# JBoss Rule Engine 6

**Rule Basic Structure**

Depending on the specific syntax of each rule engine, the syntax might vary a bit. Nonetheless, there are primal structures that are common to all the rule engines and they look something similar to the following

**when** <a condition is found to be true>,

**then** <an action is executed>

A condition is basically a constraint or filter. These filters will look at the information available in a domain to try and find data that meets the defined criteria. Once a group of data is obtained matching the condition, an action gets executed taking the matching data as a parameter.

Source code for the samples can be available at below location

**git clone https://bitbucket.org/drools-6-developer-guide/drools6-dev-guide.git**

Following maven dependencies are required to creat simple rule

<dependency>

<groupId>org.kie</groupId>

<artifactId>kie-api</artifactId>

<version>${drools.version}</version>

<type>jar</type>

</dependency>

<dependency>

<groupId>org.drools</groupId>

<artifactId>drools-compiler</artifactId>

<version>${drools.version}</version>

<type>jar</type>

</dependency>

<dependency>

<groupId>org.drools</groupId>

<artifactId>drools-core</artifactId>

<version>${drools.version}</version>

<type>jar</type>

</dependency>

**<drools.version>6.3.0.Final</drools.version>**

Create **kmodule.xml** file under META-INF folder to create. This file will be used to configure how to load the rules defines in the project in the rule engine. For simple rule content of **kmodule.xml** will be quite simple, as we will be using all the default configurations.

<kmodule xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://jboss.org/kie/6.0.0/kmodule"*>

</kmodule>

This file will be picked up when we instantiate Rule Engine session automatically to figure out what needs to be loaded.

Once the maven project is setup with above simple steps.

We can write one simple rule now here.

Write a file under resource folder of maven project with extension .drl that is treated as rule.

**package myfirstproject.rules**

**import org.drools.devguide.eshop.model.Item;**

**import org.drools.devguide.eshop.model.Item.Category;**

**rule "Classify Item - Low Range"**

**when**

**$i: Item(cost < 200)**

**then**

**$i.setCategory(Category.LOW\_RANGE);**

**end**

This rule checks for each item that costs less than 200 USD and automatically tags it with a category, in this case, LOW\_RANGE. For our shop, it makes sense to differentiate our items in different ranges so that we can apply different discounts and marketing strategies for them. This classification process can be done automatically using rules, which centralize the point where we have this business definition of what LOW\_RANGE, MID\_RANGE or HIGH\_RANGE items they are.

DRL files will be structured as follow:

**Package definition:** This is the same as in Java, we will declare a package for our rules

**Imports section:** We need to import all the classes that we are going to use in of our rules.

**(Optional) Declared types and events:**

**Rules**: (1..N)/Queries (1..N)

Next step we need to create Java file to run the rule.

System.***out***.println("Bootstrapping the Rule Engine ...");

// Bootstrapping a Rule Engine Session

**final** KieServices ks = KieServices.Factory.*get*();

**final** KieContainer kContainer = ks.getKieClasspathContainer();

**final** KieSession kSession = kContainer.newKieSession();

**final** Item item = **new** Item("A", 123.0, 234.0);

System.***out***.println("Item Category: " + item.getCategory());

kSession.insert(item);

**final** **int** fired = kSession.fireAllRules();

System.***out***.println("Number of Rules executed = " + fired);

System.***out***.println("Item Category: " + item.getCategory());

**Rule Language**

**rule** "name"

**when** (Conditions) - also called Left Hand Side of the Rule (LHS)

**then** (Actions/Consequence) - also called Right Hand Side of the Rule (RHS)

**end**

**Understanding Drools Runtime**

Drools allow us to create instance of the Rule engine in different ways. So that we can choose which fits better to the problem that we are trying to solve. Each rule Engine instance is an encapsulated context. Where the rules that we define will be evaluated aginst the data that we provide to this particular instance. Drools allow us to locally spawn lightweight instances to our applications.

It is good to have multiple instances dealing with different data and rules than one big instance.

Inorder to spawn a new instance of the rule engine, we need to understand the following concepts:

* KieServices
* KieContainer
* KieModule
* KieBase
* KieSession

These five concepts are extended versions of what was provided in the previous versions of Drools. Prefix KIE (**Knowledge is everything**) indicates the fact that now we are now only dealing with Rule Engine instances, but with more ways dealing and executing business knowledge in general.

**KieServices**: KieServices class gives us access to all these other concepts by providing a registry of services where we can find helpers for different purposes. Future versions of Drools, more services may be included to fulfill different use cases.

**final** KieServices ks = KieServices.Factory.*get*();

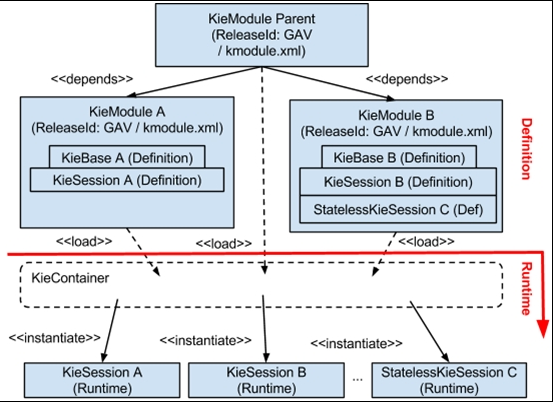
Using **KieServices**, we can access a number of factories, services and utility methods used along with Rule Engine instances.

We will use **KieServices** to create a new instance of **KieContainer**, which defines the scope of the rules that will be used to create new instances of the Rule Engine.

**final** KieServices ks = KieServices.Factory.*get*();

**final** KieContainer kContainer = ks.getKieClasspathContainer();

A KieContainer can host a KieModule and its dependencies.



In Drools 6, everything is created around **KieModules**, Each **KieModule** contains business assets (business rules, business processes, decision tables and so on) related to a certain area or domain.

These **KieModules** can include othe KieModules, allowing us to **compose** a top level KieModule, containing several assets from different domains.

A **KieModule** is a standard Java-Maven project containing the rules, business process and other assets among its resources.

A special file called **kmodule.xml** (inside **META-INF**) that defines the internal configuation about how to group and consume these particular assets must also be present in it.

**KieModule& KieContainer**

Once we get the **KieServices** instance, we can get **KieContainer** instance that internally loads all the business assets (rules, processes, spreadsheets, PMML documents and so on) when we create new Rule Engine instances.

In Drools 6, we can choose betweeen two options to define the scope of the resources and configurations that will be included in an instance of **KieContainer**.

* **Based on the classpath:** this will look business assets in the classpath and allow us to load them in different instance of the rule engine.
* **Using Maven dependency resolution techniques (KIE-CI):** it delegates the responsibility of finding out predefined artifacts and their transitive dependencies into Maven to find out all the resources that need to be included.

**Using Classpath Resources to load Rules**

1. Create KieContainer instance from KieServices

**final** KieServices ks = KieServices.Factory.*get*();

**final** KieContainer kContainer = ks.newKieClasspathContainer();

1. Verify all the resources loaded correctly or not using below code snippet

**final** Results results = kContainer.verify();

results.getMessages().stream().forEach((message) -> {

System.***out***.println(">> Message ( " + message.getLevel() + " ): " + message.getText());

});

1. Using below code we can make sure Kbase and Ksessions are loaded properly

kContainer.getKieBaseNames().stream().forEach((kieBase) -> {

System.***err***.println(">> Loading KieBase: " + kieBase);

kContainer.getKieSessionNamesInKieBase(kieBase).stream().forEach((kieSession) -> {

System.***err***.println("\t KieSession: " + kieSession);

});

});

1. Pass the keySessionName to create new kieSession

**final** KieSession kieSession = kContainer.newKieSession("rules.cp.discount.session");

1. Final step pass the objects to session and fire the rules.

**Using Maven Dependencies to load rules**

1. Add below dependency to your maven build file.

<dependency>

<groupId>org.kie</groupId>

<artifactId>kie-ci</artifactId>

<version>${drools.version}</version>

</dependency>

1. This new dependency enable drools to use maven mechanism to resolve artifacts that are outside of the application classpath.
2. Below code snippet gives an idea how to create KieContainer

**KieContainer kContainer = ks.newKieContainer (ks.newReleaseId ("groupId", "artifactId", "version"));**

1. Which loads the resources from the maven repository
2. Load the metadata using and packages that contain the rules can be accesed using below code.

KieModuleMetaData kieModuleMetaData = KieModuleMetaData.Factory.*newKieModuleMetaData*(ks.newReleaseId("groupId", "artifactId","versionId"));

kieModuleMetaData.getPackages().stream().forEach((pkg) -> {

kieModuleMetaData.getRuleNamesInPackage(pkg).stream().forEach((rule) -> {

System.***err***.println("\t >> Contain Rule: " + rule);

});

});

**Stateful Vs Stateless Sessions:**

**Stateful Session:** (also called KieSessions) The stateful KieSession allows us to keep the state between several interactions with the Rule Engine. In Drools 6, Stateful Knowledge Sessions were renamed to **KieSessions**, as there is the most common type of session, the name was kept short.

**Stateless Session: StatelessKieSession** only allows us to interact once, take the results out, and no state is stored for the next interaction.

**KieScanner**

KieContainer can be manually updated when a new version of a KieModule is deployed; one of the limitations of this approach is that we have to manually notify each of the applications that depend on the modified KieModule.

What if there was a way to automatically let out the applications to be notified when a **KieModule** that they depend on gets updated? Fortunately for us, Drools provides this mechanism out of the box, its name is **KieScanner**.

In order to use **KieScanner** in our application, the **org.kie: kie-ci**artifact must be added to the application's classpath.

The **KieScanner** component in Drools is nothing but a wrapper around a **KieContainer** that can be configured to automatically detect changes in the resources that the container depends on.

NOTE: Resources referenced by the KieContainer being monitored must be KieJars residing in a maven repository, by default; kie-cli will use Maven’s settings.xml in the .m2 folder of the user’s home.

This behaviour can be overridden using the below system property.

**-Dkie.maven.settings.custom=/my/path/to/maven/settings.xml**

And can be created using below code snippet

KieServices ks = KieServices.Factory.get ();

KieContainer kieContainer = ks.newKieContainer (ks.newReleaseId ("group.test","artifact-test", "1.0"));

KieScanner scanner = ks.newKieScanner (kieContainer);

After KieScanner instatiated, we have two options we could either configure it to poll for new versions of the underlying KieJars on every fixed amount of milliseconds or we can manually force it to check for new versions on demand.

**OPTION-1**: Scanner.scanNow (); //manually run a check for new versions.

**OPTION-2:** scanner.start (10\_000); //check every 10 seconds

If the **start ()** method was used to start the KieScanner, the antagonic **stop ()** method can be used to stop the polling mechanism.

When **KieScanner** finds a new version of a KieJar, an incremental build resource is triggered. From this moment, all the new **KieBases** and **KieSessions** created from the **KieContainer** being wrapped will use the new version of the resources. It is important to mention that any pre-existing **KieBase** and **KieSession** will also be updated to the latest version of the resources found by the **KieScanner**.