Building ODM Applications in Scala

# What this article is about

This article explains how to integrate IBM Operational Decision Manager rule execution services in a Scala application.

IBM Operational Decision Manager is IBM's Business Rule Management System. Said differently, it is software you use to expose operational decisions, such as the price a product will be proposed to a given client, as services and to externalize the business rules behind these decisions in a dedicated repository so that line of business stakeholders can author and manage them without requiring systematic intervention of IT.

Scala is a programming language that drastically extends Java’s object oriented mechanisms and adds the functional paradigm to it beyond what Java 8 does. I highly recommend you the tutorials at <https://www.scala-lang.org/>

Just as with Java code, Scala code is compiled into JVM bytecode and can interoperate with Java code. This article is taking advantage of this interoperability, as the ODM engine can run on Java objects (actually, bytecode objects) and has a Java API.

The following figure shows the various options:



This article shows how to make option 4 happen.

The rest of this article makes the following assumptions:

* You know how to build a Hello World app with ODM and Java
* You have installed ODM 8.9, Scala IDE 4.5 (http://scala-ide.org/) and git

Most of the details in this article will run if you have alternate versions of these IDEs or alternate versions of the language compilers. Please reach out to me at [friedlan@fr.ibm.com](mailto:friedlan@fr.ibm.com) if there are errors, and I’ll update this article.

The example is reasonably sophisticated and shows that various Scala mechanisms are perfectly managed in ODM.

All considered, this exercise is quite simple, and will be even simpler when we will be able to use a single IDE for both Scala and ODM. The reason this article is so long is that I am giving all details to make sure you will succeed with this approach.

# Why it is interesting

There are two reasons why the ODM – Scala integration is interesting.

First, Scala can be viewed as a better Java than Java, as explained above. Therefore, building ODM applications with Scala is a valuable option.

Second, Scala is the preferred language for building Apache Spark applications, the new generation Big Data technology. The following article, written by Pierre Feillet from the IBM ODM Lab, explains well the power of combining Apache Spark, Big Data analytics, machine learning and predictive models with rule based decision management: <https://developer.ibm.com/odm/docs/solutions/odm-and-analytics/odm-business-rules-with-apache-spark-batch-operations/>

# Installing Rule Designer on top of the Scala IDE

This paragraph is actually a placeholder for future versions of this article, as Rule Designer, the ODM Eclipse plugin requires Eclipse 4.4.2. Scala IDE 4.5 runs on Eclipse 4.6.1

Working with two IDEs makes things slightly more complex. We need to work on two different workspaces and move libraries around at set up. Once things are in place you do not notice it.

Once Rule Designer and Scala IDE will support common Eclipse versions, things will be simpler.

# Installing and running the example

This section explains how to install and run the example provided in GitHub while the next sections explain how to build your own example from scratch. Please, reach out to me if there are errors, and I’ll update this article.

Here are the instructions to install and run the example.

1. Create a home directory on your hard drive
2. From there, clone the GIT repository from GitHub: git clone [https://github.com/FrancisFriedlander/scala-odm.git](https://github.com/FrancisFriedlander/ScalaODM.git)
3. Open Scala IDE on code/WorkspaceScalaIDE
4. File > Import… > General > Existing Projects into Workspace and select the Workspace as root directory
5. To correct the error, right click project BankingAppScala, then Properties > Java Build Path > Libraries > Add Variable… > Configure Variables… > New… and define ILOG\_EXECUTION\_SERVER\_HOME as your ODM Execution Server directory (in my case /opt/ibm/odm890/executionserver); Accept the full build
6. Note the two libraries required to use Scala classes from Java code. They are referenced in the “Scala Library container” folder. In my case (macOS Sierra), these libraries are:
   * /Applications/eclipse\ scala.app/Contents/Eclipse/plugins/org.scala-lang.scala-library\_2.11.8.v20160304-115712-1706a37eb8.jar
   * /Applications/eclipse\ scala.app/Contents/Eclipse/plugins/org.scala-lang.scala-reflect\_2.11.8.v20160304-115712-1706a37eb8.jar
7. Copy these two libraries to WorkspaceRuleDesigner/Libraries/lib, as they are needed in the two projects located in WorkspaceRuleDesigner. Alternatively, the projects could reference the libraries in their original locations.
8. Open Rule Designer on code/WorkspaceRuleDesigner
9. Import Existing Projects into Workspace (As in 4 above)
10. Double click the error (“*The deployment configuration 'BankingRuleapp' references a target 'Local RES' that cannot be retrieved*”), open the Target Servers tab, delete Local RES and create it again as a “Local Rule Execution Server on Java SE” and select the directory shared parallel to the workspace, then save

Your setup is now fully functional! You can now:

* Run the Scala App: In Scala IDE, select project BankingAppScala and run bankingapp.MainScala as a Scala Application
* Author and deploy the rules: In Rule Designer, select project Banking Rules, deployment/BankingRuleapp.dep and Proceed to RuleApp deployment…
* Proceed with the next section to build the example from scratch

# Creating the example from scratch

# Creating the Scala XOM (eXecution Object Model) in Scala IDE

Before starting, please create the three following parallel sub-directories:

* WorkspaceScalaIDE – self explanatory
* WorkspaceRuleDesigner – self explanatory
* Shared – the directory where you will store the shared ruleapp directory

Then, create Scala classes for the XOM. Here is the code for the Scala XOM from the example:

**package** banking\_scala\_xom

**import** scala.collection.mutable.ListBuffer

**class** Asset(value2: Int) {

**var** value = value2

}

**trait** AssetOwner {

**var** assets: Array[Asset]

**def** sumAssetValues = {

**if** (assets.length > 0) assets map (\_.value) reduce (\_ + \_)

**else** 0

}

}

**class** Customer(accountBalance: Int) {

**var** alerts: ListBuffer[String] = ListBuffer()

**def** totalBalance = accountBalance

// Verbalization-friendly methods

**def** addAlert(alert: String): Unit = alerts += alert

// Return type Unit, else will be considered a ListBuffer and we will not be allowed to verbalize it

**def** clearAlerts = alerts.clear

}

**class** CustomerOwningAssets(accountBalance: Int, assets2: Array[Asset]) **extends** Customer(accountBalance) **with** AssetOwner {

**override** **var** assets = assets2

**override** **def** totalBalance = sumAssetValues + accountBalance

}

Those of you who are familiar with Java but no with Scala can really appreciate how concise it is. The same code in Java would be three times as long.

A few things are worth mentioning:

* This model takes advantage of Mixin Class Composition, leading to a hierarchy with two different classes of customers: Customer and CustomerOwningAssets. ODM manages that hierarchy as expected (ie: as in Java)
* The Scala compiler I am using does not follow JavaBean conventions (and I guess all Scala compilers behave identically). For example, the Java accessors for Scala attribute value are not getValue and setValue, but value() and value\_$eq. This means we need to either implement these accessors as Scala methods, as BOM methods, or manually verbalize these methods. I followed the latter approach.
* Specifically for collections, I had to create two “verbalization-friendly” methods, addAlert and clearAlerts, as the automatically generated getter and setter cannot be verbalized. I recommend you to follow that same pattern for all mutable collections.

Before moving on to the next paragraph (Creating the Main Rule Project):

* I strongly recommend you to create a Java application in the same workspace to check how the Scala constructs map to Java constructs and correct the Scala XOM as needed. You have to simplify as much as possible the Scala XOM => BOM => vocabulary mappings, given the two constraints mentioned above.
* Once you feel confident about the XOM, switch to Rule Designer and create a project named Libraries (a plain one, not a Java project) and, back to Scala IDE, create a JAR library that packages your Scala XOM, and save it as WorkspaceRuleDesigner/Libraries/lib/BankingScalaXom.jar.
* Copy also in that directory the 2 Scala-to-Java libraries as in step 7 above

# Creating the Main Rule Project

In Rule Designer, create a Main Rule Project using the wizard (File > New > Other… > Rule Designer > Main Rule Project), then add the three libraries in Libraries/lib to the Java Execution Object model.

You can then proceed as usual: Define rule packages, ruleflows, rules, variables, Decision Operations, Deployment Configurations, etc... There are NO limitations.

You will, however, need to tailor the verbalizations as required. In the example above:

# Vocabulary Properties

uuid = bc18b7a3-f4b8-40cc-9783-4fe6f5f7e8bf

# banking\_scala\_xom.Asset

banking\_scala\_xom.Asset#concept.label = asset

banking\_scala\_xom.Asset.value()#phrase.navigation = the value of {this}

# banking\_scala\_xom.AssetOwner

banking\_scala\_xom.AssetOwner#concept.label = asset owner

banking\_scala\_xom.AssetOwner.assets()#phrase.navigation = the assets of {this}

banking\_scala\_xom.AssetOwner.sumAssetValues()#phrase.navigation = the sum of the asset values of {this}

# banking\_scala\_xom.Customer

banking\_scala\_xom.Customer#concept.label = customer

banking\_scala\_xom.Customer.addAlert(java.lang.String)#phrase.action = add {0} to the alerts of {this}

banking\_scala\_xom.Customer.clearAlerts()#phrase.action = clear the alerts of {this}

banking\_scala\_xom.Customer.totalBalance()#phrase.navigation = the total balance of {this}

# banking\_scala\_xom.CustomerOwningAssets

banking\_scala\_xom.CustomerOwningAssets#concept.label = customer owning assets

# Creating the Scala Project for Rule Execution Server on Java SE

You can skip this step if you want your Scala App to consume your ruleapp:

* As a Decision Connect API (<https://developer.ibm.com/odm/docs/business-operations-connect/decision-connect/)>, in which case you can use Java stubs
* As a REST API (after deploying the ruleapp to the JEE RES), in which case you can use standard Scala calling mechanisms

Creating a Scala project for Rule Execution Server on Java SE is almost straightforward:

* Create a Java Project for Rule Execution on Java SE (File > New > Other… > Client Project for Decision Service > Java Project for Rule Execution on Java SE)
* Add the three libraries to the Java Build Path and make the required changes in main. In the example, the project is called BankingAppJava
* Switch back to the Scala IDE and create a Scala project (BankingAppScala in the example). Add the Scala XOM project to the Required projects on the build path
* Copy the content of src from the BankingAppJava to BankingAppScala.
* Align the Java Build Path of BankingAppScala to that of BankingAppJava except for the three libraries mentioned above, that are not needed
* Create a Scala App that does similar things as the Java main. In the example it is called MainScala.scala

# Next steps

I personally like the idea to write and maintain in Scala XOMs and applications that call rule-based decision services.

A few things could be done better than what I described here. First, when Scala IDE and Rule Designer will support the same Eclipse versions things will be simplified.

Second, using Scala introspection, we could probably generate generic Java wrappers around Scala classes that follow JavaBean conventions. Alternatively, and it is probably easier and better from a performance viewpoint, we could generate the verbalization file automatically.

In any case, please reach out to me ([friedlan@fr.ibm.com)](mailto:friedlan@fr.ibm.com)) if you wish to discuss.