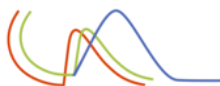




Contenido

SPRING – MÓDULO CORE	2
INTRODUCCIÓN.....	2
¿Qué es el módulo core de Spring?.....	2
¿Por qué hablar de anotaciones?	2
Componentes clave del módulo core (con enfoque en anotaciones)	3
Inversión de Control (IoC) e Inyección de Dependencias (DI) — con anotaciones	3
CONFIGURACIÓN BASADA EN JAVA (ANOTACIONES)	6
Cómo arrancar un contexto de anotaciones	6
¿Y los XML?	7
PRINCIPALES ANOTACIONES DEL MÓDULO CORE: DESCRIPCIÓN, USO Y CONSIDERACIONES	7
BUENAS PRÁCTICAS Y CONSIDERACIONES DIDÁCTICAS	9
QUÉ OCURRE (LÍNEA TEMPORAL) AL EJECUTAR UNA APLICACIÓN SPRING CENTRADA EN EL MÓDULO CORE	10
0) Punto de arranque (bootstrap).....	10
1) Preparación del <i>Environment</i> y registro de <i>PropertySources</i>	10
2) Registro / lectura de definiciones de beans (<i>BeanDefinition</i>)	10
3) Fase de <i>BeanDefinitionRegistryPostProcessor</i> (por ejemplo: parsing de <i>@Configuration</i>).....	11
4) Fase de <i>BeanFactoryPostProcessor</i>	11
5) Registro de <i>BeanPostProcessor</i> y otras infraestructuras	11
6) Instanciación de beans singleton (creación y pre-procesado)	11
7) Resolución de ambigüedades en autowiring	12
8) Publicación del evento <i>ContextRefreshedEvent</i>	12
9) Registro de <i>shutdown hook</i> (si procede) y la aplicación corre.....	12
10) Cierre del contexto (shutdown) — destrucción ordenada	12
CASOS ESPECIALES Y NOTAS PRÁCTICAS.....	13
ENLACES DE INTERÉS	14



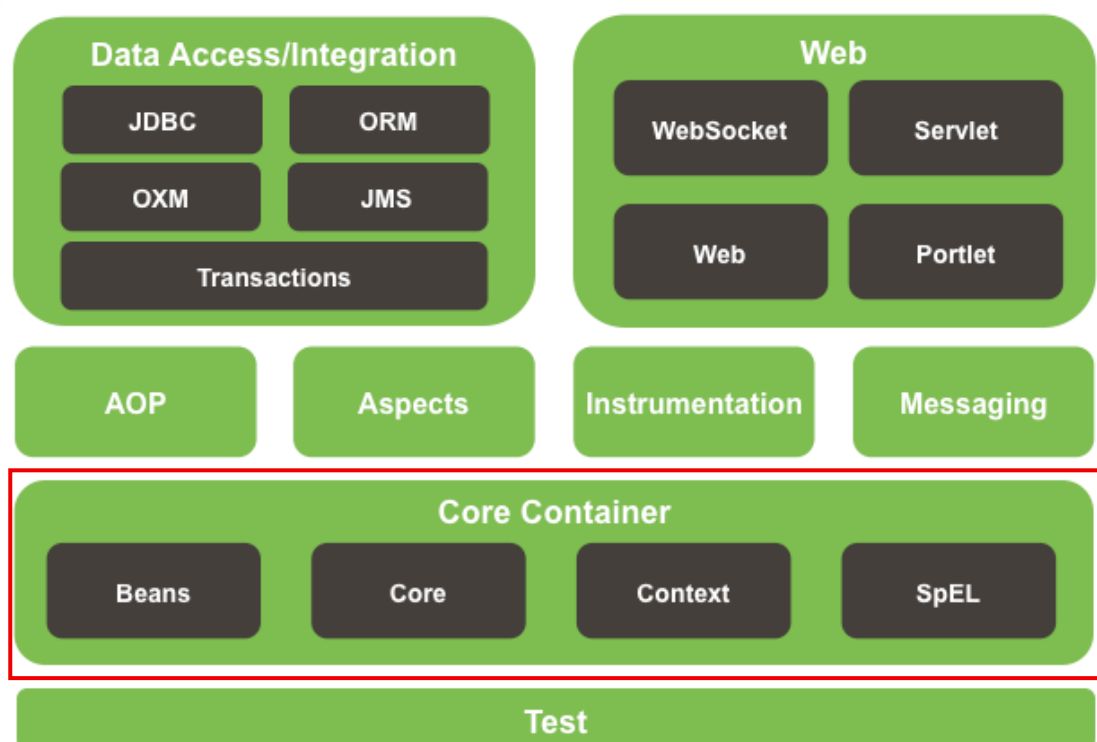
INTRODUCCIÓN

¿Qué es el módulo core de Spring?

El módulo *core* de Spring es el núcleo del framework. Proporciona las funcionalidades fundamentales como el contenedor de inversión de control (IoC) y la inyección de dependencias (DI), que permiten crear aplicaciones modulares, extensibles y fáciles de mantener.

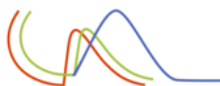
A partir de las versiones modernas de Spring, la **configuración basada en anotaciones** (annotation-based configuration) juega un papel central en ese módulo core.

En esencia, IoC (Inversión de Control) es el principio en el que el contenedor de Spring gestiona la creación y gestión de objetos, en lugar de que el programador lo haga directamente. DI (Inyección de Dependencias) es la implementación común de IoC en Spring: los objetos declaran qué dependencias necesitan, y el contenedor se las inyecta.



¿Por qué hablar de anotaciones?

La evolución de Spring ha ido migrando de una configuración mayoritariamente basada en XML hacia una configuración por anotaciones y clases Java-config. El módulo core admite explícitamente esta vía de configuración basada en anotaciones como una forma más moderna, tipo seguro (type-safe), y fácil de mantener. Por ello, cuando presentamos el módulo core hoy, conviene centrar la atención en cómo se usan esas **anotaciones** para registrar beans, escanear componentes, definir dependencia, establecer ámbito, etc.



Componentes clave del módulo core (con enfoque en anotaciones)

1. BeanFactory y ApplicationContext

Aunque no se usan directamente en muchos casos cuando se emplean las anotaciones, conviene conocerlos:

- **BeanFactory**: interfaz original del contenedor IoC de Spring, con funcionalidades básicas para gestionar beans.
- **ApplicationContext**: subinterfaz de BeanFactory que añade funcionalidades adicionales (eventos, internacionalización, etc.). En la configuración por anotaciones, es común usar clases como `AnnotationConfigApplicationContext` para instanciar el contexto a partir de clases de configuración anotadas.

2. Beans gestionados por el contenedor

Los *beans* son objetos cuya vida (creación, configuración, destrucción) está gestionada por el contenedor de Spring. Con anotaciones podemos declarar beans de distintas maneras: mediante clases anotadas con estereotipos (como `@Component`), mediante métodos anotados con `@Bean`, etc. El uso de anotaciones reduce la necesidad de XML, facilita el refactoring y mejora la claridad del código.

3. Uso de anotaciones en el módulo core

Algunas de las anotaciones más importantes que entran dentro del módulo core incluyen:

- **@Configuration**: Indica una clase que declara uno o más métodos `@Bean`.
- **@Bean**: Indica que un método produce un bean que debe registrarse en el contenedor.
- **@Component, @Service, @Repository, @Controller**: Estereotipos que permiten el escaneo de componentes.
- **@Autowired, @Qualifier, @Primary**: Anotaciones para inyección de dependencias automática y resolución de ambigüedades.
- **@Scope, @Lazy, @DependsOn**: Para controlar el ámbito del bean, inicialización perezosa, orden de creación, etc.
- **@Value**: Para inyectar valores simples (cadenas, primitivos) a campos o parámetros.

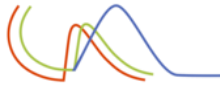
Más adelante profundizaremos en cada anotación.

Inversión de Control (IoC) e Inyección de Dependencias (DI) — con anotaciones

La inyección de dependencias es la forma en que Spring implementa IoC: los objetos (beans) declaran qué dependencias necesitan y el contenedor se las proporciona. En la era de anotaciones, se puede hacer lo siguiente:

Qué cambia con anotaciones

- Ya no es necesario declararlo todo mediante XML (aunque sigue siendo posible) — ahora podemos usar clases de configuración Java y anotaciones.
- El escaneo de componentes (component-scanning) permite detectar clases anotadas automáticamente.



- Se usan anotaciones en campos, constructores o métodos setter para marcar las dependencias a inyectar.

Ejemplo:

```
public interface MovieFinder {

    /**
     * Devuelve la lista completa de películas disponibles.
     * Este contrato define el "qué", pero no el "cómo".
     */
    List<String> findAll();

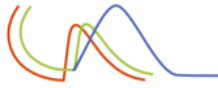
    /**
     * Busca una película por su título exacto o parcial.
     * @param title el título de la película.
     * @return el título si existe, o un mensaje si no se encuentra.
     */
    String findByTitle(String title);
}
```

```
@Component
public class SimpleMovieFinder {

    // Simulación de un origen de datos (puede ser BD, API, fichero, etc.)
    private final List<String> movies = List.of(
        "The Matrix",
        "Inception",
        "Interstellar",
        "The Godfather",
        "Pulp Fiction"
    );

    public List<String> findAll() {
        // Aquí podrías consultar una BD, un repositorio o un servicio REST
        return movies;
    }

    public String findByTitle(String title) {
        return movies.stream()
            .filter(m -> m.equalsIgnoreCase(title))
            .findFirst()
            .orElse("Película no encontrada");
    }
}
```



```
@Configuration
public class AppConfig {

    @Bean
    public MovieFinder movieFinder() {
        return new SimpleMovieFinder();
    }

    @Bean
    public SimpleMovieLister movieLister(MovieFinder movieFinder) {
        return new SimpleMovieLister(movieFinder);
    }
}

public class SimpleMovieLister {
    private final MovieFinder movieFinder;

    @Autowired // inyección automática por constructor
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

En este ejemplo:

- La clase de configuración está anotada con `@Configuration`.
- Se definen métodos `@Bean` para los beans que queremos que Spring gestione.
- En la clase `SimpleMovieLister`, se usa `@Autowired` en el constructor para que Spring inyecte la dependencia `MovieFinder`.

Esta forma sustituye muchas de las secciones de XML que antes explicaban inyección por constructor o setter.

Constructor vs Setter vs Campo

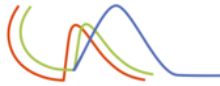
Aunque con anotaciones se tiende a usar la inyección por constructor (por claridad, inmutabilidad y facilidad de pruebas), las otras modalidades siguen existiendo:

- **Constructor:** preferida para dependencias obligatorias.
- **Setter:** útil para dependencias opcionales o cambiantes.
- **Campo:** se puede inyectar directamente en un campo con `@Autowired`, aunque muchos advierten que es menos indicado para pruebas.

Resolución de ambigüedades

Cuando hay múltiples beans compatibles con una dependencia, podemos usar:

- `@Qualifier("beanName")` para especificar el nombre.
- `@Primary` en la definición de bean para declarar que es la opción predeterminada.



Beneficios del enfoque por anotaciones

- Disminuye el "ruido" de XML.
- Mejora la inyección en tiempo de compilación (type-safety).
- Facilita el refactoring y mantenimiento.
- Integra mejor con la comunidad moderna de Spring.

CONFIGURACIÓN BASADA EN JAVA (ANOTACIONES)

Cómo arrancar un contexto de anotaciones

Una de las clases que se usa con anotaciones para arrancar el contenedor es `AnnotationConfigApplicationContext` (o variantes como en contexto web). Esta clase inicialmente carga una o varias clases de configuración anotadas.

Spring Core puro

```
@Configuration
@ComponentScan(basePackages = "com.hazerta.core")
public class AppConfig {}

public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(AppConfig.class);

        MovieLister lister = context.getBean(MovieLister.class);
        lister.showMovies();
        context.close();
    }
}
```

El programador gestiona: La creación del contexto, su cierre `context.close()` y la configuración del paquete a escanear.

Spring Boot

```
@SpringBootApplication
public class CoreApplication {

    public static void main(String[] args) {
        SpringApplication.run(CoreApplication.class, args);
    }
}
```

Spring Boot internamente crea el mismo contexto (`ApplicationContext`), pero de forma automática, configurando todo por convención.



¿Y los XML?

Aunque las anotaciones son muy usadas, Spring sigue soportando la configuración XML: muchas de las bases históricas siguen vigentes. La ventaja de anotaciones es la claridad y la integración con Java. La documentación del módulo core lo considera como parte de la "Annotation-based Container Configuration".

PRINCIPALES ANOTACIONES DEL MÓDULO CORE: DESCRIPCIÓN, USO Y CONSIDERACIONES

@Configuration

- Marca una clase como fuente de definiciones de beans.
- Es equivalente (en configuración Java) a un archivo XML que contiene definiciones de bean.
- Dentro de una clase @Configuration, los métodos anotados con @Bean producen beans que Spring registra.
- Ejemplo:

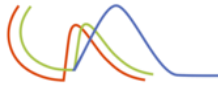
```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new FooImpl();
    }
}
```

@Bean

- Se aplica a métodos dentro de una clase @Configuration.
- El retorno del método es el bean que Spring registrará.
- Permite especificar atributos como name, initMethod, destroyMethod.
- Cuando se combina con escaneo de componentes, un método @Bean puede incluso sobrescribir (o no) una definición generada automáticamente.

@Component (y estereotipos derivados: @Service, @Repository, @Controller)

- @Component: anotación genérica para marcar una clase como bean gestionado por Spring.
- @Service: especialización de @Component para clases de lógica de negocio.
- @Repository: especialización para capa de acceso a datos; además puede involucrar traducción automática de excepciones de persistencia.
- @Controller: especialización para la capa de presentación (MVC). Gracias al escaneo de componentes (por ejemplo con @ComponentScan), Spring detecta estas clases automáticamente.



@Autowired

- Marca un constructor, método o campo para que la dependencia sea inyectada automáticamente por Spring.
- Si hay múltiples beans candidatos, se puede combinar con @Qualifier o @Primary para decidir cuál usar.
- Ejemplo de inyección en el constructor:

```
@Service
public class OrderService {
    private final PaymentProcessor processor;

    @Autowired
    public OrderService(PaymentProcessor processor) {
        this.processor = processor;
    }
}
```

@Qualifier / @Primary

- @Qualifier("beanName") permite elegir específicamente cuál bean inyectar cuando hay ambigüedad.
- @Primary marca un bean como la opción predeterminada cuando se requiere por tipo.

@Scope

- Permite definir el ámbito (scope) del bean, como singleton (por defecto), prototype, etc.
- Ejemplo:

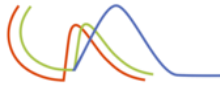
```
@Scope("prototype")
@Component
public class TaskProcessor { ... }
```

@Lazy

- Indica que el bean debe inicializarse de forma perezosa (cuando se requiere), y no necesariamente al arranque del contexto.
- Útil para optimizar arranque o para dependencias costosas.

@DependsOn

- Indica que este bean depende de uno o más beans, de modo que esos deben inicializarse primero.
- Ejemplo:



```
@Component
@DependsOn("cacheInitializer")
public class ReportGenerator { ... }
```

@Value

- Inyecta valores simples, por ejemplo, de propiedades externas o literales, directamente en campos o parámetros.
- Ejemplo:

```
@Component
public class DatabaseConfig {
    @Value("${db.url}")
    private String url;
}
```

@ComponentScan

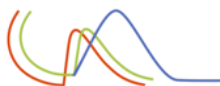
- Se coloca en una clase de configuración para indicar los paquetes que Spring debe escanear en busca de componentes anotados (@Component, @Service, etc.).
- Ejemplo:

```
@Configuration
@ComponentScan(basePackages = "com.example.app")
public class AppConfig { ... }
```

Estos son los juegos de anotaciones más centrales en el módulo core de Spring. Existen muchas más (incluyendo meta-anotaciones), pero estas proporcionan la base que un desarrollador debe dominar.

BUENAS PRÁCTICAS Y CONSIDERACIONES DIDÁCTICAS

- Prefiere la **inyección por constructor** cuando la dependencia es obligatoria y deseas promover inmutabilidad (immutability) y facilidad de pruebas.
- Usa @Autowired en el constructor en vez de en campos cuando sea posible (mejor visibilidad de dependencias).
- Evita depender excesivamente de inyección por campo, pues reduce claridad en pruebas unitarias.
- Usa @Qualifier o @Primary para resolver ambigüedades de beans; mantén los nombres de beans descriptivos.
- Evita la sobre-configuración: si tu proyecto ya usa configuración por anotaciones, intenta limitar el uso de XML solo cuando sea necesario.



- Mantén tus clases de configuración (@Configuration) organizadas (por módulo, funcionalidad) para que la navegación sea más clara.
- Ten en cuenta el ciclo de vida de los beans: la inicialización temprana/perezosa, el alcance, la destrucción, etc. Aunque con anotaciones muchas cosas se manejan automáticamente, conviene saber cómo funciona.

QUÉ OCURRE (LÍNEA TEMPORAL) AL EJECUTAR UNA APLICACIÓN SPRING CENTRADA EN EL MÓDULO CORE

Al ejecutar una app Spring centrada en *core*, el ApplicationContext se construye, registra y procesa metadatos (BeanDefinition), ejecuta post-procesadores que transforman dichas definiciones, registra procesadores e infraestructuras, crea y rellena los beans (resolviendo dependencias y aplicando BeanPostProcessor para inyección/AOP), publica el evento de "contexto listo" y, al cerrar, ejecuta los callbacks de destrucción en orden.

0) Punto de arranque (bootstrap)

- Si usas Spring "puro" con configuración Java, típicamente hay un main que crea el contexto:

```
AnnotationConfigApplicationContext ctx = new  
AnnotationConfigApplicationContext(AppConfig.class);
```

- Si usas Spring Boot, SpringApplication.run(...) internamente crea y refresca un ApplicationContext (usualmente AnnotationConfigApplicationContext o una subclase).

Por qué importa: este paso crea la **instancia del contexto** que gestionará las definiciones de beans y su ciclo de vida.

1) Preparación del *Environment* y registro de *PropertySources*

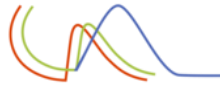
- Antes de cargar beans, el contexto construye su Environment (propiedades, perfiles, property sources: application.properties, variables de entorno, argumentos de línea de comandos).
- Estas propiedades se registran como PropertySource y quedan disponibles para @Value, PropertySourcesPlaceholderConfigurer, y la resolución de valores en @Configuration.

Por qué importa: las propiedades configuran beans (URLs, credenciales, flags de perfil) y afectan qué beans se crean.

2) Registro / lectura de definiciones de beans (BeanDefinition)

- El contexto obtiene definiciones de beans de las fuentes configuradas:
 - Clases @Configuration y métodos @Bean.
 - Component scanning → detecta @Component, @Service, @Repository, @Controller.
 - Definiciones registradas programáticamente (p. ej. register(Class...)).
- Internamente cada bean se representa como una **BeanDefinition** (metadatos: clase, scope, init/destroy, dependencia).

Por qué importa: hasta este punto solo existen metadatos; aún no se han instanciado los singletons.



3) Fase de *BeanDefinitionRegistryPostProcessor* (por ejemplo: parsing de *@Configuration*)

- Antes de crear beans, Spring invoca los *BeanDefinitionRegistryPostProcessor* registrados (un subtipo de *BeanFactoryPostProcessor*).
- Un ejemplo importante: *ConfigurationClassPostProcessor* — es el responsable de procesar las clases *@Configuration*, *@Bean*, *@ComponentScan*, *@Import*, y generar más *BeanDefinition* basadas en anotaciones. Esto puede registrar beans adicionales que aparecieron por escaneo o por *@Import*.

Por qué importa: transforma/expande las definiciones antes de que cualquier bean sea instanciado. Sin esto, las clases anotadas no producirían sus *BeanDefinition*.

4) Fase de *BeanFactoryPostProcessor*

- Tras registrar todas las *BeanDefinition*, Spring ejecuta los *BeanFactoryPostProcessor* (como *PropertySourcesPlaceholderConfigurer*) que permiten modificar las definiciones (p. ej. resolver placeholders *\${...}* en las propiedades de definición).

Por qué importa: garantiza que las definiciones finales (tipos, valores, referencias) estén correctas antes de crear instancias.

5) Registro de *BeanPostProcessor* y otras infraestructuras

- Spring registra *BeanPostProcessor* (p. ej. *AutowiredAnnotationBeanPostProcessor*) que interceptarán la creación de beans para realizar tareas como inyección *@Autowired*, procesamiento de *@Value*, creación de proxies AOP, etc.
- También se registran AOP infrastructure, *LifecycleProcessor*, conversion service, *ApplicationEventMulticaster*, y demás componentes infraestructurales.

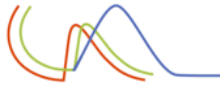
Por qué importa: los *BeanPostProcessor* son los hooks que realizan la inyección automática y transformaciones sobre los objetos antes/después de su inicialización.

6) Instanciación de beans singleton (creación y pre-procesado)

- Spring recorre las *BeanDefinition* configuradas para los beans singleton (por defecto) y comienza a instanciarlas en el orden que sea necesario (respetando *depends-on* y resolución de dependencias).

- **Pasos por bean:**

1. **Instanciar** el objeto (llamar constructor o método de fábrica).
2. Registrar **objetos en creación** para resolver dependencias circulares (en constructor la limitación puede causar fallos; setters permiten resolver ciclos).
3. **Resolver y inyectar dependencias:** por constructor (preferido), setter o campo. *AutowiredAnnotationBeanPostProcessor* y resolución por tipo/*@Qualifier* entran aquí.
4. Aplicar Aware callbacks (si el bean implementa *BeanNameAware*, *ApplicationContextAware*, etc., se les llama para inyectar contexto/metadata).
5. Llamar *BeanPostProcessor.postProcessBeforeInitialization(...)*.



6. Ejecutar callbacks de inicialización: métodos `@PostConstruct`, `afterPropertiesSet()` (si implementa `InitializingBean`), o el `initMethod` declarado en `@Bean`.
7. Llamar `BeanPostProcessor.postProcessAfterInitialization(...)` — aquí es donde, por ejemplo, se crean proxies AOP (el bean resultante devuelto puede ser un proxy).
Por qué importa: este es el núcleo: creación, inyección y preparación del bean listo para usarse; las transformaciones en `postProcessAfterInitialization` son las que, p. ej., habilitan AOP.

7) Resolución de ambigüedades en autowiring

- Si hay múltiples candidatos por tipo, Spring usa:
 - `@Qualifier("name")` si está presente.
 - Bean marcado con `@Primary`.
 - Si hay ambigüedad sin resolución, se lanzará excepción `NoUniqueBeanDefinitionException`.**Por qué importa:** determina qué instancia concreta se inyecta en cada dependencia.

8) Publicación del evento *ContextRefreshedEvent*

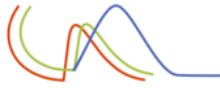
- Una vez que todos los singletons necesarios han sido creados e inicializados, el contexto publica un `ContextRefreshedEvent`.
- Los listeners registrados (`ApplicationListener`) reciben el evento y pueden ejecutar lógica dependiente del arranque (p. ej. inicializadores de app, tareas programadas, etc.).
Por qué importa: señal para ejecutar trabajo que requiere el contexto completamente inicializado.

9) Registro de *shutdown hook* (si procede) y la aplicación corre

- El contexto puede registrar un hook de JVM para invocar `close()` en `SIGTERM/JVM shutdown` (útil para entornos no-managed).
- La aplicación ahora está "lista": los beans están inicializados, los proxies AOP activos y los `@EventListener` escuchando eventos. En un servidor web, el contenedor web recibe peticiones y delega a los controladores ya inyectados.

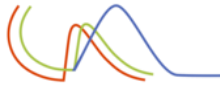
10) Cierre del contexto (shutdown) — destrucción ordenada

- Al cerrar (`context.close()` o shutdown hook), el contexto:
 1. Publica `ContextClosedEvent`.
 2. Llama al `LifecycleProcessor` para detener beans que implementen `Lifecycle/SmartLifecycle` (p. ej. contenedores de tareas).
 3. Invoca callbacks de destrucción: métodos `@PreDestroy`, `DisposableBean.destroy()`, y `destroyMethod` declarados.
 4. Destruye singleton instances y libera recursos.
Por qué importa: asegura liberación de recursos (conexiones, hilos) y permite que componentes realicen limpieza ordenada.



CASOS ESPECIALES Y NOTAS PRÁCTICAS

- **Dependencias circulares:** Spring puede resolver ciclos mediante inyección por setter (referencias tempranas a singletons), pero un ciclo por constructor directo no puede resolverse y dará error. (constructor >> setter para obligatorias).
- **AOP y proxies:** el bean final inyectado puede NO ser la instancia "cruda" sino un proxy (CGLIB o JDK). Esto ocurre durante `postProcessAfterInitialization`. Por eso `this.someAopMethod()` dentro de la misma clase no pasa por el proxy.
- **Orden de ejecución de post-processors:**
`BeanDefinitionRegistryPostProcessor` → `BeanFactoryPostProcessor` → registro de `BeanPostProcessor` → instanciación de beans. El orden es crítico y documentado en el "application startup steps".
- **@Configuration y @Bean semantics:** las clases `@Configuration` reciben tratamiento especial: por defecto Spring crea un *proxy* de la clase `@Configuration` para garantizar el comportamiento "singleton" entre métodos `@Bean` y permitir llamadas internas entre métodos de configuración.



ENLACES DE INTERÉS

https://docs.spring.io/spring-framework/reference/core.html?utm_source=chatgpt.com

https://www.jrebel.com/blog/spring-annotations-cheat-sheet?utm_source=chatgpt.com

https://springframework.guru/spring-framework-annotations/?utm_source=chatgpt.com

- **Application Startup Steps (appendix)** — describe la secuencia de registro, post-processors, y fases de refresh. [Home](#)
- **Core Technologies — IoC container** — capítulo de referencia sobre BeanFactory / ApplicationContext y conceptos base. [Home](#)
- **Container Extension Points (BeanFactoryPostProcessor / BeanPostProcessor)** — explica hooks para personalizar y extender el contenedor. [Home](#)
- **Bean lifecycle / Initialization callbacks** — detalles sobre InitializingBean, @PostConstruct, DisposableBean, etc. [Home](#)
- **AnnotationConfigApplicationContext javadoc** — cómo se crea un contexto a partir de clases anotadas y su comportamiento. [Home](#)