Spring - Core Introducción

Francisco Philip

francisco.philip@gmail.com
@franciscophilip

Introducción a Spring

Que es? Que ofrece?

Que es?

Spring es un framework alternativo al stack de tecnologías estándar en aplicaciones Java EE. Nació en una época en la que las tecnologías estándar Java EE y la visión "oficial" de lo que debía ser una aplicación Java Enterprise tenían todavía muchas aristas por pulir. Los servidores de aplicaciones eran monstruos devoradores de recursos y los EJB eran pesados, inflexibles y era demasiado complejo trabajar con ellos.

En ese contexto, Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio, que suponían un soplo de aire fresco. Estas ideas permitían un desarrollo más sencillo y rápido y unas aplicaciones más ligeras. Eso posibilitó que de ser un framework inicialmente diseñado para la capa de negocio pasará a ser un completo stack de tecnologías para todas las capas de la aplicación.

Que ofrece?

Desde un punto de vista genérico, Spring se puede ver como un soporte que nos proporciona tres elementos básicos:

- Servicios enterprise
- Estereotipos configurables
- Inyección de dependencias

Servicios enterprise

Podemos hacer de manera sencilla que un objeto sea transaccional, o que su acceso esté restringido a ciertos roles, o que sea accesible de manera remota y transparente para el desarrollador, o acceder a otros muchos servicios más, sin tener que escribir el código de manera manual. En la mayoría de los casos solo es necesario anotar el objeto.

Estereotipos configurables

Podemos anotar nuestras clases indicando por ejemplo que pertenecen a la capa de negocio o de acceso a datos. Se dice que son configurables porque podemos definir nuestros propios estereotipos "a medida": por ejemplo podríamos definir un nuevo estereotipo que indicara un objeto de negocio que además sería cacheable automáticamente y con acceso restringido a usuarios con determinado rol.

Inyección de dependencias

Ya hemos visto este concepto cuando se hablaba de CDI de Java EE. La inyección de dependencias nos permite solucionar de forma sencilla y elegante cómo proporcionar a un objeto cliente acceso a un objeto que da un servicio que este necesita. Por ejemplo, que un objeto de la capa de presentación se pueda comunicar con uno de negocio. En Spring las dependencias se pueden definir con anotaciones o con XML.

Beneficios arquitectónicos de Spring

Beneficios

La misión principal de Spring es la de simplificar el desarrollo de aplicaciones Java, el mismo se acopla a tu aplicación sin requerir implementar interfaces o extender abstracciones en las funcionalidades. Esta característica , hasta el momento en que surgió este framework era impensable en otro frameworks. Permite a los desarrolladores desarrollar aplicaciones de clase empresarial utilizando POJOs.

Mediante anotaciones o configuraciones XML, Spring permite utilizar un muy amplio set de funcionalidades y API, así como una gran variada cantidad de integraciones. Esto nos permite que nuestro código sea limpio y enfocado en la funcionalidad, así como reutilizable, debido a su orientación a interfaces, que nos permite de manera fácil que toda nuestra aplicación sea altamente modular y posea bajo acoplamiento.

IoC

La inversión del control (Inversion of Control) está implementada en el módulo Core, que quizás es el más importante de Spring, ya que aplicando este patrón nos permite la aplicar el patrón Dependency Injection en nuestras aplicaciones.

El objetivo del contenedor IoC spring-core es encargarse de instanciar los objetos de nuestro sistema, denominados beans, y asignarle sus dependencias.

La idea de la DI es proveer a los objetos de nuestro sistema otros objetos, denominados colaboradores, que les permiten llevar a cabo su finalidad. El patrón es parecido al Builder, pero con un objetivo diferente, proveer desacoplamiento entre los objetos de nuestros sistema.

AOP

La orientación a aspectos es un eje del desarrollo con Spring, donde este pretende mediante esta técnica nos permite resolver aspectos claves en el desarrollo de aplicaciones como las operaciones que atraviesan varios puntos de la aplicación (Cross-cutting concerns), como lo son seguridad, trazabilidad y otros permitiéndonos aplicar patrones como DRY, SoC y 1:1, es decir “Una determinada funcionalidad debe estar implementada en un sólo sitio de nuestra aplicación y sólo debe llevar a cabo esa funcionalidad“..

avoiding boilerplate

Este término se refiere al código que es necesario para nuestra aplicación, pero que resulta muy pesado escribirlo una y otra vez, no siendo inútil, ya que este código ‘boilerplate‘ es necesario, sin él no funcionará nuestra aplicación, pero es verdaderamente molesto tener que estar escribiendo en cada método que lo necesitamos.

Ejemplos de este tipo de código lo encontramos cuando utilizamos JDBC , siendo muy molesto todos los pasos que debemos escribir para gestionar la conexión sin aún haber colocado una linea de codigo de la funcionalidad.

Para ello, el framework nos proporciona clases como JdbcTemplate, RestTemplate o JmsTemplate que permite mejorar mucho la calidad del código.

Conceptos Fundamentales de Spring

Que es un Bean?

Los objetos que forman la columna vertebral de su aplicación y que son administrados por el contenedor Spring IoC se denominan **beans** . Un bean es un objeto que está instanciado, ensamblado y gestionado por un contenedor Spring IoC.

Estos beans se crean con los metadatos de configuración que proporciona al contenedor. Por ejemplo, en la forma de XML <bean />.

Que es un Contexto?

En Spring, un contexto es donde se alojan los objetos beans, productores y diversos componentes de una arquitectura donde la dependencia de inyección es delegada mediante la inversión del control.

Que es Inyeccion?

La inyección de dependencias es quizás la característica más destacable del core de Spring Framework, que consiste que en lugar de que cada clase tenga que instanciar los objetos que necesite, sea Spring el que “inyecte” esos objetos, es decir que es Spring el que creará los objetos y cuando una clase necesite usarlos se le pasarán.

La inyección de dependencias es una forma distinta de diseñar aplicaciones y en muchos sitios usan los términos inyección de dependencia (DI) e inversión de control (IoC) de forma indistinta y aunque no son sinónimos, sino que más bien la inyección de dependencia sería una forma de inversión de control.

La DI consiste en que en lugar de que sean las clases las encargadas de crear (instanciar) los objetos que van a usar (sus atributos), los objetos se inyectaran mediante los métodos setters o mediante el constructor en el momento en el que se cree el objeto y cuando se quiera el objeto en cuestión ya estará lista, en cambio sin usar DI la clase necesita crear los objetos que necesita cada vez que se use.

BeanFactory y ApplicationContext

BeanFactory

BeanFactory

Los paquetes `org.springframework.beans` y `org.springframework.contexts` son la base del contenedor IoC de Spring Framework. La interface `BeanFactory` proporciona un mecanismo de configuración avanzada capaz de gestionar cualquier tipo de objeto. `ApplicationContext` es una sub-interface de `BeanFactory`. Se agrega:

- Integración más fácil con las características AOP de Spring
- Manejo de recursos de mensajes (para uso en internacionalización).
- Publicación de eventos

Contextos específicos de la capa de aplicación, como el `WebApplicationContext` uso en aplicaciones web están disponibles dentro del framework.

BeanFactory (cont)

En resumen, BeanFactory proporciona el marco de configuración y la funcionalidad básica, y ApplicationContext agrega una funcionalidad más específica de la empresa. El ApplicationContext es un super conjunto completo de BeanFactory .

Dependencias

Dependency Injection

Uno de los pilares del framework Spring

Existen en dos variantes principales, la inyección de dependencias basada en Constructor y la inyección de dependencias basada en Setter.

Inyección de Dependencias (DI)

Una aplicación empresarial típica no consiste en un solo objeto (o bean en el lenguaje Spring). Incluso la aplicación más simple tiene algunos objetos que trabajan juntos para presentar lo que el usuario final ve como una aplicación coherente.

La inyección de dependencia (DI) es un proceso mediante el cual los objetos definen sus dependencias (es decir, los otros objetos con los que trabajan) solo a través de argumentos de constructor, argumentos a un método de fábrica o propiedades que se establecen en la instancia del objeto después de que se construye o devuelto de un método de fábrica..

Inyección de Dependencias (cont)

El contenedor luego inyecta esas dependencias cuando crea el bean. Este proceso es fundamentalmente el inverso (de ahí el nombre, Inversión de control) del propio bean que controla la instanciación o ubicación de sus dependencias por sí mismo mediante la construcción directa de clases o el patrón del Localizador de servicios.

El código está más limpio con el principio DI y el desacoplamiento es más efectivo cuando los objetos cuentan con sus dependencias. El objeto no busca sus dependencias y no conoce la ubicación o clase de las dependencias. *Como resultado, sus clases se vuelven más fáciles de probar, particularmente cuando las dependencias están en interfaces o clases base abstractas, lo que permite el uso de implementaciones simuladas o simuladas en pruebas unitarias.*

DI basado en Constructor

El DI basado en el constructor se logra mediante el contenedor invocando a un constructor con varios argumentos, cada uno representando una dependencia. Llamar a un método static de factoría con argumentos específicos para construir el bean es casi equivalente, y esta discusión trata los argumentos a un constructor y a un método static de fábrica de manera similar.

DI basado en Setter

La DI basada en Setter se logra mediante el contenedor que llama a los métodos de establecimiento en sus beans después de invocar un constructor sin argumentos o un método static de factoría sin argumentos para crear una instancia de su bean.

Inyección de Colecciones

Uso de colecciones para la inyección

Mediante interfaces y Spring posee mecanismos para convertir valores al tipo target inyección. Estos mecanismos permite determinar tipos como colecciones donde mediante xml o declaraciones se pueden resolver dependecias.

Tipos de ámbito de una bean

Bean Scopes

Uno de los principales atributos que se utilizan en la definición de un bean es el atributo 'scope'. Dicho atributo permite definir el alcance de un bean. Spring posee, por defecto, 4 alcances principales:

- Singleton: Se crea una sola instancia del bean por IoC container.
- Prototype: Permite crear cualquier cantidad de instancias del bean.
- Request: Se crea automáticamente una instancia del bean por cada request. Podemos modificar el bean y solo será modificado para el request en el que nos encontramos trabajando, los otros request no verán estos cambios. Solo válido utilizando ApplicationContext para aplicaciones web.

Bean Scopes (cont)

- Session: Se crea automáticamente una instancia del bean por cada sesión. Podemos modificar el bean y solo será modificado para la session en la que nos encontramos trabajando, las otras session no verán estos cambios. Solo válido utilizando ApplicationContext para aplicaciones web.
- Global Session: Similar al Session, es utilizado por portlets para compartir la session entre los distintos portlets. En caso de que no estemos utilizando portlets este scope degrada al scope session. Solo válido utilizado ApplicationContext.
- Websocket: Una única definición de bean al ciclo de vida de un WebSocket. Solo válido en el contexto de un Spring ApplicationContext compatible con la web.

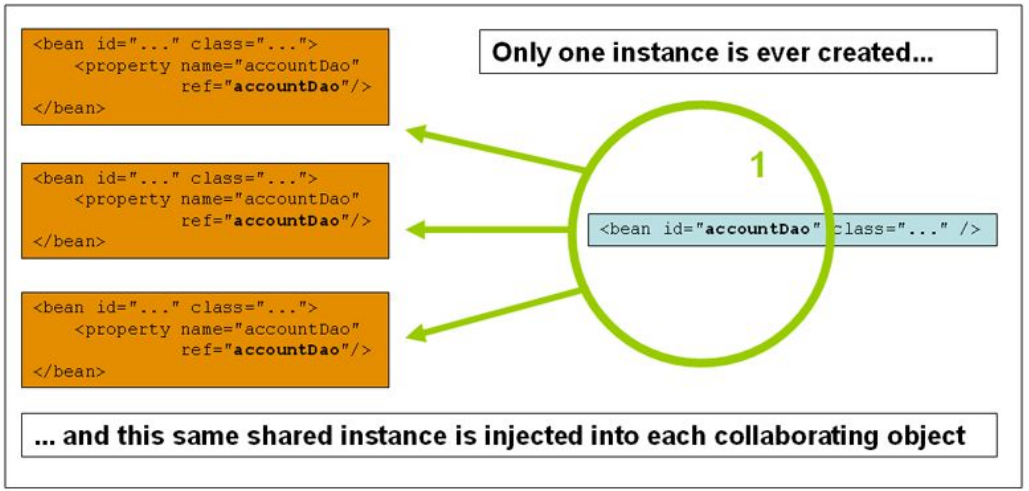
Singelton

Solo se gestiona una instancia compartida de un bean singleton, y todas las solicitudes de beans con una ID o ID que coinciden con la definición de bean dan como resultado que el contenedor Spring devuelva una instancia de bean específica.

Para decirlo de otra manera, cuando define una definición de bean y tiene un ámbito como singleton, el contenedor Spring IoC crea exactamente una instancia del objeto definido por esa definición de bean.

Esta instancia única se almacena en un caché de dichos beans singleton, y todas las solicitudes y referencias posteriores para ese bean con nombre devuelven el objeto almacenado en caché.

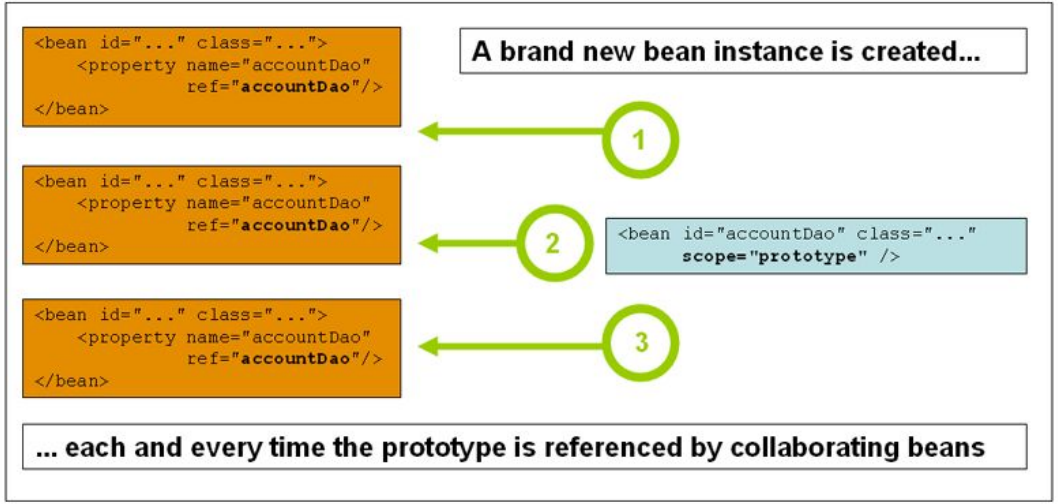
El concepto de spring de un bean singleton difiere del patrón singleton como se define en el libro de patrones de Gang of Four (GoF). El GoF singleton codifica el alcance de un objeto de tal manera que se crea una y solo una instancia de una clase particular por ClassLoader



Prototype

El scope prototype de la implementación del bean resulta en la creación de una nueva instancia de bean cada vez que se realiza una solicitud para ese bean específico. Es decir, el bean se inyecta en otro bean o lo solicita a través de una `getBean()` llamada de método en el contenedor. *Como regla general, debe utilizar el ámbito de prototipo para todos los beans con estado y el ámbito de singleton para los beans sin estado*

A diferencia de los otros ámbitos, Spring no administra el ciclo de vida completo de un bean prototipo. El contenedor crea una instancia, configura y, por lo demás, ensambla un objeto prototipo y se lo entrega al cliente, sin registro adicional de esa instancia de prototipo.



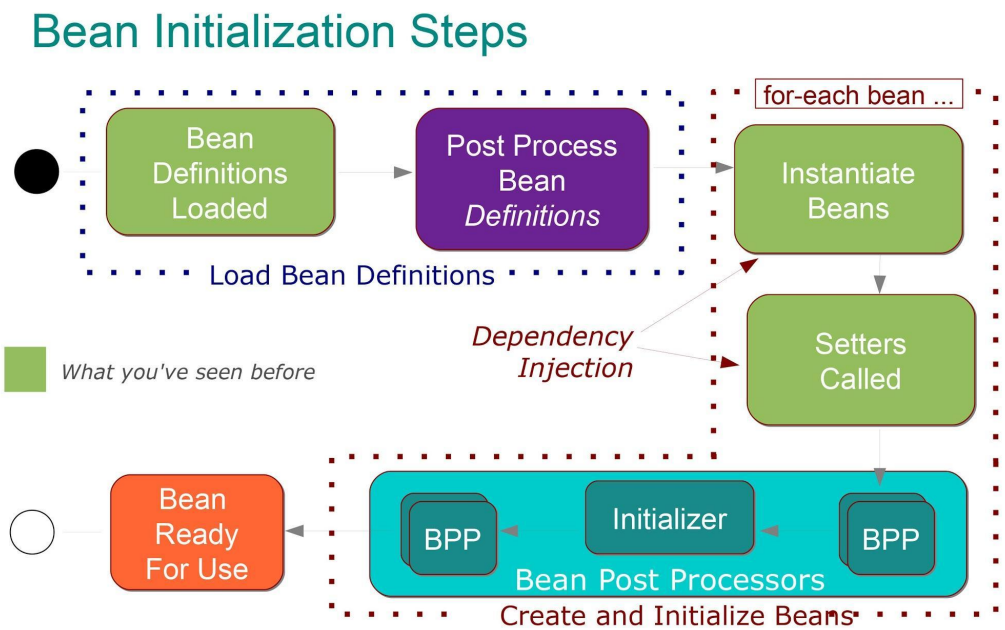
Web scopes

El request, session, application, y websocket scope están disponibles sólo si se utiliza Spring en la web consciente ApplicationContext (como `XmlWebApplicationContext`). Si utiliza estos ámbitos con contenedores Spring IoC regulares, como el `ClassPathXmlApplicationContext`.

`IllegalStateException` se lanza un reclamo sobre un alcance de bean desconocido.

Ciclo de vida de una bean

Creación de un Bean



Pasos del ciclo de vida

- Dentro del contenedor IoC, se crea un bean spring utilizando el constructor de clase.
- Luego la inyección de dependencia se realiza utilizando el método de setter.
- Una vez que se completa la inyección de dependencia, en caso con cumplir con la interfaz `BeanNameAware`, se invoca al metodo `setBeanName` de la misma, estableciendo el nombre de bean en la fábrica de beans donde se creó este bean.
- En caso de cumplir con la interfaz `BeanClassLoaderAware`., el metodo `setBeanClassLoader` es invocado, suministrando el classloader utilizado.
- En caso de ser un `BeanFactoryAware`, se invoca el método `setBeanFactory` que proporciona la factoría propietaria a una instancia de bean.
- Si el bean cumple con la interfaz `BeanPostProcessor`, el contenedor IoC llama al metodo `postProcessBeforeInitialization` en el bean.
- Se invocan los métodos anotados con `@PostConstruct`.
- Después de `@PostConstruct`, se llama al método `InitializingBean.afterPropertiesSet`
- Ahora se llama al método especificado por el atributo init-method de bean en la configuración XML.

Pasos del ciclo de vida (cont)

- Si el bean cumple con `BeanPostProcessor`, se invocará el método `postProcessAfterInitialization`. También se puede utilizar para aplicar envoltorio en el bean original.
- Ahora la instancia de bean está lista para ser usada. Realizar la tarea utilizando el bean.
- Ahora, cuando `ApplicationContext` se cierra, por ejemplo, mediante el uso de `registerShutdownHook`, se llama al método anotado con `@PreDestroy`.
- Después de eso, se llama al método `destroy` en el bean, en caso de ser un `DisposableBean`.
- Ahora se llama al método especificado por el atributo destroy-method de bean en la configuración XML.
- Antes de la recolección de basura, se llama al método `finalize` de Object.

InitializingBean y DisposableBean

Para interactuar con la administración del contenedor del ciclo de vida del bean, puede implementar las interfaces Spring `InitializingBean` y `DisposableBean`.

El contenedor requiere `afterPropertiesSet` que el primero y `destroy` el segundo le permitan al bean realizar ciertas acciones durante la inicialización y destrucción de los beans.

JSR 250

Las anotaciones `@PostConstruct` y las `@PreDestroy` pertenecen al estándar JSR-250.

Generalmente se consideran las mejores prácticas para recibir devoluciones de llamadas de ciclo de vida en una aplicación Spring moderna.

El uso de estas anotaciones significa que sus beans no están acoplados a interfaces específicas de Spring.

Lifecycle interfaces

Una interfaz común que define métodos para `start()/stop()` el control del ciclo de vida. El caso de uso típico para esto es controlar el procesamiento asíncrono.

Esta interfaz no implica semántica específica de inicio automático. Considere implementar `SmartLifecycle` para ese propósito.

Puede ser implementado por ambos componentes (típicamente un bean Spring definido en un contexto Spring) y contenedores (típicamente un Spring ApplicationContext). Los contenedores propagarán las señales de `start()/stop()` a todos los componentes que se aplican dentro de cada contenedor, por ejemplo. para un escenario de parada / reinicio en tiempo de ejecución.

LifecycleProcessor interface

Cualquier objeto gestionado por Spring puede implementar la interfaz `Lifecycle`. Luego, cuando el `ApplicationContext` mismo recibe señales de inicio y detención (por ejemplo, para un escenario de parada/reinicio en tiempo de ejecución), conecta en cascada esas llamadas a todas las `Lifecycle` implementaciones definidas dentro de ese contexto. Existen especificaciones como `SmartLifecycle` que suman más funcionalidades.

Interfaces “Aware”

Que son?

Las denominadas interfaces Aware, nos permiten obtener datos y componentes relacionados al contexto, como lo son el nombre del bean dentro del mismo, el classloader utilizado y por supuesto el contexto.

Tanto `BeanNameAware`, `BeanFactoryAware` como `BeanClassLoaderAware` pertenecen a la interfaz `org.springframework.beans.factory.Aware`.

La interfaz Aware es una combinación de patrones callback, listeners y observe, la misma indica que el bean es elegible para ser notificado por el contenedor Spring a través de los métodos de devolución de llamada.

ApplicationContextAware
BeanNameAware

Cuando se `ApplicationContext` crea una instancia de objeto que implementa la interface `ApplicationContextAware` la instancia se proporciona con una referencia a eso `ApplicationContext`.

```
public interface ApplicationContextAware {
	void setApplicationContext(ApplicationContext applicationContext) throws BeansException;
}
```

Cuando se crea `ApplicationContext` una clase que implementa la interface `BeanNameAware`, la clase recibe una referencia al nombre definido en su definición de objeto asociada.

```
public interface BeanNameAware {
	void setBeanName(String name) throws BeansException;
}
```

Otras Aware interfaces

Además `ApplicationContextAware` y `BeanNameAware` , Spring ofrece una gama de Aware interfaces que permiten a los beans indicar al contenedor que requieren una cierta dependencia de infraestructura. Como regla general, el nombre es una buena indicación del tipo de dependencia.

Nombre	Dependencia inyectada
`ApplicationContextAware`	Declarando `ApplicationContext`.
`ApplicationEventPublisherAware`	Editorial de eventos del recinto `ApplicationContext`.
`BeanClassLoaderAware`	Cargador de clases usado para cargar las clases de bean.

`BeanFactoryAware`	Declarando `BeanFactory`.
`BeanNameAware`	Nombre del bean declarante.
`BootstrapContextAware`	Adaptador de recursos en `BootstrapContext` el que se ejecuta el contenedor. Normalmente está disponible solo en `ApplicationContext`instancias con conocimiento de JCA .
`LoadTimeWeaverAware`	Tejedor definido para la definición de la clase de procesamiento en el tiempo de carga.
`MessageSourceAware`	Estrategia configurada para resolver mensajes (con soporte para parametrización e internacionalización).

`NotificationPublisherAware`	Publicador de notificaciones Spring JMX.
`ResourceLoaderAware`	Cargador configurado para acceso de bajo nivel a recursos.
`ServletConfigAware`	El `ServletConfig` contenedor se ejecuta en el momento. Válido solo en un Spring web `ApplicationContext`.
`ServletContextAware`	El `ServletContext`contenedor se ejecuta en el momento. Válido solo en un Spring web `ApplicationContext`.

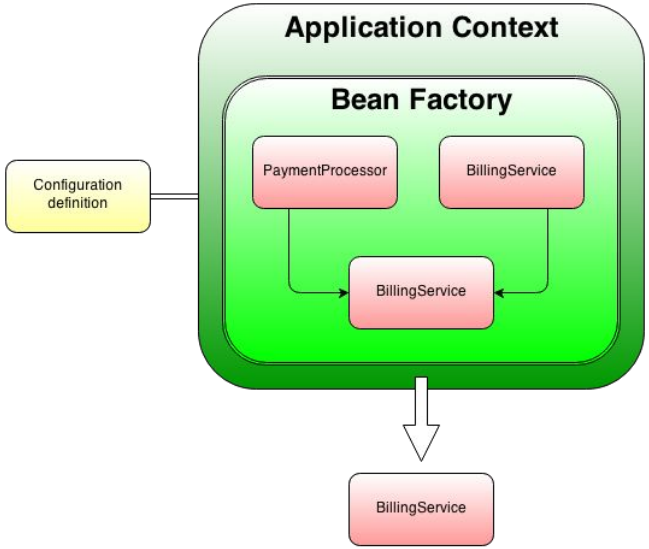
Application Context

Que son?

`ApplicationContext` es la interfaz central dentro de una aplicación Spring para proporcionar información de configuración a la aplicación.

Es de solo lectura en tiempo de ejecución, pero se puede volver a cargar si es necesario y es compatible con la aplicación.

Métodos de fábrica de beans para acceder a los componentes de la aplicación.



Bean Factory

BeanFactory interfaz

La interfaz `BeanFactory` proporciona un mecanismo de configuración simple y flexible para administrar objetos de cualquier naturaleza a través del contenedor Spring IoC.

BeanFactory contiene definiciones de bean y las crea una instancia cada vez que lo solicita la aplicación cliente, lo que significa:

- Se encarga del ciclo de vida de un bean mediante la creación de instancias y los métodos de destrucción apropiados.
- Es capaz de crear asociaciones entre objetos dependientes mientras se crea una instancia de ellos.

Es importante señalar que BeanFactory no admite la inyección de dependencia basada en anotación, mientras que ApplicationContext, un superconjunto de BeanFactory, sí lo hace.

FactoryBean

Diferencias

Un `FactoryBean` es una interfaz que usted, como desarrollador, implementa al escribir clases de fabrica y desea que el objeto creado por la fábrica sea administrado como un bean por Spring, mientras que `BeanFactory`, por otra parte, representa el contenedor Spring IoC, contiene los beans gestionados y proporciona acceso para recuperarlos. Es parte del núcleo del framework que implementa la funcionalidad básica de una inversión del contenedor de control.

Autowiring / Inject / Resource

Que es?

La anotación @Autowired / @Inject proporciona un control más preciso sobre dónde y cómo se debe realizar la conexión automática.

La anotación @Autowired se puede usar para autowire bean en el método de establecimiento tal como la anotación @Required, el constructor, una propiedad o métodos con nombres arbitrarios y / o múltiples argumentos.

Autowiring por Tipo

Autowiring por Nombres

Autowired en fields

Autowired en Setters

Autowired en Constructors

La anotación @Autowired también se puede utilizar en constructores.

Autowiring con Qualifiers

La anotación @Qualifier se usa en el campo o nivel de parámetro. Define un calificador para el @Autowiring. El uso de @Qualifier es para arreglar que el bean con un nombre dado solo califique para la inyección de dependencia en el cableado automático (autowiring). En el escenario en el que más de un bean es elegible para el cableado automático, podemos usar la anotación @Qualifier.

El mismo escenario también puede ser manejado por la anotación @Primary. @Qualifier define criterios de selección más sólidos que la anotación @Primary.

Autowired con Custom Qualifier

Las anotaciones del calificador personalizado se crean con el método value() o sin ningún método. También podemos usar uno o más de los métodos personalizados para crear calificadores personalizados. Los calificadores personalizados son útiles para corregir la inyección de dependencias de manera personalizada cuando más de un bean son elegibles para el autoconexión. Nuestro componente se puede marcar con un calificador personalizado a nivel de clase que participará en la selección de bean para inyección de dependencia.

Autowiring de Generic Types

AOP

Que es AOP?

La Programación orientada a los aspectos (AOP) complementa la Programación orientada a objetos (OOP) al proporcionar otra forma de pensar acerca de la estructura del programa. La unidad clave de modularidad en OOP es la clase, mientras que en AOP la unidad de modularidad es el aspecto. Los aspectos permiten la modularización de inquietudes (como la gestión de transacciones) que abarcan múltiples tipos y objetos. ("crosscutting concerns").

Uno de los componentes clave de Spring es el framework AOP. Si bien el contenedor Spring IoC no depende de AOP (lo que significa que no necesita usar AOP si no lo desea), AOP complementa Spring IoC para proporcionar una solución empresarial o middleware muy potente.

Capabilities

Capacidades y objetivos

Spring AOP se implementa en Java puro. No hay necesidad de un proceso de compilación especial. Spring AOP no necesita controlar la jerarquía del classloader y, por lo tanto, es adecuado para su uso en un contenedor de servlets o servidor de aplicaciones.

Spring AOP actualmente solo admite **join points** de ejecución de método (que **advising** la ejecución de métodos en Spring beans). La intercepción de campos no se implementa, aunque se podría agregar soporte para la intercepción de campo sin romper las API de Spring AOP principales.

El objetivo es proporcionar una integración estrecha entre la implementación de AOP y Spring IoC, para ayudar a resolver problemas comunes en aplicaciones empresariales.

Así, por ejemplo, la funcionalidad AOP de Spring Framework se usa normalmente junto con el contenedor Spring IoC. Los aspectos se configuran utilizando la sintaxis de definición de bean normal (aunque esto permite capacidades de "auto-proxying").

No puede hacer algunas cosas fácil o eficientemente con Spring AOP, como **advise** objetos de grano muy fino (generalmente, objetos de dominio). AspectJ es la mejor opción en tales casos.

Spring AOP es un framework basado en proxies.

Terminología

Terminología

Aspect: Es la modularización de un "**concern**" que atraviesa múltiples clases. La administración de transacciones es un buen ejemplo de un "asunto" transversal en las aplicaciones Java empresariales. En Spring AOP, los aspectos se implementan utilizando clases regulares (el enfoque basado en xml) o clases regulares anotadas con la anotación @Aspect).

Join point: Es el punto de aplicación que une la lógica de negocios de la aplicación con las **concern** centralizadas. Spring solo admite la ejecución de métodos como un **join point**, es decir, antes o después de la ejecución de un método (acción), la lógica de negocios puede unirse a un **concern** según sea necesario. Es un punto durante la ejecución de un programa, en Spring AOP, siempre representa una ejecución de método.

Advice: Acción tomada por un aspecto en un **join point** particular. Los diferentes tipos incluyen **advices** del tipo "**around**", "**before**" y "**after**". Muchos framework de AOP, incluido Spring, modelan un **advice** como un interceptor y mantienen una cadena de interceptores alrededor del **join point**. Es una acción que hay que ejecutar en determinado/s punto/s de un código, para conseguir implementar un aspecto.

Pointcut: Es un EL que se usa para definir la condición cuando el flujo de ejecución pasa de la lógica de negocios a las **concern** y luego regresa a la lógica de negocios. Una expresión PointCut se define como anotación @PointCut

Introduction: declaración de métodos o campos adicionales en nombre de un tipo. Spring AOP le permite introducir nuevas interfaces (y una implementación correspondiente) a cualquier objeto recomendado. Por ejemplo, podría usar una **introduction** para hacer que un bean implementa una interfaz `MyCache`, para simplificar el almacenamiento en caché. .

Target object: Un objeto que está siendo **advised** por uno o más aspectos. También se conoce como el "**advised object**". Dado que Spring AOP se implementa mediante el uso de proxies de tiempo de ejecución, este objeto siempre es un objeto de proxy.

Proxy AOP: un objeto creado por el framework de AOP para implementar los contratos de aspectos (**advice** sobre ejecuciones de métodos, etc.). En Spring, un proxy AOP es un proxy dinámico JDK o un proxy CGLIB.

Weaving: vinculación de aspectos con otros tipos de aplicación u objetos para crear un objeto aconsejado. Esto se puede hacer en tiempo de compilación (utilizando el compilador AspectJ, por ejemplo), tiempo de carga o en tiempo de ejecución. Spring AOP, al igual que otros marcos de Java AOP puros, realiza el tejido en tiempo de ejecución.

Tipos de advice

Tipos de advises

Before: se ejecuta antes de un **join point** pero que no tiene la capacidad de evitar que el flujo de ejecución avance al punto de unión (a menos que arroje una excepción).

After return: se debe ejecutar después de que un **join point** se complete normalmente (por ejemplo, si un método regresa sin lanzar una excepción).

After throwing : se debe ejecutar el **advise** si sale un método lanzando una excepción.

After (finally) : **advises** a ser ejecutado independientemente de los medios por los cuales sale un **join point** (retorno normal o excepcional).

Around: **advises** que rodean un **join point**, como la invocación de un método. Este es el tipo de **advises** más completo, se puede ejecutar un procedimiento personalizado antes y después de la invocación del método. También es responsable de elegir si proceder al **join point** o al **shortcut advised** del método recomendado devolviendo su propio valor de retorno o lanzando una excepción.

jee - cdi

En el estándar JEE, existe el concepto de interceptor y se encuentran bajo el paquete `javax.interceptor.*`

Eventos

Que son?

Existe una variedad de eventos integrados en Spring, que permiten que un desarrollador se enganche en el ciclo de vida de una aplicación y el contexto y realice alguna operación personalizada.

Aunque raramente usamos estos eventos manualmente en una aplicación, el marco lo usa intensamente dentro de sí mismo. Comencemos por explorar varios eventos incorporados en Spring.

ContextRefreshedEvent

Al inicializar o actualizar `ApplicationContext`, Spring genera el objeto `ContextRefreshedEvent`. Normalmente, una actualización puede activarse varias veces siempre que el contexto no se haya cerrado.

Tenga en cuenta que también podemos hacer que el evento se active manualmente llamando al método `refresh()` en la interfaz `ConfigurableApplicationContext`.

Este evento se publica cuando se cierra `ApplicationContext`, utilizando el método `close()` en `ConfigurableApplicationContext`.

En realidad, después de cerrar un contexto, no podemos reiniciarlo.

ContextStartedEvent

Al llamar al método `start()` en el `ConfigurableApplicationContext`, desencadenamos este evento e iniciamos el `ApplicationContext`. De hecho, el método se usa normalmente para reiniciar beans después de una parada explícita. También podemos usar el método para tratar componentes sin configuración para el inicio automático.

Aquí, es importante tener en cuenta que la llamada a `start()` siempre es explícita en lugar de `refresh()`.

ContextStoppedEvent

Un `ContextStoppedEvent` se publica cuando se detiene `ApplicationContext`, invocando el método `stop()` en el `ConfigurableApplicationContext`. Como se mencionó anteriormente, podemos reiniciar un evento detenido utilizando el método `start()`.

ContextClosedEvent

Este evento se publica cuando se cierra `ApplicationContext`, utilizando el método `close()` en `ConfigurableApplicationContext`.

En realidad, después de cerrar un contexto, no podemos reiniciarlo.

Un contexto llega al final de su vida útil al cerrarlo y, por lo tanto, no podemos reiniciarlo como en un `ContextStoppedEvent`.

@EventListener

`@EventListener` es una anotación central y, por lo tanto, no necesita ninguna configuración adicional. De hecho, el elemento <context: annotation-driven /> existente le proporciona soporte completo.

Se utiliza para declarar el método que recibe el mismo:

```
@EventListener
public void handleContextRefreshEvent(ContextStartedEvent ctxStartEvt) {
    System.out.println("Context Start Event received.");
}
```

Un método anotado con `@EventListener` puede devolver un tipo no vacío. Si el valor devuelto no es nulo, el mecanismo de eventos publicará un nuevo evento para él.

Links y notas

https://github.com/bcntec-learning/

https://www.concretepage.com/spring/
https://howtodoinjava.com/spring-core
https://www.baeldung.com/spring-5
https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/
http://java2novice.com/spring/
https://www.arquitecturajava.com/categoria/spring/spring-core/

LINKS