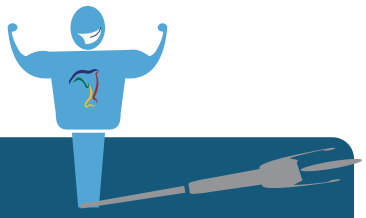
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

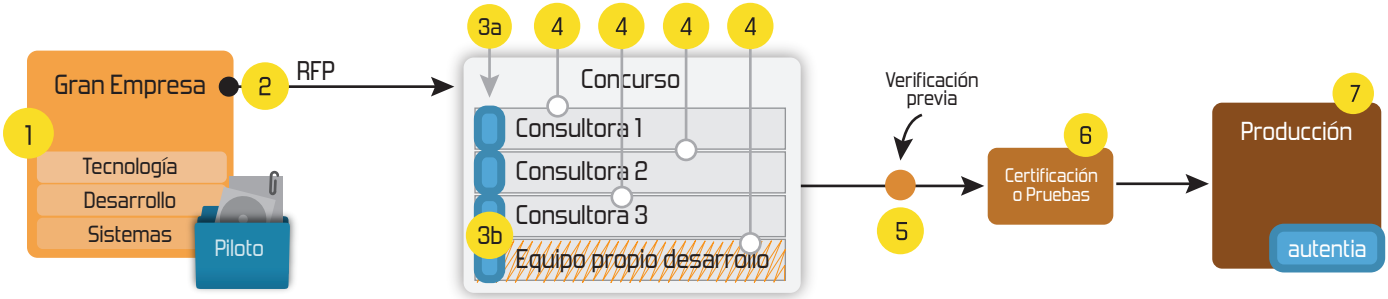
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Miguel Arlandy Rodríguez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2012-09-29

Creación de plantillas DSL con Drools.

0. Índice de contenidos.

1. Introducción

Como vimos en el anterior tutorial Drools es un BRMS que nos permite centralizar y gestionar reglas de negocio. Una de las ventajas (entre muchas) que ofrecen los sistemas de gestión de reglas de negocio es que facilitan el acceso y la comprensión de las reglas al personal de la organización.

Los DSLs (Domain Specific Languages) son una forma de crear un lenguaje de reglas específico para un contexto dado. Con el uso de un DSL (o varios) **se esconden los detalles de la implementación y se puede asimilar mucho mejor la verdadera lógica de las reglas**, sobre todo para personal no técnico. Drools permite el uso de DSLs para codificar nuestras reglas de negocio.

En este tutorial intentaremos explicar cómo utilizar DSLs (Domain Specific Languages) con Drools para facilitar la comprensión de nuestras reglas de negocio.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS Mountain Lion 10.8
- Entorno de desarrollo: Eclipse Juno 4.2.
- Drools 5.4.0.Final

3 Creando DSLs.

Crear un lenguaje específico con Drools (DSL) es muy sencillo. Basta con **crear un fichero con extensión .dsl donde definiremos las expresiones**, que normalmente serán expresiones del lenguaje natural, y **reemplazar dichas expresiones en nuestro fichero de reglas** .drl o .dslr.

Me gustaría destacar que el uso de DSLs **no tiene ninguna implicación en el tiempo de ejecución de nuestras reglas**.

En este punto nos centraremos en cómo crear sentencias específicas del lenguaje y, para ello, vamos a explicar los diferentes aspectos a tener en cuenta.

3.1 El formato.

Nuestro fichero .dsl no es más que un fichero en texto plano. Está compuesto por líneas que definen sentencias que pueden ser usadas en nuestro fichero de reglas. Cada una de las líneas de nuestro fichero .dsl tiene el siguiente formato:

```
[ámbito][tipo] expresión DSL=sentencia DSLR
```

Donde:

- **ámbito:** indica la parte de la regla donde irá la expresión. Acepta los siguientes valores:
  - *condition* (o when): indica que la expresión será usada en la parte condicional de nuestra regla (parte when).
  - *consequence* (o then): indica que la expresión será usada en la parte de la acción a realizar de nuestra regla (parte then).
  - *keyword*: indica que la expresión será en cualquier parte de nuestra regla.
  - *: indica que la expresión será usada indistintamente en la parte condicional (when) o en la acción (then) de nuestra regla.
- **tipo**: es una especie de categorización de la expresión. En teoría sirve para indicar el objeto sobre el que actua la expresión. Es opcional y su valor no tendrá ninguna consecuencia.
- **expresión DSL:** es la expresión con la que nos referiremos DSLR que mapearemos. Suele ser una expresión en lenguaje natural.
- **sentencia DSLR:** mapeada sobre la expresión anterior.

Algunos ejemplos muy sencillos pueden ser:

```
[condition][]Cuando haya un pedido=order : Order()

[keyword][]GOLD=Customer.GOLD_CUSTOMER
```

Como se ve en la primera sentencia del ejemplo, la parte derecha del signo igual contiene una sentencia DRL que, como vimos en el anterior tutorial, significa que asignamos un objeto "Order" que entre en la memoria de trabajo a la variable "order". Esto expresado en el lenguaje natural sería algo como: *Cuando haya un pedido*. Con [condition] estamos diciendo que esta sentencia será usada en la parte "when" de nuestra regla. Recordemos que las reglas de negocio eran expresiones del tipo: **Cuando** pase algo **entonces** se hace lo que sea...

3.2 Las variables.

Por supuesto podemos añadir variables a nuestras expresiones. Para ello basta con darles un nombre y ponerlas entre llaves. Por ejemplo:

```
[*][]Log {mensaje}=System.out.println("{mensaje}");

[keyword][]prioridad {prioridad}=salience {prioridad}
```

En el ejemplo vemos que cuando en nuestra regla aparezca **Log hola**, será lo equivalente a ejecutar System.out.println("hola"); lo que nos mostrará la cadena de caracteres por pantalla. Lo mismo para la prioridad.

3.3 Las expresiones regulares.

También podemos utilizar expresiones regulares en nuestras expresiones DSL. Podemos hacerlo tanto en la propia expresión como en las variables que usemos.

Imaginemos que queremos que en nuestra expresión DSL "Cuando haya un pedido", queremos que la sentencia pueda comenzar tanto por mayúsculas como minúsculas. Es sencillo, basta con remplazar la letra "C" por la que empieza nuestra expresión, por la expresión regular "[C|c]" (igual a C ó c) y listo.

Supongamos que en nuestra sentencia "prioridad {prioridad}" queremos asegurarnos que la variable sea numérica. Para definir variables que deben cumplir con una expresión regular, dentro de las llaves, escribimos el **nombre de nuestra variable**, luego **dos puntos** (:) y la **expresión regular** (en Java) que debe cumplir. En nuestro caso hay que definir la variable prioridad como: **nombre:expresion_regular** de la siguiente forma {prioridad:\d+}, o lo que es lo mismo, nuestra variable prioridad debe ser un número compuesto por uno o más dígitos. Nótese que la "d" de la expresión regular (que indica que la cadena debe ser numérica) debe escaparse con \.

```
// son válidas las expresiones: "Cuando haya un pedido" y "cuando haya un pedido"
[condition][][C|c]uando haya un pedido=order : Order()

// la prioridad debe estar precedida por un número
[keyword][]prioridad {prioridad:\d+}=salience {prioridad}
```

3.4 Las funciones.

Además podemos utilizar ciertas funciones predefinidas sobre las variables de nuestras expresiones DSLR. Son las siguientes:

- **uc:** Uppercase. Convierte todas las minúsculas en mayúsculas.
- **lc:** Lowercase. Convierte todas las mayúsculas en minúsculas.
- **ucfirst:** Capitalize. La primera letra en mayúscula y las demás en minúsculas.
- **a?b/c:** Compara la variable con la cadena "a", si es igual la reemplaza por "b", sino la reemplaza por "c".

Para aplicar funciones a nuestras variables de la parte derecha de la expresión (DSLR) debemos escribir el **nombre de la variable** seguido de **una exclamación** (!) y **la función** a aplicar. En el siguiente ejemplo se ve mucho mejor:

```
[*][]Log {mensaje}=System.out.println("{mensaje!uc}");
// al escribir Log Miguel la salida sería MIGUEL

[*][]Log {mensaje}=System.out.println("{mensaje!lc}");
// al escribir Log Miguel la salida sería miguel

[*][]Log {mensaje}=System.out.println("{mensaje!ucfirst}");
// al escribir Log miguel la salida sería Miguel

[*][]Log {mensaje}=System.out.println("{mensaje!true?Verdadero/Falso}");
// al escribir Log true la salida sería Verdadero
// con cualquier otra cosa sería Falso
```

3.5 Añadiendo condiciones.

» Introducción a Drools.

» Jugando con JSON en Java y la librería Gson

» WebSockets con Java y Tomcat 7

» Introducción a Apache ActiveMQ

» Transiciones y animaciones con CSS3

Una de las características más interesantes, bajo mi punto de vista, es la de ir añadiendo condiciones sobre un mismo objeto en diferentes expresiones.

Imaginemos que tenemos que evaluar un objeto como el que define la siguiente clase:

```
public class Persona {

    private int edad;

    private String nombre;

    private String sexo;

    // getters y setters...
}
```

Podemos querer evaluar si existe una persona en nuestra memoria de trabajo. Además podemos querer evaluar si existe una persona con determinada edad o una persona que se llame de una forma y que tenga tantos años o una persona de cierta edad de tal sexo, etc, etc, etc... La casuística es muy elevada y cuantas más propiedades contenga nuesto objeto, mucho más.

Para solucionar esto, Drools nos permite **anidar condiciones aplicadas sobre un mismo objeto** de manera independiente. Únicamente debemos añadir un guión (-) al principio de nuestra expresión DSL para indicar que la condición será aplicada sobre el objeto que se evaluó anteriormente. Lo vemos en dos ejemplos.

Esta sería la **forma menos apropiada** de hacerlo:

```
[condition][]Cuando haya cualquier persona=persona : Persona()
[condition][]Cuando haya una persona de {edad} años=persona : Persona(edad == {edad})
[condition][]Cuando haya una persona que se llame {nombre} y con {edad} años=persona : Pers
[condition][]Cuando haya una persona con sexo {sexo} y con {edad} años=persona : Persona(se
// etc, etc, etc ...
```

Y esta sería la **forma más adecuada**, añadiendo condiciones:

```
[condition][]Cuando haya una persona=persona : Persona()
[condition][]- que tenga {edad} años=edad == {edad}
[condition][]- que se llame {nombre}=nombre == "{nombre}"
[condition][]- de sexo {sexo}=sexo == "{sexo}"
```

Y con esto bastaría con ir añadiendo las condiciones que quisiésemos en nuestro fichero DSLR:

```
Cuando haya una persona
- que tenga 25 años
- de sexo Femenino
```

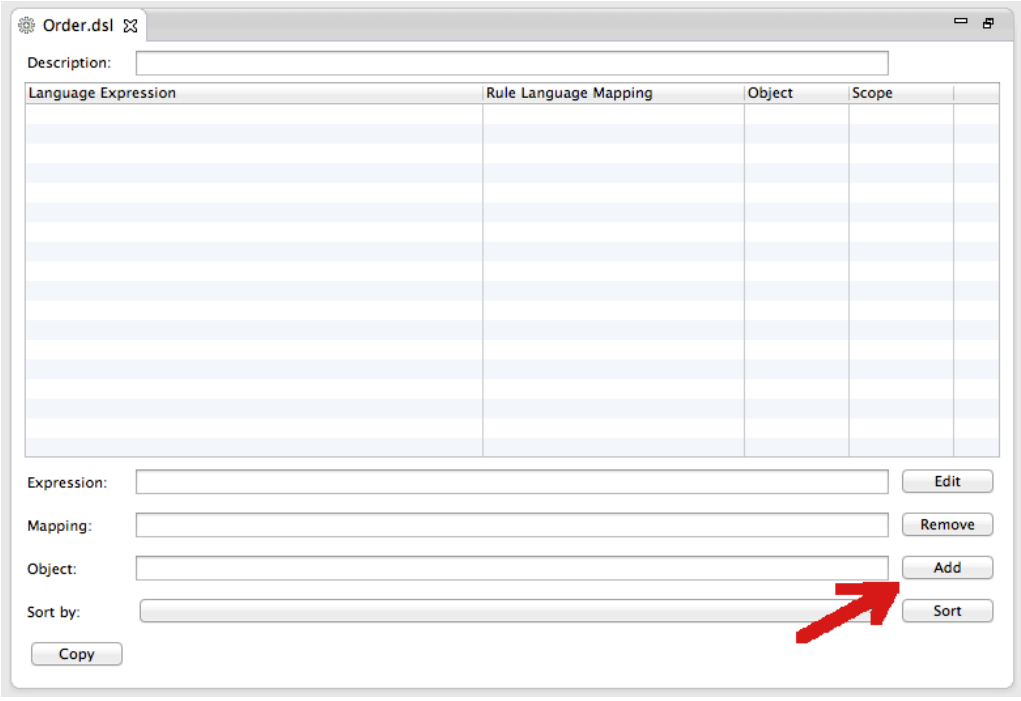
4. ¿Vemos un ejemplo?.

Vamos a ver un ejemplo con lo que hemos comentado en el punto anterior. Para ello nos basaremos en el ejemplo del anterior tutorial, donde creábamos las reglas necesarias para un pequeño sistema de gesión de descuentos y promociones.

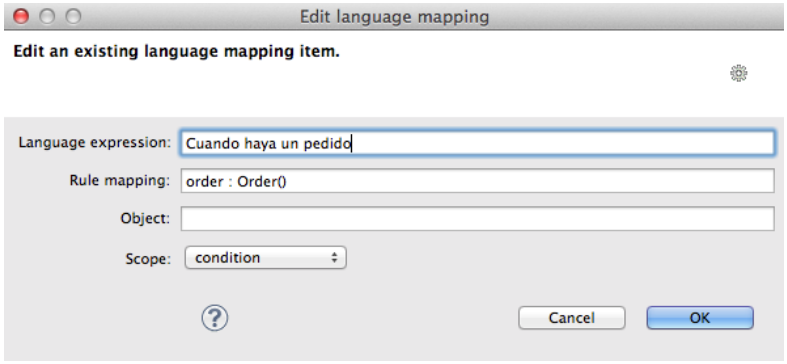
Para ello necesitaremos **dos ficheros**: el .dsl donde escribiremos nuestras expresiones DSL y el fichero .dslr, donde escribiremos las reglas en base al lenguaje natural.

Recordemos que Eclipse cuenta con un **plugin de Drools**. Lo utilizaremos para crear nuestro fichero Order.dsl. Personalmente, no me parece que este plugin aporte gran valor para la creación de DSLs, pero ya que lo tenemos, lo usamos.

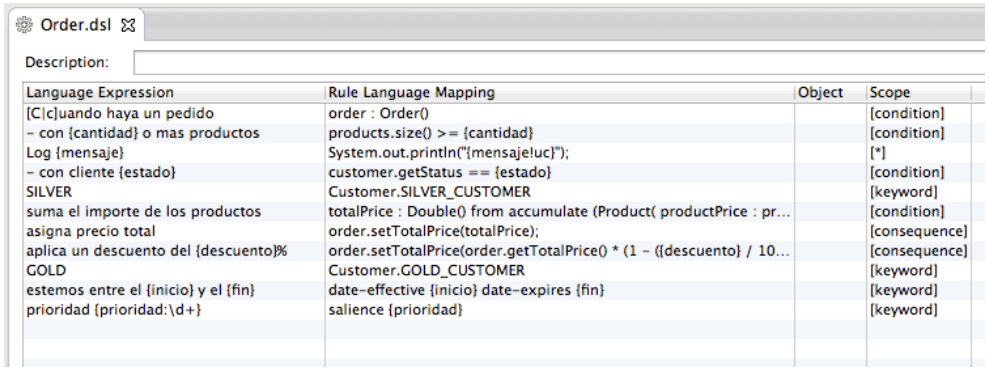
Lo primero que haremos será crear el fichero Order.dsl y, al abrirlo con Eclipse y el plugin instalado, nos aparecerá una pantalla como la siguiente que nos guiará en la creación de expresiones. También podemos abrirlo como un fichero de texto y escribirlas en texto plano.



Pulsamos sobre el botón "Add" para ir añadiendo nuestras expresiones al fichero.



Una vez que las hemos escrito todas el resultado sería algo como esto:



Language Expression	Rule Language Mapping	Object	Scope
[C|c]uando haya un pedido	order : Order()		[condition]
- con {cantidad} o mas productos	products.size() >= {cantidad}		[condition]
Log {mensaje}	System.out.println("{mensaje!uc}");		[*]
- con cliente {estado}	customer.getStatus == {estado}		[condition]
SILVER	Customer.SILVER_CUSTOMER		[keyword]
suma el importe de los productos	totalPrice : Double() from accumulate (Product(productPrice : pr...		[condition]
asigna precio total	order.setTotalPrice(totalPrice);		[consequence]
aplica un descuento del {descuento}%	order.setTotalPrice(order.getTotalPrice() * (1 - ({descuento} / 10...		[consequence]
GOLD	Customer.GOLD_CUSTOMER		[keyword]
estemos entre el {inicio} y el {fin}	date-effective {inicio} date-expires {fin}		[keyword]
prioridad {prioridad:\d+}	salience {prioridad}		[keyword]

Y el fichero en texto plano tendría este aspecto:

```
[condition][][C|c]uando haya un pedido=order : Order()
[condition][]- con {cantidad} o mas productos=products.size() >= {cantidad}
[*][]Log {mensaje}=System.out.println("{mensaje!uc}");
[condition][]- con cliente {estado}=customer.getStatus == {estado}
[keyword][]SILVER=Customer.SILVER_CUSTOMER
[condition][]suma el importe de los productos=totalPrice : Double() from accumulate (Produ
[consequence][]asigna precio total=order.setTotalPrice(totalPrice);
[consequence][]aplica un descuento del {descuento}%=order.setTotalPrice(order.getTotalPric
[keyword][]GOLD=Customer.GOLD_CUSTOMER
[keyword][]estemos entre el {inicio} y el {fin}=date-effective {inicio} date-expires {fin}
[keyword][]prioridad {prioridad:\d+}=salience {prioridad}
```

Ahora creamos el fichero de reglas DSLOrder.dslr y escribimos las reglas en función de las expresiones DSL del fichero anterior. Si lo comparamos con el fichero Order.drl del anterior tutorial, vemos que ahora las reglas son mucho más legibles para cualquier persona. El resultado sería este:

```
package com.autentia.tutorial.drools

expander Order.dsl

import com.autentia.tutorial.drools.data.*;

rule "Initial rule"
    prioridad 20
    when
        Cuando haya un pedido
        suma el importe de los productos
    then
        asigna precio total
end

rule "SILVER customer rule"
    prioridad 15
    when
        Cuando haya un pedido
        - con cliente SILVER
    then
        aplica un descuento del 5%
end

rule "GOLD customer rule"
    prioridad 15
    when
        Cuando haya un pedido
        - con cliente GOLD
    then
        aplica un descuento del 10%
end


rule "Number of products rule"
    prioridad 10
    estemos entre el "01-SEP-2012" y el "01-OCT-2012"
    when
        Cuando haya un pedido
        - con 10 o mas productos
    then
        aplica un descuento del 15%
end
```

Para lanzar el ejemplo lo haremos del mismo modo que en el ejemplo del anterior tutorial pero con la diferencia de que ahora añadimos a nuestro "KnowledgeBuilder" los ficheros .dsl y .dslr

```
private static KnowledgeBase readKnowledgeBase() {
    final KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
```

```
    kbuilder.add(ResourceFactory.newClassPathResource("Order.dsl"), ResourceType.DSL);
    kbuilder.add(ResourceFactory.newClassPathResource("DSLOrder.dslr"), ResourceType.DSLR
    if (kbuilder.hasErrors()) {
        for (KnowledgeBuilderError error : kbuilder.getErrors()) {
            System.err.println(error);
        }
        throw new IllegalArgumentException("Imposible crear knowledge con Order.dsl y DSL(
    }
    final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
    return kbase;
}
```

Lanzamos el ejemplo y todo perfecto... :)

```
Console  Declaration
<terminated> DSLOrderTest [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (
Cliente Customer [status=0, name=Cliente estandar] productos: 1 Precio total: 100.0
Cliente Customer [status=1, name=Cliente SILVER] productos: 1 Precio total: 95.0
Cliente Customer [status=2, name=Cliente GOLD] productos: 1 Precio total: 90.0
Cliente Customer [status=1, name=Cliente SILVER] productos: 10 Precio total: 807.5
```

5. Referencias.

- Introducción a Drools.
- Documentación de JBoss Drools.
- Human readable rules with Drools.

6. Conclusiones.

Como sabemos, una de las ventajas que nos ofrecen los sistemas de gestión de reglas de negocio como Drools es centralizan y facilitan la comprensión de la lógica de negocio de una organización. Si además creamos un lenguaje propio (DSL) para codificar estas reglas en base a un lenguaje de un contexto de una organización (lenguaje natural), la comprensión de la lógica será mucho más sencilla, y más sabiendo que no afecta al tiempo de ejecución de nuestras reglas.

Espero que este tutorial os haya sido de ayuda. Un saludo.

Miguel Arlandy

marlandy@autentia.com

Twitter: @m_arlandy