

Drools Integration User Guide

| | |
|--|----|
| | v |
| 1. Apache Camel Integration | 1 |
| 1.1. Camel | 1 |
| 2. Spring Integration | 5 |
| 2.1. Integration with Drools Expert | 5 |
| 2.1.1. Define a Resource | 5 |
| 2.1.2. Event Listeners | 11 |
| 3. Drools Server | 17 |
| 3.1. Introduction | 17 |
| 3.2. Deployment | 17 |
| 3.3. Configuration | 17 |
| 3.3.1. REST/Camel Services configuration | 17 |
| 4. Drools Commands | 23 |
| 4.1. API | 23 |
| 4.1.1. XStream | 23 |
| 4.1.2. JSON | 23 |
| 4.1.3. JAXB | 23 |
| 4.2. Commands supported | 24 |
| 4.2.1. BatchExecutionCommand | 26 |
| 4.2.2. InsertObjectCommand | 27 |
| 4.2.3. RetractCommand | 29 |
| 4.2.4. ModifyCommand | 30 |
| 4.2.5. GetObjectCommand | 31 |
| 4.2.6. InsertElementsCommand | 33 |
| 4.2.7. FireAllRulesCommand | 34 |
| 4.2.8. StartProcessCommand | 36 |
| 4.2.9. SignalEventCommand | 37 |
| 4.2.10. CompleteWorkItemCommand | 38 |
| 4.2.11. AbortWorkItemCommand | 39 |
| 4.2.12. QueryCommand | 40 |
| 4.2.13. SetGlobalCommand | 42 |
| 4.2.14. GetGlobalCommand | 43 |
| 4.2.15. GetObjectsCommand | 44 |
| 5. OSGi Integration | 47 |
| Index | 51 |



Chapter 1. Apache Camel Integration

1.1. Camel

Camel provides a light weight bus framework for getting information into and out of Drools.

Drools introduces two elements to make easy integration.

- Drools Policy

Augments any JAXB or XStream data loaders. For JAXB it adds drools related paths of the contextpath, for XStream it adds custom converters and aliases for Drools classes. It also handles setting the ClassLoader to the targetted ksession.

- Drools Endpoint

Executes the payload against the specified drools session

Drools can be configured like any norma camel component, but notice the policy that wraps the drools related segments. This will route all payloads to ksession1

Example 1.1. Drools EndPoint configured with the CXFRS producer

```
<bean id="droolsPolicy" class="org.drools.camel.component.DroolsPolicy" />

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="cxfrs://bean://rsServer"/>
      <policy ref="droolsPolicy">
        <unmarshal ref="xstream" />
        <to uri="drools:nodel/ksession1" />
        <marshal ref="xstream" />
      </policy>
    </route>
  </route>
</camelContext>
```

It is possible to not specify the session in the drools endpoint uri, and instead "multiplex" based on an attribute or header. In this example the policy will check either the header field "DroolsLookup" for the named session to execute and if that isn't specified it'll check the "lookup" attribute on the incoming payload. It then attempts to "lookup" the session from the execution-node context and execute against it.

Example 1.2. Drools EndPoint configured with the CXFRS producer

```
<bean id="droolsPolicy" class="org.drools.camel.component.DroolsPolicy" />

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="cxfrs://bean://rsServer"/>
      <policy ref="droolsPolicy">
        <unmarshal ref="xstream" />
        <to uri="drools:nodel" />
        <marshal ref="xstream" />
      </policy>
    </route>
  </camelContext>
```

Example 1.3. Java Code to execute against Route from a Spring and Camel Context

```
public class MyTest extends CamelSpringTestSupport {

    @Override
    protected AbstractXmlApplicationContext createApplicationContext() {
        return new ClassPathXmlApplicationContext("org/drools/camel/component/
CxfrsSpring.xml");
    }

    public void test1() throws Exception {
        String cmd = "";
        cmd += "<batch-execution lookup=\"ksession1\">\n";
        cmd += "  <insert out-identifier=\"salaboy\">\n";
        cmd += "    <org.drools.pipeline.camel.Person>\n";
        cmd += "      <name>salaboy</name>\n";
        cmd += "    </org.drools.pipeline.camel.Person>\n";
        cmd += "  </insert>\n";
        cmd += "  <fire-all-rules/>\n";
        cmd += "</batch-execution>\n";

        Object object =
this.context.createProducerTemplate().requestBody("direct://client", cmd);
        System.out.println( object );
    }
}
```

The following urls show sample script examples for jaxb, xstream and json marshalling using:

- <http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD>

- <http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD>
- <http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/xstream.mvt?r=HEAD>

Chapter 2. Spring Integration

2.1. Integration with Drools Expert

In this section we will explain the drools namespace.

2.1.1. Define a Resource

Execution nodes are a context to register sessions and kbases against for lookup.

2.1.1.1. <drools:execution-node>'s parameters as attributes:

Table 2.1.

| Attribute | Description | Required |
|-----------|--|----------|
| id | Bean's <i>id</i> is the name to be reference from other beans. | Yes |

2.1.1.2. <drools:resource>'s parameters as attributes:

Table 2.2.

| Attribute | Description | Required |
|-----------|---|--|
| id | Bean's <i>id</i> is the name to be reference from other beans. | Yes, but can be omitted when this tag is nested. |
| source | The path to the file containing the actual resource. | Yes |
| type | Type of the Resource as is defined in class <code>ResourceType</code> . | No, default to DRL |

2.1.1.3. <drools:resource>'s parameters as nested elements:

When configuring a *decision table* resource you can nest a `<drools:decisiontable-conf>` tag to configure the resource.

Table 2.3.

| Attribute | Description | Required |
|----------------|--|----------|
| input-type | Decision Table input type: XLS or CSV | No |
| worksheet-name | The worksheet name to be used in the when using an XLS | No |

Example 2.1. resource definition example

```
<drools:resource source="classpath:org/drools/IntegrationExampleTest.xls"
                type="DTABLE">
  <drools:decisiontable-conf input-type="XLS" worksheet-name="Tables_2" />
</drools:resource>
```

2.1.1.4. Define a KnowledgeBase

2.1.1.4.1. <drools:kbase>'s parameters as attributes:

Table 2.4.

| Attribute | Description | Required |
|-----------|---|----------|
| id | Bean's <i>id</i> is the name to be referenced from other beans. | Yes |
| node | Execution-Node context to register the kbase with | no |

2.1.1.4.2. <drools:kbase>'s parameters as nested elements:

A Knowledge Base has a 0..n resources. Has two nested elements, resources and configurations. Those resources can be inner elements or references, using the "ref" attribute.

Example 2.2. kbase definition example

```
<drools:kbase id="kbase1" node="node1">
  <drools:resources>
    <drools:resource source="classpath:org/drools/spring/
IntegrationExampleTest.xls"
                    type="DTABLE">
      <drools:decisiontable-conf input-type="XLS" worksheet-name="Table_2" />
    </drools:resource>

    <drools:resource ref="resource1"/>

    <drools:resource source="classpath:org/drools/container/spring/model.xsd" /
  >
</drools:resources>

  <drools:configuration>
    <drools:mbeans enabled="true" />

    <drools:accumulate-functions>
```

```
<drools:accumulate-function name="func1" ref="func1Instance" />
  <drools:accumulate-function name="func1" ref="func2Instance" />
</drools:accumulate-functions>
</drools:configuration>

</drools:kbase>
```

- advanced-process-rule-integration
 - enabled = true : false
- multithread
 - enabled = true : false
 - max-threads = 1..n
- mbeans
 - enabled = true : false
- event-processing-mode
 - mode = STREAM : CLOUD
- accumulate-functions
 - accumulate-function 0..n
 - name = String
 - ref = String
- evaluators
 - evaluator 0..n
 - name = String
 - ref = String
- assert-behavior
 - mode = IDENTITY : EQUALITY

Figure 2.1. Knowledge Base Configuration Options

2.1.1.5. Define KnowledgeSessions

2.1.1.5.1. <drools:ksession>'s parameters as attributes:

Table 2.5.

| Attribute | Description | Required |
|-----------|---|---|
| id | Bean's <i>id</i> is the name to be referenced from other beans. | Yes |
| type | is the session <i>stateful</i> or <i>stateless</i> ? | Yes |
| name | | No; defaults to <i>id</i> when omitted. |
| node | Execution-Node context to register the ksession with | no |
| listeners | Specifies the reference to the event listeners group (see 'Defining a Group of listeners' section below). | no |

Example 2.3. ksession definition example

```

<drools:ksession id="ksession1" type="stateless"
                name="stateless1" kbase="kbase1" />

<drools:ksession id="ksession2" type="stateful" kbase="kbase1" />

<drools:ksession id="ksession3" type="stateful" kbase="kbase2">

  <drools:batch>
    <drools:insert-object ref="person" />

    <drools:set-global identifier="list1">
      <bean class="java.util.ArrayList" />
    </drools:set-global>

    <drools:startProcess process-id="start fire">

  </drools:batch>

  <drools:configurations>
    <drools:keep-reference enabled="false" />
    <drools:clock-type type="PSEUDO" />
  </drools:configurations>

```

```
</drools:ksession>
```

- keep-reference
 - enabled = true : false
- clock-type
 - type = REALTIME : PSEUDO
- jpa-persistence
 - transaction-manager
 - ref = String
 - entity-manager-factory
 - ref = String
 - variable-persisters
 - for-class = String
 - implementation = String

Figure 2.2. Knowledge Session Configuration Options

- insert-object
 - ref = String (optional)
 - Anonymous bean
- set-global
 - identifier = String (required)
 - reg = String (optiona)
 - Anonymous bean
- fire-all-rules
 - max : n
- fire-until-halt
- start-process

- parameter
 - identifier = String (required)
 - ref = String (optional)
 - Anonymous bean
- signal-event
 - ref = String (optional)
 - event-type = String (required)
 - process-instance-id =n (optional)

Figure 2.3. Initialization Batch Commands

Example 2.4. ksession JPA configuraiton example

```
<drools:kstore id="kstore" /> <!-- provides KnowledgeStoreService
implementation -->

<bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="ds" />

  <property name="persistenceUnitName" value="org.drools.persistence.jpa.local" /
>
</bean>

<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="myEmf" />
</bean>

<drools:ksession id="jpaSingleSessionCommandService" type="stateful" kbase="kbProcessWorkItems"
  <drools:configuration>
    <drools:jpa-persistence>
      <drools:transaction-manager ref="txManager" />
      <drools:entity-manager-factory ref="myEmf" />
    </drools:jpa-persistence>
  </drools:configuration>
</drools:ksession>
```

2.1.1.5.2. <drools:ksession>'s parameters as nested elements:

When configuring a *Event Listener* resource you can nest a `<drools:eventListener>` tag to configure the listener.

2.1.2. Event Listeners

Drools supports adding 3 types of listeners to KnowledgeSessions - *AgendaListener*, *WorkingMemoryListener*, *ProcessEventListener*

The drools-spring module allows you to configure these listeners to KnowledgeSessions using XML tags. These tags have identical names as the actual listener interfaces i.e., `<drools:agendaEventListener....>`, `<drools:workingMemoryEventListener....>` and `<drools:processEventListener....>`.

drools-spring provides features to define the listeners as standalone (individual) listeners and also to define them as a group.

2.1.2.1. Defining Stand alone Listeners:

2.1.2.1.1. Attributes:

Table 2.6.

| Attribute | Required | Description |
|-----------|----------|---------------------------------------|
| ref | No | A reference to another declared bean. |

Example 2.5. Listener configuration example - using a bean:ref.

```

<drools:resource id="resource1" type="DRL" source="classpath:org/drools/
container/spring/testSpring.drl"/>

<drools:kbase id="kbase1">
  <drools:resources>
    <drools:resource ref="resource1"/>
  </drools:resources>
</drools:kbase>

      <bean id="mock-agenda-
listener" class="org.drools.container.spring.MockAgendaEventListener" />
      <bean id="mock-wm-
listener" class="org.drools.container.spring.MockWorkingMemoryEventListener" />
      <bean id="mock-process-
listener" class="org.drools.container.spring.MockProcessEventListener" />

<drools:ksession id="statefulSession" type="stateful" kbase="kbase1" node="node1">
  <drools:agendaEventListener ref="mock-agenda-listener"/>
  <drools:processEventListener ref="mock-process-listener"/>
  <drools:workingMemoryEventListener ref="mock-wm-listener"/>
</drools:ksession>

```

2.1.2.1.2. Nested Elements:

- bean
 - class = String
 - name = String (optional)

Example 2.6. Listener configuration example - using nested bean.

```
<drools:resource id="resource1" type="DRL" source="classpath:org/drools/
container/spring/testSpring.drl"/>

<drools:kbase id="kbase1">
  <drools:resources>
    <drools:resource ref="resource1"/>
  </drools:resources>
</drools:kbase>

<drools:ksession id="statefulWithNestedBean" type="stateful" kbase="kbase1" node="node1">

  <drools:agendaEventListener>
    <bean class="org.drools.container.spring.MockAgendaEventListener"/>
  </drools:agendaEventListener>

  <drools:workingMemoryEventListener>
    <bean class="org.drools.container.spring.MockWorkingMemoryEventListener"/>
  </drools:workingMemoryEventListener>

  <drools:processEventListener>
    <bean class="org.drools.container.spring.MockProcessEventListener"/>
  </drools:processEventListener>

</drools:ksession>
```

2.1.2.1.3. Empty Tag : Declaration with no 'ref' and without a nested bean

When a listener is defined without a reference to a implementing bean and does not contain a nested bean, `<drools:workingMemoryEventListener/>` the underlying implementation adds the Debug version of the listener defined in the API.

The debug listeners print the corresponding Event toString message to `System.err`.

Example 2.7. Listener configuration example - defaulting to the debug versions provided by the Knowledge-API .

```

<drools:resource id="resource1" type="DRL" source="classpath:org/drools/
container/spring/testSpring.drl"/>

<drools:kbase id="kbase1">
  <drools:resources>
    <drools:resource ref="resource1"/>
  </drools:resources>
</drools:kbase>

<drools:ksession id="statefulWithDefault" type="stateful" kbase="kbase1" node="node1">

    <drools:agendaEventListener/>      <!-- attaches the
org.drools.event.rule.DebugAgendaEventListener -->
    <drools:workingMemoryEventListener/>      <!-- attaches the
org.drools.event.rule.DebugWorkingMemoryEventListener -->
    <drools:processEventListener/>      <!-- attaches the
org.drools.event.DebugProcessEventListener -->

</drools:ksession>

```

2.1.2.1.4. Mix and Match of different declaration styles

The drools-spring module allows you to mix and match the different declarative styles within the same KnowledgeSession. The below sample provides more clarity.

Example 2.8. Listener configuration example - mix and match of 'ref'/nested-bean/empty styles.

```

<drools:resource id="resource1" type="DRL" source="classpath:org/drools/
container/spring/testSpring.drl"/>

<drools:kbase id="kbase1">
  <drools:resources>
    <drools:resource ref="resource1"/>
  </drools:resources>
</drools:kbase>

    <bean id="mock-agenda-
listener" class="org.drools.container.spring.MockAgendaEventListener" />

<drools:ksession id="statefulWithMixAndMatchOfStyles" type="stateful" kbase="kbase1" node="n

  <drools:workingMemoryEventListener>

```

```
<bean class="org.drools.container.spring.MockWorkingMemoryEventListener"/>
</drools:workingMemoryEventListener>

<drools:agendaEventListener ref="mock-agenda-listener"/>

        <drools:processEventListener/>        <!-- attaches the
org.drools.event.DebugProcessEventListener -->

</drools:ksession>
```

2.1.2.1.5. Defining multiple listeners of the same type

It is also valid to define multiple beans of the same event listener types for a KnowledgeSession.

Example 2.9. Listener configuration example - multiple listeners of the same type.

```
<drools:resource id="resource1" type="DRL" source="classpath:org/drools/
container/spring/testSpring.drl"/>

<drools:kbase id="kbase1">
  <drools:resources>
    <drools:resource ref="resource1"/>
  </drools:resources>
</drools:kbase>

        <bean                                id="mock-wm-
listener" class="org.drools.container.spring.MockWorkingMemoryEventListener" />

<drools:ksession id="statefulWithMultipleSameType" type="stateful" kbase="kbase1" node="node1">

  <!-- 2 different implementations of the same listener attached to one
ksession -->
  <drools:workingMemoryEventListener>

<bean class="org.drools.container.spring.ConsoleWorkingMemoryEventListener"/>
  </drools:workingMemoryEventListener>

  <drools:workingMemoryEventListener ref="mock-wm-listener"/>

</drools:ksession>
```

2.1.2.2. Defining a Group of listeners:

drools-spring allows for grouping of listeners. This is particularly useful when you define a set of listeners and want to attach them to multiple sessions. The grouping feature is also very useful, when we define a set of listeners for 'testing' and then want to switch them for 'production' use.

2.1.2.2.1. Attributes:

Table 2.7.

| Attribute | Required | Description |
|-----------|----------|-------------------|
| ID | yes | Unique identifier |

2.1.2.2.2. Nested Elements:

- drools:agendaEventListener...
- drools:workingMemoryEventListener...
- drools:processEventListener...



Note

The above mentioned child elements can be declared in any order. Only one declaration of each type is allowed in a group.

2.1.2.2.3. Example:

Example 2.10. Group of listeners - example

```

<drools:resource id="resource1" type="DRL" source="classpath:org/drools/
container/spring/testSpring.drl"/>

<drools:kbase id="kbase1">
  <drools:resources>
    <drools:resource ref="resource1"/>
  </drools:resources>
</drools:kbase>

                                <bean id="mock-wm-
listener" class="org.drools.container.spring.MockWorkingMemoryEventListener" />
<drools:kbase id="statelessWithGroupedListeners" type="stateless" kbase="kbase1" node="listeners" mockListeners="/
>

```

```
<drools:eventListeners id="mockListeners">
  <drools:agendaEventListener ref="mock-agenda-listener"/>
  <drools:processEventListener ref="mock-process-listener"/>
  <drools:workingMemoryEventListener ref="mock-wm-listener"/>
</drools:eventListeners>

<drools:eventListeners id="debugListeners">
  <drools:agendaEventListener/>
  <drools:processEventListener/>
  <drools:workingMemoryEventListener/>
</drools:eventListeners>
```

Chapter 3. Drools Server

3.1. Introduction

The drools execution server (drools-server) module is a war which you can deploy to execute KnowledgeBases remotely for any sort of client application. This is not limited to JVM application clients, but any technology that can use HTTP, through a REST interface. This version of the execution server supports stateless and statefull sessions in a native way.

3.2. Deployment

Drools Server is a war file, which can be deployed in a application server (such as JBoss AS). As the service is stateless, it is possible to have as many of these services deployed as you need to serve the client load. Deploy on JBoss AS 4.x / Tomcat 6.x works out-of-the-box, instead some external dependencies must be added and the configuration must be changed to be deployed in JBoss AS 5

3.3. Configuration

Inside the war file you will find a few XML configuration files.

- breans.xml
 - Skeleton xml that imports knowledge-services.xml and camel-server.xml
- camel-server.xml
 - Configures CXF endpoints with Camel Routes
 - Came Routes pipeline messages to various configured knowledge services
- knowledge-services.xml
 - Various Knowege Bases and Sessions
- camel-client.xml
 - Sample camel client showing how to send and receive a message
 - Used by "out of the box" test.jsp

3.3.1. REST/Camel Services configuration

The next step is configure the services that are going to be exposed through drools-server. You can modify this configuration in camel-server.xml file.

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

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cxf="http://camel.apache.org/schema/cxf"
  xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring-beans-2.5.xsd
    http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-
cxf.xsd
    http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/
camel-spring.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml" />
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

<!--
!   If you are running on JBoss you will need to copy a camel-jboss.jar into
the lib and set this classloader configuration
!   http://camel.apache.org/camel-jboss.html
!
!                                     <bean           id="jbossResolver"
class="org.apache.camel.jboss.JBossPackageScanClassResolver"/>
-->

<!--
!   Define the server end point.
!   Copy and paste this element, changing id and the address, to expose
services on different urls.
!   Different Camel routes can handle different end point paths.
-->
<cxf:rsServer id="rsServer"
  address="/kservice/rest"
  serviceClass="org.drools.jax.rs.CommandExecutorImpl">
  <cxf:providers>
    <bean class="org.drools.jax.rs.CommandMessageBodyReader"/>
  </cxf:providers>
</cxf:rsServer>

<!-- Leave this, as it's needed to make Camel "drools" aware -->
<bean id="droolsPolicy" class="org.drools.camel.component.DroolsPolicy" />

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <!--
!   Routes incoming messages from end point id="rsServer".
!   Example route unmarshals the messages with xstream and executes against
ksession1.
!   Copy and paste this element, changingmarshallers and the 'to' uri, to
target different sessions, as needed.
-->
```



```

!-->

<route>
  <from uri="cxfrs://bean://rsServer"/>
    <policy ref="droolsPolicy">
      <unmarshal ref="xstream" />
      <to uri="drools:nodel/ksession1" />
      <marshal ref="xstream" />
    </policy>
  </route>

</camelContext>

</beans>

```

3.3.1.1. RESTful service endpoint creation

In the next xml snippet code we are creating a RESTful (JAX-RS) endpoint binded to /kservice/rest address and using org.drools.jax.rs.CommandExecutorImpl as the service implementor. This class is only used to instantiate the service endpoint because all the internal implementation is managed by Camel, and you can see in the source file that the exposed execute service must be never called.

Also a JAX-RS Provider is provided to determine if the message transported can be processed in this service endpoint.

```

<cxfrs:rsServer id="rsServer"
  address="/kservice/rest"
  serviceClass="org.drools.jax.rs.CommandExecutorImpl">
  <cxfrs:providers>
    <bean class="org.drools.jax.rs.CommandMessageBodyReader"/>
  </cxfrs:providers>
</cxfrs:rsServer>

```

Ideally this configuration doesn't need to be modified, at least the Service Class and the JAX-RS Provider, but you can add more endpoints associated to different addresses to use them in another Camel Routes.

After all this initial configuration, you can start config your own Knowledge Services.

3.3.1.2. Camel Drools Policy & Context creation

DroolsPolicy is used to add Drools support in Camel, basically what it does is to add interceptors into the camel route to create Camel Processors on the fly and modify the internal navigation route. If you want to have SOAP support you need to create your custom Drools Policy, but it's going to be added in the next release.

But you don't need to know more internal details, only instantiate this bean:

```
<bean id="droolsPolicy" class="org.drools.camel.component.DroolsPolicy" />
```

The next is create the camel route that will have the responsibility to execute the commands sent through JAX-RS. Basically we create a route definition associated with the JAX-RS definition as the data input, the camel policy to be used and inside the "execution route" or ProcessorDefinitions. As you can see, we set XStream as the marshaller/unmarshaller and the drools execution route definition

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="cxfrs://bean://rsServer" />
    <policy ref="droolsPolicy">
      <unmarshal ref="xstream" />
      <to uri="drools:nodel/ksession1" />
      <marshal ref="xstream" />
    </policy>
  </route>
</camelContext>
```

The drools endpoint creation has the next arguments

```
<to uri="drools:{1}/{2}" />
```

1. Execution Node identifier that is registered in the CamelContext
2. Knowledge Session identifier that was registered in the Execution Node with identifier {1}

Both parameters are configured in knowledge-services.xml file.

3.3.1.3. Knowledge Services configuration

The next step is create the Knowledge Sessions that you are going to use.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:drools="http://drools.org/schema/drools-spring"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://drools.org/schema/drools-spring http://drools.org/schema/drools-spring.xsd">
```

```

<drools:execution-node id="node1" />

<drools:kbase id="kbase1" node="node1">
  <drools:resources>
    <drools:resource type="XSD" source="classpath:model.xsd"/>
    <drools:resource type="DRL" source="classpath:test.drl"/>
  </drools:resources>
</drools:kbase>

<drools:ksession id="ksession1" type="stateless" kbase="kbase1" node="node1"/>
</beans>

```

The execution-node is a context or registered kbases and ksessions, here kbase1 and ksession1 are planned in the node1 context. The kbase itself consists of two knowledge definitions, a drl and an xsd. The Spring documentation contains a lot more information on configuring these knowledge services.

3.3.1.4. Test

With drools-server war unzipped you should be able to see a test.jsp and run it. This example just executes a simple "echo" type application. It sends a message to the rule server that pre-appends the word "echo" to the front and sends it back. By default the message is "Hello World", different messages can be passed using the url parameter msg - test.jsp?msg="My Custom Message".

Under the hood the jsp invokes the Test.java class, this then calls out to Camel which is where the meet happens. The camel-client.xml defines the client with just a few lines of xml:

```

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct://kservice"/>
    <policy ref="droolsPolicy">
      <to uri="cxfrs://http://localhost:8080/drools-server-app/kservice/
rest"/>
    </policy>
  </route>
</camelContext>

```

"direct://kservice" is just a named hook, allowing java to grab a reference and push data into it. In this example the data is already in xml, so we don't need to add any DataFormat's to do the marshalling. The DroolsPolicy adds some smarts to the route and you'll see it used on the server side too. If JAXB or XStream were used, it would inject custom paths and converters, it can also set the classloader too on the server side, on the client side it automatically unwraps the Response object.

The rule itself can be found here: test.drl. Notice the type Message is declared part of the drl and is thus not present on the Classpath.

```
declare Message
    text : String
end

rule "echo" dialect "mvel"
when
    $m : Message();
then
    $m.text = "echo:" + $m.text;
end
```

Chapter 4. Drools Commands

4.1. API

XML marshalling/unmarshalling of the Drools Commands requires the use of special classes, which are going to be described in the following sections.

The following urls show sample script examples for jaxb, xstream and json marshalling using:

- <http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD>
- <http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD>
- <http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/xstream.mvt?r=HEAD>

4.1.1. XStream

To use the XStream commands marshaller you need to use the DroolsHelperProvider to obtain an XStream instance. We need to use this because it has the commands converters registered.

- Marshalling

```
BatchExecutionHelperProviderImpl.newXStreamMarshaller().toXML(command);
```

- Unmarshalling

```
BatchExecutionHelperProviderImpl.newXStreamMarshaller().fromXML(xml)
```

4.1.2. JSON

JSON API to marshalling/unmarshalling is similar to XStream API:

- Marshalling

```
BatchExecutionHelper.newJsonMarshaller().toXML(command);
```

- Unmarshalling

```
BatchExecutionHelper.newJsonMarshaller().fromXML(xml)
```

4.1.3. JAXB

There are two options for using JAXB, you can define your model in an XSD file or you can have a POJO model. In both cases you have to declare your model inside JAXBContext, and in order

to do that you need to use Drools Helper classes. Once you have the JAXBContext you need to create the Unmarshaller/Marshaller as needed.

4.1.3.1. Using an XSD file to define the model

With your model defined in a XSD file you need to have a KnowledgeBase that has your XSD model added as a resource.

To do this, the XSD file must be added as a XSD ResourceType into the KnowledgeBuilder. Finally you can create the JAXBContext using the KnowledgeBase created with the KnowledgeBuilder

```
Options xjcOpts = new Options();
xjcOpts.setSchemaLanguage(Language.XMLSCHEMA);
JaxbConfiguration jaxbConfiguration = KnowledgeBuilderFactory.newJaxbConfiguration( xjcOpts, "x
kbuilder.add(ResourceFactory.newClassPathResource("person.xsd", getClass()), ResourceType.XSD,
KnowledgeBase kbase = kbuilder.newKnowledgeBase();

List<String> className = new ArrayList<String>();
className.add("org.drools.test.Person");

JAXBContext jaxbContext = KnowledgeBuilderHelper.newJAXBContext(className.toArray(new String
```

4.1.3.2. Using a POJO model

In this case you need to use DroolsJaxbContextHelper to create the JAXBContext. This class has two parameters:

1. `classNames`: A List with the canonical name of the classes that you want to use in the marshalling/unmarshalling process.
2. `properties`: JAXB custom properties

```
List<String> classNames = new ArrayList<String>();
classNames.add("org.drools.test.Person");
JAXBContext jaxbContext = DroolsJaxbContextHelper.createDroolsJaxbContext(classNames, null);
Marshaller marshaller = jaxbContext.createMarshaller();
```

4.2. Commands supported

Currently, the following commands are supported:

- BatchExecutionCommand
- InsertObjectCommand
- RetractCommand
- ModifyCommand
- GetObjectCommand
- InsertElementsCommand
- FireAllRulesCommand
- StartProcessCommand
- SignalEventCommand
- CompleteWorkItemCommand
- AbortWorkItemCommand
- QueryCommand
- SetGlobalCommand
- GetGlobalCommand
- GetObjectsCommand



Note

In the next snippets code we are going to use a POJO `org.drools.test.Person` that has two fields

- name: String
- age: Integer



Note

In the next examples, to marshall the commands we have used the nexts snippet codes:

- XStream

```
String xml = BatchExecutionHelper.newXStreamMarshaller().toXML(command);
```

- JSON

```
String xml = BatchExecutionHelper.newJSONMarshaller().toXML(command);
```

- JAXB

```
Marshaller marshaller = jaxbContext.createMarshaller();  
StringWriter xml = new StringWriter();  
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);  
marshaller.marshal(command, xml);
```

4.2.1. BatchExecutionCommand

- Description: The command that contains a list of commands, which will be sent and executed.
- Attributes

Table 4.1. BatchExecutionCommand attributes

| Name | Description | required |
|----------|--|----------|
| lookup | Sets the knowledge session id on which the commands are going to be executed | true |
| commands | List of commands to be executed | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();  
command.setLookup("ksession1");  
InsertObjectCommand insertObjectCommand = new InsertObjectCommand(new Person("john", 25));  
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();  
command.getCommands().add(insertObjectCommand);  
command.getCommands().add(fireAllRulesCommand);
```

- XML output
 - XStream

```
<batch-execution lookup="ksession1">
```



```

<insert>
  <org.drools.test.Person>
    <name>john</name>
    <age>25</age>
  </org.drools.test.Person>
</insert>
<fire-all-rules/>
</batch-execution>

```

- JSON

```

{"batch-execution":{"lookup":"ksession1","commands":[{"insert":{"object":{"org.drools.test.Person":{"name":"john","age":25}}},"fire-all-rules":""}]}}

```

- JAXB

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <insert>
    <object xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <age>25</age>
      <name>john</name>
    </object>
  </insert>
  <fire-all-rules max="-1"/>
</batch-execution>

```

4.2.2. InsertObjectCommand

- Description: Insert an object in the knowledge session.
- Attributes

Table 4.2. InsertObjectCommand attributes

| Name | Description | required |
|--------|---------------------------|----------|
| object | The object to be inserted | true |

| Name | Description | required |
|---------------|---|----------|
| outIdentifier | Id to identify the FactHandle created in the object insertion and added to the execution results | false |
| returnObject | Boolean to establish if the object must be returned in the execution results. Default value: true | false |
| entryPoint | Entrypoint for the insertion | false |

- Command creation

```
List<Command> cmds = ArrayList<Command>();

Command insertObjectCommand = CommandFactory.newInsert(new Person("john", 25), "john", false,
cmds.add( insertObjectCommand );

BatchExecutionCommand command = CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <insert out-identifier="john" entry-point="my stream" return-object="false">
    <org.drools.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.test.Person>
  </insert>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"insert":{"entry-point":"my stream", "out-identifier":"john","return-object":false,"object":{"org.drools.test.Person":{"name":"john","age":25}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <insert out-identifier="john" entry-point="my stream" >
    <object xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
      <age>25</age>
      <name>john</name>
    </object>
  </insert>
</batch-execution>
```

4.2.3. RetractCommand

- Description: Retract an object from the knowledge session.
- Attributes

Table 4.3. RetractCommand attributes

| Name | Description | required |
|--------|---|----------|
| handle | The FactHandle associated to the object to be retracted | true |

- Command creation: we have two options, with the same output result:

1. Create the Fact Handle from a string

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand();
retractCommand.setFactHandleFromString("123:234:345:456:567");
command.getCommands().add(retractCommand);
```

2. Set the Fact Handle that you received when the object was inserted

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand(factHandle);
```

```
command.getCommands().add(retractCommand);
```

- XML output
- XStream

```
<batch-execution lookup="ksession1">
  <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"retract":{"fact-handle":"0:234:345:456:567"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

4.2.4. ModifyCommand

- Description: Allows you to modify a previously inserted object in the knowledge session.
- Attributes

Table 4.4. ModifyCommand attributes

| Name | Description | required |
|---------|---|----------|
| handle | The FactHandle associated to the object to be retracted | true |
| setters | List of setters object's modifications | true |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
```

```
command.setLookup("ksession1");
ModifyCommand modifyCommand = new ModifyCommand();
modifyCommand.setFactHandleFromString("123:234:345:456:567");
List<Setter> setters = new ArrayList<Setter>();
setters.add(new SetterImpl("age", "30"));
modifyCommand.setSetters(setters);
command.getCommands().add(modifyCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <modify fact-handle="0:234:345:456:567">
    <set accessor="age" value="30"/>
  </modify>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"modify":{"fact-handle":"0:234:345:456:567","setters":{"accessor":"age","value":30}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <modify fact-handle="0:234:345:456:567">
    <set value="30" accessor="age"/>
  </modify>
</batch-execution>
```

4.2.5. GetObjectCommand

- Description: Used to get an object from a knowledge session
- Attributes

Table 4.5. GetObjectCommand attributes

| Name | Description | required |
|---------------|--|----------|
| factHandle | The FactHandle associated to the object to be retracted | true |
| outIdentifier | Id to identify the FactHandle created in the object insertion and added to the execution results | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectCommand getObjectCommand = new GetObjectCommand();
getObjectCommand.setFactHandleFromString("123:234:345:456:567");
getObjectCommand.setOutIdentifier("john");
command.getCommands().add(getObjectCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <get-object fact-handle="0:234:345:456:567" out-identifier="john"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-object":{"fact-handle":"0:234:345:456:567","out-identifier":"john"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <get-object out-identifier="john" fact-handle="0:234:345:456:567"/>
</batch-execution>
```

4.2.6. InsertElementsCommand

- Description: Used to insert a list of objects.
- Attributes

Table 4.6. InsertElementsCommand attributes

| Name | Description | required |
|---------------|---|----------|
| objects | The list of objects to be inserted on the knowledge session | true |
| outIdentifier | Id to identify the FactHandle created in the object insertion and added to the execution results | false |
| returnObject | Boolean to establish if the object must be returned in the execution results. Default value: true | false |
| entryPoint | Entrypoint for the insertion | false |

- Command creation

```
List<Command> cmds = ArrayList<Command>();

List<Object> objects = new ArrayList<Object>();
objects.add(new Person("john", 25));
objects.add(new Person("sarah", 35));

Command insertElementsCommand = CommandFactory.newInsertElements( objects );
cmds.add( insertElementsCommand );

BatchExecutionCommand command = CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
 - XStream

```
<batch-execution lookup="ksession1">
```

```
<insert-elements>
  <org.drools.test.Person>
    <name>john</name>
    <age>25</age>
  </org.drools.test.Person>
  <org.drools.test.Person>
    <name>sarah</name>
    <age>35</age>
  </org.drools.test.Person>
</insert-elements>
</batch-execution>
```

- JSON

```
{ "batch-execution": { "lookup": "ksession1", "commands": { "insert-elements":
  { "objects": [ { "containedObject":
    { "@class": "org.drools.test.Person", "name": "john", "age": 25 } },
    { "containedObject":
      { "@class": "org.drools.test.Person", "name": "sarah", "age": 35 } } ] } } }
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <insert-elements return-objects="true">
    <objects xsi:type="person" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
      <age>25</age>
      <name>john</name>
    </objects>
    <objects xsi:type="person" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
      <age>35</age>
      <name>sarah</name>
    </objects>
  </insert-elements>
</batch-execution>
```

4.2.7. FireAllRulesCommand

- Description: Allow execution of the rules activations created.

- Attributes

Table 4.7. FireAllRulesCommand attributes

| Name | Description | required |
|---------------|---|----------|
| max | The max number of rules activations to be executed. default is -1 and will not put any restriction on execution | false |
| outIdentifier | Add the number of rules activations fired on the execution results | false |
| agendaFilter | Allow the rules execution using an Agenda Filter | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
fireAllRulesCommand.setMax(10);
fireAllRulesCommand.setOutIdentifier("firedActivations");
command.getCommands().add(fireAllRulesCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <fire-all-rules max="10" out-identifier="firedActivations"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"fire-all-rules":
{"max":10,"out-identifier":"firedActivations"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <fire-all-rules out-identifier="firedActivations" max="10"/>
</batch-execution>
```

4.2.8. StartProcessCommand

- Description: Allows you to start a process using the ID. Also you can pass parameters and initial data to be inserted.
- Attributes

Table 4.8. StartProcessCommand attributes

| Name | Description | required |
|------------|--|----------|
| processId | The ID of the process to be started | true |
| parameters | A Map<String, Object> to pass parameters in the process startup | false |
| data | A list of objects to be inserted in the knowledge session before the process startup | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
StartProcessCommand startProcessCommand = new StartProcessCommand();
startProcessCommand.setProcessId("org.drools.task.processOne");
command.getCommands().add(startProcessCommand);
```

- XML output
- XStream

```
<batch-execution lookup="ksession1">
  <start-process processId="org.drools.task.processOne"/>
</batch-execution>
```

- JSON

```
{ "batch-execution": { "lookup": "ksession1", "commands": { "start-process": { "process-id": "org.drools.task.processOne" } } } }
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <start-process processId="org.drools.task.processOne">
    <parameter/>
  </start-process>
</batch-execution>
```

4.2.9. SignalEventCommand

- Description: Send a signal event.
- Attributes

Table 4.9. SignalEventCommand attributes

| Name | Description | required |
|-------------------|-------------|----------|
| event-type | | true |
| processInstanceId | | false |
| event | | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SignalEventCommand signalEventCommand = new SignalEventCommand();
signalEventCommand.setProcessInstanceId(1001);
signalEventCommand.setEventType("start");
signalEventCommand.setEvent(new Person("john", 25));
command.getCommands().add(signalEventCommand);
```

- XML output
 - XStream

```
<batch-execution lookup="ksession1">
  <signal-event process-instance-id="1001" event-type="start">
    <org.drools.pipeline.camel.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.pipeline.camel.Person>
  </signal-event>
</batch-execution>
```

- JSON

```
{ "batch-execution": { "lookup": "ksession1", "commands": { "signal-event":
{ "process-instance-id": 1001, "@event-type": "start", "event-
type": "start", "object": { "org.drools.pipeline.camel.Person":
{ "name": "john", "age": 25 } } } } } }
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <signal-event event-type="start" process-instance-id="1001">
    <event xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
      <age>25</age>
      <name>john</name>
    </event>
  </signal-event>
</batch-execution>
```

4.2.10. CompleteWorkItemCommand

- Description: Allows you to complete a WorkItem.
- Attributes

Table 4.10. CompleteWorkItemCommand attributes

| Name | Description | required |
|------------|--|----------|
| workItemId | The ID of the WorkItem to be completed | true |

| Name | Description | required |
|---------|-------------|----------|
| results | | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
CompleteWorkItemCommand completeWorkItemCommand = new CompleteWorkItemCommand();
completeWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(completeWorkItemCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <complete-work-item id="1001"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"complete-work-item":
{"id":1001}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <complete-work-item id="1001"/>
</batch-execution>
```

4.2.11. AbortWorkItemCommand

- Description: Allows you abort an WorkItem. The same as `session.getWorkItemManager().abortWorkItem(workItemId)`
- Attributes

Table 4.11. AbortWorkItemCommand attributes

| Name | Description | required |
|------------|--|----------|
| workItemId | The ID of the WorkItem to be completed | true |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
AbortWorkItemCommand abortWorkItemCommand = new AbortWorkItemCommand();
abortWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(abortWorkItemCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <abort-work-item id="1001"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"abort-work-item":
{"id":1001}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <abort-work-item id="1001"/>
</batch-execution>
```

4.2.12. QueryCommand

- Description: Executes a query defined in knowledge base.

- Attributes

Table 4.12. QueryCommand attributes

| Name | Description | required |
|---------------|--|----------|
| name | The query name | true |
| outIdentifier | The identifier of the query results. The query results are going to be added in the execution results with this identifier | false |
| arguments | A list of objects to be passed as a query parameter | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
QueryCommand queryCommand = new QueryCommand();
queryCommand.setName("persons");
queryCommand.setOutIdentifier("persons");
command.getCommands().add(queryCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <query out-identifier="persons" name="persons" />
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"query":{"out-identifier":"persons","name":"persons"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <query name="persons" out-identifier="persons" />
</batch-execution>
```

```
</batch-execution>
```

4.2.13. SetGlobalCommand

- Description: Allows you to set a global.
- Attributes

Table 4.13. SetGlobalCommand attributes

| Name | Description | required |
|---------------|--|----------|
| identifier | The identifier of the global defined in the knowledge base | true |
| object | The object to be setted into the global | false |
| out | A boolean to add, or not, the set global result into the execution results | false |
| outIdentifier | The identifier of the global execution result | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SetGlobalCommand setGlobalCommand = new SetGlobalCommand();
setGlobalCommand.setIdentifier("helper");
setGlobalCommand.setObject(new Person("kyle", 30));
setGlobalCommand.setOut(true);
setGlobalCommand.setOutIdentifier("output");
command.getCommands().add(setGlobalCommand);
```

- XML output
- XStream

```
<batch-execution lookup="ksession1">
  <set-global identifier="helper" out-identifier="output">
    <org.drools.test.Person>
      <name>kyle</name>
      <age>30</age>
    </org.drools.test.Person>
  </set-global>
</batch-execution>
```



```

    </org.drools.test.Person>
  </set-global>
</batch-execution>

```

- JSON

```

{"batch-execution":{"lookup":"ksession1","commands":{"set-global":{"identifier":"helper","out-identifier":"output","object":{"org.drools.test.Person":{"name":"kyle","age":30}}}}}

```

- JAXB

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <set-global out="true" out-identifier="output" identifier="helper">
    <object xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
      <age>30</age>
      <name>kyle</name>
    </object>
  </set-global>
</batch-execution>

```

4.2.14. GetGlobalCommand

- Description: Allows you to get a global previously defined.
- Attributes

Table 4.14. GetGlobalCommand attributes

| Name | Description | required |
|---------------|--|----------|
| identifier | The identifier of the global defined in the knowledge base | true |
| outIdentifier | The identifier to be used in the execution results | false |

- Command creation

```

BatchExecutionCommand command = new BatchExecutionCommand();

```

```
command.setLookup("ksession1");
GetGlobalCommand getGlobalCommand = new GetGlobalCommand();
getGlobalCommand.setIdentifier("helper");
getGlobalCommand.setOutIdentifier("helperOutput");
command.getCommands().add(getGlobalCommand);
```

- XML output
 - XStream

```
<batch-execution lookup="ksession1">
  <get-global identifier="helper" out-identifier="helperOutput"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-global":
{"identifier":"helper","out-identifier":"helperOutput"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <get-global out-identifier="helperOutput" identifier="helper"/>
</batch-execution>
```

4.2.15. GetObjectsCommand

- Description: Returns all the objects from the current session as a Collection.
- Attributes

Table 4.15. GetObjectsCommand attributes

| Name | Description | required |
|--------------|---|----------|
| objectFilter | An ObjectFilter to filter the objects returned from the current session | false |

| Name | Description | required |
|---------------|--|----------|
| outIdentifier | The identifier to be used in the execution results | false |

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectsCommand getObjectsCommand = new GetObjectsCommand();
getObjectsCommand.setOutIdentifier("objects");
command.getCommands().add(getObjectsCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <get-objects out-identifier="objects"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-objects":{"out-identifier":"objects"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
  <get-objects out-identifier="objects"/>
</batch-execution>
```


Chapter 5. OSGi Integration

OSGi is a dynamic module system for declarative services. So what does that mean? Each jar in OSGi is called a bundle and has its own Classloader. Each bundle specifies the packages it exports (makes publicly available) and which packages it imports (external dependencies). OSGi will use this information to wire the classloaders of different bundles together; the key distinction is you don't specify what bundle you depend on, or have a single monolithic classpath, instead you specify your package import and version and OSGi attempts to satisfy this from available bundles.

It also supports side by side versioning, so you can have multiple versions of a bundle installed and it'll wire up the correct one. Further to this Bundles can register services for other bundles to use. These services need initialisation, which can cause ordering problems - how do you make sure you don't consume a service before its registered? OSGi has a number of features to help with service composition and ordering. The two main ones are the programmatic ServiceTracker and the xml based Declarative Services. There are also other projects that help with this; Spring DM, iPOJO, Gravity.

Each of the Drools factories is now also available as a FactoryService interface. You can either have OSGi inject those into a pojo, or retrieve them yourself from OSGi.

The following modules should work with OSGi;

- knowledge-api
- drools-core
- drools-compiler
- drools-templates
- drools-decisiontables

The following Services can be located as OSGi Bundles

- KnowledgeBuilderFactoryService
- KnowledgeBaseFactoryService
- ResourceFactoryService

I'll cover injection here. The below example injects the KnowledgeBuilderFactoryService, KnowledgeBaseFactoryService and ResourceFactoryService into the TestComponent pojo.

Example 5.1. Osgi Declarative Services

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
  <implementation class="testosgi.TestComponent"/>

  <reference bind="setKnowledgeBaseFactoryService"
```

```
        unbind="unsetKnowledgeBaseFactoryService"
        interface="org.drools.KnowledgeBaseFactoryService"
    />

<reference bind="setResourceFactoryService"
        unbind="unsetResourceFactoryService"
        interface="org.drools.io.ResourceFactoryService"
    />

<reference bind="setKnowledgeBuilderFactoryService"
        unbind="unsetKnowledgeBuilderFactoryService"
        interface="org.drools.builder.KnowledgeBuilderFactoryService"
        target="(org.drools.compiler.DecisionTableProvider=true)" />
</scr:component>
```

The TestComponent will only be activated when all of the referenced services are available and injected into the pojo. You'll also notice the "target" attribute for the KnowledgeBuilderFactoryService. The reason for this is that OSGi DS has no built in way to declaratively say which optional services must be present to satisfy your component. As a work around I made any Drools service that has optional services set a property if/when the optional service is available. Filters can then be applied, via the target attribute, to make sure the Service is in a desired state before consuming it. And that is pretty much it :)

Example 5.2. Basic Rule Compilation

```
ServiceReference          serviceRef          =
    bundleContext.getServiceReference( ServiceRegistry.class.getName() );
ServiceRegistry          registry            =
    bundleContext.getService( serviceRef );

KnowledgeBuilderFactoryService    knowledgeBuilderFactoryService    =
    registry.get( KnowledgeBuilderFactoryService.class );

KnowledgeBaseFactoryService    knowledgeBaseFactoryService    =
    registry.get( KnowledgeBaseFactoryService.class );
ResourceFactoryService    resourceFactoryService    =
    registry.get( ResourceFactoryService.class );

KnowledgeBuilderConfiguration    kbConf          =
    knowledgeBuilderFactoryService.newKnowledgeBuilderConfiguration( null,
                                                                    getClass().getClassLoader() );

KnowledgeBuilder    kbuilder          =
    knowledgeBuilderFactoryService.newKnowledgeBuilder( kbConf );
ResourceFactoryService resource = resourceFactoryService;
kbuilder.add( resource.newByteArrayResource( string.getBytes() ),
```

```

        ResourceType.DRL );

if ( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
    throw new RuntimeException( kbuilder.getErrors().toString() );
}

KnowledgeBaseConfiguration          kbaseConf          =
    knowledgeBuilderFactoryService.newKnowledgeBaseConfiguration( null,

                                getClass().getClassLoader() );

KnowledgeBase kbase = knowledgeBuilderFactoryService.newKnowledgeBase( kbaseConf );
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

```

Example 5.3. Decision Table Example

```

ServiceReference                    serviceRef          =
    bundleContext.getServiceReference( ServiceRegistry.class.getName() );
ServiceRegistry                    registry            =
    bundleContext.getService( serviceRef );

KnowledgeBuilderFactoryService      knowledgeBuilderFactoryService =
    registry.get( KnowledgeBuilderFactoryService.class );
KnowledgeBaseFactoryService        knowledgeBaseFactoryService =
    registry.get( KnowledgeBaseFactoryService.class );
ResourceFactoryService             resourceFactoryService =
    registry.get( ResourceFactoryService.class );

KnowledgeBaseConfiguration          kbaseConf          =
    knowledgeBuilderFactoryService.newKnowledgeBaseConfiguration( null,

                                getClass().getClassLoader() );

KnowledgeBuilderConfiguration      kbConf             =
    knowledgeBuilderFactoryService.newKnowledgeBuilderConfiguration( null,

                                getClass().getClassLoader() );

KnowledgeBuilder                   kbuilder           =
    knowledgeBuilderFactoryService.newKnowledgeBuilder( kbConf );
kbuilder.add( resourceFactoryService.newClassPathResource( "changeset1Test.xml",
                                                            Dummy.class ),

              ResourceType.CHANGE_SET );

kbaseConf = knowledgeBuilderFactoryService.newKnowledgeBaseConfiguration( null,

```

```
        getClass().getClassLoader() );  
KnowledgeBase kbase = knowledgeBaseFactoryService.newKnowledgeBase( kbaseConf );  
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );  
  
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

Index

