Spring Expression Language

Desde la versión 3.0, Spring ofrece un lenguaje dinámico llamado *Spring Expression Language* (Lenguaje de expresiones Spring, a partir de ahora SpEL). SpEL permite la consulta y manipulación de objetos en tiempo real (de manera similar a Expression Language en JSP/JSF) además de otras interesantes características como la invocación de metodos. SpringSource Tool Suite (una variación de Eclipse altamente customizada para trabajar con Spring) provee soporte para SpEL, permitiendo autocompletado de expresiones además de otras caracteristicas realmente útiles para trabajar con el lenguaje. SpEL puede ser utilizado:

* De forma nativa en una aplicación Spring (insertado en archivos XML o anotaciones)
* A través de un parser, creando explicitamente toda la infraestructura necesaria para interpretarlo

Requisitos

Si usas Maven en tus proyectos, la única dependencia necesaria para usar empezar a usar SpEL es:

<dependency>

<groupid>org.springframework</groupid>

<artifactid>spring-context</artifactid>

<version>3.0.3.RELEASE</version>

</dependency>

Si deseas gestionar tus librerías sin Maven, además de todos los JAR's necesarios para levantar una aplicación Spring 3.0 (*org.springframework.context-3.0.X.RELEASE.jar*,*org.springframework.beans-3.0.X.RELEASE.jar*, ...) necesitas incluir en tu classpath el siguiente JAR que encontrarás en el directorio *dist* de tu distribución de Spring 3:

org.springframework.expression-3.0.X.RELEASE.jar

Sintaxis

La sintaxis de SpEL toma la siguiente forma cuando es usada de forma nativa:

#{EXPRESIÓN}

Cuando la expresión es usada por un parser, debemos omitir el caracter de almohadilla y la pareja de llaves:

EXPRESIÓN

A lo largo de este artículo, al introducir cada tipo de expresión se utilizara la primera versión, de manera que se enfatice su naturaleza y se diferencie más facilmente del resto del contenido. Solo cuando una expresión se utilice a través de un parser se utilizará la versión correspondiente, sin almohadilla ni llaves.

Construyendo expresiones (I)

Comencemos viendo los tipos mas básicos de expresión para realizar una toma de contacto con el lenguaje, de manera que en las dos próximas secciones (SPEL DE FORMA NATIVA y SPEL A TRAVES DE UN PARSER) entendamos el 100% del código y no nos perdamos. El primer tipo de expresión que vamos a ver, y la más simple, es la de tipo literal:

#{12.7}

La expresión anterior es un literal del número flotante 12.7. Otro tipo de literal frecuentemente utilizado es la cadena de texto:

#{'Hola SpEL'}

Como puedes ver, las expresiones literales de tipo cadena de texto son construidas dentro una pareja de comillas simples. Ambas expresiones literales devuelven 'literalmente' su valor (de ahí su nombre).

El siguiente tipo de expresión es la de tipo booleano:

#{7 > 9}

El significado de la expresión anterior es totalmente obvio, y el resultado devuelto al evaluar la expresión sería false.

La siguiente de la lista es la llamada *expresión estándar*, la cual nos permite acceder a las propiedades de un bean o javabean mediante notación de puntos:

#{usuario.nombre}

La expresión anterior devolverá el valor de la propiedad nombre del bean llamado usuario que, evidentemente, deberá estar registrado dentro del contexto de la aplicación Spring actual. Si lo que queremos es evaluar un bean en si mismo, nada tan fácil como usar su nombre como expresión:

#{usuario}

Antes de continuar viendo más tipos de expresiones, veamos cómo y dónde pueden ser usadas. Esto, además de completar el binomio 'sintaxis-uso' te permitirá experimentar con las expresiones mientras continuas leyendo el artículo.

SpEL de forma nativa

Como se indicó al inicio de este artículo, SpEL puede ser usado de forma nativa en una aplicación Spring 3.0 mediante anotaciones o en archivos de configuración XML. Veamos algunos ejemplos:

class Login {

@Value("#{usuario.nombre}")

private String nombreUsuario;

// ...

}

La anotación @Value puede ser aplicada tanto en variables como métodos setter, así como en parámetros de un método o constructor. El valor devuelto por la expresión será el valor por defecto del campo donde se aplique, en el caso del ejemplo anterior la variable nombreUsuario. Para que Spring pueda detectar la presencia de la anotación @Value en nuestras clases debemos activar la configuración mediante anotaciones en el fichero de configuración de Spring:

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context" xsi:schemalocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config>

</context:annotation-config></beans>

Para usar la misma expresión en un archivo de configuración de Spring 3.0:

<bean class="Login" id="login">

<property name="nombreUsuario" value="#{usuario.nombre}">

</property></bean>

El resultado es el mismo que al utilizar la anotación @Value.

SpEL a través de un parser

SpEL puede ser usado fuera de un archivo XML o anotación @Value mediante un parser. Este parser requiere el ensamblaje de cierta infraestructura:

ExpressionParser parser = new SpelExpressionParser();

Expression expresion = parser.parseExpression("'Hola SpEL'");

String nombreUsuario = expresion.getValue(String.class);

El código anterior es bastante simple. Primero crea un parser de la clase SpelExpressionParser. A continuación crea una expresión llamando al métodoparseExpression(EXPRESION) del recien creado parser. Por último, obtenemos el resultado de la evaluación de la expresión a traves del objeto expresion. Cuando la expresión hace referencia a un objeto, debemos crear un contexto de ejecución para dicho objeto:

Usuario usuario = new Usuario("David Marco");

EvaluationContext contexto = new StandardEvaluationContext(usuario);

ExpressionParser parser = new SpelExpressionParser();

Expression expresion = parser.parseExpression("nombre");

String nombreUsuario = expresion.getValue(contexto, String.class);

En el ejemplo anterior hemos creado un objeto usuario así como un contexto de evaluación para dicho objeto. En el momento de crear la expresión, solo indicamos la propiedad que queremos evaluar del objeto que hemos pasado al contexto (en lugar del binomio 'objeto.propiedad' que se utilizaría con la anotación @Value o en un archivo de configuración XML). Finalmente, pasamos el contexto de evaluación a expresion.getValue() de manera que la expresión (la propiedad nombre) es aplicada contra el contexto (el objeto usuario) resultando en la evaluación de usuario.nombre. En caso de que el contexto de evaluación tienda a cambiar entre ejecuciones es preferible sustituir el contexto de evaluación que hemos creado explicitamente por el objeto que subyace a dicho contexto, y Spring creará por nosotros un nuevo contexto de evaluación en cada llamada agetValue():

List usuarios = new ArrayList();

// Inicializar la lista de usuarios

ExpresionParser parser = new SpelExpressionParser();

Expression expresion = parser.parseExpression("nombre");

for(Usuario usuario: usuarios) {

String nombreUsuario = expresion.getValue(usuario, String.class);

}

Construyendo expresiones (II)

Ahora que ya hemos visto como y donde podemos escribir expresiones, sigamos viendo más tipos. Empecemos con las expresiones de clase:

#{T(java.lang.Math)}

El caracter T indica que queremos actuar sobre el tipo de una clase (en nuestro caso java.lang.Math), en lugar de sobre una instancia. Una vez obtenido el tipo, podemos asignarlo a una variable mediante @Value, XML o con el parser. También podemos actuar sobre el tipo devuelto por la expresión llamando a cualquiera de sus métodos estáticos dentro de la expresión:

#{T(java.lang.Math).random()}

SpEL nos permite definir expresiones para gestionar colecciones. Para acceder a un array o a un objeto de tipo List:

#{empresa.empleados[1]}

El acceso a listas y arrays se realiza mediante su índice, puesto que son colecciones indizadas. El acceso a una colección de tipo Map se realiza, como ya habras imaginado, mediante sus claves:

#{empresa.telefonosDepartamentos['Ventas']}

En ambos casos, se utiliza la notación de corchetes ya sea para obtener el índice en array y listas como la clave en mapas.

El siguiente tipo de expresión es la invocación de un método. Lo vimos en acción anteriormente al invocar random() en una expresión de clase, pero por su importancia merece un nuevo ejemplo:

#{'Hola SpEL'.toUpperCase()}

La expresión anterior es muy interesante, consistiendo en la invocación de un método sobre una expresión literal. Por supuesto, dicha invocación de método puede realizarse sobre cualquier expresión, clase, u objeto.

SpEL nos permite invocar el constructor de una clase con el siguiente tipo de expresión:

#{new Date()}

El siguiente tipo de expresión de nuestra lista es la expresión con operador ternario, que funciona exactamente igual de como lo haría en Java:

#{usuario.edad > 18 ? true : false}

Recuerda que cuando evaluamos una expresión sobre un objeto, y utilizamos el parser, omitimos el nombre del objeto puesto que dicho objeto ha sido o será pasado como contexto de evaluación:

Usuario usuario = new Usuario("David Marco", 30");

ExpressionParser parser = new SpelExpressionParser();

Expression expresion = parser.parseExpression("edad > 18 ? true : false");

Boolean mayorDeEdad = expresion.getValue(usuario, Boolean.class);

System.out.println("Es mayor de edad? " + mayorEdad);

También podemos hacer referencia al contexto de evaluación mediante una expresión de variable:

Expression expresion = parser.parseExpression("#this.edad > 18 ? true : false");

En la expresión anterior, #this hace referencia al objeto que conforma el contexto actual de evaluación, en nuestro caso usuario.

Los dos últimos tipos de expresión que vamos a mostrar en este artículo estan relacionadas con la manipulación de colecciones. La primera es *proyección de colección*:

#{usuarios.![nombre]}

En la expresión anterior, usuarios es una coleccion que contiene objetos de tipo Usuario. Una vez evaluada la expresión, el resultado devuelto es una lista que contiene los nombres (mediante la propiedad nombre) de todos los usuarios de la primera lista. El segundo tipo de expresión para manipular colecciones es *selección de colección*:

#{usuarios.?[nombre.startsWith('D')]}

Selección de colección permite, como muy bien expresan tanto su nombre como su sintaxis, filtrar los objetos de una colección (usuarios y devolver una nueva colección que contiene solo aquellos objetos que cumplian las condiciones de filtrado.

Resumen

Como puedes ver, Spring Expression Language provee un lenguaje rico y dinámico que te permitirá configurar los beans de tu aplicación Spring 3.0 de manera aún más flexible que en versiones anteriores, además de proveer un lenguaje de scripting que puedes usar dentro de tus clases a traves del parser.