

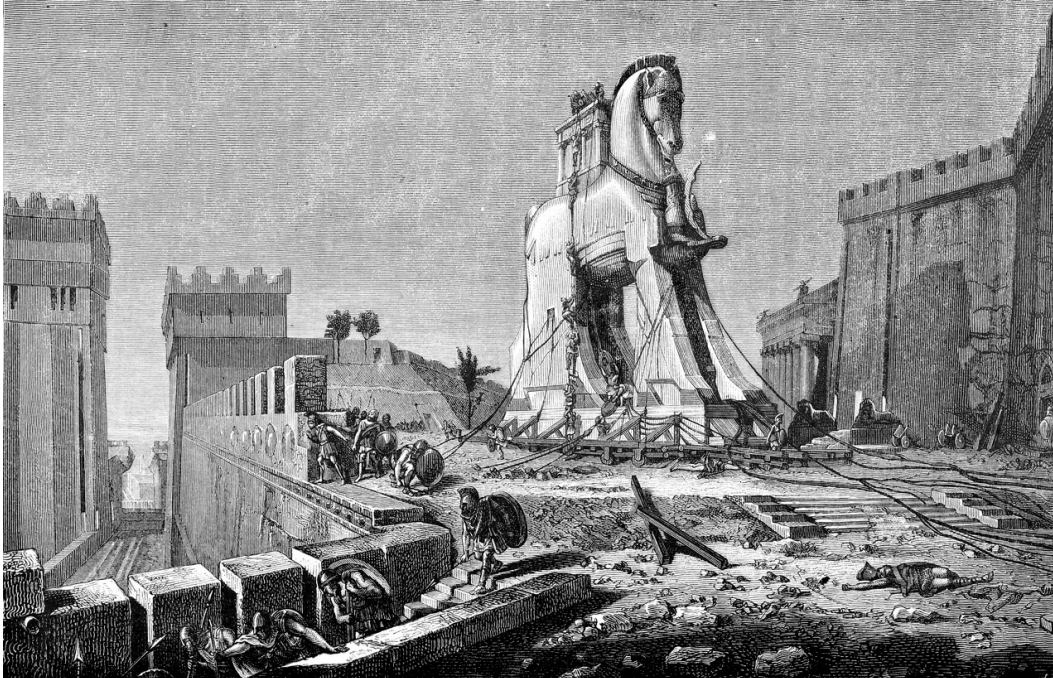
# Make informed automated decisions with business rules and Predictive Model Markup Language

## Build a decision app that integrates PMML with the Business Rules service on IBM Bluemix

Jean-Louis Ardoint

August 25, 2017

While business rules written by experts can be used to automate business decisions, there is a popular trend toward using predictive models, built from data mining, as the foundation for better decisions. This article shows how to take advantage of both worlds by integrating predictive model evaluation in business rules.



*...At break of day, when the Trojans beheld the camp of the Greeks deserted and believed that they had fled, they with great joy dragged the horse, and stationing it beside the palace of Priam deliberated what they should do.*

*As Cassandra said that there was an armed force in it, and she was further confirmed by Laocoon, the seer, some were for burning it, and others for throwing it down a precipice; but as most were in favor of sparing it as a votive offering sacred to a divinity, they betook them to sacrifice and feasting. ...*

**-Apollodorus: The Library**, translated by Sir James George Frazer

Even with the knowledge of a perfect prediction, the Trojans didn't make the right decision. The Greek army hidden inside the horse massacred the Trojans.

In today's world, the consequences might not be as dire as this ancient story, but businesses operating in a high risk, fast-moving, and disruptive environment know the importance of making the right decisions.

This tutorial shows how you can take advantage of predictions to make right – or at least *informed* – automated decisions.

You can use business rules written by experts to automate business decisions. However, there is a popular trend toward using predictive models, computed from big data, as the common building block to make informed decisions. Business rules are very effective to capture "conscious" or white box policies that come laws or regulations, for example. Predictive models, built using analytics on past data, can reproduce insights that are too complex, or not accessible for humans ("unconscious" or black box insights). Follow this tutorial to learn how to take advantage of both worlds by integrating predictive model evaluation in business rules, including both divination and decision steps.

You can integrate other services in business rules in the following ways:

- Integrate the call as a regular function or method call, using a statically or dynamically linked library that provides the service.
- Call to a remote service, for example, a Representational State Transfer (REST) API.

Calling a remote service is trendier and fits better into a microservice architecture, but it brings difficulties, like added complexity and latency. For a good introduction to monolith versus microservice architecture, see Martin Fowler's blog post at [martinfowler.com/bliki/MicroservicePremium.html](http://martinfowler.com/bliki/MicroservicePremium.html).

This tutorial demonstrates a monolith integration. An evaluation of a Predictive Model Markup Language (PMML) model is called as a regular Java™ method from the conditions of a rule that is run in the Business Rules service on IBM® Bluemix. The evaluation of the PMML model uses the `JPMML-evaluator` Java library.

## What you need to build your application

This tutorial shows how to build a Business Rules service that computes an insurance premium using a combination of business rules and a PMML neural network model. The PMML model predicts the claim amounts, depending on a number of parameters (such as car age, driver

gender, and an urban or rural environment). Rules take advantage of this prediction and other parameters to compute the insurance premium. To demonstrate the integration path, this tutorial uses a simplified version of rules.

You should be familiar with Java development, and optionally with Maven. In addition, you should have some familiarity with rule in IBM Operational Decision Manager (ODM), IBM ODM on Cloud, or the Business Rules service on IBM Bluemix.

You can use git commands, or download the source as a compressed file to begin the tutorial.

To complete the tutorial, you need a Bluemix account. To write the Java code of the execution model, and the rules, you need an Eclipse installation with the Business Rules extension. Follow the [Using Rule Designer to develop business rules applications](#) instructions in the IBM Bluemix documentation.

Get sample code for this tutorial from GitHub at <https://github.com/JeanLouisArdoint/IBM-ODM-Rules-PMML>.

This tutorial walks through the following main steps:

1. Creating the execution model, including some classes to represent the Business Rules service parameters, and one class to represent the PMML evaluator.
2. Authoring the rules.
3. Creating the decision operation.
4. Deploying the service on Bluemix.
5. Testing the decision service using the Bluemix console.

First download or clone the sample code from <http://github.com/JeanLouisArdoint/IBM-ODM-Rules-PMML> on your computer. This creates a `IBM-ODM-Rules-PMML` directory.

Steps 1, 2, 3 describe in detail how to set up your execution model and decision project. You can skip those steps by running Rule Designer and importing the projects in the `IBM-ODM-Rules-PMML` directory.

## 1. Create the execution model

The Business Rules service can use a Java archive as its execution object model (XOM). You can import the model in Rule Designer as the business object model (BOM). Business rules, written in controlled natural language, can call methods of the BOM.

Because the Business Rules service can directly use Java classes, you can provide the PMML evaluator as a regular Java class. The class takes a PMML file as its constructor parameter. The PMML file is enclosed in the Java archive, which means that you need to redeploy the service to change the predictive model. (An alternative and more flexible way is to pass the entire PMML model as a parameter of the rule service.)

1. First, create a Maven project, using the settings in the following screen capture:

**New Maven Project**  
Configure project

**Artifact**

Group Id: odm-pmml

Artifact Id: odm-pmml-xom

Version: 0.0.1-SNAPSHOT

Packaging: jar

Name: PMML XOM to be used with ODM

Description:

**Parent Project**

Group Id:

Artifact Id:

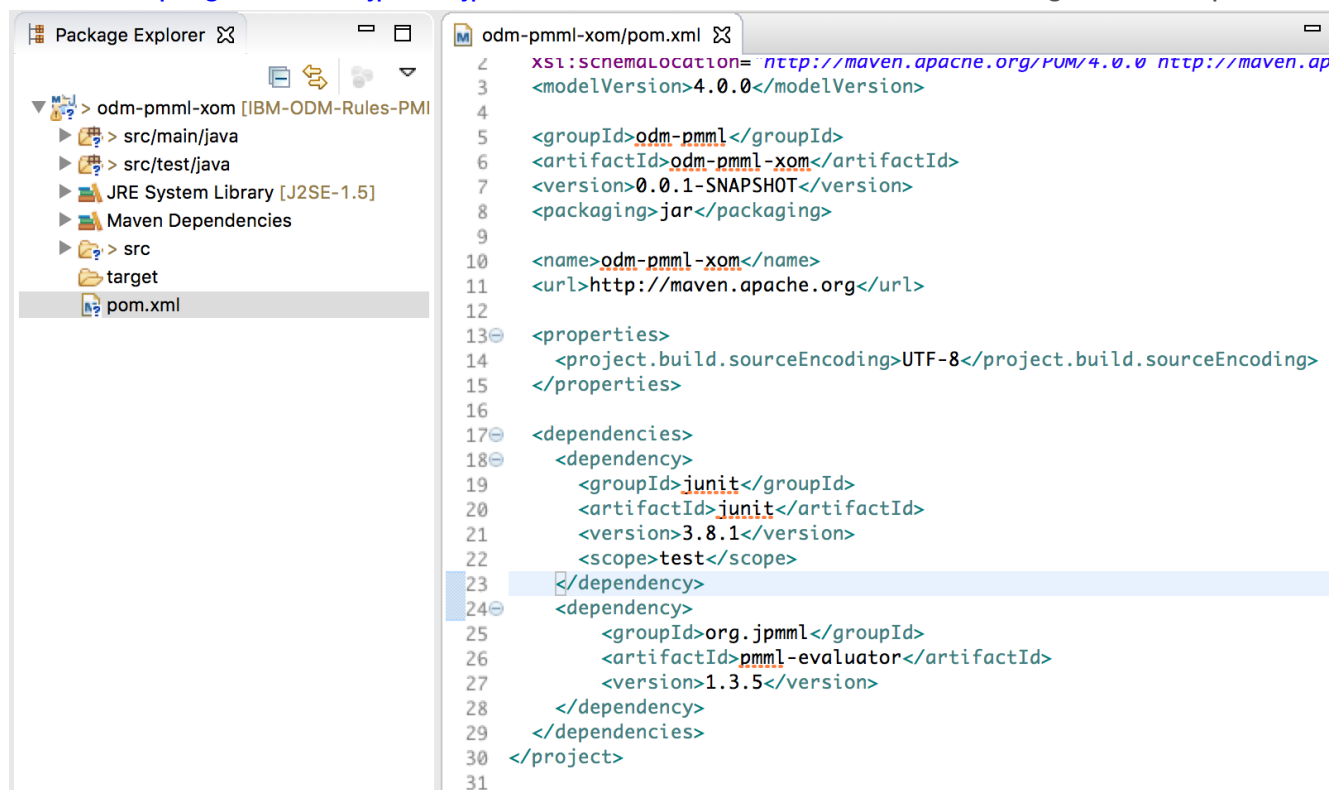
Version: Browse... Clear

Advanced

< Back Next > Cancel Finish

2. Add the Maven dependency to Java PMML API evaluator libraries in the Maven descriptor (pom.xml file). Get the exact `<groupId>`, `<artifactId>` and `<version>` from the sample

code at <http://github.com/jpmml/jpmml-evaluator>, as shown in the following screen capture:



### 3. Get the PMML file to use.

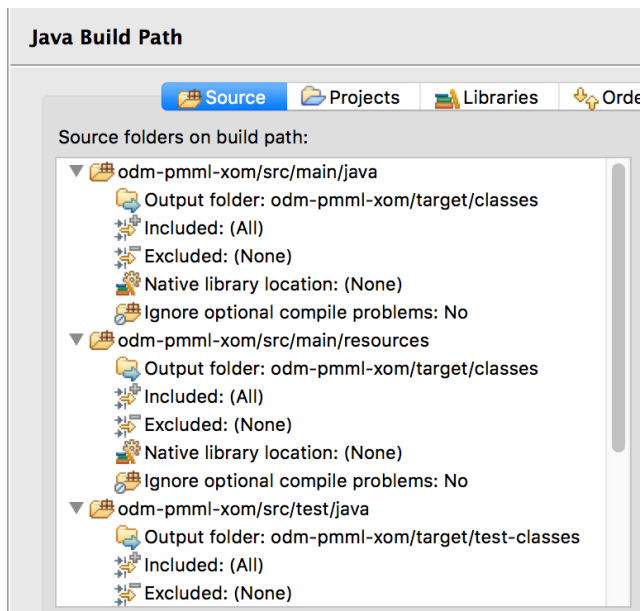
This tutorial uses an example from <http://dmg.org/pmml/v4-3/NeuralNetwork.html>. Put the file in the resources of the project, in the `com.test` package. The file has been slightly modified to remove an issue in types and some blank characters in the field names.

See the changes at <https://github.com/JeanLouisArdoint/IBM-ODM-Rules-PMML/commit/04f1af072fee364ee13b68689ec4aeaa33fa6350>.

### 4. Declare the resource path and export the resource file.

To ensure that the resource file is exported, check the Java build path in the Java project properties. If the values in the `odm-pmml-xom/src/main/resources` folder don't match

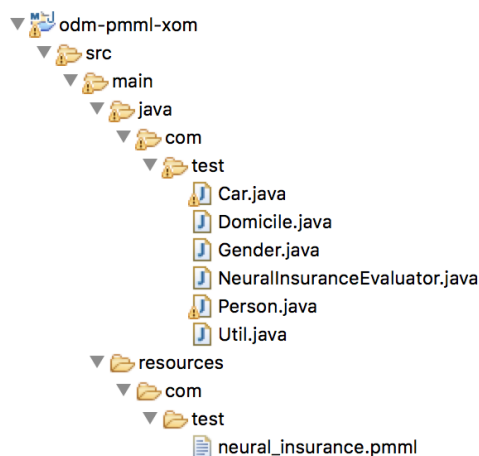
the following screen capture, select **Included** and **Excluded** and click the **Remove**



button.

##### 5. Author the Java XOM file.

The example for this tutorial deals with a Car and a Person, which are represented as Java classes. There are also some enumerated values, like Gender and Domicile. Use the `NeuralInsuranceEvaluator` class handles the call to the Java PMML API evaluator. The code that is doing the actual calling is generic. Only the adaptation from the input types to Java PMML API and the output is specific to this PMML example. Finally, add a utility class to provide a method to round the result, because the Java PMML API typically provides a calculated result with 17 digital digits, which are difficult to read and not meaningful for an estimated value. The XOM project looks like the following screen capture:



##### 6. Write a small unit test.

You can write a small unit test just to check that the PMML evaluation works:



```

package com.test;

import junit.framework.TestCase;

public class PMMLEvaluateTest extends TestCase {

    public void testUrbanMale() throws Exception {
        NeuralInsuranceEvaluator app = new NeuralInsuranceEvaluator("neural_insurance.pmml");
        double result = app.evaluate(Gender.MALE, 3, Domicile.URBAN, 3.0);
        System.out.println("Predicted claim amount: " + result);
    }

    public void testRuralFemale() throws Exception {
        NeuralInsuranceEvaluator app = new NeuralInsuranceEvaluator("neural_insurance.pmml");
        double result = app.evaluate(Gender.FEMALE, 5, Domicile.RURAL, 13.0);
        System.out.println("Predicted claim amount: " + result);
    }
}

```

The execution of this test should produce the following results:

```

Predicted claim amount: 3412.6243607549823
Predicted claim amount: 2284.2148634469913

```

## 7. Complete XML and JSON serialization.

The classes used as parameters must be serialized in Extensible Markup Language (XML), JavaScript Object Notation (JSON), or both. The XML serialization is based on Java Architecture for XML Binding (JAXB), and the JSON serialization is based on the Jackson library. The annotations from JAXB are recognized by Jackson, so they should be enough to get both XML and JSON serialization for this tutorial.

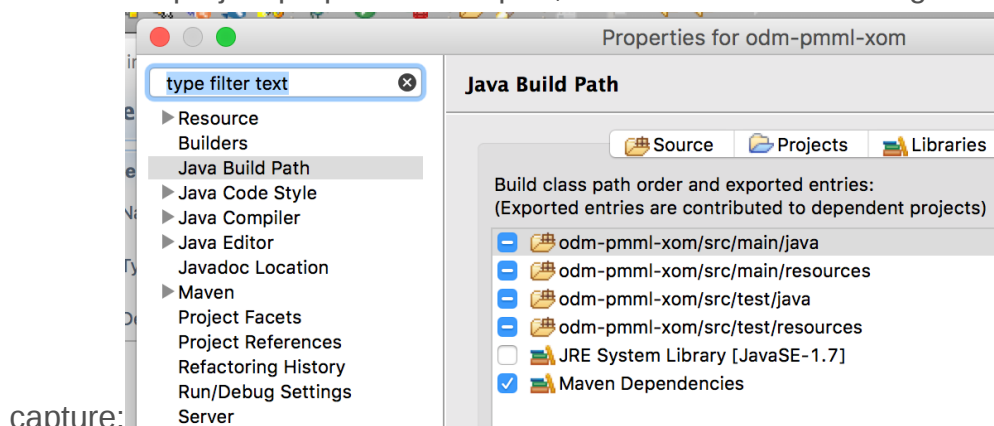
For JAXB, the following rules apply:

- Constructor without args (which can be private). A consequence is that fields cannot be final.
- Fields without setters must be annotated with `@XmlElement`.

See the [Executing rules by using the REST service](#) and [Hosted transparent decision services](#)

## 8. Export all the dependencies.

To ensure that the Java project exports a complete XOM, select **Maven Dependencies** in the Java project properties to export, as shown in the following screen



capture:

## 2. Create the rules and ruleflow

Now that the Java classes are created, you can import them so you can create a BOM and then write the business logic that takes advantage of PMML scoring as a rule within a ruleflow.

Complete the following steps to create rules and ruleflow:

1. Create a main rule project.

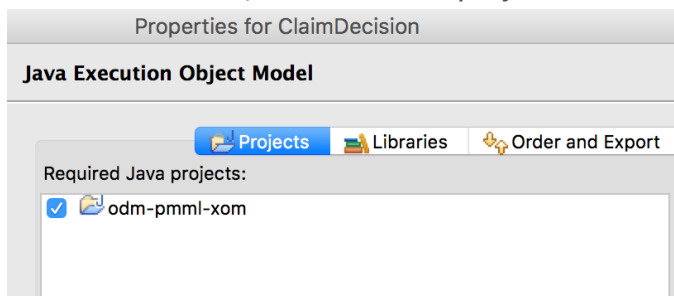
In the Decision Service Map (make sure the Rules perspective is selected and the Decision Service Map is visible), click on **Create main rule project**. (Or, you can click **File > New > Project**.)

Create a main rule project named `ClaimDecision`.

2. Import the XOM.

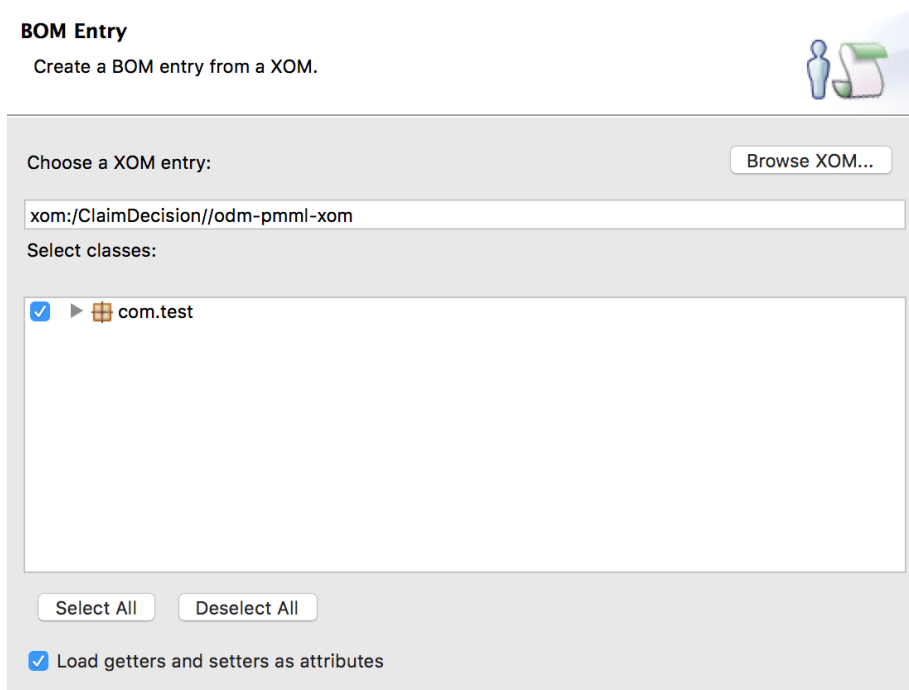
In the Decision Service Map, click **Import XOM**. Select **Java execution object model**.

3. Choose the `odm-pmml-xom` Java project, as shown in the following screen capture:



4. Create the BOM.

In the Decision Server map, click **create BOM**. In the New BOM Entry window, click **Next**. Click **Browse XOM**. Select the **com.test** package, as shown in the following screen capture:

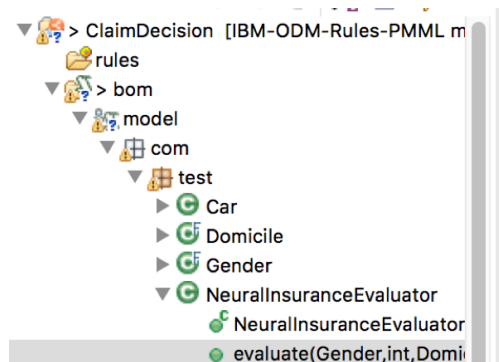


Click **Finish**.



## 5. Verbalize the methods.

Open the `NeuralInsuranceEvaluator.evaluate` method in the BOM editor by selecting it in the Rule Explorer, as shown in the following screen capture:

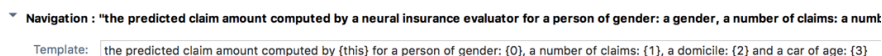


Under Member Verbalization, click **Create** to create a default verbalization.

Replace the default verbalization with

the predicted claim amount computed by {this} for a person of gender: {0}, a number of claims: {1}, a residence: {2} and a car of age: {3}

The result should look like the following screen capture:



There is one more method to verbalize, in a very similar way. Open the `util.round` method, and verbalize it with the `round({0}, {1})` expression.

Now save the BOM.

## 6. Create variables.

Click **New > Variable set**. Select `/ClaimDecision/rules` as the source folder, and name the variable set `variables`.

The input variables for the sample decision service are the car and the person.

The sample decision service has the following outputs:

- the premium, a number computed by the rules from the predicted claim amount and some discount policies
- the acceptance, a boolean, that indicates if the insurance request was accepted

The evaluator is directly created inline using the `new com.test.NeuralInsuranceEvaluator("/com/test/neural_insurance.pmm1")` expression. Then, the PMML model is automatically loaded when the rule execution starts. The evaluator parameter doesn't need to be in the decision operation parameters.

The resulting variable set looks like the following screen capture:

Variable Set: variables

Name	Type	Verbalization	Initial Value
evaluator	com.test.NeuralInsuranceEvaluator	the evaluator	new com.test.NeuralInsuranceEvaluator("/com/test/neural_insurance.pmm1")
car	com.test.Car	the car	
person	com.test.Person	the person	
premium	double	the premium	
accepted	boolean	accepted	

## 7. Define a simple rule.

Define a rule that stores the result of the PMML evaluator inside the `predicted claim amount` variable. The rule just uses this amount, and accepts only requests when the predicted

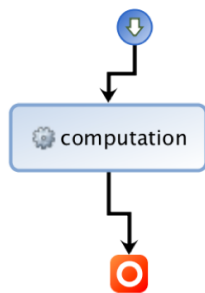
claim amount is less than 2000. In this case, it also computes the premium, as shown in the following example:

```
definitions
set claims to the number of claims of 'the car';
set gender to the gender of 'the person';
set domicile to the domicile of 'the person';
set age to the age of 'the car';
set 'predicted claim amount' to the predicted claim amount computed by 'the evaluator' for a person of
gender:
'gender', a number of claims: 'claims' , a domicile: 'domicile' and a car of age: 'age' ;
if 'predicted claim amount' is less than 2000
then
set 'the premium' to round( 'predicted claim amount' * 0.1, 2) ;
set 'accepted' to true ;
```

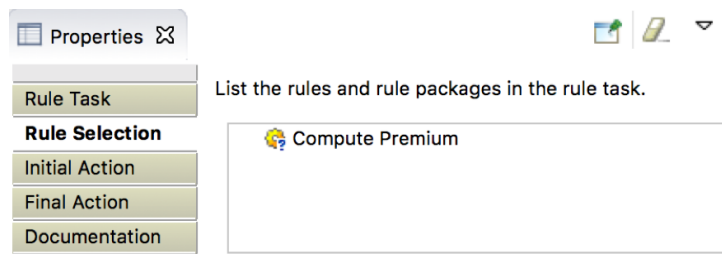
If you want to use the same kind of test using the `predicted claim amount` in a decision table, you can copy the `definitions` block in the previous example code into the preconditions of the decision table.

#### 8. Create a simple rule flow with a task.

After the rule is created, the project is completed with a ruleflow, a single task named `computation`, as shown in the following screen capture:



The computation task uses the default rule execution algorithm, called Fastpath. (For more information, see [Fastpath mode](#).) The task refers to the `compute Premium` rule. Modify the Rule Selection properties, as shown in the following screen capture:



## 3. Create the decision operation

After you author the business logic as a ruleset, you can define a decision operation that takes advantage of the business logic. The decision operation relates input and output parameters to a ruleset.



Complete the following steps to create the decision operation:

1. Click **Add decision operation** in the Decision Server map.

## 2. Define the signature of the decision operation.

The car and person variables are its input parameters, and it returns the premium and the accepted boolean output parameters. Drag and drop the variables from eligible variables to the input or output parameter, as shown in the following screen capture:



**Input Parameters**  
Define the parameters required to call the execution.

Parameter name	Verbalization	Type
 car	the car	com.test.Car
 person	the person	com.test.Person

**Input - Output Parameters**  
Define the parameters that are required, modified, and then returned by the execution.

Parameter name	Verbalization	Type

**Output Parameters**  
Define the parameters that are initialized and returned by the execution.

Parameter name	Verbalization	Type
 premium	the premium	double
 accepted	accepted	boolean

## 4. Deploy the decision service to Bluemix

Deploying the decision service to Bluemix involves creating a business rules service instance, creating a deployment configuration, and then deploying. All these operations are described in detail at [Creating a Business Rules service instance](#) in the Business Rules service documentation for IBM Bluemix.





Creating the business rules service instance is specific to using the Business Rules service on Bluemix. After you complete the steps, you can bind the service to an application, or you can use the Bluemix console to directly test it as a service.

## 5. Test the decision service in the Bluemix console

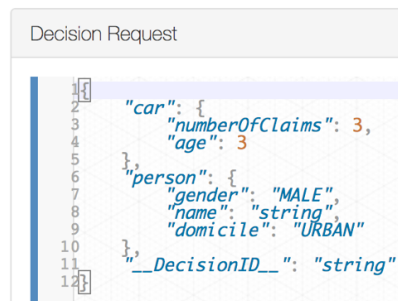
You can test your decision service from the Bluemix console.

You can call the decision service as a SOAP/XML, REST/XML, or REST/JSON web service. The REST service passes incoming data as HTTP content in XML or JSON format.

1. In Bluemix, go to the services page. It should show the service that you created in step 4. Click the service.
2. Click the **Decision Services** tab. You should see the following decision service:

Decision Services	Actions
IPOnBluemix 1.0	
insurancePremium 1.13	  

3. To test the decision service, click the arrow icon next to it. The Test page shows the Decision Request and the Decision Response. Choose the relevant data format (JSON or XML). It generates a corresponding sample payload under Decision Request, similar to the following



screen capture:

Change the payload or just click **Run Test**.

If you get an error such as Error 500: `java.lang.NoClassDefFoundError: org.jpmmml.evaluator.Evaluator`, the XOM was not correctly deployed. To deal with the error, see the end of Step 1.

If you get an error without a clear explanation message, it might be from a missing resource. Check the export of resources in Step 1.

In all error cases, after a change in Rule Designer, you must redeploy the RuleApp, come back to the Decision Services page, refresh, and choose the latest version of the service. Otherwise, the decision service is run. The result is displayed under Decision Response.

The response when using the default JSON payload should match the following example:

```
{
  "__DecisionID__": "string",
  "accepted": false,
  "premium": 0
}
```

You can try a different input value to check that the service works. For example, try the following input:

```
{
  "car": {
    "numberOfClaims": 5,
    "age": 3
  },
  "person": {
    "gender": "FEMALE",
    "name": "string",
    "domicile": "RURAL"
  },
  "__DecisionID__": "string"
}
```

The service returns should look like the following example:

```
{
  "__DecisionID__": "string",
  "accepted": true,
  "premium": 172.8
}
```

## Conclusion

In this tutorial, you learned how to integrate a PMML evaluation into a Business Rules Service that runs on IBM Bluemix. You can apply most of the instructions in this tutorial to IBM ODM Decision Server Rules and IBM ODM on Cloud.

Now you can write business policies that take advantage of predictions from any PMML model that is built from historical data. You can make better, informed automated decisions.

## Acknowledgements

The author thanks the numerous team members from the IBM ODM team for their help, notably Laurent Gâteau, Benjamin Ratiarisolo, Jose de Freitas, Nicolas Peulvast and Nicolas Sauterey.

## Related topics

- [PMML 4.3 - General Structure](#)
- [The Business Rules service on IBM Bluemix](#)
- [jpmml/jpmml-evaluator on GitHub](#)
- [FasterXML/jackson on GitHub](#)
- [Introduction to JAXB documentation](#)
- [IBM Operational Decision Manager product documentation](#)
- [Cloud journeys on developer.ibm.com/code](#)
- [Hybrid Integrations journeys on developer.ibm.com/code](#)

© Copyright IBM Corporation 2017

([www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml))

**Trademarks**

([www.ibm.com/developerworks/ibm/trademarks/](http://www.ibm.com/developerworks/ibm/trademarks/))