



Red Hat Decision Manager 7.0

Designing a decision service using DRL rules

Red Hat Decision Manager 7.0 Designing a decision service using DRL rules

Red Hat Customer Content Services
brms-docs@redhat.com

Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document describes how to design a decision service using DRL rules in Red Hat Decision Manager 7.0.

Table of Contents

PREFACE	3
CHAPTER 1. RULE-AUTHORING ASSETS IN RED HAT DECISION MANAGER	4
CHAPTER 2. DRL RULES	6
CHAPTER 3. DATA OBJECTS	7
3.1. CREATING DATA OBJECTS	7
CHAPTER 4. CREATING DRL FILES IN DECISION CENTRAL	9
4.1. ADDING WHEN CONDITIONS IN DRL RULES	12
4.2. ADDING THEN ACTIONS IN DRL RULES	16
4.2.1. Rule attributes	17
CHAPTER 5. OTHER METHODS FOR CREATING DRL FILES	20
5.1. CREATING DRL FILES IN RED HAT JBOSS DEVELOPER STUDIO	20
5.2. CREATING DRL FILES USING JAVA	23
5.3. CREATING DRL FILES USING MAVEN	26
CHAPTER 6. NEXT STEPS	32
APPENDIX A. VERSIONING INFORMATION	33

PREFACE

As a business rules developer, you can define business rules using the DRL (Drools Rule Language) designer in Decision Central. DRL rules are defined directly in free-form `.drl` text files instead of in a guided or tabular format like other types of rule assets in Decision Central. These DRL files form the core of the decision service for your project.

Prerequisite

The team and project for the DRL rules have been created in Decision Central. Each asset is associated with a project assigned to a team. For details, see [Getting started with decision services](#).

CHAPTER 1. RULE-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager provides several assets that you can use to create business rules for your decision service. Each rule-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights each rule-authoring asset in Decision Central to help you decide or confirm the best method for creating rules in your decision service.

Table 1.1. Rule-authoring assets in Decision Central

Asset	Highlights	Documentation
Guided decision tables	<ul style="list-style-type: none"> • Are tables of rules that you create in a UI-based table designer in Decision Central • Are a wizard-led alternative to uploaded decision table spreadsheets • Provide fields and options for acceptable input • Support template keys and values for creating rule templates • Support hit policies, real-time validation, and other additional features not supported in other assets • Are optimal for creating rules in a controlled tabular format to minimize compilation errors 	Designing a decision service using guided decision tables
Uploaded decision tables	<ul style="list-style-type: none"> • Are XLS or XLSX decision table spreadsheets that you upload into Decision Central • Support template keys and values for creating rule templates • Are optimal for creating rules in decision tables already managed outside of Decision Central • Have strict syntax requirements for rules to be compiled properly when uploaded 	Designing a decision service using uploaded decision tables

Asset	Highlights	Documentation
Guided rules	<ul style="list-style-type: none"> • Are individual rules that you create in a UI-based rule designer in Decision Central • Provide fields and options for acceptable input • Are optimal for creating single rules in a controlled format to minimize compilation errors 	Designing a decision service using guided rules
Guided rule templates	<ul style="list-style-type: none"> • Are reusable rule structures that you create in a UI-based template designer in Decision Central • Provide fields and options for acceptable input • Support template keys and values for creating rule templates (fundamental to the purpose of this asset) • Are optimal for creating many rules with the same rule structure but with different defined field values 	Designing a decision service using guided rule templates
DRL rules	<ul style="list-style-type: none"> • Are individual rules that you define directly in <code>.drl</code> text files • Provide the most flexibility for defining rules and other technicalities of rule behavior • Can be created in certain standalone environments and integrated with Red Hat Decision Manager • Are optimal for creating rules that require advanced DRL options • Have strict syntax requirements for rules to be compiled properly 	Designing a decision service using DRL rules

CHAPTER 2. DRL RULES

DRL rules are business rules that you define directly in `.drl` text files. These DRL files are the source in which all other rule assets in Decision Central are ultimately rendered. You can create and manage DRL files within the Decision Central interface, or create them externally using Red Hat Developer Studio, Java objects, or Maven archetypes. A DRL file can contain one or more rules that define at minimum the rule conditions (**when**) and actions (**then**). The DRL designer in Decision Central provides syntax highlighting for Java, DRL, and XML.

All data objects related to a DRL rule must be in the same project package as the DRL rule in Decision Central. Assets in the same package are imported by default. Existing assets in other packages can be imported with the DRL rule.

CHAPTER 3. DATA OBJECTS

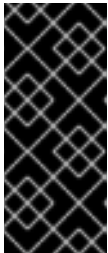
Data objects are the building blocks for the rule assets that you create. Data objects are custom data types implemented as Java objects in specified packages of your project. For example, you might create a **Person** object with data fields **Name**, **Address**, and **Date of Birth** to specify personal details for loan application rules. These custom data types determine what data your assets and your decision service are based on.

3.1. CREATING DATA OBJECTS

The data objects that you define are the building blocks for rule assets in your project and determine what data your assets and your decision service are based on.

Procedure

1. Go to **Menu** → **Design** → **Projects** and click the project name.
2. Click **Create New Asset** → **Data Object**.
3. Enter a unique **Data Object** name and select the **Package** where you want the data object to be available for other rule assets. Data objects with the same name cannot exist in the same package. The package that you specify must be the same package where the rule assets that require those data objects have been assigned or will be assigned.

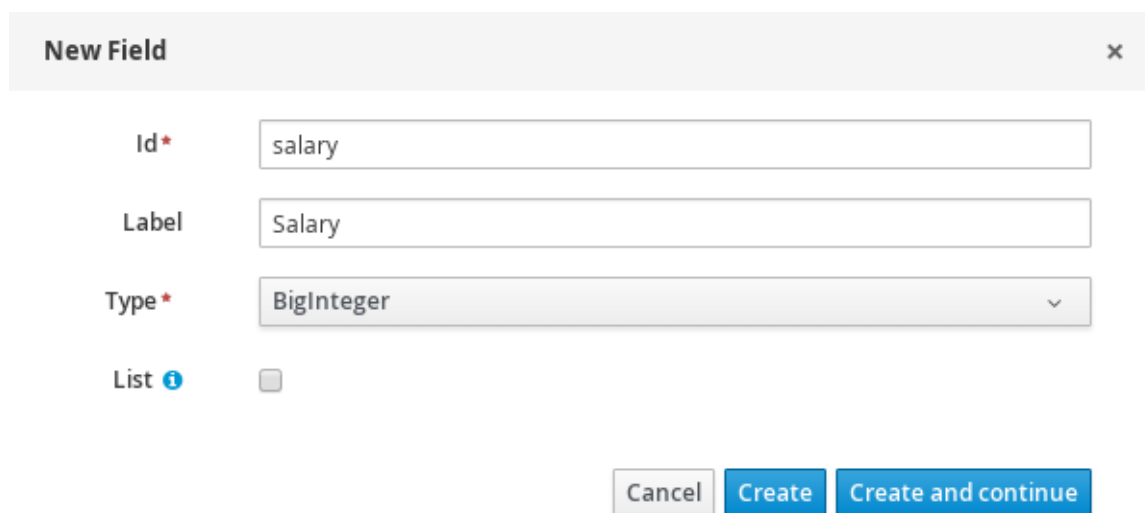


IMPORTING DATA OBJECTS FROM OTHER PACKAGES

You can also import an existing data object from another package into the package of the rule asset. In the **Project Explorer**, expand the asset panel (such as for guided decision tables or guided rules), select the specific asset, and in the asset designer, go to **Data Objects** → **New item** to select the object to be imported.

4. To make your data object persistable, select the **Persistable** checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.
5. Click **Ok**.
6. In the data object designer, click **add field** to add a field to the object with the attributes **Id**, **Label**, and **Type**. Required attributes are marked with an asterisk (*).
 - **Id**: Enter the unique ID of the field.
 - **Label**: (Optional) Enter a label for the field.
 - **Type**: Enter the data type of the field.
 - **List**: Select this check box to enable the field to hold multiple items for the specified type.

Figure 3.1. Add data fields to a data object



New Field x

Id*

Label

Type*

List i

7. Click **Create** to add the new field, or click **Create and continue** to add the new field and continue adding other fields.

**NOTE**

To edit a field, select the field row and use the **general properties** on the right side of the screen.

CHAPTER 4. CREATING DRL FILES IN DECISION CENTRAL

You can create and manage DRL files for your project in Decision Central. In each DRL file, you define rule conditions, actions, and other components related to the rule, based on the data objects you create or import in the package.

Procedure

1. Go to **Menu** → **Design** → **Projects** and click the project name.
2. Click **Create New Asset** → **DRL file**.
3. Enter an informative **DRL file** name and select the appropriate **Package**. The package that you specify must be the same package where the required data objects have been assigned or will be assigned.
You can also select **Use Domain Specific Language (DSL)** if any DSL assets have been defined in your project (in the **Domain Specific Language Definitions** panel in the **Project Explorer**). These DSL assets will then become usable objects for conditions and actions that you define in the DRL designer.
4. Click **Ok** to create the rule asset.
The new DRL file is now listed in the **DRL** panel of the **Project Explorer**, or in the **DSL** panel if you selected the **Use Domain Specific Language (DSL)** option. The package to which you assigned this DRL file is listed at the top of the file.
5. In the **Fact types** list in the left panel of the DRL designer, confirm that all data objects and data object fields (expand each) required for your rules are listed. If not, you can either import relevant data objects from other packages by using **import** statements in the DRL file, or [create data objects](#) within your package.
6. After all data objects are in place, return to the **Editor** tab of the DRL designer and define the DRL file with any of the following components:

Components of a DRL file

```
package //automatic

import

function //optional

query //optional

declare //optional

rule

rule

...
```

- **package:** (automatic) This was defined for you when you created the DRL file and selected the package.

- **import**: Use this to identify the data objects from either this package or another package that you want to use in the DRL file. Specify the package and data object in the format `package.name.object.name`, one import per line.

Importing data objects

```
import mortgages.mortgages.LoanApplication;
```

- **function**: (optional) Use this to include a function to be used by rules in the DRL file. Functions put semantic code in your rule source file. Functions are especially useful if an action (**then**) part of a rule is used repeatedly and only the parameters differ for each rule. Above the rules in the DRL file, you can declare the function or import a static method as a function, and then use the function by name in an action (**then**) part of the rule.

Declaring and using a function with a rule (option 1)

```
function String hello(String applicantName) {
    return "Hello " + applicantName + "!";
}

rule "Using a function"
    when
        eval( true )
    then
        System.out.println( hello( "James" ) );
    end
```

Importing and using the function with a rule (option 2)

```
import function my.package.applicant.hello;

rule "Using a function"
    when
        eval( true )
    then
        System.out.println( hello( "James" ) );
    end
```

- **query**: (optional) Use this to search the decision engine for facts related to the rules in the DRL file. Queries search for a set of defined conditions and do not require **when** or **then** specifications. Query names are global to the KIE base and therefore must be unique among all other rule queries in the project. To return the results of a query, construct a traditional **QueryResults** definition using `ksession.getQueryResults("name")`, where **"name"** is the query name. This returns a list of query results, which enable you to retrieve the objects that matched the query. Define the query and query results parameters above the rules in the DRL file.

Query and query results for people under the age of 21, with a rule

```
query "people under the age of 21"
    person : Person( age < 21 )
end

QueryResults results = ksession.getQueryResults( "people under
```

```

the age of 21" );
System.out.println( "we have " + results.size() + " people under
the age of 21" );

System.out.println( "These people are are under 21:" );

rule "Underage"
  when
    application : LoanApplication( )
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
  end

```

- **declare:** (optional) Use this to declare a new fact type to be used by rules in the DRL file. The default fact type in the `java.lang` package of Red Hat Decision Manager is **Object**, but you can declare other types in DRL files as needed. Declaring fact types in DRL files enables you to define a new fact model directly in the decision engine, without creating models in a lower-level language like Java.

Declaring and using a new fact type

```

declare Person
  name : String
  dateOfBirth : java.util.Date
  address : Address
end

rule "Using a declared type"
  when
    $p : Person( name == "James" )
  then // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
  end

```

- **rule:** Use this to define each rule in the DRL file. Rules consist of a rule name in the format **rule "name"**, followed by optional attributes that define rule behavior (such as **salience** or **no-loop**), followed by **when** and **then** definitions. The same rule name cannot be used more than once in the same package. The **when** part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition for an "Underage" rule would be **Applicant(age < 21)**. The **then** part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when the loan applicant is under 21 years old, the **then** action would be **setApproved(false)**, declining the loan because the applicant is under age. Conditions (**when**) and actions (**then**) consist of a series of stated fact patterns with optional constraints, bindings, and other supported DRL elements, based on the available data objects in the package. These patterns determine how defined objects are affected by the rule.

Rule for loan application age limit

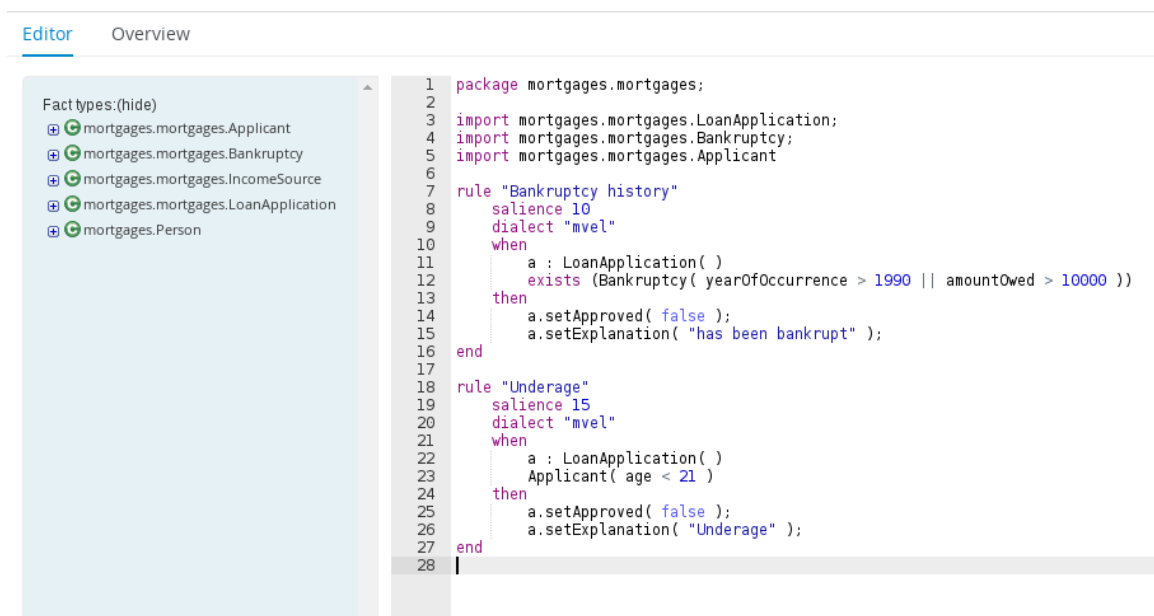
```

rule "Underage"
  salience 15
  dialect "mvel"
  when
    application : LoanApplication( )
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
  end
end

```

At minimum, each DRL file must specify the **package**, **import**, and **rule** components. All other components are optional.

Figure 4.1. Sample DRL file with required components and optional rule attributes



7. After you define all components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.

8. Click **Save** in the DRL designer to save your work.

For more details about adding conditions to DRL rules, see [Section 4.1, “Adding WHEN conditions in DRL rules”](#).

For more details about adding actions to DRL rules, see [Section 4.2, “Adding THEN actions in DRL rules”](#).

4.1. ADDING WHEN CONDITIONS IN DRL RULES

The **when** part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition of an "Underage" rule would be **Applicant(age < 21)**. Conditions consist of a series of stated patterns and constraints, with optional bindings and other supported DRL elements, based on the available data objects in the package.

Prerequisites

- The **package** is defined at the top of the DRL file. This should have been done for you when you created the file.
- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Decision Central.
- The **rule** name is defined in the format **rule "name"** below the **package**, **import**, and other lines that apply to the entire DRL file. The same rule name cannot be used more than once in the same package. Optional rule attributes (such as **salience** or **no-loop**) that define rule behavior are below the rule name, before the **when** section.

Procedure

1. In the DRL designer, enter **when** within the rule to begin adding condition statements. The **when** section consists of zero or more fact patterns that define conditions for the rule. If the **when** section is empty, then actions in the **then** section are executed every time a **fireAllRules()** call is made in the decision engine. This is useful if you want to use rules to set up the decision engine state.

Rule without conditions

```
rule "bootstrap"
  when // empty

  then // actions to be executed once
    insert( new Applicant() );
  end

// The above rule is internally rewritten as:

rule "bootstrap"
  when
    eval( true )
  then
    insert( new Applicant() );
  end
```

2. Enter a pattern for the first condition to be met, with optional constraints, bindings, and other supported DRL elements. A basic pattern format is **patternBinding : patternType (constraints)**. Patterns are based on the available data objects in the package and define the conditions to be met in order to trigger actions in the **then** section.

- **Simple pattern:** A simple pattern with no constraints matches against a fact of the given type. For example, the following condition is only that the applicant exists.

```
when
  Applicant( )
```

- **Pattern with constraints:** A pattern with constraints matches against a fact of the given type and the additional restrictions in parentheses that are true or false. For example, the following condition is that the applicant is under the age of 21.

```
when
    Applicant( age < 21 )
```

- **Pattern with binding:** A binding on a pattern is a shorthand reference that other components of the rule can use to refer back to the defined pattern. For example, the following binding **a** on **LoanApplication** is used in a related action for underage applicants.

```
when
    a : LoanApplication( )
    Applicant( age < 21 )
then
    a.setApproved( false );
    a.setExplanation( "Underage" )
```

3. Continue defining all condition patterns that apply to this rule. The following are some of the keyword options for defining DRL conditions:

- **and:** Use this to group conditional components into a logical conjunction. Infix and prefix **and** are supported. By default, all listed conditions or actions are combined with **and** when no conjunction is specified.

```
a : LoanApplication( ) and Applicant( age < 21 )

a : LoanApplication( )
and Applicant( age < 21 )

a : LoanApplication( )
Applicant( age < 21 )

// All of the above are the same.
```

- **or:** Use this to group conditional components into a logical disjunction. Infix and prefix **or** are supported.

```
( Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount == 20000 ) )

Bankruptcy( amountOwed == 100000 )
or IncomeSource( amount == 20000 )
```

- **exists:** Use this to specify facts and constraints that must exist. Note that this does not mean that a fact exists, but that a fact must exist. This option is triggered on only the first match, not subsequent matches.

```
exists (Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000
))
```

- **not:** Use this to specify facts and constraints that must not exist.

```
not (Applicant( age < 21 ))
```

- **forall**: Use this to set up a construct where all facts that match the first pattern match all the remaining patterns.

```
forall( app : Applicant( age < 21 )
        Applicant( this == app, status = 'underage' ) )
```

- **from**: Use this to specify a source for data to be matched by the conditional pattern.

```
Applicant( ApplicantAddress : address )
Address( zipcode == "23920W" ) from ApplicantAddress
```

- **entry-point**: Use this to define an **Entry Point** corresponding to a data source for the pattern. Typically used with **from**.

```
Applicant( ) from entry-point "LoanApplication"
```

- **collect**: Use this to define a collection of objects that the construct can use as part of the condition. In the example, all pending applications in the decision engine for each given mortgage are grouped in **ArrayLists**. If three or more pending applications are found, the rule is executed.

```
m : Mortgage()
a : ArrayList( size >= 3 )
    from collect( LoanApplication( Mortgage == m, status ==
'pending' ) )
```

- **accumulate**: Use this to iterate over a collection of objects, execute custom actions for each of the elements, and return one or more result objects (if the constraints evaluate to **true**). This option is a more flexible and powerful form of **collect**. Use the format **accumulate(<source pattern>; <functions> [<constraints>])**. In the example, **min**, **max**, and **average** are accumulate functions that calculate the minimum, maximum and average temperature values over all the readings for each sensor. Other supported functions include **count**, **sum**, **variance**, **standardDeviation**, **collectList**, and **collectSet**.

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



ADVANCED DRL OPTIONS

These are examples of basic keyword options and pattern constructs for defining conditions. For more advanced DRL options and syntax supported in the DRL designer, visit the [Drools Documentation](#) online.

4. After you define all condition components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.

5. Click **Save** in the DRL designer to save your work.

4.2. ADDING THEN ACTIONS IN DRL RULES

The **then** part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when a loan applicant is under 21 years old, the **then** action of an "Underage" rule would be `setApproved(false)`, declining the loan because the applicant is under age. Actions execute consequences based on the rule conditions and on available data objects in the package.

Prerequisites

- The **package** is defined at the top of the DRL file. This should have been done for you when you created the file.
- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Decision Central.
- The **rule** name is defined in the format **rule "name"** below the **package**, **import**, and other lines that apply to the entire DRL file. The same rule name cannot be used more than once in the same package. Optional rule attributes (such as **salience** or **no-loop**) that define rule behavior are below the rule name, before the **when** section.

Procedure

1. In the DRL designer, enter **then** after the **when** section of the rule to begin adding action statements.
2. Enter one or more actions to be executed on fact patterns based on the conditions for the rule. The following are some of the keyword options for defining DRL actions:
 - **and**: Use this to group action components into a logical conjunction. Infix and prefix **and** are supported. By default, all listed conditions or actions are combined with **and** when no conjunction is specified.

```
application.setApproved ( false ) and application.setExplanation(
"has been bankrupt" );

application.setApproved ( false );
and application.setExplanation( "has been bankrupt" );

application.setApproved ( false );
application.setExplanation( "has been bankrupt" );

// All of the above are the same.
```

- **set**: Use this to set the value of a field.

```
application.setApproved ( false );
application.setExplanation( "has been bankrupt" );
```

- **modify**: Use this to specify fields to be modified for a fact and to notify the decision engine of the change.

```
modify( LoanApplication ) {
```

```
setAmount( 100 )
}
```

- **update**: Use this to specify fields and the entire related fact to be modified and to notify the decision engine of the change. After a fact has changed, you must call **update** before changing another fact that might be affected by the updated values. The **modify** keyword avoids this added step.

```
update( LoanApplication ) {
    setAmount( 100 )
}
```

- **delete**: Use this to remove an object from the decision engine. The keyword **retract** is also supported in the DRL designer and executes the same action, but **delete** is preferred for consistency with the keyword **insert**.

```
delete( LoanApplication );
```

- **insert**: Use this to insert a **new** fact and define resulting fields and values as needed for the fact.

```
insert( new Applicant() );
```

- **insertLogical**: Use this to insert a **new** fact logically into the decision engine and define resulting fields and values as needed for the fact. The Red Hat Decision Manager decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts have to be retracted explicitly. After logical insertions, facts are automatically retracted when the conditions that originally asserted the facts are no longer true.

```
insertLogical( new Applicant() );
```



ADVANCED DRL OPTIONS

These are examples of basic keyword options and pattern constructs for defining actions. For more advanced DRL options and syntax supported in the DRL designer, visit the [Drools Documentation](#) online.

3. After you define all action components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.
4. Click **Save** in the DRL designer to save your work.

4.2.1. Rule attributes

Rule attributes are additional specifications that you can add to business rules to modify rule behavior. The following table lists the names and supported values of the attributes that you can assign to rules:

Table 4.1. Rule attributes

Attribute	Value
salience	<p>An integer defining the priority of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue.</p> <p>Example: salience 10</p>
enabled	<p>A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled.</p> <p>Example: enabled true</p>
date-effective	<p>A string containing a date and time definition. The rule can be activated only if the current date and time is after a date-effective attribute.</p> <p>Example: date-effective "4-Sep-2018"</p>
date-expires	<p>A string containing a date and time definition. The rule cannot be activated if the current date and time is after the date-expires attribute.</p> <p>Example: date-expires "4-Oct-2018"</p>
no-loop	<p>A Boolean value. When the option is selected, the rule cannot be reactivated (looped) if a consequence of the rule re-triggers a previously met condition. When the condition is not selected, the rule can be looped in these circumstances.</p> <p>Example: no-loop true</p>
agenda-group	<p>A string identifying an agenda group to which you want to assign the rule. Agenda groups allow you to partition the agenda to provide more execution control over groups of rules. Only rules in an agenda group that has acquired a focus are able to be activated.</p> <p>Example: agenda-group "GroupName"</p>
activation-group	<p>A string identifying an activation (or XOR) group to which you want to assign the rule. In activation groups, only one rule can be activated. The first rule to fire will cancel all pending activations of all rules in the activation group.</p> <p>Example: activation-group "GroupName"</p>
duration	<p>A long integer value defining the duration of time in milliseconds after which the rule can be activated, if the rule conditions are still met.</p> <p>Example: duration 10000</p>
timer	<p>A string identifying either int (interval) or cron timer definition for scheduling the rule.</p> <p>Example: timer "*/5 * * * *" (every 5 minutes)</p>

Attribute	Value
calendar	<p>A Quartz calendar definition for scheduling the rule.</p> <p>Example: calendars "*" * 0-7,18-23 ? * * (exclude non-business hours)</p>
auto-focus	<p>A Boolean value, applicable only to rules within agenda groups. When the option is selected, the next time the rule is activated, a focus is automatically given to the agenda group to which the rule is assigned.</p> <p>Example: auto-focus true</p>
lock-on-active	<p>A Boolean value, applicable only to rules within rule flow groups or agenda groups. When the option is selected, the next time the ruleflow group for the rule becomes active or the agenda group for the rule receives a focus, the rule cannot be activated again until the ruleflow group is no longer active or the agenda group loses the focus. This is a stronger version of the no-loop attribute, because the activation of a matching rule is discarded regardless of the origin of the update (not only by the rule itself). This attribute is ideal for calculation rules where you have a number of rules that modify a fact and you do not want any rule re-matching and firing again.</p> <p>Example: lock-on-active true</p>
ruleflow-group	<p>A string identifying a rule flow group. In rule flow groups, rules can fire only when the group is activated by the associated rule flow.</p> <p>Example: ruleflow-group "GroupName"</p>
dialect	<p>A string identifying either JAVA or MVEL as the language to be used for code expressions in the rule. By default, the rule uses the dialect specified at the package level. Any dialect specified here overrides the package dialect setting for the rule.</p> <p>Example: dialect "JAVA"</p>

CHAPTER 5. OTHER METHODS FOR CREATING DRL FILES

As an alternative to creating and managing DRL files within the Decision Central interface, you can create DRL files in external standalone projects using Red Hat Developer Studio, Java objects, or Maven archetypes. These standalone projects can then be integrated as knowledge JAR (kJAR) dependencies in existing Red Hat Decision Manager projects in Decision Central. The DRL files in your standalone project must contain at minimum the required **package** specification, **import** lists, and **rule** definitions. Any other DRL components, such as global variables and functions, are optional. All data objects related to a DRL rule must be included with your standalone DRL project or deployment.

5.1. CREATING DRL FILES IN RED HAT JBOSS DEVELOPER STUDIO

You can use Red Hat JBoss Developer Studio to create DRL files with rules and integrate the files with your Red Hat Decision Manager decision service. This method of creating DRL rules is helpful if you already use Red Hat Developer Studio for your decision service and want to continue with the same work flow. If you do not already use this method, then the Decision Central interface of Red Hat Decision Manager is recommended for creating DRL files and other rule assets.

Prerequisite

Red Hat JBoss Developer Studio has been installed from the [Red Hat Customer Portal](#).

Procedure

1. In the Red Hat JBoss Developer Studio, click **File** → **New** → **Project**.
2. In the **New Project** window that opens, select **Drools** → **Drools Project** and click **Next**.
3. Click the second icon to **Create a project and populate it with some example files to help you get started quickly**. Click **Next**.
4. Enter a **Project name** and select the **Maven** radio button as the project building option. The GAV values are generated automatically. You can update these values as needed for your project:
 - **Group ID: com.sample**
 - **Artifact ID: my-project**
 - **Version: 1.0.0-SNAPSHOT**
5. Click **Finish** to create the project.
This configuration sets up a basic project structure, class path, and sample rules. The following is an overview of the project structure:

```
my-project
|-- src/main/java
|   |-- com.sample
|   |   |-- DecisionTable.java
|   |   |-- DroolsTest.java
|   |   |-- ProcessTest.java
|   |
|   |-- src/main/resources
|   |   |-- dtables
|   |   |-- Sample.xls
|   |   |-- process
```



```

|     |-- sample.bpmn
|     |-- rules
|     |-- Sample.drl
|     |-- META-INF
|
|-- JRE System Library
|
|-- Maven Dependencies
|
|-- Drools Library
|
|-- src
|
|-- target
|
|-- pom.xml

```

Notice the following elements:

- A **Sample.drl** rule file in the **src/main/resources** directory, containing an example **Hello World** and **GoodBye** rules.
 - A **DroolsTest.java** file under the **src/main/java** directory in the **com.sample** package. The **DroolsTest** class can be used to execute rules.
 - The **Drools Library** directory, which acts as a custom class path containing JAR files necessary for execution.
6. Create a fact model with all necessary data objects for the DRL file. The **DroolsTest.java** file contains a sample Java object **Message** with getter and setter methods. You can edit this class or create a different Java object. In this example, a class **Person** containing methods to set and retrieve the first name, last name, hourly rate, and wage of a person is used.

```

public static class Person {

    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}

```

```

public Integer getHourlyRate() {
    return hourlyRate;
}

public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
}

public Integer getWage(){
    return wage;
}

public void setWage(Integer wage){
    this.wage = wage;
}
}

```

7. Update the `main()` method to pass the Java object to a rule.

The `DroolsTest.java` file contains a `main()` method that loads the knowledge base, inserts facts, and executes rules. The following method update passes the object `Person` to a rule:

```

public static final void main(String[] args) {
    try {
        // Load the knowledge base:
        KieServices ks = KieServices.Factory.get();
        KieContainer kContainer = ks.getKieClasspathContainer();
        KieSession kSession = kContainer.newKieSession("ksession-
rules");

        // Go!
        Person p = new Person();
        p.setWage(12);
        p.setFirstName("Tom");
        p.setLastName("Summers");
        p.setHourlyRate(10);

        kSession.insert(p);
        kSession.fireAllRules();
    }

    catch (Throwable t) {
        t.printStackTrace();
    }
}

```

To load the knowledge base, get a `KieServices` instance and a class-path-based `KieContainer` and build the `KieSession` with the `KieContainer`. In the previous example, a session `ksession-rules` matching the one defined in `kmodule.xml` file is passed.

8. Create a DRL file containing at minimum a package specification, an import list of data objects to be used by the rule or rules, and one or more rules with `when` conditions and `then` actions. The rule file `Sample.drl` contains an example of two rules. You can edit this file or create a new one.

■

```

package com.sample

import com.sample.DroolsTest.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
  end

```

9. Go to **File** → **Save** to save the file.
10. After you create and save all DRL assets in your project, right-click your project folder and select **Run As** → **Java Application** to build the project. If the project build fails, address any problems described in the **Problems** tab of the lower window in Developer Studio, and try again to validate the project until the project builds.



IF THE RUN AS → JAVA APPLICATION OPTION IS NOT AVAILABLE

If **Java Application** is not an option when you right-click your project and select **Run As**, then go to **Run As** → **Run Configurations**, right-click **Java Application**, and click **New**. Then in the **Main** tab, browse for and select your **Project** and the associated **Main class**. Click **Apply** and then click **Run** to test the project. The next time you right-click your project folder, the **Java Application** option will appear.

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new project as a knowledge JAR (kJAR) and add it as a dependency in the **pom.xml** file of the project in Decision Central.

5.2. CREATING DRL FILES USING JAVA

You can use Java objects to create DRL files with rules and integrate the objects with your Red Hat Decision Manager decision service. This method of creating DRL rules is helpful if you already use external Java objects for your decision service and want to continue with the same work flow. If you do not already use this method, then the Decision Central interface of Red Hat Decision Manager is recommended for creating DRL files and other rule assets.

Procedure

1. Create a Java object on which the rule or rules will operate.
In this example, a **Person.java** file in a directory **my-project** is created. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```

public class Person {
  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

```

```

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Integer getHourlyRate() {
    return hourlyRate;
}

public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
}

public Integer getWage(){
    return wage;
}

public void setWage(Integer wage){
    this.wage = wage;
}
}

```

2. Create a rule file in **.drl** format under the **my-project** directory.

The following **Person.drl** rule calculates the wage and hourly rate values and displays a message based on the result:

```

dialect "java"

rule "Wage"
    when
        Person(hourlyRate * wage > 100)
        Person(name : firstName, surname : lastName)
    then
        System.out.println("Hello" + " " + name + " " + surname + "!");
        System.out.println("You are rich!");
    end

```

3. Create a main class and save it to the same directory as the Java object that you created. The main class will load the knowledge base and execute rules.
4. In the main class, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the knowledge base, insert facts, and execute the rule from the **main()** method that passes the fact model to the rule.

In the following example, the required imports are listed and a main class `DroolsTest.java` is created:

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class DroolsTest {
    public static final void main(String[] args) {
        try {
            // Load the knowledge base:
            KieServices ks = KieServices.Factory.get();
            KieContainer kContainer = ks.getKieClasspathContainer();
            KieSession kSession = kContainer.newKieSession();

            // Go!
            Person p = new Person();
            p.setWage(12);
            p.setFirstName("Tom");
            p.setLastName("Summers");
            p.setHourlyRate(10);

            kSession.insert(p);
            kSession.fireAllRules();
        }

        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

- Download the **Red Hat Decision Manager 7.0 Core Engine** ZIP file from the [Red Hat Customer Portal](#) and extract it under `my-project/dm-engine-jars/`.
- In the `my-project/META-INF` directory, create a `kmodule.xml` metadata file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

This `kmodule.xml` file is a descriptor that selects resources to knowledge bases and configures sessions. This file enables you to define and configure one or more KIE bases, and to include DRL files from specific **packages** in a specific KIE base. You can also create one or more KIE sessions from each KIE base.

The following example shows a more advanced `kmodule.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
    <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
```

```

        <ksession name="KSession1_1" type="stateful" default="true" />
        <ksession name="KSession1_2" type="stateless" default="false"
beliefSystem="jtms" />
    </kbase>
    <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
        <ksession name="KSession2_1" type="stateful" default="false"
clockType="realtime">
            <fileLogger file="debugInfo" threaded="true" interval="10" />
            <workItemHandlers>
                <workItemHandler name="name" type="new
org.domain.WorkItemHandler()" />
            </workItemHandlers>
            <listeners>
                <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener" />
                <agendaEventListener type="org.domain.FirstAgendaListener"
/>
                <agendaEventListener type="org.domain.SecondAgendaListener"
/>
                <processEventListener type="org.domain.ProcessListener" />
            </listeners>
        </ksession>
    </kbase>
</kmodule>

```

This example defines two KIE bases. Two KIE sessions are instantiated from the **KBase1** KIE base, and one KIE session from **KBase2**. Specific **packages** of rule assets are included with both KIE bases. When you specify packages in this way, you must organize your DRL files in a folder structure that reflects the specified packages.

- After you create and save all DRL assets in your Java object, navigate to the **my-project** directory in the command line and run the following command to build your Java files. Replace **DroolsTest.java** with the name of your Java main class.

```
javac -classpath "./dm-engine-jars/*:." DroolsTest.java
```

If the build fails, address any problems described in the command line error messages, and try again to validate the Java object until the object passes.

- After your Java files build successfully, run the following command to execute the rules. Replace **DroolsTest** with the prefix of your Java main class.

```
javac -classpath "./dm-engine-jars/*:." DroolsTest
```

- Review the rules to ensure that they executed properly, and address any needed changes in the Java files.

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new Java project as a knowledge JAR (kJAR) and add it as a dependency in the **pom.xml** file of the project in Decision Central.

5.3. CREATING DRL FILES USING MAVEN

You can use Maven archetypes to create DRL files with rules and integrate the archetypes with your Red Hat Decision Manager decision service. This method of creating DRL rules is helpful if you already use external Maven archetypes for your decision service and want to continue with the same work flow. If you do not already use this method, then the Decision Central interface of Red Hat Decision Manager is recommended for creating DRL files and other rule assets.

Procedure

1. Navigate to a directory where you want to create a Maven archetype and run the following command:

```
mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -
-DinteractiveMode=false
```

This creates a directory **my-app** with the following structure:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |       |-- com
    |           |-- sample
    |               |-- app
    |                   |-- App.java
    |-- test
    |   |-- java
    |       |-- com
    |           |-- sample
    |               |-- app
    |                   |-- AppTest.java
```

The **my-app** directory contains the following key components:

- A **src/main** directory for storing the application sources
 - A **src/test** directory for storing the test sources
 - A **pom.xml** file with the project configuration
2. Create a Java object on which the rule or rules will operate within the Maven archetype. In this example, a **Person.java** file in the directory **my-app/src/main/java/com/sample/app** is created. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```
package com.sample.app;

public class Person {

    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;
```

```

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Integer getHourlyRate() {
    return hourlyRate;
}

public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
}

public Integer getWage(){
    return wage;
}

public void setWage(Integer wage){
    this.wage = wage;
}
}

```

3. Create a rule file in `.drl` format under the `my-app/src/main/resources/rules` directory. The following `Person.drl` rule imports the `Person` class and calculates the wage and hourly rate values and displays a message based on the result:

```

package com.sample.app;
import com.sample.app.Person;

dialect "java"

rule "Wage"
    when
        Person(hourlyRate * wage > 100)
        Person(name : firstName, surname : lastName)
    then
        System.out.println("Hello " + name + " " + surname + "!");
        System.out.println("You are rich!");
    end

```

4. In the `my-app/src/main/resources/META-INF` directory, create a `kmodule.xml` metadata file with the following content:


```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

This **kmodule.xml** file is a descriptor that selects resources to knowledge bases and configures sessions. This file enables you to define and configure one or more KIE bases, and to include DRL files from specific **packages** in a specific KIE base. You can also create one or more KIE sessions from each KIE base.

The following example shows a more advanced **kmodule.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
    <ksession name="KSession1_2" type="stateless" default="false"
beliefSystem="jtms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateful" default="false"
clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new
org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener" />
        <agendaEventListener type="org.domain.FirstAgendaListener"
/>
        <agendaEventListener type="org.domain.SecondAgendaListener"
/>
        <processEventListener type="org.domain.ProcessListener" />
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

This example defines two KIE bases. Two KIE sessions are instantiated from the **KBase1** KIE base, and one KIE session from **KBase2**. Specific **packages** of rule assets are included with both KIE bases. When you specify packages in this way, you must organize your DRL files in a folder structure that reflects the specified packages.

5. In the **my-app/pom.xml** configuration file, specify the libraries that your application requires. Provide the Red Hat Decision Manager dependencies as well as the **group ID**, **artifact ID**, and **version** (GAV) of your application.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.sample.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0</version>
<repositories>
  <repository>
    <id>jboss-ga-repository</id>
    <url>http://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

For the Maven artifact version supported in Red Hat Decision Manager, see [Installing Red Hat Decision Manager on premise](#).

- Use the `testApp` method in `my-app/src/test/java/com/sample/app/AppTest.java` to test the rule. The `AppTest.java` file is created by Maven by default.
- In the `AppTest.java` file, add the required `import` statements to import KIE services, a KIE container, and a KIE session. Then load the knowledge base, insert facts, and execute the rule from the `testApp()` method that passes the fact model to the rule.
In the following example, the required imports are listed and a fact model `DroolsTest.java` is created:

```

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public void testApp() {

  // Load the knowledge base:
  KieServices ks = KieServices.Factory.get();
  KieContainer kContainer = ks.getKieClasspathContainer();
  KieSession kSession = kContainer.newKieSession();

```

```

// Set up the fact model:
Person p = new Person();
p.setWage(12);
p.setFirstName("Tom");
p.setLastName("Summers");
p.setHourlyRate(10);

// Insert the person into the session:
kSession.insert(p);

// Fire all rules:
kSession.fireAllRules();
}

```

- After you create and save all DRL assets in your Maven archetype, navigate to the **my-app** directory in the command line and run the following command to build your files:

```
mvn clean install
```

The first time you run this command, the build process can take more time than usual. After the build completes, the results are displayed in the command line:

```

...

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
1.194 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
...
[INFO] -----
----
[INFO] BUILD SUCCESS
[INFO] -----
----
[INFO] Total time: 6.393 s
...
[INFO] -----
----

```

- Review the build results to ensure that the build ran properly, and address any errors in the files.

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new Maven project as a knowledge JAR (kJAR) and add it as a dependency in the **pom.xml** file of the project in Decision Central.

CHAPTER 6. NEXT STEPS

- *Testing a decision service using test scenarios*
- *Packaging and deploying a decision service*

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on: Monday, October 1, 2018.