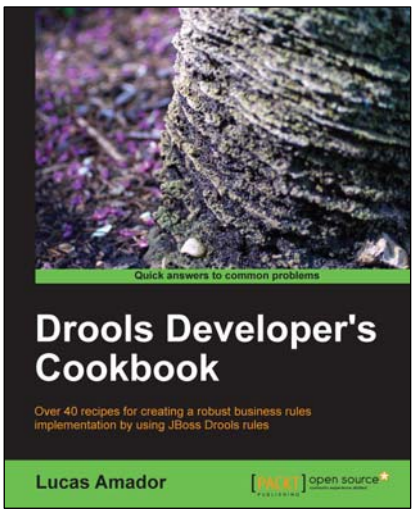
[PACKT] open source
community experience distilled
PUBLISHING

Drools Developer's Cookbook

Lucas Amador



Chapter No. 2
"Expert: Behind the Rules"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Expert: Behind the Rules"

A synopsis of the book's content

Information on where to buy this book

About the Author

Lucas Amador is a software developer born and raised in Buenos Aires, Argentina. His open source interest started when he was young but he finally got completely involved in it in 2008 while working with a JBoss partner giving consultancy and developing software using the JBoss middleware platform for Telco, some financial companies, and some other companies. At this time he obtained the Sun Java Developer and JBoss Advanced Developer certifications.

He got involved in the JBoss Drools community through the Google Summer of Code 2009 program, implementing a refactoring module for the Eclipse Drools Plugin, and since then he is a jBPM5/Drools committer spending his spare time implementing new features and fixing bugs. He actually works as a freelance developer and is always looking for something interesting to work on. You can check his daily works and new projects on his personal blog `lucazamador.wordpress.com` and his Github account `www.github.com/lucazamador`. He can also be contacted at `lucazamador@gmail.com`.

I would like to dedicate this book to my family who always support me in any decision taken in my life, always cheering me up in my new adventures.

Drools Developer's Cookbook

JBoss Drools is an open source project that is always in a continuous evolution adding more modules and features to the existing ones. This evolution is possible, thanks to the vision of the core developers and the open source community that is continuously pushing it to a new level. And since version 5, Drools has been evolved to provide a unified platform for business rules, business processing, event processing, and automated planning.

With this book you will learn new features to create a robust business rules implementation, starting with tips to write business rules and ending with frameworks integration to create a full business rules solution. The recipes included in this book cover all the Drools modules and will help you to learn how your business rules can be integrated with other frameworks to create a full solution, and will also help you discover how to use complex features such as complex event processing, remote execution, declarative services, and more.

What This Book Covers

Chapter 1, Expert: The Rule Engine, will introduce you to new features available in the 5.2.0 release and in the previous releases to improve your expertise with the main Drools module.

Chapter 2, Expert: Behind the Rules, will cover more recipes for Drools Expert, and you will learn different ways to persist the knowledge and the use of the Drools Verifier project. You will also learn how to monitor Drools using JMX, and so on.

Chapter 3, Guvnor: Centralized Knowledge Management, contains recipes that use the Drools Guvnor as your knowledge repository using new authoring methods.

Chapter 4, Guvnor: Advanced Features and Configuration, will teach you how to back up and configure the rules repository in order to use an external database, how to interact with Guvnor using the REST API, and so on.

Chapter 5, Fusion: Processing Complex Events, will get you started with the Complex Event Processing (CEP) and Drools Fusion concepts to add event processing capabilities to your project.

Chapter 6, Executing Drools Remotely, will introduce to you all the concepts and requirements needed to use and test the Drools Server module to execute your business rules remotely using the HTTP protocol, which will enable you to interact with it using different programming languages.

Chapter 7, Integration: How to Connect Drools, covers all the possible integrations between Drools, the Spring Framework, and Apache Camel to create declarative services.

Chapter 8, Planner: Optimizing Your Automated Planning, will teach you how to optimize automated planning problems using the Drools Planner module, covering a step-by-step explanation of the configuration possibilities and how to benchmark it to improve your solution.

Chapter 9, jBPM5: Managing Business Processes, will show you how to use some features of the new jBPM5 project, such as BPMN2 process creation using the API, and testing, monitoring, and report creation.

2
Expert: Behind the Rules

In this chapter, we will cover:

- Marshalling knowledge sessions
- Using persistence to store knowledge
- How to discard duplicated facts on insertion
- Using a custom classloader in knowledge agent
- Verifying the quality of rules with the Drools Verifier
- Monitoring knowledge with JMX

Introduction

Beyond knowing and using rules, it is necessary to know what other tasks can be performed with the knowledge of using the Drools API. These tasks begin with two possible ways to store knowledge, to how monitor the knowledge sessions to know their internal state and its behavior. This chapter will be focused on recipes that are not completely related to rules authoring, but will help in the integration process.

Marshalling knowledge sessions

Knowledge storage can be an important feature of your application if you want to resume the execution with the previous working memory state. This recipe will cover a simple approach to how you can store your knowledge sessions using the Drools API and how to recover them so that they can used immediately.

How to do it...

Follow the steps given here in order to complete this recipe:

1. Create your business rules, knowledge base, and knowledge session as you normally do.
2. Implement the `java.io.Serializable` interface in all your business objects, so that they can be marshalled.
3. Obtain a `Marshaller` instance using the Drools `MarshallerFactory` object, create a `java.io.File` object, and save the working memory data in a `ksession.info` file. Don't forget to close the file stream. The following code shows this:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller = MarshallerFactory
    .newMarshaller(kbase);
File file = new File("ksession.info");
FileOutputStream foStream;
foStream = new FileOutputStream(file);
marshaller.marshall(baos, ksession);
baos.writeTo(foStream);
baos.close();
```

4. At this point of the recipe the working memory data was stored in the `ksession.info` file. Now, to unmarshall the stored knowledge session it is necessary to use a `Marshaller` object that is created with the same knowledge base used to create the knowledge session, which is now going to be unmarshalled. It can be the same one used to store the knowledge session or a new instance:

```
Marshaller marshaller = MarshallerFactory
   .newMarshaller(kbase);
```

5. The last step is to invoke the `unmarshall(InputStream is)` method of the `Marshaller` object, which will return the restored `StatefulKnowledgeSession`:

```
FileInputStream fileInputStream =
    new FileInputStream("ksession.info");
StatefulKnowledgeSession ksession = marshaller
    .unmarshall(fileInputStream);
```

How it works...

The process of marshalling/unmarshalling a working memory is relatively easy because Drools provides the entire API to do it. In order to do this, it is necessary to use a `MarshallerFactory` together with the `KnowledgeBase` and the `StatefulKnowledgeSession` that is going to be stored:

Obtain an `org.drools.marshalling.Marshaller` object instance through the `newMarshaller(KnowledgeBase kbase)` static method of the `org.drools.marshalling.MarshallerFactory` class.

```
Marshaller marshaller = MarshallerFactory
    .newMarshaller(kbase);
```

The marshaller can use different strategies to know how to marshall your business objects and these strategies can be specified using the `newMarshaller(KnowledgeBase kbase, ObjectMarshallingStrategy[] strategies)` method, but the usage of strategies will be covered later in the *There's more* section of this recipe. The only thing that you should know at this moment is that the default strategy is the **SerializeMarshallingStrategy** and works using the `writeObject()` method of your business classes that implement the `Serializable` interface.

The next step is the creation of a `java.io.File` and a `FileOutputStream` object to store the `StatefulKnowledgeSession`:

```
File file = new File("ksession.info");
FileOutputStream foStream = new FileOutputStream(file);
```

Once it's done, the `StatefulKnowledgeSession` is ready to be marshalled. In this step a `ByteArrayOutputStream` is necessary to convert the `KnowledgeSession` into `bytes[]` objects and then write it into the `FileOutputStream`.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
marshaller.marshall(baos, ksession);
baos.writeTo(foStream)
baos.close();
```

After these lines of code, the current state of the `StatefulKnowledgeSession` is stored into the `ksession.info` file. Keep in mind that any working memory modifications that happen after the marshalling process aren't going to be stored in the file unless the marshalling process is executed again.

There's more...

Unmarshalling is the inverse process that will allow us to restore our knowledge session from the disk and it's important to understand how to do it to be able to save and restore our working memory data. The following sections explain how to achieve this.

Unmarshalling the knowledge session

Now, it is time to unmarshall the stored `StatefulKnowledgeSession`. As said before, it's necessary to have a `Marshaller` created with the same `KnowledgeBase` with which the `StatefulKnowledgeSession` was marshalled (for example, the `KnowledgeBase` needs to have the same rules):

```
Marshaller marshaller = MarshallerFactory
    .newMarshaller(kbase);
```

Once the marshaller is created, the `unmarshall(InputStream is)` method is used to obtain the `StatefulKnowledgeSession`. This code snippet shows how this method receives an `InputStream` object, and how the knowledge was stored into a file. We are now going to use a `FileInputStream` created with the name of the file in which the `StatefulKnowledgeSession` was stored:

```
FileInputStream fis = new FileInputStream("ksession.info");
StatefulKnowledgeSession ksession = marshaller
    .unmarshall(fis);
```

Once this method is invoked, it returns a `StatefulKnowledgeSession` that is ready to be used as you usually use it.

If you take a look into the `Marshaller` methods, you will find that there are other umarshaller methods with different parameters, for example, the `unmarshall(InputStream is, StatefulKnowledgeSession ksession)` method. This will unmarshall the `InputStream` into the `StatefulKnowledgeSession` that is passed as the second parameter, losing the current session state.

Session can be marshalled using different strategies to serialize the data in different ways. Next, you will see how to use another type of marshalling without implementing the `Serializable` interface in your domain objects.

Using marshalling strategies

Marshalling strategies allow you to specify how the session is marshalled. By default there are two implementations, `IdentityMarshallingStrategy` and `SerializeMarshallingStrategy`, but the users can implement their own strategy implementing the `ObjectMarshallingStrategy` interface (for example, one possible implementation is the creation of a `XMLMarshallingStrategy` that uses XStream to marshall/unmarshall your Plain Old Java Objects, that is, POJOs).

In the previous example, only the `SerializeMarshallingStrategy` was used, which needs the user POJOs to implement the `Serializable` interface. However, if you don't want to implement this interface, then the `IdentityMarshallingStrategy` can be your choice. In this case, you must keep in mind that you have to use the same `Marshaller` to unmarshall the knowledge session, because this strategy works by marshalling/unmarshalling a `Map` object in which only the object's ID is stored. However, it keeps the object references in another object, which isn't marshalled.

In order to use an `IdentityMarshallingStrategy`, follow the steps given below:

1. Create a `ClassFilterAcceptor` to specify the packages in which the strategy is going to be applied. Otherwise the default `*.*` pattern is going to be used and all the files will use the strategy. In this example, the classes inside the `drools.cookbook.chapter02.virtualization` do not implement `Serializable` because these classes are going to be marshalled using an `IdentityMarshallingStrategy`.

```
String[] patters = new String[]
    {"drools.cookbook.chapter02.virtualization.*" }
ObjectMarshallingStrategyAcceptor identityAcceptor =
MarshallerFactory.newClassFilterAcceptor(patterns);
```

2. After the `ClassFilterAcceptor` creation, obtain the `ObjectMarshallingStrategy` objects. Here two strategies are obtained, one using the *identity* approach together with the `ClassFileAcceptor` and the other one using the *serialize* strategy to serialize all the others packages.

```
ObjectMarshallingStrategy identityStrategy = MarshallerFactory
    .newIdentityMarshallingStrategy(identityAcceptor);
ObjectMarshallingStrategy serializeStrategy = MarshallerFactory.
newSerializeMarshallingStrategy();
```

3. The final step consists of obtaining a `Marshaller` instance, and letting the `MarshallerFactory` know that these two strategies are going to be used.

```
Marshaller = MarshallerFactory.newMarshaller(kbase,
   new Objectmarshallingstrategy[]{identityStrategy,
serializeStrategy});
```

And that is all that needs to be done to use marshalling strategies. Now you can start using the marshaller as was seen in this recipe.

Using persistence to store knowledge

Another option to store the working memory state is using the JPA persistence provided by the **drools-persistence-jpa module**, which will basically store all the inserted facts in a relational database. This recipe will cover the configuration that can only be used with business rules. However, in the next chapter, it will be explained how to configure it to be used together with business processes.

Getting ready

The only preliminary step is to add the extra dependencies needed by the project and that is going to be described in the following Apache Maven dependencies code snippet. In addition to the **drools-persistence-jpa**, which is the module that provides the JPA feature, the two more dependencies that are required are: the **Bitronix Transaction Manager**, a simple and embeddable JTA Transaction Manager, and a **JDBC** Driver, in this case, the embedded and in-memory **H2** database engine driver.

```
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-persistence-jpa</artifactId>
    <version>5.2.0.Final</version>
  </dependency>
  <dependency>
    <groupId>org.codehaus.btm</groupId>
    <artifactId>btm</artifactId>
    <version>1.3.3</version>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.2.128</version>
  </dependency>
</dependencies>
```

How to do it...

Follow the steps given here in order to complete this recipe:

1. After adding the required dependencies in the Apache Maven project, implement the `java.io.Serializable` interface in all the classes that are going to be inserted into the working memory.

2. Create a `persistence.xml` file into the `META-INF` directory of the project with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
                        http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd">
  <persistence-unit name="drools.cookbook.persistence.jpa"
      transaction-type="JTA">
   <provider>org.hibernate.ejb.HibernatePersistence</provider>
   <jta-data-source>jdbc/testDatasource</jta-data-source>
   <class>org.drools.persistence.info.SessionInfo</class>
   <properties>
     <property name="hibernate.dialect"
               value="org.hibernate.dialect.H2Dialect" />
     <property name="hibernate.max_fetch_depth" value="3" />
     <property name="hibernate.hbm2ddl.auto"
               value="create" />
     <property name="hibernate.show_sql" value="true" />
     <property name="hibernate.connection.autocommit"
               value="true" />
     <property
       name="hibernate.transaction.manager_lookup_class"
value= "org.hibernate.transaction.BTMTransactionManagerLookup"
     />
   </properties>
  </persistence-unit>
</persistence>
```

3. Create a `jndi.properties` file inside the `resources` folder with the following content:

```
java.naming.factory.initial=bitronix.tm.jndi.
BitronixInitialContextFactory
```

4. Now is the time to configure and initialize the data source. Configure the environment, and create a `StatefulKnowledgeSession` through the `JPAKnowledgeService`. Create a new Java class and add the following code inside the `main` method:

```
PoolingDataSource dataSource = new PoolingDataSource();
dataSource.setUniqueName("jdbc/testDatasource");
dataSource.setMaxPoolSize(5);
dataSource.setAllowLocalTransactions(true);

dataSource.setClassName("org.h2.jdbcx.JdbcDataSource");
dataSource.setMaxPoolSize(3);
dataSource.getDriverProperties().put("user", "sa");
dataSource.getDriverProperties().put("password", "sa");
dataSource.getDriverProperties().put("URL","jdbc:h2:mem:");

dataSource.init();

Environment env = KnowledgeBaseFactory.newEnvironment();
EntityManagerFactory emf = Persistence
.createEntityManagerFactory("drools.cookbook.persistence.jpa");

env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
env.set(EnvironmentName.TRANSACTION_MANAGER,
    TransactionManagerServices.getTransactionManager());
env.set(EnvironmentName.GLOBALS, new MapGlobalResolver());

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
    .newKnowledgeBuilder();
kbuilder.add(new ClassPathResource("rules.drl", getClass()),
    ResourceType.DRL);

if (kbuilder.hasErrors()) {
    if (kbuilder.getErrors().size() > 0) {
        for (KnowledgeBuilderError kerror : kbuilder.getErrors())
{
            System.err.println(kerror);
        }
    }
}

kbase = kbuilder.newKnowledgeBase();
ksession = JPAKnowledgeService
    .newStatefulKnowledgeSession(kbase, null, env);
```

5. Once the `StatefulKnowledgeSession` is created, you can start to insert/modify/retract facts into the working memory. But in order to use the JPA persistence, a `UserTransaction` should be started before inserting a fact into the working memory and should be committed after it:

```
UserTransaction ut = (UserTransaction) new InitialContext()
    .lookup("java:comp/UserTransaction");

ut.begin();
Server debianServer = new Server("debianServer", 4, 4096, 1024, 0);
ksession.insert(debianServer);
ksession.fireAllRules();
ut.commit();
```

How it works...

Once you have implemented the `java.io.Serializable` interface in your domain model, we have to configure the JPA persistence. In the second step, we created a `persistence.xml` file, which creates a persistence unit and configures the JPA `EntityManager` provider, and this file will be explained next in order to detail the most important parameters of this configuration file.

One of most important parameters is the `provider`, where you can specify which JPA implementation you would like to use. The recommended implementation is the one provided by Hibernate, which is currently used in the Drools JPA Persistence internal tests:

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

Another parameter is the configuration of the JNDI name of the database. Choose a name that is easy to remember to later programmatically configure the datasource.

```
<jta-data-source>jdbc/testDatasource</jta-data-source>
```

Then a class XML tag will be added to make a reference to the Drools object that will store all the information about the working memory state. The jBPM5 business processes are not going to be used in this recipe and hence, we only need to add the `SessionInfo` object, which will contain the working memory state serialized in one of its fields:

```
<class>org.drools.persistence.info.SessionInfo</class>
```

Now, it is time to configure the Hibernate properties, where the most important configurations are: the Hibernate Dialect and the Transaction Manager configuration, which will depend on your database engine (for example, H2), and the JTA Transaction Manager (for example, Bitronix).

Follow the steps given here to configure the Hibernate properties:

1. The following code snippet shows a complete Hibernate configuration using an H2 database and a Bitronix Transaction Manager:

```
<properties>
  <property name="hibernate.dialect"
            value="org.hibernate.dialect.H2Dialect" />
  <property name="hibernate.max_fetch_depth" value="3" />
  <property name="hibernate.hbm2ddl.auto" value="create" />
  <property name="hibernate.show_sql" value="true" />
  <property name="hibernate.connection.autocommit"
            value="true" />
  <property name="hibernate.transaction.manager_lookup_class"
 value="org.hibernate.transaction.BTMTransactionManagerLookup" />
</properties>
```

   At this point the `persistence.xml` file is already configured and now you can move to the next step.

   The `jndi.properties` file created in the `resources` folder will register the Bitronix Transaction Manager inside the JNDI directory, which is mostly used in plain J2SE applications running outside an application server. In this step you only have to add the following code in the `jndi.properties` file:

```
java.naming.factory.initial=bitronix.tm.jndi.
BitronixInitialContextFactory
```

2. Once all the configuration files are created, you are ready to create a `StatefulKnowledgeSession` to start interacting with. This process involves the creation of various resources such as the `PoolingDataSource`, an `Environment`, and the `StatefulKnowledgeSession` itself.
3. Create a new Java class file and add the following code inside a Java `main` method. In this example, we are using Bitronix as the transaction manager, and hence it is necessary to create the datasource using a `bitronix.tm.resource.jdbc.PoolingDataSource` object. To configure this datasource properly, it is necessary to at least configure the following properties:
   - **Unique name**: This will be the same value as the `<jta-data-source/>` `persistence.xml` property.
   - **Class name**: This is the canonical name of the database engine JDBC datasource class.
   - **Driver properties**: These are the user, password, and URL of the JDBC engine, which depend of the database engine driver.

```
PoolingDataSource dataSource = new PoolingDataSource();
dataSource.setUniqueName("jdbc/testDatasource");
```

```
dataSource.setMaxPoolSize(5);
dataSource.setAllowLocalTransactions(true);

dataSource.setClassName("org.h2.jdbcx.JdbcDataSource");
dataSource.setMaxPoolSize(3);
dataSource.getDriverProperties().put("user", "sa");
dataSource.getDriverProperties().put("password", "sa");
dataSource.getDriverProperties().put("URL", "jdbc:h2:mem:");
```

4. After this configuration, the datasource must be started and stopped once the application execution is finished:

```
datasource.init();
```

5. Once the datasource is started, it is time to create an `EntityManagerFactory` using the JPA API that will load the `persistence.xml` file with the definitions of the persistent units, and create the `EntityManagerFactory` if the `drools.cookbook.persistence.jpa` persistence unit definition is found.

```
EntiManagerFactory emf = Persistence
   .createEntityManagerFactory(
       "drools.cookbook.persistence.jpa");
```

6. After the `EntityManagerFactory` creation, it is time to configure the Drools Environment using the `newEnvironment()` static method of `KnowledgeBaseFactory` to obtain a new instance, shown in the following code snippet. The `Environment` object needs at least these two properties configured:
   - **EnvironmentName.ENTITY_MANAGER_FACTORY**: The `EntityManagerFactory` object that was created in the previous step
   - **EnvironmentName.TRANSACTION_MANAGER**: The property that holds the `TransactionManager` object created with the selected implementation

```
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set(EnvironmentName.ENTITY_MANAGER_FACTORY, emf);
env.set(EnvironmentName.TRANSACTION_MANAGER,
        TransactionManagerServices.getTransactionManager());
```

7. Once the `Environment` object is configured, it is time to create the `StatefulKnowledgeSession` using a different API; but the `KnowledgeBase` must be created using the common API. To create a JPA-enabled `StatefulKnowledgeSession`, you need to invoke the `newStatefulKnowledgeSession(args..)` static method of the `JPAKnowledgeService` class, and pass the `KnowledgeBase`, the `KnowledgeSessionConfiguration` (if required), and the `Environment` objects as the method arguments:

```
StatefulKnowledgeSession ksession = JPAKnowledgeService
    .newStatefulKnowledgeSession(kbase, null, env);
```

After these steps, the `StatefulKnowledgeSession` is ready to save the working memory state using JPA, but to do this, it is necessary to use JTA `UserTransactions` when objects are inserted/modified/retracted into the `KnowledgeSession`. If no `UserTransaction` is started or committed, then the working memory state is not going to be stored, losing all the information:

```
UserTransaction ut = (UserTransaction) new InitialContext()
    .lookup("java:comp/UserTransaction");

ut.begin();
Server debianServer = new Server("debianServer", 4, 4096, 1024, 0);
Ksession.insert(debianServer);

Ut.commit();
```

There's more...

In a production environment, it is recommended to use another database engine rather than the in-memory and embeddable H2 due to obvious reasons. In this section, we are going to see the configuration changes needed to use **MySQL** as the database engine:

1. The initial step is to add the MySQL driver dependency into the project. If you are using Apache Maven, you can add the following XML code in your Project Object Model (POM), which is also referred to as `pom.xml` file:

   ```
   <dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <version>5.0.8</version>
   </dependency>
   ```

2. Next, you will see all the necessary modifications in the previously-created files. The `hibernate.dialect` property in the `persistence.xml` file needs to be modified with the `MySQLDialect` class canonical name:

   ```
   <property name="hibernate.dialect"
       value="org.hibernate.dialect.MySQLDialect" />;
   ```

3. The last step is to change the datasource driver properties, in this case, modifying the classname property value, removing the URL property, and adding the database and `serverName` properties.

   ```
   dataSource.setClassName("com.mysql.jdbc.jdbc2.optional.
   MysqlXADataSource");

   dataSource.setMaxPoolSize(3);
   dataSource.getDriverProperties().put("user", "user");
   ```

```
dataSource.getDriverProperties().put("password", "password");
dataSource.getDriverProperties().put("databaseName",
                                     "databaseName");
dataSource.getDriverProperties().put("serverName ", "localhost");
```

After these modifications you are ready to start using MySQL as your JPA database engine.

How to discard duplicated facts on insertion

In some scenarios, you will have to discard equal objects (objects of the same type and values) when they are inserted into the working memory, to avoid data inconsistency and unnecessary activations. In this recipe, we will see how to change the insert behavior just by changing one configuration property.

How to do it...

Follow the steps given here in order to complete this recipe:

1. Override the `hashcode()` and `equals(Object obj)` methods in your business classes, as shown in the following code:

```
public class Server {

    private String name;
    private int processors;
    private int memory;
    private int diskSpace;
    private int cpuUsage;


    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof Server)) {
            return false;
        }
        Server server = (Server) obj;
        return server.name.equals(name)
               && server.processors == processors
               && server.memory == memory
               && server.diskSpace == diskSpace
```

```
                && server.virtualizations.size() ==
                   virtualizations.size();
        }

        @Override
        public int hashCode() {
            int result = 17;
            result = 31 * result + name.hashCode();
            result = 31 * result + processors;
            result = 31 * result + memory;
            result = 31 * result + diskSpace;
            return result;
        }

    }
```

2. In order to configure the default assert behavior property to equality, you have to configure it using a `KnowledgeBaseConfiguration` object, which will be also used to create `KnowledgeBase`. Create a new Java class and add the following code inside the `main` method:

```
KnowledgeBaseConfiguration kbaseConfig = KnowledgeBaseFactory
    .newKnowledgeBaseConfiguration();
kbaseConfig.setOption(AssertBehaviorOption.EQUALITY);
KnowledgeBase kbase = KnowledgeBaseFactory
    .newKnowledgeBase(kbaseConfig);
StatefulKnowledgeSession ksession = kbase
    .newStatefulKnowledgeSession();
```

3. Now that the `StatefulKnowledgeSession` is created, you can start to instantiate facts with the same data and insert them into the knowledge session.

```
Server debianServer = new Server("debianServer", 8, 8192, 2048, 0);
ksession.insert(debianServer);

Server anotherDebianServer = new Server("debianServer", 8,
                                        8192, 2048, 0);
ksession.insert(anotherDebianServer);

System.out.println(ksession.getObjects().size());
```

4. Once you execute the `main` method, you will see how the `StatefulKnowledgeSession` discards the facts with the same data using the overridden `equals()` and `hashCode(object)` methods.

How it works...

As discussed earlier, it is possible to modify the default insert behavior to discard duplicated facts. To do so, follow the steps given here:

The first step involves the overrides of the `hashcode()` and `equals(object)` methods of the objects of which you want to discard the duplications.

Once your domain objects override the `hashCode()` and `equals(object)` methods, it is time to configure the assertion behavior using a `KnowledgeBaseConfiguration` object, and this is done by setting the `AssertBehaviorOption.EQUALITY` option:

```
KnowledgeBaseConfiguration kbaseConfig = KnowledgeBaseFactory
    .newKnowledgeBaseConfiguration();
kbaseConfig.setOption(AssertBehaviorOption.EQUALITY);
KnowledgeBase kbase = KnowledgeBaseFactory.
newKnowledgeBase(kbaseConfig);
kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
```

Once the `KnowledgeBase` is configured to use the equality assert behavior, the `KnowledgeSession` is ready to discard duplicated inserted objects. Internally, the working memory uses a `HashMap` to store all the inserted facts and every new inserted object will return a new `FactHandle` object, the one that was inserted into the working memory, if it isn't already present in the `HashMap`.

By default the working memory works using the EQUALITY assertion mode. This means that the working memory uses an `IdentityHashMap` to store the inserted facts, which uses the equality operator (==) to compare the keys and doesn't use the `hashCode()` method. Using this assertion mode, new fact insertions will always return a new `FactHandle` object, but when a fact is inserted again, the original `FactHandle` object will be returned.

Using a custom classloader in a knowledge agent

Knowledge agents were created to monitor resources and dynamically rebuild the `KnowledgeBase` when these resources are modified, removed, and even new resources are added. Now, it is possible to add or modify rules with new fact types that the `KnowledgeAgent` classloader cannot recognize without ending in compilation errors when the resources are compiled. This can be done using the custom classloaders feature that will be explained in the next recipe.

How to do it...

Follow the steps given here in order to complete this recipe:

1. Create a new Change Set resource file named `change-set.xml` and add your business rules resources files. In this example, only one DRL file is declared:

```
<?xml version="1.0" encoding="UTF-8"?>
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd'>

  <add>
    <resource source =
      'classpath:drools/cookbook/chapter02/rules.drl'
      type='DRL' />
  </add>

</change-set>
```

2. Once you have configured the Change Set resource, create a new Java class file and inside the `main` method add the following code snippet. This code is used to create a custom `URLClassLoader` object using a JAR file named `model.jar`, which will contain your business model:

```
URL modelJarURL = getClass().getResource("model.jar");
URLClassLoader customURLClassloader =
  new URLClassLoader(new URL[] { modelJarURL });
```

3. Create a knowledge base using the previously created `URLClassLoader` object as shown in the following code. All this code must be included in the previously created `URLClassLoader` object:

```
KnowledgeBuilderConfiguration kbuilderConfig =
KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration(null,
    customURLClassloader);
KnowledgeBaseConfiguration kbaseConfig = KnowledgeBaseFactory
    .newKnowledgeBaseConfiguration(null,
    customURLClassloader);
KnowledgeBase kbase = KnowledgeBaseFactory
    .newKnowledgeBase(kbaseConfig);
```

4. Next, you have to create a `KnowledgeAgent` object using the previously created `KnowledgeBase` and `KnowledgeBuilderConfiguration` objects:

```
KnowledgeAgentConfiguration aconf =
    KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent("test", kbase,
                                            aconf,
                                            kbuilderConfig);
```

5. Finally, it is time to add the knowledge resources and obtain the `KnowledgeBase` updated by the knowledge agent:

```
kagent.applyChangeSet(
    new ClassPathResource("change-set.xml", getClass()));
kbase = kagent.getKnowledgeBase();
```

How it works...

In order to use a custom classloader, it is necessary to have a business model packaged in a JAR file, which can be located in the project structure or in an external directory.

In this case, a `java.net.URLClassLoader` object is used to point to resources loaded by a `java.net.URL` object, which will point to the JAR file:

```
URL modelJarURL = getClass().getResource("model.jar");
URLClassLoader customURLClassloader = new URLClassLoader(new URL[] {
modelJarURL });
```

Once the custom classloader is created, you have to configure a `KnowledgeBuilder` object using a `KnowledgeBuilderConfiguration` object, which will be instantiated with the custom classloader, which was created previously. If you are wondering why it is necessary to configure the `KnowledgeBuilder`, remember that all the resources are compiled using it and the `KnowledgeAgent` doesn't make an exception. The first parameter of the `new KnowledgeBuilderConfiguration(Properties properties, ClassLoader... classLoaders)` static method is used to configure settings as your own accumulate functions, disable MVEL strict mode, change the Java dialect compiler, and so on. The second parameter, which is the one that we are interested in, is used to pass the classloader being used to compile the rules, which can be more than one if required:

```
KnowledgeBuilderConfiguration kbuilderConfig = KnowledgeBuilderFactory
    .newKnowledgeBuilderConfiguration(null, customURLClassloader);
```

Once it is done, the next step is to create an initial and empty `KnowledgeBase` using the `KnowledgeBaseConfiguration` object that was created with the custom classloader:

```
KnowledgeBaseConfiguration kbaseConfig = KnowledgeBaseFactory
    .newKnowledgeBaseConfiguration(null, customURLClassloader);
KnowledgeBase kbase = KnowledgeBaseFactory
    .newKnowledgeBase(kbaseConfig);
```

The last step on the knowledge agent creation process is to create a `KnowledgeAgentConfiguration` object, which in this case doesn't have to be configured, and obtain a `KnowledgeAgent` instance using the `newKnowledgeAgent(String name, KnowledgeBase kbase, KnowledgeAgentConfiguration configuration, KnowledgeBuilderConfiguration builderConfiguration)` static method of the `KnowledgeAgentFactory` together with all the previously created objects:

```
KnowledgeAgentConfiguration aconf = KnowledgeAgentFactory
    .newKnowledgeAgentConfiguration();
KnowledgeAgent kagent = KnowledgeAgentFactory
    .newKnowledgeAgent("test", kbase, aconf, kbuilderConfig);
```

Once the knowledge agent is created, it is time to apply the change set and obtain the generated knowledge base:

```
kagent.applyChangeSet(new ClassPathResource("change-set.xml",
                                            getClass()));
kbase = kagent.getKnowledgeBase();
```

Finally, the knowledge agent is ready to compile the knowledge resources files using the custom classloader generated in the first step. As you may have figured out, the only restriction in this approach is that the configuration could only be passed on the creation instance, without a way to add your classloader in runtime.

Verifying the quality of rules with the Drools Verifier

This recipe will cover how to verify the quality of knowledge packages using the **Drools Verifier** project, which is also built using rules (to verify the quality of rules). Drools Verifier already comes with a full set of verification rules that can be used to test incoherence, equivalence, incompatibility patterns, and so on. It also allows you to create and add your own verification rules.

How to do it...

Follow the steps given here in order to understand how to verify the quality of your business rules:

1. Add the Drools Verifier dependency into the `pom.xml` file of your Apache Maven project:

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-verifier</artifactId>
  <version>5.2.0.Final</version>
</dependency>
```

2. Create your business rules and be ready to verify them. In this example, we are going to use really simple rules with visible problems, only for the purpose of showing which information can be collected:

```
package drools.cookbook.chapter02

import drools.cookbook.chapter02.Server
import java.util.Date
import java.util.List

rule "rule 1"
when
   $server : Server(processors==2, memory==1024)
then
   retract($server);
end

rule "rule 2"
when
   $server : Server(processors==2, processors!=2)
then
   retract($server);
end

rule "rule 3"
when
   $server : Server(processors==2, memory==1024)
then
   retract($server);
end
```

3. Create a new Java class named `RulesVerification` and add the following code. Here, we are going to configure Drools Verifier to analyze the quality of the rules:

```
package drools.cookbook.chapter02;

public class RulesVerification {

    public static void main(String[] args) {

        VerifierBuilder vbuilder = VerifierBuilderFactory
          .newVerifierBuilder();

        Verifier verifier = vbuilder.newVerifier();

        verifier.addResourcesToVerify(new
            ClassPathResource("rules.drl",
                              RulesVerification.class),
                              ResourceType.DRL);

        if (verifier.hasErrors()) {
            List<VerifierError> errors = verifier.getErrors();
            for (VerifierError ve : errors) {
                System.err.println(ve.getMessage());
            }
            throw new RuntimeException("rules with errors");
        }

        verifier.fireAnalysis();

        VerifierReport result = verifier.getResult();

        Collection<VerifierMessageBase> noteMessages = result
            .getBySeverity(Severity.NOTE);
        for (VerifierMessageBase msg : noteMessages) {
            System.out.println("Note: " + msg.getMessage() +
                " type: " + msg.getMessageType() +
                " on: " + msg.getFaulty());
        }

        Collection<VerifierMessageBase> errorMessages = result
            .getBySeverity(Severity.ERROR);
        for (VerifierMessageBase msg : errorMessages) {
            System.out.println("Note: " + msg.getMessage() +
                " type: " + msg.getMessageType() +
                " on: " + msg.getFaulty());
```

```
            }

            Collection<VerifierMessageBase> warningMessages =
                result.getBySeverity(Severity.WARNING);
            for (VerifierMessageBase msg : warningMessages) {
                System.out.println("Note: " + msg.getMessage() +
                    " type: " + msg.getMessageType() +
                    " on: " + msg.getFaulty());
            }

            Collection<MissingRange> rangeCheckCauses = result
                .getRangeCheckCauses();
            for (MissingRange missingRange : rangeCheckCauses) {
                System.out.println(missingRange);
            }

            verifier.dispose();

        }

    }
```

4. Execute the Java `main` method to visualize the analysis output.

How it works...

As we discussed in the introduction of this recipe, the Drools Verifier module works using a specific set of rules that analyzes the rules added using their API. These rules are added by default when the `Verifier` module is initialized into its internal `StatefulKnowledgeSession` (remember that Verifier is built by rules and it is wrapping a knowledge session and its knowledge base) and also the API gives us the possibility to create more verification rules using the Fact Types provided by Drools Verifier.

In the second step, after declaring some rules to use as example, we created a new Java class with the necessary code to analyze the rules.

Next, you will see an explanation about how it was configured and how Verifier works:

1. The first step is to start analyzing rules using the Verifier API to obtain an `org.drools.verifier.Verifier` object instance using the `VerifierBuilder` obtained through the `VerifierBuilderFactory` object factory.

```
VerifierBuilder vbuilder = VerifierBuilderFactory
    .newVerifierBuilder();
Verifier verifier = vbuilder.newVerifier();
```

2. Once it is done, you can start adding the knowledge resources as DRL and Change Set using the `addResourcesToVerify(Resource resource, ResourceType type)` method in the same way as you add resources to a `KnowledgeBuilder`. When you use a `KnowledgeBuilder` to compile the rules, you can know when there are compilation errors, and it also can be done using `Verifier`:

```
verifier.addResourcesToVerify(new
    ClassPathResource("rules.drl", getClass()),
                        ResourceType.DRL);

if (verifier.hasErrors()) {
    List<VerifierError> errors = verifier.getErrors();
    for (VerifierError verifierError : errors) {
        System.err.println(verifierError.getMessage());
    }
    throw new RuntimeException("rules with errors");
}
```

3. After all the rules that needed to be analyzed are added, it is time to initiate the analysis process using the `fireAnalysis()` method, which will return true if no analysis errors occur:

```
boolean works = verifier.fireAnalysis();
```

4. Once the analysis is completed, the next step is to obtain the rules analysis information output:

```
VerifierReport result = verifier.getResult();
```

Using this `VerifierReport` object, you can obtain the information about the analysis invoking its `getBySeverity(Severity serverity)` method, where `Severity` is one of the next three predefined Verification Severity types:

- **NOTE**: Messages that can be ignored (for example, a repeated constraint in a rule pattern)
- **WARNING**: A possible problem, (for example, an inexistent restriction or a missing complement pattern)
- **ERROR**: Something that needs correction (for example, a pattern that could never be satisfied)

For example, the following code is used to obtain and display the ERROR severity types:

```
Collection<VerifierMessageBase> errorMessages = result
    .getBySeverity(Severity.ERROR);
for (VerifierMessageBase msg : errorMessages) {
    System.out.println("Error: " + msg.getMessage() +
                       " type: " + msg.getMessageType() +
                       " on: " + msg.getFaulty());
}
```

Using this code with the rules added in the example, the following output is obtained:

```
Error: Restriction LiteralRestriction from rule [rule 2] value '== 2'
and LiteralRestriction from rule [rule 2] value '!= 2'are in conflict.
Because of this, pattern that contains them can never be satisfied. type:
INCOHERENCE on: Pattern, name: Server


Error: Restriction LiteralRestriction from rule [rule 2] value '!= 2'
and LiteralRestriction from rule [rule 2] value '== 2'are in conflict.
Because of this, pattern that contains them can never be satisfied. type:
INCOHERENCE on: Pattern, name: Server


Error: Pattern, name: Server in Rule 'rule 2' can never be satisfied.
type: ALWAYS_FALSE on: Pattern, name: Server
```

As we can see in the preceding output, the first two error message outputs are pretty similar because the rules are analyzed in all the pattern constraints, and one constraint is in conflict with another one, due to which, the second one will be also in conflict with the first one. Also, this execution output shows that these errors are an *incoherence* and can be found on the `name: Server` pattern of the rule named `rule 2`. As a consequence of this pattern analysis, the last error message confirms that it is an `ALWAYS_FALSE` type and will never be satisfied.

```
Warning: Rule base covers == 1024, but it is missing != 1024 type:
MISSING_EQUALITY on: LiteralRestriction from rule [rule 3] value '==
1024'


Warning: Rule base covers == 1024, but it is missing != 1024 type:
MISSING_EQUALITY on: LiteralRestriction from rule [rule 1] value '==
1024'


Warning: Rules rule 3 and rule 1 are redundant. type: REDUNDANCY on: null
```

As we can see in the preceding output, by obtaining the WARNING type analysis messages, we can discover that the first two messages are again similar. But it is only because the rules `rule 1` and `rule 3` are equals, and this message says that the pattern is only being applied to the 1024 memory value and all the other values aren't going to be analyzed by other rules. This shows a possible point where the knowledge processing is not going to be applied. Earlier, I said that there are two equals rules, and I also told you about how Drools Verifier has the knowledge to detect *redundancy*. This *redundancy* was detected and it gave us a warning.

In this example, we didn't get any **NOTE** analysis message; however, we are going to obtain the field gaps to know which field restrictions are missing. Think about this as more detailed information than the generic WARNING message obtained earlier, which can be obtained using the `getRangeCheckCauses()` method, shown as follows:

```
Collection<MissingRange> rangeCheckCauses = result
    .getRangeCheckCauses();
for (MissingRange missingRange : rangeCheckCauses) {
    System.out.println(missingRange);
}
```

The output will be as follows:

```
Gap: (Field 'memory' from object type 'Server') Operator = '<' 1024 from
rule: [rule 3]
Gap: (Field 'memory' from object type 'Server') Operator = '>' 1024 from
rule: [rule 3]
Gap: (Field 'processors' from object type 'Server') Operator = '<' 2 from
rule: [rule 3]
Gap: (Field 'processors' from object type 'Server') Operator = '>' 2 from
rule: [rule 3]
Gap: (Field 'processors' from object type 'Server') Operator = '>' 2 from
rule: [rule 2]
Gap: (Field 'memory' from object type 'Server') Operator = '<' 1024 from
rule: [rule 1]
Gap: (Field 'memory' from object type 'Server') Operator = '>' 1024 from
rule: [rule 1]
Gap: (Field 'processors' from object type 'Server') Operator = '<' 2 from
rule: [rule 1]
Gap: (Field 'processors' from object type 'Server') Operator = '>' 2 from
rule: [rule 1]
Gap: (Field 'processors' from object type 'Server') Operator = '<' 2 from
rule: [rule 2]
```

All the fields gap message outputs are pretty similar. For example, from the first two messages, we can discern that the memory field of the `Server` object is not being compared in rules with values less than 1024 and greater than 1024, leaving a nonexistent pattern where `Server` objects aren't going to be evaluated by the inference engine.

As you have seen in this recipe, Drools Verifier is a powerful module to analyze the quality of your knowledge, detecting possible points of failure, and also is integrated in Guvnor to build a complete set of rules authoring.

There's more...

Drools Verifier also has the feature to create HTML/TXT reports of the analysis results and, as you can imagine, the reports can only be created once the analysis execution is done. To create these reports, the `VerifierReportWriterFactory` is used to obtain a `VerifierReportWriter` instance, which depends on the report style needed. In the following steps, you will learn quickly how to create your Verifier reports:

1. In this case, an HTML report is going to be generated, so the `newHTMLReportWriter()` static method is invoked to obtain a `VerifierReportWriter`:

```
VerifierReportWriter htmlWriter = VerifierReportWriterFactory
    .newHTMLReportWriter();
```

2. The next step is to specify the output file creating a `FileOutputStream` object with the filename as the constructor parameter. In this case, the file extension is ZIP because the `HTMLReportWriter` generates a ZIP file with all the HTML files compressed.

```
FileOutputStream htmlOut = new
    FileOutputStream("htmlReport.zip");
```

3. The last step of the report generation is to use the `VerifierReportWriter` instance to write the report, using the previously created `FileOutputStream` object as the first parameter and the verifier execution result as the second one.

```
htmlWriter.writeReport(htmlOut, verifier.getResult());
```

   Once this method is invoked, the report should appear in the specified directory path, ready to be unzipped and read with any HTML browser/editor.

Monitoring knowledge with JMX

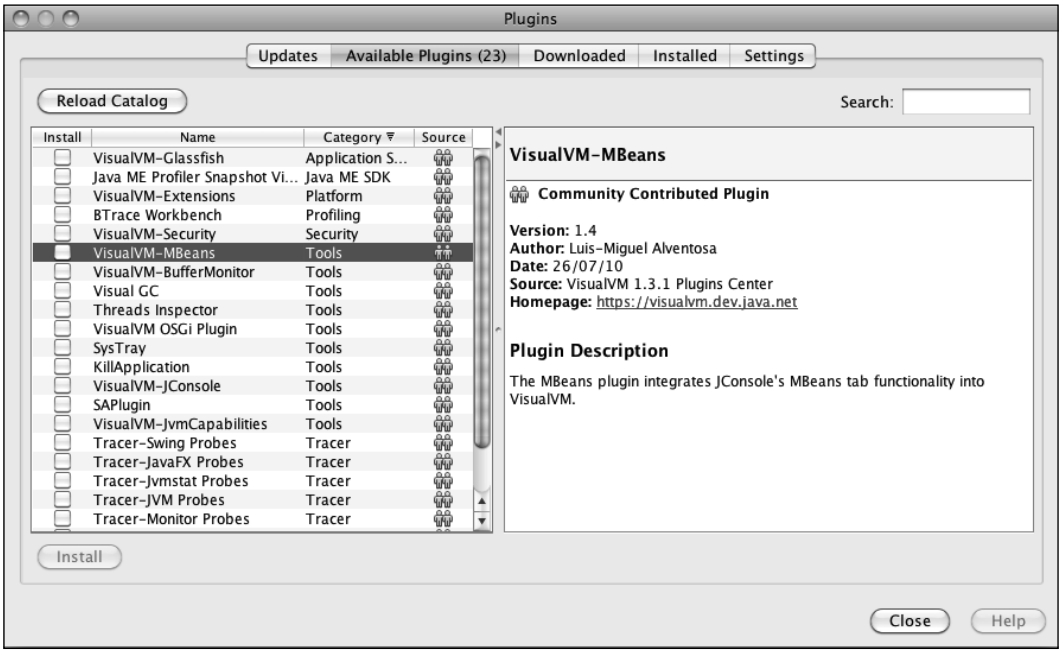
In this recipe, we will see how to enable the Drools monitoring features and how knowledge bases and knowledge sessions can be monitored. This feature is useful to obtain internal state information about the number of inserted facts in the working memory, fired/created/cancelled activations, rules state information, and so on that was implemented using the JMX technology, which is an industry standard, used to manage and monitor applications/resources.

Getting ready

Before you start enabling the monitoring features, we need to download a **JMX** monitoring tool. In this case, we are going to use **VisualVM**, which actually is more than a JMX monitoring tool as it is primarily used as a profiling application. Download VisualVM 1.3.1 or higher version (if available) from `https://visualvm.dev.java.net/download.html`. It is possible that VisualVM is already installed in your environment because it is bundled in Sun JDK 6 since update 7 and Apple's Java for Mac OS X 10.5 Update 4.

The minimal requisite to execute VisualVM is to have Oracle/Sun JDK 6+ installed.

Once the application has been downloaded, decompress the ZIP file and execute the `visualvm` binary file located in the bin directory. After the License Agreement has been accepted, it's time to install the MBean Plugin (for more information about MBeans refer to the *There's more...* section of this recipe) to access to the Drools metrics information. The plugins installation options can be accessed through the **Tools** | **Plugins** menu, in which the **Available Plugins** tab is the one of most interest to us. Check the install column's checkbox of the **VisualVM-Mbeans** plugin in the list. Click the **Install** button and proceed with the installation, as shown in the following screenshot:



Once the installation is done, we are ready to enable the monitoring feature in our Drools-enabled application.

How to do it...

Follow the steps given here, in order to monitor your knowledge session using JMX.

1. First, you have to enable the Drools MBean management. This is done configuring the knowledge base using the `MBeansOption.ENABLED` option.

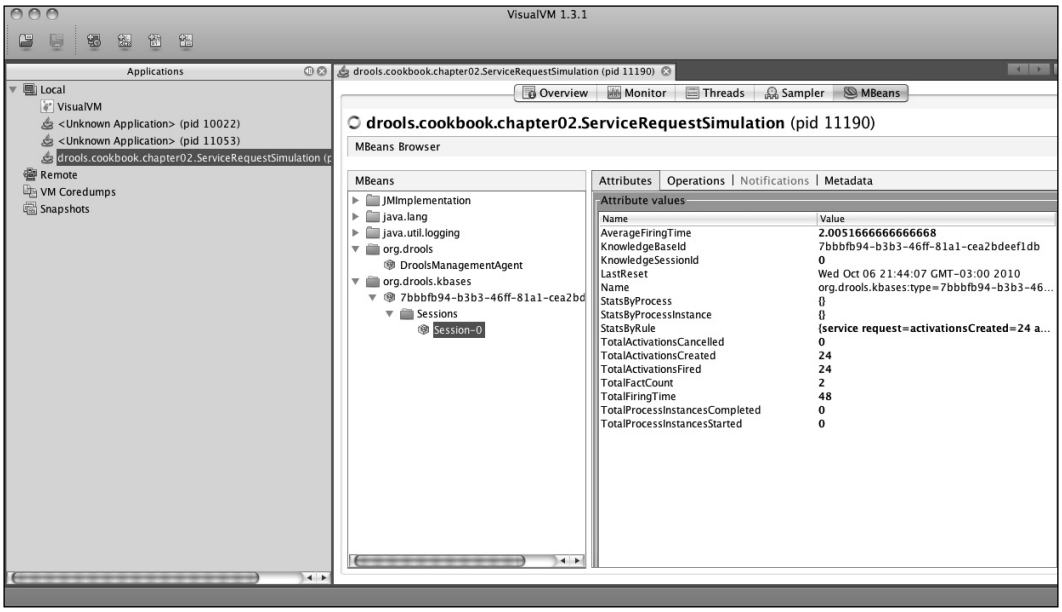
```
KnowledgeBaseConfiguration config = KnowledgeBaseFactory
    .newKnowledgeBaseConfiguration();
config.setOption(MBeansOption.ENABLED);

KnowledgeBase kbase = KnowledgeBaseFactory
    .newKnowledgeBase("kbase", config);
        kbase.addKnowledgePackages(kbuilder.
getKnowledgePackages());
```

2. After the knowledge base is configured, and before executing the application, the JVM must be configured with the following arguments to enable the JMX remote management:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=3000
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

3. Once the application is executed, VisualVM is ready to be started. In the left panel, there is a list of the applications ready to be monitored. Select your application and the **MBeans** tab to see the drools-related information, as shown in the following screenshot:



How it works...

Configuring a knowledge base with the `MBeansOption.ENABLED` option enables the Drools JMX feature and it will start an internal `MBeanServer` that will register all the knowledge sessions created from this knowledge base. In this implementation, the resources can be knowledge bases and stateful knowledge sessions. Stateless knowledge sessions can't be monitored because of their nature. It is a wrapped stateful knowledge session that is regenerated each time the rules are fired, and as a consequence the working memory state is lost.

There's more...

There is a lot of information exposed and this will be explained in detail in the following sections:

KnowledgeBase information

This information can be accessed in the `org.drools.kbases` tree node, where each knowledge base is identified by a unique UID.

The following information is displayed in the **Attributes** tab:

- `Globals`: Globals defined into the knowledge rules
- `Id`: Auto-generated or specified knowledge base UID
- `Packages`: Rules packages names that are part of the knowledge rules
- `SessionCount`: Count of knowledge sessions created from this knowledge base

Also, you will notice that there are three more tabs available. In this case, the **Operations** tab has two methods that can be used to initialize internal MBeans to gather extra configuration regarding the knowledge base (these MBeans methods are commonly used to configure the MBeans). The **Metadata** tab displays information about operations and attributes, and the **Notifications** tab is used to subscribe to MBeans notifications and this functionality is not implemented in Drools MBeans.

The knowledge base MBean available methods are as follows:

- `startInternalMBeans()`: Starts the internal MBeans that display the knowledge base configuration and entry points related information
- `stopInternalMBeans()`: Used to stop the previously started internal MBeans

KnowledgeSession information

In each knowledge base tree node and inside the `Sessions` folder, you will discover all the knowledge sessions that were created from it.

The following information is displayed in the **attributes** tab:

- `TotalProcessInstancesStarted`: Total number of process instances started

- `TotalProcessInstancesCompleted`: Total number of process instances completed
- `TotalFiringTime`: Total activations firing time
- `TotalFactCount`: Total working memory fact count
- `TotalActivationsFired`: Total number of activations fired in the knowledge session
- `TotalActivationsCreated`: Total number of activations created in the knowledge session
- `TotalActivationsCancelled`: Total number of activations cancelled in the knowledge session
- `StatsByRule`: Description of total number of fired/created/cancelled activations generated by each rule
- `StatsByProcessInstance`: Description of total number of fired/created/cancelled activations generated by each process instance
- `StatsByProcess`: Description of total number of fired/created/cancelled activations generated by each process
- `Name`: Knowledge session name
- `LastReset`: Last time that the information was reset
- `KnowledgeSessionId`: Autogenerated knowledge session UID
- `KnowledgeBaseId`: Autogenerated or specified UID of the knowledge base from which the knowledge session was created
- `AverageFiringTime`: Average activation firing time

Also, in the **Operations** tab there are four methods that can be used to retrieve specific information about rule/process/process instance statistics and one to reset the metrics statistics.

With this information, you are able to monitor the framework's internal state, visualizing the data, creating simple graphics, and also monitoring the JVM performance.

Another option to consume the Drools MBeans information is using the JMX API. To achieve this, you have to create the connection URL that will contain the IP address and port of the JMX-enabled application, which was previously configured through the JVM arguments.

```
String url = "service:jmx:rmi:///jndi/rmi://ip_address:port/jmxrmi"
```

Once the connection URL is defined, it is time to create the connection to be able to consume the knowledge information.

```
JMXConnector jmxConnector = JMXConnectorFactory.connect(
    new JMXServiceURL(url));
MbeanServerConnection connection = jmxConnector.
getMBeanServerConnection();
```

If the connection was successful, it is time to begin retrieving the MBean information. This is the most complicated step because you have to discover which MBeans were registered (each knowledge base and knowledge session has its associated MBean) and how to consume the attributes.

The registered knowledge base MBeans can be discovered using the following namespace: `org.drools.kbases:type=*` , where the `*` is the wildcard to discover all of them, using the following code snippet:

```
String resourceFilter = "org.drools.kbases:types=*";
Set<ObjectName> names = connection.queryNames(new
ObjectName(resourceFilter), null);
```

Once we have the `ObjectNames` objects, they can be used to consume the MBean attributes of their associated `KnowledgeBase` using their `getAttribute()` method (where the `attributeName` is the attribute's name that can be viewed using VisualVM, for example, Packages attribute):

```
Object value = connection.getAttribute(objectName, attributeName);
```

Finally, to discover the knowledge session MBeans the following namespace can be used, in the same way as the knowledge base MBeans were discovered: `org.drools.kbases:type=*,group=Sessions,sessionId=Session-*`

Where to buy this book

You can buy Drools Developer's Cookbook from the Packt Publishing website: `http://www.packtpub.com/drools-developers-using-jboss-cookbook/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/drools-developers-using-jboss-cookbook/book