



Curso Spring Boot



Temario

- Expresiones regulares (SpEL)
- Programación Orientada a Aspectos (AOP)
 - ¿Qué es AOP?
 - Casos de uso de utilización de AOP
 - AspectJ

Expresiones Regulares (SpEL)

- ¿Qué es SpEL?
 - Spring Expression Language (SpEL para abreviar) es un poderoso lenguaje de expresión que permite consultar y manipular un grafo de objetos en tiempo de ejecución. La sintaxis del lenguaje es similar a Unified EL pero ofrece características adicionales, sobre todo la invocación de métodos y la funcionalidad básica de creación de plantillas de String.
- Configuración basada en XML

```
<bean id="generadorNumero" class="org.cursospringboot.utils.GeneradorNumero">  
  <property name="numeroAleatorio" value="#{ T(java.lang.Math).random() * 100.0 }"/>  
</bean>
```

- Configuración basada en anotaciones

```
public class Informe implements GeneradorInformes {  
    @Value("#{ systemProperties['user.region'] }")  
    private String region;  
  
    @Override  
    public String getInforme() {  
        return "Informe Generado para la región " + region;  
    }  
}
```



Expresiones Regulares (SpEL)

- SpEl soporta la siguiente funcionalidad:
 - Expresiones literales
 - Boolean and operadores relacionales
 - Regular expressions
 - Expresiones de Clase
 - Acceso a propiedades, arrays, lists, maps
 - Invocación de métodos
 - Operadores relacionales
 - Asignación de variables
 - Invocación a constructores
 - Referencias a Beans
 - Construcción de Arrays
 - Listas Inline
 - Operador ternario
 - Variables
 - Funciones definidas por el usuario
 - Proyección de Colecciones
 - Selección de proyecciones
 - Expresiones de plantilla

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = (Expression) parser.parseExpression("'Hola mundo'.concat('!)");
String message = (String) exp.getValue();
```

```
String fraseRandom = parser.parseExpression("El número aleatorio es #{T(java.lang.Math).random()}",
new TemplateParserContext()).getValue(String.class);
```



Anotaciones

- ¿Qué es?
 - Las anotaciones de Java proporcionan cierta información sobre el código (Metadatos), pero no afectan directamente el contenido del código que anota.
 - Una anotación puede, por ejemplo, ser procesada por un generador de código fuente, por el compilador o por una herramienta de despliegue.
- Anotaciones de Spring
 - @RestController, @Service, @Repository, @Transactional, @Cacheable, @RequestParam
- Creación de una anotación
 - Logging
 - Gestión de transacciones
 - Monitorización de tiempo de procesamiento de métodos
 - Seguridad (Comprobación de credenciales)



Anotaciones

- Componentes de una anotación

- **@Target** – Especifica el tipo de elemento al que se va a asociar la anotación.
 - `ElementType.TYPE` – se puede aplicar a cualquier elemento de la clase.
 - `ElementType.FIELD` – se puede aplicar a un miembro de la clase.
 - `ElementType.METHOD` – se puede aplicar a un método
 - `ElementType.PARAMETER` – se puede aplicar a parámetros de un método.
 - `ElementType.CONSTRUCTOR` – se puede aplicar a constructores
 - `ElementType.LOCAL_VARIABLE` – se puede aplicar a variables locales
 - `ElementType.ANNOTATION_TYPE` – indica que el tipo declarado en sí es un tipo de anotación.
- **@Retention** – Especifica el nivel de retención de la anotación (cuándo se puede acceder a ella).
 - `RetentionPolicy.SOURCE` — Retenida sólo a nivel de código; ignorada por el compilador.
 - `RetentionPolicy.CLASS` — Retenida en tiempo de compilación, pero ignorada en tiempo de ejecución.
 - `RetentionPolicy.RUNTIME` — Retenida en tiempo de ejecución, sólo se puede acceder a ella en este tiempo.
- **@Documented** – Hará que la anotación se mencione en el javadoc.
- **@Inherited** – Indica que la anotación será heredada automáticamente.

Anotaciones

- Ejemplo de anotación
 - Anotación cuya función será la de monitorear el tiempo de ejecución del método anotado

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface LogueaTiempo {

    int maxTiempo() default 2000; //Máximo tiempo en milisegundos para lanzar una entrada en el log de tiempo excedido
}
```



Programación Orientada a Aspectos (AOP)

- ¿Qué es?
 - La programación orientada a aspectos (Aspect Oriented Programming AOP) es un paradigma de programación que intenta formalizar los elementos que son transversales “cross-cutting” a todo el sistema
 - Evita la duplicidad de código para tareas que se deben ejecutar a lo largo de varias clases y/o métodos
- Casos de uso
 - Logging
 - Gestión de transacciones
 - Monitorización de tiempo de procesamiento de métodos
 - Seguridad (Comprobación de credenciales)
 - Caché

Programación Orientada a Aspectos (AOP)

- Paradigma

- En los lenguajes orientados a objetos, la estructura del sistema se basa en la idea de *clases y jerarquías de clases*.
- La herencia permite modularizar el sistema, eliminando la necesidad de duplicar código. No obstante, siempre hay aspectos que son transversales a esta estructura: el ejemplo más clásico es el de control de permisos de ejecución de ciertos métodos en una clase:

```
public class MiObjetoDeNegocio {  
    public void metodoDeNegocio1() throws SinPermisoException {  
        chequeaPermisos();  
        //resto del código  
        ...  
    }  
  
    public void metodoDeNegocio2() throws SinPermisoException {  
        chequeaPermisos();  
        //resto del código  
        ...  
    }  
  
    protected void chequeaPermisos() throws SinPermisoException {  
        //chequear permisos de ejecucion  
        ...  
    }  
}
```

- El problema es que en POO los aspectos transversales no son modularizables ni se pueden formular de manera concisa



Programación Orientada a Aspectos (AOP)

- Aspecto (aspect)
 - En AOP, a los elementos que son transversales a la estructura del sistema y se pueden modularizar gracias a las construcciones que aporta el paradigma se les denomina Aspectos (aspects). En el ejemplo anterior el control de permisos de ejecución, modularizado mediante AOP sería un aspecto
- Consejo (advice)
 - Es una acción que hay que ejecutar en determinados puntos de un código para conseguir implementar un aspecto. Siguiendo con el ejemplo anterior, la acción a ejecutar sería la llamada a “chequeaPermisos()”.
- Punto de corte (Pointcut)
 - El conjunto de puntos del código donde se debe ejecutar un advice se conoce como pointcut. En el ejemplo los métodos metodoDeNegocio1() y metodoDeNegocio2()
 - Al definir un pointcut realmente no estamos todavía diciendo que vayamos a ejecutar nada, simplemente se marca como “punto de interés”.
 - La combinación de pointcut + advice es la que realmente hace algo útil



Programación Orientada a Aspectos (AOP)

- Punto de corte (Pointcut): Expresiones más comunes
 - La expresión más usada en pointcuts de Spring es `execution()`, que representa la llamada a un método que encaje con una determinada signatura. Se puede especificar la signatura completa del método incluyendo tipo de acceso (`public`, `protected`,...), tipo de retorno, nombre de clase (incluyendo paquetes), nombre de método y argumentos.
 - El tipo de acceso y el nombre de clase son opcionales, pero no así el resto de elementos
 - Podemos usar el comodín `*` para sustituir a cualquiera de ellos, y también el comodín `..`, que sustituye a varios tokens, por ejemplo varios argumentos de un método, o varios subpaquetes con el mismo prefijo.
 - En los parámetros, `()` indica un método sin parámetros, `(..)` indica cualquier número de parámetros de cualquier tipo, y podemos también especificar los tipos, por ejemplo `(String, *, int)` indicaría un método cuyo primer parámetro es `String`, el tercero `int` y el segundo puede ser cualquiera.
 - Para especificar todos los métodos con acceso "public" de cualquier clase dentro del paquete `org.cursospringboot.aop` pondríamos:

```
"execution( public * org.cursospringboot.aop.*(..) )"
```



Programación Orientada a Aspectos (AOP)

- Punto de corte (Pointcut): Expresiones más comunes

- La expresión más usada en pointcuts de Spring es `execution()`, que representa la llamada a un método que encaje con una determinada signatura. Se puede especificar la signatura completa del método incluyendo tipo de acceso (`public`, `protected`,...), tipo de retorno, nombre de clase (incluyendo paquetes), nombre de método y argumentos.
 - El tipo de acceso y el nombre de clase son opcionales, pero no así el resto de elementos
 - Podemos usar el comodín `*` para sustituir a cualquiera de ellos, y también el comodín `..`, que sustituye a varios tokens, por ejemplo varios argumentos de un método, o varios subpaquetes con el mismo prefijo.
 - En los parámetros, `()` indica un método sin parámetros, `(..)` indica cualquier número de parámetros de cualquier tipo, y podemos también especificar los tipos, por ejemplo `(String, *, int)` indicaría un método cuyo primer parámetro es `String`, el tercero `int` y el segundo puede ser cualquiera.
- Para especificar todos los métodos con acceso `"public"` de cualquier clase dentro del paquete `"org.cursospringboot.aop"` pondríamos:

`"execution(public * org.cursospringboot.aop.*.*(..))"`
- Donde el primer `*` representa cualquier tipo de retorno, el segundo `*` cualquier clase y el tercer `*` cualquier método. Los `..` representan cualquier conjunto de parámetros

Programación Orientada a Aspectos (AOP)

- Punto de corte (Pointcut): Expresiones más comunes

- Ejecución de cualquier getter (método público cuyo nombre comience por "get" y que no tenga parámetros). El tipo de acceso y el nombre de clase son opcionales, pero no así el resto de elementos

```
"execution(public * get*())"
```

- Ejecución de cualquier método public de cualquier clase en el paquete org.cursospringboot o subpaquetes

```
"execution(public * org.cursospringboot..*.*(..))"
```

- Ejecución de cualquier método de cualquier clase en el paquete org.cursospringboot que devuelva void y cuyo primer parámetro sea de tipo String Se omite el modificador de acceso

```
"execution (void org.cursospringboot.*.*(String,..))"
```

- Pointcuts con nombre

```
@Pointcut("execution(public * get*())")  
public void unGetterCualquiera() {}
```

```
@Pointcut("within(org.cursospringboot.negocio.*"))  
public void enNegocio() {}
```

```
@Pointcut("unGetterCualquiera() && enNegocio()")  
public void getterDeNegocio() {}
```



Programación Orientada a Aspectos (AOP)

- Aspecto (aspect)
 - Un *advice* es algo que hay que hacer en un cierto punto de corte, ya sea antes, después, o "alrededor" (antes y después) del punto.
 - Los *advices* se especifican con una anotación con el pointcut y la definición del método Java a ejecutar (signatura y código del mismo). Como en Spring los puntos de corte deben ser ejecuciones de métodos los casos posibles son:
 - Antes de la ejecución de un método (anotación @Before)
 - Después de la ejecución normal, es decir, si no se genera una excepción (anotación @AfterReturning)
 - Después de la ejecución con excepción/es (anotación @AfterThrowing)
 - Después de la ejecución, se hayan producido o no excepciones (anotación @After)
 - Antes y después de la ejecución (anotación @Around)

Programación Orientada a Aspectos (AOP)

- Aspecto (aspect)

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class EjemploAspect {

    @Before("execution(public * get*())")
    public void controlaPermisos() {
        // ...
    }

    @AfterReturning(pointcut="execution(public * get*())", returning="valor")
    public void log(Object valor) {
        // ...
    }

    @AfterThrowing(pointcut="execution(public * get*())", throwing="ex")
    public void logException(Exception ex) {
        // ...
    }

    @Around("execution(public * get*())")
    public Object ejemploAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("ANTES");
        Object valorRetorno = pjp.proceed();
        System.out.println("DESPUES");
        return valorRetorno;
    }
}
```



Programación Orientada a Aspectos (AOP)

- Caso práctico I
 - Implementar un medidor de tiempo para todas las funciones de un paquete el cual saque el tiempo consumido por dicha función mediante una traza
 - Agregar un tiempo configurable máximo de ejecución y en caso de superarlo lanzar una traza WARN
- Caso práctico II
 - Emular la creación de un Bean de sesión el cual contenga el rol del usuario logueado
 - No permitir el acceso a ciertos métodos protegidos si no contienen el rol específico