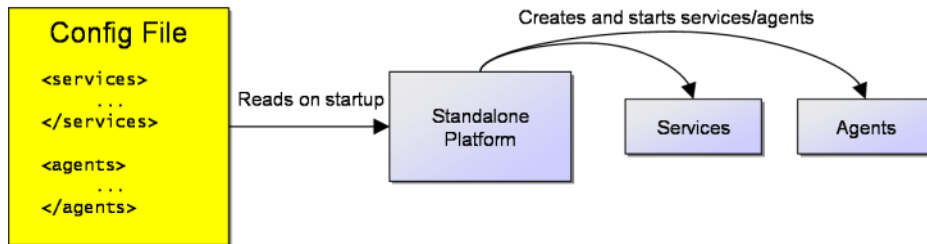


Introduction

The Jadex Standalone Platform is a lightweight pure Java SE based execution environment. It has been built based on the idea of a minimal infrastructure that offers all its functionalities by so called *platform services*. These services can be customized freely and hence the Standalone Platform can be adapted to the concrete application scenario by simply changing its declarative configuration. The basic functioning of the platform is depicted in the figure below.



Startup process of the Standalone platform

On startup the Standalone Platform reads the specified configuration file and evaluates it with respect to the initial services and agents to start. The platform then first starts the specified services in the order of their declaration in the configuration file. After all services have been initialized the platform also starts the initial agents in the given order. Hereby, between two kinds of agents is distinguished. *Daemon agents* are distinguished from normal application agents for platform shutdown determination. When in the configuration the platform auto shutdown feature is enabled the platform will be terminated when the last application agent has been killed, i.e. even when some daemon agents (e.g. ams or df) are still active at this point in time.

Starting the Platform

The Jadex Standalone Platform comes in two slightly different flavors, which offer both the same kinds of functionalities and only differ in the way they can be configured. They can be started using the following command lines:

java jadex.adapter.standalone.Platform [-conf conf.xml]

If you omit the last part specifying the configuration file the platform will search the fallback configuration jadex/adapter/standalone/standalone_conf.xml via the classpath.

java jadex.adapter.standalone.SpringPlatform [-conf springconf.xml]

If you omit the last part specifying the configuration file the platform will search the fallback configuration jadex/adapter/standalone/standalone_springconf.xml via the classpath.

Given that the standard configurations are used, after startup the Jadex Control Center (JCC) pops up. It is useful for loading and starting Jadex agents. Please refer to the tool guide (ref) if you are interested in more information about the available JCC tools.

If the platform does not start or the Control Center user interface does not show up, check if you have all necessary libraries (see below). Alternatively, you can start the platform also via the java jar option without having to adjust the classpath via:

java -jar jadex-launch-2.0-beta4.jar

or by simply double clicking this jar (if jar files are correctly associated with the Java SE).

The Jadex Core Libraries

- jadex-standalone-2.0-beta4.jar
- jadex-platform-base-2.0-beta4.jar
- jadex-commons-2.0-beta4.jar
- jadex-javaparser-2.0-beta4.jar
- jadex-runtimetools-2.0-beta4.jar
- jadex-applib-bdi-2.0-beta4.jar
- jadex-kernel-bdi-2.0-beta4.jar
- jadex-rules-2.0-beta4.jar
- jadex-kernel-micro-2.0-beta4.jar
- jadex-bridge-2.0-beta4.jar
- jadex-nuggets-2.0-beta4.jar
- jadex-rules-tools-2.0-beta4.jar

XML Parsing Libraries

- stax-1.2.0.jar
- stax-api-1.0.1.jar
- xpp3-1.1.4c.jar
- janino-2.5.10.jar

Parser Generator Libraries

- javacc-4.0.jar
- antlr-runtime-3.1.3.jar
- antlr-2.7.7.jar

- stringtemplate-3.2.jar

JOGL Support Libraries (optional, for fast OpenGL graphics support of examples)

- jogl-1.jar
- gluegen-rt-1.jar

Spring Beans Libraries (only necessary when using Standalone SpringPlatform)

- spring-beans-2.5.6.jar
- spring-core-2.5.6.jar
- spring-context-2.5.6.jar
- aopalliance-1.0.jar
- commons-logging-1.1.1.jar

Email/ICQ Communication Libraries (optional, necessary for some examples)

- mail-1.4.1.jar
- activation-1.1.jar
- jcq2k-1.jar

Graph Support Libraries (necessary for tools)

- jung-api-2.0-beta1.jar
- jung-algorithms-2.0-beta1.jar
- jung-graph-impl-2.0-beta1.jar
- jung-visualization-2.0-beta1.jar
- collections-generic-4.01.jar
- colt-1.2.0.jar concurrent-1.3.4.jar

Chart Visualization Libraries (necessary for tools)

- jfreechart-1.0.9.jar
- jcommon-1.0.12.jar

Java Online Help Libraries (necessary for tools)

- javahelp-2.0.02.jar

Configuring the Platform

The configuration of the platform can be done either via a custom property based Jadex XML file or via the Spring beans XML format.

Property based Jadex XML

```
<properties xmlns=
"http://jadex.sourceforge.net/jadexconf"
  xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
"http://jadex.sourceforge.net/jadexconf
          http://jadex.sourceforge.net/configuration.xsd"
  name=
"standalone_conf">

<properties type=
"platform_standalone" name=
"lars">

<property type=
"platformname">
lars
</property>

<property type=
"platform_shutdown_time">
1000
</property>

<property type=
"welcome">
false
</property>

<property type=
"autoshtutdown">
true
</property>

<property type=
"application_factory">
    new jadex.adapter.base.appdescriptor.ApplicationFactory(...)
```

Standalone Platform Guide - Introduction

```
</property>

<property type=
"messagetype" name=
"fipa">
new jadex.adapter.base.fipa.FIPAMessageType()
</property>

<properties type=
"services">

<property type=
"jadex.bridge.ILibraryService" name=
"library_service">
    new jadex.adapter.base.libraryservice.LibraryService()
</property>
...
</properties>

<property type=
"daemonagent" name=
"ams">
jadex/bdi/amsagent/AMS.agent.xml
</property>

<property type=
"daemonagent" name=
"df">
jadex/bdi/dfagent/DF.agent.xml
</property>

<property type=
"agent" name=
"jcc">
jadex/tools/jcc/JCC.agent.xml
</property>

<properties type=
"kernel" id=
"kernel_v2_bdi">

<property type=
"agent_factory">
new jadex.bdi.interpreter.BDIAgentFactory(...)
```

```
</property>

</properties>
...

</properties>

</properties>
```

Simplified version of a configuration XML property file

The properties format is very simple. Properties can contain basic property values (<property>) as well as further property containers (<properties>). A property as well as a property container is defined by a type and a name attribute. The type describes the meaning of the property (container) and the name further qualifies a certain instance, e.g. there is a properties section with type "kernel", which contains information about kernel settings. In the example only the "kernel_v2_bdi" kernel has been specified. In the original config file contained in the distribution also another kernel called "kernel_v2_micro" has been defined.

Spring Beans XML

```
<?xml version=
"1.0" encoding=
"UTF-8"?>

<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-2.0.xsd">

<bean name=
"platformName" class=
"java.lang.String">

<constructor-arg type=
"java.lang.String">

<value>lars
</value>

</constructor-arg>

</bean>

<bean id=
"platform" class=
```

Standalone Platform Guide - Introduction

```
"jadex.adapter.standalone.SpringPlatform">
```

```
<property name=
```

```
"platformName" ref=
"platformName" />
```

```
<property name=
```

```
"shutdownTime">
```

```
<value type=
```

```
"long">
10000
```

```
</value>
```

```
</property>
```

```
<property name=
```

```
"messageTypes">
```

```
<map>
```

```
<entry>
```

```
<key>
```

```
<value>fipa
```

```
</value>
```

```
</key>
```

```
<bean class=
```

```
"jadex.adapter.base.fipa.FIPAMessageType"/>
```

```
</entry>
```

```
</map>
```

```
</property>
```

```
<property name=
```

```
"applicationFactory">
```

```
<bean id=
```

```
"app_fac">
...
```

```
</bean>
```

Standalone Platform Guide - Introduction

```
</property>

<property name=
"agentFactories">

<list value-type=
"jadex.bridge.IJadexAgentFactory">

<bean id=
"kernel_v2_bdi_factory" class=
"jadex.bdi.interpreter.BDIAgentFactory">
...
</bean>

</list>

</property>

<property name=
"services">

<map>

<entry>

<key>

<value type=
"java.lang.Class">
jadex.bridge.ILibraryService
</value>

</key>

<map>

<entry>

<key>

<value type=
"java.lang.String">
library_service
</value>
```

Standalone Platform Guide - Introduction

```
</key>

<bean class=
"jadex.adapter.base.libraryservice.LibraryService">

</bean>

</entry>

</map>

</entry>

</map>

</property>

<property name=
"daemonAgents">

<map>

<entry>

<key>

<value>ams
</value>
</key>

<value type=
"java.lang.String">
jadex/bdi/amsagent/AMS.agent.xml
</value>

</entry>

<entry>

<key>

<value>df
</value>
</key>

<value type=
"java.lang.String">
jadex/bdi/dfagent/DF.agent.xml
</value>
```

```
</entry>

</map>

</property>

<property name=
"agents">

<map>

<entry>

<key>

<value>jcc

</value>

</key>

<value type=
"java.lang.String">
jadex/tools/jcc/JCC.agent.xml

</value>

</entry>

</map>

</property>

</bean>

</beans>
```

Simplified version of a spring beans configuration file

Configuration Aspects

Please note that the exact name of the corresponding properties differs slightly between both kinds of configuration files. The reason is that in case of a Spring beans configuration the name of a property is directly mapped to a bean property. Hence, the properties here are very similar to the set-methods of the SpringPlatform class (without the preceding "set" or "add". In the list that follows, both versions are mentioned (the second one being the Spring name).

Details about the different properties are described in the following list:

- **platformname | platformName [String]:** The name of the platform
- **platform_shutdown_time | shutdownTime [long]:** Time between the platform shutdown is initiated and the agents are forcefully terminate

Standalone Platform Guide - Introduction

- **autoshtutdown [boolean]**: If true, the platform shuts down automatically when the last application agent has been killed (and only daemon agents are left)
- **application_factory | applicationFactory [ApplicationFactory]**: The application factory, which is used to create agent applications
- **services [IPlatformService]**: May contain 0..* platform services. There are several predefined services available which are described in the following section. Each service is then specified by its type, name and the service object itself.
- **kernels | agentFactories [IAgentFactory]**: Specification of kernel dependent agent factories. Each kernel has to provide a means for creating agents of its corresponding type (e.g BDI or micro). The specification here depends on the configuration file. In case of a properties file, a properties section for a kernel needs to be defined. In the section a property agent_factory can be specified. In case of a Spring configuration file an agentFactories section is the container for an arbitrary number of agent factories.
- **daemonagent | daemonAgents**: Agent(s) that should be started at startup of the platform. It is specified via name and implementation file. A daemon agent is an agent that does not prevent the platform shutdown
- **agent | agents**: Agent(s) that should be started at startup of the platform. It is specified via name and implementation file.

Platform Services

In general, the usage of platform services by agents is done by fetching the service that is needed, casting it to the requested service type interface and then use the interface to invoke specific methods of the service. The concrete syntax for getting the service can be kernel dependent, but looks similar to the following:

```
<IServiceType> service = getPlatform().getService(<IServiceType.class>);
service.<serviceMethod(>);
```

Please note that most of the services will offer interfaces without direct return values. Using signatures with direct return values would force a tight binding between the caller and the called service. In order to allow a loose coupling often a callback mechanism is employed. It requires the caller to provide a *jadex.commons.concurrent.IResultListener* implementation as parameter of the call. The service will then call either the *resultAvailable(Object result)* method or the *exceptionOccurred(Exception e)* method according to the outcome of the service call. A call then typically has the following form:

```
IResultListener listener =
service.<serviceMethod>(...,

new IResultListener()
{

public void resultAvailable(
Object result)
{
    // handle result
}

public void exceptionOccurred(Exception e)
{
    // handle exception
}
});
```

AMS (Agent Management System)

The AMS service represents the white pages service of the platform. It can be used for all agent lifecycle operations, e.g. starting and killing a specific agent instance. The AMS internally keeps track of all created agent instances and manages entries which contains their agent identifier (which in turn hold the current transport addresses of an agent). Details about the methods provided by the AMS can be found in the API docs of the *jadex.adapter.base.fipa.IAMS* interface. Basically the interface provides methods for:

- creating, starting, suspending, resuming, destroying an agent
- searching for agents by AMS search constraints
- getting information about agents (the AMS agent description, the number of agents etc.)
- getting the external access of a local agent for direct OO-style interaction with an agent
- creating AMS service objects (AMS service descriptions, search constraints etc.)

DF (Directory Facilitator)

The directory facilitator is the yellow pages service of the agent platform and allows for searching for specific services. An agent may register an agent description at the DF, which contains the services that the agent offers. Registrations can either be persistent, meaning that the DF will keep the registered entry until it will be deregistered, or volatile, meaning that the registration will only be kept for a time interval. The interface of the df service is specified in *jadex.adapter.base.fipa.IDF*. It contains methods for the following purposes:

- registering and deregistering a DF agent description
- modifying an already registered description (e.g. for updating the lease time or changing the offered services)
- searching for registered services via a template DF agent description and further search constraints
- creating DF service objects for further usage (DF agent description, search constraints, etc.)

Clock Service

The clock service has two purposes. First, it can be used to get the current time. Secondly, timers can be created, which will be called when the specified amount of time has elapsed. The clock service abstraction allows different kinds of clock to be used, e.g. a real-time clock versus a simulation clock. Jadex currently supports the following clocks:

- system clock: delivers the time produced by the system hardware clock
- continuous clock: similar to real-time, but also provides a dilation and an offset
- simulation event clock: allows event-driven simulation, i.e. advances time to the next registered time points
- simulation tick clock: allows time-driven simulation, i.e. advances time in discrete time steps (ticks)

Execution Service

The execution service is a central unit, which has the responsibility to execute agents and other tasks when they announce to be executable. One main advantage of using an execution service instead of a decentralized approach (e.g. each agent executes on its own behalf) is that the execution service can be used to control the way the different tasks are executed. Jadex currently supports two execution services, the `AsyncExecutionService` and the `SyncExecutionService`. The former one emphasizes parallel execution of the tasks and hence may achieve very good performance results (especially on multi-core machines). The latter one focuses on sequentially ordered execution and should be used for simulations, as it helps to guarantee the repeatability of simulation runs that use the same settings. Basically, the execution service offers the following kinds of methods:

- Execute or cancel a task
- Add a command that will be executed whenever the executor gets idle, i.e. there are no tasks to execute.

Thread Pool Service

The thread pool service is a basic service that represents a service wrapper for a thread pool. Hence, it allows `Runnable`s to be executed via the service. This service will typically be used from other services that need to execute some task, e.g. an execution service that wants to execute an agent can rely on this service. The service only offers one method to:

- Execute a task (`java.lang.Runnable`)

Library Service

The classpath management is done via the library service in Jadex. This service can be used to modify the classpath of the running platform by adding new or removing old classpath entries represented as URLs. In addition, it is possible to fetch a classloader from the library service that represents the current configuration. Note, that the library service will change the classloader when the classpath is changed. The fetching of a classloader allows to work on a specific classpath configuration without having to worry about future changes. In general, the following kinds of methods are available through the service:

- Adding and removing URLs of classpath entries, i.e. either jar files or java classpath root directories (classes/bin)
- Fetching the managed entries (i.e. excluding system classpath settings) as URLs
- Getting a classloader that represents the current state of the classpath
- Adding/removing change listeners of the service. The listeners will be notified whenever changes with respect to the managed classpath occur.

Message Service

The message service has the purpose to facilitate the sending and delivering of messages. Typically, the message send process consists of the following steps: Sending of messages means that a message will be routed from the message service to one of the suitable message transports that are installed in the message service. The message service then marshalls the message and sends it to the receiving part of the (same) kind of transport. The transport on the receiving side unmarshalls the message and calls the deliver method on the message service. The message service finally delivers the message to the receiving agents of its platform. The offered methods allow for:

- Sending a message of a specified message type (e.g. FIPA). The message itself has to be prepared as map with parameter names as key and parameter values as map values.
- Delivering a message locally to the available receiver agents.

Simulation Service

The simulation service acts as a facilitator for agent simulations. It can be used to control the simulation progress by starting, pausing and stepping the application. The simulation service acts as mediator between the execution and the clock service by activating them in an alternating fashion. In normal mode it will wait until the execution service has finished and will then invoke the clock service to advance the time. This will cause new activities and trigger the execution service again. Hence the simulation works as a ping-pong between execution and time service mediated by the simulation service. Additionally, the service provides means for changing the simulation mode at runtime, i.e. changing the clock type that it used. The main types of methods are:

- start, pause and step the application. Stepwise execution is only possible when the application has been paused before. There are two different step modes available. An event step executes exactly one entry from the registered timing events of the clock. This does not necessarily advance the time as more than one event can be registered at the same point in time. In contrast with a time step the clock will be advanced to the next point in time and all event associated with that timepoint will be executed. Start also continues a paused application and will set it to normal run mode.
- Set clock type allows to change the underlying clock (cf. the explanation about different clock types within the section about the clock service).
- Listers can be added on the service to get notifications whenever changes occur with respect to settings of the service, e.g. when the clock type changes.