Perfiles

Los perfiles permiten registrar los **beans** en el contexto de Spring de acuerdo a alguna condición, por ejemplo:

-El sistema operativo
-El ambiente que se está utilizando(**dev**, **qa**, **prod**, **etc**.)
-El país para el que se ejecutará la aplicación

Veamos el siguiente ejemplo:

```
public interface EnvironmentService {
 String getEnvironmentName();
}
@Service
@Profile("dev")
public class DevEnvironmentService implements
EnvironmentService{
	@Override
	public String getEnvironmentName() {
		return "dev";
	}
}
@Service
@Profile("qa")
public class QaEnvironmentService implements
EnvironmentService{
	@Override
	public String getEnvironmentName() {
		return "qa";
	}
}
@Service
@Profile("prod")
public class ProdEnvironmentService implements
EnvironmentService{
	@Override
	public String getEnvironmentName() {
		return "prod";
	}
}
```

Se definió una interfaz llamada **EnvironmentService** y 3 implementaciones, cada una tiene un **profile** y un comportamiento diferente.

Definiendo el perfil a utilizar

Una vez definidos los **beans**, el siguiente paso es activar un perfil, para hacerlo podremos:

-Agregar la siguiente línea en el archivo **application.properties**:

```
spring.profiles.active=dev
```

-Definir la siguiente propiedad en la VM:

```
-Dspring.profiles.active=dev
```

-Anotar un test como se muestra:

```
@ActiveProfiles("dev")
```

Ejecución de la aplicación

Para ejecutar la aplicación utilizaremos el siguiente código:

```
@SpringBootApplication
public class Devs4jSpringCoreApplication {

public static void main(String[] args) {
  ConfigurableApplicationContext applicationContext =
SpringApplication.run(
Devs4jSpringCoreApplication.class, args);

  EnvironmentService service =
applicationContext.getBean(EnvironmentService.class);

  System.out.printf("Active environment %s ",
service.getEnvironmentName());
 }

}
```

Definiendo un perfil por defecto

Si se desea que un **bean** se registre en caso de que no se defina ningún perfil, es posible definir un perfil por defecto del siguiente modo:

```
@Service
@Profile({"dev","default"})
public class DevEnvironmentService implements
EnvironmentService{
	@Override
	public String getEnvironmentName() {
		return "dev";
	}
}
```

Spring scopes

Los **scopes** soportados por Spring framework son los siguientes:

Singleton - Crea una sola instancia del **bean** por contenedor de Spring

Prototype - Crea una nueva instancia cada vez que se solicita

Request - Crea una nueva instancia por cada petición HTTP, solo se puede utilizar en una aplicación web

Session - Crea una nueva instancia por cada sesión HTTP.

Puedes crear tu propio scope implementando la interfaz Scope

@Autowire con listas

Spring permite hacer **@Autowire** con objetos simples y con listas de objetos:

```
interface Figure {
	double calculateArea();
}
@Component
class Circle implements Figure {
@Override
	public double calculateArea() { ...  }
}
@Component
class Square implements Figure {
	@Override
	public double calculateArea() { ...  }
}
@Component
class AreaCalculator{
	@Autowired
	private List<Figure> figures;
	public double getTotalArea() {
		return figures.stream().
		mapToDouble(f->f.calculateArea()).sum();
	}
}
```

El bean **AreaClaculator** inyecta una lista de figuras, entre ellas estará el objeto Circle y el objeto Square.

Explicit bean declaration

Es posible declarar beans de forma explícita del siguiente modo:

```
@Configuration
public class BeanConfig {
	@Bean
	public Circle getCircle() {
		return new Circle();
	}
}
```

@SpringBootApplication

@SpringBootApplication es equivalente a definir las siguientes anotaciones:

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
```

Carga de properties

La carga de **properties** en una aplicación es un caso de uso muy común, a continuación se muestra paso a paso como realizarlo:

-Se creará un archivo llamado **areas.properties** en src/main/resources con lo siguiente:

```
circle.radius = 10.0
```

-Se definirá la siguiente configuración para cargar las **properties** al contexto de Spring:

```
@Configuration
@PropertySource("classpath:areas.properties")
public class FigurePropertyConfiguration {
	@Bean
	public PropertySourcesPlaceholderConfigurer
	loadProperties() {
		return new
PropertySourcesPlaceholderConfigurer();
	}
}
```

-Utilizar la información en nuestros **beans**:

```
@Component
	public class Circle {
	@Value("${circle.radius:0}")
	private double radius;

	public double getArea() {
		return Math.pow(Math.PI * radius, 2);
	}
}
```

El bean Circle utiliza la información que proviene del archivo properties haciendo uso de la anotación @Value.

SpEL

Spring expression language es un lenguaje de expresiones que permite realizar operaciones sobre la información en tiempo de ejecución, los operadores disponibles son:

Operadores aritmeticos +, -, *, /, %, ^, div, mod

Relacionales <, >, ==, !=, <=, >=, lt, gt, eq, ne, le, ge

Lógicos and, or, not, &&, ||, !

Condicionales ?:

Expresiones regulares Matchers

Evaluación de expresiones

Es posible evaluar expresiones **SpEL** sin iniciar el contexto de Spring como se muestra a continuación:

```
ExpressionParser expressionParser = new
SpelExpressionParser();
Expression expression =
expressionParser.parseExpression("10 + 20");
log.info("String expression {}", expression.getValue());
```

SpEL soporta funciones definidas por el usuario.

< DEVS4J >
Hecho por Devs para Devs

www.twitter.com/devs4j

www.facebook.com/devs4j