Desarrollo web en entorno servidor

UT1.2 – Programación web

5 – IoC en Spring – Beans – Definición y uso de beans

IES Clara del Rey – Madrid

Spring IoC – Beans

En Spring existe un contenedor de inversión de control, que es el responsable de crear las dependencias e inyectarlas en los objetos que las necesitan.

Todo el sistema de inyección de dependencias se mueve alrededor del concepto de "bean".

Un "bean" es cualquier objeto que el contenedor de IoC de Spring crea, gestiona, y controla.

Cualquier clase puede ser un bean. Sólo hay que definir, de alguna forma de las varias posibles, que esa clase es un bean.

Spring IoC – Beans – Declaración con XML

Es la forma más antigua de declarar beans. Algo en desuso en favor de la declaración con anotaciones. Para declarar el bean:

- Crear una clase para el bean. Es una clase normal y corriente, puede tener constructor, métodos, atributos, getters, setters, etc.
- Crear un fichero xml en la carpeta "resources" del proyecto Spring. El fichero puede llamarse como queramos, por ejemplo, "beans.xml"
- Registrar el bean en el XML.
- Usar la anotación *@ImportResource("classpath:beans.xml")* en la clase del programa principal, anotado con *@SpringBootApplication*

Spring IoC – Beans – Declaración con XML

Supongamos la siguiente clase que queremos que sea un bean:

```
public class HelloWorldXmlBean {
        private final String message;
        public HelloWorldXmlBean(String message){ this.message = message; }
        public void sayHello(){ System.out.println(message); }
}
```

Podremos declarar el bean en el xml de la siguiente forma:

```
<bean id="helloWorldXmlBean" class=".....HelloWorldXmlBean">
        <constructor-arg value="Hello, this is a message from XML!"/>
</bean>
```

El id puede ser cualquier valor que queramos. Hay que indicar el nombre completo de la clase, con paquetes. Se pueden pasar parámetros al constructor.

Spring IoC – Beans – Declaración con XML

El XML completo podría ser similar a este:

```
<?xml version="1.0" encoding="utf-8" ?>

<!-- Los namespaces se usan para que se identifiquen los elementos y se valide el XML -->

<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd">

        <!-- Definición del bean para mensajes -->
        <bean id="helloWorldXmlBean" class="es.paquete.paquete.HelloWorldXmlBean">
                <constructor-arg value="Hello, this is a message from XML!"/>
        </bean>

</beans>
```

Spring IoC – Beans
Declaración con anotaciones en la clase

Esta es la forma más habitual para declarar clases completas como beans.

En general, para que una clase sea un bean tiene que:

- Estar dentro del mismo paquete que la aplicación Spring. Puede ser en el mismo paquete o en un subpaquete.
- Estar anotada con @Component o anotación derivada. Las anotaciones derivadas más habituales son @Controller, @RescContoller, @Service, @Repository, @Configuration.

La aplicación Spring, al arrancar, escanea su paquete y todos los subpaquetes en busca de este tipo de anotaciones, y registra automáticamente los beans.

Spring IoC – Beans
Declaración con anotaciones en la clase

Ejemplo de clase declarada como bean usando anotación @Component.

```
@Component
public class HelloWorldAnnotatedBean {
        public void sayHello(){
                System.out.println(message);
        }
}
```

La aplicación Spring, al detectar la anotación @Component, automáticamente registra el bean para que pueda inyectarlo en aquellos objetos que necesiten una instancia de HelloWorldAnnotatedBean como dependencia.

Spring IoC – Beans
Declaración en clases de configuración

La última forma para crear beans es usar la anotación @Bean en un método de una clase anotada con @Configuration.

La aplicación Spring, al arrancar, busca todas las clases anotadas con @Configuration, y, dentro de ellas, los métodos anotados con @Bean.

Estos métodos tienen que devolver un objeto de la clase que se quiere declarar como bean.

Al igual que con @Component y derivados, sólo se escanea en busca de estas anotaciones el paquete de la aplicación y sus subpaquetes. Por defecto, cualquier clase que no esté dentro del paquete de la aplicación, será ignorada.

Spring IoC – Beans
Declaración en clases de configuración

Ejemplo de configuración que declara dos bean de dos tipos distintos:

```
@Configuration
public class ServicesConfiguration {

        @Bean
        public MessagingService getMessagingService(){
                return new EmailMessagingService();
        }

        @Bean
        public AuthenticationService getAuthenticationService(){
                return new JwtAuthenticationService();
        }
}
```

Spring – IoC – Uso de un bean

Para utilizar un bean, hay que "inyectarlo" en una clase para que lo utilice.

Hay tres formas de realizar la inyección de un bean:

- Inyección por constructor. Recomendada (código más "testeable“).
- Inyección por atributo.
- Inyección por setter.

Vamos a ver las tres, suponiendo que queremos inyectar un objeto de la clase "HelloWorldBean“.

Este bean se puede haber configurado por XML o por anotaciones (@Bean, @Component, @Service, etc.). Para la inyección, la forma en la que se ha declarado y configurado el bean es indiferente.

Spring – IoC – Inyección por constructor

La clase que recibe el bean (el bean es una dependencia que se inyecta) recibirá en su constructor un objeto del mismo tipo que el bean que queremos inyectar.

El constructor se debe encargar de utilizar esta dependencia recibida.

Suele haber dos casos:

- Guardar el bean en un atributo privado para usarlo posteriormente en distintos métodos de la clase. Lo más habitual.
- Usar el bean durante la construcción del objeto de la clase, y descartarlo si no se va a utilizar posteriormente.

La clase que recibe el Bean tiene que ser a su vez un bean, para que se haga correctamente la inyección de dependencias.

Spring – IoC – Inyección por constructor

Ejemplo de clase que recibe un bean HelloWorldBean y que lo guarda para usarlo posteriormente en un método.

```
@Component
public class ClaseConDependencias {

	// Atributo para guardar el bean para uso posterior
	private final HelloWorldBean helloWorldBean;

	// El constructor recibe el bean
	public ClaseConDependencias(HelloWorldBean dependencia){
		this.helloWorldBean = dependencia;
	}

	// Otros métodos que usan el objeto this.helloWorldBean.
}
```

Spring – IoC – Inyección por atributo

También conocida como inyección por campo o por propiedad. En otros lenguajes los campos y las propiedades son los atributos de Java.

En estos casos se utiliza la anotación “@Autowired”, para identificar aquellos atributos que deben ser inyectados por el contendor de IoC.

No es necesario que se use un constructor o un setter. El contendor de dependencias de Spring asignará un valor al atributo automáticamente.

Al igual que en la inyección por constructor, la clase que recibe el bean tiene que ser a su vez un bean, para que se haga correctamente la inyección de dependencias.

Este tipo de inyección es menos "testable" que la realizada en constructor, porque no se pueden usar "mocks" para sustituir a la dependencia "real"

Spring – IoC – Inyección por atributo

Ejemplo de la misma clase de ejemplo de inyección por constructor, modificada para que use inyección por atributo.

```
@Component
public class ClaseConDependencias {

        // Atributo para guardar el bean para uso posterior
        // No puede ser final porque en este tipo de inyección Spring inyecta
        // el bean DESPUÉS de la creación del objeto

        @Autowired
        private HelloWorldBean helloWorldBean;

        // No es necesario el constructor

        // Otros métodos que usan el objeto this.helloWorldBean.
}
```

Spring – IoC – Inyección por setter

Es, en lo fundamental, igual que la inyección por atributo, pero se hace colocando @Autowired al setter del atributo, no al atributo directamente.

Hay que tener en cuenta que:

- La dependencia inyectada puede cambiarse después de la inyección realizada por Spring. Esto hace que:
  - Sea más flexible, permite usar una dependencia distinta en ciertos casos.
  - Pero también puede ser más problemática, porque podemos, virtualmente, eliminar una dependencia, dejarla a null, porque podemos crear objetos de la clase sin pasar la dependencia al constructor.

Spring – IoC – Inyección por setter

Otra vez, la misma clase con dependencia por setter

```
@Component
public class ClaseConDependencias {

	// Atributo para guardar el bean para uso posterior. No puede ser final porque
	// Spring inyecta el bean DESPUÉS de la creación del objeto. Además, hay un setter.

	private HelloWorldBean helloWorldBean;

	@Autowired
	public void setHelloWorldBean(HelloWorldBean helloWorldBean){
		this.helloWorldBean = helloWorldBean;
	}

	// Otros métodos que usan el objeto this.helloWorldBean.
}
```

Buenas prácticas – Uso de interfaces

Aunque todos los ejemplos presentados muestran clases, se considera buena práctica el uso de interfaces para la inyección de dependencias.

Existe un principio de diseño denominado "Principio de inversión de dependencias" que dice: "Depende de la abstracción, no de los detalles".

Esto se traduce, en el contexto de la inyección de dependencias en Spring (y en cualquier framework similar) en:

- Cualquier clase que se quiera usar como un bean debería ser la implementación de una interfaz
- Cuando se inyecta la dependencia, se debe usar siempre la interfaz, no la implementación.

Buenas prácticas – Interfaces – Ejemplo

Supongamos una clase "MailMessagingService" que se quiere usar como bean e inyectar en otros beans:

```
@Component
public class MailMessagingService {
        // Atributos privados, no tiene constructor
        public void sendMessage(String message){ ... }
}
```

Para cumplir con el principio de inversión de dependencias, se debe extraer una interfaz del servicio:

```
 // Debe implementarse en la clase MailMessagingService
public interface MessagingService {
        void sendMail(String message); // Por defecto public abstract

}
```

Buenas prácticas – Interfaces – Ejemplo

Ya al hacer la inyección de dependencias (sea cual sea el mecanismo usado) se debe usar la interfaz, no la implementación:

```
@Service
public class ClaseQueRecibeDependencias {

        // Atributo para la dependencia inyectada
        private final MessagingService messagingService;

        // Inyección por constructor
        public ClaseQueRecibeDependencias(MessagingService messagingService){
                this.messagingService = messagingService;
        }
}
```

Múltiples implementaciones de interfaz

Cuando hay varias implementaciones de una interfaz inyectada, Spring puede tener dificultades para identificar cuál de las implementaciones se debe inyectar. Hay varias opciones:

- Especificar la implementación primaria, la implementación que se debe inyectar cuando se inyecte la interfaz. Esto se hace con la anotación "@Primary" en la implementación por defecto. Spring usará, salvo instrucciones en otro sentido, esa implementación siempre que se deba inyectar la interfaz.
- Al realizar la inyección de la interfaz, especificar qué implementación se desea utilizar, si es distinta a la implementación por defecto. Esto se hace con la anotación "@Qualifier" en la inyección (sea por constructor, por atributo o por setter).

Múltiples implementaciones – Ejemplo

Supongamos la interfaz "MessagingService", que se quiere usar como bean para inyectarla en distintos componentes.

```
public interface MessagingService {
        // sendMail es por defecto public abstract
        void sendMail(String message);
}
```

Esta clase se implementará de dos formas diferentes, una para enviar mensajes por correo electrónico, y otra para enviarlos por SMS.

Por defecto se desea que se use la implementación que envía mensajes por correo electrónico.

Múltiples implementaciones – Ejemplo

Habría dos implementaciones:

"MailMessagingService", implementación por defecto o "primaria":

```
@Component
@Primary
public classs MailMessagingService implements MessagingService {
        // Implementación de métodos abstractos de la interfaz para usar email
}
```

"SmsMessagingService", una implementación alternativa:

```
@Component
public classs SmsMessagingService implements MessagingService {
        // Implementación de métodos abstractos de la interfaz para usar SMS
}
```

Múltiples implementaciones – Ejemplo

Se inyectaría la implementación por defecto, pero puede forzarse la otra:

```
@Component
public classs ConDependenciaPorDefecto {
        private final MessagingService messagingService;
        public ConDependenciaPorDefecto (MessagingService messagingService){
        ...


@Component
public classs ConDependenciaAlternativa {
        private final MessagingService messagingService;
        public ConDependenciaPorDefecto (
                @Qualifier("smsMessagingService")
                MessagingService messagingService){
        ...
```

@Component y @Bean – Observaciones

Al usar @Component, Spring registra el bean con el mismo nombre de la clase, pero comenzando en minúscula. Así, @Component en la clase "MailService", registra el componente con el nombre/id "mailService".

Si se usa @Bean en un método se registra con el nombre del método.

Ambas anotaciones pueden recibir el nombre con el que se desea registrar el bean. Ejemplos:

- @Component("nombrePersonalizado")
- @Bean(name="nombrePersonalizado")

@Qualifier – Observaciones

Al usar @Qualifier, se debe usar el nombre/id del bean, que será:

- Si se ha usado @Service sin parámetro, el nombre de la clase comenzando en minúscula.
- Si se ha usado @Service con parámetro, el valor del parámetro usado.
- Si se ha usado @Bean sin parámetro en un método, el nombre del método.
- Si se ha usado @Bean con parámetro, el valor del parámetro usado.
- Si se ha registrado el bean en XML, el valor del atributo "id" en el fichero XML