Introducción a Drools con Spring Boot: toma de contacto

por Daniel Peña (https://www.paradigmadigital.com/author/dpena/)

Madrid, España 22 de mayo del 2019 Desarrollo

4 comentarios

No hay duda de que la parte principal y corazón de cualquier aplicación es **el bloque de acciones de negocio que realiza**, lo que comúnmente llamamos su **core**.

Con la premisa de proporcionar un ecosistema completo que permita definir, mantener, encapsular y ejecutar como una entidad única e independiente todas estas reglas de negocio nacen los **Business Rule Management System (BRMS)**.

Hoy en el blog veremos cómo encaja Drools en todo este entramado, otros elementos de la suite de Red Hat BRMS y jugaremos un poco con la definición de reglas, para degustar la sencillez y potencia de esta herramienta.

Aplicaciones como herramientas y no como soluciones finales

En aplicaciones tradicionales cambiar algún flujo de negocio generalmente implica un evolutivo que conlleva un cambio de código fuente, planificación, redespliegue, etc., esto acaba transformándose en un proceso tedioso, tardío y poco flexible.

La historia suele suceder así: un cliente nos pide construir una aplicación para dar solución a un problema o necesidad.

Esta petición siempre viene acompañada de unos requisitos de negocio: que se accesible por la web, que mande un correo cuando te registres, que guarde archivos, que haga tal o cual cosa. El cliente tiene claras 3 o 4 funcionalidades y con nuestra ayuda la termina de definir.

Si el día de mañana el negocio del cliente evoluciona y quiere cambiar algún flujo o funcionalidad, debe contratar otra vez el servicio y volver a comenzar la rueda de los evolutivos, con la demora y coste que esto conlleva.

Reflexionando sobre esta historia, **¿no es mejor orientar la construcción de la solución como una herramienta con requisitos, pero que su comportamiento core sea editable?**

Construye una única vez la aplicación con sus requisitos básicos: accesible por la web, escalable, con integraciones para mandar correos, guardar archivos, específicas que den solución a tal o

cual funcionalidad y ofrezcamos un fácil acceso al corazón del negocio para poder modificarlo y ajustarlo de manera sencilla si fuera necesario.

Consideraciones antes de entrar en materia

Cuando desarrollamos aplicaciones, habitualmente nuestro código describe "a mano" qué funciones y en qué orden se deben invocar para cumplir una determinada funcionalidad.

El concepto de un motor de reglas, "es un poco al revés". Las aplicaciones están compuestas por entidades. Estas entidades tienen distintos estados y estos pueden variar.

Los estados de estas entidades harán que el motor de reglas se comporte de una manera u otra, desencadenando acciones sobre esas entidades o cualquier otro tipo de funcionalidad.

El motor de reglas se ejecuta como una **«caja negra»** portable dentro de nuestra aplicación, de forma que el desacoplado de capas está garantizado.

Business Rule Management System (BRMS)

El uso de un BRMS nos proporciona muchas ventajas. Estas son algunas de ellas:

- Un **lugar centralizado donde editar las reglas de negocio** de manera rápida y sencilla de forma colaborativa.
- Nos **proporciona un lenguaje sencillo y comprensible** para la definición de las reglas.
- Al proporcionarnos un motor de ejecución propio, **añadir cambios a nuestra aplicación es algo sencillo** y que tiene un coste mínimo.

¡Vamos allá!

En este tutorial vamos a ver explorar Drools, que es la implementación de Red Hat para un modelo BRMS, pero existen otros productos en el mercado como:

- IBM Operational Decision Manager
- SAS Bussines Rules Manager

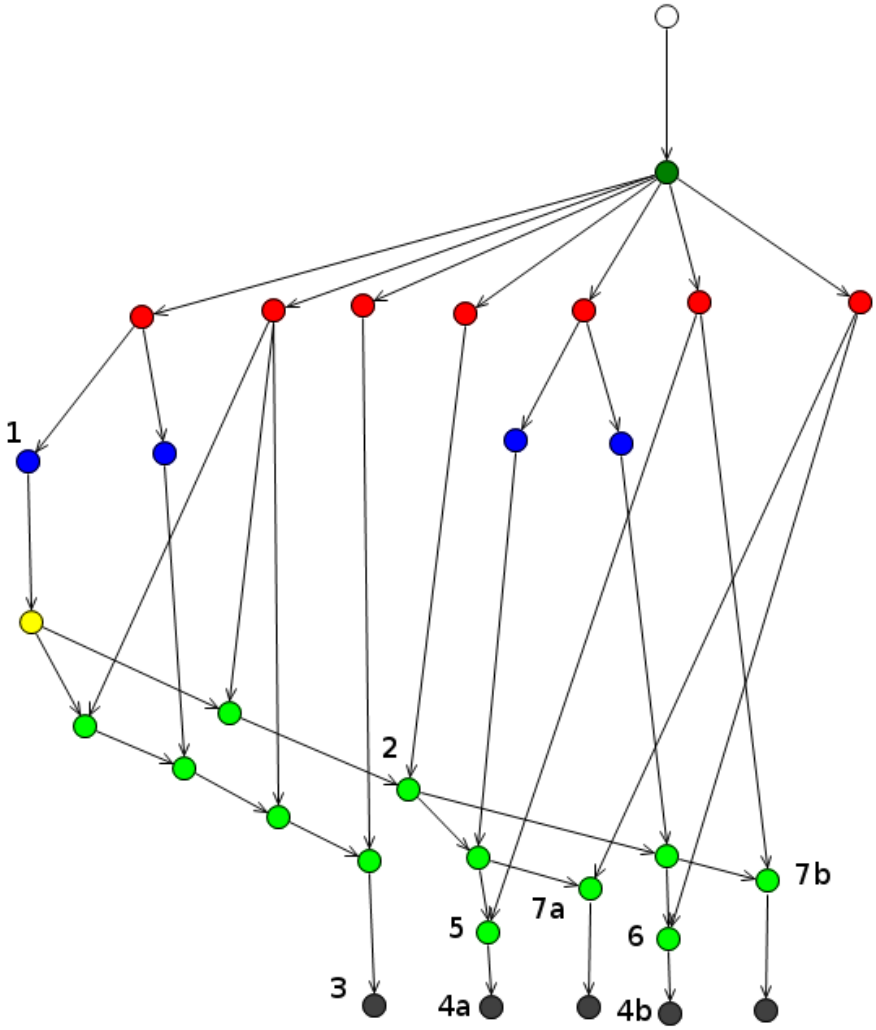
- Drools
- AGILOF Custom WorkFlow
- InRule

DROOLS (JBoss Rules)

Drools (https://www.drools.org/) es un Business Rule Engine implementado por Red Hat que **provee de herramientas para la definición de las reglas**. También se ocupa de su proceso de compilación, recreación del árbol de flujos/decisiones (basado en el algoritmo de inferencia Rete).

Rete (https://es.wikipedia.org/wiki/Algoritmo_Rete) **es un algoritmo de reconocimiento de patrones**, que en base a las reglas definidas y posibles acciones derivadas (inferidas) se genera un árbol de decisiones.

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.



Drools se caracteriza por **ofrecernos una forma sencilla de declarar las reglas**, ya sea **a través de ficheros .drl** (drools rule language (https://access.redhat.com/documentation/en-us/red_hat_process_automation_manager/7.2/html/designing_a_decision_service_using_drl_rules/drl-rules-con_drl-rules)) o directamente **a través de ficheros excel**.

Esta capacidad de definición a través de ficheros excel es muy valorable desde el punto de vista de negocio, ya que perfiles no técnicos pueden definir cómo se debe comportar la aplicación.

KIE (DROOLS6 + OPTAPLANNER + JBPM)

KIE Engine (knowledge is everything) también pertenece a la suite de Red Hat y engloba todas las herramientas para la ejecución del motor de reglas. En la ejecución de un motor de reglas encontramos 2 fases fundamentales:

1. Generación del árbol de decisiones, cuyo encargado es el Business Rule Engine (Drools).
2. El proceso de toma de decisiones al navegar por el árbol en tiempo de ejecución, implementada por el framework

OptaPlanner (Java).

Todo el proceso es gobernado y manejado por el motor JBPM (http://www.jbpm.org/), que es el encargado de ejecutar flujos de trabajo y reglas. Esta en Java y también es un producto de Red Hat .

Este motor BPM nos permite suscribirnos a los eventos y acciones que suceden dentro del él y nos proporcionan de un conjunto bastante amplio de APIs para manejarlo.

KIE Execution Server (Wildfly)

Aunque el propósito de este tutorial es ejecutar el motor de reglas dentro de una aplicación propia construida sobre Spring Boot, la suite de Red Hat BRMS tiene otro producto a modo servidor centralizado de motor de reglas.

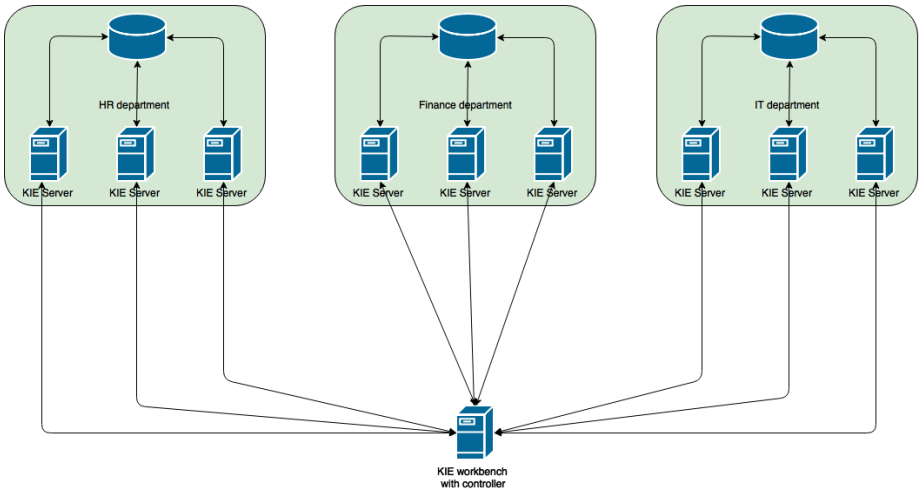
KIE Execution Server es una aplicación distribuida en formato .war, **clusterizable** que se despliega sobre el servidor J2EE de la casa WildFly (https://wildfly.org/).

Esta aplicación expone un motor de reglas accesible a través de un API Rest donde nuestro distintos servicios pueden hacer sus invocaciones. Puedes encontrar más información aquí (http://www.mastertheboss.com/jboss-jbpm/jbpm6/running-rules-on-wildfly-with-kie-server).

Business Central WorkBench (Drools workbench)

Otro de los tentáculos de la suite de Red Hat para los sistemas BRMS es el proyecto **Business Central Workbench.** Es una herramienta que nos permite, a través de un servidor web, **manejar de manera centralizada y colaborativa en la definición de las reglas, editarlas, versionarlas y servir configuración compartida a distintos servidores de ejecución**.

Aunando las herramientas actuales, podemos ver una foto general de cómo se puede manejar una infraestructura compleja, donde tenemos un WorkBench para unificar, definir nuestras reglas, y proporcionar de repositorio común a todos los KIE Execution Servers que den soporte a distintas áreas de nuestro negocio. La imagen es bastante ilustrativa:



Definiciones

En el ecosistema de un BRMS, encontramos varios conceptos y agentes que intervienen en su configuración y ejecución.

- **Facts**: son los argumentos que entran al motor de reglas. Básicamente son POJOs. Estos Facts/Pojo/Bean, podemos definirlos dentro de drools directamente para ser compiladas por el motor.
- **Rule**: indican cuándo se deben aplicar y qué se debe ejecutar. Tiene dos partes fundamentales:
  - **RHS**: **Right Hand Side**, donde se definen los criterios que la dispararán.
  - **LHS**: **Left Hand Side**, donde se definen las acciones que se ejecutarán.
- La definición de las reglas puede implementarse de dos maneras, mediante:
  - **Excel** (.xls, .xlsx). También llamadas **Decision Tables**. Nuestras reglas se pueden definir en este tipo de ficheros más amigables para perfiles no tan técnicos, que solo quieran preocuparse de definir las reglas de negocio abstrayéndose de todo lo que hay por debajo.
  - **Ficheros .drl** (Drools Rule File): ficheros de texto plano mucho más versátiles, donde se puede especificar a bajo nivel comportamientos que no seríamos capaces de definir en un amigable xls/xlsx.
- **WorkingMemory**: memoria de trabajo. Es el contexto de ejecución, en él se evalúan y ejecutan las reglas, es decir, el runtime.

- **KnowledgeSession**: la sesión se crea dentro de la Working Memory. Esta es la encargada de preparar la ejecución y montar el ecosistema de reglas asociadas a dicha ejecución.
- **KnowledgeBase**: repositorio donde encontrar las reglas y construir el engine, es decir, la KnowledgeSession. Ese repositorio puede nutrirse de diferentes fuentes donde leer la definición de las reglas: de un directorio local, un repositorio remoto centralizado y versionado de las reglas... Al fin y al cabo, cualquier cosa convertible a un array de bytes.

La forma en la cual se buscan, indexan e interpretan las definiciones de reglas, el estado de contexto a usar (stateless/stateful) y, en definitiva, la forma en la que el BPM compila y actúa es totalmente editable desde el API que nos proporciona jBPM y Drools.

Preparación del entorno

En este tutorial vamos a **integrar la ejecución del motor de reglas sobre un proyecto de Spring Boot**. Para ello nos basaremos en las libs de drools core para la definición de las reglas y el framework KIE para la ejecución de las mismas.

Podemos importar todas las dependencias desde los repositorios centrales de maven:

```
<!-- JBPM and Spring integration -->
<dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-core</artifactId>
    <version>7.18.0.Final</version>
</dependency>
<dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-spring</artifactId>
    <version>7.18.0.Final</version>
</dependency>
```

Preparando la ejecución para unos test aplicativos

Para que nos vayamos introduciendo en el manejo de Drools, vamos a hacer unos ejemplos muy sencillos donde aprenderemos a **definir con unas reglas sencillas**.

Este primer ejemplo es muy simple, al contexto de ejecución de Drools le vamos introducir un Fact que define el precio de un producto, este será analizado por las reglas y se dispararán una serie de acciones.

Lo primero generar la configuración

Para instanciar el motor del engine mediante Spring Boot basta con generar nuestra clase de configuración usando el archiconocido @Configuration y un método que inyecte en el contenedor un Bean de tipo KIEContainer.

Este KIEContainer encapsula todos los elementos de los que hablamos al principio (KnowledgeBase, configuración Working Memory, etc.) y nos proporcionará un API sencillo para interactuar con todos ellos.

En este caso lo usaremos para crear la KnowledgeSession cada vez que requiramos una ejecución.

Cada ejecución tiene su contexto, por lo tanto su runtime, que es configurable en modo stateless o stateful:

```
1  @Configuration
2  public class BPMConfigurations {
3    //lives on classpath --
   src/main/resources/rules/*.drl
4    private static final String[] drlFiles = {
   "rules/discountRules.drl" };
5    @Bean
6    public KieContainer kieContainer() {
7      KieServices kieServices =
   KieServices.Factory.get();
8      //Load Rules and Ecosystem Definitions
9      KieFileSystem kieFileSystem =
   kieServices.newKieFileSystem();
10     for (String ruleFile : drlFiles) {
11       kieFileSystem.write(ResourceFactory.newClassPathResource(ruleFile));
12     }
13     //Generate Modules and all internal Structures
14     KieBuilder kieBuilder =
   kieServices.newKieBuilder(kieFileSystem);
15     kieBuilder.buildAll();
16     KieModule kieModule = kieBuilder.getKieModule();
17     return
   kieServices.newKieContainer(kieModule.getReleaseId());
18   }
19 }
```

Como podemos ver, **KIE nos abstrae de la configuración de los Modules BPM** (y demás configuración subyacente).

Con este Bean en el contexto ya podemos empezar a trastear. Le especificamos que cargue un fichero de reglas que se encuentran en el classpath: *"src/main/resources/rules/discountRules.drl"* (el KIEContainer hace todo por nosotros).

Ahora generamos un POJO para actuar como Fact (ProductPrice.java) muy sencillo (he usado lombok framework (https://www.paradigmadigital.com/dev/proyecto-lombok-facilitame-la-vida/) para excluir código genérico "innecesario"):

```java
@Getter
@Setter
@NoArgsConstructor
@ToString
public class ProductPrice {
  private Integer basePrice;
  public ProductPrice(Integer basePrice) {
    this.basePrice=basePrice;
  }
}
```

Ahora una clase de Servicio muy sencilla, que nos provea del acceso a la ejecución:

```java
@Service
public class PriceCalculatorService {
  @Autowired
    private KieContainer kieContainer;
  public void executeRules(ProductPrice productPrice) {
      KieSession kieSession =
kieContainer.newKieSession();
        kieSession.insert(productPrice);
        kieSession.fireAllRules();
        kieSession.dispose();
    }
```

Y, por último, un punto de entrada para poder ejecutar nuestro código. Es este caso un JUnit test:

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class DroolsDemoApplicationTests {
  @Autowired
  private PriceCalculatorService
priceCalculatorService;
  @Test
  public void executeCalculations() {
    ProductPrice productPrice = new
ProductPrice(5);//Create the Fact
    priceCalculatorService.executeRules(productPrice);//Call
service and internal
                    //BlackBox rules engine
    System.out.println(productPrice);//final object
state after rules execution
  }
}
```

Vemos que:

- Instanciamos un precio de producto con basePrice = 5.
- Se lo pasamos al service y este ejecuta «la caja negra» del motor de reglas.
- Por último, vemos cómo queda nuestro ProductPrice después de que lo modifique el motor de reglas.

Esta configuración y el código puesto son más que suficientes para hacer nuestras pruebas y juguetear un poco.

Vamos a ver el contenido de nuestra definición de reglas del archivo **discountRules.drl**.

Esta primera versión es muy sencilla, al introducir el producto en el engine chequeará si el precio del producto es mayor que 2, y si es así, hará un print por consola.

discountRules.drl

```
1  package myAppRules;
2
3  import com.dppware.droolsDemo.bean.*;
4
5  dialect  "mvel"
6
7  rule "Adjust Product Price"
8      when
9        $p : ProductPrice(basePrice > 2 )
10     then
11       System.out.println("EJECUTANDO -Adjust Product
   Price- para el producto [" + $p + "]");
12 end
```

Podemos intuir que importa unos tipos de objeto y que hay una rule que tiene una condición y que si se cumple hace un system Out. Por partes:

- **package**: es una agrupación lógica de reglas. **No tiene que ver con paquetería física.** Debemos entenderlo más como un namespace, donde grupos de elementos tienen relación (globales, functions y demás, conceptos que veremos más adelante). Los nombres de las rules deben ser únicos dentro de un mismo package (namespace).
- **import**: importamos definiciones de clases que necesitará drools en la compilación de las reglas y su ejecución. Por defecto, indicar que Drools importa siempre el paquete java.lang.*, por lo que podemos usar todas las clases del paquete en nuestras definiciones de reglas.
- **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. Ofrece una sintaxis que casa con la nomenclatura java standar. Su uso es casi extendido exclusivamente a la seccion RHS.
    - Hay ya muchos DSL que se basan en esto, pero dejo aquí un ejemplo de equivalencia de código para que se vea:
        - java version:

```
$person.getAddresses().get("home").setStreetName("my street");
```

- mvel version:

```
$person.addresses["home"].streetName = "my street";
```

- Mvel también nos permite asignacion de variables (en el scope de una rule) de manera sencilla ($varName), así como la definición de nuevos tipos (classes) de manera sencilla a nivel de package (lo vemos más adelante).
- **«java»**-> podemos incluir nuestra sintaxis java dentro del .drl. Su única restricción es que solo se puede usar en el LHS (lefHandSide), es decir, en el then.

Ejecución

Si ejecutamos el test, estos son los pasos fundamentales:

- Entra el fact a evaluar con baseprice = 5 desde nuestro código Java.

```
ProductPrice productPrice = new ProductPrice(5);//Create the Fact
priceCalculatorService.executeRules(productPrice);//Call service and internal
                                                   //
BlackBox rules engine
```

- El motor de Drools comprueba el when y, como en este caso se cumple la condición, pues asigna a la variable local $p el objeto ProductPrice. Mvel nos proporciona el acceso al getBasePrice sin necesidad de declararlo.

```
$p : ProductPrice(basePrice > 2 )  //
Object({conditions})
```

Las condiciones disponibles (Fact({conditions}) las vamos a ver más adelante.

- Como se cumple, ejecuta el then haciendo el print de la variable seteada $p.

System.out.println(«EJECUTANDO -Adjust Product Price- para el producto [» + $p + «]»);

Conditions (revisando condiciones en LHS – Left Hand Side)

Como hemos visto en el paso 2 anterior, se cumple la condición de que basePrice > 2 (ya que era 5 cuando entró al contexto de ejecución).

Para el ejemplo anterior pondríamos condiciones más complejas:

```
( > , < , >=, =<, || , &amp;&amp; , == , % , ^,
contains, not contains, memberof, not memberof, matches
(regExp), not matches (regExp), starswith , etc...)
```

Ejemplos:

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

En vez de anidarlos también podríamos usar la cláusula por defecto **and** y convertir la condición anterior en:

```
when
    $p : ProductPrice((basePrice / 5) == 1))
    $p : ProductPrice((basePrice % 5) == 0 ) //se deben
cumplir todas
  then
```

Existe una evaluación "en modo manual" con la palabra reservada "eval" que nos 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 el segundo eval, dejo ver que Drools, como compilador, nos permite referenciar a métodos estáticos de nuestro código o definir funciones dentro del mismo fichero .drl que como ya vimos estarán disponibles en todo el package (namespace).

Modify{...: aplicando órdenes en el then RHS

Ya hemos visto las conditions, pero vamos a ver las órdenes. Hemos visto una sencilla ejecución con el system out, referenciando a la variable local de la rule $p:

```
then
      System.out.println("EJECUTANDO -Adjust Product
Price- para el producto [" + $p + "]");
```

Si queremos aplicar una modificación, vamos a usar el bloque modify. Vamos a bajarle el basePrice un punto y nos quedará así:

```
import com.dppware.droolsDemo.bean.*;
dialect  "mvel"
rule "Adjust Product Price"
  when
      $p : ProductPrice(basePrice > 2 )
    then
      modify($p){
        setBasePrice($p.basePrice - 5);
      }
      System.out.println("el precio ajustado es " +
$p.basePrice);
```

```
11 end
```

Sencillo, ¿verdad? Abrimos un bloque modify con el scope de la variable definida dentro de la función (ya que se cumplió el LHS- *Left Hand Side*).

En el contexto encontramos la función setBasePrice del objeto ProductPrice. Si modificamos el código y ejecutamos el Test, veremos este output: el precio ajustado es 0.

Rules Attributes

El comportamiento en runtime de una rule se puede restringir. Para ello, debemos usar los **rule attributes** que nos ofrece drools. Estos atributos se definen justo debajo al empezar la rule:

```
1  rule 'rulename'
2    //rules Atributtes (availables: no-loop, salience,
   ruleflow-group, lock-on-active, agenda-group,
3    // activation-group, auto-focus, dialect, date-
   effective, date-expires, duratio, timer, calendars
4    when:
5      ...
6    then:
7      ...
8  end
```

Vamos a comentar dos de ellas (las que me parecen más genéricas y obligatorias de conocer), si quieres profundizar tienes la referencia oficial aquí. (https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch05.html#d0e3761)

No-loop

Imaginemos que metemos un Fact (ProductPrice.java) que su basePrice = 5. Cuando se cumple la condición solo le restamos 1 a su basePrice.

```
1  rule "Adjust Product Price"
2    when
3        $p : ProductPrice(basePrice > 2 )
4    then
5        modify($p){
6          setBasePrice($p.basePrice -1);
7        }
8        System.out.println("el precio ajustado es " +
   $p.basePrice);
9  end
```

Drools, por defecto, volverá a evaluar la LHS y verá que sigue cumpliéndose así que **la regla será disparada varias veces, hasta que no se cumpla la condición.**

Y el Ouput:

- El precio ajustado es 4
- El precio ajustado es 3
- El precio ajustado es 2

Esto puede ser un quebradero de cabeza en la ejecución, porque si la ejecución es de tipo void y no se modifica el Fact, podríamos caer en un bucle infinito.

Para asegurar que la regla (si se cumple) solo se ejecute una vez, usamos el atributo **no-loop**:

```
1  rule "Adjust Product Price"
2    no-loop
3      when
4        $p : ProductPrice(basePrice > 2 )
5      then
6        modify($p){
7          setBasePrice($p.basePrice -1);
8        }
9        System.out.println("el precio ajustado es " +
   $p.basePrice);
10 end
```

Y ahora el Output:

- El precio ajustado es 4.
- Solo se ha ejecutado 1 vez ya que solo se ha evaluado 1 vez.

Salience

Por defecto, Drools ordena las reglas de ejecución en el orden que se las va encontrando al parsear los ficheros de reglas (.drl). De modo que en tiempo de ejecución se ejecutarán las reglas que su RHS cumpla y por el orden en el que se almacenaron en el motor.

Imagina esta definición de drl en la que hemos introducido dos reglas (ya añadimos el no-loop también):

```
1  rule "Adjust Product Price"
2    no-loop
3      when
4        $p : ProductPrice(basePrice > 2 )
5      then
6        System.out.println("EJECUTANDO -Adjust Product
   Price-");
7  end
8  rule "Sending Notification"
9    no-loop
10     when
11       $p : ProductPrice(basePrice > 2 )
12     then
13       System.out.println("EJECUTANDO -Sending
   Notification-");
14 end
```

Y su Output:

- EJECUTANDO -Adjust Product Price-
- EJECUTANDO -Sending Notification-

Salience (prominencia) **es un sistema de pesos que nos permite indicar prioridades sobre las ejecuciones de las reglas en caso de coincidencia**. Si añadimos el atributo salience a las reglas vemos como podemos especificar el orden:

```
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
```

Y su Output:

- EJECUTANDO -Sending Notification-
- EJECUTANDO -Adjust Product Price-

Ahora se han ejecutado ordenadamente en función del peso (valor) de nuestra rule dentro del engine. Los valores de salience pueden ser negativos si queremos que nuestra regla tenga prioridad mínima y ser lanzada en última instancia.

Usando el compilador

Podemos importar funciones, crear nuevos tipos, funciones dentro del drl, etc...

functions – > Importando métodos de utilidades:

Pongamos que tenemos una librería de utilidades, parseos o que realiza cualquier otro tipo de funcionalidad atómica y que nos vendría fenomenal usarlo dentro de nuestras reglas.

Desde Drools podemos importar ese método para usarlo dentro de las rules (hay que tener en cuenta que será accesible dentro de todo su namespace/package).

Pues es tan fácil como definir el método con acceso static en nuestra clase java y luego importarlo en nuestro fichero .drl:

La clase Java:

```
public class Utils {
  public static void prettyTraces(Object message) {
    System.out.println("PrettyTraces ->
***"+message+"***");
  }
}
```

Para usarla en nuestra rule basta con importarlo de manera declarativa completa.

```
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
```

Output: PrettyTraces -> ***el precio ajustado es 0***

También puedes observar que he metido // para los comentarios.

functions – > Creandolas en drl:

Definir funciones dentro del .drl es muy sencillo, usamos la palabra reservada **function**. Vamos a incrementar el código metiendo la definición también:

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

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

dialect  "mvel"

//functions inline definition
function Integer calculateIncrement(Integer value, int
quantity) {
    return value + quantity;
}



rule "Adjust Product Price"
  when
      $p : ProductPrice(basePrice > 2 )
    then
      modify($p){
        setBasePrice($p.basePrice - 5);
      }
```

```
22        prettyTraces("el precio ajustado es " +
$p.basePrice);
23        prettyTraces(calculateIncrement($p.basePrice,
3));
24  end
```

Output:

- PrettyTraces -> ***el precio ajustado es 0***
- PrettyTraces -> ***3***

Vemos cómo hacemos la llamada al método que acabamos de crear y además al prettyTraces que habíamos añadido anteriormente.

Nuevos Tipos (TYPE – class)

Es posible definir nuestras nuevas clases dentro del ecosistema de manera declarativa y sin compilacion previa, ya que el compilador leerá la definición e introducirá en el classloader 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.

El compilador creará dos constructores por defecto (uno vacío y otro con todos los campos como argumentos).Si queremos customizar el constructor para usar solo determinados argumentos, debemos usar la anotación @key. Puedes ver aquí (https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch05.html#d0e3418) más ejemplos en la documentación oficial.

```
1  declare Product
2     code : int
3     name : String
4     description : String
5  ends
```

Y para su instanciación se puede hacer de esta manera usando sintaxis java (recuerda que esta sintaxis solo es aceptada en RHS):

```
1  import com.dppware.droolsDemo.bean.*;
2
3  //Imported specified functions
4  import function
   com.dppware.droolsDemo.utils.Utils.prettyTraces;
```

```
5
6  dialect  "mvel"
7
8  //functions inline definition
9  function Integer calculateIncrement(Integer value, int
   quantity) {
10     return value + quantity;
11 }
12
13 //New Types definition
14 declare Product
15    code : int
16    name : String
17    description : String
18 end
19
20 rule "Adjust Product Price"
21   when
22       $p : ProductPrice(basePrice > 2 )
23     then
24       modify($p){
25         setBasePrice($p.basePrice - 5);
26       }
27       prettyTraces("el precio ajustado es " +
   $p.basePrice);
28       prettyTraces(calculateIncrement($p.basePrice,
   3));
29       //Instanciacion y print del object
30       Product pro = new Product();
31       pro.setCode(3321);
32       pro.setName("Leche");
33       pro.setDescription("Rica en Calcio");
34       prettyTraces(pro);
35
36 end
```

Y el Output al ejecutarlo es:

- PrettyTraces -> ***el precio ajustado es 0***
- PrettyTraces -> ***3***
- PrettyTraces -> ***Product(code=3321, name=Leche, description=Rica en Calcio)***

Como podemos observar, nos está quedando el código bastante feo en este archivo .drl, ya que se nos empieza a ir de las manos.

Empieza a tener sentido el concepto de «package». Organicemos ficheros .drl (uno con definiciones de tipos, otro con funciones y otro con las rules). Todos deben definir el mismo package (namespace) en su cabecera y así seguirán disponibles de manera «global» dentro del mismo namespace.

En la siguiente sección veremos cómo incluir varios ficheros, pero antes vamos a ver los «Global» que va muy relacionado con lo que acabamos de ver.

Importar objects desde el contexto java (globals)

Hemos visto ya que podemos:

- Definir nuestras funciones (function).
- Definir nuestros tipos (classes).
- Importar tipos para usarlos en nuestros .drl (import pck.subpck1.subpck2.className).

Y ahora vamos a ver **cómo meter objetos ya instanciados y disponibles en la máquina virtual** al contexto de ejecución de Drools.

Puede ser muy útil inyectar un bean de servicio de nuestro contexto de Spring para que sea utilizado en las LHS para realizar operaciones complejas (piensa en insertar un DAO o cualquier tipo de Servicio).

A este tipo de inyecciones se las denomina **globals**. Para estas sí que necesitamos editar nuestro test Java, ya que se lo pasamos como argumento en el proceso de construcción de la ejecución.

Definimos un @Service Java y le añadimos un método muy básico que simule que publica en un topic (es solo para que te hagas una idea):

```
1  @Service
2  public class PushSubService {
3    //@Autowired
4    //private KafkaTemplate<String, String>
   kafkaTemplate;
5    public void publishNewProductCreated(Object o) throws
   JsonProcessingException {
6      String rawJSON = new
   ObjectMapper().writeValueAsString(o);
7      //kafkaTemplate.send("newProduct", rawJSON); ...or
   whatelse
8      System.out.println("Publishing newProduct Topic ,
   content ["+rawJSON+"]");
9    }
10 }
```

Ahora nos interesa tener este objeto dentro del contexto de ejecución de las rules de drools, así que lo inyectamos como global cuando solicitamos la ejecución.

La clase *PriceCalculatorService.java* (la del principio, ¡recuerda!) llevamos un buen rato sin tocarla y, en mi opinión, esto es lo bueno de drools, que solo montamos los «hierros» en java y lo delegamos todo en el motor de reglas que debería actuar como una caja negra para nosotros.

Modificamos el método para añadir un «globals» al contexto de ejecución. Para ello usamos el API que nos proporciona la KieSession:

```
@Autowired
private PushSubService pushSubService;
public void executeRules(ProductPrice productPrice) {
      KieSession kieSession =
kieContainer.newKieSession();
      kieSession.setGlobal("publishTool",
pushSubService);//adding globals
      kieSession.insert(productPrice);
        kieSession.fireAllRules();
        kieSession.dispose();
}
```

He inyectado el servicio a través del global name «publishTool» y lo referencio dentro del .drl en la parte de definiciones que estábamos usando para las functions y demás tipos que vimos anteriormente.

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

En la RHS (then) lo tenemos disponible:

```
publishTool.publishNewProductCreated(pro);
```

El archivo .drl ahora tiene este contenido:

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

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

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

dialect  "mvel"

//functions inline definition
function Integer calculateIncrement(Integer value, int
quantity) {
    return value + quantity;
}

//New Types definition
declare Product
   code : int
   name : String
   description : String
end

rule "Adjust Product Price"
  when
      $p : ProductPrice(basePrice > 2 )
    then
      modify($p){
        setBasePrice($p.basePrice - 5);
      }
      prettyTraces("el precio ajustado es " +
$p.basePrice);
      prettyTraces(calculateIncrement($p.basePrice,
3));
      //Instanciacion y print del object
      Product pro = new Product();
      pro.setCode(3321);
      pro.setName("Leche");
      pro.setDescription("Rica en Calcio");
      prettyTraces(pro);
      publishTool.publishNewProductCreated(pro);

```

```
40 end
```

Output:

- PrettyTraces -> ***el precio ajustado es 0***
- PrettyTraces -> ***3***
- PrettyTraces -> ***Product(code=3321, name=Leche, description=Rica en Calcio)***
- Publishing newProduct Topic , content [{«code»:3321,»name»:»Leche»,»description»:»Rica en Calcio»}]

Conclusiones

Hasta aquí hemos visto que Drools es un motor de reglas y nos hemos empezado a utilizarlo un poquito. Intentar abarcar Drools en un tutorial es una osadía, te recomiendo seguir los links de la documentación oficial que he ido dejando a lo largo del documento.

En la segunda parte del tutorial vamos a ver cómo organizar el código y algunos apuntes más del API que proporciona KIE.

Referencias

- BRMS
- Drools
- KIE
- OptaPlanner
- Rete
- JBPM
- DRL Language

Daniel Peña

Curioso por naturaleza. En mi opinión, escuchar, observar y aprender de lo que nos rodea, son las únicas herramientas con las que nacemos y las únicas que necesitaremos en la vida. A veces el conocimiento nos hace mas felices y a veces no. De momento intento abarcar todos los palos que pueda (informático, electrónico, músico, cocinero, mago, multideportista, hijo, padre, marido, trabajador multidisciplinar y un largo etcétera que seguro está por llegar...).

Ver toda la actividad de Daniel Peña

4 comentarios

Leandro 27 mayo, 2019 a las 19:18

Muy buén artículo. No me quedo claro si todo lo explicado aquí es para la versión gratuita. Gracias

Responder

Daniel Peña Perez 2 junio, 2019 a las 18:33

Hola Leandro, muchas gracias! (la verdad que queda una segunda parte del post que saldrá en breve donde explica un poco mas como versionar, distribuir las reglas, reload de reglas en caliente y demas..)
Si, todo lo que se usa en los ejemplos de este post solo tiene dependencia con el repo opensource de KIEGroup drools-core.
Te recomiendo visitar el Repositorio de Kiegroup https://github.com/kiegrou (https://github.com/kiegrou) y allí encontraras el producto JBPM, que es el workbench que te permite gestionar una infraestructura global de BRMS. Este producto tiene interfaces visuales para la gestión y edición de reglas en modo colaborativo, manejo de nodos ejecutores (KIE Execution Server), edicion de workflows, etc.. y ese fijo que es de pago.
Si quieres usarlo de manera rápida, puedes usar las imagenes

que tienen en dockerhub
https://github.com/kiegroup/kie-docker-ci-images (https://github.com/kiegroup/kie-docker-ci-images) y ver un poco lo que te ofrece.
Saludos!

Responder

Javier

4 junio, 2019 a las 11:02

Buenas,
muy buen post, nosotros tenemos un proyecto antiguo con Drools, procesando más de 35 millones de operaciones al año, y nos viene muy bien ver este tipo de posts.
Saludos,

Responder

Pablo

8 julio, 2019 a las 08:23

Excelente artículo, se resume muy bien lo que tardas en aprender un tiempo por uno mismo.

¿Sabes cuando vas a publicar la segunda parte del post? Me parece muy interesante el versionado de reglas, carga en caliente y demás.

Un saludo

Responder

ESCRIBE UN COMENTARIO

Nombre *

Mail (no será publicado) *

Página web

Comentario:

Enviar comentario

En Twitter

@paradigmate

Arranca en Paradigma la semana de la Diversidad para reivindicar la libertad, la inclusión, los derechos humanos..... https://t.co/oVrOZZSNkM (https://t.co/oVrOZZSNkM)

En nuestro blog

Las mejores buenas prácticas para tener en cuenta en nuestro día a día

¿Cómo lograr una buena calidad de código y cobertura de test? ¿Qué podemos hacer para no caer en la monotonía? ¿Cómo cambiar la rutina por buenas prác...

{paradigma

91 352 59 42

El responsable de los datos es Paradigma Digital SL
Vía de las Dos Castillas, 33 - Ática 2, 28224
Pozuelo de Alarcón (Madrid)

contacto

aviso legal y política de privacidad

cookies

newsletter

Your e-mail

Suscribirme

Acepto Los Tratamientos De Datos Indicados En La Política De Privacidad *

Utilizaremos la información que nos facilitas para suscribirte a nuestro envío periódico de newsletters así como mandarte otras publicaciones de Paradigma que puedan ser de tu interés. Nuestra legitimación es tu consentimiento. Tus datos no se cederán a terceros, y puedes modificarlos, darte de baja o eliminarlos cuando quieras. Aquí encontrarás toda la información sobre nuestra política de privacidad. (https://www.paradigmadigital.com/legal/)