Spring Framework: The Container That Builds Your App

Part 1 of 3 · November 4, 2026 · spring · java · dependency-injection · fundamentals · spring-series

> **Part 1 of 3** in the series "Spring, from Beginner to Expert". Continue with part 2: Spring Boot, from your first app to a complete API and part 3: Spring Boot in production.

Most of us learn Spring backwards: first Spring Boot, which starts an application with one annotation, and then, the day something breaks, we discover there was a container, proxies and transactions underneath that nobody ever explained to us.

This series goes in order. There's no Spring Boot in this first part: just Spring Framework on its own, the container that builds your application, with every term defined when it shows up. All you need is to know how to program in Java.

I compiled and tested all the code with Spring Framework 7.0.10: I checked every behavior I describe with a test, and the outputs you see are the ones I got.

▼ IN THIS ARTICLE

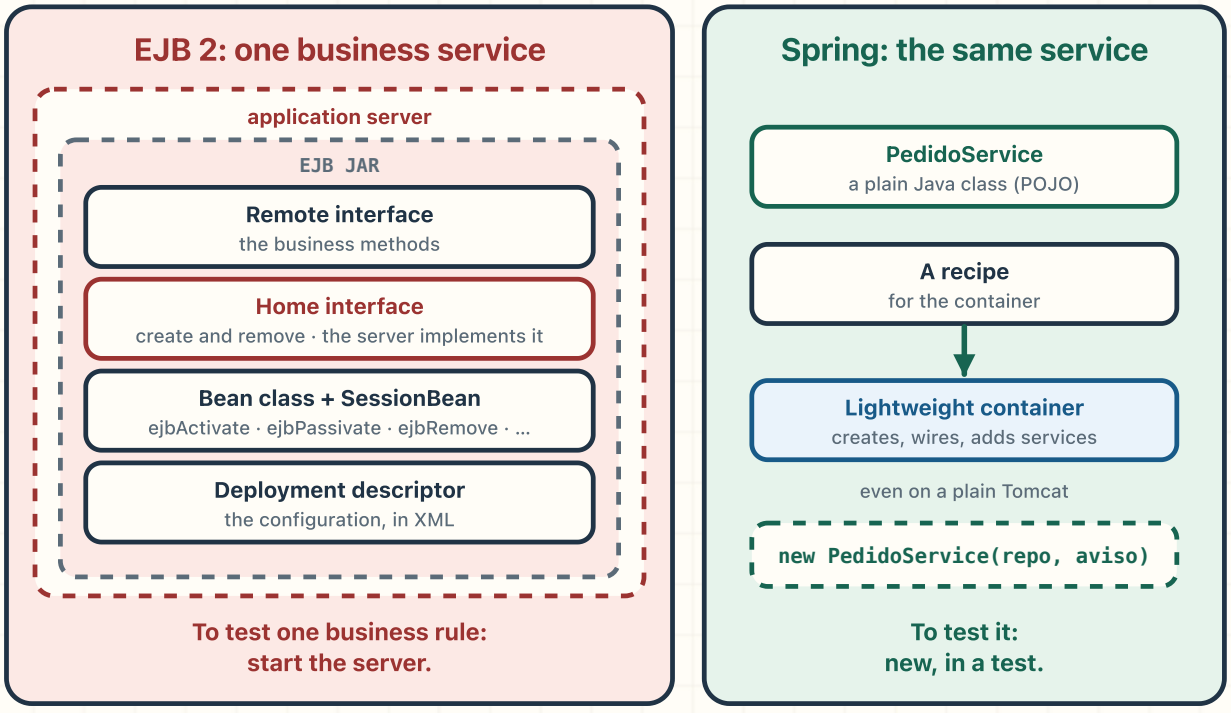
01. Why Spring exists
02. What you'll learn and in what order
03. Your working environment
04. Inversion of control and dependency injection
05. The patterns behind Spring
06. Configuring the container: XML, scanning or classes
07. A bean's lifecycle
08. Properties: configuration outside the code
09. Exceptions, the Spring way
10. Proxies and AOP: the code that repeats everywhere
11. Transactions
12. Events: notify without knowing who
13. The ecosystem map
14. What only experience teaches
15. References

Why Spring exists

Picture this: it's 2002 and you're asked to build the part of the system that records an order. In enterprise Java, the standard answer was J2EE (*Java 2 Platform, Enterprise Edition*), Java's set of specifications for enterprise applications, and the business logic went into an **EJB** (*Enterprise JavaBean*).

An EJB with remote access wasn't one class. It was several pieces the specification required (The J2EE 1.4 Tutorial):

- a **remote interface**, with the business methods;
- a **home interface**, to create and remove the bean, which **the server implemented** (Java EE 6 API, **`EJBHome`**);
- the **bean class**, which also implemented **`SessionBean`**: four methods the server used to notify it about its lifecycle (Java EE 6 API, **`SessionBean`**);
- a **deployment descriptor** in XML;
- and all of it packaged into an *EJB JAR*, deployed on an application server.



A business service before and after Spring: from five pieces tied to the server to a plain class you test with new.

Since the server implemented the home interface, testing a business rule meant starting the server.

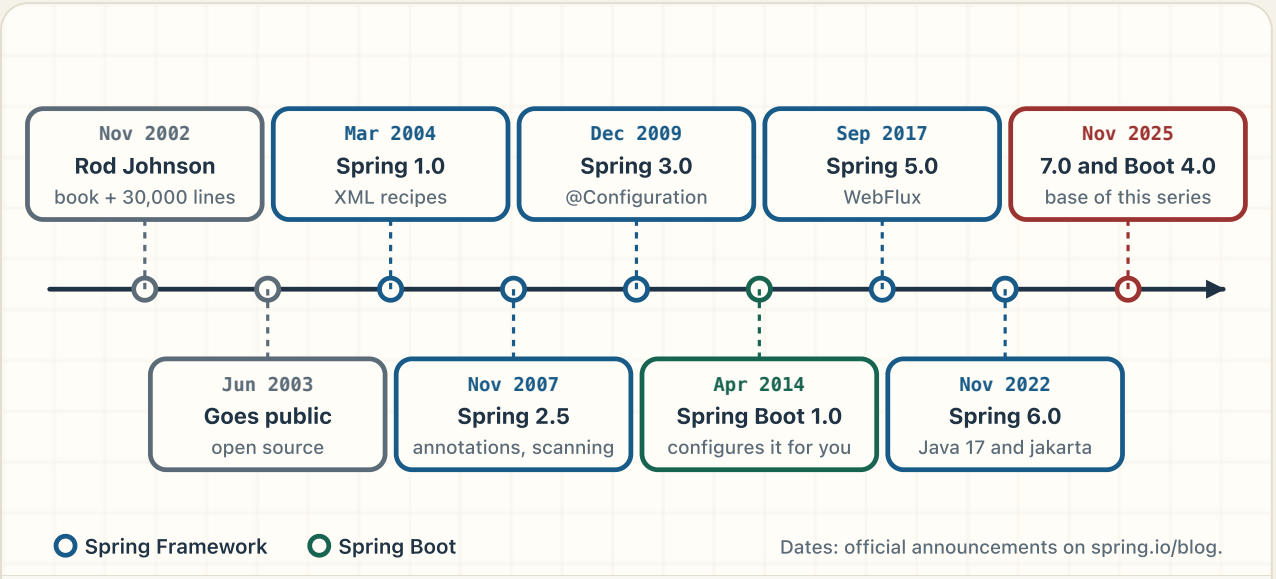
In November 2002, Rod Johnson published *Expert One-on-One J2EE Design and Development*, a book that came with about 30,000 lines of framework code. His readers convinced him to turn that code into an open source project, which went public in June 2003. The name, Spring, signaled a fresh start after the "winter" of traditional J2EE (Rod Johnson, *The Origins of a Project and a Name*).

The idea fits in one sentence. Your business logic goes into **POJOs** (*Plain Old Java Objects*): ordinary Java classes that don't extend or implement anything from the framework. And a lightweight **container** creates them, wires them together and adds from the outside what the server used to demand, such as transactions.

Spring 1.0 shipped on March 24, 2004 with exactly that: a container with setter and constructor injection, AOP, a layer over JDBC and declarative transactions "applicable to any POJO", on a plain Tomcat (Spring Framework 1.0 Final Released). Today's reference sums it up: Spring was born "in response to the complexity of the early J2EE specifications" (Spring Framework Reference, 7.0).

Why it's still relevant

Every generation of Spring kept the container and the POJOs, and changed how you write the recipes and what surrounds them:



More than twenty years with the same idea: the way you write the recipes changes, the container doesn't.

The dates come from the official announcements on the Spring blog. Spring Framework 7.0, released on November 13, 2025, is the foundation of Spring Boot 4.0, which shipped a week later.

> **Note:** On this site, the backend is a Spring Boot 4.1.2 application, built on the same Spring Framework 7 this guide covers.

What you gain and what you pay

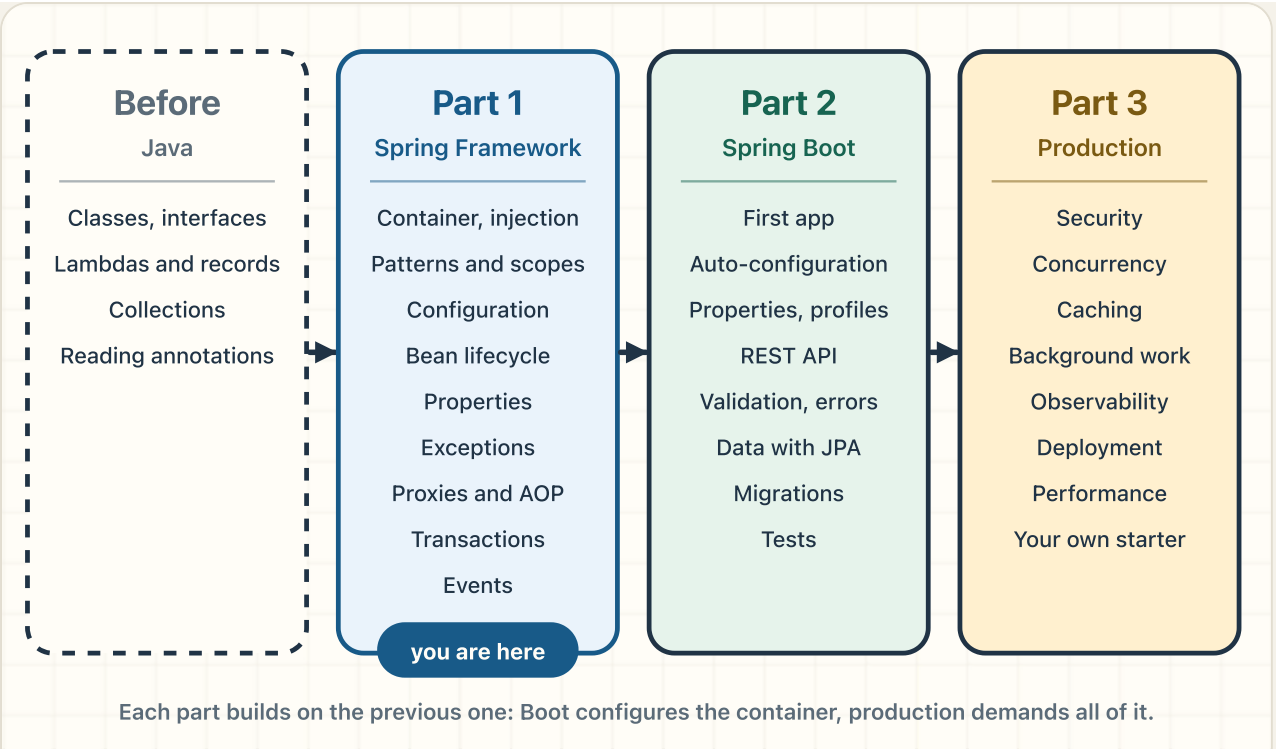
What you gain:

- **Your code doesn't depend on the framework.** A class that receives its dependencies in the constructor can be tested with `new`, with no server and no Spring.
- **Cross-cutting concerns are declared.** A transaction is an annotation on a method, not a block of open, commit and roll back in every one of them.
- **Individual standards, not the whole platform.** Spring integrates selected specifications, such as Servlet, JPA, Bean Validation or JMS (Spring Framework Reference, 7.0).

What you pay:

- **A learning curve.** Container, scopes, proxies, propagation: lots of concepts before you can explain why something fails.
- **The "magic".** Part of the behavior lives in objects the container puts in front of yours, and when it fails, it fails in strange ways. I reproduce that with tests in the section on proxies.
- **Startup.** The container does its work at startup, so that errors show up early. That takes time, and Part 3 measures it.
- **Major migrations.** Spring 6.0 dropped the `javax` packages for `jakarta` and raised the minimum to Java 17 (Spring Framework 6.0 goes GA). Moving to a new generation is planned work.

What you'll learn and in what order



First the container, then Boot, which configures it, and finally what production demands.

First the container without Spring Boot; then Boot, which doesn't replace anything from Part 1 but configures it for you; and finally, what production demands.

Before you start, you should be comfortable with:

- **Object-oriented Java:** classes, interfaces and implementations, and reading an annotation like `@Override`.
- **Lambdas and records.** If you're missing them, they're in the guide to Java through its LTS releases.
- **Collections**, and the difference between an interface and its implementation, which is the foundation of dependency injection: Java Collections: List Is Not ArrayList.

Your working environment

To follow the guide, you'll build your own project with the `pom.xml` and the classes I show.

A JDK 21

Spring Framework 7 requires Java 17 at minimum (Spring Framework 7.0 General Availability); in this series I use Java 21, the version this site's backend runs on. To check which one you have:

```
java -version
```

If the first line doesn't say 21 or higher, install a JDK 21 from any vendor, for example Eclipse Temurin.

An IDE

- **IntelliJ IDEA**: since 2025.3 it's a single product, with no Community or Ultimate edition, and the core features are free (JetBrains, *IntelliJ IDEA as a unified product*). It comes with Maven built in.
- **Visual Studio Code** with Microsoft's **Extension Pack for Java**.

A Maven project

Maven downloads the project's libraries (its **dependencies**), compiles the code and runs the tests. Create a new Maven project from your IDE. You don't need to install Maven if the project includes the **Maven Wrapper**: two scripts that download the version pinned in **`.mvn/wrapper/maven-wrapper.properties`** and run it from there (Apache Maven Wrapper). You generate it once with **`mvn wrapper:wrapper`**, from the IDE, and from then on:

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

Spring is split into modules (**`spring-context`**, **`spring-jdbc`**, **`spring-tx`**...) that must all be on the same version. That's what a **BOM** (*Bill of Materials*) is for: a list of versions you import once in the **`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>
```

With the BOM, Spring dependencies go without a **`<version>`**. To get started, one is enough:

pom.xml

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

7.0.10 is the version that ships with Spring Boot 4.1.2, the one this site's backend uses. The second BOM pins JUnit's version; for the tests, add **`junit-jupiter`** and **`assertj-core`** with **`<scope>test</scope>`**. The database dependencies come in the section on exceptions.

Your first app with the container

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());
        }
    }
}
```

I take the pieces apart in the next section. Run the **`main`** from your IDE: if **`Hola, Spring`** shows up in the console, your environment is ready.

Inversion of control and dependency injection

There's a question worth asking about any class you write: who decides which objects it works with? It sounds like a philosophy question. It's the question that separates code you can change and test from code you can't, and it's the reason Spring exists.

Let's answer it with the order store that comes with us through the whole series. When someone buys, **`PedidoService`** saves the order and notifies the customer by email. The code keeps the Spanish names I tested it with: a **`Pedido`** is an order, a **`Notificador`** is a notifier and **`registrar`** means to record. The first version almost all of us write is this one:

PedidoServiceAcoplado.java

```
public class PedidoServiceAcoplado {

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

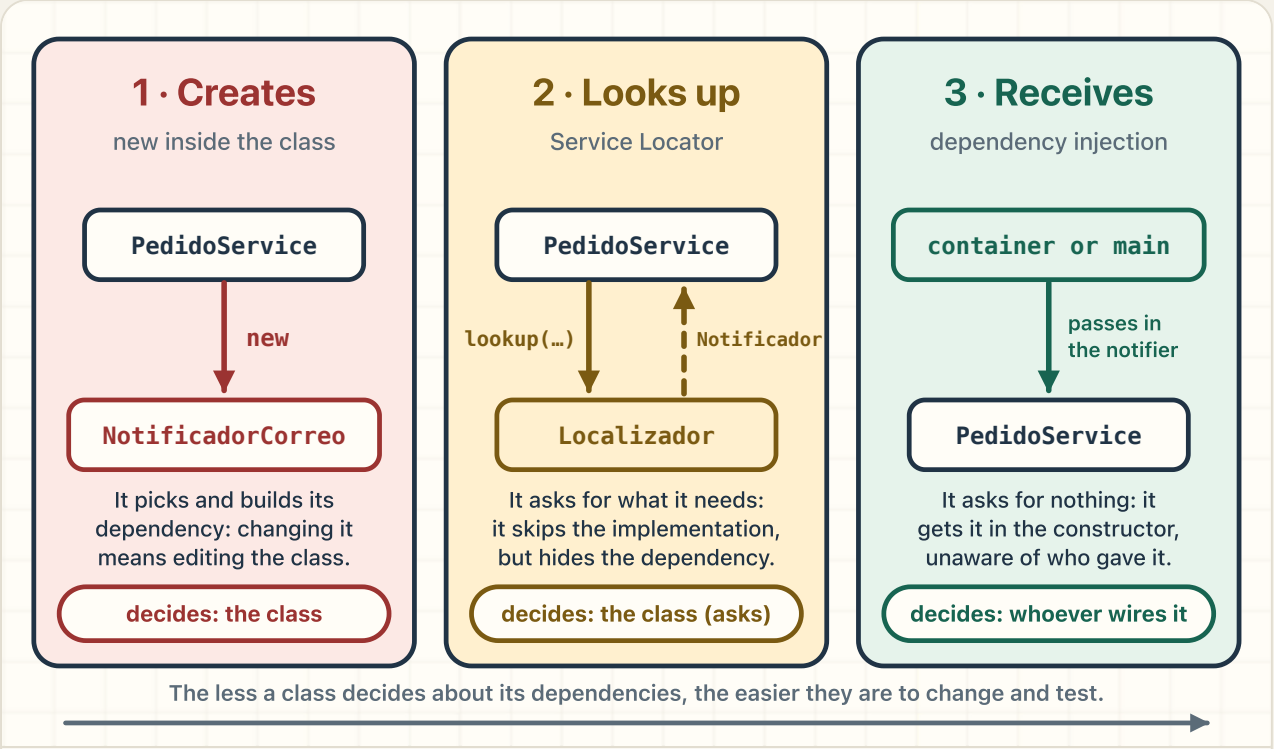
    public void registrar(Pedido pedido) {
        repositorio.guardar(pedido);
        notificador.enviar(pedido.correoCliente(), "Recibimos tu pedido " +
pedido.numero());
    }
}
```

It works. And it has a problem you don't see until you try to test it: **the class decided on its own** that it notifies by email, with **`NotificadorCorreo`**, and built that notifier itself with **`new`**. There's no way to call **`registrar()`** without running the real notifier. I checked it with a test: testing the service sends the email, every time.

If tomorrow you want to notify by SMS, you have to open this class and edit it. If you want to test it without sending emails, you can't. The **`new`** tied you to a concrete implementation.

Three ways to get a dependency

A **dependency** is any object a class needs to do its job. **`PedidoService`** needs two: somewhere to save and some way to notify. There are three ways to get them, and what changes between them is **who decides**:



What changes between the three isn't the method's code, it's who picks the notifier.

We already saw the first one: the class **creates** what it needs. The second is a pattern with a name of its own, the *Service Locator*: a global registry where each class goes to **look up** what it needs.

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());
    }
}
```

It's an improvement: the service no longer knows the notifier sends emails, it just asks for "a **`Notificador`**", and in a test you can register a fake one. But the constructor is empty: from the outside, nobody knows this class needs a repository and a notifier. If you forget to register one, you only find out when you run it, with a **`NullPointerException`** in the middle of **`registrar()`**. The dependency is still there, just hidden.

The third way is the one that matters. The class **receives** what it needs in the constructor and decides nothing:

BEFORE

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());
    }
}
```

AFTER

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());
    }
}
```

Now **`PedidoService`** works with any **`RepositorioPedidos`** and any **`Notificador`**: interfaces, not concrete classes. And the constructor says out loud what the class needs in order to exist.

What inversion of control is

In the first two versions, the class is in control: it picks its notifier, either by building it or by asking for it. In the third, that control has been **inverted**: whoever builds the class holds it. Hence the name.

Inversion of control (IoC) is a more general principle than this example: instead of your code calling the framework, the framework calls your code. It's also known as the Hollywood principle: "don't call us, we'll call you" (Martin Fowler, *InversionOfControl*). You've already used it, even if you never named it: when you write a **`@Test`** method, you never write the **`main`** that runs it. JUnit finds it and calls it when it's time. That's precisely the difference between a library, which you call, and a framework, which calls you.

Dependency injection (DI) is one concrete way to apply that principle to dependencies: the class declares them in its constructor and someone from the outside **injects** them when creating it. The Spring reference defines it like this: objects declare their dependencies only through constructor arguments, arguments to a factory method, or properties set after the object is constructed, and the container injects them, which is the inverse of the object itself controlling their creation or lookup "by using direct construction of classes or a mechanism such as the Service Locator pattern" (Spring Framework Reference, 7.0).

> **The usual mistake:** thinking the Service Locator is a form of dependency injection because "there's no **`new`** anymore". It isn't: the class is still **asking**. With injection it asks for nothing, and that's why it doesn't depend on the locator either.

Dependency injection without Spring

Dependency injection is a pattern, not a library. Somebody has to do the **`new`**, and in an app without a framework that somebody is **`main`**:

AppManual.java

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

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

And the test, which was impossible before, is now trivial. You build the service with a fake notifier that, instead of sending emails, writes down what it was asked to send:

PedidoServiceTest.java

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

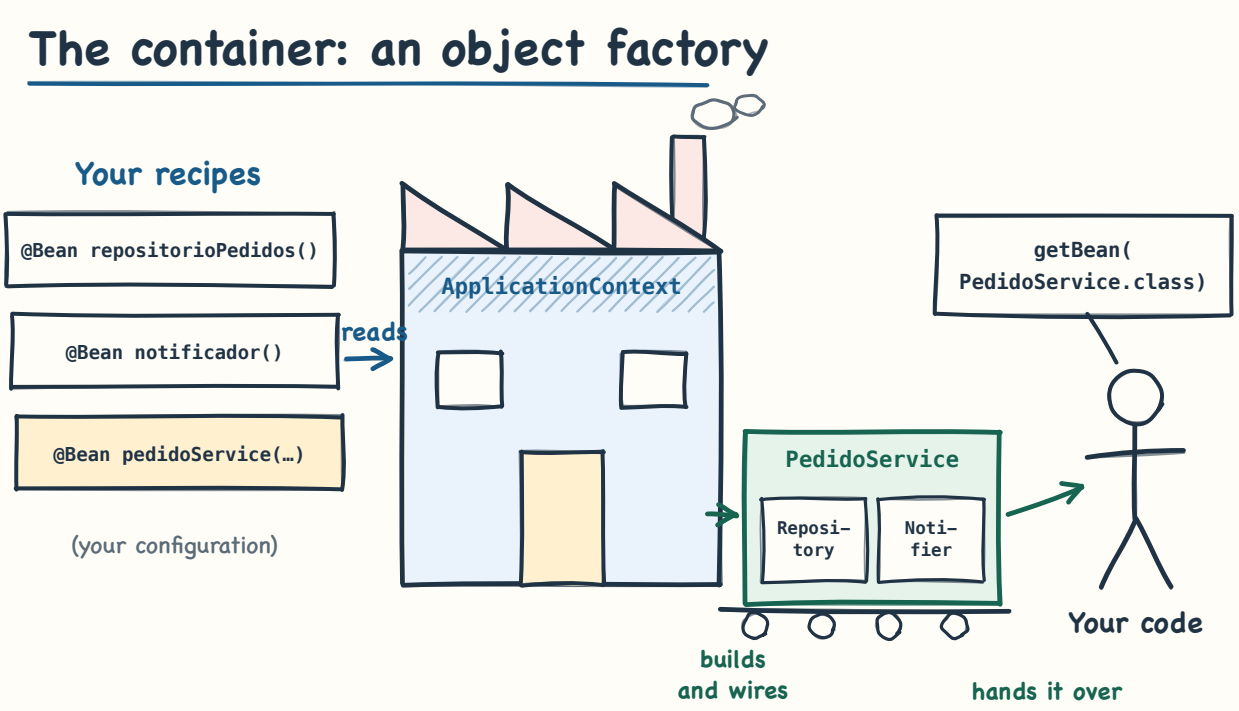
    pedidos.registrar(new Pedido(42, "ana@example.com"));
```

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

No Spring, no mail server, in milliseconds.

So why do we want Spring? With three classes, that **`main`** is three lines. In a real application it's hundreds of classes, and wiring them by hand becomes a program of its own: you have to create them in the right order, decide which ones are shared and which aren't, close connections on shutdown, swap pieces depending on the environment. That work is exactly what the Spring container does.

The container: a factory that follows your recipes



You write the recipes; the container builds each piece, wires them together and hands you the finished object.

The Spring **container** is the object that creates, wires and manages your application's objects. In code you'll see it as **`ApplicationContext`**. Each object the container manages is called a **bean**: an ordinary Java object, except the container created it instead of you.

For the container to know what to create, you give it recipes. They're the same three lines from the hand-written **`main`**, each in its own method:

```
ConfiguracionPedidos.java
```

```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` marks the class as a recipe book, and each **`@Bean`** is a recipe: whatever the method returns is the bean. If a recipe has parameters, like **`pedidoService(...)`**, the container looks among its beans for one of each type and passes it in. That's the injection.

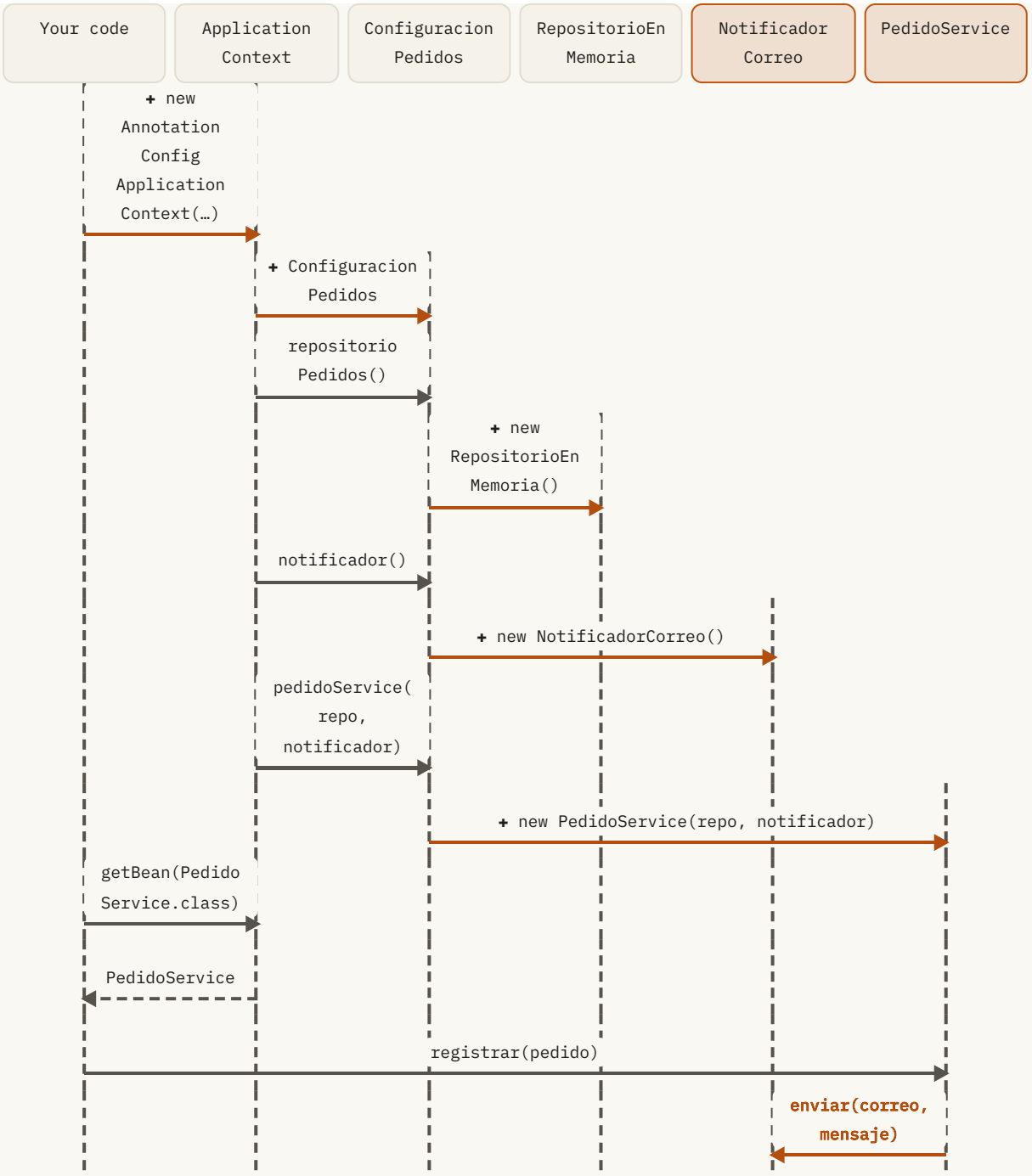
And this is how you use it:

AppConContenedor.java

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

This is what happens, message by message, between the first line and the email:

From new AnnotationConfigApplicationContext to the email being sent



Your code creates the container and passes it the configuration class. No bean exists yet.

The recipe book is a bean too: the first of your beans the container creates is ConfiguracionPedidos itself.

Now it follows the recipes. The first one has no parameters: it calls it as is.

The method does the new and returns the object. The container stores it as the repositorioPedidos bean.

Second recipe, same path.

There are two beans now: a repository and a notifier.

The third recipe asks for a RepositorioPedidos and a Notificador. The container looks among its beans by type and passes in the two it just created. This is the injection.

PedidoService is born with its dependencies inside. All of this happened at startup, before your code asked for anything.

Only now does your code ask for the service. Nothing gets created: the container already had it built.

It hands it back, ready to use.

From here on it's plain Java: you call a method on an object.

And the service uses the notifier it received, without knowing who picked it or who created it.

Two details of that sequence aren't obvious. The first is that the container creates the singleton beans **at startup**, not when you ask for them. It does that on purpose: if a recipe is wrong, you'd rather find out when the application starts than at three in the morning, when someone asks for that bean for the first time (Spring Framework Reference, 7.0). I checked it with a test.

The second detail is what **doesn't** appear: **`PedidoService`** doesn't import anything from Spring. It's the same class we wired by hand in **`main`** and in the test. That's why changing the wiring doesn't touch the class. Another recipe book is enough, for example a test one with the fake notifier:

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);
    }
}
```

Through the constructor, not the field

There's another way to ask Spring for dependencies that you'll find in a huge amount of code, because it's shorter: annotating the fields with **`@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());
    }
}
```

Inside the container it works: Spring creates the object and then fills in the fields through reflection. Outside of it, a test that creates the class with **`new`** finds the fields set to **`null`**:

PedidoServiceConCampoTest.java

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

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

With constructor injection that can't happen: the compiler won't let you create the object without its dependencies. It's what the Spring team recommends, for three reasons (Spring Framework Reference, 7.0):

- **Fields can be `final`**: the object is immutable, and nobody swaps its notifier halfway through.
- **Required dependencies are never `null`**: if one is missing, the object doesn't get created.
- **The object comes out complete**: whoever receives it never sees one that's half built.

Also, if a class the container creates on its own has a single constructor, you don't need to annotate it with **`@Autowired`**: Spring uses it on its own. **`PedidoService`** doesn't even need that, because your recipe creates it with **`new`**; you'll see the rule at work in **`ServicioEscaneado`**, further down.

> What to take with you:
>
> - **Inversion of control**: the framework calls your code, not the other way around. With dependencies, it means the class stops choosing them.
> - **Dependency injection**: the class declares what it needs in the constructor and someone from the outside provides it. The Service Locator isn't that: there, the class is still asking.

- **Container and bean:** the `ApplicationContext` creates, wires and manages your objects; each of them is a bean, and by default the singletons are created at startup.
- **Through the constructor:** `final` fields, nothing `null`, complete objects, and classes you can test without Spring.

> – Going deeper
>
> the name "dependency injection" was settled on in 2004 by Martin Fowler and the authors of IoC containers, precisely because he found "inversion of control" too generic a term (Martin Fowler, *Inversion of Control Containers and the Dependency Injection pattern*), where he also compares it with the Service Locator.
>
> The container resolves the parameters of a constructor or a `@Bean` method by type. When there's more than one candidate of the same type you need to disambiguate, and we'll get to that later, with `@Qualifier` and `@Primary`.
>
> A constructor cycle, where A needs B and B needs A, has no solution: neither can be created first. The container detects it at startup and throws `BeanCurrentlyInCreationException`. In Spring without Boot, the same cycle through fields or setters does start, and that's why it hides a design problem the constructor forces you to see.
>
> Setter injection still has its place, but only for **optional** dependencies that have a reasonable default value inside the class.

The patterns behind Spring

Spring uses classic patterns, the ones from the *Design Patterns* book known as "GoF" (*Gang of Four*). Careful: the book's pattern and Spring's version sometimes share a name without being the same thing.

Factory: the container is a factory

A **factory** is an object whose job is to create other objects, so whoever uses them doesn't need to know how they're built. Spring has two levels:

- `BeanFactory`, the basic interface: it creates beans and wires them.
- `ApplicationContext`, which extends it and adds AOP, events, internationalized messages and web contexts (Spring Framework Reference, 7.0).

A plain **`BeanFactory`** doesn't detect **`BeanPostProcessor`**s, the pieces that process the annotations and create the proxies (Spring Framework Reference, 7.0). I checked it with a test: a **`DefaultListableBeanFactory`** only creates the bean when you ask for it and doesn't run its **`@PostConstruct`**; an **`ApplicationContext`** creates it at startup and does run it. That's why the official recommendation is to use an **`ApplicationContext`** unless you have a good reason not to.

Each **`@Bean`** method is also a small factory, a **factory method**: Spring 3.0 introduced them as "annotated factory methods" (Spring Framework 3.0 goes GA). And when creating a bean takes logic of its own, there's **`FactoryBean<T>`**, a bean whose job is to make another one:

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;
    }
}
```

When you ask for it, the container doesn't give you the factory but what it makes. To get the factory itself, you put **`&`** in front of the name (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: one per container, not one per class

The book's **Singleton** is a class that guarantees on its own that only one instance exists: it hides its constructor and keeps the instance in a static field. In Spring, the class is an ordinary one, and the container is what decides there's only one, because **singleton** is every bean's default scope:

BEFORE

CatalogoGlobal.java

```
public final class CatalogoGlobal {

    private static final CatalogoGlobal
INSTANCIA = new CatalogoGlobal();

    private CatalogoGlobal() {}

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

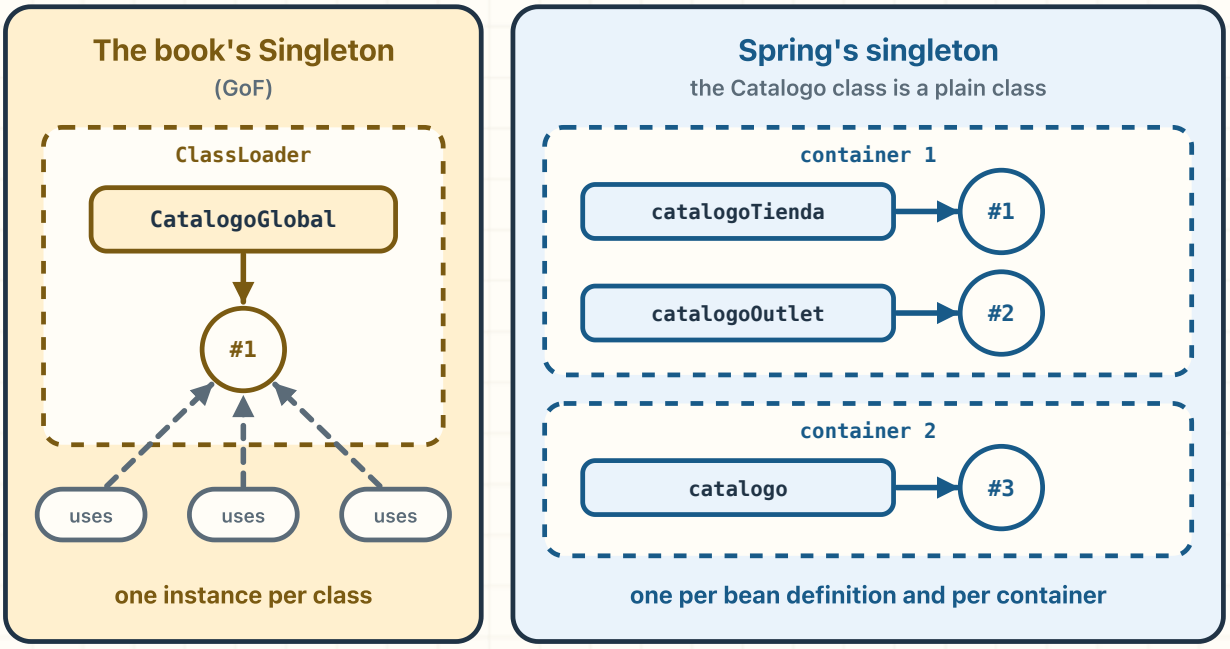
AFTER

ConfiguracionCatalogo.java

```
@Configuration
public class ConfiguracionCatalogo {

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

What "only one" means changes too. The book's is one per *ClassLoader* (the JVM component that loads classes): in practice, one per class. Spring's is one **per container and per bean definition** (Spring Framework Reference, 7.0).



The book's Singleton is one per class; Spring's is one per recipe and per container.

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));
    }
}
```

What doesn't change is that the singleton is **shared**: everyone who asks for it gets the same object, and in a web application that means every request at once. That's why the reference's rule is singleton for stateless beans and prototype for stateful ones (Spring Framework Reference, 7.0).

> **The usual mistake:** storing something that belongs to one customer in a singleton's field. This calculator remembers the discount of whoever used it last:
>
> 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 and Luis get the same object; Luis applies his discount after Ana, and Ana pays without hers:
>
> 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);
>     }
> ```

```
}
```

The fix is to hold no state: the discount comes in as a parameter of **`total`**.

Prototype: a new one every time

An order draft, the list a customer puts together before confirming, has to be one per customer. That's what the **prototype** scope is for: every time you ask for the bean, the container runs the recipe and gives you a new one.

ConfiguracionAlcances.java

```
@Configuration
public class ConfiguracionAlcances {

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

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

Three requests to a singleton and to a prototype



At startup, the container creates the CalculadoraEnvio singleton and keeps it. For the prototype it only has the recipe: no BorradorPedido exists yet.

Customer A asks for the calculator. Nothing gets created: the container hands over the one it already had.

Customer B asks for the same thing and gets the same object, #1.

And so does customer C. Three requests, a single object: that's why a singleton can't hold anything that belongs to one customer.

Now customer A asks for a BorradorPedido. With prototype, the container runs the recipe right then and hands over a new object.

Customer B asks for another one and gets another one: #2. Whatever A adds to their draft, B doesn't see.

Customer C gets #3. Notice that the container kept none of them: it handed them over and forgot about them.

When the container closes, Spring destroys the singleton it manages. It doesn't touch the drafts: cleaning them up is the job of whoever asked for them.

In the test, **`isSameAs`** checks that two references point to the same object, not that they're equal:

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);
    }
}
```

I checked the last step with a test too: Spring hands over a prototype and never sees it again, so **it doesn't call its destruction methods** (Spring Framework Reference, 7.0).

The prototype-inside-a-singleton trap

This cash register, a singleton, receives a draft in its constructor:

CajaConBorradorFijo.java

```
@Component
public class CajaConBorradorFijo {

    private final BorradorPedido borrador;

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

    public BorradorPedido nuevoPedido() {
```

```
        return borrador;
    }
}
```

The container resolves a bean's dependencies **once, when it creates it** (Spring Framework Reference, 7.0). The register is created only once and gets only one draft; it doesn't matter that the draft is a prototype. Luis finds Ana's book in his order:

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");
    }
}
```

The fix is to receive something that knows how to ask for a new draft every time: **`ObjectProvider<T>`**, which asks the container for the bean on every **`getObject()`**. With **`Caja`**, Luis's order starts empty.

BEFORE

CajaConBorradorFijo.java

```
@Component
public class CajaConBorradorFijo {

    private final BorradorPedido
borrador;

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

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

AFTER

Caja.java

```
@Component
public class Caja {

    private final
ObjectProvider<BorradorPedido>
borradores;

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

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

There are also the **`request`**, **`session`**, **`application`** and **`websocket`** scopes, which only work in a web context (Spring Framework Reference, 7.0). They arrive with the web, in Part 2.

Proxy, Template Method and Observer, in one line each

- **Proxy:** an object that stands in front of another and decides what happens before and after each call (proxies and AOP).
- **Template Method:** a fixed skeleton with a gap for whatever changes, like **`JdbcTemplate`** (exceptions).
- **Observer:** an object announces that something happened and others react; the reference uses that name for the container's events (Spring Framework Reference, 7.0) (events).

Pattern	In the book (GoF)	In Spring
Factory	an object that creates others	**`BeanFactory`**, **`ApplicationContext`**, each **`@Bean`**, **`FactoryBean`**
Singleton	the class guarantees one instance per ClassLoader	the container: one per bean definition and per container
Prototype	an object is created by copying a model one	the container runs the recipe again; it copies nothing
Proxy	a stand-in that controls access to the real one	the proxies for AOP, **`@Transactional`**, **`@Async`**...
Template Method	the skeleton lives in a superclass	**`JdbcTemplate`** and friends, with *callbacks*
Observer	a subject notifies its observers	**`ApplicationEventPublisher`** and **`@EventListener`**

> **What to take with you:** everyone shares the singleton: never store a customer's state in it. For a prototype inside a singleton, ask for it with **`ObjectProvider`**.

Configuring the container: XML, scanning or classes

Spring reads the recipes in three ways, and you'll find all three in real code.

XML, the original way

In Spring 1.0, recipes were written in XML. These are the ones for **`ConfiguracionPedidos`**:

pedidos.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>
```

You load them with **`new ClassPathXmlApplicationContext("pedidos.xml")`**. The configuration lives outside the code, but classes are named as text: if you rename **`PedidoService`**, the compiler doesn't warn you and the error shows up at startup.

Component scanning

Since Spring 2.5 you can mark the class and let the container find it (Spring Framework 2.5 Released):

ServicioEscaneado.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` with no parameters walks the package of the class that declares it and its subpackages (Spring Framework API, **`@ComponentScan`**), and registers every class marked with **`@Component`** or one of its specializations: **`@Service`** for the service layer, **`@Repository`** for the data layer and **`@Controller`** for

the web (Spring Framework Reference, 7.0). It's the shortest way, at the cost of your class importing a Spring annotation.

Configuration classes

The third way, since Spring 3.0, is the **`@Configuration`** class with **`@Bean`** methods from the previous section. It works with any class, even the ones you didn't write: you can't put **`@Component`** on **`JdbcTemplate`** or a **`DataSource`**, but you can create them with **`@Bean`**. The usual approach is scanning for your own classes and **`@Bean`** for third-party ones.

When there are two candidates

The container resolves dependencies by type. If there are two beans of the same type, it doesn't guess:

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");
}
```

There are two ways to break the tie. **`@Primary`** marks the preferred one when nobody asks for another (Spring Framework Reference, 7.0):

ConfiguracionConPrimary.java

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

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

`@Qualifier` chooses at the injection point. A bean's name is its default qualifier, so **`@Qualifier("sms")`** picks the **`sms`** bean:

ConfiguracionConQualifier.java

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

> – **Going deeper**
>
> since Spring 6.2 there's **`@Fallback`**, the inverse of **`@Primary`**: it marks beans that are only used if nothing else is left. And if there's no qualifier and no primary, Spring compares the **parameter name** with the bean names; since 6.1 that requires compiling with **`-parameters`** (Spring Framework

> Reference, 7.0). My project compiles without that option on purpose, so every tiebreak is written down in the code.

Profiles: one recipe per environment

On your machine you don't want to send real emails; in production, you do. A **profile** is a name that turns recipes on or off: **`@Profile("local")`** registers the bean only if the **`local`** profile is active (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);
    }
}
```

Outside a test, you activate the profile with the **`spring.profiles.active`** property, for example **`-Dspring.profiles.active=produccion`**. With none active, the **`default`** profile applies (Spring Framework Reference, 7.0); since no recipe here is **`default`**, without a profile there's no notifier.

A bean's lifecycle

An email connection is opened when the object is ready and closed when the application shuts down. Without help, you'd have to call **`abrir()`** (open) and **`cerrar()`** (close) on every possible path. The container knows when each thing happens and lets you know: **`@PostConstruct`** runs once the bean has its dependencies, and **`@PreDestroy`**, before it's destroyed.

ConexionCorreo.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");
    }
}
```

The life of ConexionCorreo, from the recipe to shutdown



The container starts and reads both recipes. No object exists yet.

ConexionCorreo needs a Bitacora in its constructor, so the container creates the Bitacora first.

The Bitacora has nothing to initialize: it's ready. Now the container calls the ConexionCorreo constructor and passes it the Bitacora. With constructor injection, creating and injecting are a

single step.

With the dependencies in place, the container calls the @PostConstruct method: it opens the connection. This is the first moment the object can use what it received.

Then it goes through the BeanPostProcessors again, now in their after-initialization phase (one of them ran the @PostConstruct itself, in the before phase). They can return the bean as is or swap it for another object, such as a proxy.

The bean is ready. Your code asks for it with getBean and uses it: it sends an email. Everything before this happened at startup, before this moment.

When the container closes, the dependent bean is destroyed first: ConexionCorreo's @PreDestroy closes the connection while the Bitacora still exists.

Only then is the Bitacora destroyed, the bean the connection depended on.

The test checks that order (a **`Bitacora`** is a log book where each step writes an entry). Another one checks that, on destruction, the **`@PreDestroy`** of the bean that depends on another runs first.

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");
}
```

Three practical details:

- **`@PostConstruct`** and **`@PreDestroy`** don't come from Spring. They used to be **`javax.annotation`** and shipped with the JDK, which removed them in Java 11; today they're **`jakarta.annotation`** and you add them with **`jakarta.annotation-api`** (Spring Framework Reference, 7.0). It's in the dependency block in the section on exceptions.
- The **`InitializingBean`** and **`DisposableBean`** interfaces do the same thing, but the Spring team recommends the annotations, which keep your class from depending on Spring (Spring Framework Reference, 7.0).

- Outside a web application, someone has to close the container for the **`@PreDestroy`** methods to run: the try-with-resources in the examples, or **`registerShutdownHook()`**, which closes it when the JVM shuts down (Spring Framework Reference, 7.0).

> The usual mistake: using **`@PostConstruct`** for heavy work, like loading data or calling other services. It runs while the container is still creating beans, and the reference warns that it's meant for validating and preparing structures, with a risk of deadlocks during startup if it does more. For the heavy work it suggests **`SmartInitializingSingleton`** or a listener for the event that signals the context has finished starting (Spring Framework Reference, 7.0). There's another reason in transactions.

> – Going deeper
>
> a **`BeanPostProcessor`** receives every bean the container creates, before and after its initialization methods (Spring Framework Reference, 7.0), and can return it as is or replace it: that's how the container puts a proxy in front of your bean. This one only notes that it saw it:
>
> 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;
>     }
> }
> ```
>
> A lot of the "magic" is post-processors: **`@Autowired`** is handled by **`AutowiredAnnotationBeanPostProcessor`**, and **`@PostConstruct`**/**`@PreDestroy`** by **`CommonAnnotationBeanPostProcessor`** (Spring Framework Reference, 7.0). That's why a bare **`BeanFactory`**, which doesn't register them, ignores those annotations.
>
> A post-processor is created before the beans it processes. That's why its **`@Bean`** method is **`static`** (it doesn't force its configuration class to be created first) and ideally has no dependencies, because a

bean created to build it is left without post-processing (Spring Framework API, **`@Bean`**). **`ObservadorDeBeans`** receives the **`Bitacora`** through an **`ObjectProvider`**, which only asks for it when it's needed.

Properties: configuration outside the code

If the shipping cost lives in the code, changing it means recompiling and deploying. Whatever changes without the logic changing goes into **properties**: key-value pairs read at startup.

tienda.properties

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

They're managed by the **`Environment`**, the container object that models profiles and properties (Spring Framework Reference, 7.0). Its properties come from an ordered list of **`PropertySource`**s, any source of key-value pairs. Out of the box it has two: the JVM system properties (**`-Dkey=value`**) and the environment variables, in that order; if a key is in both, the system property wins.

`@PropertySource` adds a file to the list and **`@Value`** injects a value:

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}` is a **marker** that gets replaced with the value of that key, converted to the **`int`** the parameter asks for. Since the file comes after the system properties, a system property beats it: that's how you change a value in one environment without touching the file.

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();
    }
}
```

The usual mistake: a typo in the key. This recipe asks for **`tienda.nombr`**:

ConfiguracionConErrata.java

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

There's no error at startup: by default the resolver is lenient and, if it can't find the key, it injects the marker text as is (Spring Framework Reference, 7.0). The store introduces itself as **`${tienda.nombr}`**. To make the container fail at startup, declare a **`PropertySourcesPlaceholderConfigurer`** in a **`static`** method:

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'");
    }
```

> If a key can be missing on purpose, give it a default value: **`${envio.costo:0}`**.

In Part 2, Spring Boot builds this list of sources for you; the model stays the same: the first source that has the key wins.

Exceptions, the Spring way

For this section and the next two, your project adds these dependencies:

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 is a database that runs in memory, inside the same process: it's born with the test and dies with it. This is its orders table:

schema.sql

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

The problem: SQLException

With plain JDBC, saving an order looks like this:

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` is **checked**: the compiler forces every method that calls **`guardar`** (save) to catch it or declare it. And almost nobody can do anything about it: if the database went down, the order service has no way to fix that. The typical result is a chain of **`throws SQLException`** through every layer, or an empty **`catch`**.

It also says very little. If you save order 42 twice, it doesn't say "duplicate order": it gives you a code, **`SQLState 23505`**, which in H2 is a uniqueness violation. I checked it with a test. To react, your code would have to know its database's codes.

The solution: DataAccessException

`JdbcTemplate` takes care of the repetitive parts: it creates and releases the resources, runs the statement and leaves you only the SQL (Spring Framework Reference, 7.0). It's the Template Method from the patterns section.

BEFORE

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();
    }
}
```

AFTER

RepositorioJdbc.java

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

It also **translates** the **`SQLException`** into Spring's hierarchy, whose root is **`DataAccessException`**: **unchecked** exceptions (they extend **`RuntimeException`**) that say what happened through their name and wrap the original, so no detail gets lost (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);
    }
}
```

Why unchecked? Because most database errors can't be fixed where they happen. That way the layers in between don't need catch-and-rethrow blocks or **`throws`** declarations: the exception travels up to the layer that can actually do something (Spring Framework Reference, 7.0).

The **`DataSource`** (the source of connections) and the **`JdbcTemplate`** aren't your classes, so they go with **`@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);
    }
}
```

> – Going deeper
>
> since Spring 6.0, the default translator is **`SQLExceptionSubclassTranslator`**, which relies on the JDBC 4 **`SQLException`** subclasses and, if that's not enough, on the **`SQLState`** (Spring Framework Reference, 7.0). Spring also translates JPA and Hibernate exceptions: **`@Repository`** marks the class, and a **`PersistenceExceptionTranslationPostProcessor`** registered in the container makes the translation automatic; Spring Boot registers it for you (Spring Framework Reference, 7.0). You'll see it with Spring Data in Part 2.

Proxies and AOP: the code that repeats everywhere

Logging who called what, measuring how long it took, opening a transaction, checking permissions: some code shows up in every method without belonging to any of them. It's called a **crosscutting concern**. Written by hand, it gets copied into every method, mixed in with the business logic:

BEFORE

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");
    }
}
```

AFTER

ServicioDePedidos.java

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

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

On the right, a custom annotation marks where you want the log (**`Auditado`** means audited):

Auditado.java

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

On its own it does nothing. What makes it work is an **aspect**:

AspectoDeAuditoria.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);
        }
    }
}
```

Aspect-oriented programming (AOP) writes the crosscutting logic once, in a separate module, and declares where it applies. Its vocabulary (Spring Framework Reference, 7.0):

- The **aspect** is that module, the **`AspectoDeAuditoria`** class.
- A **join point** is a point in the execution where you can step in; in Spring, it's always the execution of a method.
- The **pointcut** says which ones it applies to: **`@annotation(…Auditado)`**, the methods annotated with **`@Auditado`**.
- The **advice** is what runs there. **`@Around`** wraps the call: it does something before, decides whether to continue with **`proceed()`** and does something after.

You turn it on with **`@EnableAspectJAutoProxy`** and the **`aspectjweaver`** library, which Spring uses to read the pointcuts (Spring Framework Reference, 7.0):

ConfiguracionAop.java

```
@Configuration
@EnableAspectJAutoProxy
public class ConfiguracionAop {

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

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

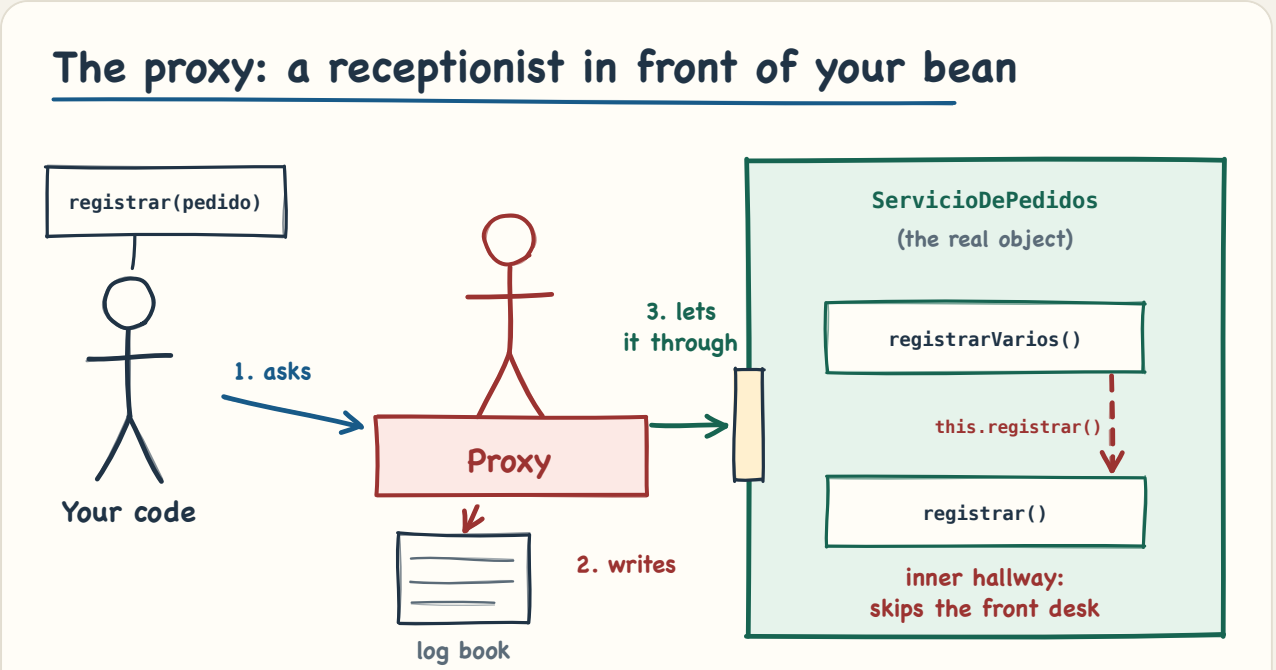
    @Bean
    RepositorioPedidos repositorioPedidos() {
        return new RepositorioEnMemoria();
```

```
    }

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

How it works: the proxy

Spring doesn't change your class. It puts another object in front of it:



What comes through the door passes the front desk; what the service asks itself, does not.

> The proxy is the receptionist: everything that comes in through the door goes past her; what the service asks of itself doesn't.

When an aspect applies to a bean, the container creates a **proxy** with the same shape, and the proxy is what it hands you (Spring Framework Reference, 7.0). If the class implements interfaces, the proxy is a *JDK dynamic proxy* for those interfaces; if not, a subclass generated with CGLIB (Spring Framework Reference, 7.0). **`ServicioDePedidos`** doesn't implement any:

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");
    }
}
```

The same mechanism powers **`@Transactional`**, **`@Async`** and other Spring annotations, with the same two traps.

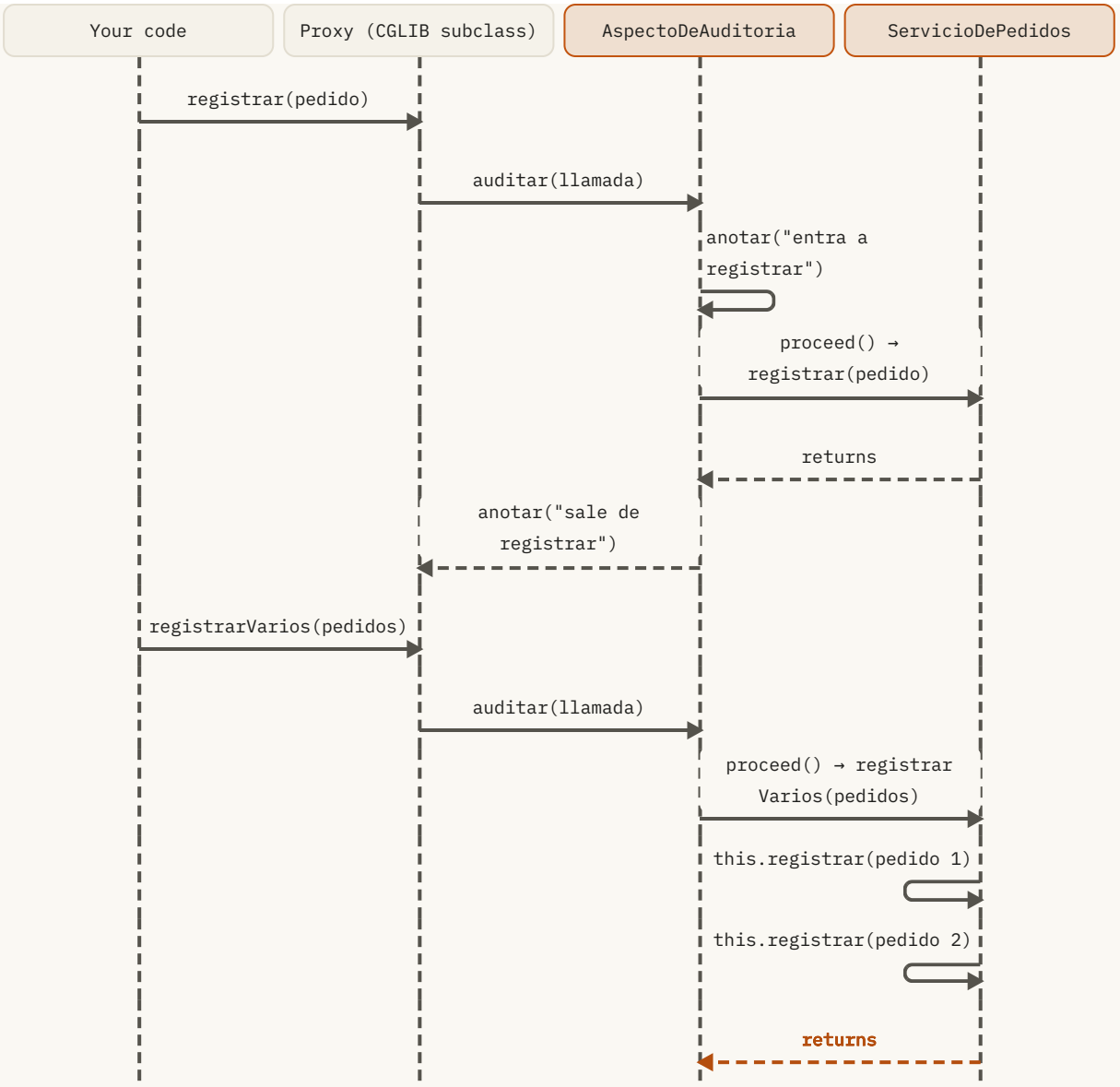
The self-invocation trap

This method records several orders by calling **`registrar`** in a loop:

ServicioDePedidos.java

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

A call goes through the proxy; self-invocation doesn't



What you got from getBean isn't your ServicioDePedidos: it's the proxy the container put in front of it. Your code doesn't notice, because it has the same shape.

The method has @Auditado, so the proxy hands the call to the aspect before touching your object.

The @Around advice does its before part.

proceed() continues to the real object, which saves the order. It's the only moment the aspect lets the call through.

The method finishes and control goes back to the aspect.

The finally block logs the exit and the response goes back through the proxy to your code. Two lines are left in the log.

Now you call registrarVarios. It has @Auditado too, and it comes in through the proxy.

The aspect logs "entra a registrarVarios" and continues with proceed().

The call reaches the real object.

Here's the trap. Inside the object, this is the real object, not the proxy. The call never leaves the object: nobody intercepts it and the aspect never finds out.

Same with the second order. Both get saved, but neither shows up in the log.

registrarVarios finishes, the aspect logs "sale de registrarVarios" and the response goes back through the proxy. The log has two lines, not six.

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);
    }
}
```

According to the reference, self-invocation skips the advice, and the best fix is to reorganize the code, for example by moving the inner method to another bean. The alternative is for the bean to inject itself (*self injection*) and call through that reference, which is the proxy; **`AopContext.currentProxy()`** is the last resort, and the reference strongly discourages it (Spring Framework Reference, 7.0).

What a proxy can't intercept

A CGLIB proxy is a subclass, and it can only step in where it can override: **`public`** and **`protected`** methods (and package-private ones, if needed). A JDK proxy, only the interface's public methods. **`final`** and **`private`** methods can't be intercepted (Spring Framework Reference, 7.0).

A **`final`** method is worse than it looks, and Spring warns you about it at startup:

```
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();
}
```

Since the subclass can't override it, the call never reaches the real object: it runs on the proxy itself, which never went through your constructor and has empty fields. I checked it with a test, and the result isn't a method that skips the log: it's a **`NullPointerException`**, because **`repositorio`** is **`null`** inside the 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();
    }
}
```

What to take with you: what the container hands you is the proxy, and only what comes in through the proxy gets intercepted: calls through **`this`** don't, and neither do **`final`** or **`private`** methods. Signatures that need to go through a proxy should be public.

– **Going deeper**

the Framework suggests interface-based proxies, and Spring Boot uses class-based proxies by default (**`spring.aop.proxy-target-class=true`**). Since 7.0, **`@Proxyable(INTERFACES)`** or **`@Proxyable(TARGET_CLASS)`** pins the type for a single bean (Spring Framework Reference, 7.0). All of this is Spring AOP, which is proxy-based; if you need more, Spring can use the AspectJ compiler or weaver, which modify the classes themselves (Spring Framework Reference, 7.0). This series doesn't use them.

Transactions

What a transaction is

Buying means saving the order and charging for it. What happens if the order is saved and the charge fails?

BEFORE

ComprasSinTransaccion.java

AFTER

Compras.java

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

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

Without a transaction, the database ends up with an order nobody paid for. A **transaction** groups several operations so that either all of them apply or none do: if everything goes well, it's **committed**; if something fails, it's **rolled back** and the database goes back to how it was. In both tests, **`Pagos`** (payments) rejects a payment with an amount of 0:

TransaccionesTest.java

```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();
}
```

How Spring does it

`@Transactional` works with a proxy in front of **`Compras`**: before entering the method it opens a transaction; if the method finishes normally, it commits it; if an exception comes out, it rolls it back. It needs two pieces:

ConfiguracionTransacciones.java

```java
@Configuration
@EnableTransactionManagement
```

```java
@Import(ConfiguracionDatos.class)
public class ConfiguracionTransacciones {

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

`@EnableTransactionManagement` turns on the proxies, and the **`DataSourceTransactionManager`** knows how to open, commit and roll back a transaction on that **`DataSource`**. When it opens one, it binds a connection to the current thread, and **`JdbcTemplate`** uses that connection instead of asking for another (Spring Framework API, **`DataSourceTransactionManager`**). That's why the repository and **`Pagos`** join the same transaction and roll back together.

The annotation goes on **`comprar`** (buy), in the service, not in the repository: the unit of work is "buy", which touches two tables, and the order repository doesn't know a charge comes next.

With no parameters, **`@Transactional`** uses **`REQUIRED`** propagation, the database's isolation level, read-write, and rollback on **`RuntimeException`** or **`Error`**, but **not on a checked exception** (Spring Framework Reference, 7.0).

> – Going deeper
>
> the hand-written JDBC from the previous section, which calls **`dataSource.getConnection()`**, would **not** join the transaction: it would ask for a new connection. You have to get it with **`DataSourceUtils.getConnection(dataSource)`**, as **`JdbcTemplate`** does internally (Spring Framework API, **`DataSourceTransactionManager`**).

Rollback only on unchecked

`SaldoInsuficiente` (insufficient balance) is a checked exception. This method saves the order and then throws it:

Compras.java

```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);
}
```

The exception leaves the method, but the transaction **is committed** and the order stays saved. For a checked exception to roll back too, you declare it with **`rollbackFor`**, which also covers its subclasses (Spring Framework Reference, 7.0). Since 6.2 you can also change the rule for the whole application with **`@EnableTransactionManagement(rollbackOn = RollbackOn.ALL_EXCEPTIONS)`**, which the javadoc recommends unless you rely on business exceptions that commit on purpose:

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();
}
```

Propagation: join or start another

When one **`@Transactional`** method calls another, **propagation** decides what happens:

- **`REQUIRED`**, the default: if there's already a transaction, it joins it; if not, it opens one. Everyone who joins shares the same physical transaction (Spring Framework Reference, 7.0).

- **`REQUIRES_NEW`**: it always opens a new, independent transaction, with its own connection, which commits or rolls back on its own.

`REQUIRES_NEW` is for whatever has to stay written even if the main operation fails, like auditing:

Auditoria.java

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

Compras.java

```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);
}
```

The payment fails, the order is rolled back and the audit line stays:

TransaccionesTest.java

```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");
}
```

> Watch out: while the inner transaction runs, the outer one stays open with its connection. With many threads doing this at once, the connection pool can run dry and the threads end up waiting on each other. The reference says not to use **`REQUIRES_NEW`** unless the pool has at least one more connection than concurrent threads (Spring Framework Reference, 7.0).

The rollback-only trap

`Cupones.aplicar` (apply a coupon) is transactional and throws **`CuponInvalido`**, an unchecked exception, if the coupon doesn't exist:

Cupones.java

```
public class Cupones {

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

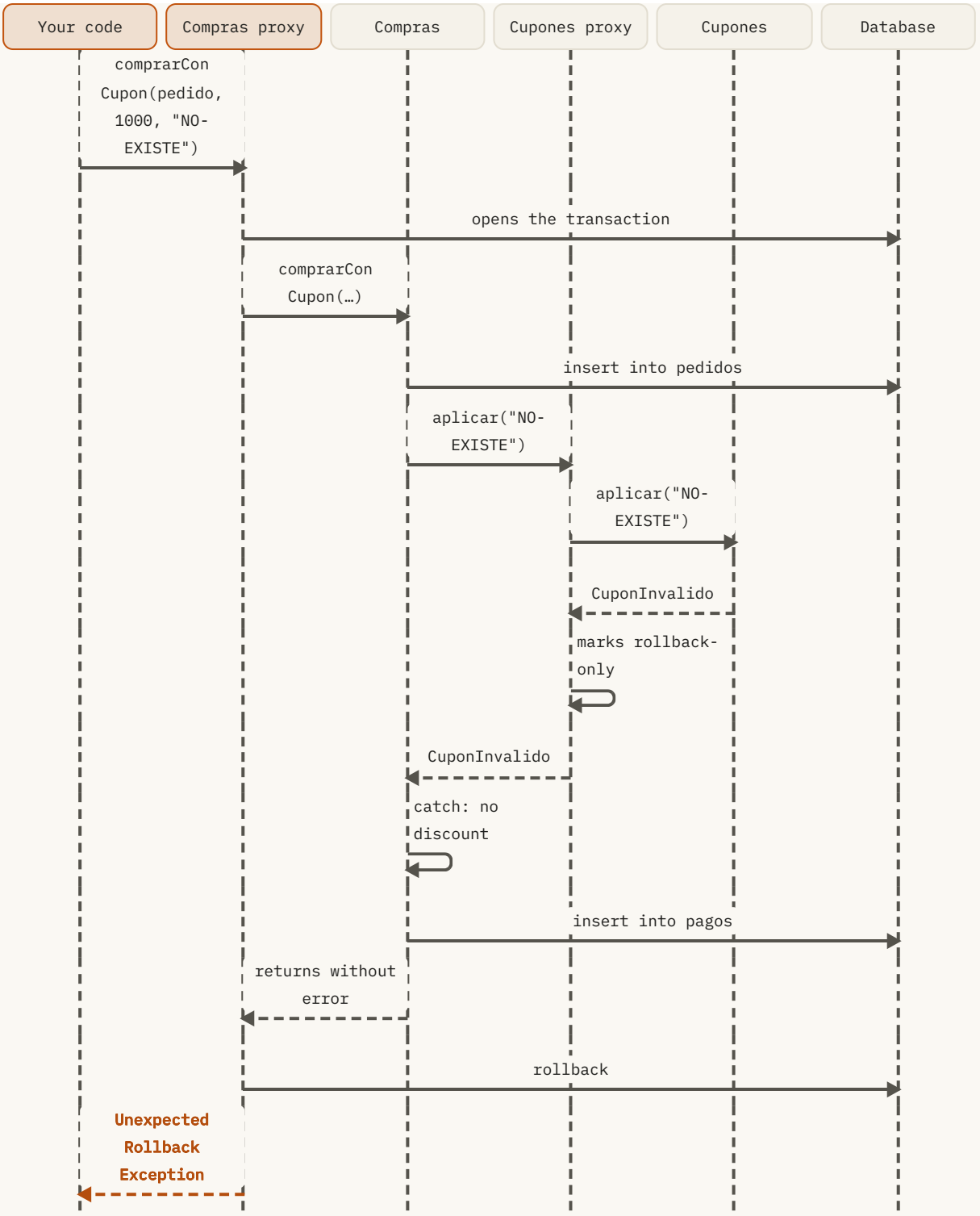
The purchase catches that exception so it can carry on without a discount:

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);
}
```

It looks reasonable, but the purchase isn't saved and your code gets an **`UnexpectedRollbackException`**:

A caught exception that still rolls back the purchase



The call comes in through the Compras proxy, because the method is @Transactional.

There's no transaction in progress: the proxy opens one and binds its connection to the thread. Every JdbcTemplate on this DataSource running on this thread will use it.

Only now does the call reach your object.

The order is inserted, inside the transaction. It isn't final yet.

Cupones is another bean, so the call goes through its proxy. aplicar is also @Transactional with REQUIRED: since there's already a transaction, it joins it. It doesn't open another one.

The real object checks the coupon.

The coupon doesn't exist: an unchecked exception comes out.

The Cupones proxy sees a RuntimeException leave a transactional method. It can't roll back the transaction, because it didn't open it, so it marks it: this transaction can only end in a rollback.

The exception keeps going up to Compras.

Compras catches it and carries on as if nothing happened. But the mark is already set, and catching the exception doesn't clear it.

The charge is inserted at full price.

The method finishes normally. The Compras proxy, which opened the transaction, is about to commit it.

When it commits, it finds the rollback-only mark. It can't commit: it rolls everything back, the order and the payment.

And it tells you with an exception, so you never believe something was committed when it was rolled back.

The reference describes it like this: with **`REQUIRED`**, a rollback-only mark set on the inside affects the outer transaction, and the exception exists so the caller doesn't believe there was a 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();
}
```

If an invalid coupon is an expected outcome and not a reason to roll back the purchase, tell Spring with **`noRollbackFor`**. With that change, the purchase is saved at full price, and I checked it with a test:

CuponesTolerantes.java

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

The other way out is for coupon validation not to be transactional: with no proxy, nobody marks anything.

Isolation and readOnly

Two transactions running at the same time can see each other's changes to different degrees. **Isolation** says how much. **`Isolation.DEFAULT`** uses your database's level; the other levels are JDBC's (Spring Framework API, **`Isolation`**):

Level	Dirty read	Non-repeatable read	Phantom read
`READ_UNCOMMITTED`	can happen	can happen	can happen
`READ_COMMITTED`	prevented	can happen	can happen
`REPEATABLE_READ`	prevented	prevented	can happen
`SERIALIZABLE`	prevented	prevented	prevented

A **dirty read** is seeing a change another transaction hasn't committed; a **non-repeatable read**, reading the same row twice and getting different values; a **phantom read**, repeating a query and finding new rows. The stricter the level, the fewer anomalies and the more locking.

`readOnly = true` declares that the transaction only reads, so the system can optimize. It's a **hint**, not a prohibition: it doesn't necessarily make a write fail, and a transaction manager that doesn't understand it ignores it silently (Spring Framework API, **`@Transactional`**).

@PostConstruct runs without a transaction

This bean loads data at startup, from its **`@PostConstruct`**, by calling a **`@Transactional`** method that returns whether a transaction is active:

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;
    }
}
```

At startup there isn't one; called later, through the bean, there is:

TransaccionesTest.java

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

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

`@PostConstruct` runs on the real object, before the proxy arrives (Spring Framework Reference, 7.0), and **`cargar()`** (load) is a self-invocation on top of that. The reference warns about it: don't rely on **`@Transactional`** inside a **`@PostConstruct`** (Spring Framework Reference, 7.0).

> What to take with you: **`@Transactional`** lives in a proxy, so its rules apply: no self-invocation and no **`@PostConstruct`**. And by default it commits on checked exceptions, unless you use **`rollbackFor`** or **`rollbackOn = ALL_EXCEPTIONS`**.

Events: notify without knowing who

When an order is recorded you have to notify the customer, reserve the stock and, maybe tomorrow, add loyalty points. If every interested party is a dependency of the service, **`PedidoService`** grows with every new requirement and you have to edit it each time:

BEFORE

PedidoServiceQueAvisaATodos.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;
```

AFTER

PedidoServiceConEventos.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));
    }
}
```

On the right, the service only **announces** what happened: it publishes an event, which can be any object. Since Spring 4.2 you don't need to extend **`ApplicationEvent`** (Spring Framework Reference, 7.0), so a **`record`** is enough:

PedidoRegistrado.java

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

Each interested party listens with **`@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());
    }
}
```

The container injects the **`ApplicationEventPublisher`** and, at startup, looks for its beans' **`@EventListener`** methods.

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);
    }
```

```
}
```

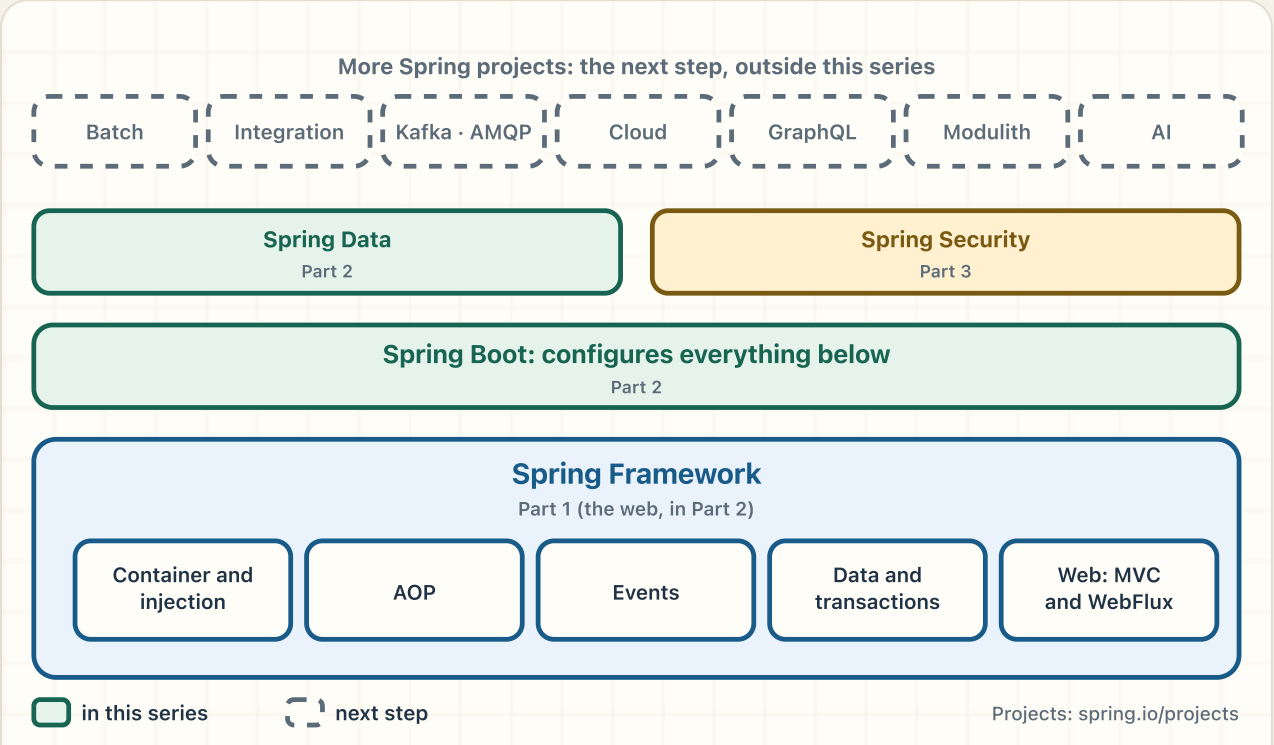
Adding an interested party means adding a bean with its **`@EventListener`**, without touching the service.

By default, listeners run **synchronously**: **`publishEvent`** doesn't return until all of them finish, on the same thread and inside the publisher's transaction (Spring Framework Reference, 7.0); I checked the thread part with a test. With **`@Async`** (and **`@EnableAsync`** in the configuration) the listener runs separately, but if it fails, the exception never reaches whoever published.

> **The usual mistake:** having the listener send the email inside the publisher's transaction. If the transaction later rolls back, the order doesn't exist, but the email already went out. To notify only if the transaction committed there's **`@TransactionalEventListener`**, in Part 3.

The ecosystem map

Everything in this part is Spring Framework. The rest is built on that foundation:



A single container at the base; Boot configures it; each project solves one problem on top.

- **Spring Framework** also includes the web: Spring MVC, on top of the Servlet API, and since 5.0 Spring WebFlux, which is reactive (Spring Framework 5.0 goes GA).

- **Spring Boot** configures all of the above with sensible defaults. That's Part 2.
- **Spring Data** gives you a single data access model for relational and non-relational databases; **Spring Security**, authentication and authorization. They show up in Parts 2 and 3.
- The rest is left as a next step, and all of it rests on what you learned here: **Spring Batch** (batch processing), **Spring Integration** (integration patterns), **Spring for Apache Kafka** and **Spring AMQP** (messaging), **Spring Cloud** (distributed systems), **Spring for GraphQL**, **Spring Modulith** (domain modules) and **Spring AI** (AI models) (spring.io, *Projects*).

What only experience teaches

These traps are usually learned through an incident:

- **State in a singleton.** A field that holds something from one customer is shared by every request.
- **Circular dependencies.** Through the constructor, the container won't start, and that's fine: the cycle is a design problem. Through fields, in Spring without Boot, it starts and hides it.
- **Long constructors.** If a constructor asks for seven dependencies, the class does too much. The Spring reference calls it a *code smell* (Spring Framework Reference, 7.0).
- **A prototype inside a singleton** is injected only once: **`ObjectProvider`**.
- **A typo in `@Value`** arrives as text: a strict resolver or default values.
- **Rollback with a checked exception:** there isn't one, unless you use **`rollbackFor`** or **`rollbackOn = ALL_EXCEPTIONS`** for the whole application.
- **Catching the exception from a transactional method** doesn't save the transaction if it joined yours.
- **Self-invocation and `final` methods** skip the proxy, and with it the transaction or the aspect.

If you take away just one idea, make it this one: what the container hands you isn't always your object. Sometimes it's a proxy, and almost every Spring surprise comes from there.

Which of these traps took you the longest to find the first time? And if none of them has bitten you yet, which one would you bet is hiding in your project today?

Continue with Part 2: Spring Boot, from your first app to a complete API, where Spring Boot sets all of this up for you, and you'll know what it's doing.

References

- 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`.
- Spring blog: 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) and InversionOfControl (2005).
- Java EE 6 API: `SessionBean` and `EJBHome`; The J2EE 1.4 Tutorial: The Contents of an Enterprise Bean.
- spring.io, Projects; Apache Maven Wrapper; IntelliJ IDEA as a unified product.