DZone Refcardz

CONTENTS INCLUDE:

Business Rule Management with Drools

By: Marcin Grzejszczak & Mario Fusco

ABOUT BRMS

A Business Rule Management System is software that creates, supports, and executes decision logic and business rules. A business rule is composed of actions that are taken when certain conditions are satisfied in the input data.

This Refcard focuses on Drools, a rules engine for Java and .NET, and is aimed at experienced developers who are new to BRMS.

BENEFITS OF A BRMS

- BRMS allows decoupling management of business logic from the software development team.
- Business rules can be versioned separately from the code.
- Rules are constructed in a manner more understandable by non-technical staff by means of using flows, decision tables, and specific languages.
- Requirements can be naturally translated into rules.
- Each rule models an isolated, small portion of the business logic and is not part of a monolithic program.
- It is easier to modify a rule than a Java program and to determine the impact of this change on the rest of the application.

Use a Rule Engine When . . .

- The issue is too complex to be dealt with using the standard if... else... approach to the code.
- The issue is not too complex to be dealt with using the standard if... else... approach in the code, but the solution might be subject to frequent changes that could break the existing architecture.
- The solution to the problem would require too many nested if... else... statements.
- The hardcoded version of the solution would be unmaintainable.
- The issue is complex and you cannot think of any way to divide the problem into a series of smaller ones.
- You want to provide ways for a nontechnical business analyst to enter new rules.

Do Not Use a Rule Engine When . . .

- The logic behind the rules is simple.
- It is impossible to write the rules without using a series of if... else... statements inside the rule files.
- It is not subject to frequent changes or doesn't change at all (regardless of complexity).
- The problem can be divided into a small set of conditions and actions.

ABOUT DROOLS EXPERT

Drools is a business rule management system (BRMS) that supports the JSR-94 standard and uses the forward chaining inference-based rules engine, while also supporting backward chaining, that runs on the enhanced implementation of the Rete algorithm.

Advantages of a Rule Engine

Declarative vs. Imperative programming
Instead of focusing on the direct implementation of the problem in imperative "how to" programming, the Rule engine allows you to declare the issue in a way more understandable to the user. The declaration of the problem focuses more on what is to be done than on how to do it.

This requires a cognitive shift when thinking about rule-based programs because rules are not called directly, as with methods, but are selected and fired by the rule engine.

A method must be called directly

Specific passing of arguments

```
public void helloMark(Person person) {
        if ( person.getName().equals( "mark" ) {
                System.out.println( "Hello Mark" );
        }
}
```

Rules can never be called directly

Specific instances cannot be passed but are automatically selected with pattern-matching

```
rule "Hello Mark"
        when
                Person( name == "mark" )
        then
                System.out.println( "Hello Mark" );
end
```

Speed and scalability
The ReteOO (improved version of the Charles Forgy's Rete algorithm – tailored for object-oriented forward chaining) algorithm provides optimized matching of rules with objects.

Separating the rule logic from the code
Thanks to the Rule engine, it is possible to separate the code from the rule logic. You can keep the business logic in a set of files or decision tables that can be versioned separately.

Rules created in an understandable fashion
Drools provides a series of possibilities for defining rules. The simplest are either the Rule language used in the DRL files (thanks to the use of the mvel dialect that makes them even more user friendly) or DSL (Domain Specific Language), which allows for the creation of rules by means of sentences, or decision tables that can be defined in spreadsheets.

KnowledgeBase – a repository of rules
The KnowledgeBase interface in Drools defines a single point of entry to all rules. It is clear which rules belong to which knowledge base, which makes configuration easier to maintain.

Business Rule Management with Drools

SAMPLE CONFIGURATIONS

Jars

Drools offers the choice of either (a) compiling the rules at runtime or (b) precompiling them in order to minimize the number of runtime libraries. Bearing this in mind, here is the list of the most important libraries for Drools that you can include in your classpath:

Library name	Description
knowledge-api.jar	This contains factories and interfaces. Factories have static methods that provide instance of objects implementing the most crucial interfaces. It also distinguishes the engine API from the user API.
knowledge-internal-api.jar	These are factories and interfaces for the project. The module contains all the internal APIs between Drools, jBPM, and Guvnor. Unlike the knowledge-api, everything in this module is subject to frequent changes.
drools-core.jar	This is required at runtime and contains the core engines (including Rete). The module contains the PackageBuilder interface that is used to assemble binary rule Packages that are then used by the RuleBase and WorkingMemory interfaces of the core engine.
drools-compiler.jar	This library is required to compile the rule sources and to build the proper executable rule base. If you choose to precompile your sources, this library does not need to be present at runtime.
drools-jsr94.jar	This is the JSR94 standard compliant implementation - an abstraction over the drools-compiler component. Because the JSR94 standard does not define the rule language as such, and because it has fewer functionalities than Drools as a framework, it is necessary to make an export of the additional Drools functionalities through property maps. This, however, leads to tight coupling with the Drools API.
drools-decisiontables.jar	This is the compiler for the decision table representation of rules, which depends on the drools-compiler component.

Maven

Drools can be configured using Maven. Only drools-core, with its implicit dependencies, knowledge-api and knowledge-internal-api, is necessary to run the engine.

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-core</artifactId>
  <version>5.5.0.Final</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-compiler</artifactId>
  <version>5.5.0.Final</version>
</dependency>
```

In the code

Once all of the necessary libraries are in the classpath, you can use the KnowledgeBuilder to compile the rules and load them into the repository of rules – the KnowledgeBase. In the following example we will load a single DRL called rules.drl.

```
// Construct a KnowledgeBuilder from the KnowledgeBuilderFactory
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.
newKnowledgeBuilder();

// Add to the KnowledgeBuilder a file rules.drl from the classpath
kbuilder.add( ResourceFactory.newClassPathResource( “rules.drl”,
              getClass() ),
                    ------------Con’t next column----------->>
    ResourceType.DRL );

// Check for errors and print them if necessary
if ( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors().toString() );

}

// Construct the KnowledgeBase from the KnowledgeBaseFactory
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
// add the compiled package to the KnowledgeBase
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

In order to fire the rules that were compiled and put into the KnowledgeBase, you have to create a Stateful or Stateless session, insert some Facts into it, and then perform the execution of the rules as shown in the following example. We assume that the DRL file contained a rule that will switch the ProductType from MEDICAL to ELECTRONIC and vice versa:

```
StatelessKnowledgeSession statelessKnowledgeSession = kbase.
newStatelessKnowledgeSession();
Product medicalProduct = new Product(ProductType.MEDICAL);
Product electronicProduct = new Product(ProductType.ELECTRONIC);

assertTrue(ProductType.MEDICAL == medicalProduct.getType());
assertTrue(ProductType.ELECTRONIC == electronicProduct.
getType());

// Our rule is supposed to change the type of the product from
medical to electronic and vice versa
statelessKnowledgeSession.execute(com.google.common.collect.
Lists.newArrayList(medicalProduct, electronicProduct));

assertTrue(ProductType.ELECTRONIC == medicalProduct.getType());
assertTrue(ProductType.MEDICAL == electronicProduct.getType());
```

Camel

Apache Camel is an open source integration framework. Drools can easily be integrated with it. In this example, we are using the Drools Component together with the TypeConverter. The TypeConverter transforms the input to the Command object, which is used by the Drools Component (this solution can be used if only one type of Command is desired).

```
<?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:camel=”http://camel.apache.org/schema/spring”
       xmlns:context=”http://www.springframework.org/schema/
context”
       xmlns:drools=”http://drools.org/schema/drools-spring”
       xsi:schemaLocation=”http://www.springframework.org/schema/
context http://www.springframework.org/schema/context/spring-
context-3.0.xsd
                           http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd
                           http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring-2.8.0.xsd
                           http://drools.org/schema/drools-spring
http://drools.org/schema/drools-spring.xsd”>

    <!-- Grid Node identifier that is registered in the
CamelContext -->
    <drools:grid-node id=”node1”/>

    <drools:kbase id=”productsKBase” node=”node1”>
        <drools:resources>
            <drools:resource type=”DRL” source=”classpath:rules/
product.drl”/>
        </drools:resources>
    </drools:kbase>

    <drools:ksession id=”productsKSession”
name=”productsKSession” type=”stateless” kbase=”productsKBase”
node=”node1”/>

    <context:component-scan base-package=”pl.grzejszczak.marcin.
drools.decisiontable” />

    <camel:camelContext id=”camelContext”>
        <camel:route id=”discountRoute”>
            <camel:from uri=”direct:discountRoute”/>
            <camel:to uri=”drools:node1/productsKSession”/>
        </camel:route>
    </camel:camelContext>

</beans>
```

Spring

Since Spring IOC is one of the most commonly used IOC frameworks, Drools is designed to easily integrate with the Spring container by using the drools-spring Maven dependency or a respective jar.

```
<?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:drools="http://drools.org/schema/drools-spring"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd
  http://drools.org/schema/drools-spring http://drools.org/
schema/drools-spring.xsd">

    <!-- KNOWLEDGE BASE FOR A GIVEN TYPE -->
    <drools:kbase id="productsKBase">
        <drools:resources>
            <drools:resource type="DRL"
                             source="classpath:rules/products.
drl"/>
        </drools:resources>
    </drools:kbase>

    <drools:ksession id="productsKSession"
name="productsKSession" type="stateless" kbase="productsKBase"/>
    <alias name="productsKSession" alias="electronicKSession"/>
    <alias name="productsKSession" alias="medicalKSession"/>

</beans>
```

Drools Server

If the number of processed rules is substantial, a dedicated machine that remotely executes the rule logic compiled in the KnowledgeBases may be required. In such cases, the Drools Execution Server is deployed in an application server. It uses Spring and Camel to route the inbound XML Command messages to the proper node and Knowledge Session. CXF as the services framework listens to REST/SOAP requests.

In the default configuration, sending a POST message goes like this:

```
<batch-execution>
   <insert out-identifier='outMessage'>
          <org.test.Message>
                    <text>SOME TEXT</text>
          </org.test.Message>
   </insert>
</batch-execution
```

To the URI: ${drools_server_app_uri}/kservice/rest/execute gives this result:

```
<?xml version='1.0' encoding='UTF-8'?>
<execution-results>
          <result identifier="outMessage">
                    <org.test.Message>
                              <text>echo:SOME TEXT</text>
                    </org.test.Message>
          </result>
          <fact-handle identifier="outMessage" external-form="0:2:
10868722:10868722:2:DEFAULT"/>
</execution-results>
```

RULE LANGUAGE ESSENTIALS

DRL file structure

Drools has a native language called Drools Rule Language (DRL). A DRL file is a simple text file in which you specify a package name, multiple rules, queries, types, functions, and resource declarations such as imports and globals. The order in which these elements are declared is not important except for the package name that, if declared, must be the first element in the rules file. All elements are optional.

Rule structure

A rule has the following general structure:

```
rule "name"
    attributes
    when
        LHS
    then
        RHS
end
```

Punctuation is generally unnecessary, even the double quotes for "name" and newlines are optional. Attributes are simple optional hints as to how the rule should behave. **LHS** (Left Hand Side) is the **conditional** part of the rule, and **RHS** (Right Hand Side) is basically a block that specifies the dialect-specific **semantic (action) code to be executed**.

Rule Attributes

Rule attributes are a declarative way to influence the behavior of the rule engine.

Name	Type	Default	Description
no-loop	Boolean	false	When a rule's consequence modifies a fact, it may cause the rule to activate again, thus causing an infinite loop. Setting no-loop to true will skip the creation of another Activation for the rule with the current set of facts.
ruleflow-group	String	N/A	Rules having the same ruleflow-group identifier fire only when their group is active.
lock-on-active	Boolean	false	Whenever a ruleflow-group becomes active or an agenda-group receives the focus, any rule within that group that has lock-on-active set to true will not be activated any more.
salience	Integer	0	Salience is a form of priority where rules with higher salience values are given higher priority when ordered by the conflict resolution strategy.
agenda-group	Boolean	false	If auto-focus is true and the rule's agenda group does not have focus yet, when a rule is activated, then it is given focus, thus allowing the rule to potentially fire.
activation-group	String	N/A	Only one rule in the each activation-group will fire i.e., the first rule in an activation-group to fire cancels the other rules' activations.
dialect	String	as specified by the package	The dialect specifies the language to be used for any code expressions in the LHS or the RHS code block. Currently, two dialects are available: Java and MVEL.
date-effective	Date	N/A	A rule can only activate after the date and time of the date-effective attribute.
date-expires	Date	N/A	A rule cannot activate after the date-expires attribute.
duration	Long	N/A	The duration dictates that the rule will fire after a specified duration if it is still true.

Timers

Rules supports both interval- and cron-based timers. Interval (indicated by "int:") timers follow the semantics of java.util.Timer objects, with an initial delay and an optional repeat interval:

```
timer ( int: <initial delay> <repeat interval>? )
```

Cron (indicated by "cron:") timers follow standard Unix cron expressions. For example:

```
rule "Send SMS every 15 minutes" timer (cron:* 0/15 * * * ?)
when
    $a : Alarm( on == true )
then
    channels["sms"].insert( new Sms( $a.mobileNumber, "Alarm on"
);
end
```

The Left Hand Side

The Left Hand Side (LHS) is the conditional part of the rule. It consists of zero or more Conditional Elements. If the LHS is empty, it is considered always true and is activated once, when a new session is created. A pattern is the most important Conditional Element. It can potentially match on any fact inserted in the working memory. A pattern contains zero or more constraints and has an optional pattern binding, meaning that the object used in a pattern, or some of its attributes, can be bound to variables that can then be used in the following patterns or in the consequence.

In its simplest form with no constraints, a pattern matches against a fact of the given type:

```
Person() // match all objects of type Person in the WorkingMemory
```

Patterns may refer to superclasses or even interfaces, thereby potentially matching facts from many different classes:

```
Object() // match all objects in the WorkingMemory
```

A comma-separated list of constraints with an implicit AND connective semantic can be added between the pattern parenthesis:

```
Person( name == "Mark", age > 30 ) //named Mark AND older than 30
```

It is possible to bind the matched object, or the value of one of its properties, to a variable in order to refer to that variable in other subsequent patterns or in the RHS:

```
$p : Person( $name : name, $age : age > 30 )
```

The prefixed dollar symbol ($) is just a convention: it can be useful in complex rules where it helps to easily differentiate between variables and fields, but it is not mandatory.

Patterns can be combined into logical conjunctions (AND) and/or disjunctions (OR). Drools supports both infix form:

```
A() or ( B() and C() )
```

and prefix form:

```
or( A() and( B() C() ) )
```

Patterns can also be modified by prepending them with a first order logic existential (exists) and non-existential (not) quantifier. While the function of the not quantifier is obvious ("there must be none of..."), the existential quantifier might appear redundant. But it is not redundant, because it has the meaning of "there is at least one..." while the pattern on its own can be thought as "for each one of..."

Advanced Conditional Elements

Any LHS pattern is a conditional element. Drools provides some advanced conditional elements that greatly enrich its semantics.

for all

The conditional element forall completes the first order logic support in Drools: it evaluates to true when all facts that match the first pattern match all the remaining patterns. For example, to check that all the English buses are red, write:

```
forall( $bus : Bus( type == "English" )
              Bus( this == $bus, color = "red" ) )
```

from

The conditional element from enables users to specify an arbitrary source for data to be matched by LHS patterns. This allows the engine to reason over data not in the Working Memory. The data source could be the results of a method call or a sub-field on a bound variable, for example:

```
Person( $personAddress : address )
Address( zipcode == "23920W") from $personAddress
```

collect

The conditional element collect allows rules to reason over a collection of objects obtained from the given source or from the working memory. In terms of first order logic, this is the cardinality quantifier. For example, to check if a system has 3 pending alarms or more:

```
$s : System()
$alarms : ArrayList( size >= 3 )
          from collect( Alarm( system == $s, status == 'pending'
) )
```

accumulate

The conditional element accumulate is a more flexible and powerful form of collect because it allows a rule to iterate over a collection of objects by executing custom actions for each of the elements, and at the end it returns a result object. accumulate supports both the use of pre-defined accumulate functions and the use of inline custom code. The top level accumulate syntax is:

```
accumulate ( <source pattern>; <functions> [;<constraints>] )
```

For instance, a pattern to calculate the minimum, maximum and average temperature reading for a given sensor and to match if the minimum temperature is under 20C degrees and the average is over 70C degrees could be written in the following way:

```
$s : Sensor()
accumulate( Reading( sensor == $s, $temp : temperature );
            $min : min( $temp ),
            $max : max( $temp ),
            $avg : average( $temp );
            $min < 20, $avg > 70 )
```

eval

In the end, the conditional element eval is essentially a catch-all that allows any semantic code that returns a boolean to be executed. eval can refer to variables that were bound in the LHS, and it functions in the rule package. Still, it should be used sparingly because it reduces the declarativeness of your rules and can result in a poorly performing engine.

The Right Hand Side

The Right Hand Side (RHS) contains the list of actions to be executed when the rule is fired. It should be kept small and avoid any conditional code, thus keeping it declarative and readable. It allows any Java or mvel code (depending on the chosen dialect), but its main purpose is to insert, retract, or modify Working Memory data and to interact with the engine. To do that, Drools provides the following convenience methods:

Method	Description
update(object)	Informs the rule engine that an object has changed, and then the LHS potentially interesting to it has to be reconsidered
insert(object)	Inserts a new object in the Working Memory
insertLogical(object)	Similar to insert, but the object will be automatically retracted when there are no more facts to support the truth of the currently firing rule
retract(object)	Retracts the object from the Working Memory
drools.halt()	Immediately terminates a rule execution
drools.setFocus(string)	Sets the focus to the specified agenda group
drools.getRule().getName()	Returns the name of the fired rule

As an alternative to the update method, Drools provides the **modify** statement that is a more structured approach to notify the engine that the state of an object is changed and how it is changed. The syntax of this

statement is:

```
modify ( <fact-expression> ) { <expression> [,<expression> ]* }
```

For example, this example illustrates a simple fact modification:

```
modify( $person ) { setName( “Mark” ), setAge( 35 ) }
```

Package

A package represents a namespace that groups a set of rules. The package name itself is the namespace, and it is not related to files or folders in any way. Its declaration, if present, must be the first statement in the DRL file, for example:

```
package org.mydomain.mypackage
```

Functions

Functions are a way to put semantic code in a rule source file, alternative and equivalent to static Java methods in a helper class. Indeed, the main advantage of using functions in a rule is that you can keep the logic all in one place. A typical function declaration looks like this:

```
function String hello(String name) {
    return “Hello “+name+”!”;
}
```

Type declarations

Despite Drools' working out of the box with plain Java objects as facts, it also allows the declaration of new types. To do that, it is enough to use the declare keyword followed by the name of the type and the list of its attributes with the respective types as follows:

```
declare Person
    name : String
    age  : int
end
```

Query

A query is a simple way to search the working memory for facts that match the stated conditions and, therefore, it contains only the structure of the LHS of a rule as in the following example:

```
query “over 30”
    person : Person( age > 30 )
end

It is then possible to iterate over the query results, as in:
QueryResults results = ksession.getQueryResults( “over 30” );
for ( QueryResultsRow row : results ) {
    Person person = ( Person ) row.get( “person” );
}
```

DECISION TABLES

Decision tables are a way of modeling logic simply and clearly. Business Analysts often prefer Decision Tables because the rules, both conditions and actions, can be modified more easily and without having to know DRL, Java, or MVEL.

Advantages of Decision Tables

- Clear and compact presentation using tools known by the non-technical staff
- Need to split more complex problems into a set of simpler ones, thus leading to enhanced readability
- Decision tables represent real-life business problems
- Easy navigation over even the most complex set of conditions

Disadvantages of Decision Tables

- It is hard to implement more complex rules (the conditions and actions should be straightforward)
- Adding a condition may force a series of changes to the graphical design of the decision table
- Since decision tables are binary files, it is difficult to version them
- It is difficult to perform fast customization/modification of the rules

When to use Decision Tables

When to use	When to not use
Rules follow a set of templates and patterns.	The conditions and actions do not follow a set of templates. The logic is complicated and impossible to categorize.
You want to limit the values placed in conditions and actions.	You want to have custom implementations of rules.
Spreadsheets are already used in the company as a source of rule definition. It may be possible to transform the existing ones into decision tables or to create new ones on the basis of the existing ones.	Your team is not keen to keep rule logic in spreadsheets.
The versioning of the spreadsheet stored in a binary format is not problematic for you.	You want to keep your rules in non-binary files in order to have problem-free versioning.

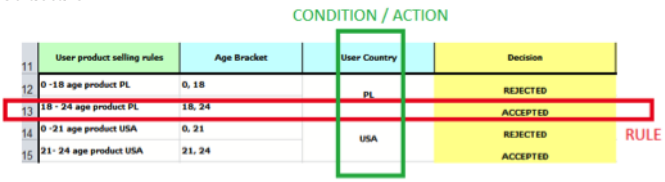
Configuration

Dependencies

In order to use the JBoss Drools decision tables, you have to add the drools-decisiontables dependency, which contains all the necessary jars to use decision tables and drools as such. In Maven, we would add such a dependency in the following manner:

```
<dependency>
          <groupId>org.drools</groupId>
          <artifactId>drools-decisiontables</artifactId>
          <version>5.5.0.Final</version>
</dependency>
```

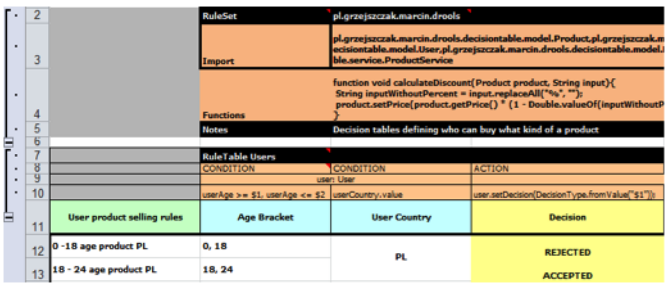
Structure



Each row represents a single rule, whereas each column is either an action, condition, or parameter.

Decision table configuration

All decision table configuration is written in hidden cells to prevent inadvertent edits by rule developers.



Row No	Description
2	The optional RuleSet keyword defines the rule package.
3	Values for the Import keyword define the classes to be imported by the decision table.
4	Values for the Function keyword define functions that can be executed in the rules.
5	Any other keywords regarding decision table configuration should be entered in a key-value fashion, where key should be placed in the same column (all the values in columns to the left will be ignored).
7	The RuleTable (followed by the name that is then used as a prefix for the names of the generated rules) keyword defines where the rules start.
8	The keywords CONDITION and ACTION indicate whether the data in the columns below refer to either the LHS or the RHS of a rule. You can define some other attributes in the same manner.
9	The next row is optional and contains declarations of ObjectTypes. When used, the values in the cell below it are considered to be of this type.
10	The subsequent row is a rule as such. The "&param" keyword refers to the value in the cell. If there is a comma-separated list in the cell, then in order to access a given element one has to use "$1", "$2", etc.
11	Next, there is a textual representation of the condition or action (it gets ignored by the compiler).
12	Subsequent rows contain data that combines with the rule templates to generate rules. If no data is inserted, then it is ignored.

DO'S

- **If possible, use listeners and auditing instead of loggers in each rule.**
- **Try to make rules as small as possible.**
- **Include common code that will not change to static methods, and import them in the rules.**
- **Put common code that is prone to changes in functions.**
- **Remember that == operator for strings is null safe but is case sensitive.**

DONT'S

- **Don't add nested ifs. Split ifs into separate rules.**
- **Don't use property accessors in the LHS that change the state of the object.**
- **Don't change the state of a fact without notifying the rule engine.**
- **Don't use any blocking operation in the RHS.**
- **Don't put all rules in one file - divide it into subfiles and load it into the KnowledgeBase.**

Websites

JBoss Drools Expert homepage:
http://www.jboss.org/drools/drools-expert.html

Mailing Lists

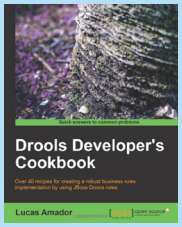
JBoss Mailing Lists:
http://www.jboss.org/drools/lists

ABOUT THE AUTHOR

Marcin (@MGrzejszczak) is a Java software developer who is passionate about his work and enjoys solving technical issues and exploring new approaches of application development. He is fond of clean code and is an enthusiast of JVM based languages. Marcin is a committer to the JBoss Drools project. Contact Marcin at marcin@grzejszczak.pl or read his blog at http://toomuchcoding.blogspot.com

Mario (@mariofusco) is a senior software engineer at Red Hat working at the development of the core of Drools. He has a great deal of experience as Java developer, having been involved in (and often leading) many enterprise-level projects in several industries ranging from media companies to the financial sector. His interests also include functional programming and Domain Specific Languages.

RECOMMENDED BOOK



JBoss Drools is an open source business rules engine that provides agility and flexibility to your business logic. Drools 5 has evolved to provide a unified and integrated platform for business rules, business processes, event processing and automated planning. With this book in hand you will be able to use any of these modules and their specific features quickly and with ease.

Buy Here

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1
50795
9 781936 502806
$7.95

Version 1.0