Business Rules for Medical Payments

Recondo Technology

Michael Henry, Krista Horn, Zack Webb

CSM Field Session
June 17, 2014

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

1.1 Client Information

From Recondo Technology's Project Description:

> Recondo connects providers, payers, and patients using cloud computing solutions throughout the healthcare revenue cycle. Our software services are designed to ensure Proper Payments across the continuum of US healthcare and bring efficiencies and cost savings to healthcare payment processing, which currently costs US healthcare a staggering $480 billion in annual expense.

1.2 Product Vision

The product is intended to help convert the current systems from using spreadsheets to a newer technology, a Guided Decision Table in Drools Workbench. The Guided Decision Table has an easier to navigate interface so that the Business Analysts will be able to add and modify rules in a faster and more efficient manner. The convertor tool compiles the 155,772 rules into 42,847 rules to reduce redundancy. These rules are compressed based on the client specified column sets.

The Business Analysts are responsible for updating the Authorization Required Rules table so that the hospitals can bill insurance companies correctly. The conversion tool can be used multiple times so that the people who still are using the spreadsheet do not have to convert to Drools Workbench all at the same time.

2.0 REQUIREMENTS

For this project, the client specified certain requirements that needed to be taken into consideration. The functional and non-functional project requirements are listed below.

2.1 Functional Requirements

1. Needs to be able to import current rules into the Drools Workbench.
2. Needs to be able to import other rule sets into the Drools Workbench for other services (optional).

2.2 Non-Functional Requirements

1. Need to add effective and end dates to each rule
2. Need the ability to provision both “Required” and “Not Required” rules - default will be “Unknown”
3. Remove Provider NPI as a requirement (optional for rule overrides)
4. Add PayerRecoId to rules
5. Rule will be either “Required” or “Not Required” and service can handle default of “Unknown”
6. Ability to add plan type if needed (e.g. HMO, PPO) in case there are different requirements for the same payer with different plan types
7. Build a centralized set of Auth Required Rules that all clients can use
8. Ability to have an override rule when there are client specific Auth Required Rules that don’t fall within the default rules

3.0 SYSTEM ARCHITECTURE

In the existing architecture, shown in Figure 1, the Glassfish server was connected to a Drools instance that fired rules against a DRL source file. This DRL file was generated based on the rules created in the Auth Required spreadsheet, which is easier for the Business Analysts, who research these rules, to maintain and edit. Whenever a change was made to the spreadsheet, the converter program had to be run to replace the old DRL file with an updated version.

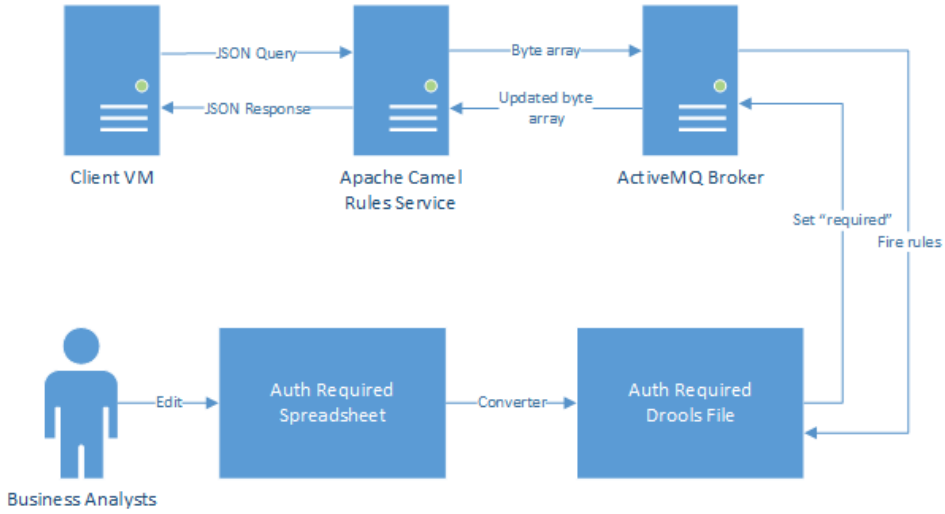

Figure 1: Previous System Architecture.

In our new implementation, shown in Figure 2, the Glassfish server instead points to an instance of Drools Workbench, which internally handles conversion from a Guided Decision Table (which is edited like a spreadsheet) to a DRL file that can be fired against. Our migrator script converted the Auth Required spreadsheet into a Guided Decision Table file, which could then be added to Drools Workbench by accessing its internal Git repository. This should only have to happen once, as any subsequent changes to the rules will be automatically implemented by Drools Workbench.

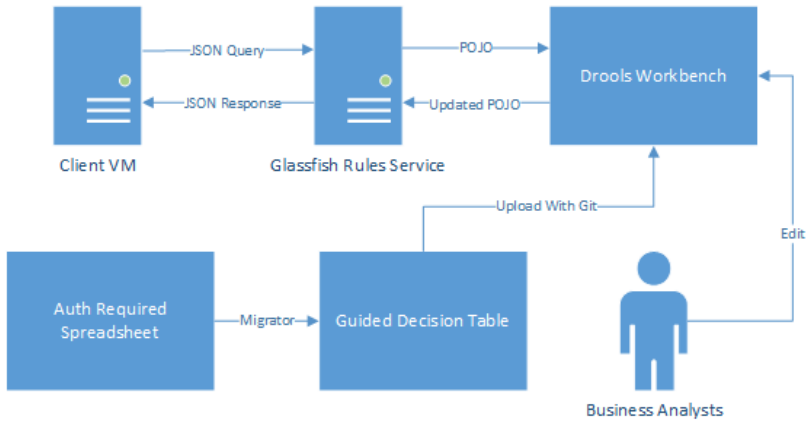

Figure 2: New Implementation System Architecture.

4.0 TECHNICAL DESIGN

4.1 Ruby Migrator

The script to migrate from CSV to GDST (Guided Decision Table) was implemented in Ruby. It has three main stages in the conversion process: collecting rules data, processing rules data, and outputting rules data.

The first stage collects data from two external files: the CSV file containing the entire set of rules, and a separate config file that must be created manually. This config file is necessary because the CSV does not have any information about the columns, such as what type of data should be expected and how incoming data should be compared to rules data for that column. This information is required by the GDST format to correctly convert the table into a Drools file. The config file also specifies the title of the table, the prefix to use for POJO dependencies, and can specify three different types of columns necessary for the GDST.

Processing the data happens in three different ways. First, the rules are consolidated. This process is discussed further in section 4.1.1. Second, the rules are grouped. In Recondo's original program that converted the CSV to a DRL file, the rules were clustered into five major groups based on what certain types of data they did or did not have. However, their implementation does not actually sort the rules based on the groupings, just ensures that they are clustered together. If these groups need to be ordered somehow, the code is all in a single function that can easily be reimplemented to fit given specifications. Finally, the rows are numbered. The GDST format expects every rule to have two extra cells at the start: one is the line number, and the other is an optional description that can be entered for each rule.

After the rules have been processed, they must be written to a GDST file. This part of the script was reverse-engineered from GDST files that we had created in the Workbench to perfectly emulate the style that we knew was accepted. This turned out to be unworkable, as GDSTs use XML to encode data, and the XML markup can take up a significant amount of storage space on top of the actual data. After converting the entire file (without consolidation, as that had not yet been implemented at this time), the resulting GDST was over 500 MB, which is well over the Workbench's file size limit. After making sure that our implementation was accepted by Drools Workbench, we began experimenting with ways to cut down on space. The following methods were effective in compressing the data:

- Removing extraneous tags. Every single cell in the table (of which there were almost 3.5 million) had the tag "<isOtherwise>false</isOtherwise>". This tag did not appear to be used for anything, and removing it had no effect on the table's functionality. Each cell also specified what type it was, but if no type was specified then the Workbench assumes it is a string. Since most

of the cells in the table are strings, we could remove almost every single type tag with no problems.

- Removing whitespace. Because of the sheer size of the file, the spaces and newlines alone used nearly 30 MB of space. The Workbench has no problem reading a file without whitespace, so there was no reason to keep adding it to the file.
- Consolidating the rules. This reduced the number of rules from about 155,000 to about 43,000, which saved a significant amount of space.

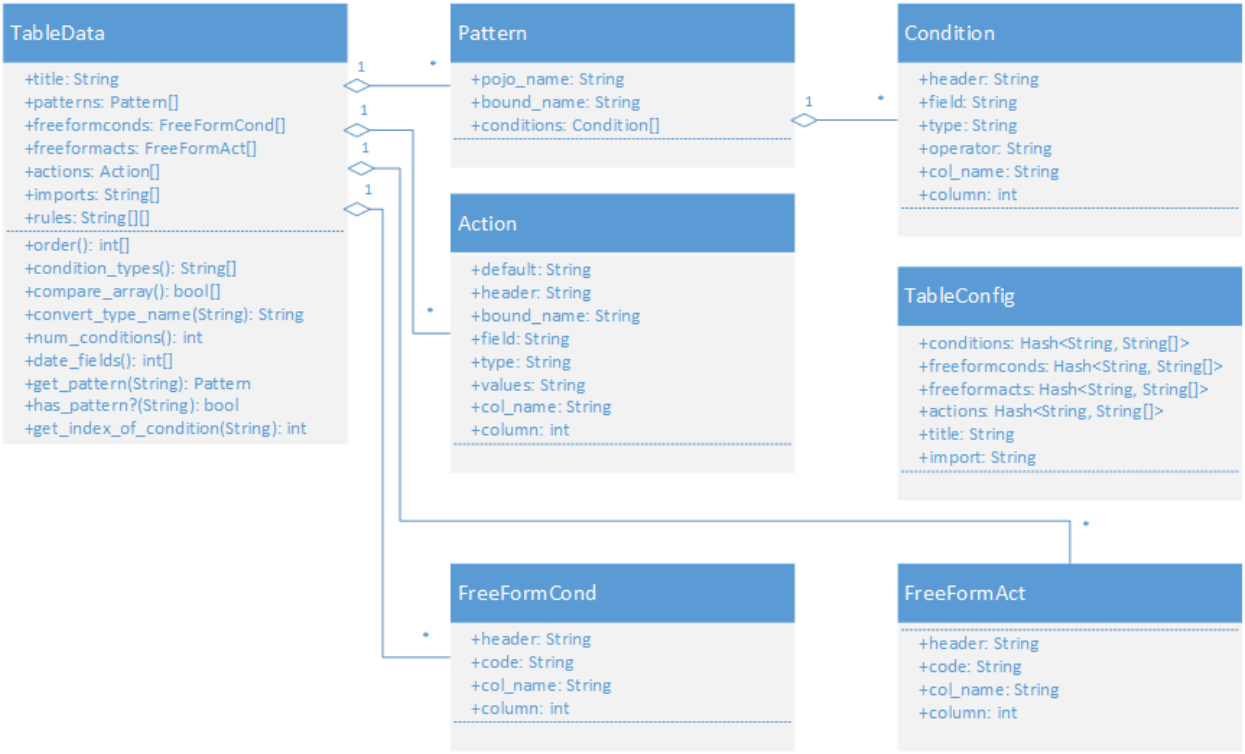

Figure 3: UML Diagram of Migrator Object Structure.

4.1.1 Consolidator

One of the requirements given to us by Recondo was to allow certain columns in the table to consolidate multiple values into one cell, allowing multiple rules to condense into a single rule with lists of data in some of the cells.

In order to accomplish this consolidation, the script has to find rules that can be consolidated safely. It does this by way of a hash map, in which the keys are arrays used to compare similar rules, and the values are arrays which will contain rules.

The keys are generated in a fashion that uniquely identifies clusters of rules that can be combined. For each value in a rule, a value is inserted into the key array. If that value is one of the columns that cannot be consolidated, its contents are copied into the key array. This is because rules should only be consolidated if all of their non-list columns match perfectly. If that value is a column that can be consolidated, a boolean is inserted into the key array: true if that rule has a value there, or false if that rule has only a blank there. The reason for this is because blank fields in Drools mean that that value can be anything and the rule will still match. Obviously, blank fields cannot be inserted into a comma-separated list of values, or else that rule would effectively vanish.

Once all of the rules have been organized into this hash map, they must be transferred back into the rules matrix. The process for this is fairly straightforward. For each key in the hash map, a rule is inserted into the new rules matrix. The string values are copied over exactly, and the boolean values are converted into empty strings. Then, each rule in the corresponding value is added onto this rule by appending its values onto the comma-separated list of the new rule.

Insurance Provider	Procedure Code	Patient Gender	Auth Required?
0100	555		No
0201	555		No
0100	555	M	No
0201	789	F	Yes
0444	789	F	Yes

Key	Value
[true, "555", "", "No"]	["0100", "555", "", "No"]
	["0201", "555", "", "No"]
[true, "555", "M", "No"]	["0100", "555", "M", "No"]
[true, "789", "F", "Yes"]	["0201", "789", "F", "Yes"]
	["0444", "789", "F", "Yes"]

Insurance Provider	Procedure Code	Patient Gender	Auth Required?
0100, 0201	555		No
0100	555	M	No
0201, 0444	789	F	Yes

Figure 4: Example Consolidation Steps. "Insurance Provider," highlighted in green, is designated as allowing comma-separated lists.

4.2 Conversion Checker

A Java script was written to make sure that all of the rules were migrated correctly. This script checks for each line of the CSV in the GDST file.

The program first reads in the files using Java's predefined libraries: CSVReader and DocumentBuilder. For each line in the CSV, the unused columns are ignored and the dates formatted. Then, the line is compared against each rule in the GDST in succession until a match is found. If no match is found, the method will return false and increment the "rule not found" count by one. After one CSV line is done comparing, the next line in the CSV starts the comparison process.

Once every line of the CSV has been checked, the program will return a count of the total rules checked, a count of the rules not found, and a count of rules found. If all rules were found, then the conversion was successful.

The conversion checker was written for the Authorization Required Rules migration for testing purposes. Since there are specific Authorization Required Rules column names used in this Java program, it is not a portable program for testing the migration of other Guided Decision Tables.

5.0 DESIGN AND IMPLEMENTATION DECISIONS

Many design decisions were made for this project. These decisions were made based on optimization and software limitations. Drools workbench interactions, software language limitations, and space limitations were analyzed.

5.1 Drools Workbench

Our first major decision was how to import the table into Drools Workbench. There are several different methods that allow one to add rulesets to the Workbench, including importing a known decision table, importing a Drools file, or creating a guided decision table. The Drools file approach was not acceptable, because it would be far too difficult and cumbersome to edit rules in this fashion. Therefore, a guided decision table was the best option for them to interact with the rules. Importing a decision table and converting it to a guided decision table did not actually work because of errors in the Workbench itself. That left us with only one option, creating a guided decision table from scratch and importing it ourselves. Because the workbench does not have any working import features in the interface, we had to gain access to the Git repository on the workbench's backend and push the table in ourselves.

5.2 Conversion Checker

Another decision we made was to create a validator tool to check that all of the rules were imported into the project. Within this decision, Java was chosen to write the script rather than Ruby. We had realized that using Ruby for the conversion script was not an optimal decision because only one team member was comfortable using Ruby. The decision to use Java was also a good choice because Java has predefined libraries to read CSV and XML formatted files.

5.3 Lessons Learned

We learned that understanding how different programs interact in the system is very important when a small change is needed. Understanding the system saves time when tracing through the problems and finding the correct place to fix. Another lesson learned was the importance of reading documentation for unknown systems. Documentation for the programs are a very good reference tool when learning how to use a specific software.

5.4 Known Problems and Temporary Solutions

As mentioned in Section 5.1 and 6.1, there are known bugs in Drools Workbench. Not being able to drop in a generated Guided Decision Table is the biggest problem because doing so is a main part of

our project. We found a workaround for this problem; however, it is still very sensitive. To temporarily fix this problem follow these steps[1]:

1. Clone the Git repository of the project you are working in within the Workbench. These are located at "APACHE_HOME\bin\.niogit\".
2. Inside the project you will be using in the Workbench, create a Guided Decision Table with the same name as your generated table. You do not need to add any columns or rules. Make sure you save inside workbench.
3. Copy the GDST file into the "src\main\resources" folder within the cloned Git repository in the file manager on your computer.
4. Open Git Bash and navigate to the correct repository.
5. Inside Git Bash, pull from the repository to make sure everything is up to date.
6. Making sure you added the GDST file copied into the folder, commit the changes.
7. Push to the repository.
8. Inside the workbench, refresh the page. The workbench will go back to the opening screen. Navigate to the Guided Decision Tables. The generated GDST you dropped in should be there.

Each time you completely shut down the workbench and your workstation, you will receive a "null" error like before upon restarting. As of now, the only way to fix this problem is to delete every GDST inside the workbench, pull from Git Bash to update your files, and follow the steps listed above.

Another known bug is Glassfish not recognizing changes made in the workbench once glassfish is already running. If changes are needed to be made to your Guided Decision Table while the glassfish service is running, make the necessary changes and "Build and Deploy" the table, then restart glassfish.

[1] Note: This is a temporary fix that is not proven to work every time.

6.0 RESULTS

The goal of this project was to move the current required rules that are stored in an Excel spreadsheet to a Guided Decision Table in Drools Workbench, which would allow the existing software to fire rules in the form of JSON messages and produce a response.

6.1 Successes

Our migrator script was able to convert the spreadsheet into a Guided Decision Table (GDST file) which can be uploaded to Drools Workbench by accessing the Workbench's git repository. However, a persistent bug in Drools Workbench severely limited our testing capabilities. A bug report was submitted and we are awaiting a reply or solution.

The converted GDST file also compressed rules that shared common features. The client specified which conditions were able to be combined into a list. To ensure that all rules were converted correctly we wrote a Java program that checks for each line of the CSV in the XML file.

JSON messages are able to fire against a Guided Decision Table in the workbench. The correct authorization required message is returning to the user.

6.2 Existing Problems

The procedure codes and diagnosis codes are entered into the rules set as alphanumeric ranges (e.g. "A0100-A8000"), and default condition columns are not powerful enough to parse this and determine if a given value falls within the range. The rules may have minimum ages, but the client software sends a date of birth to the rules service. Therefore, the date of birth must be converted to a patient's age as of today and compared to the minimum age. Again, this is more computation than a default condition column can handle. For these reasons, the three columns mentioned above must use free-form Drools columns.

However, including a free-form Drools column into the table causes the Workbench to not recognize the column. The column is showing; however, the source code for the Guided Decision Table returns an error. We believe this is an error in the Drools code we have written to attempt to solve the above problems, as the XML format was copied from an example column generated within the Workbench. This means that if correct Drools implementations of these solutions can be written, they can be added to the table through the migrator script's config file.

APPENDIX A: MIGRATOR SCRIPT INSTRUCTIONS

The migrator script has two source files:

- **convert.rb** This has the main logic for collecting the data, consolidating the rules, and writing them to a GDST file.
- **objects.rb** This just holds the classes used to store rules data during the conversion process.

In order to run the script, ensure that you have Ruby 1.9.3 installed, then run this from the command line:

```
ruby convert.rb -cg arr.csv arr.cfg
```

The "c" flag indicates that the script should consolidate rules. If this is not included in the command, the output file will not consolidate and will simply output every rule on its own row of the table.

The "g" flag indicates that the script should group rules. This runs a function in the script called `group_and_order`. As of now, this function groups the rules into five major clusters in the same way that "rddbloader/rddbloader-trunk/src/main/java/com.recondotech.ruleloader/ CentralizedAuthRequiredRuleLoader.java" does. As in the Java file, it does not order the groups in any particular way. This can be easily altered later in order to fit any specifications that become necessary.

The two filenames point to the CSV containing the rules and the config file describing the columns. Both need to be present for a proper conversion to GDST.

For a full explanation of the script's usage, type `ruby convert.rb --help` to see the help page.

The config file is a series of entries prefixed by all-caps keywords. They can be in any order within the file.

- **`TITLE <table title>`** This defines the title of the table and the filename of the eventual output. There should only be one TITLE entry.
- **`IMPORT <com.example.project.>`** This defines the prefix of the POJO imports. For Recondo, this should start as "com.recondotech". There should only be one IMPORT entry.
- **`CONDITION <column name>`** This begins a section defining a condition column. In this case, "column name" is the header of the column in the CSV. Following a CONDITION entry are five more lines in this order:
    - **`<pojo name>`** The name of the POJO that this column retrieves data from.
    - **`<header>`** The header text that will display above the column in Drools Workbench.

    - **<field>** The name of the variable in the POJO that this column compares to the rules values.
    - **<type>** The type of the variable being used.
    - **<operator>** The method of comparison used. Common operators are "==", "<", "<=", ">", ">=", and "in".
- **ACTION <column name>** This begins a section defining an action column. As before, "column name" is the name of the column in the CSV. Following an ACTION entry are six more lines in this order:
    - **<pojo name>** The name of the POJO that this column will write to.
    - **<header>** The header text that will display above the column in Drools Workbench.
    - **<field>** The name of the variable in the POJO that this column will alter.
    - **<default>** The default value to be written for a rule if a different value is not specified.
    - **<type>** The type of the variable in the POJO that this column will alter.
    - **<value list>** The list of possible values to choose from for each rule. This should be a comma-separated list.
- **FREEFORM CONDITION <column name>** This begins a section defining a free-form condition column. As before, "column name" is the name of the column in the CSV. If the column name is absent, a new column will be made without pulling data from the CSV. Following a FREEFORM CONDITION entry are two more lines in this order:
    - **<header>** The header text that will display above the column in Drools Workbench.
    - **<code>** The Drools code that will be inserted into the column. The phrase `@{param}` is used to refer to the rules value for that column.
- **FREEFORM ACTION <column name>** This begins a section defining a free-form action column. As before, "column name" is the name of the column in the CSV. If the column name is absent, a new column will be made without pulling data from the CSV. Following a FREEFORM ACTION entry are two more lines in this order:
    - **<header>** The header text that will display above the column in Drools Workbench.
    - **<code>** The code that will be inserted into the column. This may be Java code, since it is inserted in the rule's right-hand side.

APPENDIX B: ADDING A POJO AND FIRING NEW RULES

In order to deploy a Guided Decision Table from the workbench to allow the Rules Service to fire rules at it a few steps need to be taken. Code snippets and any links can be found on the Recondo Wiki.

1. Modify the RuleService project (checked in at http://svn/reco/services/ruleService).
    a. Add the POJO to the pom.xml in ruleServiceWeb
    b. Add the POJO to the pom.xml in ruleServiceEjb
2. Rebuild the RuleService project.
    a. To build and deploy the war file run the bdwar.bat script at the root of the ruleServiceWeb project.
    b. To build and deploy the ejb and ear file run the src\main\resources\bdear.bat script in ruleServiceEjb
3. Pull and rebuild all of the POJOs that the RuleService depends on. (checked in at http://svn/reco/components/rulesService/)
4. Modify files so Drools Workbench will build and deploy to the proper location.
    a. In the local Maven directory modify the settings.xml file with the following:
        i. In the <servers> section of the file:

```
<server>
      <id>recondo-guvnor-repo</id>
      <username>guvnor</username>
      <password>{PasswordOnWiki}</password>
</server>
```

        ii. In the <repositories> section of the file:

```
<repository>
      <id>recondo-guvnor-repo</id>
      <name>recondo-guvnor-repo</name>
      <releases>
            <enabled>true</enabled>
           <updatePolicy>always</updatePolicy>
           <checksumPolicy>fail</checksumPolicy>
      </releases>
      <snapshots>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
      </snapshots>
      <url>http://papaafrepo001.recondo.vci:8081/
      artifactory/guvnor-release</url>
```

```
</repository>
<repository>
    <id>recondo-guvnor-repo</id>
    <name>papaafrepo001</name>
    <releases>
        <enabled>false</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://papaafrepo001.recondo.vci:8081/
    artifactory/guvnor-snapshot</url>
</repository>
```

b. In your project in Drools Workbench edit the pom.xml with:
   i. Under the Distribution Management section:

```
<distributionManagement>
    <repository>
        <id>recondo-guvnor-repo</id>
        <url>http://papaafrepo001.recondo.vci:
        8081/artifactory/guvnor-release</url>
    </repository>
    <snapshotRepository>
        <id>recondo-guvnor-repo</id>
        <url>http://papaafrepo001.recondo.vci:
        8081/artifactory/guvnor-snapshot</url>
    </snapshotRepository>
</distributionManagement>
```

   Note: If you are deploying from a local Workbench only do not include the <repository> section.

5. Next you must configure the maven coordinates of the POJOs in Glassfish.
   a. Go to Resources -> JNDI -> Custom Resources -> properties on Glassfish Admin Console
   b. If needed create a new resource called RulesServiceProperties
      i. Set JNDI name to RulesServiceProperties
      ii. Set resource type to java.util.Properties

iii. Add a property at the bottom of the screen for a groupId, an artifactId and a version.

c. Name each property as follows:
   i. Elements in a property name should be separated by a dot.
   ii. For the AuthReq part of the rules the name should start with “arr”
   iii. A property name for the groupId should end with "groupId"
   arr.{descriptive_text}.groupId
   iv. A property name for the artifactId should end with “artifactId”
   arr.{descriptive_text}.artifactId
   v. A property name for the version should end with “version”
   arr.{descriptive_text}.version
d. The values for each property as follows:
   i. “groupId” = com.recondotech
   ii. “artifactId” = ARR
   iii. “version” = LATEST to automatically use the latest version.
e. Save the changes.
f. Through the applications menu launch the “ruleServiceWeb” application.

6. Use the Advanced Rest client to fire JSON messages into the RuleService to test.
   a. The URL should be “http://localhost:8080/ruleServiceWeb-1.0/rules/executeRules?rulesType=ARR”
   b. Make sure POST is selected.
   c. Type in your JSON message and press Send to fire it.