Spring Boot: de la primera app a una API completa

Parte 2 de 3 · 4 de noviembre de 2026 · spring-boot · spring · java · api-rest · serie-spring

> **Parte 2 de 3** de la serie «Spring, de novato a experto». Viene de la parte 1: Spring Framework, el contenedor que arma tu aplicación y sigue con la parte 3: Spring Boot en producción.

En la Parte 1 armamos la tienda de pedidos con Spring Framework solo, sin una línea de HTTP. Para exponerla como API habríamos tenido que instalar un servidor, configurar la base, elegir una biblioteca de JSON y encontrar versiones de todo eso que funcionen juntas.

En esta parte hacemos ese camino con Spring Boot: de un proyecto vacío a una API REST que valida lo que recibe, responde errores que un cliente entiende, guarda en Postgres sin el problema N+1, versiona su esquema y tiene tests en tres niveles.

Todo el código lo compilé y lo probé con Spring Boot 4.1.2: cada comportamiento que afirmo lo comprobé con un test contra un Postgres real, y las salidas de consola que ves son las que obtuve.

▼ EN ESTE ARTÍCULO

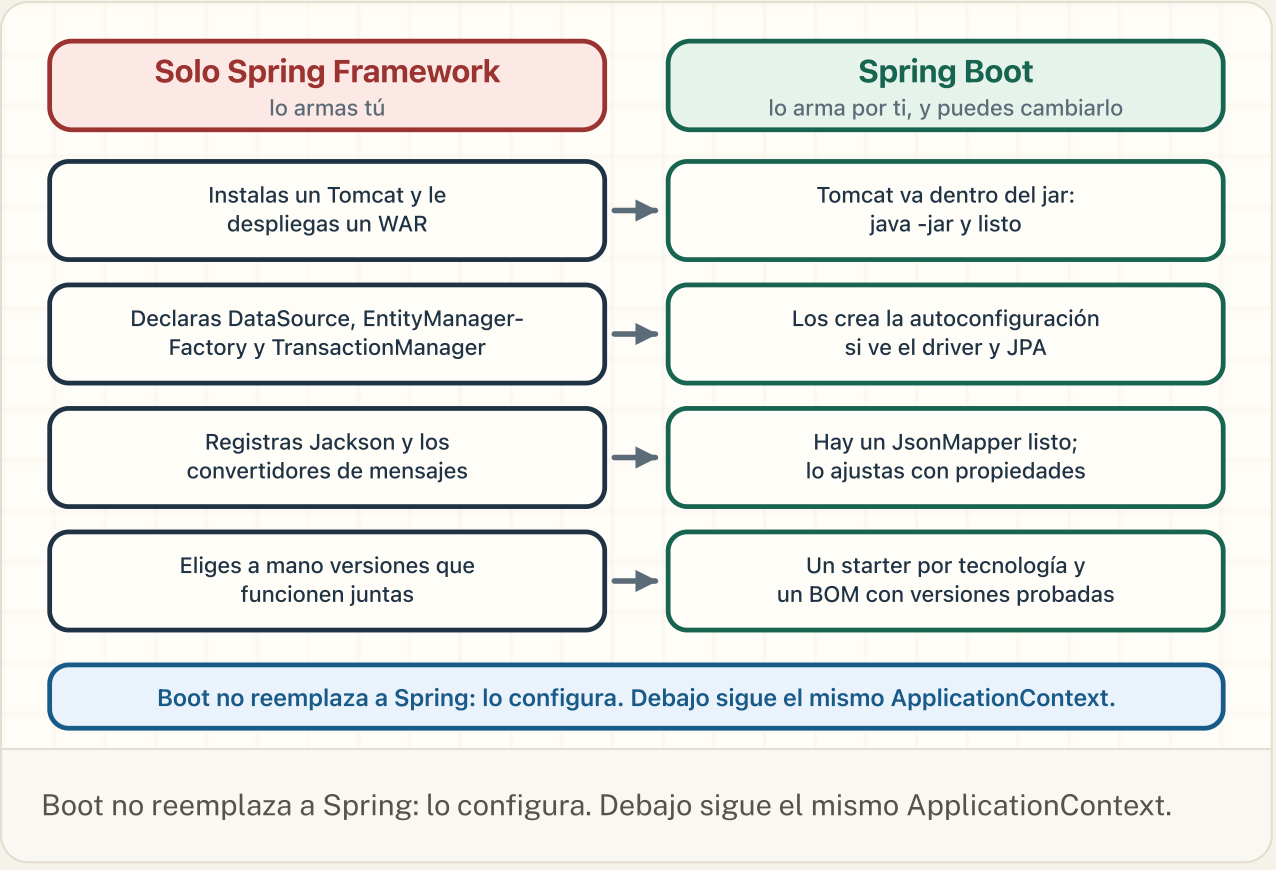
01. Qué le faltaba a Spring
02. Tu primera app en 10 minutos
03. Autoconfiguración sin magia
04. Propiedades y perfiles
05. Spring MVC por dentro
06. Validación
07. Errores que el cliente entiende
08. Datos con Spring Data JPA
09. Migraciones con Flyway
10. Llamar a otros servicios
11. Logging
12. Tests
13. Lo que solo enseña la experiencia
14. Referencias

Qué le faltaba a Spring

Spring Framework resolvió el problema de armar los objetos. Pero una aplicación web necesita más, y antes de Boot todo eso lo configurabas tú:

- **El servidor.** Instalabas un Tomcat, empaquetabas la app como WAR y la desplegabas en él.
- **La base de datos.** Declarabas a mano el **`DataSource`** (el objeto que entrega conexiones), el pool, el **`EntityManagerFactory`** de JPA y el administrador de transacciones.
- **El JSON.** Registrabas Jackson y los convertidores entre objetos y JSON.
- **Las versiones.** Elegías una de Spring, una de Hibernate, una de Jackson, una del driver… y descubrías al ejecutar cuáles no se llevaban bien.

Nada de eso era difícil por separado. Todo junto, en cada proyecto nuevo, era trabajo ajeno a tu negocio que fallaba de formas poco claras: el *infierno de configuración*.



Boot no reemplaza a Spring: lo configura. Debajo sigue el mismo ApplicationContext.

Necesitas…	Con Spring Framework solo	Con Spring Boot
un servidor	instalas uno y despliegas un WAR	el servidor va **embebido** en tu jar: **`java -jar`** y listo
los beans de siempre	los declaras tú (**`DataSource`**, JPA, Jackson…)	la **autoconfiguración** los crea si encuentra la biblioteca

Necesitas...	Con Spring Framework solo	Con Spring Boot
dependencias compatibles	eliges cada versión	un **starter** por tecnología y un BOM con versiones probadas
ajustar valores	código o XML	**propiedades** en **`application.yml`**, variables de entorno o argumentos

Boot no reemplaza a Spring. Debajo de una app de Boot está el mismo **`ApplicationContext`** de la Parte 1, con los mismos beans, proxies y transacciones. Boot es un conjunto de configuraciones ya escritas más un mecanismo para decidir cuáles aplican. Cuando algo de Boot te sorprenda, la respuesta casi siempre está en el contenedor.

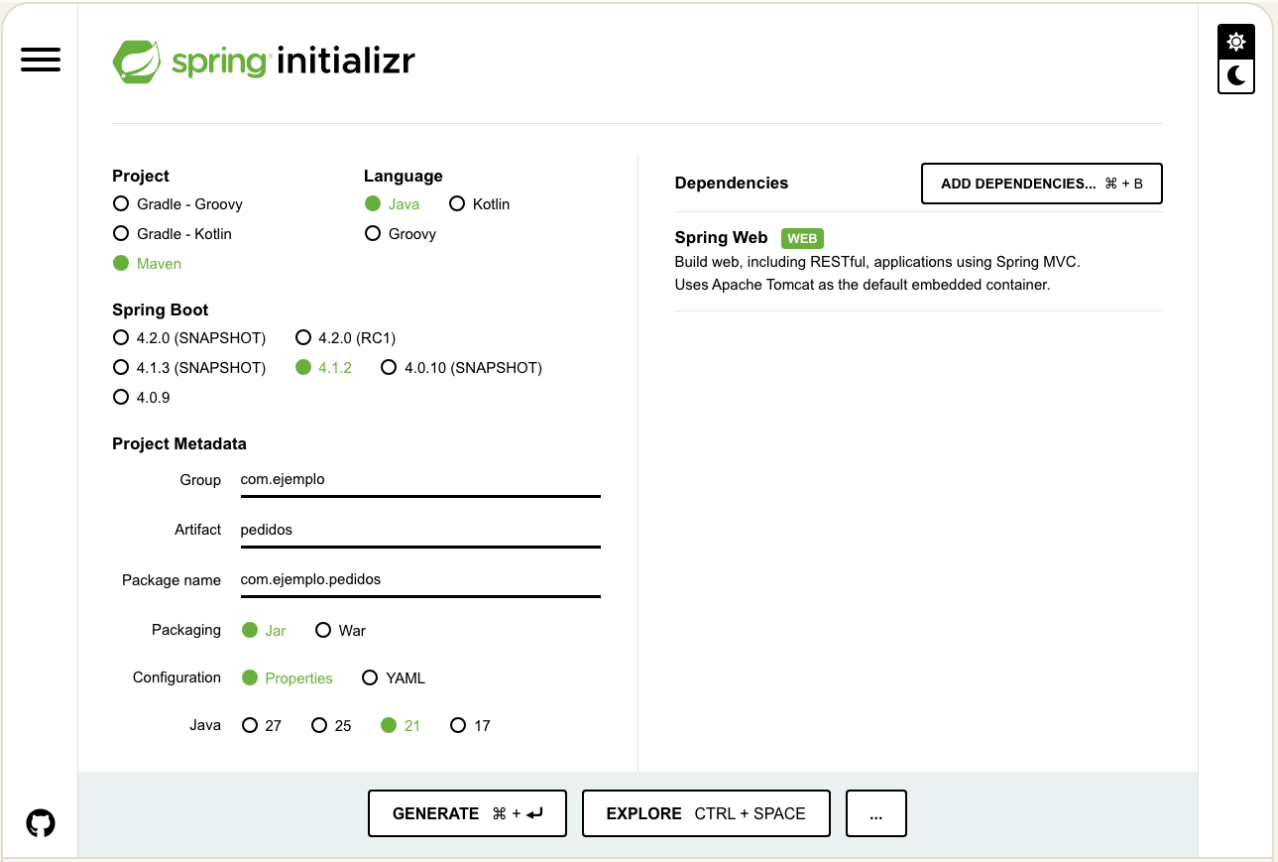
> **Nota:** esta parte asume la Parte 1: inyección por constructor, beans, **`@Configuration`** y **`@Bean`**, el **`Environment`** y **`@Value`**, los proxies, **`@Transactional`** y **`DataAccessException`**. Del lado de Java, records y lambdas.

Tu primera app en 10 minutos

Necesitas un JDK 21 y un IDE, como en la Parte 1; Maven no hace falta instalarlo.

Generar el proyecto en start.spring.io

start.spring.io, el *Spring Initializr*, es la página oficial que genera el esqueleto de un proyecto de Boot: eliges opciones, descargas un zip y lo descomprimes. Esta URL la abre con todo ya elegido: **`https://start.spring.io/#!type=maven-project&language=java&platformVersion=4.1.2&packaging=jar&jvmVersion=21&groupId=com.ejemplo&artifactId=pedidos&dependencies=web`**.



Así se veía start.spring.io el 4 de noviembre de 2026 con las opciones de esta guía.

- **Project: Maven.** La herramienta de build: baja dependencias, compila, prueba y empaqueta. La página propone Gradle; usamos Maven, como en toda la serie.
- **Spring Boot: 4.1.2.** La estable más reciente ese día, la misma del backend de este sitio. Las SNAPSHOT y RC son versiones en desarrollo.
- **Group y Artifact.** El *group* identifica a tu organización, al revés que un dominio (**`com.ejemplo`**); el *artifact* es el nombre del proyecto (**`pedidos`**). Juntos forman el **paquete raíz**, **`com.ejemplo.pedidos`**, que importa más de lo que parece.
- **Packaging: Jar.** Un jar ejecutable con el servidor adentro. War es para desplegar en un servidor que ya existe.
- **Configuration: Properties o YAML.** El formato del archivo de configuración.
- **Java: 21.** La página propone 17; Boot 4 pide 17 o más (Spring Boot 4.0 Migration Guide).
- **Dependencies: Spring Web.** Lo único que necesitamos para una API.

Presiona **Generate**, descomprime el zip y abre la carpeta en el IDE como proyecto Maven:



Cinco cosas importan al principio: el pom, el wrapper de Maven, el paquete raíz, el main y la configuración.

El **`main`** es una anotación y una llamada a **`SpringApplication.run`**. Yo le agregué **`@ConfigurationPropertiesScan`**, que veremos en la sección de propiedades:

PedidosApplication.java

```java
@SpringBootApplication
@ConfigurationPropertiesScan
public class PedidosApplication {

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

Correrla de tres formas

1. Desde el IDE, con el botón *Run* junto al **`main`**. Es lo que harás mientras desarrollas.

2. Con el plugin de Maven (en Windows, **`mvnw.cmd`**):

```
./mvnw spring-boot:run
```

3. Empaquetada, como en producción, sin Maven ni IDE de por medio:

```
./mvnw package
java -jar target/pedidos-0.0.1-SNAPSHOT.jar
```

La tercera es la que corre en un servidor: un solo archivo con tu código, sus dependencias y el servidor web adentro.

Leer el log de arranque

Esta es mi salida, sin la fecha, el hilo ni el nombre largo de los loggers. La corrí en el puerto 18083 porque el 8080 estaba ocupado; tú verás 8080:

```
INFO … PedidosApplication : Starting PedidosApplication v0.0.1-SNAPSHOT using Java
25.0.4 with PID 15769
INFO … PedidosApplication : No active profile set, falling back to 1 default profile:
"default"
INFO … TomcatWebServer    : Tomcat initialized with port 18083 (http)
INFO … TomcatWebServer    : Tomcat started on port 18083 (http) with context path '/'
INFO … PedidosApplication : Started PedidosApplication in 0.622 seconds (process running
for 0.797)
```

- **using Java 25.0.4**: con qué Java corre. Si no es el que esperas, mira aquí primero. (Un JDK más nuevo ejecuta sin problema lo compilado para Java 21.)
- **No active profile set**: no activaste ningún perfil; se usa **`default`**.
- **Tomcat initialized / started**: Boot creó un Tomcat dentro de tu proceso y ya acepta conexiones. Nadie lo instaló.
- **Started PedidosApplication**: la app está lista. Si esta línea no aparece, el motivo está más arriba.

La página de error Whitelabel

Abre **`http://localhost:8080`** en el navegador:

```
Whitelabel Error Page
This application has no explicit mapping for /error, so you are seeing this as a
fallback.
There was an unexpected error (type=Not Found, status=404).
```

Es lo esperado: la app no tiene endpoints, así que todo es 404. Boot registra un manejador para **`/error`** que a un navegador, que pide HTML, le responde con esta página genérica (la *whitelabel*), y a cualquier otro cliente, con JSON (Spring Boot Reference, 4.1):

```
$ curl -i http://localhost:8080/
HTTP/1.1 404
```

```
Content-Type: application/json

{"timestamp":"2026-11-05T00:02:32.170Z","status":404,"error":"Not Found","path":"/"}
```

Tu primer endpoint

Crea un paquete **hola** debajo de **com.ejemplo.pedidos** con esta clase:

HolaController.java

```
@RestController
public class HolaController {

    @GetMapping("/hola")
    public String hola(@RequestParam(defaultValue = "mundo") String nombre) {
        return "Hola, " + nombre + "!";
    }
}
```

- **@RestController**: los métodos devuelven el cuerpo de la respuesta, no el nombre de una página HTML.
- **@GetMapping("/hola")**: asocia el método a **GET /hola**.
- **@RequestParam(defaultValue = "mundo")**: toma **?nombre=** de la URL, o **mundo** si no viene.

```
$ curl -i "http://localhost:8080/hola?nombre=Ana"
HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8

Hola, Ana!
```

Spring crea el **HolaController** por el escaneo de componentes de la Parte 1: **@RestController** es un **@Component** especializado, y **@SpringBootApplication** escanea su paquete y los de abajo.

Un endpoint se prueba sin levantar el servidor. **@WebMvcTest** arma solo la capa web y **MockMvc** le manda requests de mentira (lo vemos a fondo en la sección de tests):

HolaControllerTest.java

```
@WebMvcTest(HolaController.class)
class HolaControllerTest {

    @Autowired
    MockMvc mvc;

    @Test
    void saludaPorDefectoAlMundo() throws Exception {
```

```
mvc.perform(get("/hola")).andExpect(status().isOk()).andExpect(content().string("Hola,
mundo!"));
    }

    @Test
    void saludaAQuienSeLePida() throws Exception {
        mvc.perform(get("/hola").param("nombre",
"Ana")).andExpect(content().string("Hola, Ana!"));
    }
}
```

Cambiar el puerto y reiniciar más rápido

El puerto sale de **`server.port`**, 8080 por defecto (Spring Boot Reference, 4.1). Se cambia en **`src/main/resources/application.properties`**:

```
server.port=8081
```

o al arrancar: **`java -jar target/pedidos-0.0.1-SNAPSHOT.jar --server.port=8081`**. Cuál gana si están las dos lo vemos en la sección de propiedades.

DevTools es una dependencia opcional que reinicia la app sola cuando cambia el classpath. Usa dos *classloaders*: uno para las bibliotecas y otro para tu código, el único que descarta y vuelve a cargar. El reinicio se dispara cuando cambia el **classpath**: en Eclipse al guardar, en IntelliJ IDEA al compilar (*Build Project*). DevTools se apaga solo con **`java -jar`** (Spring Boot Reference, 4.1):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

Los tres errores de arranque que vas a ver

> **El error típico: el puerto ocupado.** Otro proceso ya escucha en el 8080 (otra copia de tu app, por ejemplo). La salida real:
>
> ```
> APPLICATION FAILED TO START
>
> Description:
>
> Web server failed to start. Port 8080 was already in use.
> ```

```
Action:

Identify and stop the process that's listening on port 8080 or configure this
application to listen on another port.
```

En macOS y Linux, **`lsof -i :8080`** te dice quién lo tiene.

El error típico: el JDK equivocado. El jar se compiló para Java 21 y lo ejecutaste con uno más viejo. La salida real con un JDK 17:

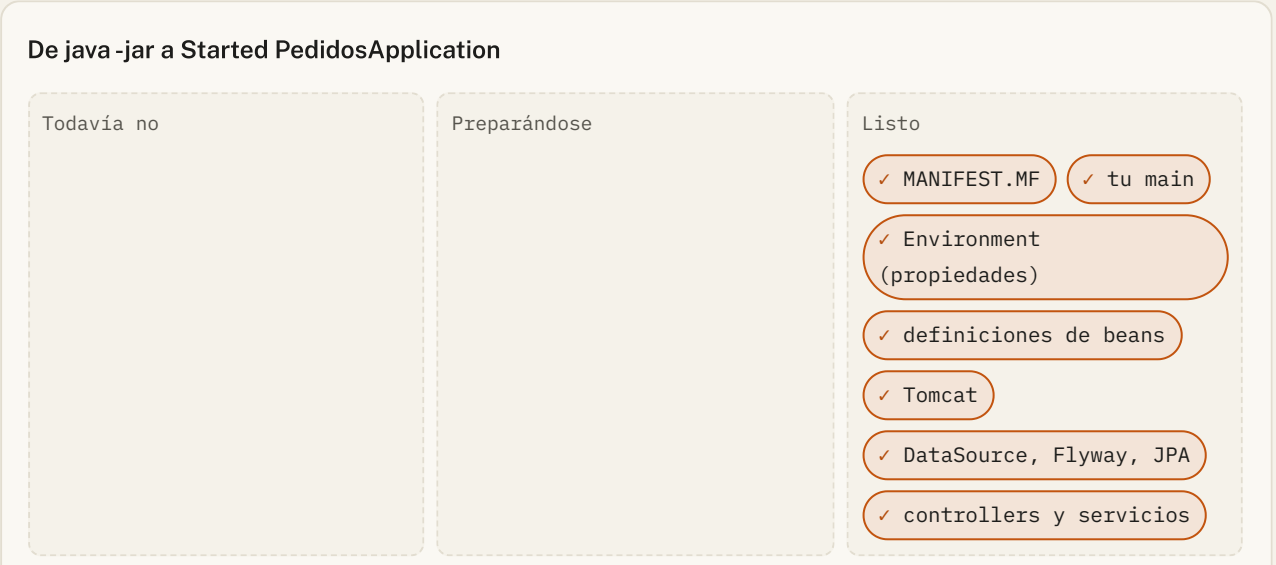
```
Exception in thread "main" java.lang.UnsupportedClassVersionError:
com/ejemplo/pedidos/PedidosApplication has been compiled by a more recent version of
the Java Runtime (class file version 65.0), this version of the Java Runtime only
recognizes class file versions up to 61.0
```

65 es la versión de archivo de clase de Java 21 y 61 la de Java 17. El **`java -version`** de la terminal y el JDK del IDE suelen ser distintos: revisa los dos.

El error típico: el main fuera del paquete raíz. No muestra ningún error, y por eso es el peor. Si mueves **`PedidosApplication`** a **`com.ejemplo.pedidos.arranque`**, la app arranca sin quejarse y **`/hola`** responde 404: **`@SpringBootApplication`** escanea su paquete y los de abajo, y **`com.ejemplo.pedidos.hola`** quedó afuera. Lo comprobé con un test: así da 404 y, con el escaneo alcanzando al controller, 200. La referencia recomienda la clase principal en el paquete raíz, porque también define dónde se buscan las entidades JPA (Spring Boot Reference, 4.1).

Qué pasa entre `java -jar` y «Started»

Esto ocurre detrás de esas líneas con la app completa de esta guía, la que tiene base de datos:



Escribes java -jar pedidos.jar. La JVM no sabe nada de Spring: abre el jar y lee META-INF/MANIFEST.MF para saber qué clase ejecutar.

El manifiesto dice Main-Class: JarLauncher, el lanzador de Boot. JarLauncher arma el classpath con BOOT-INF/classes y los jars de BOOT-INF/lib, y llama a la Start-Class: tu PedidosApplication.main.

SpringApplication.run prepara el Environment: lee application.yml, las variables de entorno y los argumentos, y decide los perfiles activos. Es la línea «No active profile set».

Crea el ApplicationContext y registra las definiciones de beans: las de tu código, que encuentra escaneando desde el paquete de PedidosApplication, y las de las autoconfiguraciones cuyas condiciones se cumplen. Todavía no existe ningún objeto.

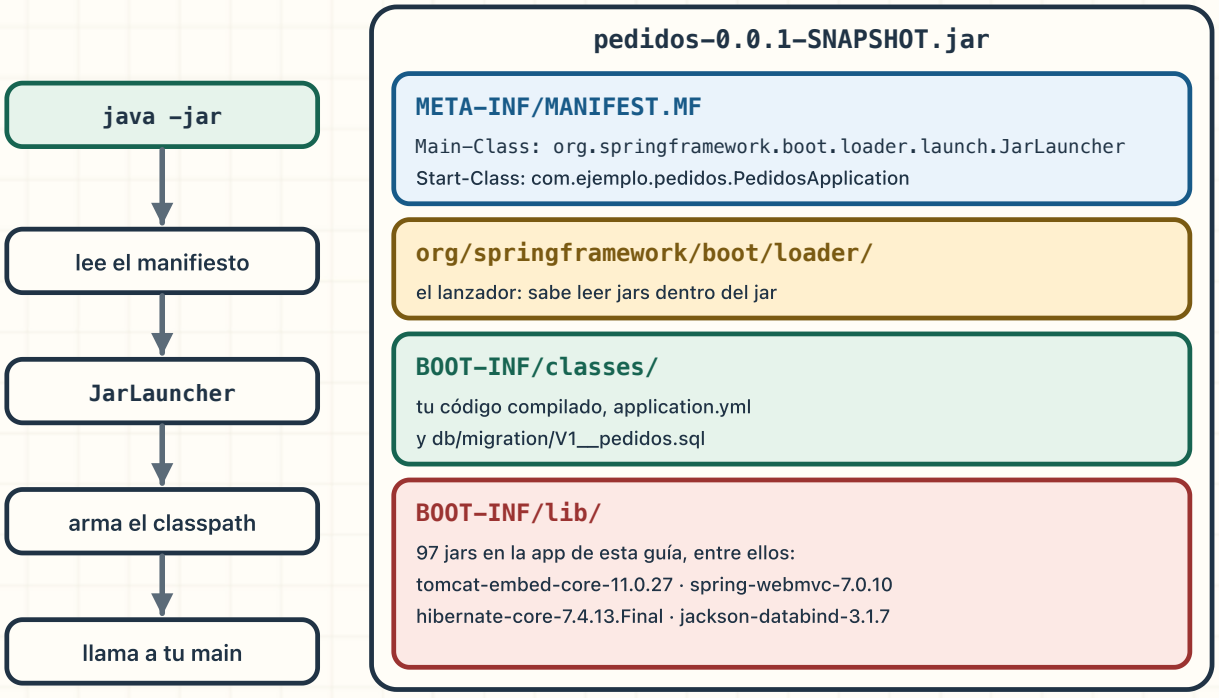
Antes de crear tus beans, crea el servidor web: «Tomcat initialized with port 8080». Existe, pero todavía no acepta conexiones.

Crea los singletons, como en la Parte 1: el pool de conexiones (HikariPool-1 - Start completed), Flyway aplica las migraciones pendientes, Hibernate arma el EntityManagerFactory y valida el esquema. Si algo falla aquí, la app no arranca.

Con todos los beans creados, arranca Tomcat: «Tomcat started on port 8080». Recién ahora entra el primer request, y ya no puede encontrar un bean a medio armar.

«Started PedidosApplication in 2.931 seconds». run devuelve el contexto y tu main termina, pero el proceso sigue vivo: los hilos de Tomcat lo mantienen escuchando.

Lo que viaja dentro del jar, listado con **`unzip -l`**:



Un jar con jars adentro: tu código en BOOT-INF/classes, las dependencias en BOOT-INF/lib y un lanzador que sabe leerlos.

Un jar normal no puede llevar otros jars en su classpath; por eso Boot trae su propio lanzador. Tus clases van en **`BOOT-INF/classes`** y las dependencias en **`BOOT-INF/lib`** (Spring Boot, *The Executable Jar Format*).

`@SpringBootApplication`, desarmada

Son tres anotaciones (Spring Boot Reference, 4.1):

- **`@SpringBootConfiguration`**: la clase es una **`@Configuration`** y puede declarar **`@Bean`**; los tests la usan para encontrar la configuración de la app.
- **`@EnableAutoConfiguration`**: activa la autoconfiguración, la próxima sección.
- **`@ComponentScan`**: escanea el paquete de la clase y los de abajo. Es la razón del tercer error típico.

Starters, servidor embebido y el BOM

pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.2</version>
    <relativePath/>
</parent>

<groupId>com.ejemplo</groupId>
<artifactId>pedidos</artifactId>
<version>0.0.1-SNAPSHOT</version>

<properties>
    <java.version>21</java.version>
</properties>
```

`spring-boot-starter-parent` es el **padre** del proyecto: le hereda configuración de plugins y la **gestión de dependencias**. Boot publica una lista curada de bibliotecas con versiones probadas juntas, un BOM (*bill of materials*) llamado **`spring-boot-dependencies`**; por eso las dependencias no llevan **`<version>`**, y al subir Boot suben todas juntas (Spring Boot Reference, 4.1).

Un **starter** es una dependencia sin código propio que trae todo lo necesario para una tecnología. Estos son los de la app completa:

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-flyway</artifactId>
</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-database-postgresql</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-restclient</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Si vienes de Boot 3: Boot 4 se reorganizó en módulos y renombró starters. El de la web es **`spring-boot-starter-webmvc`**; **`spring-boot-starter-web`** sigue existiendo, marcado como obsoleto (Spring Boot Reference, 4.1). Flyway ahora necesita **`spring-boot-starter-flyway`**, y los tests usan un starter de test por tecnología, como **`spring-boot-starter-webmvc-test`** (Spring Boot 4.0 Migration Guide). Es lo que generó start.spring.io al elegir «Spring Web».

El **servidor embebido** llega con el starter: **`spring-boot-starter-webmvc`** trae Tomcat como una dependencia más (**`tomcat-embed-core-11.0.27.jar`**, en **`BOOT-INF/lib`**). El servidor ya no es un lugar al que despliegas: es una biblioteca que tu app usa.

> **El error típico:** pisar la versión de una biblioteca que gestiona el BOM «porque salió una más nueva». A veces hace falta, por un parche de seguridad, pero esa combinación deja de ser la que Boot probó. Para Spring Framework la referencia recomienda enfáticamente no fijar su versión (Spring Boot Reference, 4.1). Lo normal es subir la versión de Boot.

> **Lo que debes llevarte:**

- **Un jar, un comando:** `java -jar` arranca tu código, sus dependencias y el servidor.
- **El main va en el paquete raíz,** o el escaneo no encuentra tus clases y nada te avisa.
- **Starters y BOM:** eliges tecnologías, no versiones.

– Para profundizar

el jar trae dos índices: **`BOOT-INF/classpath.idx`**, que fija el orden del classpath con **`java -jar`**, y **`BOOT-INF/layers.idx`**, que lo separa en capas para construir imágenes de contenedor (Spring Boot, *The Executable Jar Format*). Las capas las usamos en la Parte 3. En el log de la app completa, «Tomcat initialized» aparece antes de abrir el pool de conexiones y «Tomcat started» después de que Flyway migró y Hibernate validó: el servidor se crea temprano, pero no acepta tráfico hasta que existen todos los beans.

Autoconfiguración sin magia

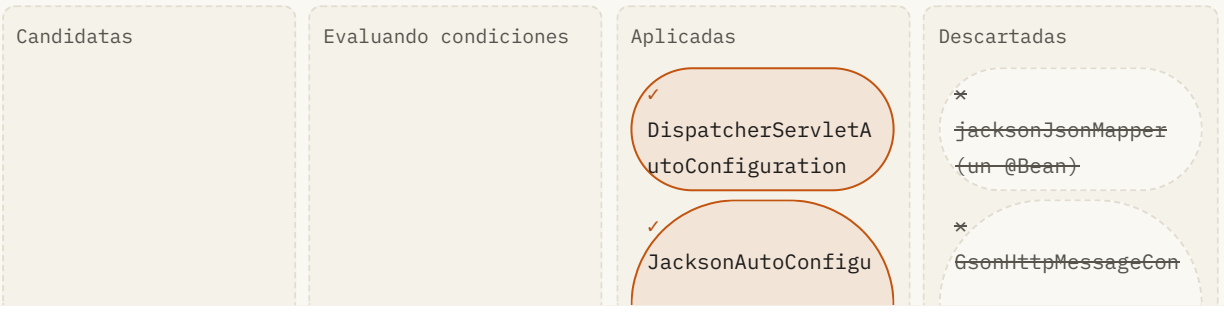
Que Boot cree un **`DataSource`** sin que escribas una línea parece magia. El día que configure algo que no querías, vas a necesitar saber de dónde salió.

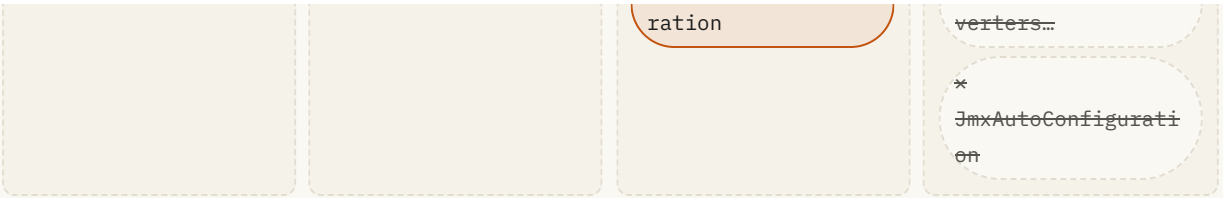
Una **autoconfiguración** es una clase **`@Configuration`** común, con dos diferencias: viene en un jar de Boot, registrada en **`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`**, y está llena de **condiciones** que dicen cuándo aplicarla (Spring Boot Reference, 4.1). Las dos que más vas a ver:

- **`@ConditionalOnClass`**: aplica solo si cierta clase está en el classpath, o sea, si agregaste esa biblioteca.
- **`@ConditionalOnMissingBean`**: crea el bean solo si **tú** no declaraste uno del mismo tipo. Tiene un valor por defecto, pero el tuyo manda.

Hay más (**`@ConditionalOnProperty`**, **`@ConditionalOnBean`**...), pero con esas dos se entiende el mecanismo:

Boot decide qué autoconfiguraciones aplicar en la primera app





Al arrancar, Boot lee la lista de candidatas de los jars que tienes en el classpath. Son decenas; tomamos cinco del informe real de la primera app, la que solo tiene Spring Web.

DispatcherServletAutoConfiguration pide @ConditionalOnClass DispatcherServlet y una aplicación web. El starter webmvc trajo las dos cosas: se aplica.

JacksonAutoConfiguration pide @ConditionalOnClass tools.jackson.databind.json.JsonMapper. Jackson 3 viene con el starter: se aplica.

Dentro, el @Bean jacksonJsonMapper tiene @ConditionalOnMissingBean JsonMapper. Tu código no declaró ninguno: el informe dice «did not find any beans» y Boot crea el suyo.

La configuración de Gson pide @ConditionalOnClass com.google.gson.Gson. No está en el classpath: se descarta sin crear nada.

JmxAutoConfiguration encuentra su clase, pero también pide la propiedad spring.jmx.enabled=true, que nadie puso. Una condición que falla alcanza para descartarla.

Si mañana declaras tu propio JsonMapper, la clase de Jackson se sigue aplicando, pero el @Bean de Boot se hace a un lado: @ConditionalOnMissingBean encuentra el tuyo. Así se reemplaza un bean autoconfigurado.

Ver qué se aplicó y por qué

Arranca la app con **`--debug`** y Boot imprime el **informe de condiciones**: cada autoconfiguración, si se aplicó y qué condición lo decidió (Spring Boot Reference, 4.1). Un recorte de mi salida con la primera app:

```
Positive matches:
-----------------

   JacksonAutoConfiguration#jacksonJsonMapper matched:
      - @ConditionalOnMissingBean (types: tools.jackson.databind.json.JsonMapper;
SearchStrategy: all) did not find any beans (OnBeanCondition)

Negative matches:
-----------------

   GsonHttpMessageConvertersConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required class 'com.google.gson.Gson'
(OnClassCondition)
```

Con la app corriendo, lo mismo está en **`/actuator/conditions`**, en JSON. Requiere **`spring-boot-starter-actuator`** y exponerlo, porque no se publica por defecto; en esta app se expone solo en el perfil **`dev`**:

AplicacionCompletaTest.java

```
@Test
void actuatorExplicaPorQueSeAplicoCadaAutoconfiguracion() throws Exception {
    var json = mvc.perform(get("/actuator/conditions"))
            .andExpect(status().isOk())
            .andReturn()
            .getResponse()
            .getContentAsString();

    assertThat(json).contains("JacksonAutoConfiguration", "positiveMatches",
"negativeMatches");
}
```

Reemplazar un bean autoconfigurado

Dos caminos: ajustarlo con propiedades (**`spring.jackson.*`**, por ejemplo) o declarar tu propio bean del mismo tipo, y **`@ConditionalOnMissingBean`** retira el de Boot (Spring Boot Reference, 4.1). Se prueba sin levantar la app con **`ApplicationContextRunner`**, que arma un contexto mínimo con las autoconfiguraciones que le pidas, aquí la real de Jackson. **`FilteredClassLoader`** esconde una clase del classpath para probar qué pasa si una biblioteca no está:

AutoconfiguracionTest.java

```
private final ApplicationContextRunner contexto =
        new
ApplicationContextRunner().withConfiguration(AutoConfigurations.of(JacksonAutoConfigurat
ion.class));

@Test
void sinNadaTuyoBootTeDaSuJsonMapper() {
    contexto.run(app -> assertThat(app).hasSingleBean(JsonMapper.class));
}

@Test
void siDeclarasElTuyoLaAutoconfiguracionSeHaceAUnLado() {
    contexto.withUserConfiguration(MiJson.class).run(app -> {
        assertThat(app).hasSingleBean(JsonMapper.class);
        assertThat(app.getBean(JsonMapper.class)).isSameAs(app.getBean("miJsonMapper"));
    });
}

@Test
void sinJacksonEnElClasspathNoHayJsonMapper() {
    contexto.withClassLoader(new FilteredClassLoader(JsonMapper.class))
            .run(app -> assertThat(app).doesNotHaveBean(JsonMapper.class));
}

@Configuration(proxyBeanMethods = false)
```

```
static class MiJson {
    @Bean
    JsonMapper miJsonMapper() {
        return JsonMapper.builder().build();
    }
}
```

> **El error típico:** declarar tu propio bean para «ajustar un detalle» y perder todo lo demás que configuraba Boot. Definir tu propio **`JsonMapper`** desactiva toda su autoconfiguración (Spring Boot How-to, 4.1): si lo construyes desde cero, las propiedades **`spring.jackson.*`** dejan de aplicarse. Para un detalle, la propiedad o un *customizer* (**`JsonMapperBuilderCustomizer`**).

> – Para profundizar
>
> en Boot 4 la autoconfiguración de cada tecnología viaja en su módulo (**`DataSourceAutoConfiguration`** está en **`spring-boot-jdbc`**). Sin el starter ni siquiera es candidata: en el informe de la primera app no hay ninguna línea sobre **`DataSource`**. Para excluir una, **`@SpringBootApplication(exclude = …)`** o **`spring.autoconfigure.exclude`**. De esas clases, lo único que la referencia considera API es su nombre, para poder excluirlas: no las importes ni las extiendas.

Propiedades y perfiles

En la Parte 1 viste el **`Environment`**, que reúne propiedades de varias fuentes. Boot le agrega archivos de configuración que lee solo y un orden de prioridad definido entre esas fuentes. Ese orden responde la pregunta de **`server.port`**: si el valor está en el archivo y en la línea de comandos, ¿cuál gana?

`.properties` o `.yml`

Boot lee **`application.properties`** o **`application.yml`** de **`src/main/resources`**. Lo mismo en las dos formas:

```
pedidos.limite-lineas=20
pedidos.moneda=EUR
```

```
pedidos:
  limite-lineas: 20
  moneda: EUR
```

YAML evita repetir prefijos y admite varios documentos en un archivo, separados por `---`, que es como se escriben los perfiles. Elige uno: si conviven en el mismo lugar, **`.properties`** gana (Spring Boot Reference, 4.1). La configuración completa de la app:

application.yml

```yaml
spring:
  application:
    name: pedidos
  datasource:
    url: jdbc:postgresql://localhost:5432/pedidos
    username: pedidos
    password: ${PEDIDOS_DB_PASSWORD:pedidos}
  jpa:
    open-in-view: false
    hibernate:
      ddl-auto: validate
  http:
    clients:
      connect-timeout: 2s
      read-timeout: 3s
    serviceclient:
      tarifas:
        base-url: ${tarifas.base-url}
        read-timeout: 2s

pedidos:
  limite-lineas: 20
  moneda: EUR
  ventana-cancelacion: 30m

tarifas:
  base-url: https://tarifas.ejemplo.com
```

Quién gana: el orden de precedencia

La referencia de Boot 4.1 lista quince fuentes en orden, y **las posteriores pisan a las anteriores** (Spring Boot Reference, 4.1). Estas son las de todos los días:



Subconjunto de las 15 fuentes que lista la referencia de Boot 4.1. Los valores son los de PropiedadesTest.

Cuanto más cerca del momento de ejecutar, más prioridad: el archivo empaquetado pone los valores por defecto y el entorno los ajusta.

El **`application.yml`** empaquetado trae valores por defecto, el servidor los ajusta con variables de entorno y quien arranca la app puede pisarlos con un argumento. Así el mismo jar corre en tu máquina, en staging y en producción sin recompilarse. Los valores del diagrama los comprobé con un test: 20 con el archivo solo, 50 con el perfil **`dev`**, 7 con la variable **`PEDIDOS_LIMITELINEAS=7`** y 5 con **`--pedidos.limite-lineas=5`**.

Perfiles: un archivo, varios ambientes

Un **perfil** es un nombre (**`dev`**, **`prod`**…) que activa configuración extra, con **`--spring.profiles.active=dev`** por ejemplo. En la Parte 1 activaban beans con **`@Profile`**; aquí activan también propiedades. Un documento del YAML que solo aplica en un perfil se marca con **`spring.config.activate.on-profile`**:

application.yml

```yaml
spring:
  config:
    activate:
      on-profile: dev
    import: optional:file:./pedidos-local.yml
pedidos:
  limite-lineas: 50
logging:
  level:
```

```
    com.ejemplo.pedidos: debug
    org.hibernate.SQL: debug
management:
  endpoints:
    web:
      exposure:
        include: health,info,conditions
```

- **`spring.config.import`** carga otro archivo. Con **`optional:`**, si no existe la app arranca igual; sin él, no arranca (Spring Boot Reference, 4.1). Sirve para ajustes personales fuera del repositorio.
- **Grupos de perfiles.** Con **`spring.profiles.group.produccion`** y la lista **`proddb`**, **`prodmq`**, activar **`produccion`** activa los tres (Spring Boot Reference, 4.1).
- **`spring.profiles.active` no va dentro de un documento de perfil**: ahí es inválido (Spring Boot Reference, 4.1).

`@Value` o `@ConfigurationProperties`

Con **`@Value("${…}")`**, cada propiedad es un texto que nadie verifica, inyectado por separado. Con **`@ConfigurationProperties`**, un grupo de propiedades con el mismo prefijo se vuelca en un objeto con tipos:

ANTES

LimitesConValue.java

```
class LimitesConValue {

    final int limiteLineas;
    final String moneda;
    final Duration ventanaCancelacion;

    LimitesConValue(
            @Value("${pedidos.limite-
lineas}") int limiteLineas,
            @Value("${pedidos.moneda}")
String moneda,
            @Value("${pedidos.ventana-
cancelacion}") Duration
ventanaCancelacion) {
        this.limiteLineas = limiteLineas;
        this.moneda = moneda;
        this.ventanaCancelacion =
ventanaCancelacion;
    }
}
```

DESPUÉS

PedidosProperties.java

```
@ConfigurationProperties("pedidos")
@Validated
public record PedidosProperties(
        @Min(1) @Max(100) int
limiteLineas,
        @NotBlank String moneda,
        Duration ventanaCancelacion) {}
```

Un **record** se llena por su constructor (Spring Boot Reference, 4.1). Para registrarlo como bean, aquí lo hace **@ConfigurationPropertiesScan**; después se inyecta como cualquier dependencia:

PedidoService.java

```
@Transactional
public PedidoRespuesta crear(NuevoPedido nuevo) {
    if (nuevo.lineas().size() > propiedades.limiteLineas())
        throw new LimiteDeLineasException(propiedades.limiteLineas());
    var pedido = new Pedido(nuevo.cliente(), nuevo.correo());
    for (var linea : nuevo.lineas()) {
        var precio = catalogo.precio(linea.producto()).orElseThrow();
        pedido.agregarLinea(linea.producto(), linea.cantidad(), precio);
    }
    var guardado = repositorio.save(pedido);
    notificador.pedidoCreado(guardado);
    return PedidoRespuesta.de(guardado);
}
```

(Aquí el **save** sí hace falta porque el pedido es nuevo; más adelante verás cuándo no.)

Con **@Value**, un límite de 0 arranca sin quejarse (lo comprobé con un test) y el problema aparece con el primer pedido. Con **@Validated**, la app no llega a levantar:

PropiedadesTest.java

```
@Test
void unValorInvalidoNoDejaArrancarLaApp(CapturedOutput salida) {
    assertThatThrownBy(() -> arrancar(Map.of(), "--pedidos.limite-lineas=0"))
            .hasRootCauseInstanceOf(BindValidationException.class);
    assertThat(salida).contains("APPLICATION FAILED TO START", "Property:
pedidos.limiteLineas");
}
```

```
APPLICATION FAILED TO START

Description:

Binding to target com.ejemplo.pedidos.config.PedidosProperties failed:

    Property: pedidos.limiteLineas
    Value: "0"
    Origin: "pedidos.limite-lineas" from property source "commandLineArgs"
    Reason: must be greater than or equal to 1
```

Origin dice de qué fuente salió el valor malo, justo lo que necesitas con quince fuentes posibles. (**Reason** sale en el idioma de la JVM.)

Característica	`@ConfigurationProperties`	`@Value`
Varias propiedades en un objeto	sí	no
Validación al arrancar	sí, con `@Validated`	no
Relaxed binding	sí	limitado
Expresiones SpEL	no	sí

Relaxed binding y variables de entorno

Los sistemas operativos no aceptan puntos en los nombres de variables, así que Boot traduce: los puntos pasan a guion bajo, los guiones se borran y todo va en mayúsculas (Spring Boot Reference, 4.1). `spring.datasource.password` se configura con `SPRING_DATASOURCE_PASSWORD`. Esa tolerancia entre formas del mismo nombre se llama **relaxed binding** y funciona completa con `@ConfigurationProperties`; con `@Value`, la referencia recomienda escribir el nombre en minúsculas y con guiones.

Los secretos, fuera del repositorio

La contraseña de producción no va en `application.yml`, que está en git y viaja dentro del jar. Va en una variable de entorno del servidor o en el gestor de secretos de tu plataforma. En el YAML de la app aparece como `${PEDIDOS_DB_PASSWORD:pedidos}`: toma esa variable de entorno y, si no existe, usa `pedidos`, que solo sirve para el Postgres de tu máquina.

> El error típico: commitear un `application-prod.yml` con la contraseña real «solo para probar». Aunque lo borres después, queda en la historia de git. Si pasó, la contraseña se rota.

> Lo que debes llevarte:
>
> - **Gana la fuente más cercana a la ejecución**: argumentos, variables de entorno, archivos.
> - **`@ConfigurationProperties` con un record y `@Validated`**: tipos, un solo lugar y error al arrancar.
> - **Secretos en el entorno**, nunca en el repositorio.

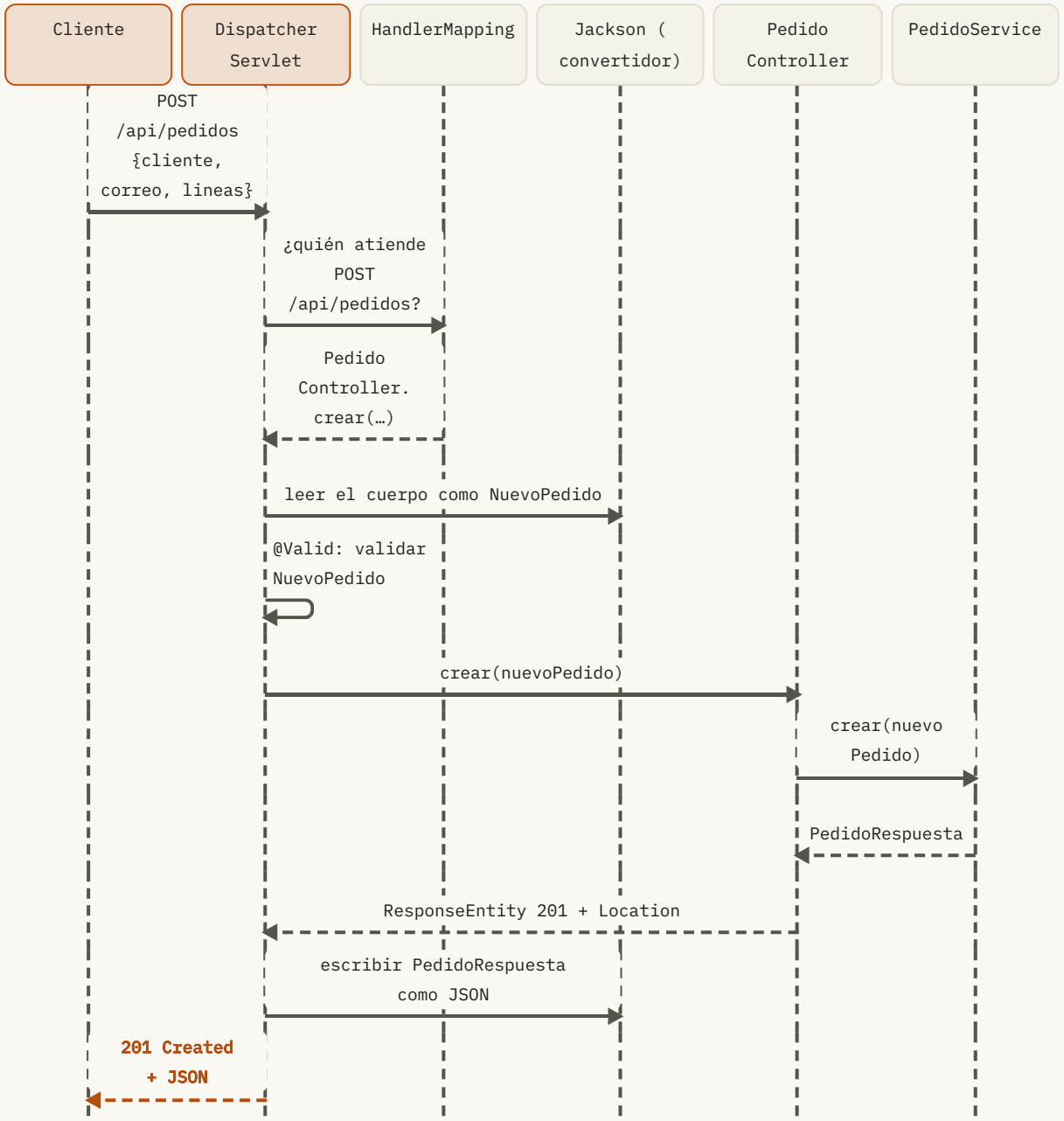
> – Para profundizar
>
> entre las variables de entorno y la línea de comandos están las propiedades del sistema (`-D…`), JNDI, los parámetros del servlet y `SPRING_APPLICATION_JSON`. Por encima, las de los tests: `properties` de `@SpringBootTest`, `@DynamicPropertySource` y `@TestPropertySource`; y, con DevTools activo, sus ajustes globales en `$HOME/.config/spring-boot` (Spring Boot Reference, 4.1). Por eso un test puede pisar cualquier valor.

Spring MVC por dentro

El recorrido de un request

Todo request que llega a Tomcat termina en el **`DispatcherServlet`**, el servlet de Spring MVC que hace de recepcionista único (el patrón *front controller*). No sabe nada de pedidos: pregunta a otros beans quién atiende cada ruta, cómo convertir el cuerpo y qué hacer si algo falla (Spring Framework Reference, 7.0). Este es el **`POST /api/pedidos`**, mensaje por mensaje:

POST /api/pedidos, del cliente a la base y de vuelta



Tomcat recibe el request y se lo entrega al DispatcherServlet. Todos los requests de la app pasan por aquí.

El DispatcherServlet no conoce tus controllers. Le pregunta al HandlerMapping, que al arrancar registró cada @GetMapping y @PostMapping.

Devuelve el método que coincide con la ruta y el verbo, junto con los interceptores que apliquen. Si ninguna ruta coincide, el resultado es un 404; si la ruta existe con otro verbo, un 405.

Antes de llamar al método hay que armar sus argumentos. Para el parámetro con @RequestBody, un HttpMessageConverter con Jackson convierte el JSON en el record NuevoPedido.

Como el parámetro tiene @Valid, se valida ahora. Si falla, el método nunca se llama: se lanza MethodArgumentNotValidException y el manejador de errores responde 400.

Recién ahora se ejecuta tu código, con los argumentos ya convertidos y validados.

El controller no decide nada del negocio: delega en el servicio, que abre la transacción, aplica las reglas y guarda.

El servicio devuelve un DTO, no la entidad. La transacción ya se confirmó.

El controller decide el código HTTP y los encabezados: 201 Created con la URL del pedido nuevo.

El mismo convertidor, en sentido contrario: el record se vuelve JSON.

Y la respuesta sale. Si cualquier paso hubiera lanzado una excepción, el DispatcherServlet se la habría pasado a los HandlerExceptionResolver, que es donde vive tu @RestControllerAdvice.

– **Para profundizar**

simplifiqué un actor. Quien invoca al controller y resuelve sus argumentos es un **`HandlerAdapter`**, que aísla al **`DispatcherServlet`** de cómo se llama a cada tipo de handler; en los controllers anotados, la respuesta se escribe dentro de ese adapter (Spring Framework Reference, 7.0). Los argumentos los resuelven **`HandlerMethodArgumentResolver`**, y ahí puedes enchufar los tuyos.

El controller de la API

PedidoController.java

```java
@RestController
@RequestMapping("/api/pedidos")
public class PedidoController {

    private final PedidoService servicio;

    public PedidoController(PedidoService servicio) {
        this.servicio = servicio;
    }

    @GetMapping("/{id}")
    public PedidoRespuesta buscar(@PathVariable long id) {
        return servicio.buscar(id);
```

```
    }

    @GetMapping
    public Pagina<ResumenPedido> listar(@PageableDefault(size = 20, sort = "creadoEn")
Pageable pagina) {
        return Pagina.de(servicio.listar(pagina));
    }

    @PostMapping
    public ResponseEntity<PedidoRespuesta> crear(@Valid @RequestBody NuevoPedido nuevo)
{
        var creado = servicio.crear(nuevo);
        return ResponseEntity.created(URI.create("/api/pedidos/" +
creado.id())).body(creado);
    }

    @PostMapping("/{id}/pago")
    public ResponseEntity<Void> pagar(@PathVariable long id) {
        servicio.pagar(id);
        return ResponseEntity.noContent().build();
    }

    @PostMapping("/{id}/cancelacion")
    public ResponseEntity<Void> cancelar(@PathVariable long id) {
        servicio.cancelar(id);
        return ResponseEntity.noContent().build();
    }
}
```

Las anotaciones de los parámetros dicen **de dónde** sale cada dato:

Anotación	De dónde lo toma	Ejemplo
`@PathVariable`	un trozo de la ruta	**`/api/pedidos/42` → `id = 42`**
`@RequestParam`	la *query string*	**`/hola?nombre=Ana`**
`@RequestBody`	el cuerpo, convertido de JSON	el **`NuevoPedido`** del POST

Si el dato no se puede convertir (un **`id`** que no es número), Spring responde 400 sin llegar a tu método. El **`Pageable`** de **`listar`** lo arma Spring Data a partir de **`?page=`**, **`?size=`** y **`?sort=`**.

Códigos HTTP: decir lo que pasó

Un método que devuelve un objeto responde 200. Para otro resultado, **`ResponseEntity`** te deja elegir el código y los encabezados:

Código	Cuándo	En la app
`200 OK`	devuelves lo que se pidió	**`GET /api/pedidos/{id}`**
`201 Created`	creaste algo; **`Location`** dice dónde quedó	**`POST /api/pedidos`**
`204 No Content`	se hizo, y no hay nada que devolver	**`POST /api/pedidos/{id}/pago`**
`400 Bad Request`	el cliente mandó algo mal formado o inválido	validación
`404 Not Found`	no existe	un pedido que no está
`409 Conflict`	choca con el estado actual	pagar un pedido ya pagado
`422 Unprocessable Content`	se entendió, pero una regla del negocio lo impide	más líneas que el límite
`500 Internal Server Error`	falló algo de tu lado	lo inesperado

> El error típico: responder **`200`** con un cuerpo **`{"error": "no existe"}`**. Los clientes, los proxies, los monitores y los reintentos automáticos leen el código, no el cuerpo: para ellos fue un éxito.

JSON con Jackson 3

La conversión entre objetos y JSON la hace **Jackson**, y Boot 4 trae **Jackson 3** por defecto (Spring Boot Reference, 4.1). Salió el 3 de octubre de 2025 y cambia cuatro cosas que vas a notar (Jackson Release 3.0):

- **El paquete** pasa de **`com.fasterxml.jackson`** a **`tools.jackson`**, y Boot configura un **`JsonMapper`**, subclase del **`ObjectMapper`** de Jackson 3 (**`tools.jackson.databind.ObjectMapper`**), que no es el mismo tipo que el de Jackson 2. Las anotaciones (**`@JsonProperty`**, **`@JsonIgnore`**…) siguen en **`com.fasterxml.jackson.annotation`**.
- **Las excepciones** son *unchecked*: **`JacksonException`** extiende **`RuntimeException`**.
- **Las fechas** de **`java.time`** vienen incluidas y se escriben como texto ISO-8601.
- **Las propiedades** salen en orden alfabético: **`MapperFeature.SORT_PROPERTIES_ALPHABETICALLY`** ahora vale **`true`**. Por eso el **`ProblemDetail`** de más abajo empieza por **`detail`**.

Un **`Instant`** sale como texto en UTC, y como el DTO no tiene **`correo`**, el JSON tampoco:

PedidoControllerTest.java

```
@Test
void elPedidoSaleComoJsonConLaFechaEnIso8601() throws Exception {
    given(servicio.buscar(42)).willReturn(PEDIDO);

    mvc.perform(get("/api/pedidos/42"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.creadoEn").value("2026-11-04T10:15:30Z"))
            .andExpect(jsonPath("$.lineas[0].precioUnitario").value(8.50))
```

```
                .andExpect(jsonPath("$.correo").doesNotExist());
}
```

Lo demás se ajusta con **`spring.jackson.*`**: **`default-property-inclusion=non_null`** omite los nulos y **`use-jackson2-defaults=true`** acerca el comportamiento al de Boot con Jackson 2, útil al migrar (Spring Boot How-to, 4.1). Para renombrar un campo, **`@JsonProperty("nombre")`** en el componente del record.

DTOs, no entidades

Un **DTO** (*data transfer object*) es un objeto que solo existe para viajar: lo que entra y sale de tu API. Aquí son records: **`NuevoPedido`** entra, **`PedidoRespuesta`** sale.

PedidoRespuesta.java

```
public record PedidoRespuesta(
        Long id, String cliente, EstadoPedido estado, Instant creadoEn, List<Linea>
lineas, BigDecimal total) {

    public record Linea(String producto, int cantidad, BigDecimal precioUnitario) {}

    static PedidoRespuesta de(Pedido pedido) {
        var lineas = pedido.getLineas().stream()
                .map(l -> new Linea(l.getProducto(), l.getCantidad(),
l.getPrecioUnitario()))
                .toList();
        var total = pedido.getLineas().stream()
                .map(LineaPedido::subtotal)
                .reduce(BigDecimal.ZERO, BigDecimal::add);
        return new PedidoRespuesta(
                pedido.getId(), pedido.getCliente(), pedido.getEstado(),
pedido.getCreadoEn(), lineas, total);
    }
}
```

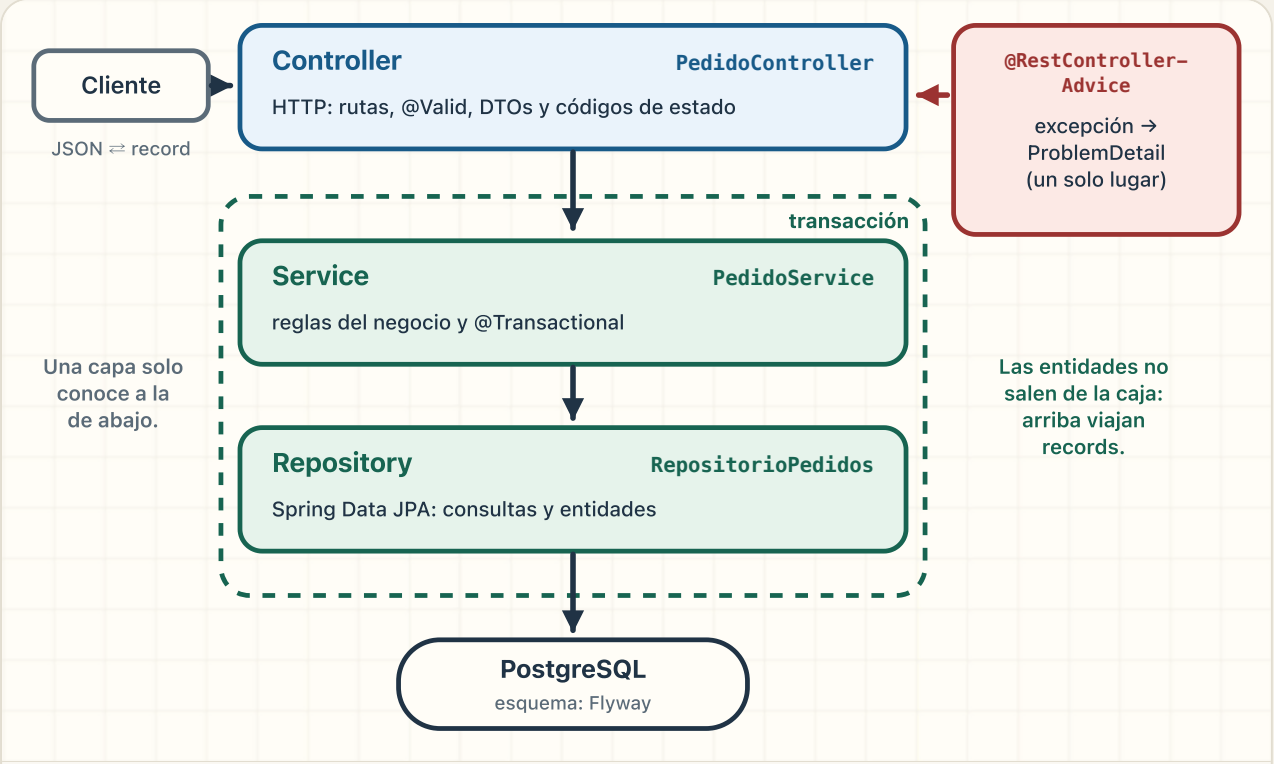
¿Por qué no devolver la entidad **`Pedido`**? Tres razones de producción:

- **Relaciones lazy.** Las líneas se cargan cuando alguien las pide. Si Jackson serializa la entidad fuera de la transacción, ese acceso falla o dispara consultas mientras escribe el JSON.
- **Recursión.** Cada **`LineaPedido`** apunta a su **`Pedido`**: serializar eso es un ciclo.
- **Exponer de más.** La entidad tiene el correo del cliente, y cualquier columna nueva saldría publicada sin que nadie lo decida.

Con un DTO el contrato lo escribes tú: sin correo y con un **`total`** que no está en la tabla. La conversión ocurre en el servicio, con la transacción abierta:

PedidoService.java

```java
@Transactional(readOnly = true)
public PedidoRespuesta buscar(long id) {
    return repositorio.findById(id)
            .map(PedidoRespuesta::de)
            .orElseThrow(() -> new PedidoNoEncontradoException(id));
}
```



Cada capa conoce solo a la de abajo; las entidades se quedan dentro de la transacción y hacia arriba viajan records.

Documentar la API con OpenAPI

OpenAPI es un formato estándar para describir una API HTTP (rutas, parámetros, cuerpos y respuestas) en un JSON que leen personas y programas. **springdoc-openapi** lo genera desde tus controllers; la línea 3.x es la de Boot 4:

pom.xml

```xml
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>3.2.0</version>
</dependency>
```

El contrato queda en **`/v3/api-docs`** y una página para probarlo en **`/swagger-ui.html`** (springdoc), con las validaciones incluidas: **`NuevoPedido`** aparece con sus campos obligatorios. Al arrancar, springdoc avisa con un **`WARN`** que esos endpoints están activos por defecto y cómo apagarlos en producción (**`springdoc.api-docs.enabled=false`**).

> Nota: En este sitio, el backend usa la misma dependencia (**`springdoc-openapi-starter-webmvc-ui`** 3.2.0). El contrato OpenAPI que genera alimenta al cliente TypeScript del frontend, y un test lo compara con una copia guardada para detectar cambios que romperían al frontend.

> Lo que debes llevarte:
>
> - Todo request pasa por el **`DispatcherServlet`**, que delega: quién atiende, cómo convertir, qué hacer si falla.
> - El código HTTP es parte de la respuesta.
> - Records como DTOs, nunca entidades en la respuesta.

Validación

Todo lo que entra a tu API viene de afuera. Validar con **`if`** en cada controller termina en reglas repetidas y distintas por endpoint. **Bean Validation** (**`jakarta.validation`**) las declara como anotaciones sobre los datos, y Boot la activa con **`spring-boot-starter-validation`**:

NuevoPedido.java

```
public record NuevoPedido(
        @NotBlank @Size(max = 120) String cliente,
        @NotBlank @Email String correo,
        @NotEmpty List<@Valid Linea> lineas) {

    public record Linea(@ProductoEnCatalogo String producto, @Positive int cantidad) {}
}
```

`@NotBlank` (no nulo, ni vacío, ni solo espacios), **`@Email`**, **`@Size`**, **`@NotEmpty`** (al menos un elemento), **`@Positive`**. Las dispara el **`@Valid`** del parámetro del controller; el **`@Valid`** de **`List<@Valid Linea>`** valida también cada línea.

Un validador propio

Que un producto exista en el catálogo depende de tus datos, y ninguna anotación estándar lo cubre. Una restricción propia tiene dos piezas, la anotación y la clase que la evalúa:

ProductoEnCatalogo.java

```
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.RECORD_COMPONENT})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ProductoEnCatalogoValidator.class)
public @interface ProductoEnCatalogo {
    String message() default "no está en el catálogo";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

ProductoEnCatalogoValidator.java

```
public class ProductoEnCatalogoValidator implements
ConstraintValidator<ProductoEnCatalogo, String> {

    private final Catalogo catalogo;

    public ProductoEnCatalogoValidator(Catalogo catalogo) {
        this.catalogo = catalogo;
    }

    @Override
    public boolean isValid(String producto, ConstraintValidatorContext contexto) {
        return producto != null && catalogo.existe(producto);
    }
}
```

El validador recibe el **`Catalogo`** por constructor: Spring crea los validadores con su propia fábrica, así que tienen inyección de dependencias (Spring Framework Reference, 7.0). Un pedido con cuatro errores, en un solo 400:

PedidoControllerTest.java

```
@Test
void unPedidoInvalidoDevuelve400ConCadaCampoQueFalla() throws Exception {
    mvc.perform(post("/api/pedidos")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("""
                            {"cliente":"","correo":"no-es-correo","lineas":
[{"producto":"tostadora","cantidad":0}]}
                            """))
            .andExpect(status().isBadRequest())
            .andExpect(content().contentType(MediaType.APPLICATION_PROBLEM_JSON))
            .andExpect(jsonPath("$.title").value("Datos inválidos"))
            .andExpect(jsonPath("$.errores.cliente").exists())
            .andExpect(jsonPath("$.errores.correo").exists())
```

```
            .andExpect(jsonPath("$.errores['lineas[0].producto']").value("no está en el
catálogo"))
            .andExpect(jsonPath("$.errores['lineas[0].cantidad']").exists());
}
```

El error típico: olvidar **`@Valid`**. El código compila y el endpoint acepta cualquier cosa. Si una regla «no funciona», mira primero el **`@Valid`** del parámetro.

– Para profundizar

el precio no está en **`NuevoPedido`** a propósito. Si lo mandara el cliente, ninguna validación evitaría una camiseta a un centavo: lo pone el servidor desde el **`Catalogo`**. Validar la forma de los datos no reemplaza decidir de qué datos te fías.

Errores que el cliente entiende

Un cliente de tu API necesita errores predecibles: el mismo formato, el código HTTP correcto y un mensaje útil, sin detalles internos. El camino que casi todos recorremos primero, y adónde conviene llegar:

ANTES

PedidoControllerConTryCatch.java

```
@RestController
@RequestMapping("/api/pedidos")
class PedidoControllerConTryCatch {

    private static final Logger log =
LoggerFactory.getLogger(PedidoControllerC
onTryCatch.class);
    private final PedidoService servicio;


PedidoControllerConTryCatch(PedidoService
servicio) {
        this.servicio = servicio;
    }

    @GetMapping("/{id}")
    ResponseEntity<?>
buscar(@PathVariable long id) {
```

DESPUÉS

ManejadorDeErrores.java

```
@RestControllerAdvice
public class ManejadorDeErrores extends
ResponseEntityExceptionHandler {

    private static final Logger log =
LoggerFactory.getLogger(ManejadorDeErrore
s.class);


@ExceptionHandler(PedidoNoEncontradoExcep
tion.class)
    ProblemDetail
noEncontrado(PedidoNoEncontradoException
e) {
        return
problema(HttpStatus.NOT_FOUND, "Pedido no
encontrado", e.getMessage());
    }
```

```
        try {
            return
ResponseEntity.ok(servicio.buscar(id));
        } catch
(PedidoNoEncontradoException e) {
            return
ResponseEntity.status(404).body(e.getMess
age());
        } catch (Exception e) {
            log.error("Error buscando el
pedido " + id, e);
            return
ResponseEntity.status(500).body(e.toStrin
g());
        }
    }

    @PostMapping("/{id}/cancelacion")
    ResponseEntity<?>
cancelar(@PathVariable long id) {
        try {
            servicio.cancelar(id);
            return
ResponseEntity.noContent().build();
        } catch
(TransicionInvalidaException e) {
            return
ResponseEntity.badRequest().body(Map.of("
error", e.getMessage()));
        }
    }
}
```

```
@ExceptionHandler(TransicionInvalidaExcep
tion.class)
    ProblemDetail
transicionInvalida(TransicionInvalidaExce
ption e) {
        return
problema(HttpStatus.CONFLICT, "Transición
no permitida", e.getMessage());
    }


@ExceptionHandler(LimiteDeLineasException
.class)
    ProblemDetail
limite(LimiteDeLineasException e) {
        return
problema(HttpStatus.UNPROCESSABLE_CONTENT
, "Pedido demasiado grande",
e.getMessage());
    }

    @ExceptionHandler(Exception.class)
    ProblemDetail inesperado(Exception e)
{
        log.error("Error no controlado",
e);
        return
problema(HttpStatus.INTERNAL_SERVER_ERROR
, "Error interno", "Ocurrió un error
inesperado.");
    }

    private static ProblemDetail
problema(HttpStatus estado, String
titulo, String detalle) {
        var problema =
ProblemDetail.forStatusAndDetail(estado,
detalle);
        problema.setTitle(titulo);
        return problema;
    }

    @Override
    protected ResponseEntity<Object>
handleMethodArgumentNotValid(

MethodArgumentNotValidException e,
```

```
HttpHeaders headers, HttpStatusCode
estado, WebRequest request) {
        Map<String, String> errores = new
TreeMap<>();

e.getBindingResult().getFieldErrors().for
Each(f -> errores.put(f.getField(),
f.getDefaultMessage()));
        var problema = e.getBody();
        problema.setTitle("Datos
inválidos");
        problema.setProperty("errores",
errores);
        return handleExceptionInternal(e,
problema, headers, estado, request);
    }
}
```

En el «antes» cada endpoint decide su error, y lo comprobé con un test: el 404 es texto plano, el conflicto de estado es un 400 con un JSON inventado ahí y el 500 le devuelve al cliente el mensaje interno, con el nombre del servidor de base de datos.

En el «después» los controllers no atrapan nada. **`@RestControllerAdvice`** junta los **`@ExceptionHandler`** de toda la app, y Spring usa el del tipo de excepción más específico. Un solo lugar, un solo formato.

ProblemDetail: el formato estándar

RFC 9457, *Problem Details for HTTP APIs* (julio de 2023, reemplaza a la RFC 7807), define un JSON estándar para errores con **`type`**, **`title`**, **`status`**, **`detail`** e **`instance`**, y el tipo **`application/problem+json`** (RFC 9457). Spring lo implementa en **`ProblemDetail`** (Spring Framework Reference, 7.0). La respuesta real para un pedido que no existe:

```
$ curl -i http://localhost:8080/api/pedidos/99
HTTP/1.1 404
Content-Type: application/problem+json

{"detail":"No existe el pedido
99","instance":"/api/pedidos/99","status":404,"title":"Pedido no encontrado"}
```

PedidoControllerTest.java

```
@Test
void unPedidoQueNoExisteEsUn404ConProblemDetail() throws Exception {
    given(servicio.buscar(7)).willThrow(new PedidoNoEncontradoException(7));

    mvc.perform(get("/api/pedidos/7"))
            .andExpect(status().isNotFound())
```

```
            .andExpect(content().contentType(MediaType.APPLICATION_PROBLEM_JSON))
            .andExpect(jsonPath("$.status").value(404))
            .andExpect(jsonPath("$.title").value("Pedido no encontrado"))
            .andExpect(jsonPath("$.detail").value("No existe el pedido 7"))
            .andExpect(jsonPath("$.instance").value("/api/pedidos/7"));
}
```

ManejadorDeErrores extiende **ResponseEntityExceptionHandler**, la clase base de Spring que ya responde con **ProblemDetail** a las excepciones de Spring MVC: JSON mal formado, un parámetro que no se convierte, un método HTTP no soportado (Spring Framework Reference, 7.0). Para la validación, sobrescribe **handleMethodArgumentNotValid** y agrega un mapa **errores** con cada campo que falló:

```
$ curl -X POST http://localhost:8080/api/pedidos -H 'Content-Type: application/json' \
    -d '{"cliente":"","correo":"x","lineas":[{"producto":"tostadora","cantidad":0}]}'
{"detail":"Invalid request
content.","instance":"/api/pedidos","status":400,"title":"Datos inválidos","errores":
{"cliente":"must not be blank","correo":"must be a well-formed email
address","lineas[0].cantidad":"must be greater than 0","lineas[0].producto":"no está en
el catálogo"}}
```

Los mensajes estándar salen en el idioma de la JVM; el de **@ProductoEnCatalogo** es el que escribimos.

> Nota: sin manejador propio, **spring.mvc.problemdetails.enabled=true** hace que los errores de Spring MVC salgan como **ProblemDetail**; viene en **false** (Spring Boot Reference, 4.1). Con excepciones de dominio, el manejador propio es el camino.

Excepciones de dominio o ResponseStatusException

ResponseStatusException lleva el código HTTP adentro. En un controller es cómoda; en un servicio, el servicio pasa a hablar HTTP, y cuando lo llame un job nocturno un **404** no significará nada.

Por eso la app usa **excepciones de dominio** (**PedidoNoEncontradoException**, **TransicionInvalidaException**, **LimiteDeLineasException**), que dicen qué pasó en términos del negocio; la traducción a HTTP vive en el manejador. **Pedido** lanza **TransicionInvalidaException** sin saber que existe HTTP:

Pedido.java

```
public void pagar() {
    if (estado != EstadoPedido.CREADO) throw new TransicionInvalidaException(id, estado,
EstadoPedido.PAGADO);
    estado = EstadoPedido.PAGADO;
}

public void cancelar() {
```

```
    if (estado == EstadoPedido.ENVIADO || estado == EstadoPedido.CANCELADO)
        throw new TransicionInvalidaException(id, estado, EstadoPedido.CANCELADO);
    estado = EstadoPedido.CANCELADO;
}
```

Son *unchecked*, por lo que vimos en la Parte 1 sobre las excepciones a la manera de Spring.

Lo que no se le muestra al cliente

> **El error típico: devolver la traza.** Un stack trace le cuenta a cualquiera qué bibliotecas usas, cómo se llaman tus clases y a veces a qué base te conectas. La RFC 9457 pide no exponer detalles de implementación como un volcado de pila (RFC 9457). El manejador por defecto de Boot ya lo evita (**`spring.web.error.include-stacktrace`** vale **`never`**), pero un **`@ExceptionHandler`** con **`body(e.toString())`** lo rompe. El de la app responde un texto genérico:

PedidoControllerTest.java

```
@Test
void unErrorInesperadoNoLeMuestraLaTrazaAlCliente() throws Exception {
    given(servicio.buscar(1)).willThrow(new IllegalStateException("conexión rota a db-interna:5432"));

    mvc.perform(get("/api/pedidos/1"))
            .andExpect(status().isInternalServerError())
            .andExpect(jsonPath("$.detail").value("Ocurrió un error inesperado."))
            .andExpect(jsonPath("$.trace").doesNotExist())
            .andExpect(content().string(org.hamcrest.Matchers.not(
                    org.hamcrest.Matchers.containsString("db-interna"))));
}
```

> **El error típico: registrar la misma excepción tres veces.** El repositorio la atrapa, la registra y la relanza; el servicio, igual; el manejador la registra otra vez. Tres trazas de un solo error, y quien esté de guardia cree que fueron tres. **O la manejas, o la propagas.** Se registra una vez, en el borde: en la app, **`ManejadorDeErrores`**, y solo lo inesperado como **`ERROR`**.

> Nota: En este sitio, un **`@RestControllerAdvice`** global responde con un formato propio, el record **`ApiError`** (**`code`**, **`message`**, **`timestamp`**), en lugar de **`ProblemDetail`**. Sigue las mismas dos reglas: los 404, 400 y 429 van como **`INFO`**, y lo inesperado se registra una vez como **`ERROR`**, se reporta a Sentry y al cliente le llega un mensaje genérico.

Datos con Spring Data JPA

JPA (*Jakarta Persistence*) es la especificación de Java para mapear objetos a tablas, **Hibernate** es la implementación que trae Boot y **Spring Data JPA** agrega los repositorios: tú escribes una interfaz y Spring genera la clase.

La entidad

Pedido.java

```java
@Entity
@Table(name = "pedidos")
@EntityListeners(AuditingEntityListener.class)
public class Pedido {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String cliente;
    private String correo;

    @Enumerated(EnumType.STRING)
    private EstadoPedido estado = EstadoPedido.CREADO;

    @CreatedDate
    private Instant creadoEn;

    @OneToMany(mappedBy = "pedido", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<LineaPedido> lineas = new ArrayList<>();

    protected Pedido() {}

    public Pedido(String cliente, String correo) {
        this.cliente = cliente;
        this.correo = correo;
    }

    public void agregarLinea(String producto, int cantidad, BigDecimal precioUnitario) {
        lineas.add(new LineaPedido(this, producto, cantidad, precioUnitario));
    }
```

(Siguen los métodos de estado y los getters.) Una **entidad** es una clase mapeada a una tabla. **`@GeneratedValue(strategy = IDENTITY)`** deja que Postgres genere la clave; **`@OneToMany(mappedBy = "pedido")`** dice que las líneas apuntan a su pedido desde la otra tabla, y **`cascade = ALL, orphanRemoval =`**

true que se guardan y se borran con él. JPA exige un constructor sin argumentos; **protected** lo deja para Hibernate. Del otro lado, la línea:

LineaPedido.java

```
@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "pedido_id")
private Pedido pedido;
```

> El error típico: dejar **@ManyToOne** sin **fetch = LAZY**. En JPA es *eager* por defecto: cada línea carga también su pedido, lo necesites o no. La guía de Hibernate 7 lo llama una característica desafortunada de JPA y recomienda que casi todas las asociaciones sean lazy (An Introduction to Hibernate 7).

> El error típico: **@Data** de Lombok en una entidad. Genera **equals**, **hashCode** y **toString** con todos los campos, incluida la colección lazy: un **toString** en un log dispara una consulta, y un **hashCode** que cambia cuando Hibernate asigna el **id** rompe los **Set**. En entidades, getters y nada más.

El repositorio

RepositorioPedidos.java

```
public interface RepositorioPedidos extends JpaRepository<Pedido, Long> {

    List<Pedido> findByEstadoOrderByCreadoEnDesc(EstadoPedido estado);

    long countByCorreo(String correo);

    @Query("select new com.ejemplo.pedidos.pedidos.ResumenPedido(p.id, p.cliente,
p.estado, p.creadoEn) from Pedido p")
    Page<ResumenPedido> resumenes(Pageable pagina);
```

JpaRepository<Pedido, Long> ya trae **save**, **findById**, **findAll**, paginación... Las consultas se agregan de dos formas:

- Derivadas del nombre. **findByEstadoOrderByCreadoEnDesc** no tiene cuerpo: Spring Data arma la consulta leyendo el nombre (Spring Data JPA Reference, 4.1). Si el nombre ya no cabe en una línea, pasa a la otra forma.
- Con **@Query**, en JPQL, el SQL de JPA que habla de entidades y campos.

resumenes es una **proyección**: construye un **ResumenPedido** con cuatro columnas, sin cargar entidades ni líneas, que es todo lo que necesita un listado. Recibe un **Pageable** y devuelve un **Page**: la página más el

total, para el que Spring Data lanza la consulta de conteo. El controller lo convierte en un record propio, **`Pagina`**, para que el contrato no dependa de una clase de Spring Data.

Auditoría y fechas en UTC

Con **`@CreatedDate`** en **`creadoEn`**, **`@EntityListeners(AuditingEntityListener.class)`** en la entidad y esta configuración, Spring Data llena la fecha al guardar (Spring Data JPA Reference, 4.1):

ConfiguracionJpa.java

```
@Configuration
@EnableJpaAuditing
public class ConfiguracionJpa {}
```

La fecha es un **`Instant`**, un instante sin zona horaria, y la columna **`timestamp with time zone`**, que Postgres guarda siempre en UTC (PostgreSQL 16 Documentation). No depende de la zona del servidor y en el JSON sale con su **`Z`**; la zona se aplica al mostrarla.

Dirty checking: guardar sin `save()`

PedidoService.java

```
@Transactional
public void pagar(long id) {
    var pedido = repositorio.findById(id).orElseThrow(() -> new
PedidoNoEncontradoException(id));
    pedido.pagar();
}
```

No hay **`save()`**. Dentro de una transacción, la entidad cargada está **gestionada**: Hibernate recuerda cómo estaba y, al confirmar, escribe lo que cambió. Es el *dirty checking* (An Introduction to Hibernate 7), y lo comprobé con un test que paga un pedido y lo relee en otra transacción. Sin **`@Transactional`**, o saltándose el proxy por la autoinvocación, no hay transacción que confirme y el cambio se pierde sin error.

El problema N+1

Es el error de rendimiento más común con JPA. Queremos sumar el total de todos los pedidos:

NMasUnoTest.java

```
BigDecimal total = BigDecimal.ZERO;
for (Pedido pedido : cargar.get()) {
    for (LineaPedido linea : pedido.getLineas()) {
        total = total.add(linea.subtotal());
    }
}
```

Parece una consulta. Con **`findAll()`** son muchas más:

Recorrer 3 pedidos con sus líneas: N+1 consultas



findAll() manda una consulta: select de la tabla pedidos. Trae los 3 pedidos, pero sus líneas son lazy: en vez de la lista real, cada pedido tiene una colección sin inicializar que se carga cuando alguien la toca.

El bucle llega al primer pedido y llama a getLineas(). La colección no está cargada, así que Hibernate manda otra consulta: las líneas del pedido 1.

Segundo pedido, otra consulta. Cada una es un viaje de ida y vuelta a la base, con su latencia de red.

Tercer pedido, tercera consulta extra. Total: 1 consulta para los pedidos + 1 por cada pedido = N+1. Con 3 pedidos son 4; con 100 son 101. El test lo mide.

El arreglo: pedir los pedidos y sus líneas en la misma consulta. Con join fetch (o @EntityGraph) Hibernate manda un solo select con un join a lineas_pedido.

Los 3 pedidos llegan con sus líneas ya cargadas. El bucle es el mismo, pero getLineas() ya no va a la base. Una consulta, sean 3 pedidos o 100.

Para medirlo, Hibernate lleva estadísticas, entre ellas cuántas sentencias preparó:

ContadorDeConsultas.java

```
final class ContadorDeConsultas {

    private final Statistics estadisticas;

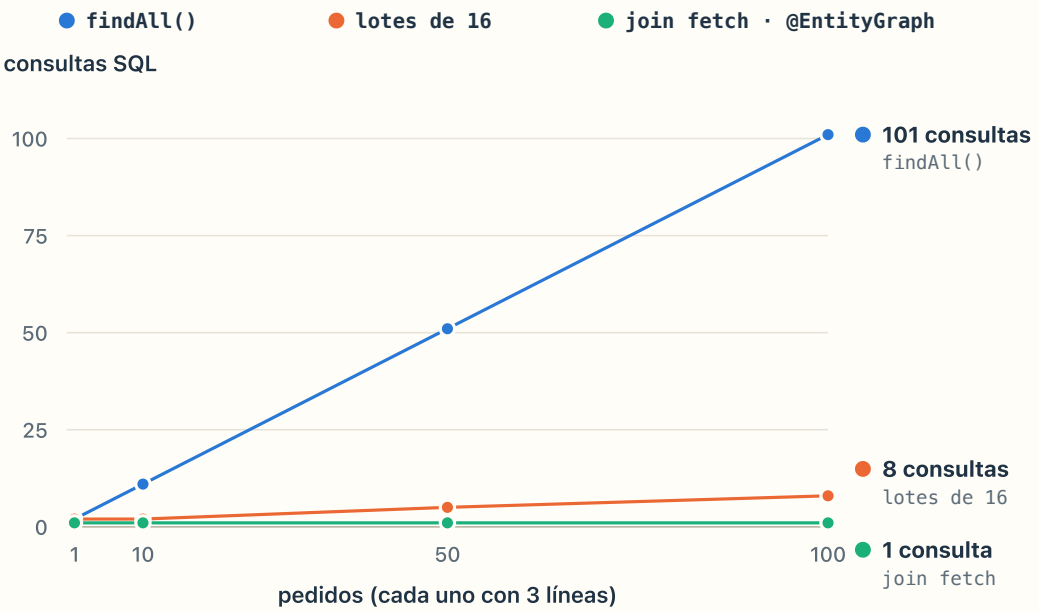
    ContadorDeConsultas(EntityManagerFactory emf) {
        this.estadisticas = emf.unwrap(SessionFactory.class).getStatistics();
        this.estadisticas.setStatisticsEnabled(true);
```

```
    }

    void reiniciar() {
        estadisticas.clear();
    }

    long consultas() {
        return estadisticas.getPrepareStatementCount();
    }
}
```

Los números que medí contra Postgres 16, con pedidos de tres líneas:



Medido con las estadísticas de Hibernate en NMasUnoTest y CargaPorLotesTest (Postgres 16).

N+1 crece con los datos; join fetch se queda en una consulta, y la carga por lotes queda en el medio.

Pedidos	`findAll()`	lotes de 16	`join fetch` / `@EntityGraph`
1	2	2	1
10	11	2	1
50	51	5	1
100	101	8	1

Con 10 pedidos en tu máquina, 11 consultas no se notan. Con miles en producción y la base en otra zona de red, la misma pantalla tarda segundos. El N+1 no se ve en el código: se ve midiendo.

RepositorioPedidos.java

```java
@Query("select p from Pedido p left join fetch p.lineas")
List<Pedido> todosConLineas();

@EntityGraph(attributePaths = "lineas")
List<Pedido> findByEstado(EstadoPedido estado);
```

- **`join fetch`**: trae los pedidos y sus líneas en la misma consulta. Desde Hibernate 6 no hace falta **`distinct`**: Hibernate ya no repite el pedido por cada línea del join (Hibernate ORM 6.0 Migration Guide), y el test lo comprueba.
- **`@EntityGraph(attributePaths = "lineas")`**: lo mismo sobre un método derivado (Spring Data JPA Reference, 4.1).

> – Para profundizar
>
> la tercera línea del gráfico es la **carga por lotes**: con **`spring.jpa.properties.hibernate.default_batch_fetch_size=16`**, Hibernate carga las líneas de hasta 16 pedidos por consulta, 1 + ⌈N/16⌉. Acota las consultas sin tocar el código y sirve cuando no sabes qué relaciones vas a recorrer. Ojo: un **`join fetch`** de una colección con paginación no puede paginar en SQL, porque las filas son líneas, no pedidos; para listados paginados, una proyección.

`open-in-view`: apágalo

En una app web, Boot registra por defecto *Open EntityManager in View*, que mantiene abierta la sesión de JPA durante **todo** el request, incluida la escritura del JSON (Spring Boot Reference, 4.1). **`spring.jpa.open-in-view`** vale **`true`** por defecto y Boot avisa al arrancar. Mi salida sin la propiedad:

```
WARN … JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by
default. Therefore, database queries may be performed during view rendering. Explicitly
configure spring.jpa.open-in-view to disable this warning
```

Con la sesión abierta, una relación lazy que Jackson toca al serializar dispara consultas desde la capa web, fuera de tu transacción, con la conexión tomada mientras se escribe la respuesta: un N+1 que nadie escribió. La app la apaga (**`open-in-view: false`**) y carga lo que necesita en el servicio.

> Nota: En este sitio, la configuración del backend también tiene **`open-in-view: false`** y **`ddl-auto: validate`**, que es el tema de la próxima sección.

> Lo que debes llevarte:

- Asociaciones lazy, **`@ManyToOne`** incluido, y nada de **`@Data`** en entidades.
- Dentro de la transacción, cambiar la entidad es guardarla.
- N+1 se mide y se arregla con **`join fetch`** o **`@EntityGraph`**.
- **`open-in-view: false`**.

Migraciones con Flyway

Hibernate puede crear las tablas desde las entidades (**`ddl-auto=update`** o **`create`**). Para un prototipo sirve; con datos de verdad no: no sabe renombrar una columna sin perder lo que tenía, no deja registro de qué cambió y cada instancia intentaría cambiar el esquema al arrancar.

Flyway usa **migraciones**: scripts SQL numerados que se aplican en orden, una vez cada uno. Con **`spring-boot-starter-flyway`** y el módulo de la base (**`flyway-database-postgresql`**), Boot los busca en **`src/main/resources/db/migration`** y los aplica al arrancar (Spring Boot How-to, 4.1). El nombre es **`V<versión>__<descripción>.sql`**, con dos guiones bajos:

V1__pedidos.sql

```
create table pedidos (
    id         bigint generated by default as identity primary key,
    cliente    varchar(120) not null,
    correo     varchar(254) not null,
    estado     varchar(20)  not null,
    creado_en  timestamp with time zone not null
);

create table lineas_pedido (
    id              bigint generated by default as identity primary key,
    pedido_id       bigint        not null references pedidos (id),
    producto        varchar(80)   not null,
    cantidad        integer       not null check (cantidad > 0),
    precio_unitario numeric(12, 2) not null
);

create index lineas_pedido_pedido_id_idx on lineas_pedido (pedido_id);
```

Flyway anota cada migración aplicada en su tabla **`flyway_schema_history`**. Mi log del primer arranque:

```
INFO … DbMigrate : Migrating schema "public" to version "1 - pedidos"
INFO … DbMigrate : Successfully applied 1 migration to schema "public", now at version
v1
```

El siguiente cambio es un **`V2__…sql`** nuevo; la Parte 3 agrega así la columna de versión.

`ddl-auto=validate`: Hibernate controla, no toca

Con Flyway a cargo del esquema, **`spring.jpa.hibernate.ddl-auto=validate`** hace que Hibernate compruebe que entidades y tablas coinciden, y si no, la app no arranca. Lo comprobé con una entidad que tiene un campo **`descuento`** que la tabla no tiene:

```
Schema validation: missing column [descuento] in table [pedidos]
```

Con Flyway y una base no embebida, el valor por defecto de **`ddl-auto`** es **`none`**, que no comprueba nada (Spring Boot How-to, 4.1): **`validate`** hay que ponerlo a mano.

Una migración aplicada no se edita

> **El error típico:** «la V1 tenía un error, la corrijo». Flyway guarda una suma de verificación (*checksum*) de cada migración aplicada. Si cambias el archivo, la suma no coincide y Flyway se niega a seguir, porque esa base ya ejecutó la versión vieja (Flyway documentation). El test aplica una V1, la «edita» y vuelve a migrar:

MigracionesTest.java

```
@Test
void editarUnaMigracionYaAplicadaImpideMigrar() {
    flyway("original").migrate();

    assertThatThrownBy(() -> flyway("editada").migrate())
            .isInstanceOf(FlywayValidateException.class)
            .hasMessageContaining("Migration checksum mismatch for migration version
1");
}
```

> *“La salida real:*
>
> ```
> Validate failed: Migrations have failed validation
> Migration checksum mismatch for migration version 1
> -> Applied to database : -1704010591
> -> Resolved locally    : -1961642280
> Either revert the changes to the migration, or run repair to update the
> schema history.
> ```
>
> *Se deshace el cambio y se escribe una V2 que corrija. **`repair`** reescribe el historial: solo si sabes que la base ya está como dice el archivo nuevo.”*

> **Nota:** En este sitio, el backend lleva veinte migraciones, desde la que creó la tabla de artículos hasta la que agregó portadas y series, y ninguna se editó después de aplicarse: cada cambio llegó como una versión nueva.

Llamar a otros servicios

La app consulta un servicio externo de tarifas de envío. Spring tiene tres clientes HTTP, y el clásico **`RestTemplate`** está obsoleto en favor de **`RestClient`** según la documentación de Spring Framework 7.0, aunque la clase todavía no lleva **`@Deprecated`** (Spring Framework Reference, 7.0).

`RestClient`

`RestClient` es el cliente síncrono con API fluida: describes el request en cadena y recibes el cuerpo convertido. Boot configura un **`RestClient.Builder`** con los convertidores y el cliente HTTP, y recomienda inyectarlo (Spring Boot Reference, 4.1). Lo trae **`spring-boot-starter-restclient`**:

ConsultaTarifas.java

```
@Component
public class ConsultaTarifas {

    private final RestClient http;

    public ConsultaTarifas(RestClient.Builder builder, @Value("${tarifas.base-url}")
String baseUrl) {
        this.http = builder.baseUrl(baseUrl).build();
    }

    public Tarifa tarifa(String pais) {
        return http.get()
                .uri("/tarifas/{pais}", pais)
                .retrieve()
                .body(Tarifa.class);
    }
}
```

Interfaces con `@HttpExchange`

Para un servicio que llamas desde varios lugares, escribes una **interfaz** y Spring genera el cliente, como con los repositorios (Spring Framework Reference, 7.0):

ClienteTarifas.java

```
@HttpExchange("/tarifas")
```

```java
public interface ClienteTarifas {

    @GetExchange("/{pais}")
    Tarifa tarifa(@PathVariable String pais);
}
```

En Boot 4 se registra con **`@ImportHttpServices`** en un grupo cuya URL base y timeouts salen de **`spring.http.serviceclient.<grupo>.*`** (Spring Boot Reference, 4.1); el del grupo **`tarifas`** está en el **`application.yml`** de más arriba:

ConfiguracionTarifas.java

```java
@Configuration
@ImportHttpServices(group = "tarifas", types = ClienteTarifas.class)
public class ConfiguracionTarifas {}
```

`WebClient` es el cliente reactivo de Spring WebFlux. Boot lo recomienda si tu app ya usa WebFlux; si no, **`RestClient`** (Spring Boot Reference, 4.1).

Timeouts, siempre

> El error típico: llamar a otro servicio sin timeout. En Boot 4.1, **`spring.http.clients.connect-timeout`** y **`read-timeout`** no tienen valor por defecto (metadata de configuración de Spring Boot 4.1.2), y con el cliente HTTP del JDK, el que usa la app, no fijar uno equivale a «bloquear para siempre» (Java SE 21 API). Si el otro servicio se cuelga, cada request tuyo queda esperando con un hilo tomado, y cuando se acaban los hilos tu API también se cuelga. Un timeout convierte un servicio lento en un error rápido.

La app fija timeouts globales (**`spring.http.clients.*`**) y uno para el grupo **`tarifas`**. El test levanta un servidor de tarifas falso en **`localhost`** que tarda tres segundos, baja los timeouts a medio segundo y comprueba que los dos clientes cortan:

ClientesHttpTest.java

```java
@DynamicPropertySource
static void tarifas(DynamicPropertyRegistry registro) {
    registro.add("tarifas.base-url", SERVIDOR::url);
    registro.add("spring.http.clients.read-timeout", () -> "500ms");
    registro.add("spring.http.serviceclient.tarifas.read-timeout", () -> "500ms");
}

@Autowired
ClienteTarifas cliente;

@Autowired
```

```
ConsultaTarifas consulta;

@Test
void laInterfazHttpExchangeLlamaAlServicio() {
    assertThat(cliente.tarifa("CR")).isEqualTo(new Tarifa("CR", new BigDecimal("12.50"),
3));
}

@Test
void unServicioLentoCortaPorTimeoutEnVezDeColgarElHilo() {
    assertThatThrownBy(() ->
cliente.tarifa("LENTO")).isInstanceOf(ResourceAccessException.class);
    assertThatThrownBy(() ->
consulta.tarifa("LENTO")).isInstanceOf(ResourceAccessException.class);
}
```

Probar sin salir a internet

Un test que llama al servicio real es lento y frágil. Hay dos alternativas: un servidor HTTP de verdad en **`localhost`** (el del JDK, **`com.sun.net.httpserver.HttpServer`**), como arriba, para lo que pasa en la red, y **`MockRestServiceServer`**, que intercepta el **`RestClient`** sin red y verifica qué se le pidió:

ConsultaTarifasTest.java

```
@Test
void leeLaTarifaDelServicio() {
    var builder = RestClient.builder();
    var servidor = MockRestServiceServer.bindTo(builder).build();
    servidor.expect(requestTo("https://tarifas.ejemplo.com/tarifas/CR"))
            .andRespond(withSuccess("""
                    {"pais":"CR","costo":12.50,"diasHabiles":3}
                    """, MediaType.APPLICATION_JSON));

    var tarifa = new ConsultaTarifas(builder,
"https://tarifas.ejemplo.com").tarifa("CR");

    assertThat(tarifa.diasHabiles()).isEqualTo(3);
    servidor.verify();
}
```

> Nota: En este sitio, el backend verifica el captcha de Cloudflare Turnstile con **`RestClient`** y lo prueba con **`MockRestServiceServer`**.

Logging

Boot trae **Logback**, y tu código escribe contra **SLF4J**, la interfaz: **`LoggerFactory.getLogger(…)`** y **`log.info(…)`**. Por defecto se registran **`ERROR`**, **`WARN`** e **`INFO`** (Spring Boot Reference, 4.1):

NotificadorLog.java

```
@Component
class NotificadorLog implements Notificador {

    private static final Logger log = LoggerFactory.getLogger(NotificadorLog.class);

    @Override
    public void pedidoCreado(Pedido pedido) {
        log.info("Aviso enviado: pedido {} con {} líneas", pedido.getId(),
pedido.getLineas().size());
        log.debug("Estado del pedido {}: {}", pedido.getId(), pedido.getEstado());
    }
}
```

- **Mensajes con `{}`**, no concatenados: SLF4J arma el texto solo si el nivel está activo.
- **El nivel dice qué tan grave es:** **`INFO`** para lo que pasó en el negocio, **`DEBUG`** para el detalle al investigar, **`WARN`** para lo raro que no rompe, **`ERROR`** para lo que alguien tiene que mirar.

El nivel se ajusta **por paquete** con **`logging.level.<paquete>`**. En el perfil **`dev`**, **`com.ejemplo.pedidos: debug`** muestra el detalle de tu código y **`org.hibernate.SQL: debug`** cada SQL (útil para ver un N+1), sin el **`DEBUG`** de todas las bibliotecas. Boot trae además los grupos **`web`** y **`sql`** (Spring Boot Reference, 4.1).

Lo que nunca se registra

Un log se copia, se envía a otros sistemas y lo lee gente que no debería ver datos de tus clientes. No van al log:

- **Contraseñas, tokens, claves de API ni encabezados `Authorization`.**
- **Datos personales**: correos, teléfonos, direcciones, documentos. Se registra el id del pedido, no el correo de quien lo hizo.
- **Cuerpos completos de requests y respuestas**, que suelen tener todo lo anterior.

El test comprueba que el correo no aparece y que el **`DEBUG`** no sale con la configuración por defecto (el id es **`null`** porque el pedido nunca se guardó):

NotificadorLogTest.java

```
@Test
void elLogDiceQuePedidoSeAvisoPeroNoAQuien(CapturedOutput salida) {
    var pedido = new Pedido("Ana", "ana@example.com");
    pedido.agregarLinea("taza", 1, new BigDecimal("8.50"));
```

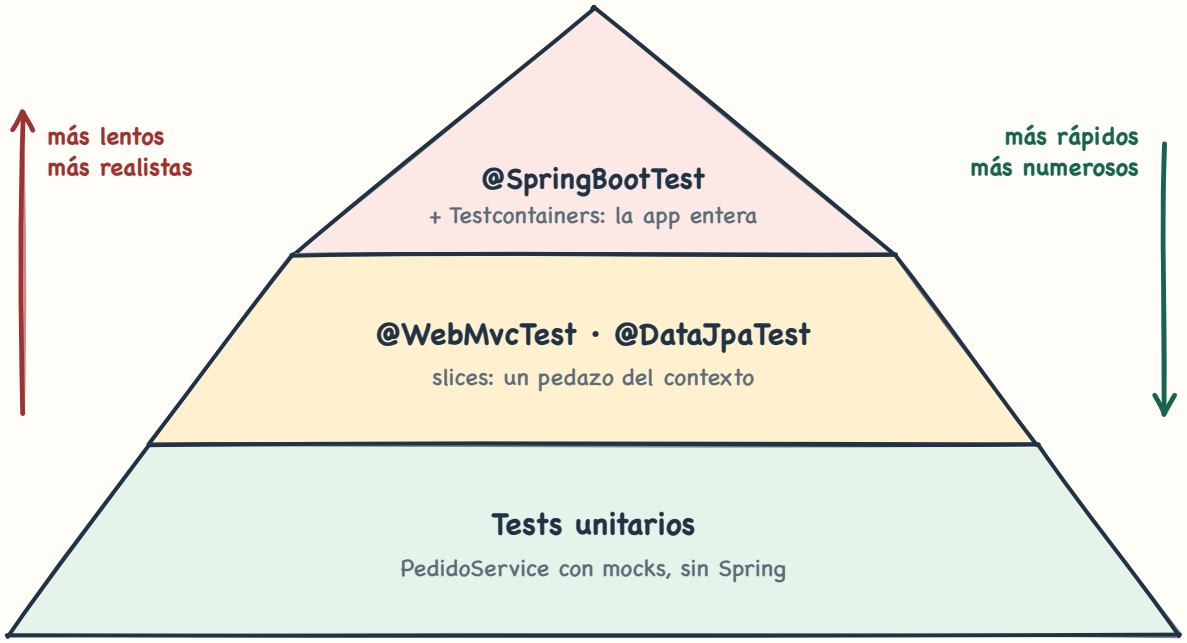
```
    new NotificadorLog().pedidoCreado(pedido);

    assertThat(salida).contains("Aviso enviado: pedido null con 1 líneas");
    assertThat(salida).doesNotContain("ana@example.com");
    assertThat(salida).doesNotContain("Estado del pedido");
}
```

> **Nota:** En este sitio, la configuración de Logback escribe texto legible con el perfil **`local`** y JSON estructurado en staging y producción, y un filtro pone un **`requestId`** en el MDC (el contexto del hilo de logging) para correlacionar las líneas de un mismo request. Los logs estructurados son tema de la Parte 3.

Tests

Boot da herramientas para probar en tres niveles, y las ordena la **pirámide de tests** de Mike Cohn: tests de distinta granularidad, y cuanto más alto el nivel, menos tests (Ham Vocke, *The Practical Test Pyramid*).



Muchos abajo, pocos arriba: cada piso prueba lo que el de abajo no puede.

Cada piso prueba lo que el de abajo no puede, y por eso hacen falta los tres.

Abajo: tests unitarios, sin Spring

Gracias a la inyección por constructor, **`PedidoService`** se prueba con **`new`** y dobles de Mockito, en milisegundos:

PedidoServiceTest.java

```
RepositorioPedidos repositorio = mock(RepositorioPedidos.class);
Notificador notificador = mock(Notificador.class);
PedidoService servicio = new PedidoService(
        repositorio, new Catalogo(), notificador, new PedidosProperties(2, "EUR",
Duration.ofMinutes(30)));

@Test
void elPrecioLoPoneElCatalogoNoElCliente() {
    given(repositorio.save(any())).willAnswer(invocacion -> invocacion.getArgument(0));

    var creado = servicio.crear(new NuevoPedido("Ana", "ana@example.com", List.of(new
NuevoPedido.Linea("taza", 2))));

    assertThat(creado.total()).isEqualByComparingTo(new BigDecimal("17.00"));
    verify(notificador).pedidoCreado(any());
}

@Test
void masLineasQueElLimiteConfiguradoNoSeGuarda() {
    var lineas = List.of(
            new NuevoPedido.Linea("taza", 1), new NuevoPedido.Linea("libreta", 1), new
NuevoPedido.Linea("camiseta", 1));

    assertThatThrownBy(() -> servicio.crear(new NuevoPedido("Ana", "ana@example.com",
lineas)))
            .isInstanceOf(LimiteDeLineasException.class);
    verify(repositorio, never()).save(any());
}
```

Aquí van las reglas del negocio. Si para probar una regla necesitas levantar Spring, la regla probablemente está en el lugar equivocado.

En el medio: *slices*

Un **slice** arranca solo un pedazo del contexto. **`@WebMvcTest`** carga la capa web (controllers, **`@ControllerAdvice`**, Jackson, validación), sin servicios ni base, y las dependencias del controller se reemplazan con **`@MockitoBean`** (Spring Boot Reference, 4.1). Así se prueban rutas, conversión, validación, códigos y errores, como en los tests de las secciones anteriores:

PedidoControllerTest.java

```
@WebMvcTest(PedidoController.class)
@Import(Catalogo.class)
class PedidoControllerTest {

    @Autowired
    MockMvc mvc;
```

```
    @MockitoBean
    PedidoService servicio;
```

`@DataJpaTest` es el slice de datos: entidades, repositorios y la base. Cada test corre en una transacción que se revierte al final (Spring Boot Reference, 4.1). **`replace = NONE`** evita que cambie la base por una en memoria: la queremos real. Con **`@ServiceConnection`** el valor por defecto (**`NON_TEST`**) ya la respeta; **`NONE`** lo deja explícito (Spring Boot API, 4.1).

RepositorioPedidosTest.java

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@Import({ContenedoresDePrueba.class, ConfiguracionJpa.class})
class RepositorioPedidosTest {

    @Autowired
    RepositorioPedidos repositorio;

    @Autowired
    TestEntityManager em;
```

– Para profundizar

en Boot 4 cambiaron los paquetes: **`@WebMvcTest`** está en **`org.springframework.boot.webmvc.test.autoconfigure`**, **`@DataJpaTest`** en **`org.springframework.boot.data.jpa.test.autoconfigure`** y **`@AutoConfigureTestDatabase`** en **`org.springframework.boot.jdbc.test.autoconfigure`**. El **`@MockBean`** de Boot ya no existe: se usa **`@MockitoBean`**, de Spring Framework (Spring Boot 4.0 Migration Guide).

Arriba: la app entera contra un Postgres real

`@SpringBootTest` arranca la aplicación completa; con **`@AutoConfigureMockMvc`** le mandas requests sin abrir un puerto:

AplicacionCompletaTest.java

```
@SpringBootTest(properties =
"management.endpoints.web.exposure.include=health,conditions")
@AutoConfigureMockMvc
@Import(ContenedoresDePrueba.class)
class AplicacionCompletaTest {

    @Autowired
```

```
    MockMvc mvc;

    @Test
    void crearYLeerUnPedidoDeVerdadContraPostgres() throws Exception {
        var creado = mvc.perform(post("/api/pedidos")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("""
                                {"cliente":"Ana","correo":"ana@example.com",
                                 "lineas":[{"producto":"taza","cantidad":2},
{"producto":"libreta","cantidad":1}]}
                                """))
                .andExpect(status().isCreated())
                .andReturn();
        String ubicacion = creado.getResponse().getHeader("Location");

        mvc.perform(get(ubicacion))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.estado").value("CREADO"))
                .andExpect(jsonPath("$.total").value(22.00));
    }
```

¿Contra qué base? Una en memoria como H2 tiene otro dialecto y otros tipos: que un test pase contra H2 dice poco de Postgres. **Testcontainers** levanta un Postgres de verdad en un contenedor Docker, y **`@ServiceConnection`** hace que Boot se conecte a él sin que escribas una URL (Spring Boot Reference, 4.1):

ContenedoresDePrueba.java

```
@TestConfiguration(proxyBeanMethods = false)
public class ContenedoresDePrueba {

    @Bean
    @ServiceConnection
    PostgreSQLContainer postgres() {
        return new PostgreSQLContainer(DockerImageName.parse("postgres:16-alpine"));
    }
}
```

Es una **`@TestConfiguration`**, una configuración solo para tests que se suma a la de la app y que cada test importa con **`@Import`**. El contenedor arranca una vez por contexto de Spring. Necesitas Docker; en mi máquina, los 61 tests de esta guía tardan unos 17 segundos.

`@Transactional` en un test puede esconder un bug

> El error típico: anotar los tests de integración con **`@Transactional`** «para que limpien solos». El test corre en una transacción que se revierte al final, y eso cambia el comportamiento de tu código: puede pasar y el mismo código fallar en producción (Spring Framework Reference, 7.0).

Esta clase tiene un bug: recorre las líneas de un pedido en un método **sin `@Transactional`**. **`findById`** abre y cierra su transacción, y cuando **`getLineas()`** intenta cargar ya no hay sesión:

ContadorDeLineas.java

```
class ContadorDeLineas {

    private final RepositorioPedidos repositorio;

    ContadorDeLineas(RepositorioPedidos repositorio) {
        this.repositorio = repositorio;
    }

    int lineasDe(long id) {
        return repositorio.findById(id).orElseThrow().getLineas().size();
    }
}
```

TransactionalEnTestsTest.java

```
@Nested
@Transactional
class ConTransactionalEnElTest {
    @Test
    void pasaPorqueLaSesionDelTestSigueAbierta() {
        assertThat(contador.lineasDe(pedidoConDosLineas())).isEqualTo(2);
    }
}

@Nested
class SinTransactionalComoEnProduccion {
    @Test
    void fallaConLazyInitializationException() {
        long id = pedidoConDosLineas();
        assertThatThrownBy(() ->
contador.lineasDe(id)).isInstanceOf(LazyInitializationException.class);
    }
}
```

Con **`@Transactional`**, la sesión del test sigue abierta y el test **pasa**. Sin él, como en producción, **`LazyInitializationException`**. Una suite entera **`@Transactional`** lleva ese bug a producción con todo en verde. Para no depender de datos ajenos, que cada test cree los suyos y mire solo esos, o borrar al empezar. Esto no va contra **`@DataJpaTest`**, transaccional por defecto: ahí pruebas consultas, no dónde empieza y termina una transacción.

La caché de contextos, o por qué tu suite se vuelve lenta

Arrancar un contexto completo cuesta segundos, así que Spring los guarda en una **caché** estática y reutiliza uno si otra clase pide **exactamente la misma configuración** (Spring Framework Reference, 7.0). Las clases, los perfiles, las propiedades y también cada **`@MockitoBean`** y cada **`@DynamicPropertySource`** forman parte de la clave: cambia uno y es otro contexto. Este test ejecuta cuatro clases y cuenta los contextos:

CacheDeContextosTest.java

```
@SpringBootTest(classes = Minima.class)
static class PruebaA {
    @Test
    void usaElContexto(@Autowired Reloj reloj) {}
}

@SpringBootTest(classes = Minima.class)
static class PruebaB {
    @Test
    void usaElMismoContexto(@Autowired Reloj reloj) {}
}

@SpringBootTest(classes = Minima.class)
static class PruebaConMock {
    @MockitoBean
    Reloj reloj;

    @Test
    void otroContexto() {}
}

@SpringBootTest(classes = Minima.class, properties = "pedidos.moneda=USD")
static class PruebaConOtraPropiedad {
    @Test
    void yOtroMas() {}
}

@Test
void cadaConfiguracionDistintaEsUnContextoNuevo() {
    contextosCreados.set(0);

    EngineTestKit.engine("junit-jupiter")
            .selectors(
                    selectClass(PruebaA.class),
                    selectClass(PruebaB.class),
                    selectClass(PruebaConMock.class),
                    selectClass(PruebaConOtraPropiedad.class))
            .execute()
            .testEvents()
            .assertStatistics(estadisticas -> estadisticas.succeeded(4).failed(0));
```

```
    assertThat(contextosCreados).hasValue(3);
}
```

Cuatro clases, **tres** contextos: **`PruebaA`** y **`PruebaB`** comparten uno, y el **`@MockitoBean`** y la propiedad distinta crean uno cada uno. Con cien **`@SpringBootTest`** que mockean cada uno algo distinto, son cien contextos con su pool y su contenedor: así una suite pasa de un minuto a quince sin un solo test lento.

> **Lo que debes llevarte:**
>
> - **Pirámide:** muchos unitarios, slices para web y datos, pocos **`@SpringBootTest`**.
> - **Base real** con Testcontainers y **`@ServiceConnection`**.
> - **Sin `@Transactional`** en los tests de integración.
> - **Pocas configuraciones de test distintas.**

> – **Para profundizar**
>
> la caché guarda hasta 32 contextos con desalojo LRU (**`spring.test.context.cache.maxSize`**) y vive en una variable estática, dentro de una JVM. Si Surefire usa una JVM nueva por clase, no hay caché entre clases y el build es mucho más lento (Spring Framework Reference, 7.0). Para contar contextos, el logger **`org.springframework.test.context.cache`** en **`DEBUG`**.

> **Nota:** En este sitio, los tests de integración heredan de una clase base que levanta un **`postgres:16-alpine`** con **`@Container`** y **`@ServiceConnection`**. Y Surefire corre con **`reuseForks=false`**, una JVM por clase de test: así se eliminaron fallas intermitentes que no tenían explicación. A cambio, cada clase arma su propio contexto, justo lo que la caché busca evitar.

Lo que solo enseña la experiencia

Casi todo esto lo aprendí con un incidente:

- **El main fuera del paquete raíz** no da error: da 404.
- **Un bean propio apaga la autoconfiguración entera** de ese tipo. Para un detalle, una propiedad o un *customizer*.
- **`--debug`** y **`/actuator/conditions`** contestan «¿de dónde salió este bean?». No adivines.
- **Una variable de entorno olvidada pisa tu `application.yml`**, y **`Origin`** te dice cuál fue.

- **Entidades fuera de la API.** Records de entrada y salida.
- **N+1 no se ve en el código**, se mide.
- **`open-in-view: false`** desde el primer día. Apagarlo después, en una app grande, duele.
- **Una migración aplicada no se toca.**
- **Sin timeout, un servicio lento te cuelga a ti.**
- **La excepción se registra una vez**, en el borde, y al cliente le llega un **`ProblemDetail`** sin traza.
- **`@Transactional` en un test** es una red que en producción no existe.
- **Una suite lenta casi nunca tiene tests lentos**: tiene demasiados contextos.

En la Parte 3 llevamos esta app a producción: seguridad, concurrencia, observabilidad, despliegue y rendimiento de arranque.

¿Cuál de estos te encontró primero a ti? Y si tuvieras que apostar, ¿cuántos contextos de Spring crea hoy la suite de tu proyecto?

Referencias

- Spring Boot Reference, 4.1.2: Build Systems, Structuring Your Code, Using the @SpringBootApplication Annotation, Auto-configuration, Creating Your Own Auto-configuration, Developer Tools, Externalized Configuration, Profiles, Logging, JSON, Servlet Web Applications, SQL Databases, Calling REST Services, Testing Spring Boot Applications, Testcontainers, The Executable Jar Format, Spring MVC How-to, Database Initialization, Common Application Properties.
- Spring Boot 4.0 Migration Guide.
- Spring Framework Reference, 7.0.10: DispatcherServlet Processing, Special Bean Types, Error Responses, Java Bean Validation, REST Clients, Context Caching, Transaction Management in tests.
- Spring Data JPA Reference, 4.1.2: JPA Query Methods, Auditing.
- An Introduction to Hibernate 7; Hibernate ORM 6.0 Migration Guide.
- Jackson Release 3.0.
- RFC 9457: Problem Details for HTTP APIs (2023).
- Redgate Flyway: Validate.
- PostgreSQL 16: Date/Time Types.
- Java SE 21 API: **`HttpRequest.Builder`**.
- springdoc-openapi; Spring Initializr.
- Ham Vocke, The Practical Test Pyramid (2018).