MOPA: Monitoreo Remoto de Pacientes Crónicos

Entregable #4

Virtual Sense S.A. - ISISTAN - IAM

1 - Introducción

El presente entregable detalla las actividades realizadas en el contexto del cuarto y último trimestre de ejecución del proyecto, y de acuerdo a lo planificado durante el trimestre anterior, el entregable se focaliza en dos objetivos a) investigación de motores de reglas, b) contextualización del soporte necesario de dichos motores en función del tipo de reglas que Virtual Sense S.A. estima requerir/implementar a futuro, y c) elaboración de prototipo ejemplo y extensible por Virtual Sense S.A. que contenga un stack de software específico con soporte de especificación reglas para un motor elegido.

En virtud de la capacitación en FHIR brindada por HL7 Argentina realizada por los investigadores del ISISTAN durante Mayo del 2023 en el contexto del proyecto, y por encontrarse disponible en stack Java las herramientas más maduras para implementar FHIR en backends en producción, se acotó la investigación de a motores Java. Además, esto no condiciona tecnológicamente a la empresa, quien prevé implementar FHIR para modelar un subconjunto de la información almacenada por la plataforma de telemonitoreo actual (según Entregable #3) y además ha desplegado el backend de la plataforma mediante una arquitectura de microservicios y orquestación mediante Docker Swarm, lo que facilita el despliegue de un nuevo microservicio (stack Java + motor de reglas en este caso) sin afectar los microservicios ya existentes, tal es el caso del backend principal implementado en Laravel + PHP.

2 - Motor de reglas: Conceptos básicos

Un motor de reglas (RE) es un módulo que automatiza la gestión de ciertos procesos muy variables. El concepto fundamental consiste en separar los objetos que participan en procesos de la lógica que implementa dichos procesos, con el fin de garantizar una alta mantenibilidad del sistema pero a la vez permitir una evolución de las reglas. Por ejemplo, en el dominio bancario, las reglas pueden codificar qué tipo de crédito tiene preaprobado un cliente dependiendo de factores como ingreso, relación laboral, edad, etc. En el dominio de salud, una regla puede disparar cierto tipo de alarmas dependiendo de ciertos valores fisiológicos observados, por ejemplo.

La lógica se define mediante reglas de escritura. En cada proceso, el RE reconoce las reglas que deben aplicarse y en qué objetos se debe operar. Si hay una variación en la lógica, las reglas se pueden modificar sin necesidad de intervenir en la arquitectura principal del sistema.

Anatómicamente, imaginemos un RE como un sistema que toma como entrada datos y reglas. Las reglas se aplican sobre los datos y devuelven una salida dependiendo de la definición en sí de las reglas. Un ejemplo fuera del dominio de salud podría ser el e-commerce, donde se pueden modelar reglas para ofrecer ciertos descuentos, por ejemplo:

- Aplicar un descuento de 20% al cliente si el carrito de compras supera los $5000.
- Aplicar un descuento de 10% para la primera compra en el sitio.

Estas reglas ejemplo definen al *cliente* y el *carrito de compra* como los datos de entrada para las reglas, las cuales serán ejecutadas si alguna de las condiciones especificadas son cumplidas por los datos.

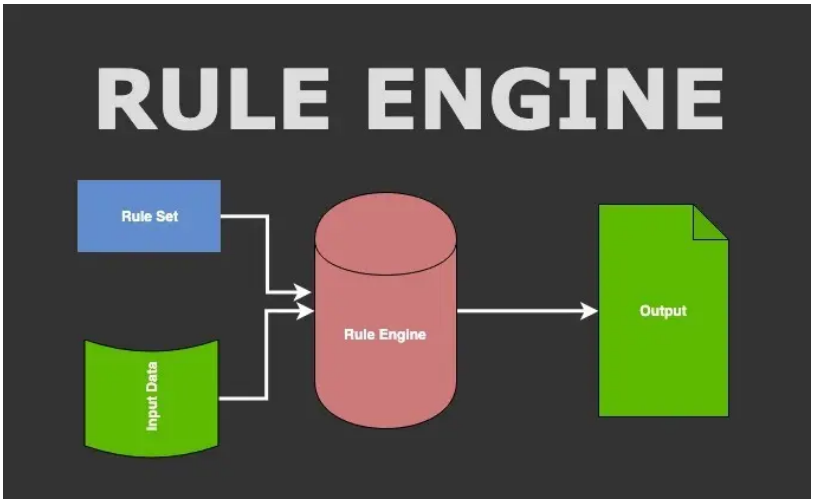


Figura 1: Vista general de un RE

Las ventajas que se desprenden del uso de RE son claras, a saber:

1. Provee una mayor flexibilidad dado que es posible modificar, agregar o borrar reglas sin mayores cambios en el código fuente del sistema.
2. Reduce la complejidad, dado que exime de construir la lógica para aplicar las reglas en el sistema.
3. Proveer mayor reusabilidad de las reglas construidas debido a que son un *asset* que se mantiene separado del sistema.

Debido a que el proyecto originalmente tuvo como objetivo la investigación de un sistema de reglas para que los profesionales -cardiología en este caso- vuelquen conocimiento experto para levantar alarmas, se relevaron primeramente los motores de reglas existentes para el

lenguaje Java. Estos se resumen en la siguiente tabla. Se incluye sitio del RE, último release y fecha, y si implementan o no la JSR (Java Specification Request) 94[1], la cual aspira a proveer una API estandarizada para motores de reglas en Java.

Motor	**Web site**	**Versión estable**	**Último commit**	**JSR 94?**
Drools	https://www.drools.org/	8.39.0	Mayo 2023	Sí
OpenL tablets	https://openl-tablets.org/	5.6.28	Mayo 2023	No
Jess	http://alvarestech.com/temp/fuzzyjess/Jess60/Jess70b7/docs/index.html	7.1p2	Noviembre 2008	Sí
Evrete	https://www.evrete.org	3.0.03	Abril 2023	Sí
Rulebook	https://github.com/deliveredtechnologies/rulebook	0.11	Diciembre 2018	No
Easy Rules	https://github.com/j-easy/easy-rules	4.1	Diciembre 2020	No

Como puede apreciarse, existen algunos motores que no cuentan con releases desde hace un tiempo prolongado, o bien carecen ya de mantenimiento.

Por un lado, Jess fue uno de los primeros RE pensado para integrarse fácilmente a Java. Utiliza una implementación eficiente del algoritmo Rete (ver Sección 4), lo que mejora implementaciones manuales de chequeo de reglas. Para escribir las reglas, Jess utiliza un lenguaje propio (Jess Rules Language), el cual extiende la sintaxis del lenguaje funcional

[1] https://www.jcp.org/en/jsr/detail?id=94

Lisp, o bien mediante un formato XML (que resulta mucho más *verbose*). Originalmente proveía una integración con el Eclipse IDE y una interfaz de línea de comandos para especificar y testear preliminarmente reglas.
Easy Rules apunta a ofrecer un motor liviano de reglas basadas en POJO (Plain Old Java Object). Las reglas complejas se crean a partir del patrón Composite y reglas más simples. A diferencia de los RE existentes, Easy Rules no se basa en un DSL (Domain Specific Language) o artefactos de configuración como XML para especificar las reglas, sino que las mismas se especifican mediante anotaciones. Un aspecto débil sin embargo es que no implementa la JSR 94 y las reglas representan artefactos mezclados con la lógica de la aplicación, desde el punto de vista de las clases implementadas. A continuación se presenta un ejemplo de regla simple:

```java
@Rule(name = "Hello World rule", description = "Always say hello world")
public class HelloWorldRule {

    @Condition
    public boolean when() {
        return true;
    }

    @Action
    public void then() throws Exception {
        System.out.println("hello world");
    }
}
```

y la forma de disparar la regla en una aplicación:

```java
public class Launcher {
    public static void main(String... args) {
        // create facts
        Facts facts = new Facts();

        // create rules
        Rules rules = new Rules();
        rules.register(new HelloWorldRule());

        // create a rules engine and fire rules on known facts
        RulesEngine rulesEngine = new DefaultRulesEngine();
        rulesEngine.fire(rules, facts);
    }
}
```

RuleBook por su parte explota las Lambdas introducidas en Java 8 y el patrón Chain of Responsibility para definir reglas. RuleBook utiliza el concepto de Fact para representar los datos de entrada a las reglas. RuleBook permite modificar el estado de estos Facts, los cuales pueden ser leídos y modificados por cualquier regla en la cadena (chain). Las reglas también pueden retornar datos en un tipo diferente al Fact de entrada, para lo cual se proveen las *Decisions*. RuleBook emplea la idea de *fluent* API para describir las reglas, y tiene integración con el framework Spring y Java DSL. Por ejemplo, una regla puede ser la siguiente:

```
public class HelloWorldRule {
    public RuleBook<Object> defineHelloWorldRules() {
        return RuleBookBuilder
          .create()
            .addRule(rule -> rule.withNoSpecifiedFactType()
              .then(f -> System.out.print("Hello ")))
            .addRule(rule -> rule.withNoSpecifiedFactType()
              .then(f -> System.out.println("World")))
            .build();
    }
}
```

y su uso en una aplicación Java:

```
public static void main(String[] args) {
    HelloWorldRule ruleBook = new HelloWorldRule();
    ruleBook
      .defineHelloWorldRules()
      .run(new FactMap<>());
}
```

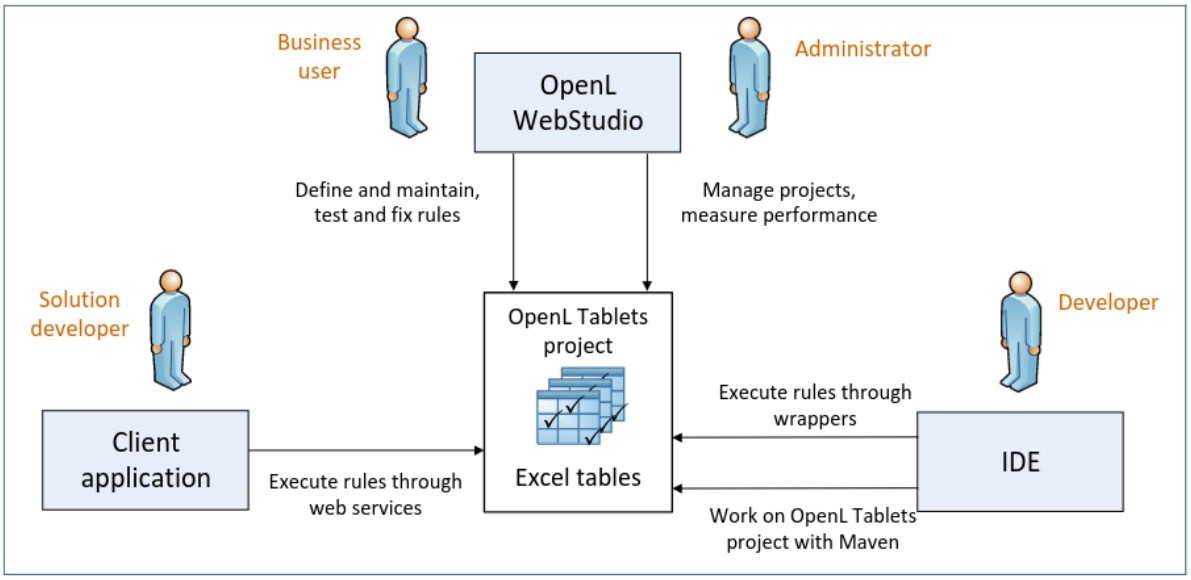
Dentro de los RE existentes que reciben mantenimiento periódico y además periódicamente tienen releases, se encuentran OpenL Tablets, Drools y Evrete. En las siguientes secciones se profundiza un poco más en las particularidades de estas herramientas.

3 - OpenL Tablets

Este RE apunta a orientar la especificación de reglas del lado del analista o experto del dominio, de forma de que esto no necesariamente recaiga en los desarrolladores. De hecho, gran parte de una implantación puede construirse mediante una GUI Web propia

denominada OpenL Tablets WebStudio. Básicamente, el RE descansa en la especificación de reglas mayormente a través de archivos Excel. El motor por otra parte se basa en tres conceptos básicos:

- Reglas: Es un enunciado lógico que consiste en condiciones y acciones. Si una regla es llamada, y se cumplen las condiciones, entonces se ejecutan las acciones (IF-THEN). Además de ejecutar las acciones, las reglas también pueden retornar datos de salida al llamador.
- Tablas: Los datos y las reglas en OpenL Tablets se presentan mediante tablas. Las tablas son únicas dentro de un *Proyecto*, y se denotan por su nombre y parámetros de entrada. Se permite el versionado de tablas que comparten nombre y parámetros de entrada. La riqueza en cuanto a especificación de reglas del motor radica en la posibilidad de definir diferentes tipos de tablas.
- Proyecto: Representa un contenedor para todos los recursos necesarios para procesar datos con reglas, tales como archivos Excel (llamados *módulos*), código Java adicional, dependencias a librerías, etc.



Vista general de OpenL Tablets

La figura anterior muestra el ciclo de vida típico de un proyecto con OpenL Tablets. El experto de dominio crea un Proyecto mediante WebStudio, y opcionalmente, un equipo de desarrollo provee soporte en caso de ser necesarias reglas con soporte de configuración complejo o funcionalidades personalizadas. El experto crea tablas correctamente estructuradas en archivos Excel que representan las reglas y la información necesaria. Aquí

es donde radica fundamentalmente la dificultad, debido a que existen un sinnúmero de convenciones para especificar las tablas, sus tipos, y la información contenida. Es decir, el experto de dominio debe especializarse en estos elementos de especificación, los cuales por otra parte están muy bien documentados. Luego, el experto también puede realizar test de unidad e integración mediante un tipo de tabla especial (Test). Los desarrolladores, por otra parte, pueden utilizar las reglas creadas en el sistema mediante una API o bien servicios Web Restful que son creados automáticamente para cada regla. Finalmente, los expertos pueden agregar, borrar o modificar reglas, para lo cual OpenL Tablets ofrece un soporte sólido para reflejar los cambios, por ejemplo publicar nuevas versiones de los servicios Web.

3.1 - Algoritmo de reconocimiento de tablas

El corazón del RE es un algoritmo que busca reglas a partir de tablas en archivos Excel. Es fundamental que dichas tablas respeten las convenciones de OpenL Tablets, de lo contrario son ignoradas por el algoritmo.

OpenL Tablets explota la idea de workbooks y worksheets de Excel, que pueden ser representados y mantenidos en varios archivos Excel. La herramienta, por otro lado, no utiliza el soporte de fórmulas de Excel, sino que los cálculos son representados en sintaxis propia que es completamente diferente. Las worksheets pueden estar dentro de un workbook, y cada worksheet comprende una o más tablas. Los workbooks pueden comprender tablas de diferentes tipos OpenL Tablets, cada uno con lógica de negocio particular.

Basado en esta entrada, el algoritmo de reconocimiento de tablas busca en cada worksheet para intentar identificar tablas lógicas, las cuales deben posicionarse de acuerdo a ciertas convenciones. El parseo de tablas se realiza de izquierda a derecha y de arriba hacia abajo. La celda izquierda superior de cada tabla determina el tipo OpenL Tablets, que debe contener un string dentro de los tipos soportados: Decision Table, Datatype Table, Data Table, Test Table, Run Table, Method Table, Configuration Table, Properties Table, Spreadsheet Table, TBasic Table, Column Match Table, Constants Table, Table Part. Finalmente, el algoritmo determina el ancho y largo de la tabla en base a las celdas completas.

3.2 - Ejemplo de tipos de tablas

Para ilustrar el concepto de tipo de tablas, veamos a continuación ejemplos del tipo de tabla DecisionTable, que describe reglas que modelan situaciones donde el estado de un número

de condiciones determinado determina la ejecución de un conjunto de acciones y valor de retorno. Es el tipo de tabla más básico y popular de OpenL Tablets.

Rules String Hello (Integer hour)		
C1	C2	RET1
min <= hour	hour <= max	greeting
Integer min	Integer max	String greeting
From	To	Greeting
0	11	Good Morning
12	17	Good Afternoon
18	21	Good Evening
22	23	Good Night

Ejemplo de tabla de decisión

El ejemplo mostrado representa una tabla de decisión, donde la primer fila debe tener el formato <table_type> <return_type> <table_name> <parameters>, donde table_type en este ejemplo es 'Rules'. Los tipos de datos de los parámetros pueden ser tipos wrapper de Java (String, Integer, Date, etc.). Además, se han definido un condición por columna (C1 y C2) y un valor de retorno. Cada columna de datos de entrada (From y To) tiene asociado un nombre lógico (min y max) y tipo de dato para poder ser manipulado por las expresiones asociadas a su vez a C1 y C2. Como puede verse, las expresiones también pueden referenciar de forma directa a los parámetros de la regla (hour en este caso).

Los tipos de datos también pueden ser de tipo complejos (records) los cuales pueden definirse con el tipo de tabla Datatype. A su vez, las tablas Datatype permiten crear modelos de datos propios en caso de dominios de aplicación con entidades sofisticadas. Así, las mismas permiten crear a través de mecanismos de herencia jerarquía de tablas Datatype, donde también se modelan elementos simples con tipos de datos básicos y se definen valores por defecto. Ahondando en esto, por ejemplo, podría definirse en un worksheet una tabla con el tipo de dato Corporation:

Datatype Corporation		
String	corporationID	
String	corporationFullName	
Industry	industry	other
Ownership	ownership	private
Integer	numberOfEmployees	1
FinancialData	financialData	_DEFAULT_
QualityIndicators	qualityIndicators	_DEFAULT_

donde se tienen campos con tipos primitivos, enumeraciones (Industry y Ownership) con sus respectivos valores por defecto ("other" y "private"), y tipos de datos extraídos de tablas

Datatype (FinancialData y QualityIndicators). El keyword _DEFAULT_ indica que los valores por defectos para los campos de los elementos complejos se toman de la tabla que define el tipo de dato complejo, por ejemplo:

Datatype FinancialData		
Date	reportDate	01/01/2010
Double	cashAndEquivalents	0
Double	inventory	0
Double	currentAssets	0.0001
Double	currentLiabilities	0.0001
Double	equity	0
Double	revenue	0.0001
Double	operatingProfit	0
Double	monthlyCashTurnover	0
Double	monthlyAccountsTurnover	0

En el siguiente ejemplo, mediante el tipo de tabla SmartRules, se modela el caso de una regla que basado en dos parámetros de entrada, que se asocian automáticamente a las dos primeras columnas de la tabla, construye como retorno un tipo de dato personalizado (definido en otra tabla), asociando de izquierda a derecha las columnas Min Type, Min Rate, Max Type y Max Rate a los elementos del tipo de dato DiscountSet:

SmartRules DiscountSet **VehicleDiscount1** (AirbagType airbagType, Boolean hasAlarm)					
Air Bags	**Alarm**	**Min Type**	**Min Rate**	**Max Type**	**Max Rate**
Driver		percent	12%	percent	14%
Driver&Passenger		percent	14%	percent	16%
	Yes	percent	10%	percent	11%
		flat	$10	flat	$15

En este caso, la evaluación de la regla se hace tomando como entrada una enumeración (AirbagType) y un booleano, y OpenL Tablets buscará el primer par de celdas Air Bags-Alarm que matcheen con los parámetros. Por defecto, las celdas booleanas están en “No”. También, es posible retornar arreglos de tipos de datos primitivos o complejos (con la notación DiscountSet). En este caso, los elementos del arreglo de retorno serán las filas que hagan match con los parámetros de entrada.

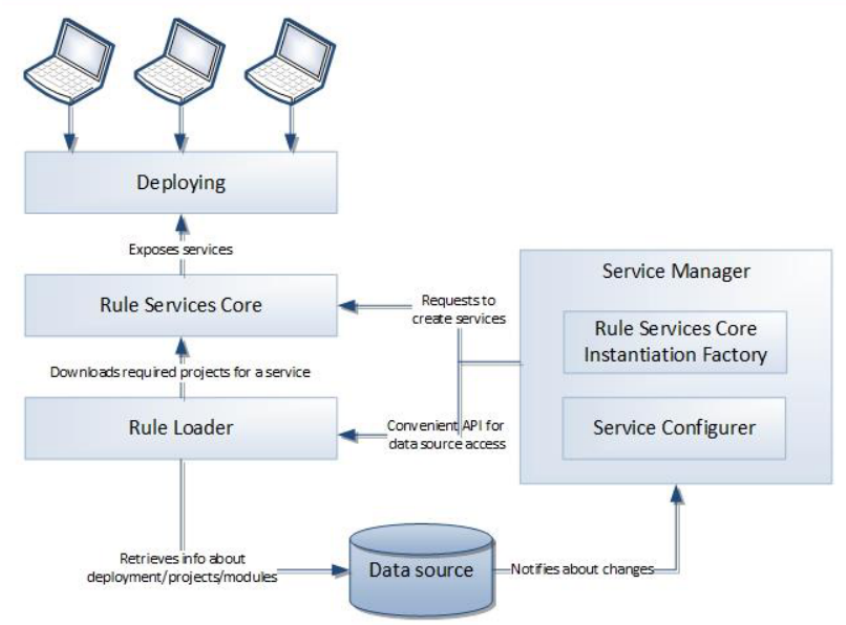
En el siguiente ejemplo, se muestra otra tabla SmartRules:

SmartRules Double **DriverPremium** (Driver driver, Double additionalCharge)			
Driver Type	**Years Driving Experience**		**Premium**
Principal	0	10	$600
	10	40	$550
Occasional	0	15	$750
	15	40	$700
Excluded, Non-Driver			= NonDriverPremiumByAge (driverType, age)
			= $600 + additionalCharge

En este caso, se muestra un ejemplo diferenciado donde se han colapsado filas para especificar las reglas. OpenL Tablets interpreta automáticamente las combinaciones de Driver Type y el rango de años de experiencia (ejemplo <Principal, [0,10)> y <Principal, (10,40]>). Si el Driver Type parámetro (driver) no es Principal u Ocasional, o bien es pero no entra en los rangos especificados, entonces se aplica la fila asociada a “Excluded” o “Non-Driver”. Sino, se aplica la regla de la última fila. Por otro lado, puede observarse que el valor de retorno Double puede ser un valor fijo, llamado a una función definida en código Java, o bien la suma de una constante más el parámetro de entrada (additionalCharge). Es posible además adicionar snippets de código Java para acciones, siendo esta funcionalidad útil para analistas de dominio con conocimientos en programación, pero representa una mezcla de archivos Excel y código.

3.3 - Integración con Web Services

Un objetivo de diseño fuerte de OpenL Tablets es la posibilidad de publicar reglas de negocio como servicios Rest para ser consumidas por clientes, bajo un esquema de pasaje de parámetros (input de las reglas) y ejecución de acciones y/o obtención de output. Para esto, se provee una arquitectura flexible y configurable que soporta la creación y actualización automática de servicios Web basado en reglas especificadas en archivos Excel almacenados en diferentes fuentes de datos. Esta arquitectura tiene integración 100% con Spring.



Vista general de la arquitectura de OpenL Tablets Rule Services

De esta manera, OpenL Tablets Rule Services permite a) integrar el mismo conjunto de reglas en diversas aplicaciones que ejecutan en diferentes plataformas de software, b) utilizar diferentes fuentes de datos para almacenar las reglas (repositorio, base de datos, sistema de archivos, etc.), y c) exponer lógicamente diferentes Proyectos OpenL Tablets mediante uno o más servicios Web.

4 - Drools

Drools fue creado en 2001 y se publicó en SourceForge. Más adelante, en el 2005, JBoss lo adquirió y por último fue comprado por Red Hat en el año 2006. Desde ahí, el grupo KIE Group de Red Hat (siglas de Knowledge Is Everything) fue creado para integrar **reglas** y **planificación** automatizada en **procesos de negocio**. Es un proyecto paragüas introducido para unificar tecnologías relacionadas. El proyecto está compuesto por los siguientes componentes:

- **Drools**: Business Rule Management System
- **jBPM**: Business Process Management Suite (anterior Drools Flow)
- **Optaplanner**: Constraint Solver System (anterior Drools Planner)
- **AppFormer**: UI Development Environment

De esta manera, Drools es en sí una herramienta dentro de un Workbench complejo, pero que puede ser utilizado como componente individual. En términos de los artefactos de

software de los componentes Drools y jBPM, que son usualmente utilizados en conjunto, se tienen:

- **Drools Engine**: Drools Expert es el motor de reglas en sí y Drools Fusion realiza procesamiento de eventos complejo (CEP). Contiene librerías de tipo JAR.
- **Business Central Workbench** (KWB): Es una aplicación web y un repositorio para manejar todo el software y los activos jBPM. Consiste en un archivo WAR para desplegar en el servidor de aplicaciones JBoss.
- **Herramientas jBPM**: Son plugins para el Eclipse IDE y que aportan soporte para Drools, jBPM (motor de workflows) y Guvnor. Este último es básicamente un repositorio centralizado de reglas de negocio que permite gestionarlas con una interfaz Web. Contiene librerías de tipo JAR.
- **KIE Execution Server** (KES): Es un servidor de ejecución independiente para ejecutar reglas utilizando servicios Rest y JMS (Java Message Service). Consiste en un archivo WAR para desplegar en el servidor de aplicaciones. JBoss provee una imagen Docker para iniciar fácilmente este componente[2].

4.1 - Estructura y reconocimiento de reglas

```
Package
Import
Declare

Rule "nombre de la regla"
    <atributo>
When
    <Condición> <Evento 1, Evento 2, Evento 3...>
Then
    <Acción> <Mensaje, Alerta, Datos generados...>
End
```

La ilustración anterior muestra la estructura de una regla Drools, expresada en DRL (Drools Rule Language)[3]. En una regla, una condición representa una restricción o filtro. Estos filtros analizarán la información del dominio para encontrar datos que cumplan con los criterios.

[2] https://hub.docker.com/r/jboss/kie-server
[3] Es posible utilizar también archivos Excel, de forma similar a OpenL Tablets: https://www.baeldung.com/drools#2-decision-tables

Cuando se obtiene un grupo de datos que coinciden con la condición, se programa una acción.

Las reglas Drools se basan en la programación declarativa mediante un conjunto de artefactos de código individuales, que conjuntamente pueden expresar lógica de negocio compleja sin tener que describir explícitamente el flujo de instrucciones global para tratar todas las condiciones. Dichos artefactos se compilan para (re)crear un *árbol de flujos/decisiones* basado en el algoritmo de inferencia Rete.

Rete es un popular algoritmo de reconocimiento de patrones, que en base a las reglas definidas y posibles acciones derivadas (inferidas) se genera un árbol de decisiones. Rete es hoy en día la base de muchos famosos *sistemas expertos* (CLIPS, Jess, Drools, Soar). Su enfoque se basa en crear una estructura completa de nodos de navegación basados en un conjunto de reglas (la base de conocimiento). Los nodos son relacionados en base a las consecuencias que se podrían inferir de los estados de los objetos que intervengan en la ejecución y vaya circulando por los nodos.

Este enfoque, aunque define más flujos posibles y, por lo tanto, hace mayor uso de memoria, otorga una ejecución más rápida del motor en comparación con un enfoque “tradicional” en el cual se comprobarán todas las reglas cada vez para saber si se tienen que ejecutar o no. Esto hace que el motor sea más escalable a costa de utilizar más memoria para almacenar total o parcialmente el árbol de decisión.

4.2 - Construcciones DRL básicas

```
package myAppRules;

import com.dppware.droolsDemo.bean.*;

dialect  "mvel"

rule "Adjust Product Price"
    when
     $p : ProductPrice(basePrice > 2 )
    then
     System.out.println("EJECUTANDO -Adjust Product Price- para el
producto [" + $p + "]");
end
```

Ejemplo de regla Drools

Desde el punto de vista de la API de Drools, una entidad importante son los **Facts**, es decir, los argumentos que entran al motor de reglas. Básicamente son POJOs. A continuación se explora las capacidades de creación de reglas basadas en Facts, bajo el soporte de Drools, y en base a un ejemplo sencillo (un POJO ProductPrice que recibe un número como parámetro al constructor). De la figura anterior se desprenden las secciones:

- **package**: es una agrupación lógica de reglas. **No tiene que ver con paquetería física.** Debe entenderse como un namespace, donde grupos de elementos tienen relación (globales, functions y otros elementos de especificación de Drools). Los nombres de las reglas deben ser únicos dentro de un mismo package (namespace).
- **import**: Definiciones de clases que necesitará Drools en la compilación de las reglas y su ejecución. Por defecto, suele indicarse que Drools importa siempre el paquete java.lang.*.
- **rule**: Es el bloque de código que indica el inicio (rule) y el fin (end) de una regla.
- **dialect**: Es el tipo de lenguaje usado para las definiciones dentro de las reglas. Los dos más extendidos son:
    - **"mvel"**: (MVFLEX Expression Language): Es un lenguaje declarativo sencillo y su única finalidad es hacer el código más legible. Su uso es casi extendido exclusivamente a la sección RHS (Right Hand Side) o la condición de una regla. Mvel también nos permite asignacion de variables (en el scope de una

regla) de manera sencilla ($varName), así como la definición de nuevos tipos (classes) de manera sencilla a nivel de package.

- **"java"**: Es posible incluir sintaxis Java dentro del .drl. Su única restricción es que solo se puede usarse en el LHS (Left Hand Side) de la regla, es decir, en la cláusula then.

Drools soporta en la cláusula when una lista importante de operadores para construir condiciones (> , < , >=, =<, || , && , == , % , ^, contains, not contains, memberof, not memberof, matches (regExp), not matches (regExp), startswith, etc.), por ejemplo:

$p : ProductPrice(((basePrice / 5) == 1) && ((basePrice % 5) == 0))

Aplicar el operator && (AND lógico) es equivalente a listar condiciones línea por línea dentro de la cláusula *when*. Existen además una evaluación de condición de forma explícita con la palabra reservada “eval”, que permite condicionar la ejecución en base a un valor booleano (**eval({true|false})**:

```
when
 eval(true)
 eval ($p.isZeroPrice()) //por ejemplo link a un método booleano interno
de la clase
 eval(callMyCustomFunctionThatReturnsABoolean)
 then
```

En ocasiones es necesario modificar el estado del Fact (POJO) producto de la ejecución de la/las acciones asociadas a una regla que matchea. En este caso, se usa la palabra reservada “modify”. En el ejemplo siguiente, se reduce en uno el valor del Fact mediante un setter del POJO:

```
import com.dppware.droolsDemo.bean.*;
dialect  "mvel"
rule "Adjust Product Price"
 when
     $p : ProductPrice(basePrice > 2 )
    then
     modify($p){
      setBasePrice($p.basePrice - 1);
     }
     System.out.println("el precio ajustado es " + $p.basePrice);
end
```

El comportamiento en runtime de una regla se puede restringir. Para ello, deben utilizarse usar los **rule attributes** que ofrece Drools[4]. Estos atributos se definen justo debajo al empezar una regla:

```
rule 'rulename'
 //rules Atributtes (availables: no-loop, salience, ruleflow-group, lock-
on-active, agenda-group,
 // activation-group, auto-focus, dialect, date-effective, date-expires,
duratio, timer, calendars
 when:
 ...
 then:
 ...
end
```

Los dos **rule attributes** más usuales son no-loop y salience. El caso de no-loop puede ejemplificarse con la regla anterior que descuenta en uno el precio del ProductPrice si este es mayor que 2. Si se alimenta el motor con ProductPrice(5), la regla se invocará varias veces, hasta que la condición no se cumpla. El no-loop evita este comportamiento y restringe la ejecución a una vez por llamado a la API de evaluación dado un Fact. Por otro lado, salience es un sistema de pesos que permite indicar prioridades sobre las ejecuciones de las reglas en caso de coincidencia. Esto es porque por defecto Drools evalúa las reglas en el orden de carga en el motor. Entonces, el siguiente código:

[4] https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch05.html#d0e3761

```
rule "Adjust Product Price"
 no-loop
 salience 1
    when
     $p : ProductPrice(basePrice > 2 )
    then
     System.out.println("EJECUTANDO -Adjust Product Price-");
end
rule "Sending Notification"
 no-loop
 salience 2
    when
     $p : ProductPrice(basePrice > 2 )
    then
     System.out.println("EJECUTANDO -Sending Notification-");
end
```

asegura una ejecución en un orden específico. Valores para salience más altos implican mayor prioridad. Es posible además utilizar valores negativos para asegurar prioridad mínima.

Otros rule attributes interesantes son date-effective and date-expires (para habilitar la operación de una regla y anularla, respectivamente, a nivel motor) o agenda-group, el cual permite definir agrupaciones de reglas y, al momento de evaluar un Fact, indicar al motor de Drools que agrupaciones deben ser consideradas en la evaluación (por ejemplo, solo aquellas de “MessagesGroup”):

```
rule "Welcome Message"
    agenda-group "MessagesGroup"
    when
 $m : Message()
 then
   System.out.println("Hello Mr.");
   m.setMessage("My message");
end
```

4.3 - Integración de especificaciones DRL y Java

Es posible crear *functions* importando métodos estáticos de clases propias o bien librerías externas. Normalmente esto sirve para incorporar una clase de utilidades, parseos o que realiza cualquier otro tipo de funcionalidad atómica y que se quiere utilizar dentro de reglas.

Por ejemplo, si se tiene una clase Utils y un método estático de impresión por pantalla *void prettyTraces(Object message)* puede utilizarse de la siguiente forma:

```
import com.dppware.droolsDemo.bean.*;

//Imported specified functions
import function com.dppware.droolsDemo.utils.Utils.prettyTraces;

dialect  "mvel"

rule "Adjust Product Price"
 when
     $p : ProductPrice(basePrice > 2 )
    then
     modify($p){
      setBasePrice($p.basePrice - 5);
     }
     prettyTraces("el precio ajustado es " + $p.basePrice);
end
```

También es posible definir funciones propias, mediante el keyword “function” y adicionando código que respete la sintaxis de Java inmediatamente después de la cláusula dialect:

```
function Integer calculateIncrement(Integer value, int quantity) {
   return value + quantity;
}
```

Es posible también definir nuevas clases (tipos) dentro del ecosistema de manera declarativa y sin compilación previa, ya que el compilador Java leerá la definición e introducirá dinámicamente las definiciones para la posible instanciación en runtime. La declaración de nuevos tipos se hace en la misma sección de declaración de las funciones, usando la palabra reservada "declare". Por defecto, con el uso de “mvel”, los getters/setters/toString/equals/hashCode serán añadidos a la definición del tipo:

```
declare Product
   code : int
   name : String
   description : String
ends
```

El compilador creará dos constructores por defecto (uno vacío y otro con todos los campos como argumentos). Luego, puede instanciarse y manipularse a través de getter el tipo creado desde el cuerpo de las reglas.

Lo hasta aquí visto evidencia la posibilidad de mejorar la modularización de los archivos .drl. Cobra sentido el concepto de “package” entonces para organizar los archivos .drl según contengan definiciones de tipos, funciones o reglas. Todos los .drl que pertenezcan al mismo package (namespace) están disponibles de manera “global” dentro del mismo namespace. No confundir con la idea de global como forma de inyectar a las reglas referencias a objetos instanciados desde la aplicación que llama a la API del motor Drools, lo cual se especifica mediante:

global com.dppware.droolsDemo.services.PushSubService publishTool;

En este caso, en el cuerpo de la regla, el objeto publishTool estará disponible para ser utilizado al igual que un POJO pasado como input al momento de evaluar las reglas.

5 - Evrete

Evrete es un motor de reglas moderno que implementa la JSR 94, está diseñado para tener una curva de aprendizaje baja para principiantes, y ser liviano en cuanto a requerimientos computacionales. Está además pensado para procesar datos etiquetados de múltiples fuentes de forma rápida, lo que lo hace especialmente apto para Big Data.
El motor se basa en una API y anotaciones de modo de utilizar y especificar reglas enteramente en Java, sin DSL de por medio. El corazón de una aplicación que usa el motor es KnowledgeService, que guarda la configuración, seguridad, pool de hilos, etc.
Como la mayoría de los motores, Evrete parsea *rulesets* a una representación optimizada en un repositorio, y luego, la aplicación obtiene una sesión a partir del repositorio. Por defecto, el repositorio opera en memoria, pero puede configurase para utilizar una base de datos relacionales, NoSQL, o archivos en disco. El motor distingue dos tipos de rulesets:

- Knowledge: Un conjunto de reglas pre-compiladas y optimizadas para ser utilizadas en futuras sesiones, y
- Rulesession: Una conexión en concreto para manipular datos dentro de una sesión, las cuales pueden ser stateless o stateful. La primera implica un modelo request/response, donde el input (conjunto de Facts) existe solo en el contexto de una ejecución. La segunda implica que el input persiste mientras la sesión esté abierta.

Cabe destacar que es posible adicionar reglas a un Knowledge o a una RuleSession, o bien, tener reglas que a su vez agregan otras reglas. Esto último debe ser tomado con cuidado cuando se tienen sesiones stateful largas, las cuales pueden hacer crecer indefinidamente el repositorio.

```
StatefulSession session = knowledge.newStatefulSession();
Customer customer = new Customer();
FactHandle customerHandle = session.insert(customer);

// Commit changes
session.fire();

// Direct memory inspection
customer = session.getFact(customerHandle);
customer.setLimit(57125.00);
session.update(customerHandle, customer)
session.fire();

//
//  More inserts, updates and deletes
//
session.close();
```

```
StatelessSession session = knowledge.newStatelessSession();
Customer customer = new Customer();
Invoice invoice = new Invoice();
FactHandle customerHandle = session.insert(customer);
FactHandle invoiceHandle = session.insert(invoice);

// Commit and inspect memory
session.fire((handle, object) -> {
    // Inspect memory objects
});

// Session has ended and can not be reused
```

Ejemplo de uso para sesiones stateful (arriba) y stateless (debajo). Los Facts deben insertarse, borrarse o actualizarse por medio de un “handle”, y en el caso de sesiones stateful, la API provee métodos para escanear el repositorio dado un handle.

Un caso de uso común implica crear objetos (POJOs), utilizarlos como input para las reglas, y obtener eventualmente el POJO modificado. Para esto, no es necesario utilizar handles combinado con escaneo de repositorio, sino que el insertAndFire de un POJO, combinado con una acción de la regla que matchea que lo modifique y actualice el repositorio es suficiente:

```
Knowledge knowledge = service
        .newKnowledge()
        .newRule()
        .forEach("$ai", AtomicInteger.class)
        .where("$ai.get() < 10")
        .execute(context -> {
            AtomicInteger obj = context.get("$ai");
            obj.incrementAndGet();
            context.update(obj);
        });

StatelessSession session = knowledge.newStatelessSession();

AtomicInteger obj = new AtomicInteger(0);
System.out.println("Pre-value:  " + obj.get()); // Prints 0
session.insertAndFire(obj);
System.out.println("Post-value: " + obj.get()); // Prints 10
```

Como puede apreciarse, las reglas pueden crearse utilizando fluent interfaces de Java, encadenando definición de una nueva regla (con nombre explícito o no), una cláusula *foreach* que selecciona el tipo de dato de entrada y establece nombre lógico ($ai), una o más condiciones *where*, y una única acción, la cual acepta una expresión lambda. Es posible también crear un RuleBuilder y programáticamente adicionar reglas, para luego ser usado en una sesión stateless o stateful.

En la figura anterior, la condición se ha especificado de forma literal (“$ai.get() < 10”) - posibilidad también disponible para las acciones- lo que puede controlarse utilizando where(this::condition, "$ai.get()") e implementando la condición en código Java:

```
boolean condition(IntToValue values) {
   return values.get(0) < 10;
}
```

Esta opción está disponible para las acciones, las cuales también pueden modificarse para una regla dada en forma dinámica, ya sea para Knowledge o RuleSession.

5.1 - Resolución de conflictos entre reglas

De forma similar a Drools, Evrete provee mecanismos para tratar los casos en que existan muchas reglas en un ruleset y sea necesario aplicarlas con cierto orden, y por otro lado,

tratar los casos en donde ciertas reglas tenga efectos secuncarios en la memoria de la sesión. Los mecanismos provistos son Salience, ActivationManager y ActivationMode.
Por defecto, Evrete asigna a las reglas un salience (prioridad) de 0, -1, -2, y así siguiendo en el orden que son insertadas en la base, siendo 0 el valor de mayor prioridad. Esto puede ser modificado utilizando el método setSalience() de las reglas.
El ActivationManager, por otro lado, funciona como un filtro para activación de reglas. Por defecto, todas las reglas están activas, pero es posible asociar un meta-parámetro a las reglas "ACTIVATION-GROUP" y luego proveer un ActivationManager personalizado para activar/desactivar grupos específicos a lo largo de las sesiones:

```
// Knowledge-wide approach
Knowledge knowledge = service
    .newKnowledge()
    .setActivationManagerFactory(SampleActivationManager.class)
    .newRule("Rule 1")
    .salience(20)
    .set("ACTIVATION-GROUP", "AG1")
    // .....

StatefulSession session = knowledge.newStatefulSession();

// Overriding activation manager for a session
session.setActivationManager(new SampleActivationManager());
```

Finalmente, se tiene el soporte de ActivationMode. Básicamente, llamar al método fire para disparar la evaluación de reglas implica activar un algoritmo interno de Evrete, que representa un loop como sigue:

while [cambios en la memoria de la sesión] **do**:

- se crea una agenda o lista ordenada de reglas afectada por el cambio
- **for** cada regla de la agenda **do**:
    - Activar la regla si no la filtra el ActivationManager
    - **if** no hubo cambios en la memoria **then**
        - continuar con la próxima regla
    - **else**
        - tratar el cambio en el medio de la evaluación

Si ninguna de las reglas modifican la memoria, los únicos cambios serán los iniciales (al agregar los Facts input), en cuyo caso el while tendrá un único ciclo. Sino (rama else) deberá configurarse una estrategia de resolución de conflictos. Esto se relaciona con el concepto de *Working Memory Action (WMA)*, o la intención de modificar algo en la memoria

de la sesión. Cuando se hace un insert/update/delete externo o desde la acción de una regla, no se aplican inmediatamente, sino se agregan en una lista WMA. En el modo CONTINUOUS, el motor ejecuta todas las reglas de la agenda. Cuando se alcanza el fin de la agenda, las modificaciones se vuelven la nueva lista inicial WMA. Cada regla de la agenda por ende no es consciente de los WMA de otras reglas aplicadas anteriormente. En el modo DEFAULT, se utiliza un mecanismo de versionado interno para los Facts, lo cual permite que reglas sucesivas vean los cambios de versionado hechos por reglas aplicadas antes. Los delete o cambio de versiones son excluidos del bloque de acción de las reglas siguientes. Este comportamiento lo hace similar a como funciona Drools.
Para configurar el modo se puede aplicar varios métodos:

```
// Global setting, each new session will inherit this choice
knowledge.setActivationMode(ActivationMode.DEFAULT);

// As a new session create argument
StatefulSession session = knowledge.newStatefulSession(ActivationMode.DEFAULT);

// The strategy can be changed at any time
session.setActivationMode(ActivationMode.CONTINUOUS);
```

Como buenas prácticas, se recomienda prescindir en lo posible del salience, debido a que ocasiona problemas de mantenibilidad. Asimismo, reducir al mínimo el uso de delete y update (sobre todo update) desde las acciones de las reglas.

5.2 - Especificación mediante anotaciones

Evrete permite utilizar como reglas artefactos anotados con ciertas anotaciones. Siempre que estén en el classpath, es posible utilizar código fuente anotado, clases compiladas, o clases que estén dentro de un JAR. El artefacto es posible apuntarlo desde un archivo o un URL, por ejemplo:

```
KnowledgeService service = new KnowledgeService();
Knowledge knowledge = service
        .newKnowledge(
                "JAVA-SOURCE", // <-- name of the DSL implementation
                new URL("https://www.evrete.org/examples/PrimeNumbersSource.java")
        );
```

Al anotar con @Rule un método público, se declara como regla. Los argumentos del método son Facts, su nombre el nombre de la regla (opcionalmente se puede modificar en @Rule), y el cuerpo la acción de la misma. @Rule también permite configurar salience. Los métodos pueden incluir un @Where para expresar la condición, o un @MethodPredicate con

referencia a un método que retorna un booleano. Los métodos no anotados de la clase y variables de instancia pueden ser usados en condiciones, acciones, resolución de conflictos, etc. A continuación se expone un ejemplo que ilustra la mayoría de los conceptos discutidos:

```
@Rule
@Where(
    value = {"$i.paid == false", "$c.active == true"},
    asMethods = {
        @MethodPredicate(
            method = "testMethod1",
            descriptor = {"$i", "$c.id"}
        )
    }
)
public void myRule(Invoice $i, Customer $c) {
    // ....
}

public boolean testMethod1(Invoice i, int customerId) {
    return i.getCustomer().getId() == customerId;
}
```

6 - Prototipo ejemplo

Se desarrolló a los fines ilustrativos un prototipo ejemplo en el lenguaje Java, utilizando el framework Spring Boot. El prototipo expone una API CRUD y modela un subconjunto de las entidades de la plataforma de monitoreo de Virtual Sense S.A., esto es Device, Patient y Observation. Esta última contiene relaciones a las dos anteriores y modela una observación (valor más unidad) de un signo vital.

El prototipo incluye una integración simple con el motor Evrete discutido anteriormente, compuesto por una session stateless y una regla que reacciona ante el llamado a la API Rest que agregado una observación. La regla simplemente imprime por consola cierta información para el caso de que el paciente de la observación tenga un documento de cierto tipo. Efectivamente, a partir de esta integración el equipo de desarrollo puede continuar complejizando el modelo de entidades, las operaciones CRUD, y las reglas especificadas.