Spring Framework: el contenedor que arma tu aplicación

Parte 1 de 3 · 4 de noviembre de 2026 · spring · java · inyeccion-de-dependencias · fundamentos · serie-spring

> **Parte 1 de 3** de la serie «Spring, de novato a experto». Sigue con la parte 2: Spring Boot, de la primera app a una API completa y la parte 3: Spring Boot en producción.

Casi todos aprendemos Spring al revés: primero Spring Boot, que levanta una aplicación con una anotación, y después, el día que algo falla, descubrimos que debajo había un contenedor, proxies y transacciones que nadie nos explicó.

Esta serie lo hace en orden. En esta primera parte no hay Spring Boot: está Spring Framework solo, el contenedor que arma tu aplicación, con cada término definido cuando aparece. Lo único que necesitas es saber programar en Java.

Todo el código lo compilé y lo probé con Spring Framework 7.0.10: cada comportamiento que afirmo lo comprobé con un test, y las salidas que ves son las que obtuve.

▼ EN ESTE ARTÍCULO

01. Por qué existe Spring
02. Qué vas a aprender y en qué orden
03. Tu entorno de trabajo
04. Inversión de control e inyección de dependencias
05. Los patrones detrás de Spring
06. Configurar el contenedor: XML, escaneo o clases
07. El ciclo de vida de un bean
08. Propiedades: la configuración fuera del código
09. Excepciones a la manera de Spring
10. Proxies y AOP: el código que se repite en todos lados
11. Transacciones
12. Eventos: avisar sin conocer a quién
13. El mapa del ecosistema
14. Lo que solo enseña la experiencia

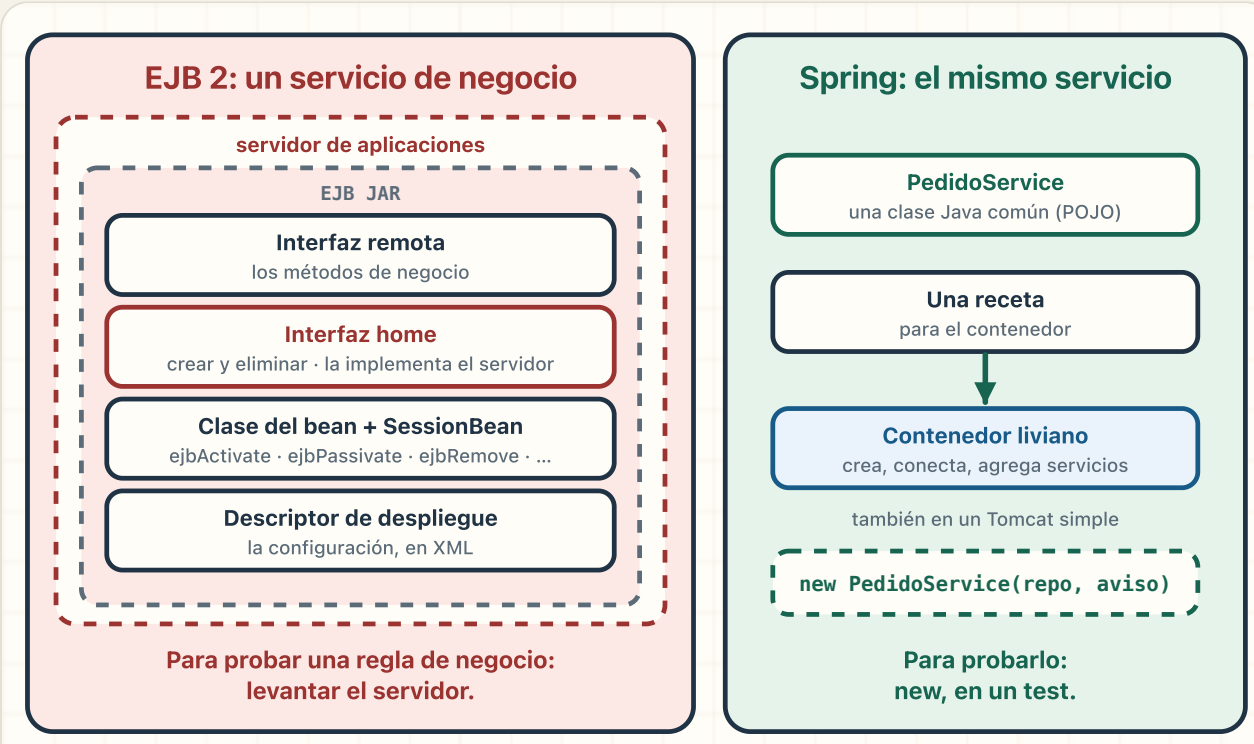
15. Referencias

Por qué existe Spring

Imagina que es 2002 y te piden la parte del sistema que registra un pedido. En Java empresarial, la respuesta estándar era J2EE (*Java 2 Platform, Enterprise Edition*), el conjunto de especificaciones de Java para aplicaciones de empresa, y la lógica de negocio iba en un **EJB** (*Enterprise JavaBean*).

Un EJB con acceso remoto no era una clase, sino varias piezas exigidas por la especificación (The J2EE 1.4 Tutorial):

- una **interfaz remota**, con los métodos de negocio;
- una **interfaz home**, para crear y eliminar el bean, que **implementaba el servidor** (Java EE 6 API, **`EJBHome`**);
- la **clase del bean**, que además implementaba **`SessionBean`**: cuatro métodos con los que el servidor le avisaba de su ciclo de vida (Java EE 6 API, **`SessionBean`**);
- un **descriptor de despliegue** en XML;
- y todo empaquetado en un *EJB JAR*, desplegado en un servidor de aplicaciones.



Un servicio de negocio antes y después de Spring: de cinco piezas atadas al servidor a una clase común que se prueba con new.

Como la interfaz home la implementaba el servidor, probar una regla de negocio exigía levantarlo.

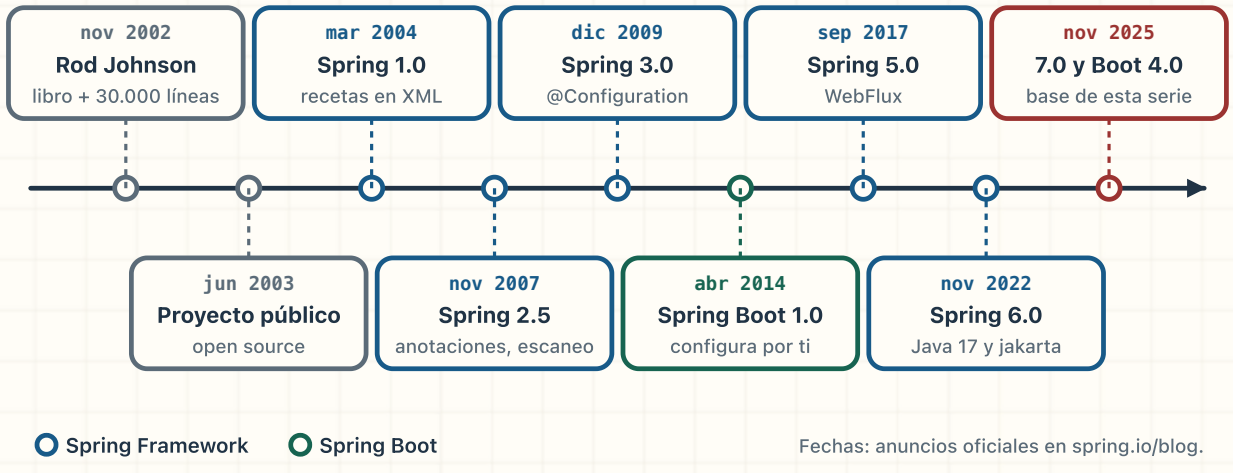
En noviembre de 2002, Rod Johnson publicó *Expert One-on-One J2EE Design and Development*, un libro acompañado de unas 30.000 líneas de código de framework. Sus lectores lo convencieron de convertir ese código en un proyecto open source, que se hizo público en junio de 2003. El nombre, Spring, marcaba un nuevo comienzo después del «invierno» del J2EE tradicional (Rod Johnson, *The Origins of a Project and a Name*).

La idea cabe en una frase. Tu lógica de negocio va en **POJOs** (*Plain Old Java Objects*): clases Java comunes, que no heredan ni implementan nada del framework. Y un **contenedor** liviano las crea, las conecta y les agrega desde afuera lo que antes exigía el servidor, como las transacciones.

Spring 1.0 salió el 24 de marzo de 2004 con eso: un contenedor con inyección por setter y por constructor, AOP, una capa sobre JDBC y transacciones declarativas "aplicables a cualquier POJO", en un Tomcat común (Spring Framework 1.0 Final Released). La referencia actual lo resume: Spring nació «como respuesta a la complejidad de las primeras especificaciones J2EE» (Spring Framework Reference, 7.0).

Por qué sigue vigente

Cada generación de Spring mantuvo el contenedor y los POJOs, y cambió cómo se escriben las recetas y qué hay alrededor:



Más de veinte años con la misma idea: cambia la forma de escribir las recetas, no el contenedor.

Las fechas salen de los anuncios oficiales del blog de Spring. Spring Framework 7.0, del 13 de noviembre de 2025, es la base de Spring Boot 4.0, que salió una semana después.

> **Nota:** En este sitio, el backend es una aplicación Spring Boot 4.1.2, construida sobre el mismo Spring Framework 7 de esta guía.

Lo que ganas y lo que pagas

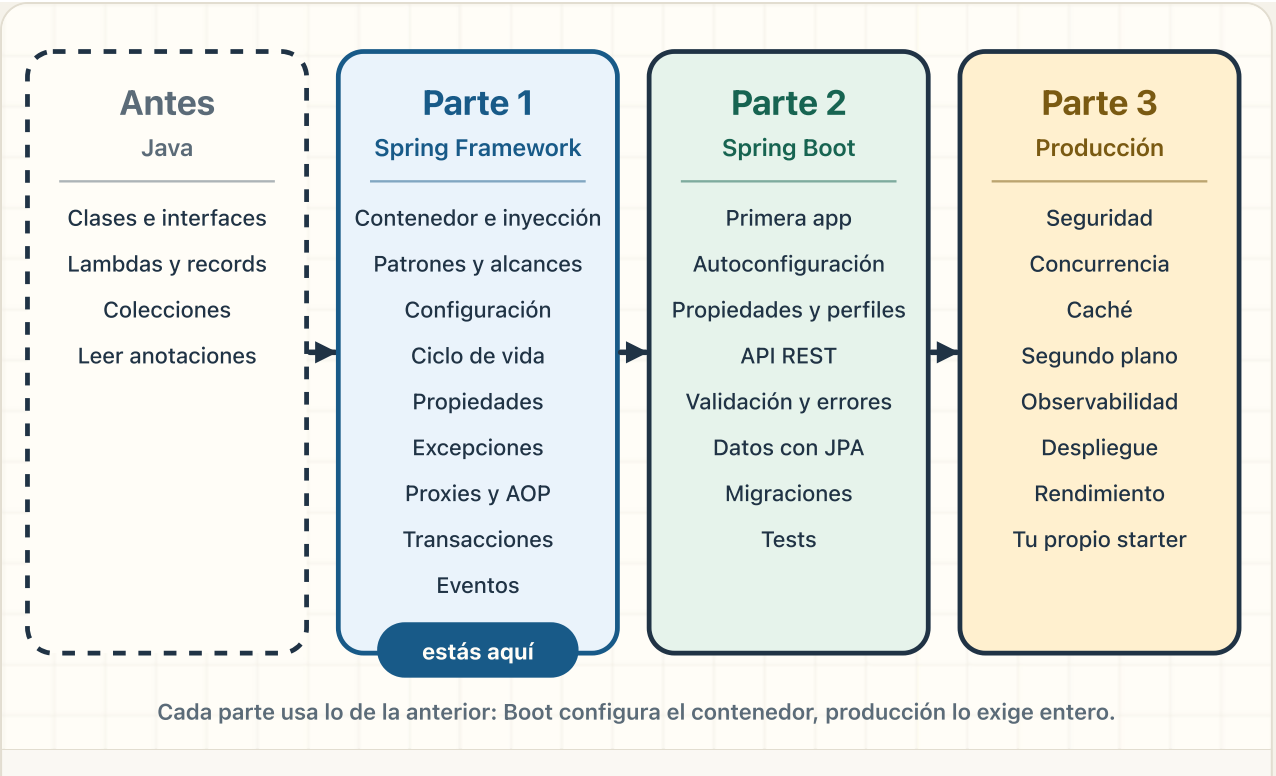
Lo que ganas:

- **Tu código no depende del framework.** Una clase que recibe sus dependencias en el constructor se prueba con `new`, sin servidor y sin Spring.
- **Lo transversal se declara.** Una transacción es una anotación sobre un método, no un bloque de abrir, confirmar y deshacer en cada uno.
- **Estándares sueltos, no la plataforma entera.** Spring integra especificaciones elegidas, como Servlet, JPA, Bean Validation o JMS (Spring Framework Reference, 7.0).

Lo que pagas:

- **Una curva de aprendizaje.** Contenedor, alcances, proxies, propagación: muchos conceptos antes de poder explicar por qué algo falla.
- **La "magia".** Parte del comportamiento está en objetos que el contenedor pone delante del tuyo, y cuando falla, falla de formas raras. Lo reproduzco con tests en la sección de proxies.
- **El arranque.** El contenedor hace su trabajo al arrancar, para que los errores aparezcan temprano. Eso cuesta tiempo, y la Parte 3 lo mide.
- **Las migraciones mayores.** Spring 6.0 dejó los paquetes `javax` por `jakarta` y subió el mínimo a Java 17 (Spring Framework 6.0 goes GA). Cambiar de generación es un trabajo planificado.

Qué vas a aprender y en qué orden



Primero el contenedor, después Boot que lo configura, al final lo que pide producción.

Primero el contenedor sin Spring Boot; después Boot, que no reemplaza nada de la Parte 1 sino que lo configura por ti; al final, lo que pide producción.

Antes de empezar, conviene que manejes:

- **Java orientado a objetos:** clases, interfaces e implementaciones, y leer una anotación como `@Override`.
- **Lambdas y records.** Si te faltan, están en la guía de Java por sus LTS.
- **Colecciones**, y la diferencia entre una interfaz y su implementación, que es la base de la inyección de dependencias: Colecciones en Java: List no es ArrayList.

Tu entorno de trabajo

Para seguir la guía vas a armar tu propio proyecto con el `pom.xml` y las clases que muestro.

Un JDK 21

Spring Framework 7 pide Java 17 como mínimo (Spring Framework 7.0 General Availability); en la serie uso Java 21, la versión del backend de este sitio. Para comprobar cuál tienes:

```
java -version
```

Si la primera línea no dice 21 o más, instala un JDK 21 de cualquier proveedor, por ejemplo Eclipse Temurin.

Un IDE

- **IntelliJ IDEA**: desde la 2025.3 es un solo producto, sin Community ni Ultimate, con las funciones básicas gratis (JetBrains, *IntelliJ IDEA as a unified product*). Trae Maven incluido.
- **Visual Studio Code** con el **Extension Pack for Java** de Microsoft.

Un proyecto Maven

Maven descarga las librerías del proyecto (las **dependencias**), compila y corre los tests. Crea un proyecto Maven nuevo desde el IDE. No hace falta instalar Maven si el proyecto trae el **Maven Wrapper**: dos scripts que bajan la versión fijada en **`.mvn/wrapper/maven-wrapper.properties`** y la usan desde ahí (Apache Maven Wrapper). Se genera una sola vez con **`mvn wrapper:wrapper`**, desde el IDE, y después:

```
./mvnw test        # macOS y Linux
mvnw.cmd test      # Windows
```

Spring está dividido en módulos (**`spring-context`**, **`spring-jdbc`**, **`spring-tx`**…) que tienen que ir en la misma versión. Para eso se usa un **BOM** (*Bill of Materials*): una lista de versiones que se importa una vez en el **`pom.xml`**.

pom.xml

```
<dependencyManagement>
    <dependencies>
        <!-- Un BOM fija la versión de TODOS los módulos de Spring a la vez:
             abajo, las dependencias van sin <version>. -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>7.0.10</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit-bom</artifactId>
            <version>6.0.3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
```

```
    </dependencies>
</dependencyManagement>
```

Con el BOM, las dependencias de Spring van sin **`<version>`**. Para empezar alcanza con una:

pom.xml

```
<!-- El contenedor: beans, ApplicationContext, anotaciones. -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
</dependency>
```

La 7.0.10 es la versión que trae Spring Boot 4.1.2, el del backend de este sitio. El segundo BOM fija la de JUnit; para los tests agrega **`junit-jupiter`** y **`assertj-core`** con **`<scope>test</scope>`**. Las dependencias de la base de datos llegan en la sección de excepciones.

Tu primera app con el contenedor

PrimeraApp.java

```
@Configuration
public class PrimeraApp {

    record Saludo(String texto) {}

    @Bean
    Saludo saludo() {
        return new Saludo("Hola, Spring");
    }

    public static void main(String[] args) {
        try (var contexto = new AnnotationConfigApplicationContext(PrimeraApp.class)) {
            System.out.println(contexto.getBean(Saludo.class).texto());
        }
    }
}
```

Las piezas las desarmo en la sección siguiente. Ejecuta el **`main`** desde el IDE: si en la consola aparece **`Hola, Spring`**, tu entorno está listo.

Inversión de control e inyección de dependencias

Hay una pregunta que conviene hacerse con cualquier clase que escribas: ¿quién decide con qué objetos trabaja? Parece una pregunta de filosofía. Es la pregunta que separa el código que se deja cambiar y probar del que no, y es la razón por la que existe Spring.

Vamos a responderla con la tienda de pedidos que nos acompaña en toda la serie. Cuando alguien compra, **`PedidoService`** guarda el pedido y le avisa al cliente por correo. La primera versión que casi todos escribimos es esta:

PedidoServiceAcoplado.java

```
public class PedidoServiceAcoplado {

    private final RepositorioEnMemoria repositorio = new RepositorioEnMemoria();
    private final NotificadorCorreo notificador = new NotificadorCorreo();

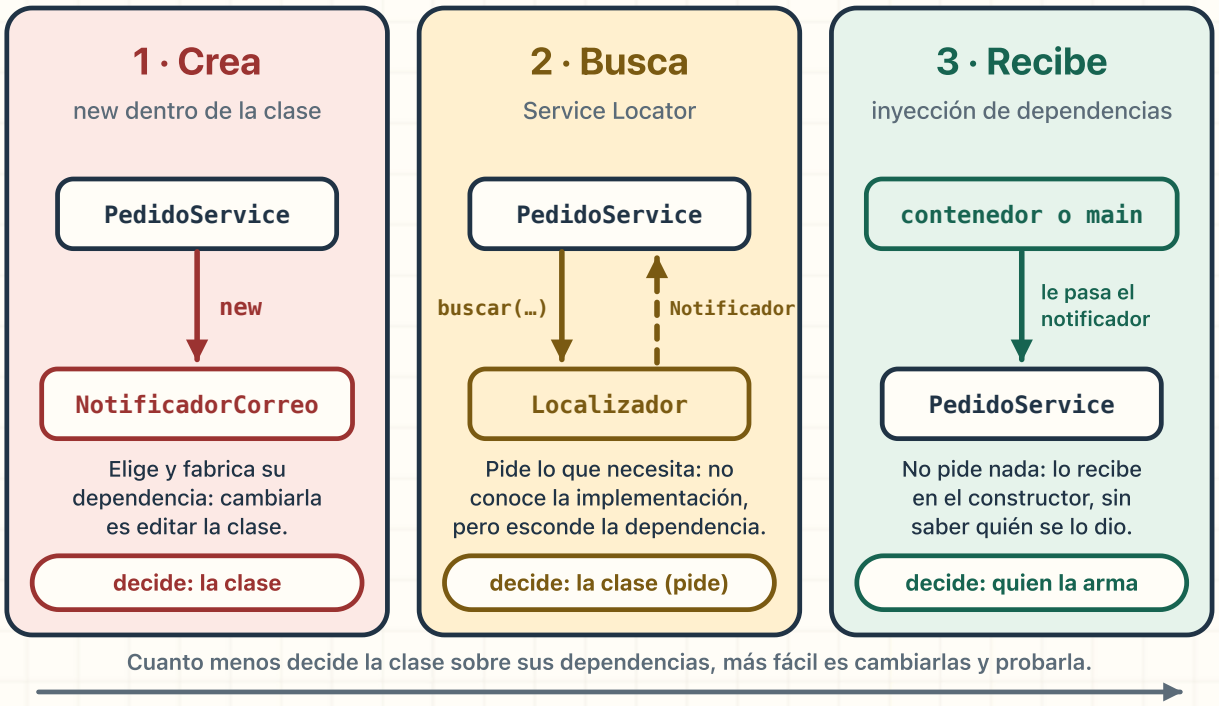
    public void registrar(Pedido pedido) {
        repositorio.guardar(pedido);
        notificador.enviar(pedido.correoCliente(), "Recibimos tu pedido " +
pedido.numero());
    }
}
```

Funciona. Y tiene un problema que no se ve hasta que intentas probarla: **la clase decidió por su cuenta** que avisa por correo, con **`NotificadorCorreo`**, y lo fabricó ella misma con **`new`**. No hay forma de llamar a **`registrar()`** sin que se ejecute el notificador de verdad. Lo comprobé con un test: probar el servicio manda el correo, siempre.

Si mañana quieres avisar por SMS, tienes que abrir esta clase y editarla. Si quieres probarla sin mandar correos, no puedes. El **`new`** te ató a una implementación concreta.

Tres formas de conseguir una dependencia

Una **dependencia** es cualquier objeto que una clase necesita para hacer su trabajo. A **`PedidoService`** le hacen falta dos: dónde guardar y cómo avisar. Hay tres formas de conseguirlas, y lo que cambia entre ellas es **quién decide**:



Lo que cambia entre las tres no es el código del método, es quién elige el notificador.

La primera ya la vimos: la clase **crea** lo que necesita. La segunda es un patrón con nombre propio, el *Service Locator*: un registro global donde cada clase va a **buscar** lo que necesita.

Localizador.java

```java
public final class Localizador {

    private static final Map<Class<?>, Object> servicios = new HashMap<>();

    public static <T> void registrar(Class<T> tipo, T servicio) {
        servicios.put(tipo, servicio);
    }

    public static <T> T buscar(Class<T> tipo) {
        return tipo.cast(servicios.get(tipo));
    }
}
```

PedidoServiceConLocalizador.java

```java
public class PedidoServiceConLocalizador {

    public void registrar(Pedido pedido) {
        Localizador.buscar(RepositorioPedidos.class).guardar(pedido);
        Localizador.buscar(Notificador.class)
```

```
                .enviar(pedido.correoCliente(), "Recibimos tu pedido " +
pedido.numero());
    }
}
```

Es una mejora: el servicio ya no sabe que el notificador manda correos, solo pide "un **`Notificador`**", y en un test puedes registrar uno falso. Pero el constructor está vacío: desde afuera nadie sabe que esta clase necesita un repositorio y un notificador. Si te olvidas de registrar uno, te enteras recién al ejecutar, con un **`NullPointerException`** en medio de **`registrar()`**. La dependencia sigue ahí, solo que escondida.

La tercera forma es la que importa. La clase **recibe** lo que necesita en el constructor y no decide nada:

ANTES

PedidoServiceAcoplado.java

```
public class PedidoServiceAcoplado {

    private final RepositorioEnMemoria
repositorio = new RepositorioEnMemoria();
    private final NotificadorCorreo
notificador = new NotificadorCorreo();

    public void registrar(Pedido pedido)
{
        repositorio.guardar(pedido);

notificador.enviar(pedido.correoCliente()
, "Recibimos tu pedido " +
pedido.numero());
    }
}
```

DESPUÉS

PedidoService.java

```
public class PedidoService {

    private final RepositorioPedidos
repositorio;
    private final Notificador
notificador;

    public
PedidoService(RepositorioPedidos
repositorio, Notificador notificador) {
        this.repositorio = repositorio;
        this.notificador = notificador;
    }

    public void registrar(Pedido pedido)
{
        repositorio.guardar(pedido);

notificador.enviar(pedido.correoCliente()
, "Recibimos tu pedido " +
pedido.numero());
    }
}
```

Ahora **`PedidoService`** trabaja con un **`RepositorioPedidos`** y un **`Notificador`** cualquiera: interfaces, no clases concretas. Y el constructor dice en voz alta lo que la clase necesita para existir.

Qué es la inversión de control

En las dos primeras versiones, la clase tiene el control: elige su notificador, sea fabricándolo o pidiéndolo. En la tercera, ese control se **invirtió**: lo tiene quien arma la clase. De ahí el nombre.

La **inversión de control** (IoC, por *Inversion of Control*) es un principio más general que este ejemplo: en lugar de que tu código llame al framework, el framework llama a tu código. Se la conoce también como el principio de Hollywood: "no nos llames, nosotros te llamamos" (Martin Fowler, *InversionOfControl*). Ya la usaste aunque no le pusieras nombre: cuando escribes un método **`@Test`**, nunca escribes el **`main`** que lo ejecuta. JUnit lo encuentra y lo llama cuando corresponde. Esa es justamente la diferencia entre una biblioteca, a la que tú llamas, y un framework, que te llama a ti.

La **inyección de dependencias** (DI, por *Dependency Injection*) es una forma concreta de aplicar ese principio a las dependencias: la clase las declara en su constructor y alguien de afuera se las **inyecta** al crearla. La referencia de Spring lo define así: los objetos declaran sus dependencias solo como argumentos del constructor, de un método fábrica o propiedades que se asignan después de crearlos, y el contenedor se las inyecta, que es lo inverso de que el propio objeto controle su creación o su búsqueda "con **`new`** o con un mecanismo como el patrón Service Locator" (Spring Framework Reference, 7.0).

> **El error típico:** pensar que el Service Locator es una forma de inyección de dependencias porque "ya no hay **`new`**". No lo es: la clase sigue **pidiendo**. Con inyección no pide nada, y por eso tampoco depende del localizador.

Inyección de dependencias sin Spring

La inyección de dependencias es un patrón, no una librería. Alguien tiene que hacer los **`new`**, y en una app sin framework ese alguien es el **`main`**:

AppManual.java

```
RepositorioPedidos repositorio = new RepositorioEnMemoria();
Notificador notificador = new NotificadorCorreo();
PedidoService pedidos = new PedidoService(repositorio, notificador);

pedidos.registrar(new Pedido(42, "ana@example.com"));
```

Y el test, que antes era imposible, ahora es trivial. Armas el servicio con un notificador falso que, en vez de mandar correos, anota lo que se le pidió mandar:

PedidoServiceTest.java

```
@Test
void registrarAvisaAlCliente() {
    var notificador = new NotificadorFalso();
    var pedidos = new PedidoService(new RepositorioEnMemoria(), notificador);

    pedidos.registrar(new Pedido(42, "ana@example.com"));

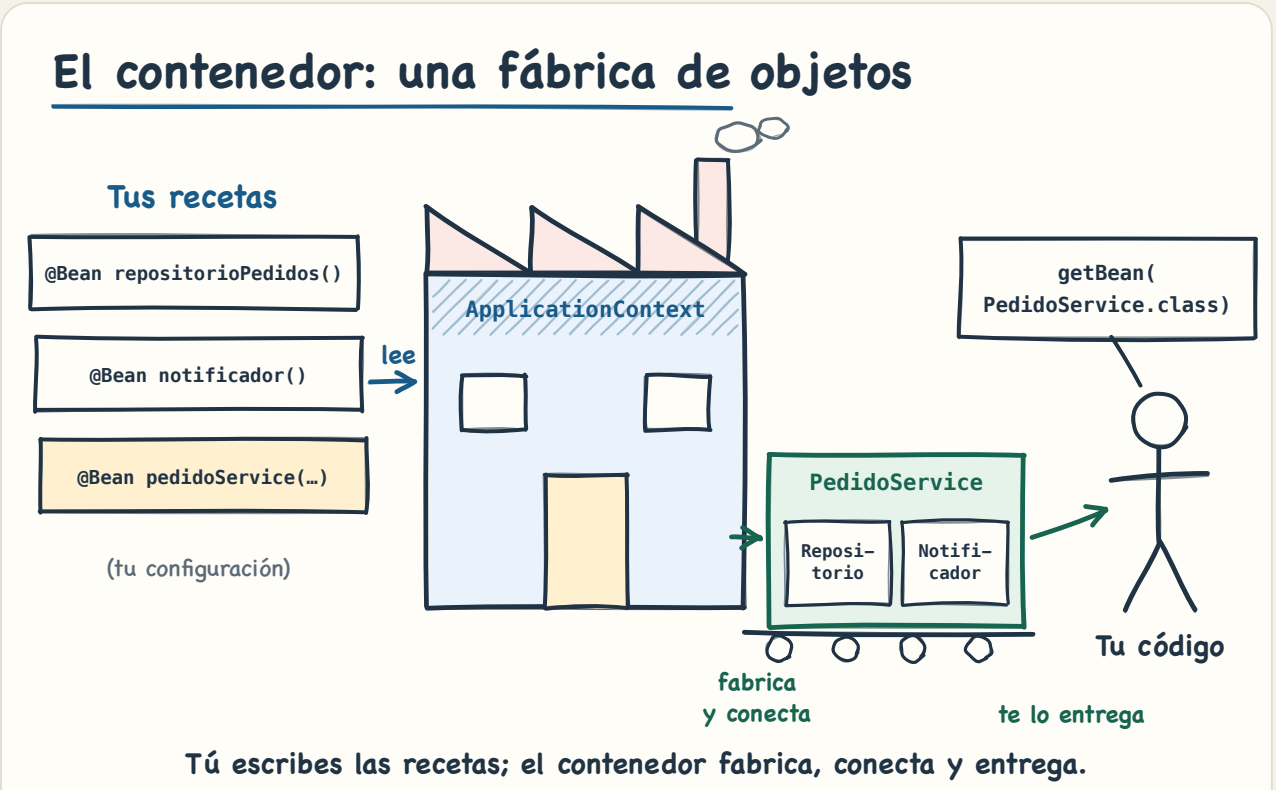
    assertThat(notificador.enviados())
            .containsExactly("ana@example.com: Recibimos tu pedido 42");
```

```
}
```

Sin Spring, sin servidor de correo, en milisegundos.

Entonces, ¿para qué queremos Spring? Con tres clases, ese **`main`** son tres líneas. En una aplicación real son cientos de clases, y armarlas a mano se vuelve un programa en sí mismo: hay que crearlas en el orden correcto, decidir cuáles se comparten y cuáles no, cerrar las conexiones al apagar, cambiar las piezas según el ambiente. Ese trabajo es exactamente el que hace el contenedor de Spring.

El contenedor: una fábrica que sigue tus recetas



Tú escribes las recetas; el contenedor fabrica, conecta y entrega.

Tú escribes las recetas; el contenedor fabrica cada pieza, las conecta y te entrega el objeto listo.

El **contenedor** de Spring es el objeto que crea, conecta y administra los objetos de tu aplicación. En el código lo vas a ver como **`ApplicationContext`**. Cada objeto que el contenedor administra se llama **bean**: un objeto Java común, solo que lo creó el contenedor y no tú.

Para que el contenedor sepa qué crear, le das recetas. Son las mismas tres líneas del **`main`** manual, cada una en su método:

ConfiguracionPedidos.java

```
@Configuration
```

```
public class ConfiguracionPedidos {

    @Bean
    RepositorioPedidos repositorioPedidos() {
        return new RepositorioEnMemoria();
    }

    @Bean
    Notificador notificador() {
        return new NotificadorCorreo();
    }

    @Bean
    PedidoService pedidoService(RepositorioPedidos repositorio, Notificador notificador)
{
        return new PedidoService(repositorio, notificador);
    }
}
```

`@Configuration` marca la clase como un recetario y cada **`@Bean`** es una receta: lo que devuelve el método es el bean. Si una receta tiene parámetros, como **`pedidoService(...)`**, el contenedor busca entre sus beans uno de cada tipo y se lo pasa. Esa es la inyección.

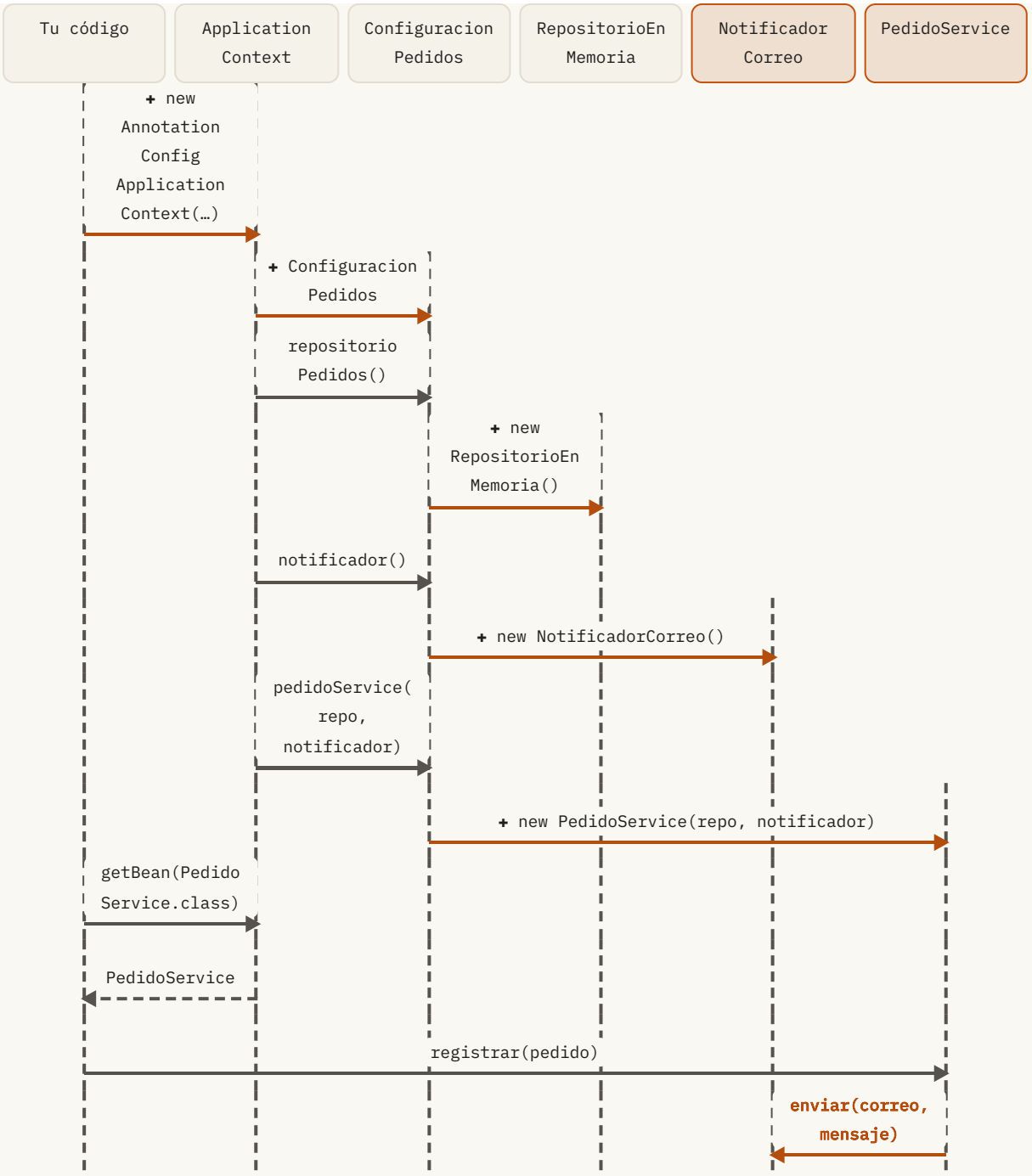
Y así se usa:

AppConContenedor.java

```
try (var contexto = new AnnotationConfigApplicationContext(ConfiguracionPedidos.class))
{
    PedidoService pedidos = contexto.getBean(PedidoService.class);
    pedidos.registrar(new Pedido(42, "ana@example.com"));
}
```

Esto es lo que pasa, mensaje por mensaje, entre la primera línea y el correo:

Del new AnnotationConfigApplicationContext al correo enviado



Tu código crea el contenedor y le pasa la clase de configuración. Todavía no existe ningún bean.

El recetario también es un bean: el primero de tus beans que crea el contenedor es la propia ConfiguracionPedidos.

Ahora sigue las recetas. La primera no tiene parámetros: la llama tal cual.

El método hace el new y devuelve el objeto. El contenedor lo guarda como el bean repositorioPedidos.

Segunda receta, mismo camino.

Ya hay dos beans: un repositorio y un notificador.

La tercera receta pide un RepositorioPedidos y un Notificador. El contenedor busca por tipo entre sus beans y le pasa los dos que acaba de crear. Esto es la inyección.

PedidoService nace con sus dependencias adentro. Todo esto pasó al arrancar, antes de que tu código pidiera nada.

Recién ahora tu código pide el servicio. No se crea nada: el contenedor ya lo tenía armado.

Te lo devuelve listo para usar.

Desde acá es Java común: llamas a un método de un objeto.

Y el servicio usa el notificador que recibió, sin saber quién lo eligió ni quién lo creó.

Hay dos detalles de esa secuencia que no son obvios. El primero es que el contenedor crea los beans singleton **al arrancar**, no cuando los pides. Lo hace a propósito: si una receta está mal, prefieres enterarte al levantar la aplicación y no a las tres de la mañana, cuando alguien pide ese bean por primera vez (Spring Framework Reference, 7.0). Lo comprobé con un test.

El segundo detalle es lo que **no** aparece: **`PedidoService`** no importa nada de Spring. Es la misma clase que armamos a mano en el **`main`** y en el test. Por eso cambiar el armado no toca la clase. Basta con otro recetario, por ejemplo uno de prueba con el notificador falso:

ContenedorTest.java

```
@Configuration
static class ConfiguracionDePrueba {

    @Bean
    RepositorioPedidos repositorioPedidos() {
        return new RepositorioEnMemoria();
    }

    @Bean
    NotificadorFalso notificador() {
        return new NotificadorFalso();
    }

    @Bean
    PedidoService pedidoService(RepositorioPedidos repositorio, Notificador notificador)
{
        return new PedidoService(repositorio, notificador);
    }
}
```

Por constructor, no por campo

Hay otra forma de pedirle dependencias a Spring que vas a encontrar en muchísimo código, porque es más corta: anotar los campos con **`@Autowired`**.

PedidoServiceConCampo.java

```
@Component
public class PedidoServiceConCampo {

    @Autowired
```

```
    private RepositorioPedidos repositorio;

    @Autowired
    private Notificador notificador;

    public void registrar(Pedido pedido) {
        repositorio.guardar(pedido);
        notificador.enviar(pedido.correoCliente(), "Recibimos tu pedido " +
pedido.numero());
    }
}
```

Dentro del contenedor funciona: Spring crea el objeto y después rellena los campos por reflexión. Fuera de él, un test que crea la clase con **`new`** se encuentra con los campos en **`null`**:

PedidoServiceConCampoTest.java

```
@Test
void fueraDelContenedorLosCamposQuedanEnNull() {
    var servicio = new PedidoServiceConCampo();

    assertThatThrownBy(() -> servicio.registrar(new Pedido(42, "ana@example.com")))
            .isInstanceOf(NullPointerException.class);
}
```

Con la inyección por constructor eso no puede pasar: el compilador no te deja crear el objeto sin sus dependencias. Es lo que recomienda el equipo de Spring, por tres razones (Spring Framework Reference, 7.0):

- **Los campos pueden ser `final`**: el objeto es inmutable, nadie le cambia el notificador a mitad de camino.
- **Las dependencias obligatorias nunca son `null`**: si falta una, el objeto no se crea.
- **El objeto sale completo**: quien lo recibe nunca ve uno a medio armar.

Además, si una clase que el contenedor crea por su cuenta tiene un solo constructor, no hace falta anotarlo con **`@Autowired`**: Spring lo usa solo. **`PedidoService`** ni siquiera necesita eso, porque lo crea tu receta con **`new`**; la regla la vas a ver en **`ServicioEscaneado`**, más abajo.

> **Lo que debes llevarte:**
>
> - **Inversión de control**: el framework llama a tu código, no al revés. Con las dependencias, significa que la clase deja de elegirlas.
> - **Inyección de dependencias**: la clase declara lo que necesita en el constructor y alguien de afuera se lo da. El Service Locator no es eso: ahí la clase sigue pidiendo.

- **Contenedor y bean:** el `ApplicationContext` crea, conecta y administra tus objetos; cada uno de ellos es un bean, y por defecto los singleton se crean al arrancar.
- **Por constructor:** campos `final`, nada en `null`, objetos completos, y clases que se prueban sin Spring.

– Para profundizar

el nombre "inyección de dependencias" lo acordaron en 2004 Martin Fowler y los autores de contenedores IoC, justamente porque "inversión de control" le parecía un término demasiado genérico (Martin Fowler, *Inversion of Control Containers and the Dependency Injection pattern*), donde también lo compara con el Service Locator.

El contenedor resuelve los parámetros de un constructor o de un método `@Bean` por tipo. Cuando hay más de un candidato del mismo tipo hace falta desambiguar, y eso lo vemos más adelante, con `@Qualifier` y `@Primary`.

Un ciclo por constructor, en el que A necesita a B y B necesita a A, no tiene solución: no se puede crear ninguno de los dos primero. El contenedor lo detecta al arrancar y lanza `BeanCurrentlyInCreationException`. En Spring sin Boot, el mismo ciclo por campo o por setter arranca, y por eso esconde un problema de diseño que el constructor te obliga a ver.

La inyección por setter sigue teniendo su lugar, pero solo para dependencias **opcionales** que tienen un valor por defecto razonable dentro de la clase.

Los patrones detrás de Spring

Spring usa patrones clásicos, los del libro *Design Patterns* conocido como "GoF" (*Gang of Four*). Ojo: el patrón del libro y la versión de Spring a veces se llaman igual y no son lo mismo.

Factory: el contenedor es una fábrica

Una **fábrica** es un objeto cuyo trabajo es crear otros objetos, para que quien los usa no tenga que saber cómo se construyen. Spring tiene dos niveles:

- `BeanFactory`, la interfaz básica: crea beans y los conecta.
- `ApplicationContext`, que la extiende y suma AOP, eventos, mensajes internacionalizados y contextos web (Spring Framework Reference, 7.0).

Un **BeanFactory** simple no detecta los **BeanPostProcessor**, las piezas que procesan las anotaciones y crean los proxies (Spring Framework Reference, 7.0). Lo comprobé con un test: un **DefaultListableBeanFactory** crea el bean recién cuando se lo pides y no ejecuta su **@PostConstruct**; un **ApplicationContext** lo crea al arrancar y sí lo ejecuta. Por eso la recomendación oficial es usar un **ApplicationContext** salvo que tengas una buena razón.

Cada método **@Bean** es también una fábrica en pequeño, un **factory method**: Spring 3.0 los presentó como "métodos fábrica anotados" (Spring Framework 3.0 goes GA). Y cuando crear un bean requiere lógica propia existe **FactoryBean<T>**, un bean cuyo trabajo es fabricar otro:

FabricaNotificador.java

```
public class FabricaNotificador implements FactoryBean<Notificador> {

    private final String canal;

    public FabricaNotificador(String canal) {
        this.canal = canal;
    }

    @Override
    public Notificador getObject() {
        return canal.equals("sms") ? new NotificadorSms() : new NotificadorCorreo();
    }

    @Override
    public Class<?> getObjectType() {
        return Notificador.class;
    }
}
```

Al pedirlo, el contenedor no te da la fábrica sino lo que fabrica. Para obtener la fábrica misma, se antepone **&** al nombre (Spring Framework Reference, 7.0):

FabricasTest.java

```
@Configuration
static class ConFabrica {
    @Bean
    FabricaNotificador notificador() {
        return new FabricaNotificador("sms");
    }
}

@Test
void elContenedorEntregaElProductoDeLaFabrica() {
    try (var contexto = new AnnotationConfigApplicationContext(ConFabrica.class)) {
```

```
assertThat(contexto.getBean(Notificador.class)).isInstanceOf(NotificadorSms.class);
        assertThat(contexto.getBean("notificador")).isInstanceOf(NotificadorSms.class);

assertThat(contexto.getBean("&notificador")).isInstanceOf(FabricaNotificador.class);
    }
}
```

Singleton: uno por contenedor, no uno por clase

El **Singleton** del libro es una clase que garantiza ella misma que existe una sola instancia: esconde su constructor y guarda la instancia en un campo estático. En Spring, la clase es común, y que haya una sola lo decide el contenedor, porque **singleton** es el alcance por defecto de todo bean:

ANTES

CatalogoGlobal.java

```
public final class CatalogoGlobal {

    private static final CatalogoGlobal
INSTANCIA = new CatalogoGlobal();

    private CatalogoGlobal() {}

    public static CatalogoGlobal
instancia() {
        return INSTANCIA;
    }
}
```

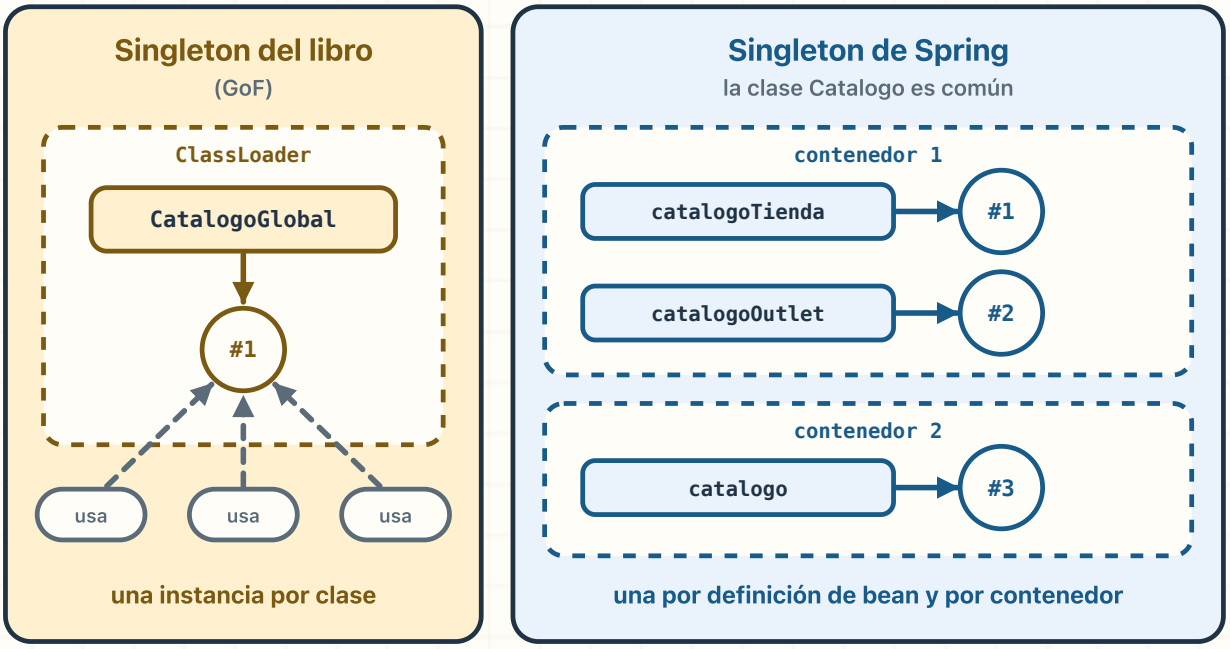
DESPUÉS

ConfiguracionCatalogo.java

```
@Configuration
public class ConfiguracionCatalogo {

    @Bean
    Catalogo catalogo() {
        return new Catalogo();
    }
}
```

Cambia también qué significa "uno solo". El del libro es uno por *ClassLoader* (el componente de la JVM que carga las clases): uno por clase, en la práctica. El de Spring es uno **por contenedor y por definición de bean** (Spring Framework Reference, 7.0).



El Singleton del libro es uno por clase; el de Spring, uno por receta y por contenedor.

AlcancesTest.java

```
@Configuration
static class DosCatalogos {

    @Bean
    Catalogo catalogoTienda() {
        return new Catalogo();
    }

    @Bean
    Catalogo catalogoOutlet() {
        return new Catalogo();
    }
}

@Test
void elSingletonDeSpringEsUnoPorDefinicionDeBean() {
    try (var contexto = new AnnotationConfigApplicationContext(DosCatalogos.class)) {
        var tienda = contexto.getBean("catalogoTienda", Catalogo.class);
        var outlet = contexto.getBean("catalogoOutlet", Catalogo.class);

        assertThat(tienda).isNotSameAs(outlet);
        assertThat(contexto.getBean("catalogoTienda")).isSameAs(tienda);
    }
}

@Test
```

```
void elSingletonDeSpringEsUnoPorContenedor() {
    try (var uno = new AnnotationConfigApplicationContext(ConfiguracionCatalogo.class);
            var otro = new
AnnotationConfigApplicationContext(ConfiguracionCatalogo.class)) {

assertThat(uno.getBean(Catalogo.class)).isNotSameAs(otro.getBean(Catalogo.class));
    }
}
```

Lo que no cambia es que el singleton se **comparte**: todo el que lo pida recibe el mismo objeto, y en una aplicación web eso son todos los requests a la vez. Por eso la regla de la referencia es singleton para los beans sin estado y prototype para los que tienen estado (Spring Framework Reference, 7.0).

> **El error típico:** guardar en un campo de un singleton algo que es de un cliente. Esta calculadora recuerda el descuento del último que la usó:
>
> CalculadoraConDescuento.java
>
> ```
> public class CalculadoraConDescuento {
>
>     private int descuento;
>
>     public void aplicarDescuento(int porcentaje) {
>         this.descuento = porcentaje;
>     }
>
>     public int total(int subtotal) {
>         return subtotal - subtotal * descuento / 100;
>     }
> }
> ```
>
> Ana y Luis reciben el mismo objeto; Luis aplica su descuento después de Ana, y Ana paga sin el suyo:
>
> AlcancesTest.java
>
> ```
> @Test
> void elDescuentoDeAnaSeLoLlevaLuis() {
>     try (var contexto = new
> AnnotationConfigApplicationContext(ConCalculadora.class)) {
>         var paraAna = contexto.getBean(CalculadoraConDescuento.class);
>         var paraLuis = contexto.getBean(CalculadoraConDescuento.class);
>
>         paraAna.aplicarDescuento(10);
>         paraLuis.aplicarDescuento(0);
>
>         assertThat(paraAna.total(1_000)).isEqualTo(1_000);
> ```

```
    }
}
```

El arreglo es no tener estado: el descuento entra como parámetro de **`total`**.

Prototype: uno nuevo cada vez

Un borrador de pedido, la lista que un cliente arma antes de confirmar, tiene que ser uno por cliente. Para eso está el alcance **prototype**: cada vez que pides el bean, el contenedor ejecuta la receta y te da uno nuevo.

ConfiguracionAlcances.java

```
@Configuration
public class ConfiguracionAlcances {

    @Bean
    CalculadoraEnvio calculadoraEnvio() {
        return new CalculadoraEnvio();
    }

    @Bean
    @Scope("prototype")
    BorradorPedido borradorPedido() {
        return new BorradorPedido();
    }
}
```

Tres pedidos a un singleton y a un prototype



Al arrancar, el contenedor crea el singleton CalculadoraEnvio y lo guarda. Del prototype solo tiene la receta: todavía no existe ningún BorradorPedido.

El cliente A pide la calculadora. No se crea nada: el contenedor le entrega la que ya tenía.

El cliente B pide lo mismo y recibe el mismo objeto, la #1.

Y el cliente C también. Tres pedidos, un solo objeto: por eso un singleton no puede guardar nada de un cliente.

Ahora el cliente A pide un BorradorPedido. Con prototype, el contenedor ejecuta la receta en este momento y le entrega un objeto nuevo.

El cliente B pide otro y recibe otro: el #2. Lo que A agregue a su borrador, B no lo ve.

El cliente C recibe el #3. Fíjate en que el contenedor no guardó ninguno: los entregó y se olvidó de ellos.

Al cerrar el contenedor, Spring destruye el singleton que administra. A los borradores no los toca: limpiarlos es trabajo de quien los pidió.

En el test, **`isSameAs`** comprueba que dos referencias apuntan al mismo objeto, no que sean iguales:

AlcancesTest.java

```
@Test
void elSingletonEsUnoParaTodosElPrototypeUnoPorPedido() {
    try (var contexto = new
AnnotationConfigApplicationContext(ConfiguracionAlcances.class)) {
        var calculadora1 = contexto.getBean(CalculadoraEnvio.class);
        var calculadora2 = contexto.getBean(CalculadoraEnvio.class);
        var calculadora3 = contexto.getBean(CalculadoraEnvio.class);
        assertThat(calculadora1).isSameAs(calculadora2).isSameAs(calculadora3);

        var borrador1 = contexto.getBean(BorradorPedido.class);
        var borrador2 = contexto.getBean(BorradorPedido.class);
        var borrador3 = contexto.getBean(BorradorPedido.class);
        assertThat(borrador1).isNotSameAs(borrador2);
        assertThat(borrador2).isNotSameAs(borrador3);
    }
}
```

El último paso también lo comprobé con un test: Spring entrega un prototype y no lo vuelve a ver, así que **no llama a sus métodos de destrucción** (Spring Framework Reference, 7.0).

La trampa del prototype dentro de un singleton

Esta caja registradora, un singleton, recibe un borrador en su constructor:

CajaConBorradorFijo.java

```
@Component
public class CajaConBorradorFijo {

    private final BorradorPedido borrador;

    public CajaConBorradorFijo(BorradorPedido borrador) {
        this.borrador = borrador;
    }

    public BorradorPedido nuevoPedido() {
        return borrador;
    }
}
```

El contenedor resuelve las dependencias de un bean **una vez, al crearlo** (Spring Framework Reference, 7.0). La caja se crea una sola vez y recibe un solo borrador; que sea prototype da igual. Luis encuentra el libro de Ana en su pedido:

AlcancesTest.java

```
@Test
void elSingletonRecibioUnSoloBorradorYLoComparteConTodos() {
    try (var contexto = new AnnotationConfigApplicationContext(ConCajas.class)) {
        var caja = contexto.getBean(CajaConBorradorFijo.class);

        caja.nuevoPedido().agregar("libro");
        var deLuis = caja.nuevoPedido();

        assertThat(deLuis.productos()).containsExactly("libro");
    }
}
```

El arreglo es recibir a alguien que sepa pedir un borrador nuevo cada vez: **`ObjectProvider<T>`**, que en cada **`getObject()`** le pide el bean al contenedor. Con **`Caja`**, el pedido de Luis empieza vacío.

ANTES

CajaConBorradorFijo.java

```
@Component
public class CajaConBorradorFijo {

    private final BorradorPedido
borrador;

    public
CajaConBorradorFijo(BorradorPedido
```

DESPUÉS

Caja.java

```
@Component
public class Caja {

    private final
ObjectProvider<BorradorPedido>
borradores;
```

```
borrador) {
        this.borrador = borrador;
    }

    public BorradorPedido nuevoPedido() {
        return borrador;
    }
}
```

```
    public
Caja(ObjectProvider<BorradorPedido>
borradores) {
        this.borradores = borradores;
    }

    public BorradorPedido nuevoPedido() {
        return borradores.getObject();
    }
}
```

Existen además los alcances **`request`**, **`session`**, **`application`** y **`websocket`**, que solo funcionan en un contexto web (Spring Framework Reference, 7.0). Llegan con la web, en la Parte 2.

Proxy, Template Method y Observer, en una línea

- **Proxy:** un objeto que se pone delante de otro y decide qué pasa antes y después de cada llamada (proxies y AOP).
- **Template Method:** un esqueleto fijo con un hueco para lo que cambia, como **`JdbcTemplate`** (excepciones).
- **Observer:** un objeto avisa que algo pasó y otros reaccionan; la referencia llama así a los eventos del contenedor (Spring Framework Reference, 7.0) (eventos).

Patrón	En el libro (GoF)	En Spring
Factory	un objeto que crea otros	**`BeanFactory`**, **`ApplicationContext`**, cada **`@Bean`**, **`FactoryBean`**
Singleton	la clase garantiza una instancia por ClassLoader	el contenedor: una por definición de bean y por contenedor
Prototype	se crea un objeto copiando uno modelo	el contenedor ejecuta la receta otra vez; no copia nada
Proxy	un sustituto que controla el acceso al real	los proxies de AOP, de **`@Transactional`**, de **`@Async`**…
Template Method	el esqueleto en una superclase	**`JdbcTemplate`** y compañía, con *callbacks*
Observer	un sujeto avisa a sus observadores	**`ApplicationEventPublisher`** y **`@EventListener`**

> **Lo que debes llevarte:** el singleton lo comparten todos: nunca guardes en él estado de un cliente. Un prototype dentro de un singleton, pídelo con **`ObjectProvider`**.

Configurar el contenedor: XML, escaneo o clases

Spring lee las recetas de tres formas, y vas a encontrar las tres en código real.

XML, la forma original

En Spring 1.0 las recetas se escribían en XML. Estas son las de **`ConfiguracionPedidos`**:

pedidos.xml

```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
           https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="repositorioPedidos" class="com.ejemplo.pedidos.RepositorioEnMemoria"/>
    <bean id="notificador" class="com.ejemplo.pedidos.NotificadorCorreo"/>

    <bean id="pedidoService" class="com.ejemplo.pedidos.PedidoService">
        <constructor-arg ref="repositorioPedidos"/>
        <constructor-arg ref="notificador"/>
    </bean>
</beans>
```

Se cargan con **`new ClassPathXmlApplicationContext("pedidos.xml")`**. La configuración vive fuera del código, pero las clases se nombran con texto: si renombras **`PedidoService`**, el compilador no te avisa y el error aparece al arrancar.

Escaneo de componentes

Desde Spring 2.5 puedes marcar la clase y dejar que el contenedor la encuentre (Spring Framework 2.5 Released):

ServicioEscaneado.java

```java
@Service
public class ServicioEscaneado {

    private final RepositorioPedidos repositorio;
    private final Notificador notificador;

    public ServicioEscaneado(RepositorioPedidos repositorio, Notificador notificador) {
        this.repositorio = repositorio;
        this.notificador = notificador;
    }

    public void registrar(Pedido pedido) {
        repositorio.guardar(pedido);
        notificador.enviar(pedido.correoCliente(), "Recibimos tu pedido " +
pedido.numero());
```

```
    }
}
```

ConfiguracionEscaneo.java

```
@Configuration
@ComponentScan
public class ConfiguracionEscaneo {}
```

`@ComponentScan` sin parámetros recorre el paquete de la clase que lo declara y sus subpaquetes (Spring Framework API, **`@ComponentScan`**), y registra cada clase marcada con **`@Component`** o con una de sus especializaciones: **`@Service`** para la capa de servicio, **`@Repository`** para la de datos y **`@Controller`** para la web (Spring Framework Reference, 7.0). Es la forma más corta, a cambio de que tu clase importe una anotación de Spring.

Clases de configuración

La tercera, desde Spring 3.0, es la clase **`@Configuration`** con métodos **`@Bean`** de la sección anterior. Funciona con cualquier clase, incluso las que no escribiste: no puedes poner **`@Component`** en **`JdbcTemplate`** o en un **`DataSource`**, pero sí crearlos con **`@Bean`**. Lo habitual es escaneo para tus clases y **`@Bean`** para las de terceros.

Cuando hay dos candidatos

El contenedor resuelve las dependencias por tipo. Si hay dos beans del mismo tipo, no adivina:

ConfiguracionDosNotificadores.java

```
@Configuration
public class ConfiguracionDosNotificadores {

    @Bean
    Notificador correo() {
        return new NotificadorCorreo();
    }

    @Bean
    Notificador sms() {
        return new NotificadorSms();
    }

    @Bean
    RepositorioPedidos repositorioPedidos() {
        return new RepositorioEnMemoria();
    }

    @Bean
```

```
    PedidoService pedidoService(RepositorioPedidos repositorio, Notificador notificador)
{
        return new PedidoService(repositorio, notificador);
    }
}
```

```
NoUniqueBeanDefinitionException: No qualifying bean of type
'com.ejemplo.pedidos.Notificador'
available: expected single matching bean but found 2: correo,sms
```

ConfiguracionTest.java

```
@Test
void conDosCandidatosElContenedorNoArranca() {
    assertThatThrownBy(() -> new
AnnotationConfigApplicationContext(ConfiguracionDosNotificadores.class))
            .hasRootCauseInstanceOf(NoUniqueBeanDefinitionException.class)
            .rootCause()
            .hasMessageContaining("expected single matching bean but found 2:
correo,sms");
}
```

Hay dos formas de desempatar. **`@Primary`** marca el preferido cuando nadie pide otro (Spring Framework Reference, 7.0):

ConfiguracionConPrimary.java

```
@Bean
@Primary
Notificador correo() {
    return new NotificadorCorreo();
}

@Bean
Notificador sms() {
    return new NotificadorSms();
}
```

`@Qualifier` elige en el punto de inyección. El nombre de un bean es su qualifier por defecto, así que **`@Qualifier("sms")`** elige el bean **`sms`**:

ConfiguracionConQualifier.java

```
@Bean
PedidoService pedidoService(RepositorioPedidos repositorio, @Qualifier("sms")
Notificador notificador) {
    return new PedidoService(repositorio, notificador);
```

```
}
```

> **– Para profundizar**
>
> desde Spring 6.2 existe **`@Fallback`**, lo inverso de **`@Primary`**: marca los beans que solo se usan si no queda otro. Y si no hay qualifier ni primario, Spring compara el **nombre del parámetro** con los nombres de los beans; desde 6.1 eso exige compilar con **`-parameters`** (Spring Framework Reference, 7.0). Mi proyecto compila sin esa opción a propósito, para que todo desempate quede escrito en el código.

Perfiles: una receta por ambiente

En tu máquina no quieres mandar correos de verdad; en producción, sí. Un **perfil** es un nombre que activa o desactiva recetas: **`@Profile("local")`** registra el bean solo si el perfil **`local`** está activo (Spring Framework Reference, 7.0).

ConfiguracionPorAmbiente.java

```java
@Configuration
public class ConfiguracionPorAmbiente {

    @Bean
    @Profile("local")
    Notificador notificadorDeConsola() {
        return new NotificadorConsola();
    }

    @Bean
    @Profile("produccion")
    Notificador notificadorDeCorreo() {
        return new NotificadorCorreo();
    }
}
```

ConfiguracionTest.java

```java
@Test
void elPerfilActivoEligeElNotificador() {
    try (var contexto = new AnnotationConfigApplicationContext()) {
        contexto.getEnvironment().setActiveProfiles("local");
        contexto.register(ConfiguracionPorAmbiente.class);
        contexto.refresh();


assertThat(contexto.getBean(Notificador.class)).isInstanceOf(NotificadorConsola.class);
```

```
    }
}
```

Fuera de un test, el perfil se activa con la propiedad **`spring.profiles.active`**, por ejemplo **`-Dspring.profiles.active=produccion`**. Sin ninguno activo aplica el perfil **`default`** (Spring Framework Reference, 7.0); como aquí ninguna receta es **`default`**, sin perfil no hay notificador.

El ciclo de vida de un bean

Una conexión de correo se abre cuando el objeto está listo y se cierra cuando la aplicación se apaga. Sin ayuda, tendrías que llamar a **`abrir()`** y a **`cerrar()`** en todos los caminos posibles. El contenedor sabe cuándo pasa cada cosa y avisa: **`@PostConstruct`** corre cuando el bean ya tiene sus dependencias, y **`@PreDestroy`**, antes de destruirlo.

ConexionCorreo.java

```java
public class ConexionCorreo {

    private final Bitacora bitacora;

    public ConexionCorreo(Bitacora bitacora) {
        this.bitacora = bitacora;
        bitacora.anotar("constructor: ya tiene sus dependencias");
    }

    @PostConstruct
    void abrir() {
        bitacora.anotar("@PostConstruct: abre la conexión");
    }

    public void enviar(String destino) {
        bitacora.anotar("en uso: envía a " + destino);
    }

    @PreDestroy
    void cerrar() {
        bitacora.anotar("@PreDestroy: cierra la conexión");
    }
}
```

La vida de ConexionCorreo, de la receta al cierre

Receta leída	Creado e inyectado	Inicializado	En uso	Destruido
				✕ ~~Bitacora~~



El contenedor arranca y lee las dos recetas. Todavía no existe ningún objeto.

ConexionCorreo necesita una Bitacora en su constructor, así que el contenedor crea primero la Bitacora.

La Bitacora no tiene nada que inicializar: queda lista. Ahora sí, el contenedor llama al constructor de ConexionCorreo y le pasa la Bitacora. Con inyección por constructor, crear e inyectar son un solo paso.

Con las dependencias ya puestas, el contenedor llama al método @PostConstruct: abre la conexión. Es el primer momento en que el objeto puede usar lo que recibió.

Después vuelve a pasar por los BeanPostProcessor, ahora en su fase posterior a la inicialización (el @PostConstruct mismo lo ejecutó uno de ellos, en la fase anterior). Pueden devolver el bean tal cual o cambiarlo por otro objeto, como un proxy.

El bean está listo. Tu código lo pide con getBean y lo usa: envía un correo. Todo lo anterior pasó al arrancar, antes de este momento.

Al cerrar el contenedor, se destruye primero quien depende: el @PreDestroy de ConexionCorreo cierra la conexión mientras la Bitacora todavía existe.

Recién entonces se destruye la Bitacora, de la que dependía la conexión.

El test comprueba ese orden. Otro comprueba que, al destruir, el **`@PreDestroy`** del bean que depende de otro corre primero.

CicloDeVidaTest.java

```
@Test
void unBeanRecorreSuCicloDeVidaEnOrden() {
    var contexto = new
AnnotationConfigApplicationContext(ConfiguracionCicloDeVida.class);
    var bitacora = contexto.getBean(Bitacora.class);

    contexto.getBean(ConexionCorreo.class).enviar("ana@example.com");
    contexto.close();

    assertThat(bitacora.entradas()).containsExactly(
            "constructor: ya tiene sus dependencias",
            "@PostConstruct: abre la conexión",
            "BeanPostProcessor: ve el bean ya inicializado",
            "en uso: envía a ana@example.com",
            "@PreDestroy: cierra la conexión");
}
```

Tres detalles prácticos:

- **`@PostConstruct`** y **`@PreDestroy`** no son de Spring. Antes eran **`javax.annotation`** y venían con el JDK, que los quitó en Java 11; hoy son **`jakarta.annotation`** y se agregan con **`jakarta.annotation-api`** (Spring Framework Reference, 7.0). Está en el bloque de dependencias de la sección de excepciones.
- Las interfaces **`InitializingBean`** y **`DisposableBean`** hacen lo mismo, pero el equipo de Spring recomienda las anotaciones, con las que tu clase no depende de Spring (Spring Framework Reference, 7.0).
- Fuera de una aplicación web, alguien tiene que cerrar el contenedor para que corran los **`@PreDestroy`**: el **`try`** con recursos de los ejemplos, o **`registerShutdownHook()`**, que lo cierra cuando se apaga la JVM (Spring Framework Reference, 7.0).

> El error típico: usar **`@PostConstruct`** para trabajo pesado, como cargar datos o llamar a otros servicios. Corre mientras el contenedor todavía crea beans, y la referencia advierte que es para validar y preparar estructuras, con riesgo de bloqueos durante el arranque si hace más. Para lo pesado propone **`SmartInitializingSingleton`** o un listener del evento que avisa que el contexto terminó de arrancar (Spring Framework Reference, 7.0). En transacciones hay otro motivo.

> – Para profundizar
>
> un **`BeanPostProcessor`** recibe cada bean que el contenedor crea, antes y después de sus métodos de inicialización (Spring Framework Reference, 7.0), y puede devolverlo tal cual o reemplazarlo: así pone el contenedor un proxy delante de tu bean. Este solo anota que lo vio:
>
> ObservadorDeBeans.java
>
> ```
> public class ObservadorDeBeans implements BeanPostProcessor {
>
>     private final ObjectProvider<Bitacora> bitacora;
>
>     public ObservadorDeBeans(ObjectProvider<Bitacora> bitacora) {
>         this.bitacora = bitacora;
>     }
>
>     @Override
>     public Object postProcessAfterInitialization(Object bean, String nombre) {
>         if (bean instanceof ConexionCorreo) {
>             bitacora.getObject().anotar("BeanPostProcessor: ve el bean ya
> inicializado");
>         }
>         return bean;
>     }
> }
> ```

> Mucha de la "magia" son post-processors: **`@Autowired`** lo procesa **`AutowiredAnnotationBeanPostProcessor`**, y **`@PostConstruct`**/**`@PreDestroy`**, **`CommonAnnotationBeanPostProcessor`** (Spring Framework Reference, 7.0). Por eso un **`BeanFactory`** solo, que no los registra, ignora esas anotaciones.
>
> Un post-processor se crea antes que los beans que procesa. Por eso su método **`@Bean`** va **`static`** (no obliga a crear antes su clase de configuración) e idealmente sin dependencias, porque un bean creado para armarlo se queda sin post-procesar (Spring Framework API, **`@Bean`**). **`ObservadorDeBeans`** recibe la **`Bitacora`** a través de un **`ObjectProvider`**, que la pide recién cuando la necesita.

Propiedades: la configuración fuera del código

Si el costo del envío está en el código, cambiarlo es recompilar y desplegar. Lo que cambia sin que cambie la lógica va en **propiedades**: pares de clave y valor que se leen al arrancar.

tienda.properties

```
tienda.nombre=La tienda de Ana
envio.gratis-desde=50000
envio.costo=4990
```

Las administra el **`Environment`**, el objeto del contenedor que modela los perfiles y las propiedades (Spring Framework Reference, 7.0). Sus propiedades salen de una lista ordenada de **`PropertySource`**, cualquier origen de pares clave-valor. De fábrica trae dos: las propiedades de sistema de la JVM (**`-Dclave=valor`**) y las variables de entorno, en ese orden; si una clave está en las dos, gana la de sistema.

`@PropertySource` agrega un archivo a la lista y **`@Value`** inyecta un valor:

ConfiguracionTienda.java

```
@Configuration
@PropertySource("classpath:tienda.properties")
public class ConfiguracionTienda {

    @Bean
    TarifaEnvio tarifaEnvio(@Value("${envio.gratis-desde}") int gratisDesde,
                            @Value("${envio.costo}") int costo) {
        return new TarifaEnvio(gratisDesde, costo);
    }
}
```

`${envio.costo}` es un **marcador** que se reemplaza por el valor de esa clave, convertido al **`int`** que pide el parámetro. Como el archivo va después de las propiedades de sistema, una propiedad de sistema le gana: así se cambia un valor en un ambiente sin tocar el archivo.

PropiedadesTest.java

```
@Test
void losValoresSalenDelArchivo() {
    try (var contexto = new
AnnotationConfigApplicationContext(ConfiguracionTienda.class)) {
        var tarifa = contexto.getBean(TarifaEnvio.class);
        assertThat(tarifa.costoPara(20_000)).isEqualTo(4_990);
        assertThat(tarifa.costoPara(60_000)).isZero();

        var entorno = contexto.getEnvironment();
        assertThat(entorno.getProperty("tienda.nombre")).isEqualTo("La tienda de Ana");
    }
}

@Test
void unaPropiedadDelSistemaLeGanaAlArchivo() {
    System.setProperty("envio.costo", "0");
    try (var contexto = new
AnnotationConfigApplicationContext(ConfiguracionTienda.class)) {
        assertThat(contexto.getBean(TarifaEnvio.class).costoPara(20_000)).isZero();
    }
}
```

El error típico: una errata en la clave. Esta receta pide **`tienda.nombr`**:

ConfiguracionConErrata.java

```
@Bean
Bienvenida bienvenida(@Value("${tienda.nombr}") String nombre) {
    return new Bienvenida(nombre);
}
```

No hay error al arrancar: por defecto, el resolvedor es tolerante y, si no encuentra la clave, inyecta el texto del marcador tal cual (Spring Framework Reference, 7.0). La tienda se presenta como **`${tienda.nombr}`**. Para que el contenedor falle al arrancar, declara un **`PropertySourcesPlaceholderConfigurer`** en un método **`static`**:

ConfiguracionEstricta.java

```
@Bean
static PropertySourcesPlaceholderConfigurer placeholders() {
    return new PropertySourcesPlaceholderConfigurer();
```

```
}
```

PropiedadesTest.java

```
@Test
void sinResolvedorEstrictoLaErrataLlegaComoTexto() {
    try (var contexto = new
AnnotationConfigApplicationContext(ConfiguracionConErrata.class)) {

assertThat(contexto.getBean(Bienvenida.class).nombreTienda()).isEqualTo("${tienda.no
mbr}");
    }
}

@Test
void conResolvedorEstrictoElContenedorNoArranca() {
    assertThatThrownBy(() -> new
AnnotationConfigApplicationContext(ConfiguracionEstricta.class))
            .hasRootCauseInstanceOf(PlaceholderResolutionException.class)
            .hasStackTraceContaining("'tienda.nombr'");
}
```

Si una clave puede faltar a propósito, dale un valor por defecto: **`${envio.costo:0}`**.

En la Parte 2, Spring Boot arma esta lista de fuentes por ti; el modelo sigue siendo el mismo: gana la primera fuente que tiene la clave.

Excepciones a la manera de Spring

Para esta sección y las dos siguientes, tu proyecto suma estas dependencias:

pom.xml

```
<!-- @PostConstruct y @PreDestroy: el JDK quitó javax.annotation en Java 11; hoy son
jakarta.annotation. -->
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>3.0.0</version>
</dependency>
<!-- Para los aspectos con @Aspect. -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.25.1</version>
```

```
</dependency>
<!-- JdbcTemplate, la traducción de excepciones y las transacciones. -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
</dependency>
<!-- Una base de datos en memoria: nace y muere con cada test. -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.4.240</version>
    <scope>runtime</scope>
</dependency>
```

H2 es una base de datos que corre en memoria, dentro del mismo proceso: nace con el test y muere con él. Esta es su tabla de pedidos:

schema.sql

```
create table pedidos (
    numero         int primary key,
    correo_cliente varchar(200) not null
);
```

El problema: SQLException

Con JDBC puro, guardar un pedido se ve así:

RepositorioJdbcAMano.java

```
public void guardar(Pedido pedido) throws SQLException {
    try (Connection conexion = dataSource.getConnection();
            PreparedStatement sentencia = conexion.prepareStatement(
                    "insert into pedidos (numero, correo_cliente) values (?, ?)")) {
        sentencia.setInt(1, pedido.numero());
        sentencia.setString(2, pedido.correoCliente());
        sentencia.executeUpdate();
    }
}
```

`SQLException` es **checked**: el compilador obliga a cada método que llame a **`guardar`** a atraparla o a declararla. Y casi nadie puede hacer algo con ella: si se cayó la base, el servicio de pedidos no tiene

cómo arreglarlo. El resultado típico es una cadena de **`throws SQLException`** por todas las capas, o un **`catch`** vacío.

Además dice poco. Si guardas dos veces el pedido 42, no dice "pedido repetido": dice un código, el **`SQLState 23505`**, que en H2 es una violación de unicidad. Lo comprobé con un test. Para reaccionar, tu código tendría que conocer los códigos de su base de datos.

La solución: DataAccessException

`JdbcTemplate` se encarga de lo repetitivo: crea y libera los recursos, ejecuta la sentencia y te deja solo el SQL (Spring Framework Reference, 7.0). Es el Template Method de la sección de patrones.

ANTES

RepositorioJdbcAMano.java

```
public void guardar(Pedido pedido) throws
SQLException {
    try (Connection conexion =
dataSource.getConnection();
            PreparedStatement sentencia =
conexion.prepareStatement(
                    "insert into pedidos
(numero, correo_cliente) values (?, ?)"))
{
        sentencia.setInt(1,
pedido.numero());
        sentencia.setString(2,
pedido.correoCliente());
        sentencia.executeUpdate();
    }
}
```

DESPUÉS

RepositorioJdbc.java

```
@Override
public void guardar(Pedido pedido) {
    jdbc.update("insert into pedidos
(numero, correo_cliente) values (?, ?)",
            pedido.numero(),
pedido.correoCliente());
}
```

Además **traduce** la **`SQLException`** a la jerarquía de Spring, cuya raíz es **`DataAccessException`**: excepciones **unchecked** (heredan de **`RuntimeException`**) que dicen qué pasó con su nombre y envuelven la original, para que no se pierda ningún detalle (Spring Framework Reference, 7.0):

ExcepcionesTest.java

```
@Test
void conJdbcTemplateLlegaUnaExcepcionDeSpringQueDiceQuePaso() {
    try (var contexto = new
AnnotationConfigApplicationContext(ConfiguracionDatos.class)) {
        var repositorio = contexto.getBean(RepositorioJdbc.class);
        repositorio.guardar(new Pedido(42, "ana@example.com"));

        assertThatThrownBy(() -> repositorio.guardar(new Pedido(42,
"luis@example.com")))
```

```
                .isInstanceOf(DuplicateKeyException.class)
                .isInstanceOf(DataIntegrityViolationException.class)
                .isInstanceOf(DataAccessException.class)
                .isInstanceOf(RuntimeException.class)
                .hasCauseInstanceOf(SQLException.class);
    }
}
```

¿Por qué unchecked? Porque la mayoría de los errores de base de datos no se pueden resolver donde ocurren. Así, las capas intermedias no necesitan bloques que atrapan y relanzan ni declaraciones **`throws`**: la excepción sube hasta la capa que sí puede hacer algo (Spring Framework Reference, 7.0).

El **`DataSource`** (la fuente de conexiones) y el **`JdbcTemplate`** no son clases tuyas, así que van con **`@Bean`**:

ConfiguracionDatos.java

```
@Configuration
public class ConfiguracionDatos {

    @Bean(destroyMethod = "shutdown")
    EmbeddedDatabase dataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .generateUniqueName(true)
                .addScript("schema.sql")
                .build();
    }

    @Bean
    JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    RepositorioJdbc repositorioPedidos(JdbcTemplate jdbc) {
        return new RepositorioJdbc(jdbc);
    }
}
```

> – Para profundizar
>
> desde Spring 6.0, el traductor por defecto es **`SQLExceptionSubclassTranslator`**, que se basa en las subclases de **`SQLException`** de JDBC 4 y, si no alcanza, en el **`SQLState`** (Spring Framework Reference, 7.0). Spring también traduce las excepciones de JPA y Hibernate: **`@Repository`** marca la clase y un **`PersistenceExceptionTranslationPostProcessor`** registrado en el contenedor hace la traducción

> automática; Spring Boot lo registra por ti (Spring Framework Reference, 7.0). Lo vas a ver con Spring Data en la Parte 2.

Proxies y AOP: el código que se repite en todos lados

Registrar quién llamó a qué, medir cuánto tardó, abrir una transacción, comprobar permisos: hay código que aparece en todos los métodos sin pertenecer a ninguno. Se le llama **preocupación transversal** (*crosscutting concern*). Escrito a mano, se copia en cada método, mezclado con el negocio:

ANTES

ServicioConBitacoraRepetida.java

```
public void registrar(Pedido pedido) {
    bitacora.anotar("entra a registrar");
    try {
        repositorio.guardar(pedido);
    } finally {
        bitacora.anotar("sale de
registrar");
    }
}

public int contar() {
    bitacora.anotar("entra a contar");
    try {
        return
repositorio.todos().size();
    } finally {
        bitacora.anotar("sale de
contar");
    }
}
```

DESPUÉS

ServicioDePedidos.java

```
@Auditado
public void registrar(Pedido pedido) {
    repositorio.guardar(pedido);
}

@Auditado
public int contar() {
    return repositorio.todos().size();
}
```

A la derecha, una anotación propia marca dónde se quiere la bitácora:

Auditado.java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditado {}
```

Por sí sola no hace nada. Lo que la hace funcionar es un **aspecto**:

AspectoDeAuditoria.java

```java
@Aspect
public class AspectoDeAuditoria {

    private final Bitacora bitacora;

    public AspectoDeAuditoria(Bitacora bitacora) {
        this.bitacora = bitacora;
    }

    @Around("@annotation(com.ejemplo.pedidos.aop.Auditado)")
    public Object auditar(ProceedingJoinPoint llamada) throws Throwable {
        String metodo = llamada.getSignature().getName();
        bitacora.anotar("entra a " + metodo);
        try {
            return llamada.proceed();
        } finally {
            bitacora.anotar("sale de " + metodo);
        }
    }
}
```

La **programación orientada a aspectos** (AOP, por *Aspect-Oriented Programming*) escribe lo transversal una vez, en un módulo aparte, y declara dónde se aplica. Su vocabulario (Spring Framework Reference, 7.0):

- El **aspecto** es ese módulo, la clase **`AspectoDeAuditoria`**.
- Un **join point** es un punto de la ejecución donde se puede intervenir; en Spring, siempre la ejecución de un método.
- El **pointcut** dice en cuáles se aplica: **`@annotation(…Auditado)`**, los métodos con **`@Auditado`**.
- El **advice** es lo que se ejecuta ahí. **`@Around`** envuelve la llamada: hace algo antes, decide si sigue con **`proceed()`** y hace algo después.

Se activa con **`@EnableAspectJAutoProxy`** y la librería **`aspectjweaver`**, que Spring usa para leer los pointcuts (Spring Framework Reference, 7.0):

ConfiguracionAop.java

```java
@Configuration
@EnableAspectJAutoProxy
public class ConfiguracionAop {

    @Bean
    Bitacora bitacora() {
        return new Bitacora();
    }
```

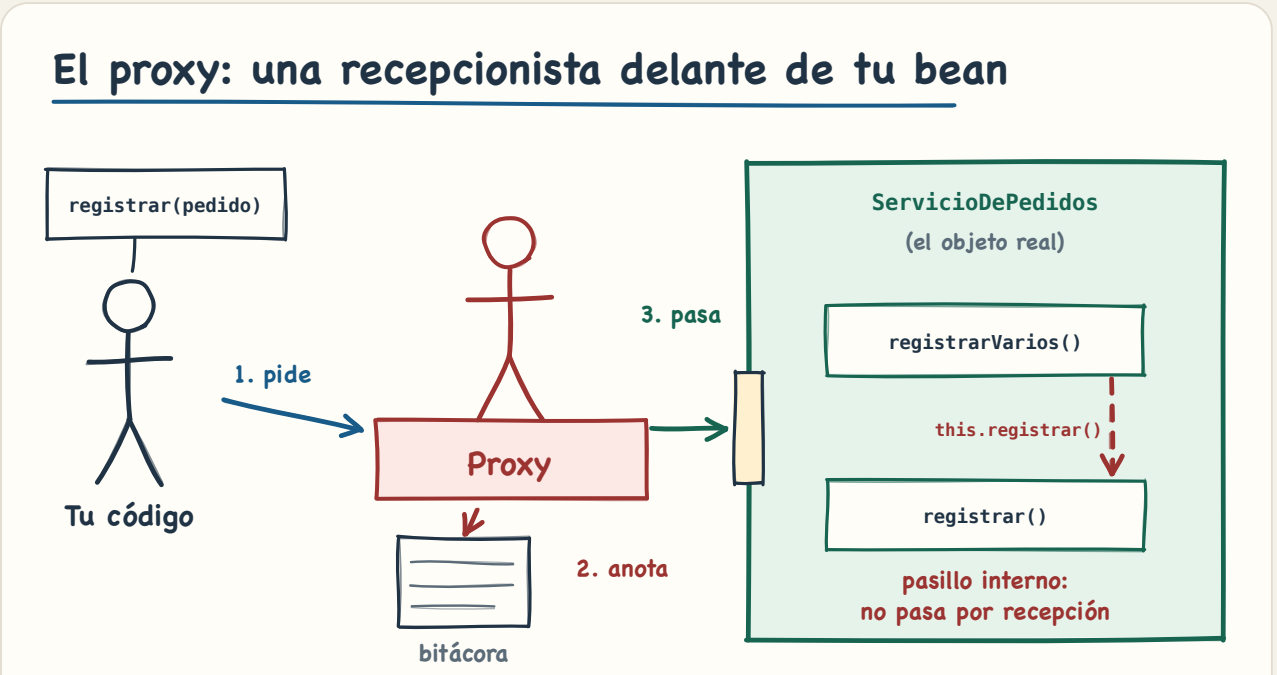
```
    @Bean
    AspectoDeAuditoria aspectoDeAuditoria(Bitacora bitacora) {
        return new AspectoDeAuditoria(bitacora);
    }

    @Bean
    RepositorioPedidos repositorioPedidos() {
        return new RepositorioEnMemoria();
    }

    @Bean
    ServicioDePedidos servicioDePedidos(RepositorioPedidos repositorio) {
        return new ServicioDePedidos(repositorio);
    }
}
```

Cómo funciona: el proxy

Spring no cambia tu clase. Pone otro objeto delante:



Lo que entra por la puerta pasa por la recepción; lo que el servicio se pide a sí mismo, no.

El proxy es la recepcionista: todo lo que entra por la puerta pasa por ella; lo que el servicio se pide a sí mismo, no.

Cuando un bean está alcanzado por un aspecto, el contenedor crea un **proxy** con su misma forma, y es el proxy lo que te entrega (Spring Framework Reference, 7.0). Si la clase implementa interfaces, el proxy es

un *JDK dynamic proxy* de esas interfaces; si no, una subclase generada con CGLIB (Spring Framework Reference, 7.0). **`ServicioDePedidos`** no implementa ninguna:

AopTest.java

```
@Test
void elContenedorEntregaUnProxyQuePasaPorElAspecto() {
    try (var contexto = new AnnotationConfigApplicationContext(ConfiguracionAop.class))
{
        var servicio = contexto.getBean(ServicioDePedidos.class);
        var bitacora = contexto.getBean(Bitacora.class);

        servicio.registrar(new Pedido(42, "ana@example.com"));

        assertThat(AopUtils.isCglibProxy(servicio)).isTrue();
        assertThat(bitacora.entradas()).containsExactly("entra a registrar", "sale de
registrar");
    }
}
```

El mismo mecanismo hace funcionar **`@Transactional`**, **`@Async`** y otras anotaciones de Spring, con las mismas dos trampas.

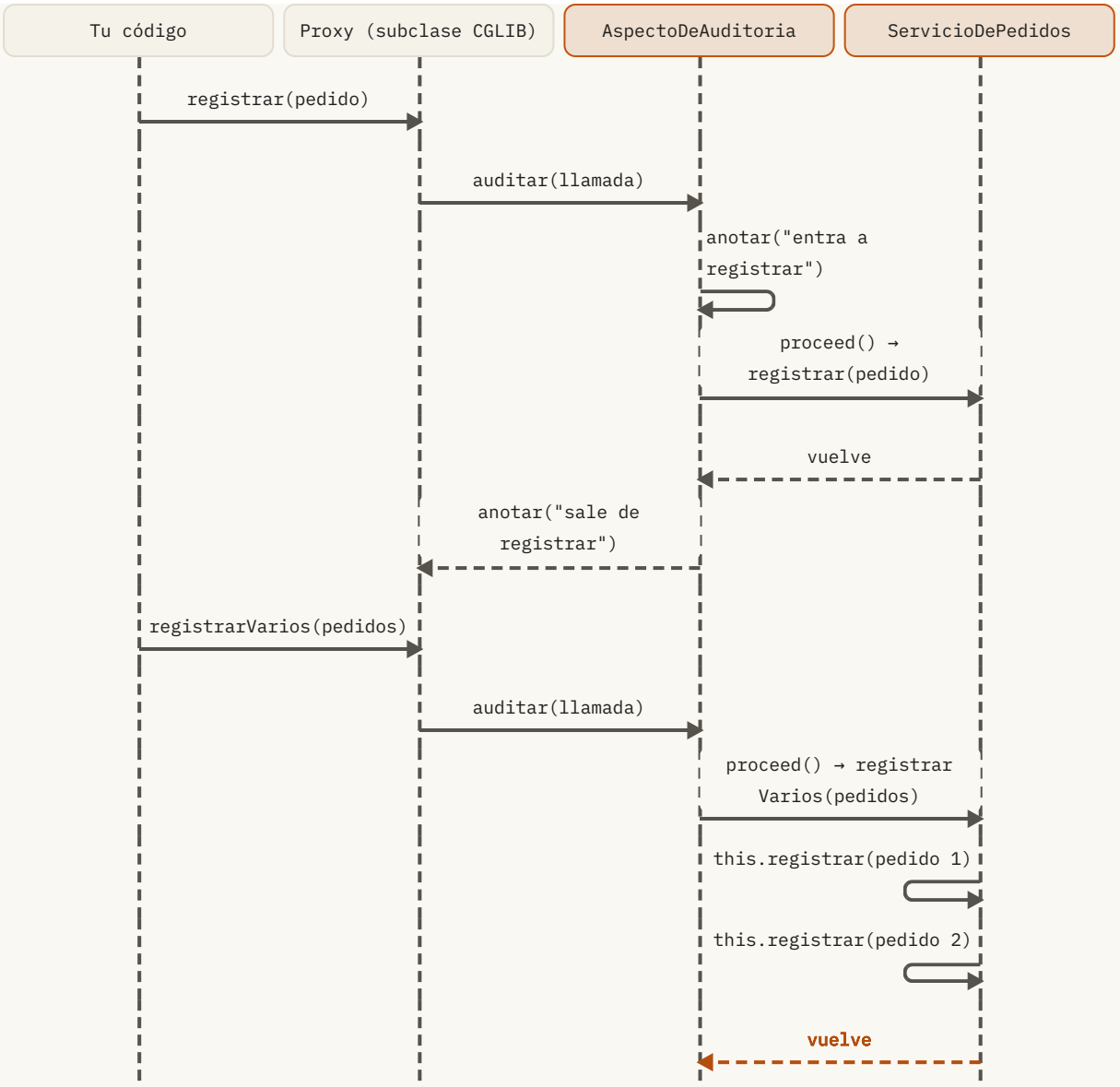
La trampa de la autoinvocación

Este método registra varios pedidos llamando a **`registrar`** en un bucle:

ServicioDePedidos.java

```
@Auditado
public void registrarVarios(List<Pedido> pedidos) {
    for (Pedido pedido : pedidos) {
        this.registrar(pedido);
    }
}
```

Una llamada pasa por el proxy; la autoinvocación, no



Lo que recibiste de getBean no es tu ServicioDePedidos: es el proxy que el contenedor puso delante. Tu código no lo nota, porque tiene la misma forma.

El método tiene @Auditado, así que el proxy le pasa la llamada al aspecto antes de tocar tu objeto.

El advice @Around hace su parte de antes.

proceed() sigue hacia el objeto real, que guarda el pedido. Es el único momento en que el aspecto deja pasar la llamada.

El método termina y el control vuelve al aspecto.

El finally anota la salida y la respuesta vuelve por el proxy hasta tu código. En la bitácora quedaron dos líneas.

Ahora llamas a registrarVarios. También tiene @Auditado, y entra por el proxy.

El aspecto anota "entra a registrarVarios" y sigue con proceed().

La llamada llega al objeto real.

Acá está la trampa. Dentro del objeto, this es el objeto real, no el proxy. La llamada no sale del objeto: nadie la intercepta y el aspecto no se entera.

Lo mismo con el segundo pedido. Se guardan los dos, pero ninguno queda en la bitácora.

registrarVarios termina, el aspecto anota "sale de registrarVarios" y la respuesta vuelve por el proxy. La bitácora tiene dos líneas, no seis.

AopTest.java

```
@Test
void laAutoinvocacionNoPasaPorElProxy() {
    try (var contexto = new AnnotationConfigApplicationContext(ConfiguracionAop.class))
{
        var servicio = contexto.getBean(ServicioDePedidos.class);
        var bitacora = contexto.getBean(Bitacora.class);

        servicio.registrarVarios(List.of(new Pedido(1, "ana@example.com"), new Pedido(2,
"luis@example.com")));

        assertThat(bitacora.entradas()).containsExactly("entra a registrarVarios", "sale
de registrarVarios");
        assertThat(servicio.contar()).isEqualTo(2);
    }
}
```

Para la referencia, la autoinvocación se salta el advice, y lo mejor es reorganizar el código, por ejemplo moviendo el método interno a otro bean. La alternativa es que el bean se inyecte a sí mismo (*self injection*) y llame por esa referencia, que es el proxy; **`AopContext.currentProxy()`** queda como último recurso, y la referencia lo desaconseja con fuerza (Spring Framework Reference, 7.0).

Lo que un proxy no puede interceptar

Un proxy CGLIB es una subclase, y solo puede intervenir en lo que puede sobrescribir: los métodos **`public`** y **`protected`** (y los de paquete, si hace falta). Un proxy JDK, solo los públicos de la interfaz. Los **`final`** y los **`private`** no se pueden interceptar (Spring Framework Reference, 7.0).

Un método **`final`** es peor de lo que parece, y Spring lo avisa al arrancar:

```
WARNING: Public final method [public final int
com.ejemplo.pedidos.aop.ServicioDePedidos.contarSinAspecto()]
cannot get proxied via CGLIB, consider removing the final marker or using interface-
based JDK proxies.
```

ServicioDePedidos.java

```
@Auditado
public final int contarSinAspecto() {
    return repositorio.todos().size();
}
```

Como la subclase no puede sobrescribirlo, la llamada no llega al objeto real: corre sobre el propio proxy, que no pasó por tu constructor y tiene los campos vacíos. Lo comprobé con un test, y el resultado no es un método sin bitácora: es un **`NullPointerException`**, porque **`repositorio`** es **`null`** dentro del proxy.

AopTest.java

```
@Test
void unMetodoFinalCorreSobreElProxyVacio() {
    try (var contexto = new AnnotationConfigApplicationContext(ConfiguracionAop.class))
{
        var servicio = contexto.getBean(ServicioDePedidos.class);


assertThatThrownBy(servicio::contarSinAspecto).isInstanceOf(NullPointerException.class);
        assertThat(contexto.getBean(Bitacora.class).entradas()).isEmpty();
    }
}
```

Lo que debes llevarte: lo que el contenedor te entrega es el proxy, y solo se intercepta lo que entra por el proxy: las llamadas con **`this`** no, y los métodos **`final`** o **`private`** tampoco. Las firmas que deben pasar por un proxy, públicas.

– Para profundizar

el Framework sugiere proxies por interfaz, y Spring Boot usa proxies por clase por defecto (**`spring.aop.proxy-target-class=true`**). Desde 7.0, **`@Proxyable(INTERFACES)`** o **`@Proxyable(TARGET_CLASS)`** fija el tipo para un bean (Spring Framework Reference, 7.0). Todo esto es Spring AOP, basado en proxies; si necesitas más, Spring puede usar el compilador o el tejedor de AspectJ, que modifican las clases mismas (Spring Framework Reference, 7.0). Esta serie no los usa.

Transacciones

Qué es una transacción

Comprar es guardar el pedido y cobrarlo. ¿Qué pasa si el pedido se guarda y el cobro falla?

ANTES

ComprasSinTransaccion.java

DESPUÉS

Compras.java

```
@Transactional
```

```
public void comprar(Pedido pedido, int
monto) {
    repositorio.guardar(pedido);
    pagos.cobrar(pedido.numero(), monto);
}
```

```
public void comprar(Pedido pedido, int
monto) {
    repositorio.guardar(pedido);
    pagos.cobrar(pedido.numero(), monto);
}
```

Sin transacción, la base queda con un pedido que nadie pagó. Una **transacción** agrupa varias operaciones para que se apliquen todas o ninguna: si todo sale bien, se **confirma** (*commit*); si algo falla, se **deshace** (*rollback*) y la base vuelve a como estaba. En los dos tests, **`Pagos`** rechaza un pago de monto 0:

TransaccionesTest.java

```
@Test
void sinTransaccionElPedidoQuedaGuardadoSinPago() {
    var compras = contexto.getBean(ComprasSinTransaccion.class);

    assertThatThrownBy(() -> compras.comprar(PEDIDO,
0)).isInstanceOf(PagoRechazado.class);

    assertThat(filas("pedidos")).isEqualTo(1);
    assertThat(filas("pagos")).isZero();
}

@Test
void conTransaccionNoQuedaNada() {
    var compras = contexto.getBean(Compras.class);

    assertThatThrownBy(() -> compras.comprar(PEDIDO,
0)).isInstanceOf(PagoRechazado.class);

    assertThat(filas("pedidos")).isZero();
    assertThat(filas("pagos")).isZero();
}
```

Cómo lo hace Spring

`@Transactional` funciona con un proxy delante de **`Compras`**: antes de entrar al método abre una transacción; si el método termina bien, la confirma; si sale una excepción, la deshace. Necesita dos piezas:

ConfiguracionTransacciones.java

```
@Configuration
@EnableTransactionManagement
@Import(ConfiguracionDatos.class)
public class ConfiguracionTransacciones {
```

```
    @Bean
    PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

`@EnableTransactionManagement` activa los proxies, y el **`DataSourceTransactionManager`** sabe abrir, confirmar y deshacer una transacción sobre ese **`DataSource`**. Al abrirla, asocia una conexión al hilo actual, y **`JdbcTemplate`** usa esa conexión en vez de pedir otra (Spring Framework API, **`DataSourceTransactionManager`**). Por eso el repositorio y **`Pagos`** entran en la misma transacción y se deshacen juntos.

La anotación va en **`comprar`**, en el servicio, no en el repositorio: la unidad de trabajo es "comprar", que toca dos tablas, y el repositorio de pedidos no sabe que después viene un cobro.

Sin parámetros, **`@Transactional`** usa propagación **`REQUIRED`**, el aislamiento de la base de datos, lectura y escritura, y rollback ante **`RuntimeException`** o **`Error`**, pero **no ante una excepción checked** (Spring Framework Reference, 7.0).

> – Para profundizar
>
> el JDBC a mano de la sección anterior, que llama a **`dataSource.getConnection()`**, **no** entraría en la transacción: pediría una conexión nueva. Hay que pedirla con **`DataSourceUtils.getConnection(dataSource)`**, como hace **`JdbcTemplate`** por dentro (Spring Framework API, **`DataSourceTransactionManager`**).

Rollback solo con unchecked

`SaldoInsuficiente` es una excepción checked. Este método guarda el pedido y después la lanza:

Compras.java

```
@Transactional
public void comprarACredito(Pedido pedido, int monto) throws SaldoInsuficiente {
    repositorio.guardar(pedido);
    if (monto > 100_000) {
        throw new SaldoInsuficiente(pedido.numero());
    }
    pagos.cobrar(pedido.numero(), monto);
}
```

La excepción sale del método, pero la transacción **se confirma** y el pedido queda guardado. Para que una checked también deshaga, se declara con **`rollbackFor`**, que abarca también a sus subclases (Spring

Framework Reference, 7.0). Desde 6.2 también se puede cambiar la regla para toda la aplicación con **`@EnableTransactionManagement(rollbackOn = RollbackOn.ALL_EXCEPTIONS)`**, que el javadoc recomienda salvo que dependas de excepciones de negocio que confirman a propósito:

Compras.java

```
@Transactional(rollbackFor = SaldoInsuficiente.class)
public void comprarACreditoSinDejarRastro(Pedido pedido, int monto) throws
SaldoInsuficiente {
    repositorio.guardar(pedido);
    if (monto > 100_000) {
        throw new SaldoInsuficiente(pedido.numero());
    }
    pagos.cobrar(pedido.numero(), monto);
}
```

TransaccionesTest.java

```
@Test
void unaExcepcionCheckedNoDeshaceLaTransaccion() {
    var compras = contexto.getBean(Compras.class);

    assertThatThrownBy(() -> compras.comprarACredito(PEDIDO,
200_000)).isInstanceOf(SaldoInsuficiente.class);

    assertThat(filas("pedidos")).isEqualTo(1);
}

@Test
void conRollbackForTambienLaDeshace() {
    var compras = contexto.getBean(Compras.class);

    assertThatThrownBy(() -> compras.comprarACreditoSinDejarRastro(PEDIDO, 200_000))
            .isInstanceOf(SaldoInsuficiente.class);

    assertThat(filas("pedidos")).isZero();
}
```

Propagación: unirse o empezar otra

Cuando un método **`@Transactional`** llama a otro, la **propagación** decide qué pasa:

- **`REQUIRED`**, la de por defecto: si ya hay una transacción, se une a ella; si no, abre una. Todos los que se unen comparten la misma transacción física (Spring Framework Reference, 7.0).
- **`REQUIRES_NEW`**: siempre abre una transacción nueva e independiente, con su propia conexión, que se confirma o se deshace por su cuenta.

REQUIRES_NEW sirve para lo que tiene que quedar escrito aunque la operación principal falle, como la auditoría:

Auditoria.java

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void registrar(String mensaje) {
    jdbc.update("insert into auditoria (mensaje) values (?)", mensaje);
}
```

Compras.java

```
@Transactional
public void comprarConAuditoria(Pedido pedido, int monto) {
    auditoria.registrar("intento de compra del pedido " + pedido.numero());
    repositorio.guardar(pedido);
    pagos.cobrar(pedido.numero(), monto);
}
```

El pago falla, el pedido se deshace y la línea de auditoría queda:

TransaccionesTest.java

```
@Test
void laAuditoriaEnSuPropiaTransaccionSobreviveAlRollback() {
    var compras = contexto.getBean(Compras.class);

    assertThatThrownBy(() -> compras.comprarConAuditoria(PEDIDO,
0)).isInstanceOf(PagoRechazado.class);

    assertThat(filas("pedidos")).isZero();
    assertThat(jdbc.queryForList("select mensaje from auditoria", String.class))
            .containsExactly("intento de compra del pedido 42");
}
```

> **Cuidado:** mientras corre la transacción interna, la externa sigue abierta con su conexión. Con muchos hilos haciendo esto a la vez, el pool de conexiones se puede agotar y los hilos quedan esperándose. La referencia pide no usar **REQUIRES_NEW** salvo que el pool tenga al menos una conexión más que hilos concurrentes (Spring Framework Reference, 7.0).

La trampa del rollback-only

Cupones.aplicar es transaccional y lanza **CuponInvalido**, unchecked, si el cupón no existe:

Cupones.java

```
public class Cupones {

    @Transactional
    public int aplicar(String codigo) {
        if (!codigo.equals("BIENVENIDA")) {
            throw new CuponInvalido(codigo);
        }
        return 10;
    }
}
```

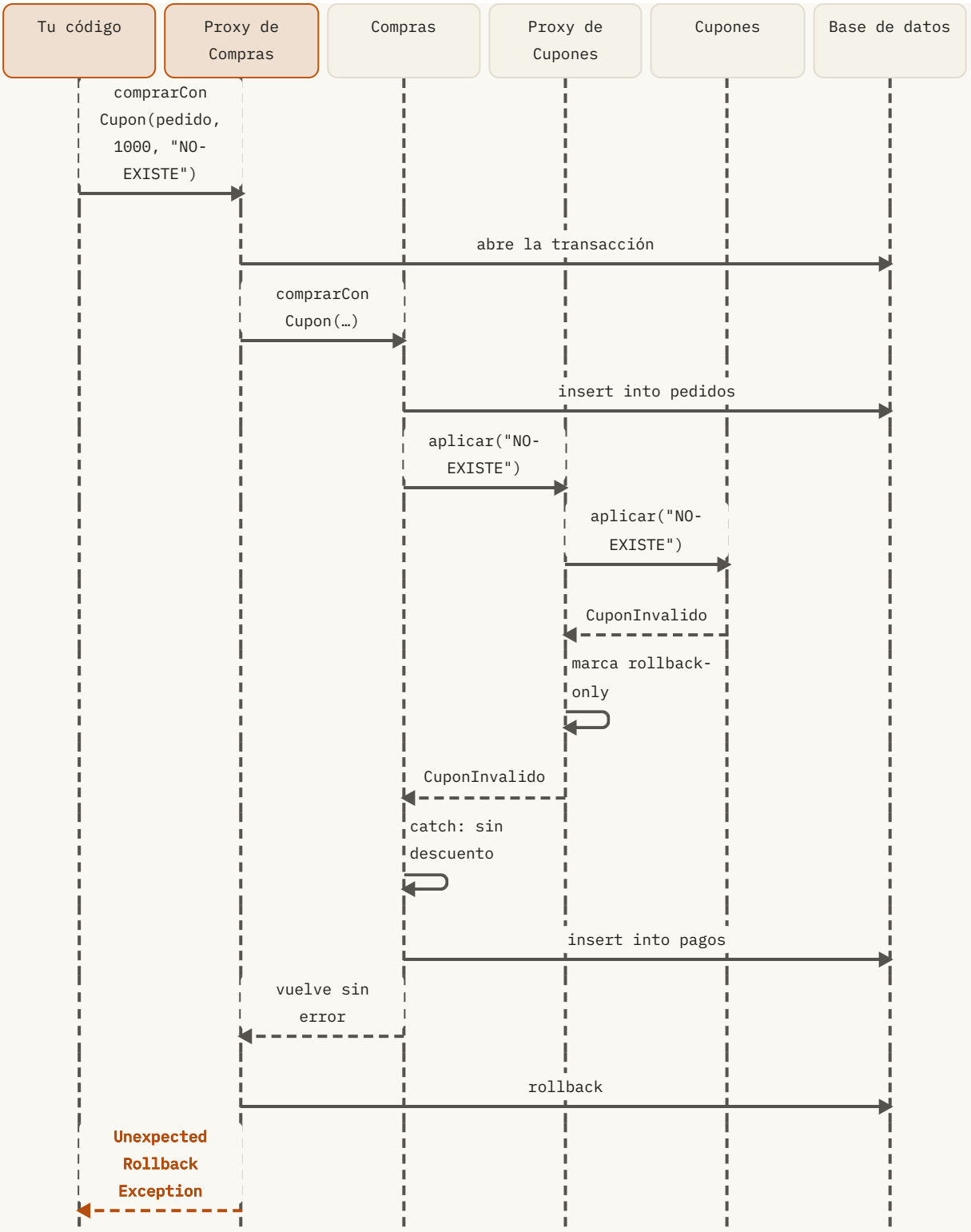
La compra atrapa esa excepción para seguir sin descuento:

Compras.java

```
@Transactional
public void comprarConCupon(Pedido pedido, int monto, String cupon) {
    repositorio.guardar(pedido);
    int descuento = 0;
    try {
        descuento = cupones.aplicar(cupon);
    } catch (CuponInvalido e) {
        // sin cupón, la compra sigue a precio completo
    }
    pagos.cobrar(pedido.numero(), monto - monto * descuento / 100);
}
```

Parece razonable, pero la compra no se guarda y tu código recibe una **`UnexpectedRollbackException`**:

Una excepción atrapada que igual deshace la compra



La llamada entra por el proxy de Compras, porque el método es @Transactional.

No hay ninguna transacción en curso: el proxy abre una y asocia su conexión al hilo. Todo JdbcTemplate sobre este DataSource que corra en este hilo va a usarla.

Recién ahora la llamada llega a tu objeto.

El pedido se inserta, dentro de la transacción. Todavía no es definitivo.

Cupones es otro bean, así que la llamada pasa por su proxy. aplicar también es @Transactional con REQUIRED: como ya hay una transacción, se une a ella. No abre otra.

El objeto real comprueba el cupón.

El cupón no existe: sale una excepción unchecked.

El proxy de Cupones ve salir una RuntimeException de un método transaccional. No puede deshacer la transacción, porque no la abrió él, así que la marca: esta transacción solo puede terminar en rollback.

La excepción sigue su camino hasta Compras.

Compras la atrapa y sigue, como si nada. Pero la marca ya está puesta, y atrapar la excepción no la borra.

El cobro se inserta a precio completo.

El método termina bien. El proxy de Compras, que abrió la transacción, va a confirmarla.

Al confirmar, encuentra la marca de rollback-only. No puede hacer commit: deshace todo, el pedido y el pago.

Y te avisa con una excepción, para que nunca creas que se confirmó algo que se deshizo.

La referencia lo describe así: con **`REQUIRED`**, una marca de rollback-only puesta adentro afecta a la transacción de afuera, y la excepción existe para que quien llamó no crea que hubo commit (Spring Framework Reference, 7.0).

TransaccionesTest.java

```
@Test
void atraparLaExcepcionDeUnMetodoTransaccionalNoSalvaLaCompra() {
    var compras = contexto.getBean(Compras.class);

    assertThatThrownBy(() -> compras.comprarConCupon(PEDIDO, 1_000, "NO-EXISTE"))
            .isInstanceOf(UnexpectedRollbackException.class);

    assertThat(filas("pedidos")).isZero();
    assertThat(filas("pagos")).isZero();
}
```

Si un cupón inválido es un resultado esperado y no un motivo para deshacer la compra, díselo a Spring con **`noRollbackFor`**. Con ese cambio, la compra se guarda a precio completo, y lo comprobé con un test:

CuponesTolerantes.java

```
@Override
@Transactional(noRollbackFor = CuponInvalido.class)
public int aplicar(String codigo) {
    return super.aplicar(codigo);
}
```

La otra salida es que validar un cupón no sea transaccional: sin proxy, nadie marca nada.

Aislamiento y readOnly

Dos transacciones que corren a la vez pueden ver los cambios de la otra en distintos grados. El **aislamiento** (*isolation*) dice cuánto. **`Isolation.DEFAULT`** usa el de tu base de datos; los demás niveles son los de JDBC (Spring Framework API, **`Isolation`**):

Nivel	Lectura sucia	Lectura no repetible	Lectura fantasma
`READ_UNCOMMITTED`	puede pasar	puede pasar	puede pasar
`READ_COMMITTED`	evitada	puede pasar	puede pasar
`REPEATABLE_READ`	evitada	evitada	puede pasar
`SERIALIZABLE`	evitada	evitada	evitada

Una **lectura sucia** es ver un cambio que otra transacción no confirmó; una **no repetible**, leer dos veces la misma fila y obtener valores distintos; una **fantasma**, repetir una consulta y encontrar filas nuevas. Cuanto más estricto el nivel, menos anomalías y más bloqueos.

`readOnly = true` declara que la transacción solo lee, para que el sistema pueda optimizar. Es una **pista**, no una prohibición: no necesariamente hace fallar una escritura, y un transaction manager que no la entienda la ignora sin avisar (Spring Framework API, **`@Transactional`**).

@PostConstruct corre sin transacción

Este bean carga datos al arrancar, desde su **`@PostConstruct`**, llamando a un método **`@Transactional`** que devuelve si hay una transacción activa:

CatalogoInicial.java

```
public class CatalogoInicial {

    private boolean huboTransaccionAlArrancar;

    @PostConstruct
    void alArrancar() {
        huboTransaccionAlArrancar = cargar();
    }

    @Transactional
    public boolean cargar() {
        return TransactionSynchronizationManager.isActualTransactionActive();
    }

    public boolean huboTransaccionAlArrancar() {
        return huboTransaccionAlArrancar;
    }
}
```

Al arrancar no la hay; llamado después, a través del bean, sí:

TransaccionesTest.java

```java
@Test
void elPostConstructCorreSinTransaccion() {
    var catalogo = contexto.getBean(CatalogoInicial.class);

    assertThat(catalogo.huboTransaccionAlArrancar()).isFalse();
    assertThat(catalogo.cargar()).isTrue();
}
```

El **`@PostConstruct`** corre sobre el objeto real, antes de que llegue el proxy (Spring Framework Reference, 7.0), y **`cargar()`** es además una autoinvocación. La referencia lo advierte: no confíes en **`@Transactional`** dentro de un **`@PostConstruct`** (Spring Framework Reference, 7.0).

> Lo que debes llevarte: **`@Transactional`** vive en un proxy, así que valen sus reglas: sin autoinvocación y sin **`@PostConstruct`**. Y por defecto confirma con checked, salvo **`rollbackFor`** o **`rollbackOn = ALL_EXCEPTIONS`**.

Eventos: avisar sin conocer a quién

Cuando se registra un pedido hay que avisarle al cliente, reservar el stock y mañana, quizás, sumar puntos de fidelidad. Si cada interesado es una dependencia del servicio, **`PedidoService`** crece con cada requisito nuevo y hay que editarlo cada vez:

ANTES

PedidoServiceQueAvisaATodos.java

```java
public class PedidoServiceQueAvisaATodos
{

    private final RepositorioPedidos
repositorio;
    private final Notificador
notificador;
    private final Inventario inventario;

    public
PedidoServiceQueAvisaATodos(RepositorioPe
didos repositorio, Notificador
notificador,

Inventario inventario) {
        this.repositorio = repositorio;
```

DESPUÉS

PedidoServiceConEventos.java

```java
public class PedidoServiceConEventos {

    private final RepositorioPedidos
repositorio;
    private final
ApplicationEventPublisher eventos;

    public
PedidoServiceConEventos(RepositorioPedido
s repositorio, ApplicationEventPublisher
eventos) {
        this.repositorio = repositorio;
        this.eventos = eventos;
    }
```

```
        this.notificador = notificador;
        this.inventario = inventario;
    }

    public void registrar(Pedido pedido)
{
        repositorio.guardar(pedido);

notificador.enviar(pedido.correoCliente()
, "Recibimos tu pedido " +
pedido.numero());

inventario.reservar(pedido.numero());
    }
}
```

```
    public void registrar(Pedido pedido)
{
        repositorio.guardar(pedido);
        eventos.publishEvent(new
PedidoRegistrado(pedido));
    }
}
```

A la derecha, el servicio solo **anuncia** lo que pasó: publica un evento, que puede ser cualquier objeto. Desde Spring 4.2 no hace falta heredar de **`ApplicationEvent`** (Spring Framework Reference, 7.0), así que alcanza con un **`record`**:

PedidoRegistrado.java

```
public record PedidoRegistrado(Pedido pedido) {}
```

Cada interesado lo escucha con **`@EventListener`** (Spring Framework Reference, 7.0):

AvisoAlCliente.java

```
public class AvisoAlCliente {

    private final Notificador notificador;

    public AvisoAlCliente(Notificador notificador) {
        this.notificador = notificador;
    }

    @EventListener
    public void alRegistrarse(PedidoRegistrado evento) {
        var pedido = evento.pedido();
        notificador.enviar(pedido.correoCliente(), "Recibimos tu pedido " +
pedido.numero());
    }
}
```

El contenedor inyecta el **`ApplicationEventPublisher`** y, al arrancar, busca los **`@EventListener`** de sus beans.

EventosTest.java

```
@Configuration
static class ConfiguracionEventos {

    @Bean
    RepositorioPedidos repositorioPedidos() {
        return new RepositorioEnMemoria();
    }

    @Bean
    PedidoServiceConEventos pedidoService(RepositorioPedidos repositorio,
ApplicationEventPublisher eventos) {
        return new PedidoServiceConEventos(repositorio, eventos);
    }

    @Bean
    NotificadorFalso notificador() {
        return new NotificadorFalso();
    }

    @Bean
    AvisoAlCliente avisoAlCliente(Notificador notificador) {
        return new AvisoAlCliente(notificador);
    }

    @Bean
    Inventario inventario() {
        return new Inventario();
    }

    @Bean
    ReservaDeStock reservaDeStock(Inventario inventario) {
        return new ReservaDeStock(inventario);
    }
}
```

EventosTest.java

```
@Test
void elServicioPublicaYCadaInteresadoEscucha() {
    try (var contexto = new
AnnotationConfigApplicationContext(ConfiguracionEventos.class)) {
        contexto.getBean(PedidoServiceConEventos.class).registrar(new Pedido(42,
"ana@example.com"));

        assertThat(contexto.getBean(NotificadorFalso.class).enviados())
                .containsExactly("ana@example.com: Recibimos tu pedido 42");
        assertThat(contexto.getBean(Inventario.class).reservados()).containsExactly(42);
    }
```

```
}
```

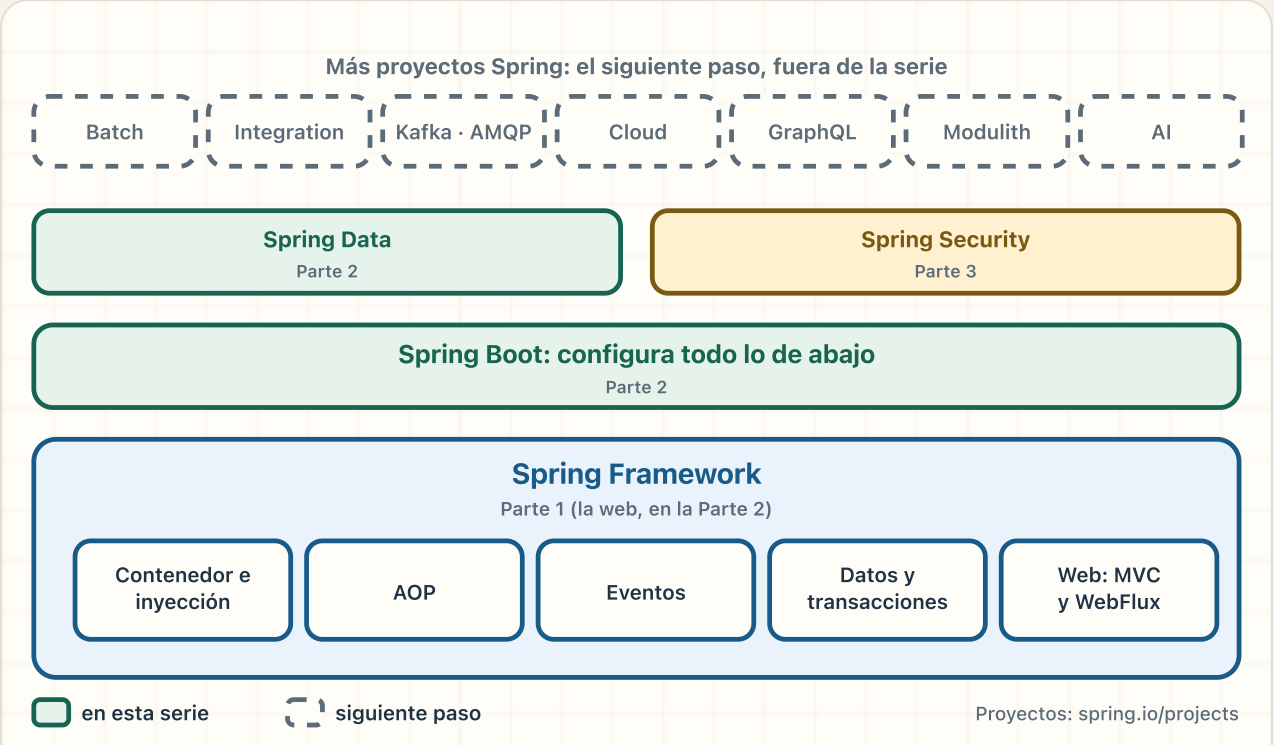
Sumar un interesado es sumar un bean con su **`@EventListener`**, sin tocar el servicio.

Por defecto los listeners corren **de forma síncrona**: **`publishEvent`** no vuelve hasta que terminaron todos, en el mismo hilo y dentro de la transacción del que publica (Spring Framework Reference, 7.0); lo del hilo lo comprobé con un test. Con **`@Async`** (y **`@EnableAsync`** en la configuración) el listener corre aparte, pero si falla, la excepción no le llega a quien publicó.

> **El error típico:** que el listener mande el correo dentro de la transacción del que publica. Si después la transacción se deshace, el pedido no existe, pero el correo ya salió. Para avisar solo si la transacción se confirmó existe **`@TransactionalEventListener`**, en la Parte 3.

El mapa del ecosistema

Todo lo de esta parte es Spring Framework. Sobre esa base se construye el resto:



Un solo contenedor en la base; Boot lo configura; cada proyecto resuelve un problema encima.

- **Spring Framework** incluye también la web: Spring MVC, sobre la API de Servlet, y desde 5.0 Spring WebFlux, reactivo (Spring Framework 5.0 goes GA).

- **Spring Boot** configura todo lo anterior con valores razonables por defecto. Es la Parte 2.
- **Spring Data** da un mismo modelo de acceso a datos para bases relacionales y no relacionales; **Spring Security**, autenticación y autorización. Aparecen en las Partes 2 y 3.
- El resto queda como siguiente paso, y todo se apoya en lo que aprendiste acá: **Spring Batch** (procesos por lotes), **Spring Integration** (patrones de integración), **Spring for Apache Kafka** y **Spring AMQP** (mensajería), **Spring Cloud** (sistemas distribuidos), **Spring for GraphQL**, **Spring Modulith** (módulos de dominio) y **Spring AI** (modelos de IA) (spring.io, *Projects*).

Lo que solo enseña la experiencia

Estas trampas suelen aprenderse con un incidente:

- **Estado en un singleton.** Un campo que guarda algo de un cliente lo comparten todos los requests.
- **Dependencias circulares.** Por constructor, el contenedor no arranca, y está bien: el ciclo es un problema de diseño. Por campo, en Spring sin Boot, arranca y lo esconde.
- **Constructores largos.** Si un constructor pide siete dependencias, la clase hace demasiado. La referencia de Spring lo llama un *code smell* (Spring Framework Reference, 7.0).
- **Un prototype dentro de un singleton** se inyecta una sola vez: **`ObjectProvider`**.
- **Una errata en `@Value`** llega como texto: resolvedor estricto o valores por defecto.
- **Rollback con una excepción checked:** no lo hay, salvo con **`rollbackFor`** o con **`rollbackOn = ALL_EXCEPTIONS`** para toda la aplicación.
- **Atrapar la excepción de un método transaccional** no salva la transacción si se unió a la tuya.
- **La autoinvocación y los métodos `final`** se saltan el proxy, y con él la transacción o el aspecto.

Si te llevas una sola idea, que sea esta: lo que el contenedor te entrega no siempre es tu objeto. A veces es un proxy, y casi todas las sorpresas de Spring salen de ahí.

¿Cuál de estas trampas te costó más tiempo encontrar la primera vez? Y si todavía no te pasó ninguna, ¿cuál apostarías que está escondida hoy en tu proyecto?

Sigue con la Parte 2: Spring Boot, de la primera app a una API completa, donde todo esto lo configura Spring Boot por ti, y vas a saber qué está haciendo.

Referencias

- Spring Framework Reference, 7.0.10: Overview, The IoC Container, Dependency Injection, Bean Scopes, Customizing the Nature of a Bean, Container Extension Points, The BeanFactory API, Classpath Scanning, Environment Abstraction, Using @Value, Standard and Custom Events, DAO

Support, JDBC Core Classes, AOP Concepts, Proxying Mechanisms, Using @Transactional, Transaction Propagation, Rolling Back.

- Spring Framework API, 7.0.10: **`@Bean`**, **`@ComponentScan`**, **`@Transactional`**, **`Isolation`**.
- Blog de Spring: Spring Framework 1.0 Final Released (2004), Spring Framework 2.5 Released (2007), Spring Framework 3.0 goes GA (2009), Spring Boot 1.0 GA Released (2014), Spring Framework 5.0 goes GA (2017), Spring Framework 6.0 goes GA (2022), Spring Framework 7.0 General Availability (2025), Spring Boot 4.0.0 available now (2025).
- Rod Johnson, Spring Framework: The Origins of a Project and a Name (2006).
- Martin Fowler, Inversion of Control Containers and the Dependency Injection pattern (2004) e InversionOfControl (2005).
- Java EE 6 API: **`SessionBean`** y **`EJBHome`**; The J2EE 1.4 Tutorial: The Contents of an Enterprise Bean.
- spring.io, Projects; Apache Maven Wrapper; IntelliJ IDEA as a unified product.