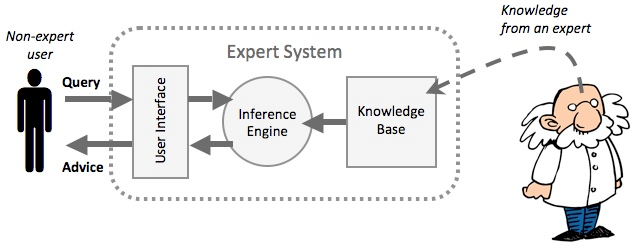
# zhttps://dzone.com/articles/introduction-to-rules-engine-using-drools-expert

# Introduction to Rules Engines Using Drools Expert

### What helps make Drools special is the in-depth decision making it can do. Dive into the rules engine of Drools to see what makes it tick and how to use it.

Dec 21 2016:

Business Rules Engines are required to execute one or more rules in an Enterprise or Software System. The ability to maintain and execute rules as separate from the application code is the greatest advantage of such engines. Business Rules Engine are a form of Expert System. Expert System in the most simple terms, allows human expert-like

[](http://3.bp.blogspot.com/-xWs1Xg66POU/ViTf8br5OoI/AAAAAAAABMc/HE0jUGexNN8/s1600/expert_systems_01.png)

[Image credit: http://www.igcseict.info]

Business Rules Engineare, primarily, of two types and classified on the basis of how rules are scheduled for execution.

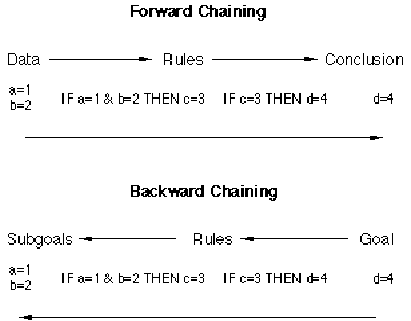
## ****Forward Chaining (Data-Driven)****

1. Inference Engine: These are based on a set of 'If-Then' kind of behaviors or evaluations.

2. Reaction Rules: These are used to process event patterns and perform actions.

## ****Backward Chaining (Goal-Driven)****

1. It tries to resolve facts based on particular goals.



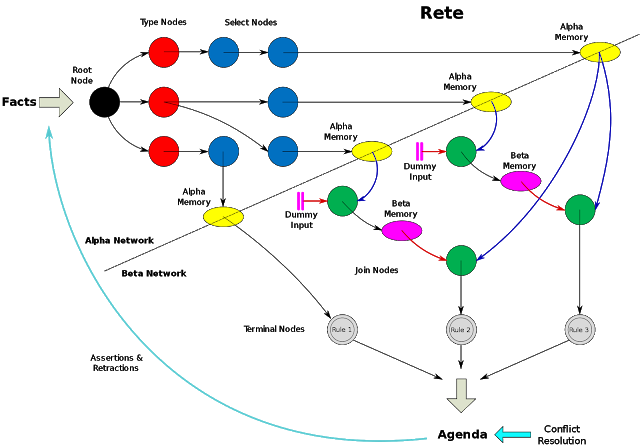
[Image credit: http://www.amzi.com]

## ****Drools Expert****

Drools Expert works on the basis of the Rete/Rete-OO Algorithm. Drools is an open-source project that has the following major components

* **Drools Guvnor** (Business Rules Manager): a centralized repository for Drools knowledge bases.
* **Drools Expert** (Rules Engine): Uses the rules to perform reasoning.
* **Drools Flow** (Process/Workflow), or jBPM 5: Provides for workflow and business processes.
* **Drools Fusion**(Event Processing/Temporal Reasoning): Provides for complex event processing.
* Drools Planner/OptaPlanner (Automated Planning): Optimizes automated planning, including NP-Hard Planning Problems.

A block diagram of the Rete/Rete-OO is given below. The Rete Algorithm requires an extensive discussion, which I am not scoping into this blog entry.

[](http://3.bp.blogspot.com/-TXyWEf4iXbQ/ViTzbLzRcwI/AAAAAAAABM8/Tc2rnZa2KA4/s1600/rete_algorithm.png)

[Image credit: http://en.wikipedia.org]

## ****Use-Case(s) Implemented Here (to Demonstrate Drools Expert)****

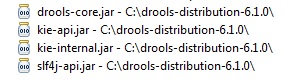
If the 'Source IP' is a Specific IP and the 'Source Port' is a Specific Number, then mark the 'Event' as 'Blacklisted' [Unsafe Event Detection]

I will demonstrate only this particular use-case in this blog entry (including how to run the 'Intelligent Data Loader') to understand rules engine processing. You may need to do the following before you can download and run the code:

1. Download Drools 6.1.0 distribution (include in classpath).
2. Download the Eclipse Plugin for Drools (include in classpath) .
3. Use JDK 1.8.0 and JEE 1.7 libraries (if required) (include in classpath) .
4. Brief Read on MVEL Dialect and Drools Expert (Above/Official)   .

### ****1. Start a Java Project in Eclipse [Classpath]****

You may choose to start a 'Java' project, as opposed to a 'Drools' project. Then include the following JARs in the classpath:

[](http://2.bp.blogspot.com/-el50kfBS75Q/ViYM0lwjv6I/AAAAAAAABNc/M8TH70p3ZbQ/s1600/DROOLS_LIBS.jpg)

### ****2. The Directory/Folder Structure Should Include a 'Resources' as Source Folder****

[](http://2.bp.blogspot.com/-WvB9L2Fn90w/ViYNRWx6ZBI/AAAAAAAABNo/DG80dXynxNQ/s1600/DROOLS_RESOURCES.jpg)

### ****3. Create the Drools Expert Configuration File (kmodule.xml)****

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

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

<kbase name="rules" packages="rules">

<ksession name="ksession-rules"/>

</kbase>

</kmodule>

### ****4. Create the Rule Using the 'mvel' Dialect****

The creation of your first rule using the 'mvel' dialect though should not be a very difficult task for the experienced Java developer. I am not explaining the use-case implementation detail, except that it checks if the source ip and source port are equal to a specific number.

package com.bw2015.sample.biz.re;

// list any import classes here.

import com.bw2015.sample.biz.vo.SampleEvent;

// use case 01

// detect if we can blacklist a specific ip and port access

rule "Port and IP Blacklist Rule"

dialect "mvel"

no-loop

when

$sampleEvent:SampleEvent(eventSourceIp=="216.39.58.18", eventSourcePort=="8080")

then

System.out.println("\*\*\*\*\* Blacklisted IP and Port Detected in Event with Remarks - " + $sampleEvent.getEventRemarks());

end

### ****5. Develop the Core Rules Engine Processing Class****

Instantiate the important Drools Runtime objects as shown below. You may also understand from the below code, how to include and refer the rules file (**.drl**) from the classpath.

private static SampleRulesEngine reService = null;

// Drools Expert Runtime Configuration

private KieServices ks;

private KieContainer kContainer;

private KieSession kSession;

public static SampleRulesEngine getInstance() {

if(reService==null) {

reService = new SampleRulesEngine();

reService.init();

}

return reService;

}

public void init() {

try {

System.out.println("initializing kie runtime for drools expert...");

ks = KieServices.Factory.get();

kContainer = ks.getKieClasspathContainer();

KieSessionConfiguration sessionConfiguration = KnowledgeBaseFactory.newKnowledgeSessionConfiguration();

kSession = kContainer.newKieSession("ksession-rules", sessionConfiguration);

System.out.println("initialized the kie runtime for drools expert...");

} catch (Exception e) {

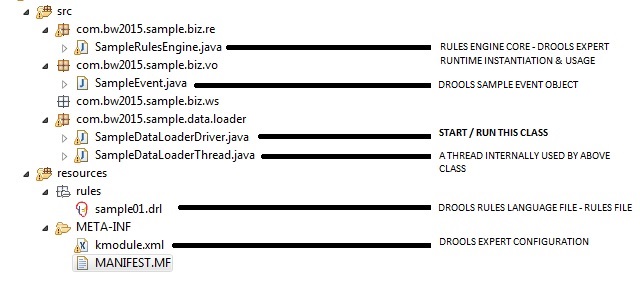
e.printStackTrace();

}

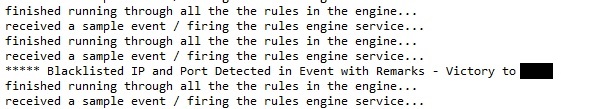
}

### ****6. Build a Data Loader (To Inject Positive Test Cases and Load Test)****

There is a data loader that starts, internally, a threaded loading mechanism for sending multiple events to the Drools Expert Runtime. You have to make sure that you 'Inject' a positive test case at every interval, to test out the functionality of Drools Expert. The Java Event Object in our example is "**SampleEvent**".

[](http://3.bp.blogspot.com/-vWmmdoxRFtc/ViYMBvPIfzI/AAAAAAAABNU/Qvb9h8eprGA/s1600/DROOLS_EXPERT.jpg)

### ****7. Output of Running the SampleDataLoaderDriver (Rules Engine Output)****

[](http://4.bp.blogspot.com/-s_CL39if4io/ViYOSvxaFCI/AAAAAAAABNw/R83yAbqS-V8/s1600/DROOLS_OUTPUT.jpg)

There are other rule forms that you may use such as Decision Tables. It is only a matter of configuration, classpath, regular expressions, and a straightforward understanding of .xls files to create Decision Tables. Also, I recommend that you make sure that you use Microsoft Excel to edition the Decision Table Spreadsheet, or else you may end up with weird issues.

[**Drools & jBPM**](http://blog.athico.com/)

All things Artificial Intelligence related: Rules, Processes, Events, Agents, Planning, Ontologies and more :)

March 30 2009

Drools Quick Start - Stateful Knowledge Session

Posted by Mark Proctor

Here is an extract from the quick start guide I just writing on using stateful sessions:  
  
Stateful sessions are longer lived and allow iterative changes over time. Some common use cases for stateful sessions are, but not limited to:

* Monitoring
  + Stock market monitorig and analysis for semi-automatic buying
* Diagnostics
  + fault finding, medical diagnostics
* Logistics
  + parcel tracking and delivery provisioning
* Compliance
  + Validation of legality for market trades.

Unlike a stateless session the dispose() method must be called afterwards to ensure there are no memory leaks, as the KnowledgeBase containes references to StatefulKnowledgeSessions when they are created. StatefulKnowledgeSession also supports the BatchExecutor interface like StatelessKnowledgeSession, the only difference is that when used with stateful the FireAllRules command is not automatically called at the end.  
  
We can use a fire alarm example to explore the monitoring use case. The simple example has just 4 classes. We are monitoring the rooms in a house, each room has one sprinkler. If a fire starts in a room, we represent that with a single Fire instance.

public class Room {  
 private String name;  
 // getter and setter methods here  
}  
  
public classs Sprinkler {  
 private Room room;  
 private boolean on;  
 // getter and setter methods here  
}  
  
public class Fire {  
 private Room room;  
 // getter and setter methods here  
}  
  
public class Alarm {  
}

In the previous section on stateless sessions the concepts of inserting and matching against data was introduced. That example assumed only a single instance of each object type was ever inserted and thus only used literal constraints. However a house has many rooms, so rules have the need to express joins that constraint to the desired objects, this can be done using a binding as a variable constraint in a pattern. This join process results in what is called cross products, which are covered in the next section.  
  
When a fire occurs an instance of the Fire class is created, for that room, and insert it. The rule uses a binding on the room field of the Fire to constrain to the Sprinkler for that room, which is currently off. When this rule fires and the consequence is executed the sprinkler is turned on

rule "When there is a fire turn on the sprinkler"  
when  
 Fire($room : room)  
 $sprinkler : Sprinkler( room == $room, on == false )  
then  
 modify( $sprinkler ) { setOn( true ) };  
 System.out.println( "Turn on the sprinkler for room " + $room.getName() );  
end

Where as the stateless session used standard java syntax to modify a field, in the above rule we use the modify keyword, which acts as a sort of with statement, that contains a series of comma separated java expressions. Stateless sessions typically do not use inference, which can be explicitly turned off by using the "sequential mode", so the engine does not need to be aware of changes to data, however a stateful session does. The modify keyword allows the setters to modify the data, while make the engine aware of those changes so it can reason over them, this process is called inference.  
  
So far we have rules that tell us when matching data exists, but what about when it doesn't exist? What about when there stops being a Fire? Previously the constraints have been propositional logic where the engine is constraining against individual intances, Drools also has support for first order logic that allows you to look at sets of data. The 'not' keyword matches when something does not exist. So for a Room with a Sprinkler that is on when the Fire for that room stops existing we can turn off the sprinkler.

rule "When the fire is gone turn off the sprinkler"  
when  
 $room : Room( )  
 $sprinkler : Sprinkler( room == $room, on == true )  
 not Fire( room == $room )  
then  
 modify( $sprinkler ) { setOn( false ) };  
 System.out.println( "Turn off the sprinkler for room " + $room.getName() );  
end

While there is a Sprinkler per room, there is just a single Alarm for the building. An Alarm is created when a Fire is occurs, but only one Alarm is needed for the entire building, no matter how many Fires occur. Previously 'not' was introduced, the compliment to ths is 'exists' which matches for one or more of something.

rule "Raise the alarm when we have one or more fires"  
when  
 exists Fire()  
then  
 insert( new Alarm() );  
 System.out.println( "Raise the alarm" );  
end

Likewise when there are no Fires we want to remove the alarm, so the 'not' keyword can be used again.

rule "Lower the alarm when all the fires have gone"  
when  
 not Fire()  
 $alarm : Alarm()  
then  
 retract( $alarm );  
 System.out.println( "Lower the alarm" );  
end

Finally there is a general health status message, that is printed when the application first starts and after the Alarm is removed and all Sprinklers have been turned off.

rule "Status output when things are ok"  
when  
 not Alarm()  
 not Sprinkler( on === true )  
then  
 System.out.println( "Everything is ok" );  
end

The above rules should be placed in a single drl file and saved to the classpath using the file name "fireAlarm.drl", as per the stateless session example. We can then build a KnowledgeBase as before, just using the new name "fireAlarm.drl". The difference is this time we create a stateful session from the kbase, where as before we created a stateless session.

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();  
kbuilder.add( ResourceFactory.newClasspathResource( "fireAlarm.drl", getClass() ),  
 ResourceType.DRL );  
if ( kbuilder.hasErrors() ) {  
 System.err.println( builder.getErrors().toString() );  
}  
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

With the session created it is now possible to iteratvely work with it over time. Four Rooms are created and inserted, a Sprinkler for each room is also inserted. At this point the engine has done all it's matching, but no rules have fired. calling "fireAllRules" on the ksession allows the matched rules to fire, currently that is just the health message.

Room kitchen = new Room( "kitchen" );  
Room bedroom = new Room( "bedroom" );  
Room office = new Room( "office" );  
Room livingRoom = new Room( "livingroom" );  
  
ksession.insert( kitchen );  
ksession.insert( bedroom );  
ksession.insert( office );  
ksession.insert( livingRoom );  
  
Sprinkler kitchenSprinkler = new Sprinkler( kitchen );  
Sprinkler bedroomSprinkler = new Sprinkler( bedroom );  
Sprinkler officeSprinkler = new Sprinkler( office );  
Sprinkler livingRoomSprinkler = new Sprinkler( livingRoom );  
  
ksession.insert( kitchenSprinkler );  
ksession.insert( bedroomSprinkler );  
ksession.insert( officeSprinkler );  
ksession.insert( livingRoomSprinkler );  
  
ksession.fireAllRules()  
  
> Everything is ok

We now create two fires and insert them, this time a referenced is kept for the returned FactHandle. The FactHandle is an internal engine reference to the inserted instance and allows that instance to be retracted or modified at a later point in time. With the Fires now in the engine, once "fireAllRules" is called, the Alarm is raised and the respectively Sprinklers are turned on.

Fire kitchenFire = new Fire( kitchen );  
Fire officeFire = new Fire( office );  
  
FactHandle kitchenFireHandle = ksession.insert( kitchenFire );  
FactHandle officeFireHandle = ksession.insert( officeFire );  
  
ksession.fireAllRules();  
  
> Raise the alarm  
> Turn on the sprinkler for room kitchen  
> Turn on the sprinkler for room office

After a while the fires will be put out and the Fire intances are retracted. This results in the Sprinklers being turned off, the Alarm being lowered and eventually the health message is printed again.

ksession.retract( kitchenFireHandle );  
ksession.retract( officeFireHandle );  
  
ksession.fireAllRules();  
  
> Turn on the sprinkler for room office  
> Turn on the sprinkler for room kitchen  
> Lower the alarm  
> Everything is ok

Every one still with me? That wasn't so hard and already I'm hoping you can start to see the value and power of a declarative rule system.