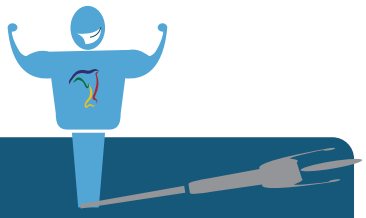
autentia

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...

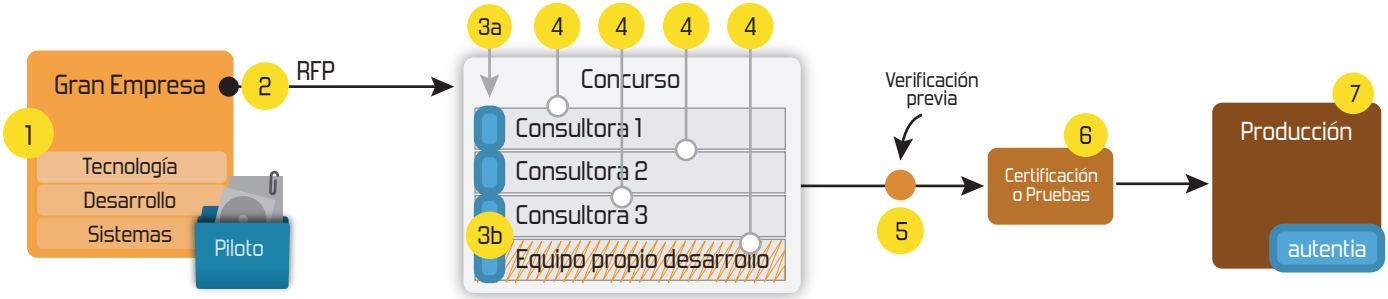
1. Desarrollo de componentes y proyectos a medida



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: 2012-09-19

Introducción a Drools.

0. Índice de contenidos.

- 1. Introducción.
- 2. Entorno.
- 3. Las reglas de negocio.
- 4. Drools y los sistemas de gestión de reglas de negocio.
  - 4.1 ¿Cuándo usar Drools?.
- 5. Dependencias y plugin de Eclipse.
- 6. Escribiendo nuestras primeras reglas en DRL.
  - 6.1 La regla "Initial rule".
  - 6.2 La regla "SILVER customer rule".
  - 6.3 La regla "GOLD customer rule".
  - 6.4 La regla "Number of products rule".
  - 6.5 Probando el ejemplo.
- 7. Referencias.
- 8. Conclusiones.

1. Introducción

En la mayoría de las empresas la lógica de negocio se encuentra dispersa por diferentes sitios: en el código de las aplicaciones, hojas de cálculo, en las mentes de los expertos en la materia, etc... Este hecho hace que, a la mayoría de las personas de la organización les sea complejo consultar y comprender las reglas que constituyen la base del negocio.

Los sistemas de gestión de reglas de negocio (BRMS) como Drools surgen ante la necesidad de **centralizar y gestionar la lógica de negocio**. Para ello, se codifica dicha lógica en forma de reglas de negocio, que sirven para tomar decisiones dentro de un contexto. Estas reglas se ejecutan dentro de un motor de reglas (BRM). Las reglas, por tanto, no sirven únicamente para representar la lógica de negocio, sino también para ejecutarla. Además, el hecho de que la lógica se encuentre codificada en una regla, hace más fácil su comprensión que cuando se encuentra en el código de una aplicación, sobre todo para el personal no técnico.

En este tutorial intentaremos explicar qué son las reglas de negocio, qué es un sistema de gestión de reglas de negocio, cuando es interesante su uso y cómo crear nuestras propias reglas con Drools.

Usaremos la versión 5.4, que es la última versión a fecha de publicación de este tutorial.

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. Las reglas de negocio.

Una regla de negocio es una **sentencia del tipo cuando/entonces** (when/then). Cuando se cumple una condición, se desencadena una acción (que puede ser cualquier cosa). Ejemplo: *"Cuando un cliente que realiza un pedido es VIP, entonces se le realiza un descuento del 10%"*.

Lo ideal es que las reglas de negocio únicamente sirvan para tomar decisiones sobre lo que se debe hacer en base a unos criterios (condiciones). En ningún caso (o al menos esa es mi opinión) deberían ejectutar las acciones que se desencadenarán tras la decisión. Sería como el caso de un rey de la edad media (y permitidme que se me vaya un poco la hoya en este ejemplo) al que le exponen el caso de un ladrón que ha robado gallinas a los vecinos de una aldea. El rey toma la decisión de que hay que cortarle una mano para que no lo vuelva hacer, pero él no es quien se la cortará (será un verdugo o quien sea) sino quien decide qué es lo que hay que hacer con el ladrón en base a lo que hizo. Eso traducido a una regla de negocio sería algo así como **cuando** alguien robe gallinas **entonces** que le corten la mano.

4. Drools y los sistemas de gestión de reglas de negocio.

Como hemos dicho anteriormente, las reglas de negocio de una empresa se suelen encontrar dispersas, ya sea codificadas en las aplicaciones, en una hoja de cálculo o en las mentes de los expertos del negocio.

El propósito de un sistema de gestión de reglas de negocio (BRMS) es centralizar todas esas reglas de negocio que se suelen encontrar dispersas en una empresa y que, a su vez, constituyen la verdadera inteligencia del negocio. Esto permitirá **poder acceder con suma facilidad a ellas y poder gestionar los cambios de que se produzcan en éstas con mayor rapidez**.

Normalmente estos sistemas de gestión de reglas, también conocidos como BRMS, suelen contar con una interface gráfica que permite trabajar de forma sencilla con las reglas del negocio. Esta característica es especialmente interesante ya que permite que los expertos del negocio (y no los técnicos) puedan gestionarlas de manera eficiente.

Los BRMS cuentan con un motor de reglas (BRM), que es el componente donde se ejecutan las reglas. Las ejecución de esas reglas, en base a unos datos de entrada, desencadenará una acción. Como dijimos en el punto anterior, lo ideal sería que la regla decidiese qué acciones hay que tomar en base a los datos analizados.

El BRM puede ser utilizado incorporándolo directamente en el código de nuestra aplicación. Sin embargo, también puede ser expuesto como un **servicio de toma de decisiones**, que probablemente sea su mejor enfoque. Lo que es lo mismo, un componente en nuestra arquitectura en el que se apoyen los distintos componentes de nuestra plataforma para saber cómo deben actuar en determinados casos. Si hablásemos de una arquitectura SOA, podríamos decir que el servicio de toma de decisiones sería **el cerebro de nuestro sistema**.

Drools es un sistema de gestión de reglas de negocio open source desarrollado por JBoss, bastante maduro y que cuenta con una documentación excelente. Sus principales características son:

- **Motor de reglas**: basado en el algoritmo Rete que permite que las reglas se ejecuten de manera muy eficiente. La gente de JBoss lo llama "Drools Expert".
- **Lenguaje DRL**: que permite la creación de reglas de manera sencilla. Permite también la creación de reglas en lenguaje específico usando DSL. Permite el tratamiento de hojas de cálculo.
- **BRMS (Guvnor)**: permite gestionar las reglas de manera centralizada con una interface web.

4.1 ¿Cuándo usar Drools?

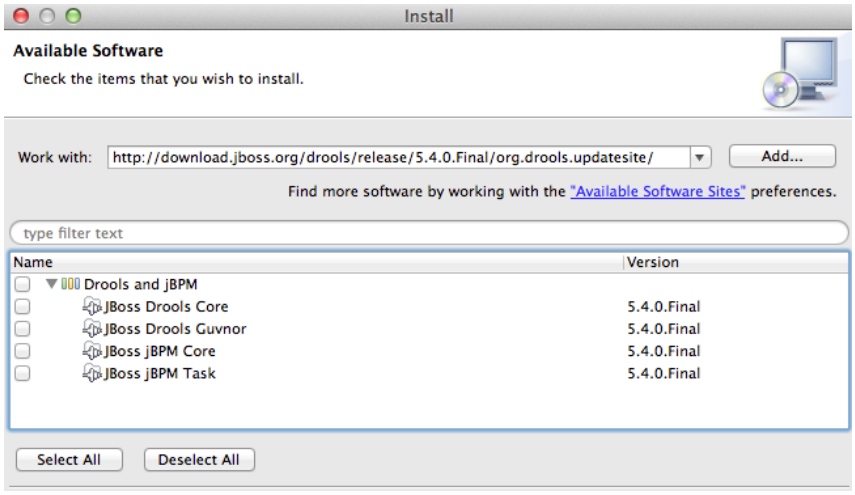
Pues el escenario ideal para usar Drools, y en general cualquier motor de reglas, es cuando queremos **separar la lógica de negocio de las aplicaciones**. Cuando queremos que esa lógica de negocio esté centralizada y sea gestionada por los expertos del negocio y no por el personal técnico. Si encima esa **lógica de negocio es muy cambiante** (sistema de incentivos, promociones, descuentos, etc...) el sistema de gestión de reglas es perfecto, ya que permite realizar cambios de forma realmente ágil mediante: la creación de nuevas reglas, su modificación, flujos de reglas, etc...

A continación veremos cómo empezar a trabajar con esta excelente herramienta.

5. Dependencias y plugin de Eclipse.

Aunque no es necesario, Eclipse cuenta con un plugin que nos facilitará el trabajo a la hora de escribir reglas de negocio. Podemos ver coloreadas las keywords (lenguaje DRL) y nos avisa de posibles errores de sintaxis entre otras cosas.

Para instalar el plugin en Eclipse: **Help > Install New Software >** http://download.jboss.org/drools/release/5.4.0.Final/org.drools.updatesite/. Podemos seleccionar únicamente los complementos de Drools (los dos primeros) o también los de JBPM.



A continuación añadiremos las dependencias necesarias para poder trabajar con Drools. Podemos hacerlo de dos formas: con Maven o sin él.

Si decidimos no usar Maven y tenemos el plugin instalado, necesitamos configurar nuestro drools-runtime. Que no es más que el conjunto de librerías que necesitará Drools para funcionar. Para ello: Preferencias > Drools > Installed Drools runtime.

» Jugando con JSON en Java y la librería Gson

» WebSockets con Java y Tomcat 7

» Introducción a Apache ActiveMQ

» Transiciones y animaciones con CSS3

» Comparando diferencias entre ficheros con java-diff-utils

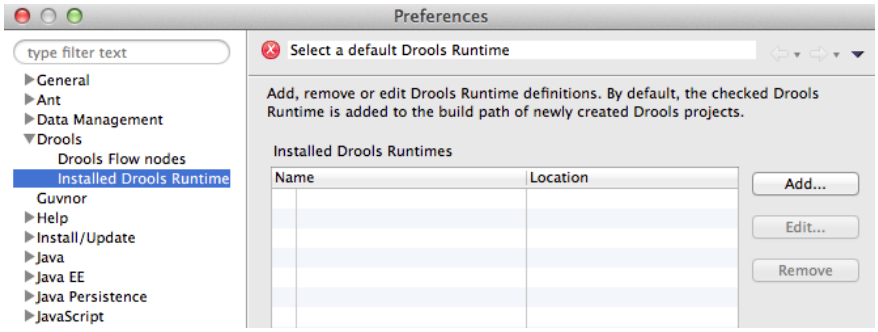
Categorías del Tutorial

SOA

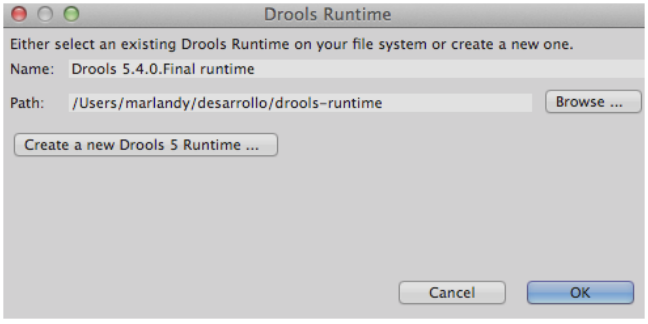
Otras Técnicas

Técnicas Avanzadas

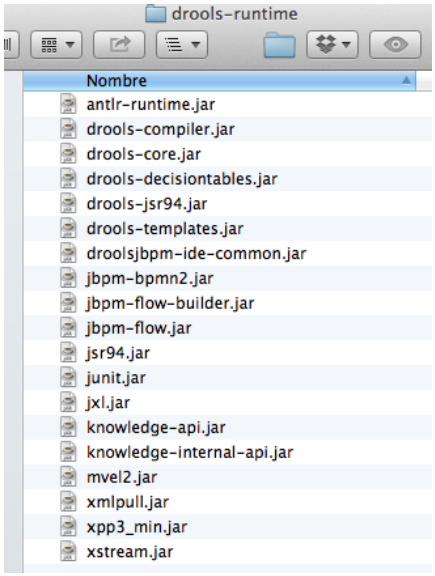
Últimas ofertas de empleo



Pulsamos sobre **Add...** y creamos una nueva pulsando sobre **Create a new Drools 5 Runtime** y seleccionando el directorio que queramos.



Automáticamente nos aparecerán ahí todas las librerías. Las añadimos al classpath y listo.



Para el que prefiera usar Maven (como será nuestro caso) aquí están las dependencias:

```
<dependency>
    <groupid>org.drools</groupid>
    <artifactid>knowledge-api</artifactid>
    <version>5.4.0.Final</version>
</dependency>
<dependency>
    <groupid>org.drools</groupid>
    <artifactid>drools-core</artifactid>
    <version>5.4.0.Final</version>
</dependency>
<dependency>
    <groupid>org.drools</groupid>
    <artifactid>drools-compiler</artifactid>
    <version>5.4.0.Final</version>
</dependency>
<dependency>
    <groupid>com.thoughtworks.xstream</groupid>
    <artifactid>xstream</artifactid>
    <version>1.3.1</version>
</dependency>
```

Pues ya tenemos todo preparado para empezar a trabajar... :)

6. Escribiendo nuestras primeras reglas en DRL.

En el ejemplo que vamos a ver, crearemos varias reglas de negocio para un **sistema de descuentos y promociones**. Imaginemos que queremos calcular el importe que se va a cobrar a un cliente que realiza un pedido. Para ello nos apoyaremos en un motor de reglas que nos calcule el precio final de un pedido en base a una serie de condiciones. En nuestro sistema, un pedido consta de un cliente que lo emite y una lista de productos que quiere comprar. Nuestros clientes están categorizados como: ESTANDAR, SILVER y GOLD.

Definiremos nuestro sistema de descuentos y promociones en base a una serie de variantes:

- A todos los **clientes que sean SILVER** se les aplicará un **descuento de un 5%** en su pedido.
- A todos los **clientes que sean GOLD** se les aplicará un **descuento de un 10%** en su pedido.

- A todos los clientes que, durante el **mes de Septiembre de 2012**, realicen un pedido de **10 o más productos** se les aplicará un **descuento de un 15%**.

Veamos cuales son los datos necesarios para poder simular el escenario.

La clase Order representa un pedido. Observemos que tiene 3 atributos:

- customer: que representa al cliente que emite el pedido.
- products: que representa la colección de productos que componen el pedido.
- totalPrice: que representa el importe final que se debe cobrar al cliente por su pedido. Este valor será el que calcule nuestro motor de reglas.

```
package com.autentia.tutorial.drools.data;

import java.util.ArrayList;
import java.util.List;

public class Order {

    private Customer customer;

    private List<Product> products;

    private double totalPrice;

    public Order(Customer customer) {
        super();
        this.customer = customer;
    }

    // getters y setters
}
```

La clase Customer representa el cliente que emite el pedido. Tiene 2 atributos:

- name: el nombre del cliente.
- status: que nos dice qué tipo de cliente es: ESTANDAR, SILVER o GOLD.

```
package com.autentia.tutorial.drools.data;

public class Customer {

    public static final int DEFAULT_CUSTOMER = 0;
    public static final int SILVER_CUSTOMER = 1;
    public static final int GOLD_CUSTOMER = 2;

    private int status;

    private String name;

    public Customer(int status, String name) {
        super();
        this.status = status;
        this.name = name;
    }

    // getters y setters
}
```

Por último, la clase Product representa uno de los productos que componen el pedido. Tiene 3 atributos:

- id: el identificador del producto.
- description: nombre o descripción del producto.
- price: el precio del producto.

```
package com.autentia.tutorial.drools.data;

public class Product {

    private int id;

    private String description;

    private double price;

    public Product(int id, String description, double price) {
        super();
        this.id = id;
        this.description = description;
        this.price = price;
    }

    // getters y setters

}
```

A continuación escribimos nuestras reglas de negocio en un archivo Order.drl

```
package com.autentia.tutorial.drools

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

// Sumamos el importe total de los productos
rule "Initial rule"
    salience 20
    when
        order : Order ();
        totalPrice : Double() from accumulate (
            Product( productPrice : price) from order.getProducts,
            init (double total = 0;),
            action (total += productPrice;),
            result (new Double(total))
        );
    then
        order.setTotalPrice(totalPrice);
end

// Comprobamos si el cliente es SILVER, si es así aplicamos un 5% de descuento
rule "SILVER customer rule"
    salience 15
    when
        order : Order ();
        customer : Customer ( status == Customer.SILVER_CUSTOMER) from order.getCustomer()
```

```
    then
        order.setTotalPrice(order.getTotalPrice() * (1 - (5 / 100d) ) );
end

// Comprobamos si el cliente es GOLD, si es así aplicamos un 10% de descuento
rule "GOLD customer rule"
    salience 15
    when
        order : Order ();
        customer : Customer ( status == Customer.GOLD_CUSTOMER) from order.getCustomer();
    then
        order.setTotalPrice(order.getTotalPrice() * (1 - (10 / 100d) ) );
end

// Comprobamos si el pedido tiene 10 o más productos, si es así aplicamos un 15% de descue
// Esta regla solo será aplicada en Septiembre de 2012
rule "Number of products rule"
    salience 10
    date-effective "01-SEP-2012"
    date-expires "01-OCT-2012"
    when
        order : Order (products.size() >= 10);
    then
        order.setTotalPrice(order.getTotalPrice() * (1 - (15 / 100d) ) );
end
```

Observamos que tenemos 4 reglas: "Initial rule", "SILVER customer rule", "GOLD customer rule" y "Number of products rule". Ahora las explicaremos de una en una, pero antes debemos fijarnos en que todas ellas contienen un atributo **salience** con un número al lado. Ese número indicará la prioridad que tiene la regla sobre las demás a la hora de ejecutarse en caso de que se den las condiciones para que se ejecute más de una regla. A mayor salience, antes se ejecutará la regla.

Observemos además, que todas las reglas tienen una parte **when** que evalua los datos de la memoria de trabajo y otra parte **then** que ejecuta una acción si se cumplen las condiciones de la parte when.

6.1 La regla "Initial rule".

Si nos fijamos, esta regla tiene un salience igual a 20 lo que indica que, en caso de que se den las condiciones para que se ejecute esta regla al mismo tiempo que otras, ésta tendrá una prioridad de valor 20. ¿Y cuáles son las condiciones que se deben dar para que se ejecute esta regla?. Pues si nos fijamos en la parte "when" de la regla, con la sentencia **order : Order()** estamos diciendo que esta regla se ejecutará cuando, en la memoria de trabajo, se haya insertado un objeto Order. Este objeto será apuntado por la variable order.

Además, cuando se de el caso de que haya un objeto Order en la memoria de trabajo, lo que hace esta regla es crear una **variable totalPrice** que valdrá la suma de todos los precios de los productos que componen la orden.

Por último, el precio total de los productos es "setteado" en el objeto Order. Por tanto, lo que realmente hace esta regla es **sumar el importe de todos los productos y dejarlo almacenado en el objeto Order**. Si volvemos a mirar las reglas, observamos que tiene un salience mayor que el resto de reglas, por lo que será la primera que se ejecute en caso de conflicto con las demás. Tiene sentido ya que la función de esta regla es inicializar el precio total del pedido antes de que se empiecen a ejecutar los descuentos y promociones.

6.2 La regla "SILVER customer rule".

La siguiente regla que vemos en fichero es "SILVER customer rule" que nos valdrá para comprobar si un cliente es SILVER para proceder a aplicarle el descuento pertinente.

Para que esta regla se ejecute deben pasar dos cosas: que haya un objeto Order en la memoria de trabajo (como ocurría con la regla anterior) y que el cliente (Customer) que realiza el pedido sea SILVER, o lo que es lo mismo, **el atributo status de nuestro Customer debe ser igual a SILVER**.

Vemos que, en el caso de que se inserte en la memoria de trabajo un cliente tipo SILVER, se ejecutará antes la regla "Initial rule" ya que "SILVER customer rule" tiene un salience de 15 frente a un 20 de la anterior. Lo que es lo mismo, cuando se ejecute la regla "SILVER customer rule", ya se habrá sumado el importe total de todos los productos.

Por último, la acción que desencadena esta regla (parte then de la regla) es aplicar el descuento del 5% al total acumulado.

6.3 La regla "GOLD customer rule".

La siguiente regla no tiene ningún misterio si se ha entendido la anterior. Es exáctamente igual que que "SILVER customer rule" con la diferencia de que, para que se ejecute la regla, se debe dar la condición de que el cliente sea GOLD en vez de SILVER.

La acción que desencadenará será aplicar un 10% de descuento al total acumulado.

6.4 La regla "Number of products rule".

Esta regla tiene una diferencia con respecto a las demás y es que, si la fecha no es de Septiembre de 2012, no se ejecutará. Para ello hacemos uso de los atributos **date-effective y date-expires** (formato de fecha dd-MMM-yyyy) que establecerán el intervalo de tiempo en la que esta regla se puede ejecutar.

Suponiendo que nos encontremos en el periodo de vigencia de la regla, ésta se ejecutará si hay un objeto Order en la memoria de trabajo cuya lista de productos sea superior o igual a 10.

Si se dan las condiciones necesarias, la acción que desencadenará esta regla será aplicar un descuento del 15% sobre el total acumulado.

6.5 Probando el ejemplo.

Como dijimos anteriormente, una de las formas de utilizar el motor de reglas es hacerlo de forma embebida en nuestra aplicación. Es lo que vamos a hacer en nuestro ejemplo. Para ello necesitamos hacer dos cosas:

- Inicializar el motor de reglas con el fichero donde las hemos escrito (Order.drl).
- Crear la memoria de trabajo en la que insertaremos los datos y se ejecutarán las reglas.

Veamos cómo hacerlo y seguidamente explicaremos el código.

```
package com.autentia.tutorial.drools;

import java.util.Arrays;
import java.util.List;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderError;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.ResourceFactory;
import org.drools.runtime.StatefulKnowledgeSession;

import com.autentia.tutorial.drools.data.Customer;
import com.autentia.tutorial.drools.data.Order;
import com.autentia.tutorial.drools.data.Product;

public class OrderTest {

    public static final void main(String[] args) {
        final KnowledgeBase kbase = readKnowledgeBase();
        final StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

        final List<Order> orders = Arrays.asList(getOrderWithDefaultCustomer(), getOrderW
                getOrderWithGoldCustomer(), getOrderWithGoldCustomerAndTenProducts());
        for (Order order : orders) {
            ksession.insert(order);
        }
        ksession.fireAllRules();

        showResults(orders);
    }

    private static KnowledgeBase readKnowledgeBase() {
        final KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
        kbuilder.add(ResourceFactory.newClassPathResource("Order.drl"), ResourceType.DRL);
        if (kbuilder.hasErrors()) {
            for (KnowledgeBuilderError error : kbuilder.getErrors()) {
                System.err.println(error);
            }
            throw new IllegalArgumentException("Imposible crear knowledge con Order.drl");
        }
        final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
        return kbase;
    }

    private static Order getOrderWithDefaultCustomer() {
        final Order order = new Order(getDefaultCustomer());
        order.addProduct(getProduct1());
        return order;
    }

    private static Order getOrderWithSilverCustomer() {
        final Order order = new Order(getSilverCustomer());
        order.addProduct(getProduct1());
        return order;
    }

    private static Order getOrderWithGoldCustomer() {
        final Order order = new Order(getGoldCustomer());
        order.addProduct(getProduct1());
        return order;
    }

    private static Order getOrderWithGoldCustomerAndTenProducts() {
        final Order order = new Order(getSilverCustomer());
        for (int i = 0; i < 10; i++) {
            order.addProduct(getProduct1());
        }
        return order;
    }

    private static Customer getDefaultCustomer() {
        return new Customer(Customer.DEFAULT_CUSTOMER, "Cliente estandar");
    }

    private static Customer getSilverCustomer() {
        return new Customer(Customer.SILVER_CUSTOMER, "Cliente SILVER");
    }

    private static Customer getGoldCustomer() {
        return new Customer(Customer.GOLD_CUSTOMER, "Cliente GOLD");
    }

    private static Product getProduct1() {
        return new Product(1, "Producto 1", 100d);
    }

    private static void showResults(List<Order> orders) {
        for (Order order : orders) {
            System.out.println("Cliente " + order.getCustomer() + " productos: " + order.g
                    + " Precio total: " + order.getTotalPrice());
        }
    }
}
```

Si nos fijamos, existe un método **readKnowledgeBase**. Este método lo que hace es generar una factoría de sesiones (memorias de trabajo) en base a nuestro fichero de reglas Order.drl. El objeto resultante es un **KnowledgeBase**. Nótese que este **proceso es relativamente pesado**, por lo que se recomienda que en nuestra aplicación esto se realice **una única vez y se utilice la misma instancia de KnowledgeBase para todas las sesiones que creemos**. Para el que esté familiarizado con Hibernate sería lo equivalente a construir un SessionFactory, se crea una única vez al iniciar la aplicación y se usa una misma factoría para crear todas las sesiones.

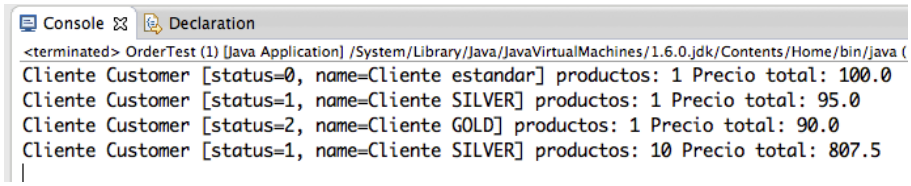
Una vez que ya tenemos nuestro KnowledgeBase, lo siguiente que hacemos es crear una sesión (que vendría a representar la memoria de trabajo). Lo haremos invocando al método **newStatefulKnowledgeSession** de

KnowledgeBase (crea una memoria de trabajo con estado). A continuación insertamos nuestros pedidos en la memoria de trabajo invocando al método **insert** de StatefulKnowledgeSession. Finalmente ejecutamos las reglas invocando a **fireAllRules**, que actuarán sobre los objetos que insertamos anteriormente.

Si nos fijamos hemos insertado 4 pedidos en nuestra memoria de trabajo:

- Un pedido realizado por un **usuario estandar** que consta de un producto de 100 euros. Según las reglas que hemos definido no se aplicará ningún descuento al pedido.
- Un pedido realizado por un **usuario SILVER** que consta de un producto de 100 euros. Según las reglas que hemos definido se aplicará un descuento de un **5%**.
- Un pedido realizado por un **usuario GOLD** que consta de un producto de 100 euros. Según las reglas que hemos definido se aplicará un descuento de un **10%**.
- Un pedido realizado por un **usuario SILVER** que consta de **10 productos** de 100 euros cada uno. Según las reglas que hemos definido se aplicará un descuento de un **5%** por ser usuario SILVER y otro de un **15%** por ser un pedido de 10 o más productos (suponemos que este pedido se procesa en Septiembre de 2012).

Lanzamos el ejemplo y el resultado es el esperado :-D.



Aquí dejo el código fuente del tutorial.

7. Referencias.

- Documentación de JBoss Drools.
- Código fuente del tutorial.
- Drools (excelente tutorial de Jorge Roldán).

8. Conclusiones.

En este tutorial hemos pretendido hacer una introducción a Drools y, en general, a los sistemas de gestión de reglas de negocio. Su uso está especialmente recomendado en los casos en los que queremos centralizar la lógica de negocio que generalmente tenemos distribuida por la empresa (aplicaciones, analistas, etc...). Cuando esa lógica es cambiante, es cuando este tipo de herramientas muestran toda su potencia.

Drools permite codificar la lógica de negocio en reglas, que suelen ser más fáciles de comprender que el código de la aplicación, sobre todo para el personal no técnico. Esta característica permite que sean los expertos en el negocio los que puedan gestionar las reglas.

Me gustaría que quedase claro que lo que hemos visto en este tutorial es sólo la punta del iceberg. Solo una pequeña parte de lo que se puede hacer con un BRMS y, en concreto, con Drools. En próximos tutoriales iremos descubriendo nuevas funcionalidades.

Dedicado a "el yayo".

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

Miguel Arlandy

marlandy@autentia.com

Twitter: @m_arlandy