor threads="10"/>
    <job-executor auto-start="false" lock="180000" threads="10"/>
    <hibernate-configuration name="hibernate-configuration:history">
      <properties resource="hibernate-history.properties"/>
      <mappings resource="hibernate/bpel.monitoring.mappings.xml"/>
      <mapping resource="hibernate/bpel.util.hbm.xml"/>
    </hibernate-configuration>
    <hibernate-session-factory configuration="hibernate-configuration:history" init="eager" name="hibernate-session-f
    <dead-job-handler class="org.ow2.orchestra.services.handlers.impl.ExitInstanceDeadJobHandler"/>
    <repository class="org.ow2.orchestra.services.impl.DbRepository"/>
    <publisher class="org.ow2.orchestra.cxf.CxfPublisher"/>
    <invoker class="org.ow2.orchestra.cxf.CxfInvoker" name="serviceInvoker"/>
  </environment-factory>
  <environment>
    <hibernate-session factory="hibernate-session-factory:core" name="hibernate-session:core"/>
    <runtime-db-session name="runtime-session:core" session="hibernate-session:core"/>
    <transaction/>
    <timer-session retries="10"/>
    <message-session retries="10" use-fair-scheduling="false"/>
    <job-db-session session="hibernate-session:core"/>
    <journal class="org.ow2.orchestra.persistence.db.DbJournal" name="journal">
      <arg>
        <ref object="querier-session:core"/>
      </arg>
    </journal>
    <querier-db-session name="querier-session:core" session="hibernate-session:core"/>
    <history class="org.ow2.orchestra.persistence.db.DbHistory" name="history">
      <arg>
        <ref object="querier-session:history"/>
      </arg>
    </history>
    <hibernate-session factory="hibernate-session-factory:history" name="hibernate-session:history"/>
    <querier-db-session name="querier-session:history" session="hibernate-session:history"/>
    <chainer name="recorder">
      <recorder class="org.ow2.orchestra.persistence.log.LoggerRecorder"/>
      <ref object="journal"/>
    </chainer>
    <chainer name="archiver">
      <archiver class="org.ow2.orchestra.persistence.log.LoggerArchiver"/>
      <ref object="history"/>
    </chainer>
    <queryApi name="queryList">
      <ref object="journal"/>
      <ref object="history"/>
    </queryApi>
    <chainer name="finished-instance-handler">
      <finished-instance-handler class="org.ow2.orchestra.services.handlers.impl.DeleteFinishedInstanceHandler"/>
      <finished-instance-handler class="org.ow2.orchestra.services.handlers.impl.ArchiveFinishedInstanceHandler"/>
    </chainer>
    <chainer name="undeployed-process-handler">
      <undeployed-process-handler class="org.ow2.orchestra.services.handlers.impl.ArchiveUndeployedProcessHandler"/>
    </chainer>
  </environment>
</environment-definition>
```

Currently, following objects implementations can be injected in the environment:

- **publisher:** object intended for publishing services of the given bpel process. For web services based on axis framework, degfault class is org.ow2.orchestra.axis.AxisPublisher. For web services based on cxf framework, the default class is org.ow2.orchestra.cxf.CxfPublisher.

- **invoker:** object intended for external web services invocations. Default implementation is based on SAAJ through the default implementation (class org.ow2.orchestra.services.impl.SOAPInvoker). For web services based on cxf framework, the default class is org.ow2.orchestra.cxf.CxfInvoker
- **repository:** data repository storing processes and instances... Db persistence (class org.ow2.orchestra.execution.services.db.DbRepository) implementation is included in this RC.
- **recorder:** object responsible of process execution logs. Default implementation handles process logs in the command line console (org.ow2.orchestra.persistence.log.LoggerRecorder). Recorder and Journal (see next) objects can be chained (new ones can be added as well on top of the recorder chainer). This give you a powerful mechanism to handle process execution data
- **journal:** object responsible for storing or retrieving process execution data. Db persistence (class org.ow2.orchestra.persistence.db.DbJournal) implementation is provided by default.
- **archiver:** object intended for process logs archiving. Default implementation handles logs on process data archiving through the default implementation (class org.ow2.orchestra.persistence.log.LoggerArchiver). Archiver and History (see next) objects can be chained (new ones can be added as well on top of the archiver chainer). This give you a powerful mechanism to handle process archived data
- **history:** object intended for storing or retrieving process archieved data. Default implementation is provided and available in the following class: org.ow2.orchestra.persistence.db.DbHistory.
- **queryList:** object intended to configure how the QueryRuntimeAPI will retrieve the process execution data. This retrieval could be configured to chain with the expected order into the journal and the history.
- **finished-instance-handler:** action to perform when a process instance is finished. This object could chain two or more distinct actions: for a given process instance, deleting the runtime object including its activities from the repository and then store data in the archive and remove data from journal. Default implementations are proposed for both chained actions.
- **undeployed-process-handler:** action to perform when a process is un-deployed. This object could chain distinct actions. Default implementation stores data in the archive and removes data from journal.
- **dead-job-handler:** action to perform when a asynchronous execution has failed all the retries. This object could chain distinct actions. Default implementation exits the process instance that failed to execute asynchronously.

* Note 1: As explained before persistence objects are provided as default implementations in the environment. Notice that in a persistence configuration additional resources are required, i.e for hibernate persistence you can specify mappings, cache configuration...

* Note 2: The environment is divided in two different contexts: environment-factory and environment. Objects declared inside the environment-factory context are created once and reused while objects declared inside the environment context are created for each operation.

4.3. Services

Services in Orchestra is all about pluggability. To allow that, each service has been thought in terms of an interface with different possible implementations. In the following lines you will find a description of main services supported in Orchestra.

The PVM includes a framework to allow the injection of services and objects that will be leveraged during the process definition and execution. Objects and services required in Orchestra are defined through an XML file. A dedicated parser and wiring framework in the PVM is in charge of creating those objects.

A default environment file (environment.xml) is provided in the installed package.

Currently, following objects are required for the execution environment :

- publisher
- invoker
- repository
- persistence
- timer
- journal and history
- querier

Example of implementation classes for these objects are embedded into the Orchestra jar and defined into the environment.xml file.

4.3.1. Publisher

The publisher service sets the way the services proposed by the BPEL processes will be published. The default implementation of this service uses the Axis Web Service Container.

4.3.2. Invoker

The invoker service sets the way the BPEL processes will call external services. The default implementation of this service uses the SAAJ implementation.

4.3.3. Repository

The repository service sets the way the data will be handled by the engine. Orchestra proposes one implementation managing data in the database.

4.3.4. Persistence

Persistence is one of key technical services injected into the services container. This service, as well as other major services in Orchestra, is based on a service interface. That means that multiple persistence implementations can be plugged on top.

The Persistence service interface is responsible to save and load objects from a relational database. By default, a persistence implementation based on the Hibernate ORM framework is provided (JPA and JCR to come).

The Process Virtual Machine core definition and execution elements (processes, nodes, transitions, events, actions, variables and executions) as well as the BPEL extension ones (activities, conditions, variables...) are persisted through this service. Process Virtual Machine core elements are also cached by leveraging the default persistence service implementation (Hibernate based). Processes and instances are stored through this persistence service. Repository is the term used in Orchestra to store those entities.

This service is only used if the repository service is set to database.

4.3.4.1. Database Access Configuration

The default configuration of Orchestra uses the Database persistence service and the H2 Database. Orchestra has also been tested with Oracle, MySQL and Postgres database system. To change to mysql, postgres or Oracle, you need to install the corresponding JDBC driver (see Chapter 3, *Installation guide*) and modify the `hibernate.properties` file : uncomment the corresponding lines :

```
# Hibernate configuration

# For using Orchestra with H2
# hibernate.dialect                      org.hibernate.dialect.H2Dialect
# hibernate.connection.driver_class      org.h2.Driver
# hibernate.connection.url               jdbc:h2:file:db_orchestra
# hibernate.connection.username          sa
# hibernate.connection.password

# For using Orchestra with postgreSQL
# hibernate.dialect                      org.hibernate.dialect.PostgreSQLDialect
# hibernate.connection.driver_class      org.postgresql.Driver
# hibernate.connection.url               jdbc:postgresql://server:port/db
# hibernate.connection.username          user
# hibernate.connection.password          pass

# For using Orchestra with MySQL
# hibernate.dialect                      org.hibernate.dialect.MySQL5InnoDBDialect
# hibernate.connection.driver_class      com.mysql.jdbc.Driver
# hibernate.connection.url               jdbc:mysql://server:port/db
# hibernate.connection.username          user
# hibernate.connection.password          pass

hibernate.dialect                      org.hibernate.dialect.H2Dialect
hibernate.connection.driver_class      org.h2.Driver
hibernate.connection.url               jdbc:h2:file:db_orchestra
hibernate.connection.username          sa
hibernate.connection.password

hibernate.hbm2ddl.auto                 update
hibernate.cache.use_second_level_cache true
hibernate.cache.provider_class         org.hibernate.cache.HashtableCacheProvider
hibernate.show_sql                     false
hibernate.format_sql                   false
hibernate.use_sql_comments             false
hibernate.bytecode.use_reflection_optimizer true
```

4.3.5. Journal and History

This module concerns the way in which the process data is stored during the process execution and archived when the execution is completed. This is indeed a crucial module in a process solution.

Orchestra unifies journal data et history data as the underlying essence of both is to handle process data. For that to be done, we created the concept of process record. A record is a minimal set of attributes describing a process entity execution. That means that each process entity related to the execution has its own associated record.

Those records are recorded during the process execution and stored depending on the persistence service implementation (db, xml...). The Orchestra API will retrieve record data from the records storage and sent them back to the users (meaning that records also acts as value objects in Orchestra APIs).

As soon as a process instance is finished, a typical scenario would be (by default) to move instance related process data from the production environment to a history one. While the physical device and the data structure could changed from one process engine deployment to another (XML, BI database...), the internal format could remain the same (records). This is exactly what is happening in Orchestra, when archiving data the engine just move execution records from the production to the history environment without data transformation in between.

Journal and history data are persisted in different database. Change hibernate.properties [hibernate-history.properties] file in conf directory to modify the journal [history] database

4.3.6. Querier

The querier is an API tool for getting process records corresponding to different criteria. It can get records from journal, from history or both. This possibility is defined in the environment. Several parameters will allow us to obtain this information by various criteria: their state (running or after delivery) or ID. Depending on the circumstances this request will return a record, a set of records or an empty table.

4.3.7. Command service

The command service manages the environments and the transactions in Orchestra. Each process or api method is executed by the command service.

Its definition in the environment is the following :

```
<command-service>
  <orchestra-retry-interceptor delay-factor="2" delay="100" max-delay="10000" retries="10"/>
  <environment-interceptor/>
  <standard-transaction-interceptor/>
</command-service>
```

The *retries* (optional) attribute can be used to define how many times commands will be tried before propagating the exception. This attribute is ignored for jobs (see Section 4.3.8, “Asynchronous Executions (Jobs)”).

The *delay*, *max-delay*, *delay-factor* (optional) attributes can be used to define the sleep time between each retry. The sleep time for the n^{th} attempt is given by

```
sleep = min($max-delay, $delay * $delay-factor * random(1, $delay-factor)^n)
```

delay and *max-delay* values are in milliseconds.

4.3.8. Asynchronous Executions (Jobs)

To optimize the execution, Orchestra splits the execution in small steps. A job represents a step of an execution. It can be executed in parallel with other jobs. Jobs are grouped in two sets: messages, which can be executed immediately, and timers, whose executions are scheduled at a precise date.

Timers are used for the BPEL statements "wait" and "onAlarm".

Messages are used for the BPEL statements "receive", "onEvent", "onMessage" and "invoke".

Orchestra uses the PVM Job executor framework to handle jobs. Jobs are created using either the timer session service or the message session service. The job executor then fetches the job from the database and perform the instructions contained in the job.

4.3.8.1. Timer session

A timer session service is required to schedule timers in the PVM. Its definition in the environment is the following :

```
<timer-session retries="5"/>
```

The timer session *retries* (optional) attribute can be used to define how many times the job will be retried before it becomes dead (see Section 4.3.8.4, “Dead jobs”).

4.3.8.2. Message session

A message session service is required to schedule messages in the PVM. Its definition in the environment is the following :

```
<message-session retries="5" use-fair-scheduling="false"/>
```

The message session *retries* (optional) attribute can be used to define how many times the job will be retried before it becomes dead (see Section 4.3.8.4, “Dead jobs”).

The message session *use-fair-scheduling* (optional) attribute can be used to define ifthe jobs should be executed with the same priority or if the jobs of older instances should be executed first.

4.3.8.3. Job Executor

The job executor fetches the jobs to execute from he database, and then executes the job.

Default implementations of the job executor uses a thread pool to execute jobs in parallel.

There are two default implementations of the job executor:

- an implementation using a thread pool with a fixed size. This service is defined in the environment file with the following line :

  ```
  <job-executor threads='10' auto-start='false' />
  ```

  The number of thread is defined by the *threads* attribute. This implementation is the default implementation.

- an implementation using a thread pool with a variable size (based on java.util.concurrent.ExecutorService). This service is defined in the environment file with the following line :

  ```
  <job-executor type='jdk' auto-start='false' />
  ```

Optional attributes can be defined in the environment to configure the job executor service:

- *command-service*: name of the command service to use to execute jobs. Only necesssary if more than one command service exists.
- *dead-job-handler*: name of the command service to use to handle dead jobs. Only necesssary if more than one dead job handler exists.
- *idle*: polling interval of the job database (in milliseconds). Note that the job executor is notified of job added by message and timer session services. Polling is just to check no notification has been missed.
- *lock*: before a job is executed by a job executor thread, the thread locks the job to be sure no other threads executes the same job. The *lock* attribute specifies the duration of the lock (in milliseconds). When the lock expires, a new thread can execute the job again (can happen if a job executor thread dies unexpectedly).
- *limit-job-per-instance*: specifies if jobs of a process instance should be executed sequentially. If this attribute is set to *false*, jobs of the same instance can be executed in parallel.

4.3.8.4. Dead jobs

If an exception occurs during a job execution, the job executor will decrement the retry counter of the job. While the retry counter is positive, the job executor will pick the job and try to execute it again.

When the job retry counter is zero, the job executor will not execute the job again. The job becomes a dead job.

If a dead job handler exists in the environment, it will be executed.

The default retry counter value for a job can be set in the message session configuration.

4.3.8.4.1. Dead Job Handler (DJH)

DJH are executed after a job retry counter has reached zero. Orchestra provides following implementations in the package *org.ow2.orchestra.services.handlers.impl*:

- *ExitInstanceDeadJobHandler* : exits the BPEL instance which has faulted.
- *ThrowBpelFaultDeadJobHandler* : throw a BPEL fault {http://orchestra.ow2.org}jobExecutionFault to the BPEL instance which has faulted. The fault can be handled by a catch activity.

By default, the *ExitInstanceDeadJobHandler* is enabled.

4.3.8.4.2. Interacting with dead jobs

The management API provides methods to find dead jobs and to reset the retry counter of a job to a specified value.

If you want to manage the dead jobs manualy, you need to disable the Dead Job Handler.

Refer to Section 6.1, “Orchestra APIs” for more information on how to use the APIs.

4.3.8.5. Invoke Executor

The invoke executor executes external web services calls. Invoke are executed by a separate thread pool. The thread is hold until the web service response is received.

Default implementation of the invoke executor uses a thread pool to execute invoke in parallel.

The default implementation of the invoke executor use a thread pool with a fixed size. This service is defined in the environment file with the following line :

```
<invoke-executor threads='10' />
```

The number of thread is defined by the *threads* attribute. This parameter defines the maximum number of invoke that can be executed simultaneously.

4.3.9. Finished instance handler (FIH)

FIH are executed after the instance finished. Orchestra provides following implementations in the package *org.ow2.orchestra.services.handlers.impl*:

- *NoOpFinishedInstanceHandler* : do nothing
- *CleanJournalFinishedInstanceHandler* : remove instance data from journal
- *ArchiveFinishedInstanceHandler* : remove instance data from journal and put it in history
- *DeleteFinishedInstanceHandler* : delete instance data from orchestra repository

4.3.10. Undeployed process handler (UPH)

UPH are executed after the process is undeployed. Orchestra provides following implementations in the package *org.ow2.orchestra.services.handlers.impl* :

- *NoOpUndeployedProcessHandler* : do nothing
- *CleanJournalUndeployedProcessHandler* : remove process data from journal
- *ArchiveUndeployedProcessHandler* : remove process data from journal and put it in history

4.3.11. Clustering configuration

Orchestra can run in a clustered environment.

In a clustered environment, all Orchestra nodes share the same database.

When a process is deployed in Orchestra, the process web services are deployed on each node of the cluster. An instance of the process can execute on any node of the cluster.

> **Important**
> *In a clustered environment, reply activities are not supported. The web services exported by Orchestra are only one-way web services.*

In this version, Orchestra cluster configuration is done by declaring the cluster nodes in the environment.xml file.

To declare a cluster, add these lines to the *environment-factory* part of the configuration file:

```
<static-cluster>
  <jmx-server serviceUrl="..." objectName="..." />
  <jmx-server serviceUrl="..." objectName="..." />
</static-cluster>
```

Each *jmx-server* element describes an Orchestra node. The *serviceUrl* and *objectName* attributes are the parameters to use to connect to the JMX interface of the node. These values are configured for each node in the *orchestra.properties* file.

Chapter 5. User guide

5.1. Start and Stop Orchestra

Orchestra is a webapp that can be deployed on Tomcat. So starting Orchestra in fact starts Tomcat with the correct environment. This can be performed from the installation directory with the following command line :

```
>cd orchestra-tomcat-4.6.2
 >ant start
```

Starting Orchestra will not be done in background. This means that the console starting Orchestra will be dedicated to the traces from Orchestra. To perform further actions, new consoles need to be opened.

To stop Orchestra, type the following command line :

```
>cd orchestra-tomcat-4.6.2
  >ant stop
```

5.2. Deploying / undeploying a process

Once Orchestra is started, it is then possible to deploy a new process on the engine :

```
>ant deploy -Dbpel=<process>.bpel -Dwsdl=<process>.wsdl -Dextwsdl=<wsdl1,wsdl2>
```

Orchestra also provides the possibility to use an archive to deploy a process. This archive should be a zip file with the extension .bar. Here is the command line to deploy such an archive :

```
>ant deploy -Dbar=<process>.bar
```

Warning : The archive should be a zip file structured as described bellow :

```
/<process>.bpel
/<process>.wsdl
/<files>.wsdl
```

To undeploy a process, use the following command line :

```
>ant undeploy -Dprocess=<process_name>
```

Warning : the process name should be fully qualified. This means that it needs to contain to namespace. For instance :

```
{http://orchestra.ow2.org/weather}weather
```

5.3. Other commands

Orchestra provides a set of other commands that can be usefull

- A command to check the status of Orchestra. This command tells if the engine is started and if so, gives the names of processes deployed on the engine :

  ```
  >ant status
  ```

- A command to simulate a Web Service call. This command will simulate a WS call to interact with a deployed process :

```
>ant call -Dendpoint=<service_url> -Daction=<SOAP_action> -Dmessage=<message>
```

For example :

```
>ant call -Dendpoint=http://localhost:8080/orchestra
          -Daction=http://orchestra.ow2.org/weatherArtifacts/process/weatherPT
          -Dmessage="<weatherRequest xmlns='http://orchestra.ow2.org/weather'>
                    <input>Grenoble,France</input>
                    </weatherRequest>"
```

5.4. Running the examples

The Orchestra package contains examples of BPEL processes:

- loanApproval: invokes two local web services. This example is taken from the BPEL 2.0 standard.

  This is an example from the BPEL 2.0 standards

- weather: invokes a remote Web Service and returns the current weather.

  This example shows how to call a real world Web Service.

- echo

  This example shows a basic synchronous bpel process.

- orderingService

  This example shows how to use pick instruction and correlations.

- producerConsumer

  This example shows how executions can be saved and restarted after a crash.

A build.xml file is provided for each of those samples. Those ant scripts provide the same targets to deploy, launch and undeploy the sample. Go to the desired example and use the command lines :

```
>ant deploy
>ant launch
>ant undeploy
```

5.5. Running the tests

Orchestra is delivered with a test suite to check if your installation is correct. There are 3 differents tests available :

- *Core test suite*. This suite tests the core functionnalities of the engine (e.g. BPEL activities, variables, etc...). To run this test suite, the server should not be started.This test suite can be launched with the following command :

  ```
  >ant test
  ```

- *Remote test suite*. This suite gives the possibility to test the Web Service stack deploying and launching real processes. This test suite can be launched with the following command (server should be started) :

  ```
  >ant test-remote
  ```

- *Stress test suite*. This suite will launch a small stress test. This test suite can be launched with the following command line :

```
>ant test-stress
```

A command is also provided to launch those 3 test suites at once :

```
>ant test-all
```

The results of the tests are available under the directory `testresults`.

5.6. Configuring Logger

It is possible to activate the logs. To do so, the file `logging.properties` under the directory `conf/` can be edited. Here is the content of that file :

```
handlers= java.util.logging.ConsoleHandler
.level= SEVERE
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter = org.ow2.orchestra.util.TraceFormatter

org.ow2.orchestra.util.TraceFormatter.alias=\
org.ow2.orchestra.pvm.internal.wire.descriptor.HibernateConfigurationDescriptor~hibernateConfiguration,\
org.ow2.orchestra.deployment.Deployer~deployer,\
org.ow2.orchestra.facade.jmx.RemoteDeployerMBean~api,\
org.ow2.orchestra.persistence.log.LoggerRecorder~recorder,\
org.ow2.orchestra.persistence.log.LoggerArchiver~archiver
# For example, set the com.xyz.foo logger to only log SEVERE messages:
# com.xyz.foo.level = SEVERE

org.hibernate.level=SEVERE
org.ow2.orchestra.pvm.internal.wire.descriptor.HibernateConfigurationDescriptor.level=FINE
org.hibernate.event.def.AbstractFlushingEventListener.level=OFF
org.ow2.orchestra.jmx.level=INFO
org.ow2.orchestra.osgi.Engine.level=INFO
org.ow2.orchestra.pvm.internal.svc.DefaultCommandService.level=OFF
org.ow2.orchestra.pvm.internal.tx.StandardTransactionInterceptor.level=OFF
org.ow2.orchestra.deployment.Deployer.level=FINE
org.ow2.orchestra.test.EnvironmentTestCase.level=FINE
org.ow2.orchestra.test.remote.RemoteTestCase.level=FINE
org.ow2.orchestra.pvm.level=WARNING
org.ow2.orchestra.level=WARNING
org.ow2.orchestra.StartupListener.level=INFO
#org.ow2.orchestra.persistence.log.level=FINE
```

Uncomment the last lines to activate the logs.

5.7. Monitoring and administration with JMX

Orchestra registers several MBeans which provide some monitoring information.

- Orchestra API is exposed as a JMX MBean (default objectName: *Orchestra:type=RemoteAPI*). Orchestra API is detailled in Section 6.1, “Orchestra APIs”.
- Orchestra job and invoke executor thread pools can be managed with JMX (default objectName: *Orchestra:type=threadPool,name=JobExecutor* and *Orchestra:type=threadPool,name=InvokeExecutor*). Orchestra thread pool mbean is detailled in Section 5.7.1, “Orchestra MBean for thread pools”.
- Hibernate MBeans (default objectName: *Orchestra:type=Hibernate,name=hibernate-session-factory_core* and *Orchestra:type=Hibernate,name=hibernate-session-factory_history*). These MBeans provide statistics about the Hibernate sessions. Refer to Hibernate documentation [http://docs.jboss.org/hibernate/stable/core/reference/en/html/] for more information about the statistics.
- C3P0 MBeans (default objectName: *com.mchange.v2.c3p0:type=PooledDataSource**). These MBeans provide statistics about the Hibernate JDBC connection pool. Refer to C3P0 documentation

[http://www.mchange.com/projects/c3p0/index.html#jmx_configuration_and_management] for more information.

- Tomcat MBeans (default objectName: *Catalina:**). These MBeans provide informations about Tomcat. Refer to Tomcat documentation [http://tomcat.apache.org/tomcat-6.0-doc/funcspecs/fs-admin-objects.html] for more information.

5.7.1. Orchestra MBean for thread pools

Available attributes:

- *ActiveCount* : Number of threads currently executing a task.
- *CompletedTaskCount* : Number of tasks completed.
- *CorePoolSize* : Maximum number of threads of this pool. This attribute can be modified.
- *PoolSize* : Current number of threads of this pool.
- *TaskCount* : Number of tasks submitted to the thread pool. This includes tasks already completed, tasks currently being executed and tasks waiting for execution.
- *WaitingTaskCount* : Number of tasks currently waiting for execution.

5.8. Using Apache Camel with Orchestra

When using Orchestra with CXF Web Service framework, Orchestra can use Apache Camel as transport for web services interactions.

Orchestra-Camel integration allows processes to produce/consume messages on the Camel context. It allows a process to use for example JMS, mail, file connectors to connect to remote services.

For more information about Apache Camel features, please read Camel documentation [http://camel.apache.org/user-guide.html]

5.8.1. How to create a Camel context for a process ?

Orchestra uses Camel Spring [http://camel.apache.org/spring.html] language to describe routes. To define the Camel routes deployed with a process, add a camel-context.xml file in your BAR archive. Orchestra will deploy and start the routes with the process. If your camel-context.xml uses external Java classes, you can add them too to the BAR archive.

Example of camel-context.xml file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-spring.xsd">
  <camelContext xmlns="http://camel.apache.org/schema/spring" autoStartup="false">
    <route>
      <from uri="file:///inputDir" />
      <to uri="direct:hello"/>
    </route>
  </camelContext>
</beans>
```

5.8.2. How to use camel context instead of HTTP for Web Service interactions ?

In the WSDL file of the service you want to invoke or expose in the camel context,

- change to transport defined in the SOAP binding element to *http://cxf.apache.org/transports/camel*
- change the location of the service defined in the SOAP address element to *camel://camel_endpoint* (where *camel_endpoint* is the endpoint you want to expose/invoke in the camel context)

Example of WSDL Service configured to use Camel:

```
<wsdl:binding name="helloworldPTSOAPBinding" type="tns:helloworldPT">
  <soap:binding style="document" transport="http://cxf.apache.org/transports/camel"/>
  <wsdl:operation name="submit">
    <soap:operation soapAction="http://orchestra.ow2.org/helloworld/submit"/>
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="helloworldService">
  <wsdl:port name="helloworldPort" binding="tns:helloworldPTSOAPBinding">
    <soap:address location="camel://direct:hello"/>
  </wsdl:port>
</wsdl:service>
```

5.9. Process versioning with Orchestra

Since Orchestra 4.6, versioning of process is supported.

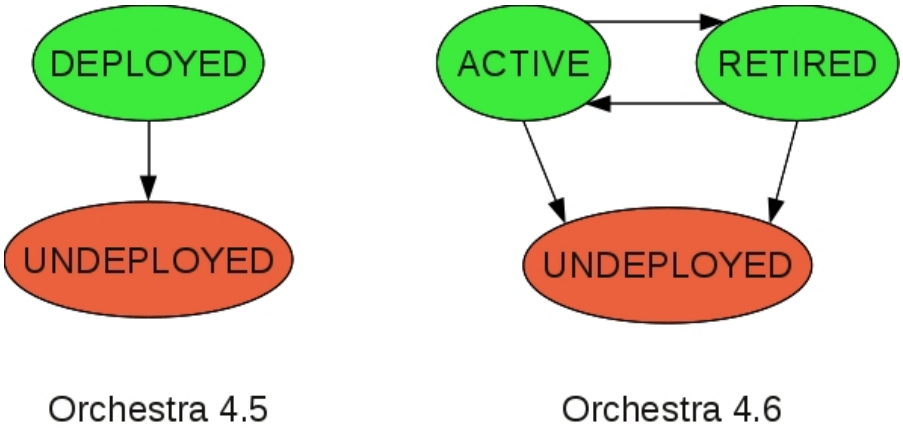
5.9.1. Process lifecycle

A process has three states:

- *active*: this is the default state when a process is deployed. The process can create new instances, and running instances can finish their execution. Process web services are deployed and active.
- *retired*: The process cannot create new instances, but running instances can finish their execution. Process web services are deployed and active.
- *undeployed*: The process cannot create new instances. there are no more running instances. Process web services are not deployed (and disabled).

A process can change state from:

- *active* to *retired*
- *retired* to *active*
- *active* to *undeployed*
- *retired* to *undeployed*



Process lifecycle.

5.9.2. Process versions

To deploy a new version of a process, just deploy a new process with the same name and namespace as an already deployed process. The version is automatically incremented.

Only one version of a process can be *active*. If multiple versions are deployed:

- when a version becomes *active*, the previously *active* version becomes *retired*.
- when the *active* version is *undeployed*, the highest *retired* version becomes *active*.
- when a new version is deployed, it becomes *active*.

5.9.3. Restrictions on versioning

There are some restrictions when changing the process web services between process versions.

If a web service is shared by two versions of a process (same endpoint address), it must have the same WSDL definition (same portType, same binding...) in the two versions.

A new version of a process can add/remove web services (if receive activities are added, removed), as long as they use a separate portType.

A web service cannot be shared by two different processes (processes with different names).

Chapter 6. Developer's guide

This chapter describes how to start playing with Orchestra:

- How to develop a simple application by leveraging Orchestra APIs

6.1. Orchestra APIs

6.1.1. Getting started with Orchestra APIs

Actually, four different APIs are available in Orchestra :

- QueryRuntimeAPI that gives runtime information about instances of process and activities.
- QueryDefinitionAPI that gives information of process definition.
- ManagementAPI that gives the possibility to manage Orchestra (deploy / undeploy processes, etc...)
- InstanceManagementAPI that gives the possibility to manage instances of process (suspend / resume / exit process instance)

You can find detailled information about APIs in the javadocs. APIs are included in a Maven module. To include this module you have to add following Maven dependency :

```
<dependency>
  <groupId>org.ow2.orchestra</groupId>
  <artifactId>orchestra-api</artifactId>
  <version>4.6.2</version>
</dependency>
```

If you do not want / can't use Maven, you can create a Maven module which depends only on this dependency and create an assembly. Then copy created jar to your project.

- **QueryRuntimeAPI**: to get recorded/runtime informations for instances and activities. It allows also to get activities per state. Then operations in this API applies to process instances and activity instances.

Hereafter you will find an example on how to access to the QueryRuntimeAPI from your client application:

```
org.ow2.orchestra.facade.QuerierRuntimeAPI querierRuntimeAPI =
               org.ow2.orchestra.facade.AccessorUtil.getQueryRuntimeAPI(String, String);
```

The method getQueryRuntimeAPI takes two arguments of type String. The first argument is the URL of the JMX service. The second is the name of the object JMX. In case we use the default configuration, two constants can be used which are: AccessorUtil.SERVICE_URL and AccessorUtil.OBJECT_NAME.

For a detailed insight on Orchestra APIs, please take a look to the Orchestra javadoc APIs (available under / javadoc directory)

Similar methods exists to access to the QueryDefinitionAPI, ManagementAPI and InstanceManagementAPI.

6.2. Orchestra Client jar

If you want to call Orchestra APIs from a remote application you can use the Orchestra client jar. It contains all the needed classes for you to build you application. Just download the jar and include it in your classpath.

6.3. Adding new Orchestra services implementations

Orchestra uses OSGi services to find extensions. To find services implementations, Orchestra use *org.ow2.orchestra.osgi.OrchestraExtensionService* services. These services simply return the classes to use in orchestra.

To use your own implementation of a service, you need to package it in an OSGi bundle. The bundle should export a *org.ow2.orchestra.osgi.OrchestraExtensionService* service. The implementation of the method *getExtension(className)* should return the extension class when the className is the name of the extension, *null* otherwise.

org.ow2.orchestra.osgi.ExtensionActivator class provides a base for registering extensions. See javadoc for more details on how to use this class.