Spring Boot en producción: nivel experto

Parte 3 de 3 · 4 de noviembre de 2026 · spring-boot · spring · java · produccion · serie-spring

> **Parte 3 de 3** de la serie «Spring, de novato a experto». Antes vienen la parte 1: Spring Framework, el contenedor que arma tu aplicación y la parte 2: Spring Boot, de la primera app a una API completa.

Una API que pasa sus tests no está lista para producción. Le falta saber quién la llama, resistir a dos clientes que editan el mismo pedido a la vez, no mandar correos de pagos que no ocurrieron, correr una tarea programada una sola vez aunque haya tres instancias, avisar cuándo está lista para recibir tráfico y apagarse sin cortar a nadie.

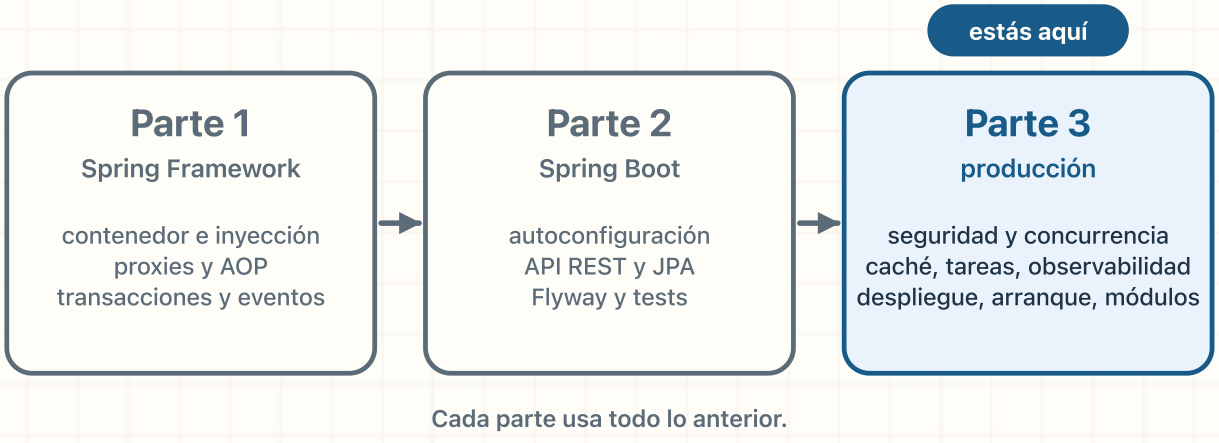
Casi todas esas fallas aparecen la primera semana en producción y tienen la misma raíz: algo que Spring hace por ti a través de un proxy, una transacción o una autoconfiguración, y que nadie miró de cerca. Seguimos con la tienda de pedidos de la Parte 2, y cada trampa tiene su antes y su después.

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

▼ EN ESTE ARTÍCULO

01. Qué se asume de las partes 1 y 2
02. Spring Security: quién eres y qué puedes hacer
03. Concurrencia: dos peticiones sobre el mismo pedido
04. Eventos después del commit
05. Caché
06. Trabajo en segundo plano
07. Observabilidad
08. Empaquetar y desplegar
09. Arranque y memoria
10. Tu propia autoconfiguración
11. Arquitectura: paquetes y módulos
12. Spring Boot 4 hoy y cómo migrar
13. Lo que solo enseña la experiencia
14. Referencias

Qué se asume de las partes 1 y 2



Esta parte usa todo lo anterior: casi cada trampa de producción es un proxy, una transacción o una autoconfiguración.

Vas a necesitar tres ideas de la Parte 1:

- **Lo que te entrega el contenedor puede ser un proxy**, y una llamada desde dentro de la misma clase no pasa por él (proxies y AOP). Seguridad por método, caché, **`@Async`** y transacciones funcionan así.
- **`@Transactional`**: qué es una transacción, la propagación **`REQUIRED`** y **`REQUIRES_NEW`**, y cuándo hay rollback (transacciones).
- **Los eventos** con **`ApplicationEventPublisher`** y **`@EventListener`** (eventos).

Y de la Parte 2:

- cómo decide Boot qué configurar y cómo reemplazar un bean suyo (autoconfiguración);
- los perfiles y **`@ConfigurationProperties`** (propiedades y perfiles);
- el controlador REST y los errores con **`ProblemDetail`** (errores);
- la entidad **`Pedido`** con Spring Data JPA y las migraciones de Flyway (datos, Flyway);
- los tests con Testcontainers y **`@ServiceConnection`** (tests).

Todo eso lo uso sin volver a explicarlo.

Spring Security: quién eres y qué puedes hacer

Hoy cualquiera que conozca la URL de la API puede crear pedidos, leer los de otros y cancelarlos. Cada petición necesita dos respuestas, y conviene no mezclarlas:

- **Autenticación**: ¿quién eres? Comprobar una identidad: una contraseña, un certificado, un token firmado.
- **Autorización**: ¿qué puedes hacer? Dada una identidad ya comprobada, decidir si puede crear, leer o cancelar.

Si falla la primera, la respuesta es **401 Unauthorized**, que pese al nombre significa "no sé quién eres". Si falla la segunda, **403 Forbidden**: "sé quién eres, y no puedes".

Lo que pasa al agregar el starter

Spring Security resuelve las dos cosas. Si agregas solo **`spring-boot-starter-security`** y no configuras nada, Boot protege todas las rutas, crea un usuario llamado **`user`** y escribe su contraseña en el log al arrancar, con nivel WARN (Spring Boot Reference, 4.1):

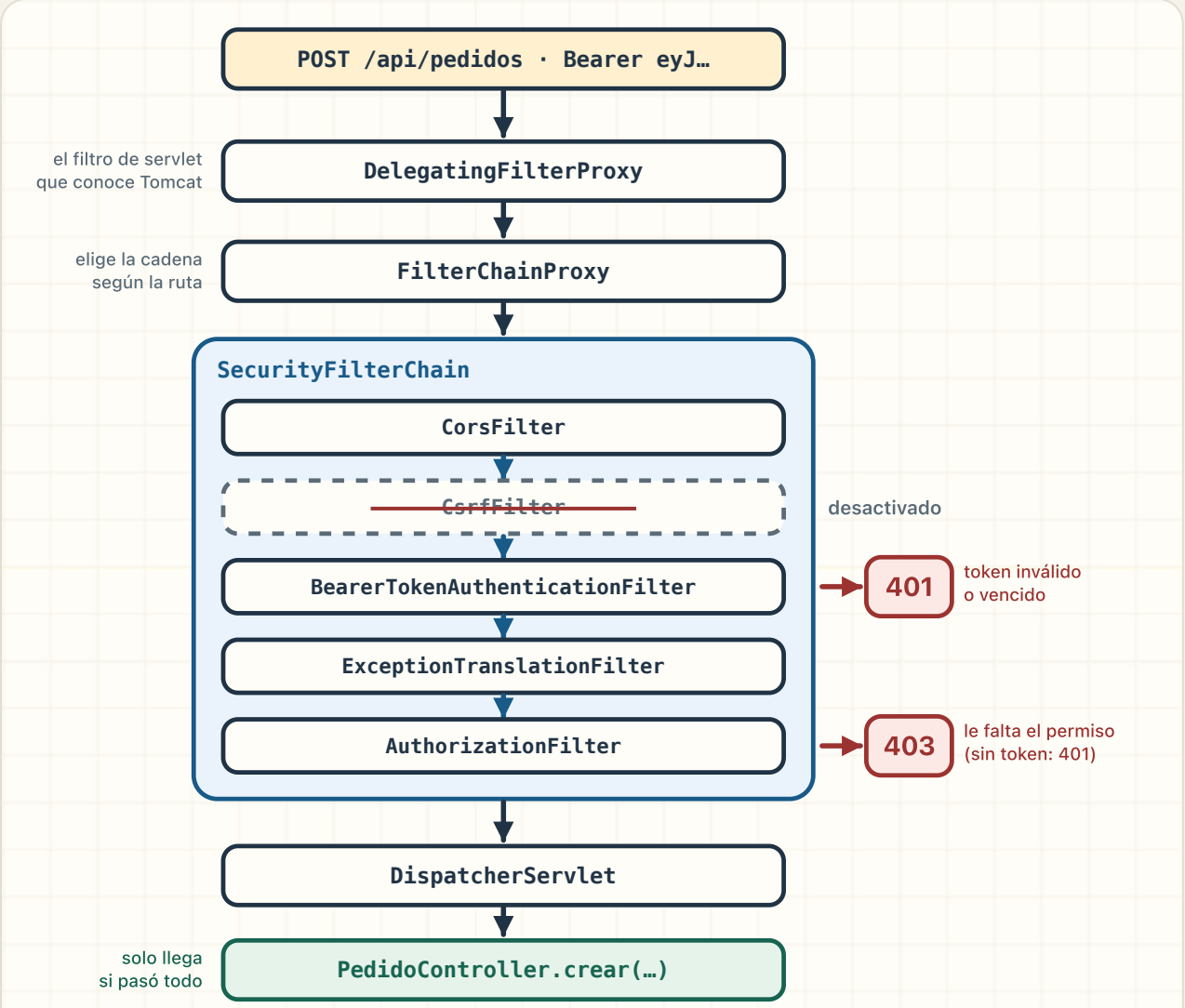
```
Using generated security password: 6c3a1f0e-…

This generated password is for development use only. Your security configuration must be
updated before running your application in production.
```

Es para que veas que la seguridad está activa, no para producción. Un detalle que comprobé con un test: si en el classpath también está el soporte de OAuth2 para servidores de recursos, que es lo que vamos a usar, Boot **no** crea ese usuario, porque las identidades van a llegar en un token.

La cadena de filtros

Spring Security no vive en tu controlador: vive antes, en la cadena de **filtros** de servlets. Un filtro es un objeto que recibe la petición antes que tu código y decide si la deja seguir, la modifica o la corta.



Tu controlador es lo último de la fila: si un filtro corta, el controlador nunca se entera.

El contenedor de servlets (Tomcat) no sabe nada de Spring. Spring Security se engancha con un filtro estándar, **`DelegatingFilterProxy`**, que le pasa la petición a un bean, **`FilterChainProxy`**. Ese bean elige cuál **`SecurityFilterChain`** corresponde a la ruta y la ejecuta: una lista ordenada de filtros de seguridad (Spring Security Reference, 7.1). El orden lo pone Spring Security; tú decides qué filtros hay y con qué reglas, en un bean:

ConfiguracionSeguridad.java

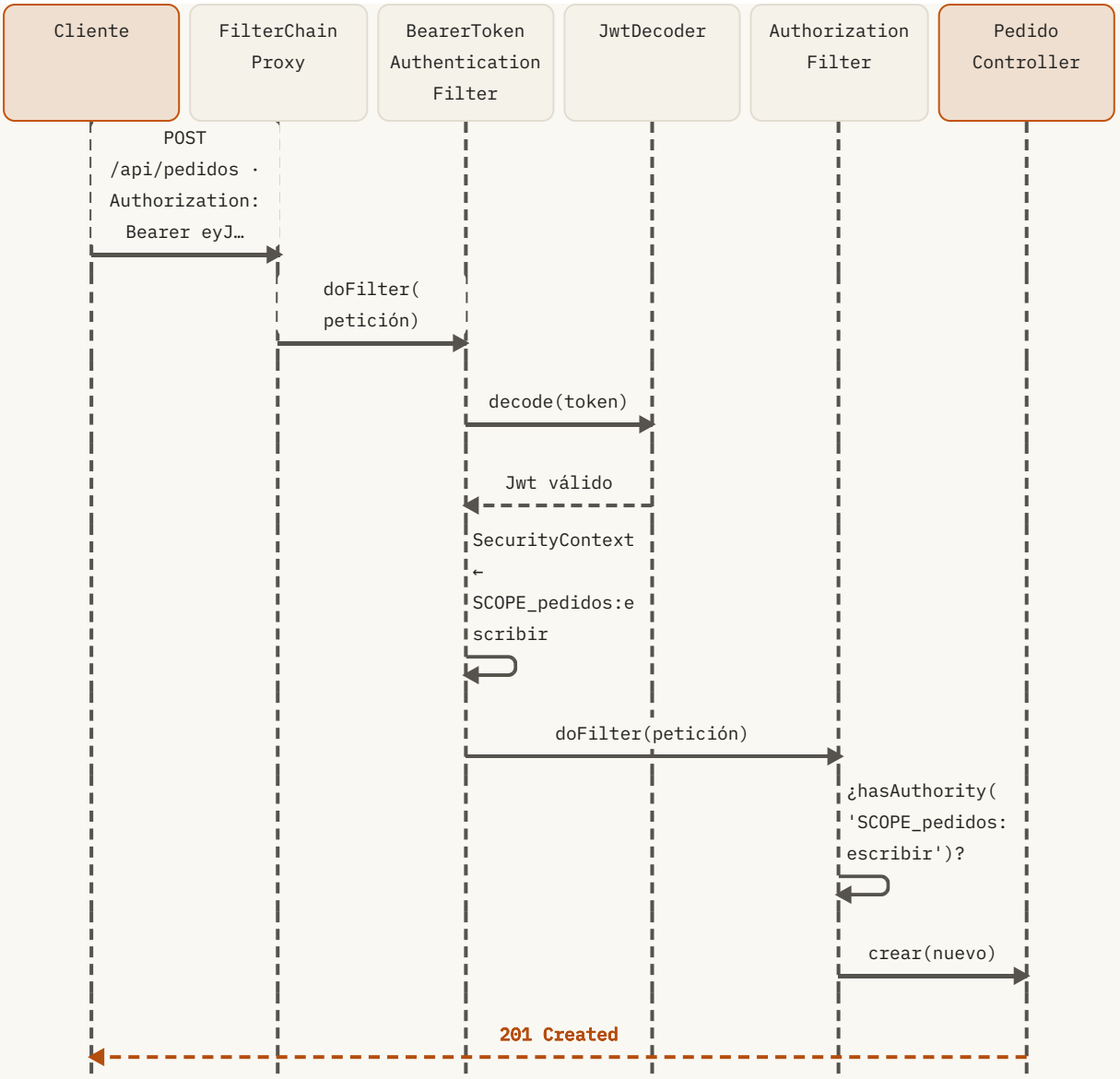
```
@Bean
SecurityFilterChain api(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(peticiones -> peticiones
                    .requestMatchers("/actuator/health/**", "/livez",
"/readyz").permitAll()
                    .requestMatchers("/actuator/**").hasAuthority("SCOPE_pedidos:admin")
                    .requestMatchers(HttpMethod.GET,
"/api/pedidos/**").hasAuthority("SCOPE_pedidos:leer")

.requestMatchers("/api/pedidos/**").hasAuthority("SCOPE_pedidos:escribir")
                    .anyRequest().denyAll())
            .oauth2ResourceServer(servidor -> servidor.jwt(withDefaults()))
            .sessionManagement(sesion ->
sesion.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(AbstractHttpConfigurer::disable)
            .cors(withDefaults());
    return http.build();
}
```

- **`authorizeHttpRequests`** son las reglas de autorización, **en orden: la primera que coincide decide**. Lo que no se nombró se niega: **`denyAll()`** al final convierte un olvido en un rechazo (401 sin token, 403 con token) en vez de una puerta abierta. Las sondas de salud son públicas porque el orquestador no manda token; **`/livez`** y **`/readyz`** son sus rutas en producción (lo verás al desplegar).
- **`oauth2ResourceServer(...jwt...)`** activa la autenticación por token JWT.
- **`STATELESS`**: la app no crea sesiones HTTP; cada petición trae su token.
- **`csrf`** y **`cors`** tienen su apartado más abajo.

Una petición para crear un pedido, filtro por filtro:

Un POST /api/pedidos atraviesa la cadena de filtros



Tomcat recibe la petición y la pasa por sus filtros. El de Spring Security es DelegatingFilterProxy, que se la entrega al bean FilterChainProxy. Este elige la SecurityFilterChain que corresponde a /api/pedidos.

Antes pasaron otros filtros de la cadena, como el de CORS y el que escribe cabeceras de seguridad. El de CSRF no está: lo desactivamos. Ahora le toca al filtro que busca un token Bearer en la cabecera Authorization.

Encontró el token y se lo pasa al JwtDecoder. El decodificador comprueba la firma con la clave pública del emisor y valida el vencimiento.

La firma es buena y el token no venció. Si algo hubiera fallado, el filtro respondería 401 con la cabecera WWW-Authenticate: Bearer, y la petición terminaría acá.

El JwtAuthenticationConverter, que usa el proveedor de autenticación al que el filtro le pasó el token, convierte cada alcance del claim scope en una autoridad con el prefijo SCOPE_. El filtro guarda la autenticación en el SecurityContext: desde acá, el resto de la petición sabe quién es el cliente.

Autenticado. Sigue al último filtro que importa: el de autorización.

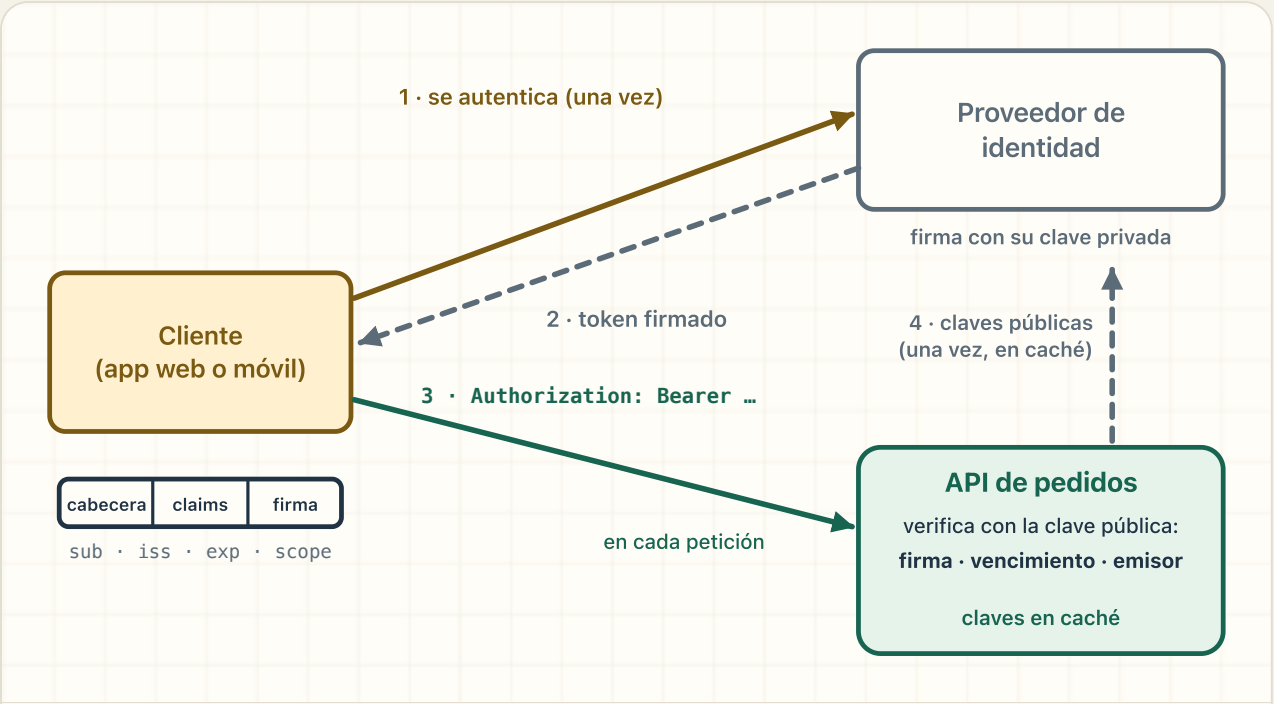
Recorre las reglas de authorizeHttpRequests en orden. La primera que coincide con POST /api/pedidos pide SCOPE_pedidos:escribir, y el token la trae. Si no la trajera, el filtro lanzaría AccessDeniedException y el ExceptionTranslationFilter, que la atrapa, respondería 403.

Recién ahora la petición llega al DispatcherServlet y a tu controlador. Simplifico: entre los dos está todo el recorrido de Spring MVC de la Parte 2.

El controlador nunca vio el token ni las reglas. Solo recibió una petición que ya alguien había dejado pasar.

Tokens JWT con un servidor de recursos

La tienda no guarda contraseñas. Las identidades las maneja un **proveedor de identidad**, un servicio aparte como Keycloak, Auth0 o el de tu nube, que autentica al usuario y le entrega un token. La API solo comprueba ese token: en el vocabulario de OAuth2, es un **servidor de recursos** (*resource server*).



El proveedor firma; la API solo verifica. Por petición, nadie llama a nadie.

Un **JWT** (*JSON Web Token*) es un texto en tres partes: una cabecera, un cuerpo con datos (*claims*: quién es, quién lo emitió, cuándo vence, qué alcances tiene) y una firma. El proveedor firma con su clave privada; la API verifica con la clave pública. Si alguien cambia una coma del cuerpo, la firma deja de coincidir.

Con el starter **`spring-boot-starter-security-oauth2-resource-server`**, basta una propiedad:

```
application.yml
```

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${JWT_ISSUER_URI:http://localhost:9000/realms/tienda}
```

Con **`issuer-uri`**, Boot arma un **`JwtDecoder`** que descubre las claves públicas del emisor y valida que cada token diga ese mismo emisor (Spring Security Reference, 7.1). Lo crea perezoso: la app arranca aunque el proveedor no responda, y lo consulta con la primera petición.

Las autorizaciones salen del claim **`scope`** (o **`scp`**) del token: cada alcance se convierte en una autoridad con el prefijo **`SCOPE_`**. Por eso las reglas dicen **`hasAuthority("SCOPE_pedidos:escribir")`**.

Cómo se prueba sin proveedor. Los tests generan un par de claves RSA, firman tokens con la privada y le dan a la app un **`JwtDecoder`** que confía en la pública. Boot solo crea su decodificador si no hay otro, así que gana el de prueba:

TokensDePrueba.java

```java
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey((RSAPublicKey) CLAVES.getPublic()).build();
}
```

Con eso, los tests mandan tokens de verdad por la cadena de verdad:

SeguridadTest.java

```java
@Test
void sinTokenLaRespuestaEs401YDiceComoAutenticarse() throws Exception {
    mvc.perform(get("/api/pedidos/1"))
            .andExpect(status().isUnauthorized())
            .andExpect(header().string("WWW-Authenticate", containsString("Bearer")));
}

@Test
void conUnTokenValidoPasa() throws Exception {
    mvc.perform(crearPedido(tokens.token("ana", "pedidos:escribir")))
            .andExpect(status().isCreated());
}
```

Otros tests comprueban que un token vencido o firmado con otra clave da 401, y uno sin el alcance necesario, 403.

CSRF y CORS: cuándo aplican

Tienen nombres parecidos y resuelven problemas opuestos.

CSRF (*Cross-Site Request Forgery*) es un ataque contra el navegador: una página maliciosa hace que tu navegador mande una petición a otro sitio, y el navegador adjunta solo las cookies de ese sitio. Si la sesión vive en una cookie, la petición falsa llega autenticada. La protección de Spring Security exige un token extra que la página maliciosa no puede conocer.

Esta API no usa cookies: el token viaja en la cabecera **`Authorization`**, que el navegador nunca agrega solo. Una página maliciosa no tiene cómo hacer que la petición falsa llegue autenticada, y por eso aquí se desactiva, aunque el cliente sea JavaScript en un navegador.

> **Cuidado:** "mi API no tiene estado" no basta para desactivar CSRF. La documentación de Spring Security advierte que una app sin sesión puede ser vulnerable si autentica con cookies o con autenticación Basic, porque el navegador también las manda solo (Spring Security Reference, 7.1). La razón válida es que **la credencial no viaje en nada que el navegador adjunte automáticamente.**

CORS (*Cross-Origin Resource Sharing*) va al revés: no protege a la API, la **abre**. El navegador no deja que el JavaScript de **`https://tienda.example`** lea respuestas de **`https://api.tienda.example`**, porque son orígenes distintos; con CORS la API le dice qué orígenes sí pueden. Antes de la petición real, el navegador manda una **petición de comprobación** (*preflight*): un **`OPTIONS`** que pregunta si puede.

ConfiguracionSeguridad.java

```
@Bean
UrlBasedCorsConfigurationSource corsConfigurationSource(@Value("${app.cors.origenes}")
List<String> origenes) {
    var cors = new CorsConfiguration();
    cors.setAllowedOrigins(origenes);
    cors.setAllowedMethods(List.of("GET", "POST"));
    cors.setAllowedHeaders(List.of("Authorization", "Content-Type"));
    var porRuta = new UrlBasedCorsConfigurationSource();
    porRuta.registerCorsConfiguration("/api/**", cors);
    return porRuta;
}
```

El preflight no trae credenciales, así que CORS tiene que resolverse **antes** que la seguridad: si no, el preflight recibe un 401 y el navegador bloquea todo. Con **`cors(withDefaults())`**, Spring Security lo resuelve en su cadena: usa el **`CorsConfigurationSource`** si hay uno solo (con varios, no configura ninguno) y, si no hay, la configuración CORS de Spring MVC (Spring Security Reference, 7.1). Comprobé que el preflight desde el origen permitido recibe **`Access-Control-Allow-Origin`** y desde otro se rechaza.

> **El error típico:** "arreglar" un error de CORS con **`allowedOrigins("*")`**. Desaparece el error y también el control. En esta API el daño es acotado: el token va en una cabecera que el navegador no adjunta solo, así que un sitio ajeno solo leería lo que se responde sin autenticar. El mismo atajo es grave en

una API que solo se alcanza desde la red interna, porque el navegador del empleado hace de puente, o con la credencial en cookies: ahí Spring rechaza `*` junto con `allowCredentials(true)`, y la salida fácil, `allowedOriginPatterns("*")`, lo abre todo (Spring Framework API, `CorsConfiguration`). La lista de orígenes va en configuración, por ambiente.

Autorización por método con @PreAuthorize

Las reglas por URL alcanzan para "leer" y "escribir", pero cancelar un pedido es una decisión de negocio que no debería depender del nombre de la ruta. `@PreAuthorize` pone la regla en el método del servicio:

PedidoService.java

```
@PreAuthorize("hasAuthority('SCOPE_pedidos:admin')")
@Transactional
public PedidoRespuesta cancelar(Long id, long versionLeida) {
    var pedido = cargarEnVersion(id, versionLeida);
    pedido.cambiarA(EstadoPedido.CANCELADO);
    return PedidoRespuesta.de(repositorio.saveAndFlush(pedido));
}
```

Funciona porque `PedidoService` es un proxy (Parte 1): evalúa la expresión antes de entrar al método y, si no se cumple, lanza `AccessDeniedException`, que la cadena de filtros convierte en 403. Se activa con `@EnableMethodSecurity`; el starter no lo hace por defecto (Spring Security Reference, 7.1).

SeguridadTest.java

```
@Test
void cancelarExigeElAlcanceDeAdministrador() throws Exception {
    var escritor = tokens.token("ana", "pedidos:escribir");
    long id = idDelPedidoCreado(mvc, escritor);

    mvc.perform(conVersion("/api/pedidos/" + id + "/cancelacion", escritor, 0))
            .andExpect(status().isForbidden());

    var admin = tokens.token("soporte", "pedidos:escribir", "pedidos:admin");
    mvc.perform(conVersion("/api/pedidos/" + id + "/cancelacion", admin, 0))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.estado").value("CANCELADO"));
}
```

Lo que debes llevarte:

- La seguridad vive en una cadena de filtros antes de tu controlador, declarada con un bean `SecurityFilterChain`; sus reglas se evalúan en orden.
- Con JWT, el proveedor firma y la API verifica; los alcances llegan como autoridades `SCOPE_…`.

- CSRF se desactiva cuando la credencial no viaja en cookies; CORS abre la API a orígenes concretos, nunca a `*`.
- `@PreAuthorize` necesita `@EnableMethodSecurity` y, como todo proxy, no actúa en una llamada desde la misma clase.

– Para profundizar

un filtro propio se ubica relativo a uno conocido con `addFilterBefore` o `addFilterAfter`; el orden completo no hace falta conocerlo (Spring Security Reference, 7.1).

Boot 4.1 agregó `authorities-claim-expressions`, para leer las autoridades de otros claims del token; junto con `authority-prefix`, que ya existía y cambia el prefijo, sirve cuando tu proveedor pone los roles en un claim propio.

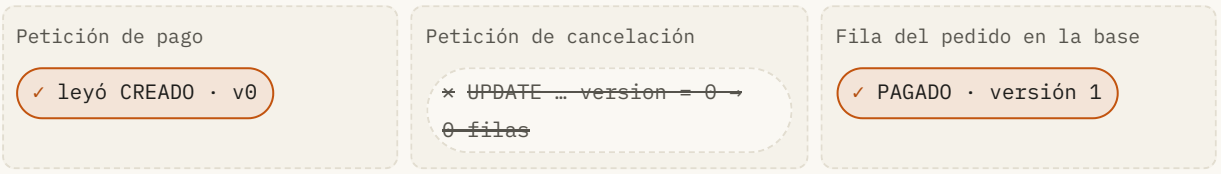
Si la app es a la vez API y sitio web con sesión, lo correcto son dos `SecurityFilterChain`, cada una con su `securityMatcher`: una sin estado para `/api/**` y otra con sesión y CSRF para el resto.

Concurrencia: dos peticiones sobre el mismo pedido

Ana paga su pedido desde el celular. En el mismo segundo, alguien de soporte lo cancela desde el panel, porque Ana había escrito una hora antes pidiendo cancelarlo. Las dos peticiones leen el pedido en estado `CREADO`, las dos dicen "perfecto, puedo cambiarlo" y las dos escriben. La que escribe última gana. El pedido queda `CANCELADO`, el cobro quedó hecho, y nadie recibió un error.

Eso es una **actualización perdida** (*lost update*): pasa siempre que dos transacciones hacen leer-modificar-escribir sobre la misma fila y nadie comprueba que la fila siga como la leyeron.

El pago y la cancelación escriben el mismo pedido



El pedido está CREADO. Con @Version, la fila tiene además una columna version, que empieza en 0.

Soporte abre el pedido para cancelarlo: su petición lee CREADO, versión 0, y se queda con esa copia en memoria.

Antes de que soporte confirme, llega el pago de Ana. Lee lo mismo: CREADO, versión 0.

El pago escribe primero y confirma. La fila queda PAGADO, y Hibernate sube la versión a 1 en el mismo UPDATE.

Sin @Version, la cancelación escribe su copia encima: UPDATE pedidos SET estado = 'CANCELADO' WHERE id = 7. La fila queda CANCELADO, el pago se perdió y las dos peticiones respondieron 200.

La misma escena con @Version: la cancelación escribe UPDATE ... WHERE id = 7 AND version = 0. Ninguna fila tiene ya la versión 0, así que el UPDATE no cambia nada, Hibernate lo detecta y la transacción falla. El pago queda.

Bloqueo optimista con @Version

El **bloqueo pesimista** bloquea la fila al leerla, y el segundo espera. El **bloqueo optimista** no bloquea nada: al escribir, comprueba que nadie haya escrito en el medio. En una API web, donde entre leer y escribir puede pasar un minuto mientras alguien mira la pantalla, sirve el optimista: nadie quiere una fila bloqueada un minuto.

En JPA se activa con una columna, que agrega la **V2** sobre la tabla de la Parte 2, y una anotación:

V2__version_en_pedidos.sql

```sql
ALTER TABLE pedidos ADD COLUMN version BIGINT NOT NULL DEFAULT 0;
```

La entidad, antes y después:

ANTES

PedidoSinVersion.java

```java
@Entity
@Table(name = "pedidos")
public class PedidoSinVersion {

    @Id
    private Long id;

    private String cliente;
    private String correo;

    @Enumerated(EnumType.STRING)
    private EstadoPedido estado;

    private Instant creadoEn;

    // …
```

DESPUÉS

Pedido.java

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

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

    private String cliente;
    private String correo;

    @Enumerated(EnumType.STRING)
    private EstadoPedido estado;

    private Instant creadoEn;

    @Version
    private long version;

    // …
```

Con **`@Version`**, Hibernate agrega **`AND version = ?`** a cada **`UPDATE`** y suma uno a la versión. Si el **`UPDATE`** no encuentra la fila con la versión que leyó, alguien escribió antes, y lanza una excepción que Spring traduce a **`ObjectOptimisticLockingFailureException`** (Spring Framework API, 7.0).

Este test reproduce la escena del tablero con dos transacciones reales. Para que el orden sea siempre el mismo uso **`REQUIRES_NEW`** (Parte 1): la transacción del pago corre completa en medio de la de la cancelación, en el mismo hilo.

ActualizacionPerdidaTest.java

```
@Test
void sinVersionLaCancelacionBorraElPagoSinQueNadieSeEntere() {
    long id = pedidoNuevo();

    pagoYCancelacionALaVez(PedidoSinVersion.class, id, PedidoSinVersion::cambiarA);

    assertThat(estadoEnLaBase(id)).isEqualTo("CANCELADO"); // y el pago se perdió
}

@Test
void conVersionLaSegundaEscrituraFallaYElPagoQueda() {
    long id = pedidoNuevo();

    assertThatThrownBy(() -> pagoYCancelacionALaVez(Pedido.class, id, Pedido::cambiarA))
            .isInstanceOf(ObjectOptimisticLockingFailureException.class);

    assertThat(estadoEnLaBase(id)).isEqualTo("PAGADO");
}

private <T> void pagoYCancelacionALaVez(Class<T> entidad, long id, BiConsumer<T,
EstadoPedido> cambiar) {

enTransaccion(TransactionDefinition.PROPAGATION_REQUIRED).executeWithoutResult(cancelaci
on -> {
        T leidoPorLaCancelacion = em.find(entidad, id); // lee CREADO


enTransaccion(TransactionDefinition.PROPAGATION_REQUIRES_NEW).executeWithoutResult(pago
->
                cambiar.accept(em.find(entidad, id), EstadoPedido.PAGADO)); // lee
CREADO, escribe PAGADO, confirma

        cambiar.accept(leidoPorLaCancelacion, EstadoPedido.CANCELADO); // escribe sobre
lo que leyó
    });
}
```

`PedidoSinVersion` mapea la misma tabla y solo existe en los tests, como el antes.

La versión tiene que viajar al cliente

`@Version` protege a dos transacciones que se solapan. Pero entre "soporte abre el pedido" y "soporte aprieta cancelar" hay dos peticiones HTTP, no una transacción: si la cancelación vuelve a leer el pedido, lee la versión 1 y la escritura pasa. La versión tiene que **viajar**: la API la devuelve al leer, el cliente la manda al escribir y el servicio compara:

PedidoService.java

```
private Pedido cargarEnVersion(Long id, long versionLeida) {
    var pedido = cargar(id);
    if (pedido.getVersion() != versionLeida) {
        throw new ObjectOptimisticLockingFailureException(Pedido.class, id);
    }
    return pedido;
}
```

PedidoService.java

```
@Transactional
public PedidoRespuesta pagar(Long id, long versionLeida) {
    var pedido = cargarEnVersion(id, versionLeida);
    pedido.cambiarA(EstadoPedido.PAGADO);
    eventos.publishEvent(new PedidoPagado(pedido.getId(), pedido.getCorreo()));
    return PedidoRespuesta.de(repositorio.saveAndFlush(pedido));
}
```

`saveAndFlush` fuerza el **`UPDATE`** antes de armar la respuesta, para que el cliente reciba la versión nueva y no la que leyó.

Qué devolverle al cliente: 409

Un conflicto no es un error del servidor: es decirle al cliente "otro llegó antes, vuelve a leer". El código HTTP es **409 Conflict**:

ErroresDePedidos.java

```
@ExceptionHandler(ObjectOptimisticLockingFailureException.class)
ProblemDetail conflicto(ObjectOptimisticLockingFailureException e) {
    var problema = ProblemDetail.forStatusAndDetail(
            HttpStatus.CONFLICT, "Otro cambio llegó antes que el tuyo. Vuelve a leer el
pedido y reintenta.");
    problema.setTitle("El pedido cambió mientras lo editabas");
    return problema;
}
```

ConflictoHttpTest.java

```
@Test
void quienEscribeConUnaVersionViejaRecibe409() throws Exception {
    var cliente = tokens.token("ana", "pedidos:escribir");
    var soporte = tokens.token("soporte", "pedidos:escribir", "pedidos:admin");
    long id = idDelPedidoCreado(mvc, cliente); // los dos leen la versión 0

    mvc.perform(conVersion("/api/pedidos/" + id + "/pago", cliente, 0))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.version").value(1));

    mvc.perform(conVersion("/api/pedidos/" + id + "/cancelacion", soporte, 0))
            .andExpect(status().isConflict())
            .andExpect(jsonPath("$.title").value("El pedido cambió mientras lo
editabas"));
}
```

El error típico: reintentar automáticamente al recibir el conflicto. Si el servidor reintenta la cancelación con la versión nueva, vuelves al punto de partida: cancelas un pedido pagado sin que nadie vea que se pagó. El reintento lo decide una persona que vio el estado nuevo.

Lo que debes llevarte: leer-modificar-escribir sin control pierde actualizaciones en silencio. **`@Version`** lo detecta en el **`UPDATE`**; entre peticiones HTTP, la versión va y vuelve con el cliente, y el conflicto se responde con **409**.

– **Para profundizar**

el mecanismo estándar de HTTP es que la respuesta lleve la versión en la cabecera **`ETag`**, el cliente la devuelva en **`If-Match`**, y si no coincide el servidor responda **412 Precondition Failed** (RFC 9110). Es el mecanismo del propio protocolo, y le pide más trabajo al cliente; mandar la versión en el cuerpo es lo más claro para empezar.

El bloqueo pesimista existe en Spring Data con **`@Lock(LockModeType.PESSIMISTIC_WRITE)`** en un método del repositorio. Tiene sentido cuando el choque es frecuente y la transacción es corta, como reservar stock de un producto muy pedido: ahí esperar un instante sale más barato que reintentar cada conflicto.

Eventos después del commit

Cuando un pedido pasa a **`PAGADO`**, la tienda le manda un correo a Ana. Como en la Parte 1, eso va desacoplado con un evento: el servicio publica **`PedidoPagado`** y un listener manda el correo.

Ahora el pago se registra dentro de una operación más grande: pagar, reservar stock, facturar. La reserva falla, la transacción entera se deshace y el pedido vuelve a **`CREADO`**. Pero el correo "Recibimos el pago de tu pedido" ya salió: **`@EventListener`** se ejecuta **en el momento** en que se publica el evento, dentro de una transacción que todavía no se sabe si va a confirmarse.

@TransactionalEventListener

`@TransactionalEventListener` registra el listener en la transacción en curso y lo ejecuta en una de sus fases; la fase por defecto es **`AFTER_COMMIT`**, solo si se confirmó (Spring Framework Reference, 7.0).

ANTES

AvisoDespuesDelCommitTest.java

```
static class AvisoInmediato {

    private final NotificadorEnLog
notificador = new NotificadorEnLog();

    @EventListener
    void alPagar(PedidoPagado evento) {

notificador.enviar(evento.correo(),
"Recibimos el pago de tu pedido " +
evento.pedidoId());
    }
}
```

DESPUÉS

AvisoDePago.java

```
@Component
class AvisoDePago {

    private final Notificador
notificador;

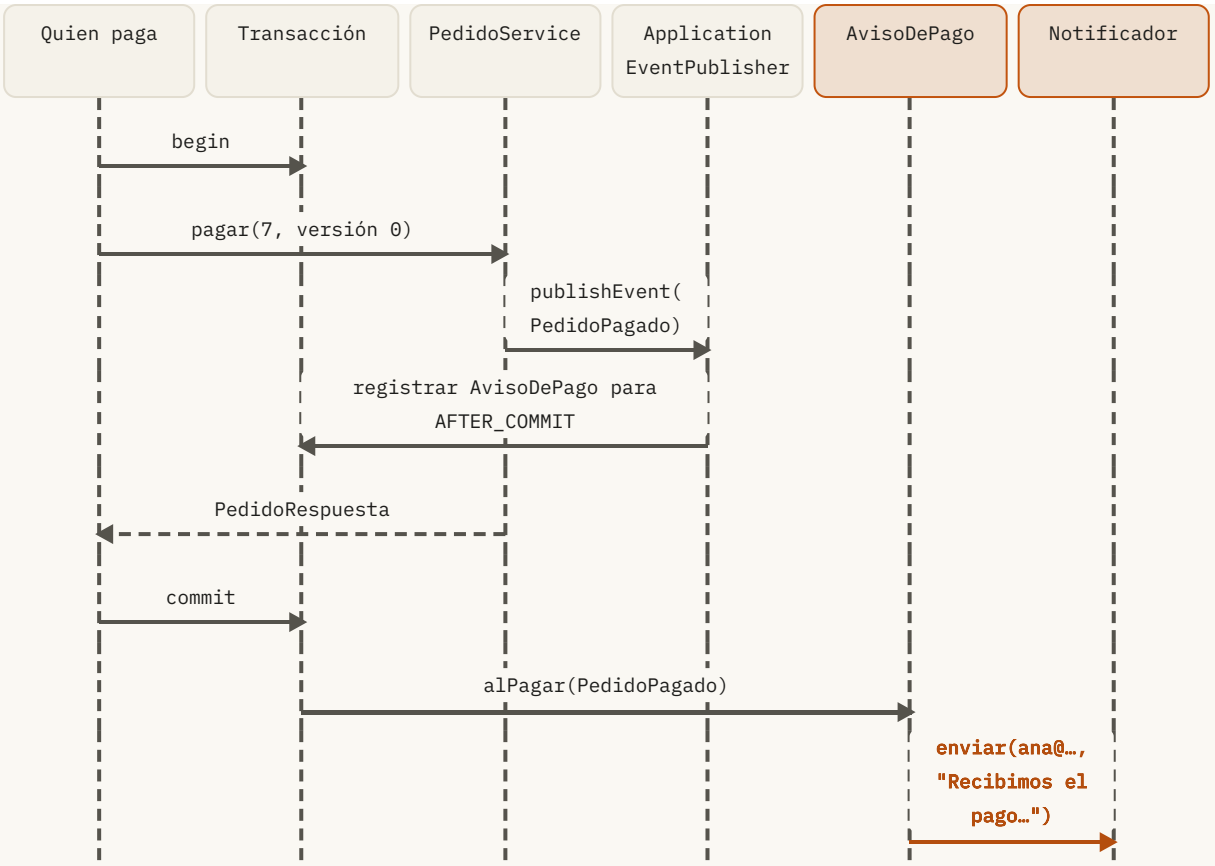
    AvisoDePago(Notificador notificador)
{
        this.notificador = notificador;
    }

    @TransactionalEventListener
    void alPagar(PedidoPagado evento) {

notificador.enviar(evento.correo(),
"Recibimos el pago de tu pedido " +
evento.pedidoId());
    }
}
```

Cambia una anotación, y con ella el orden de los mensajes:

El aviso de pago espera al commit



Empieza la transacción de la operación completa: pagar, reservar stock, facturar.

El servicio cambia el pedido a PAGADO. Todavía no hay nada confirmado en la base.

Publica el evento, igual que en la Parte 1.

Acá está la diferencia. Con @EventListener, el publicador llamaría al listener ahora mismo. Con @TransactionalEventListener, lo anota en la transacción en curso para más tarde.

pagar() termina sin que nadie haya mandado ningún correo.

La reserva de stock y la factura salieron bien: la transacción se confirma. Si hubieran fallado, aquí habría un rollback, y los dos pasos siguientes no ocurrirían nunca.

Con el commit hecho, la transacción ejecuta lo que tenía anotado para AFTER_COMMIT.

El correo sale solo cuando el pago ya es un hecho en la base.

Lo comprobé con los dos listeners a la vez, ante una transacción que se deshace:

AvisoDespuesDelCommitTest.java

```
@Test
void siLaTransaccionSeDeshaceSoloElAvisoInmediatoSale() {
    long id = pedidoNuevo();

    assertThatThrownBy(() -> transaccion.executeWithoutResult(t -> {
                pedidos.pagar(id, 0);
                throw new IllegalStateException("el cobro con la tarjeta falló");
            }))
            .hasMessage("el cobro con la tarjeta falló");
```

```
    assertThat(pedidos.buscar(id).estado()).isEqualTo(EstadoPedido.CREADO); // no se
pagó
    assertThat(inmediato.notificador.enviados()).anyMatch(m -> m.endsWith("pedido " +
id)); // y avisó igual
    assertThat(notificador.enviados()).noneMatch(m -> m.endsWith("pedido " + id)); //
este, no
}
```

El error típico: escribir en la base desde un listener **`AFTER_COMMIT`** y esperar que se guarde. La transacción original ya se confirmó, pero sus recursos pueden seguir activos: lo que escribas "participa" de ella y nunca se confirma. La documentación de Spring pide usar **`REQUIRES_NEW`** para cualquier operación transaccional que se llame desde ahí (Spring Framework API, 7.0).

Lo que debes llevarte: lo que sale de tu sistema (correos, mensajes a otros servicios) va en un **`@TransactionalEventListener`**, que por defecto corre **después del commit** y no corre si hubo rollback.

– Para profundizar

si se publica el evento sin ninguna transacción en curso, un **`@TransactionalEventListener` no se ejecuta**, salvo que le pongas **`fallbackExecution = true`** (Spring Framework Reference, 7.0). Es fácil de olvidar en un método sin **`@Transactional`**.

`AFTER_COMMIT` tampoco es una garantía de entrega: si la app se cae entre el commit y el envío, el correo no sale nunca. Cuando eso importa, el patrón es el *outbox*: guardar el evento en una tabla en la misma transacción y publicarlo después desde ahí. Spring Modulith trae un registro de publicación de eventos con esa idea: guarda cada evento en la transacción original y deja pendientes los que no se completaron, para reintentarlos (Spring Modulith Reference, 2.1).

Caché

Cada línea de un pedido pregunta el precio a **`FuenteDePrecios`**, que en una tienda real sería otro servicio o una consulta cara. Los precios cambian poco y se piden mucho: el caso de libro para una **caché**, que guarda el resultado de una llamada y lo devuelve la próxima vez sin repetirla.

En Spring anotas el método y Spring decide si llamarlo o devolver lo guardado. Con **`spring-boot-starter-cache`** y Caffeine en el classpath, Boot configura un **`CacheManager`** en memoria (Spring Boot Reference, 4.1):

pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
</dependency>
```

application.yml

```yaml
spring:
  cache:
    cache-names: precios
    caffeine:
      spec: maximumSize=1000,expireAfterWrite=10m
```

Más **`@EnableCaching`** en una clase de configuración:

Catalogo.java

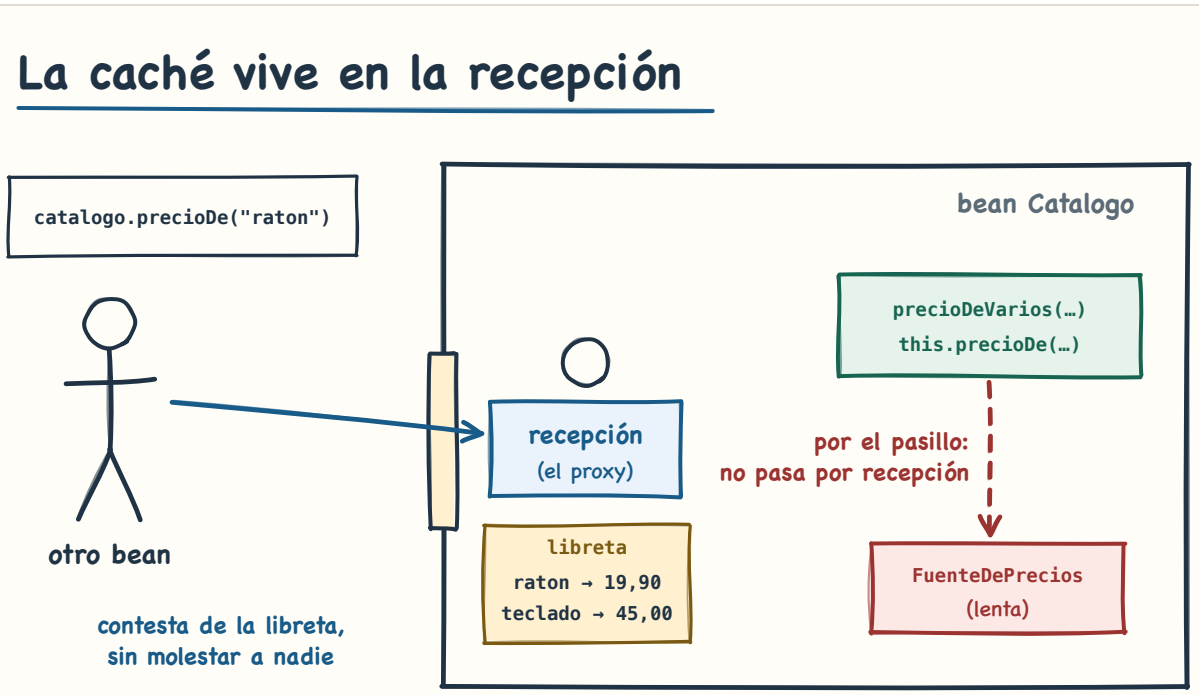
```java
@Cacheable("precios")
public BigDecimal precioDe(String producto) {
    return fuente.consultar(producto);
}

@CacheEvict(cacheNames = "precios", key = "#producto")
public void actualizarPrecio(String producto, BigDecimal precio) {
    fuente.fijar(producto, precio);
}
```

`@Cacheable("precios")` usa el argumento como clave. **`@CacheEvict`** borra la entrada cuando el precio cambia; sin eso, la caché serviría el precio viejo hasta que venza. **Invalidar** es lo difícil de una caché.

La caché también es un proxy

¿Cómo sabe Spring que tiene que mirar la caché antes de ejecutar **`precioDe`**? Porque **`Catalogo`** es un proxy, y la caché vive en el proxy.



La libreta la tiene la recepción. Lo que se pide desde adentro de la oficina no pasa por ella.

Este método suma los precios de varios productos:

Catalogo.java

```
public BigDecimal precioDeVarios(List<String> productos) {
    return productos.stream()
            .map(this::precioDe)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
}
```

`this::precioDe` es una llamada de la clase a sí misma: no pasa por el proxy y no hay caché. En modo proxy, el modo por defecto, solo se interceptan las llamadas externas (Spring Framework Reference, 7.0). Los tests cuentan las consultas a la fuente:

CacheTest.java

```
@Test
void laSegundaConsultaNoLlegaALaFuente() {
    int antes = catalogo.consultasALaFuente();

    catalogo.precioDe("raton");
    catalogo.precioDe("raton");

    assertThat(catalogo.consultasALaFuente() - antes).isEqualTo(1);
```

```
}

@Test
void desdeDentroDeLaClaseLaCacheNoExiste() {
    catalogo.precioDe("raton"); // queda en la caché
    int antes = catalogo.consultasALaFuente();

    catalogo.precioDeVarios(List.of("raton", "raton"));

    assertThat(catalogo.consultasALaFuente() - antes).isEqualTo(2);
}
```

El error típico: poner **`@Cacheable`** en un método privado o llamarlo desde la misma clase, y concluir que "la caché no anda". La caché anda; la llamada no pasa por ella. La salida es que el método cacheado lo llame otro bean, o mover la lógica que lo usa a otra clase.

Lo que debes llevarte: la caché vive en el **proxy**, así que la autoinvocación no la usa. Y Caffeine guarda en la memoria de **cada instancia**: con tres instancias hay tres cachés, y un **`@CacheEvict`** solo limpia la de la instancia que lo ejecutó.

– Para profundizar

si necesitas una caché compartida entre instancias, Boot configura un **`RedisCacheManager`** cuando Redis está disponible y configurado (Spring Boot Reference, 4.1). Ganas coherencia entre instancias y pagas una llamada de red por cada lectura, y otro servicio que operar.

`expireAfterWrite` pone un techo a lo viejo que puede estar un dato aunque falle la invalidación. Elegirlo es una decisión de negocio: cuánto tiempo puede ver un cliente un precio que ya cambió.

Trabajo en segundo plano

Hay trabajo que no tiene que hacerse mientras el cliente espera, como exportar un informe, y trabajo que nadie pide y corre solo a una hora, como cancelar los pedidos que llevan dos días sin pagarse.

@Async y su ejecutor

`@Async` hace que un método se ejecute en otro hilo: quien lo llama recibe enseguida un **`CompletableFuture`** y sigue.

ExportadorDePedidos.java

```
@Async
public CompletableFuture<Exportacion> exportar() {
    long cuantos = repositorio.count();
    return CompletableFuture.completedFuture(new Exportacion(cuantos,
Thread.currentThread()));
}
```

Funciona por proxy y se activa con **`@EnableAsync`**. Corre en el ejecutor que configura Boot, **`applicationTaskExecutor`**, un pool de hilos (Spring Boot Reference, 4.1); con un test comprobé que sus nombres empiezan con **`task-`**.

> El error típico: un **`@Async`** que devuelve **`void`** y lanza una excepción. Quien lo llamó ya siguió y nunca se entera: la excepción, como mucho, queda en el log. Devuelve un **`CompletableFuture`** y trata el error, o configura un **`AsyncUncaughtExceptionHandler`**.

@Scheduled con varias instancias

La limpieza de pedidos sin pagar es una tarea programada. **`@EnableScheduling`** y una expresión cron alcanzan:

LimpiezaDePedidos.java

```
@Scheduled(cron = "${app.limpieza.cron:0 0 3 * * *}")
@SchedulerLock(name = "cancelar-pedidos-sin-pagar", lockAtLeastFor = "PT1M",
lockAtMostFor = "PT10M")
@Transactional
public void cancelarSinPagar() {
    ejecuciones.incrementAndGet();
    int cancelados =
repositorio.cancelarSinPagarAntesDe(reloj.instant().minus(Duration.ofDays(2)));
    log.info("Pedidos sin pagar cancelados: {}", cancelados);
}
```

Ignora por un momento la segunda anotación. Con **`@Scheduled`** solo, el cron se dispara en **cada instancia**: con tres instancias detrás de un balanceador, la tarea corre tres veces a las 3:00. Si manda un resumen diario por correo, los clientes reciben tres.

`@Scheduled` no sabe que existen otras instancias. La solución más usada es **ShedLock**, una biblioteca aparte que guarda un candado en una tabla que comparten todas; su versión 7 es la compatible con Spring Framework 7 y Spring Boot 4 (ShedLock, README).

pom.xml

```
<dependency>
```

```
    <groupId>net.javacrumbs.shedlock</groupId>
    <artifactId>shedlock-spring</artifactId>
    <version>${shedlock.version}</version>
</dependency>
<dependency>
    <groupId>net.javacrumbs.shedlock</groupId>
    <artifactId>shedlock-provider-jdbc-template</artifactId>
    <version>${shedlock.version}</version>
</dependency>
```

V3__shedlock.sql

```
CREATE TABLE shedlock (
    name       VARCHAR(64)  NOT NULL,
    lock_until TIMESTAMP    NOT NULL,
    locked_at  TIMESTAMP    NOT NULL,
    locked_by  VARCHAR(255) NOT NULL,
    PRIMARY KEY (name)
);
```

ConfiguracionTareas.java

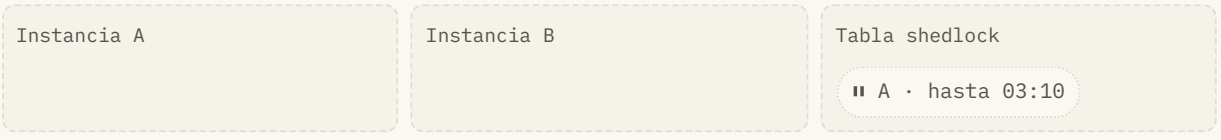
```
@Configuration(proxyBeanMethods = false)
@EnableScheduling
@EnableSchedulerLock(defaultLockAtMostFor = "PT10M")
class ConfiguracionTareas {

    @Bean
    LockProvider candados(DataSource dataSource) {
        return new
JdbcTemplateLockProvider(JdbcTemplateLockProvider.Configuration.builder()
                .withJdbcTemplate(new JdbcTemplate(dataSource))
                .usingDbTime()
                .build());
    }
}
```

`usingDbTime()` usa la hora de la base y no la de cada instancia, como recomienda el README.

Así funciona:

Dos instancias, un cron, un candado



A las 3:00 el cron se dispara en las dos instancias, con un segundo de diferencia. El candado está libre.

A intenta tomar el candado: un UPDATE (o un INSERT la primera vez) que solo funciona si el candado está libre. Lo toma hasta 03:10, que es lockAtMostFor.

B intenta lo mismo un segundo después. El UPDATE no encuentra un candado libre y no modifica ninguna fila. B no espera: se salta esta ejecución.

A termina en dos segundos. No libera el candado del todo: lo deja hasta 03:01, que es lockAtLeastFor. Una tarea tan rápida dejaría el candado libre antes de que llegue una instancia con el reloj un poco atrasado.

Justamente: una tercera instancia, o B con el reloj atrasado, dispara su cron a las 03:00:40. El candado sigue tomado, así que también se salta.

Si A se hubiera caído en medio de la tarea, nadie liberaría el candado. Por eso existe lockAtMostFor: a las 03:10 vence solo, y la próxima ejecución puede tomarlo.

El test simula las dos instancias con dos llamadas seguidas: el candado vive en la base, así que da igual de qué máquina venga cada una.

CandadoDeTareasTest.java

```
@Test
void conCandadoLaSegundaInstanciaSeSaltaLaEjecucion() {
    jdbc.update("delete from shedlock");
    int antes = limpieza.ejecuciones();

    limpieza.cancelarSinPagar(); // la instancia A toma el candado
    limpieza.cancelarSinPagar(); // la B llega segundos después: sigue tomado
(lockAtLeastFor)

    assertThat(limpieza.ejecuciones() - antes).isEqualTo(1);
    assertThat(jdbc.queryForObject(
                    "select count(*) from shedlock where name = 'cancelar-pedidos-sin-
pagar' and lock_until > now()",
                    Integer.class))
            .isEqualTo(1);
}
```

> **Cuidado:** ShedLock garantiza que la tarea **no corra dos veces a la vez**, no que corra: una ejecución saltada se pierde. Y **`lockAtMostFor`** tiene que superar lo que tarda la tarea en el peor caso, o el candado vence mientras sigue corriendo y otra instancia empieza la suya.

Hilos virtuales

Tomcat atiende cada petición en un hilo. Los hilos del sistema operativo son caros, así que el pool es limitado, y cuando todas las peticiones esperan a la base, se llena de hilos que no hacen nada.

Los **hilos virtuales** de Java 21 son hilos baratos que maneja la JVM: cuando uno se bloquea, la JVM usa el hilo real para otro. En Boot se activan con **`spring.threads.virtual.enabled=true`**, y entonces Tomcat, **`@Async`** y **`@Scheduled`** usan hilos virtuales (Spring Boot Reference, 4.1); lo de **`@Async`** lo comprobé con un test. La referencia recomienda con fuerza Java 24 o posterior, porque antes un hilo virtual puede quedar **anclado** a su hilo real y perder rendimiento. Y como son *daemon*, una app que solo tiene tareas programadas puede terminar sola: para eso está **`spring.main.keep-alive=true`**.

Lo que la propiedad no resuelve: los hilos virtuales quitan el límite de hilos, no el de conexiones. El pool de Boot, HikariCP, tiene 10 por defecto (HikariCP, README), y miles de hilos virtuales terminan en la fila de esas 10.

HilosVirtualesTest.java

```
@Test
void veinteHilosVirtualesYDosConexionesSoloTrabajanDosALaVez() throws Exception {
    var adentro = new AtomicInteger();
    var maximo = new AtomicInteger();
    long inicio = System.nanoTime();

    try (var hilos = Executors.newVirtualThreadPerTaskExecutor()) {
        for (int i = 0; i < 20; i++) {
            hilos.submit(() -> jdbc.execute((Connection conexion) -> {
                maximo.accumulateAndGet(adentro.incrementAndGet(), Math::max);
                try (var consulta = conexion.createStatement()) {
                    consulta.execute("select pg_sleep(0.2)"); // 200 ms de "trabajo" en
la base
                } finally {
                    adentro.decrementAndGet();
                }
                return null;
            }));
        }
    }

    var duracion = Duration.ofNanos(System.nanoTime() - inicio);
    assertThat(maximo.get()).isEqualTo(2); // el pool, no los hilos, pone el límite
    assertThat(duracion).isGreaterThanOrEqualTo(Duration.ofMillis(1900)); // 20 / 2 ×
200 ms
}
```

Veinte hilos, dos conexiones, 200 ms por consulta: nunca hay más de dos consultas a la vez, y el total tarda diez rondas. El cuello de botella es el pool, y detrás del pool, la base.

MVC, WebFlux o hilos virtuales

Spring MVC, el de toda la serie, es imperativo: si el método espera, el hilo espera. **WebFlux** es reactivo: el código es una cadena de operaciones sobre flujos de datos, y ningún hilo espera. Los hilos virtuales le dan

a MVC parte de lo que WebFlux prometía.

La documentación de Spring Framework es más prudente que muchos artículos: si tu aplicación Spring MVC funciona bien, no hace falta cambiar; con persistencia bloqueante, como JPA o JDBC, Spring MVC es la mejor opción para las arquitecturas comunes; y lo reactivo, en general, no hace que las aplicaciones corran más rápido (Spring Framework Reference, 7.0). Esta app usa JPA, así que WebFlux no le daría nada sin cambiar también el acceso a datos.

> **Lo que debes llevarte:** **`@Scheduled`** corre en **cada instancia**; **ShedLock** pone un candado en la base para que corra en una. Los **hilos virtuales** quitan el límite de hilos, no el de conexiones.

> **– Para profundizar**
>
> **`@Scheduled`** y **`@Async`** usan ejecutores distintos. El de **`@Scheduled`** es de un solo hilo por defecto: una tarea lenta retrasa a todas las demás. Se ajusta con **`spring.task.scheduling.pool.size`**; con hilos virtuales, las propiedades de tamaño de pool dejan de tener efecto (Spring Boot Reference, 4.1).
>
> Agrandar el pool de Hikari no es la respuesta automática al cuello de botella: cada conexión es un proceso en Postgres (PostgreSQL Documentation), y la base tiene su propio límite de conexiones. Multiplica el tamaño del pool por la cantidad de instancias antes de subirlo.

Observabilidad

Una app en producción tiene que responder, sin que nadie entre al servidor: ¿estás viva?, ¿estás lista para recibir tráfico?, ¿qué pasa adentro? Para eso, Boot tiene **Actuator**: endpoints de operación bajo **`/actuator`**.

Qué exponer y qué no

Algunos endpoints muestran lo que no debería salir de la empresa: **`env`** lista la configuración, **`heapdump`** descarga la memoria entera de la JVM. Por eso Boot solo expone **`health`** por HTTP (Spring Boot Reference, 4.1), y lo demás lo agregas tú:

application.yml

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics
  endpoint:
```

```
    health:
      group:
        readiness:
          include: readinessState,calentamiento
```

La cadena de seguridad deja pública solo la salud; las métricas piden un token de administrador:

ActuatorTest.java

```
@Test
void laSaludYLasSondasSonPublicas() throws Exception {
    mvc.perform(get("/actuator/health")).andExpect(status().isOk());
    mvc.perform(get("/actuator/health/liveness")).andExpect(status().isOk());
    mvc.perform(get("/actuator/health/readiness")).andExpect(status().isOk());
}

@Test
void lasMetricasPidenToken() throws Exception {
    mvc.perform(get("/actuator/metrics")).andExpect(status().isUnauthorized());
    mvc.perform(leer("/actuator/metrics", tokens.token("ops",
"pedidos:admin"))).andExpect(status().isOk());
}

@Test
void envYHeapdumpNoExistenNiParaElAdministrador() throws Exception {
    var admin = tokens.token("ops", "pedidos:admin");
    mvc.perform(leer("/actuator/env", admin)).andExpect(status().isNotFound());
    mvc.perform(leer("/actuator/heapdump", admin)).andExpect(status().isNotFound());
}
```

env y **heapdump** dan 404 incluso con el token más poderoso: no están protegidos, **no existen**. Es la mejor seguridad para un endpoint que no necesitas.

> **Nota:** En este sitio, el backend expone solo **`health`** e **`info`**, e **`info`** incluye el commit exacto del jar que corre. El despliegue lo usa: su último paso consulta **`/actuator/health`** y **`/actuator/info`** hasta que la salud dice **`UP`** y el commit servido es el del release.

Vivo no es lo mismo que listo

Un orquestador como Kubernetes le pregunta dos cosas distintas a cada instancia:

- **Liveness** (¿estás viva?): si la respuesta es no, la instancia está rota sin arreglo y hay que reiniciarla.
- **Readiness** (¿estás lista?): si es no, la instancia está bien, pero no hay que mandarle tráfico todavía, o ya no.

Boot expone las dos como grupos de salud, en **`/actuator/health/liveness`** y **`/actuator/health/readiness`**. En Boot 4 vienen activadas por defecto en cualquier entorno; hasta Boot 3.5, solo cuando la app detectaba que corría en Kubernetes (Spring Boot Reference, 4.1; guía de migración a Boot 4.0).

Al arrancar, la tienda precarga la caché de precios para que el primer cliente no pague el arranque en frío. Para eso alcanza con un **`ApplicationRunner`** o, como hace **`Calentamiento`**, un listener de **`ApplicationReadyEvent`**: Boot publica el estado **`ACCEPTING_TRAFFIC`** justo después de ese evento, así que mientras precarga, readiness ya dice que no (Spring Boot Reference, 4.1).

Un indicador de salud propio hace falta cuando lo que decide si la instancia está lista no coincide con el arranque: un calentamiento que corre en segundo plano, o algo que puede dejar de estar listo en caliente. Lo muestro con la precarga porque es el caso más simple, y porque deja el motivo escrito en la respuesta de salud:

CalentamientoHealthIndicator.java

```
@Component
class CalentamientoHealthIndicator implements HealthIndicator {

    private final Calentamiento calentamiento;

    CalentamientoHealthIndicator(Calentamiento calentamiento) {
        this.calentamiento = calentamiento;
    }

    @Override
    public Health health() {
        return calentamiento.terminado()
                ? Health.up().build()
                : Health.down().withDetail("motivo", "precargando la caché de
precios").build();
    }
}
```

Un **`HealthIndicator`** es un bean que responde **`UP`** o **`DOWN`**; Boot lo nombra por el nombre del bean, sin el sufijo **`HealthIndicator`**: **`calentamiento`**. El grupo **`readiness`** del **`application.yml`** de arriba lo incluye. El test apaga la precarga automática y la llama a mano, para ver el antes y el después:

ReadinessTest.java

```
@Test
void vivaPeroNoListaHastaTerminarDeCalentar() throws Exception {
    mvc.perform(get("/actuator/health/liveness")).andExpect(status().isOk());
    mvc.perform(get("/actuator/health/readiness"))
            .andExpect(status().isServiceUnavailable())
            .andExpect(jsonPath("$.status").value("DOWN"));
```

```
    calentamiento.ejecutar();

    mvc.perform(get("/actuator/health/readiness")).andExpect(status().isOk());
}
```

Mientras precarga, readiness responde **503** y el balanceador no le manda tráfico; liveness responde 200 y nadie la reinicia.

> **El error típico:** meter la base de datos en el chequeo de liveness. Si la base tiene un mal momento, todas las instancias dicen "estoy rota" a la vez, el orquestador las reinicia todas, y un problema de la base se vuelve una caída total. La documentación de Boot lo advierte: liveness no debe depender de sistemas externos (Spring Boot Reference, 4.1). Si la app no puede atender sin la base, eso es readiness.

Métricas con Micrometer

Micrometer es la biblioteca de métricas de Boot: una fachada sobre Prometheus, Datadog y compañía, como SLF4J para los logs. Actuator ya trae métricas de la JVM, Tomcat, Hikari y cada endpoint. Una de negocio es un contador que **`PedidoService`** registra con el **`MeterRegistry`**:

PedidoService.java

```
this.creados = Counter.builder("pedidos.creados")
        .description("Pedidos registrados desde que arrancó la instancia")
        .register(metricas);
```

Cada **`crear()`** llama a **`creados.increment()`**, y **`/actuator/metrics/pedidos.creados`** muestra el valor. Para mandarlas a un sistema de monitoreo, agregas su registro (por ejemplo, el de Prometheus) y Boot lo configura.

Logs estructurados con un ID de correlación

Con tres instancias y miles de peticiones por minuto, los logs se leen con una herramienta que busca y filtra, y esa herramienta necesita campos, no frases. Boot 4.1 escribe **logs estructurados** en JSON con **`logging.structured.format.console`**, en formato **`ecs`**, **`gelf`** o **`logstash`** (Spring Boot Reference, 4.1).

Para seguir una petición, se le da un **ID de correlación** que va en cada línea de log que produce. Un filtro lo pone en el **MDC** (*Mapped Diagnostic Context*), un mapa por hilo que la biblioteca de logs lee al escribir:

IdDeCorrelacion.java

```
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
class IdDeCorrelacion extends OncePerRequestFilter {
```

```
    static final String CABECERA = "X-Correlation-Id";

    @Override
    protected void doFilterInternal(HttpServletRequest peticion, HttpServletResponse
respuesta, FilterChain cadena)
            throws ServletException, IOException {
        var recibido = peticion.getHeader(CABECERA);
        var id = recibido != null && recibido.matches("[A-Za-z0-9-]{1,64}") ? recibido :
UUID.randomUUID().toString();
        MDC.put("correlacion", id);
        respuesta.setHeader(CABECERA, id);
        try {
            cadena.doFilter(peticion, respuesta);
        } finally {
            MDC.remove("correlacion");
        }
    }
}
```

Boot agrega al JSON cada valor del MDC (Spring Boot Reference, 4.1):

LogsEstructuradosTest.java

```
@Test
void cadaLineaDelLogLlevaElIdDeCorrelacionDeSuPeticion(CapturedOutput salida) throws
Exception {
    mvc.perform(crearPedido(tokens.token("ana", "pedidos:escribir")).header("X-
Correlation-Id", "compra-7f3a"))
            .andExpect(status().isCreated())
            .andExpect(header().string("X-Correlation-Id", "compra-7f3a"));

    assertThat(salida.getOut().lines().filter(l -> l.contains("creado con 1 líneas")))
            .singleElement()
            .satisfies(linea -> assertThat(linea)
                    .startsWith("{")
                    .contains("\"correlacion\":\"compra-7f3a\"")
                    .contains("\"level\":\"INFO\""));
}
```

Y esta es la línea que escribió ese test:

```
{"@timestamp":"2026-11-04T18:36:41.652616-06:00","@version":"1","message":"Pedido 1
creado con 1
líneas","logger_name":"com.ejemplo.pedidos.pedidos.PedidoService","thread_name":"main","
level":"INFO","level_value":20000,"correlacion":"compra-7f3a"}
```

– Para profundizar

si usas Micrometer Tracing, Boot ya agrega a cada línea un ID de correlación armado con el **`traceId`** y el **`spanId`** de la traza, y además viaja entre servicios en las cabeceras de cada llamada (Spring Boot Reference, 4.1). El filtro de este ejemplo es la versión mínima, para que veas el mecanismo; en un sistema con varios servicios, la traza es la solución completa.

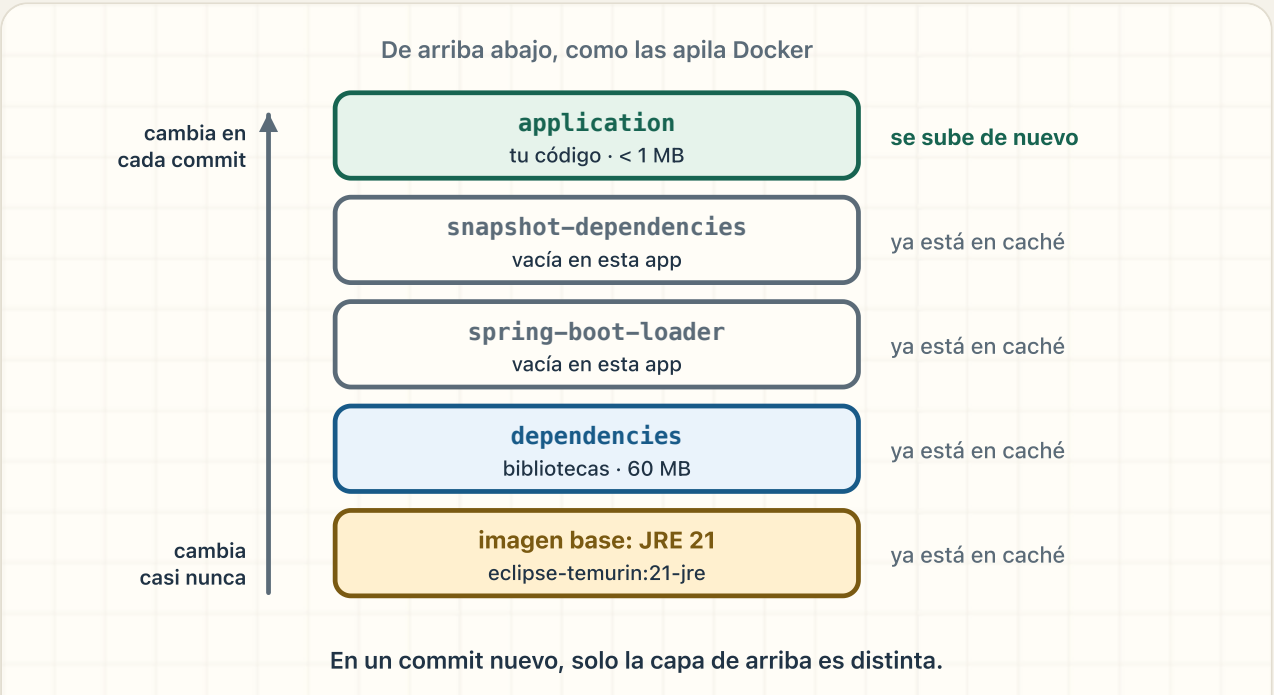
El filtro valida el ID que recibe antes de usarlo. Un valor que viene del cliente y termina en tus logs es una entrada de usuario: sin validarlo, alguien puede inyectar saltos de línea o texto que confunda a quien lee los logs.

Lo que debes llevarte: expón solo lo que necesitas. **Liveness** decide reinicios y **readiness**, tráfico; lo externo va en readiness.

Empaquetar y desplegar

La imagen: Dockerfile con capas o buildpacks

Una imagen de contenedor es una pila de **capas**, y Docker reutiliza las que no cambiaron. Un jar de Boot pesa decenas de megas: si lo copias entero, cada commit genera una capa nueva de ese tamaño, aunque solo hayas cambiado una línea.



> Lo que cambia poco va abajo. Un commit nuevo solo reconstruye la capa de arriba.

Boot sabe partir el jar en capas, de lo que cambia menos a lo que cambia más: **dependencies**, **spring-boot-loader**, **snapshot-dependencies** y **application** (Spring Boot Reference, 4.1). Un **Dockerfile** en dos etapas las extrae y las copia en ese orden. Este no lo construí: sigue el ejemplo oficial de Boot 4.1, con la imagen de JRE 21 que usa este sitio:

```
# ilustrativo: sigue el ejemplo oficial de Spring Boot 4.1
FROM eclipse-temurin:21-jre AS capas
WORKDIR /app
COPY target/*.jar application.jar
RUN java -Djarmode=tools -jar application.jar extract --layers --destination extraido

FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=capas /app/extraido/dependencies/ ./
COPY --from=capas /app/extraido/spring-boot-loader/ ./
COPY --from=capas /app/extraido/snapshot-dependencies/ ./
COPY --from=capas /app/extraido/application/ ./
ENTRYPOINT ["java", "-jar", "application.jar"]
```

La alternativa es no escribir el **Dockerfile**: **`./mvnw spring-boot:build-image`** arma la imagen con **buildpacks** de Paketo, que eligen el JRE y arman la imagen por ti; solo necesita un daemon de Docker (Spring Boot Maven Plugin, 4.1). Ganas una imagen correcta sin mantener nada; pierdes control fino sobre la imagen base.

La JVM dentro de un contenedor

En Linux, la JVM lee la memoria y las CPU del contenedor, no las de la máquina (documentación del comando **java**, JDK 21). Lo que sorprende es el heap: por defecto, el 25 % de lo disponible (**-XX:MaxRAMPercentage**). En un contenedor de 1 GB, 256 MB, y el resto casi sin usar. Por eso es común subir **-XX:MaxRAMPercentage**, dejando lugar para lo que la JVM usa fuera del heap (hilos, código compilado, buffers); el valor sale de medir la memoria de tu app con Actuator.

Perfiles por ambiente

El perfil de producción guarda lo que solo cambia ahí:

application-produccion.yml

```
server:
  shutdown: graceful
spring:
  lifecycle:
```

```
    timeout-per-shutdown-phase: 20s
management:
  server:
    port: 8081
  endpoint:
    health:
      probes:
        add-additional-paths: true
logging:
  structured:
    format:
      console: logstash
```

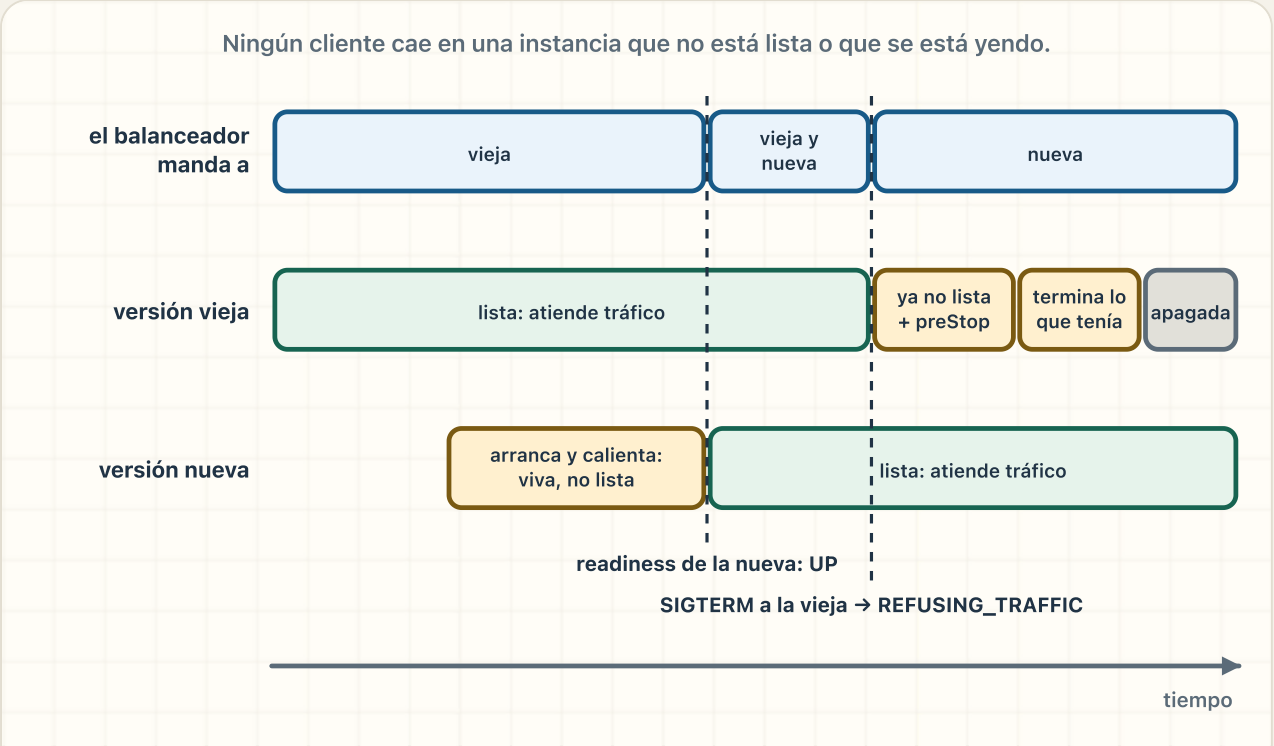
`management.server.port` saca Actuator a otro puerto: el balanceador público solo conoce el 8080, y el 8081 queda para la red interna y el monitoreo. Las sondas, en cambio, conviene que respondan en el puerto de la app: en el de gestión pueden dar bien aunque la app principal no atienda. **`add-additional-paths`** las publica también en el 8080, como **`/livez`** y **`/readyz`**, y ahí las consulta el orquestador (Spring Boot Reference, 4.1). Un test levanta este perfil con servidores de verdad y lo comprueba.

Apagado ordenado

Si la app se cierra apenas recibe la señal de apagado, las peticiones a medio atender reciben un error. Con **apagado ordenado** (*graceful shutdown*), Tomcat deja de aceptar peticiones nuevas y espera a que terminen las que están en curso, hasta un tiempo máximo.

Es el comportamiento por defecto desde Boot 3.4: **`server.shutdown`** vale **`graceful`**, con 30 s por fase en **`spring.lifecycle.timeout-per-shutdown-phase`** (Spring Boot Reference, 4.1); lo comprobé con un test. El perfil de arriba lo escribe igual, para que se lea, y baja la espera a 20 s.

Durante el apagado, readiness pasa a **`REFUSING_TRAFFIC`**, pero el balanceador tarda unos segundos en enterarse y sigue mandando peticiones. Por eso la guía de despliegue de Boot recomienda una pausa antes del apagado, un **`preStop`** en Kubernetes, y no confiar solo en el apagado ordenado (Spring Boot Reference, 4.1).



La nueva entra cuando está lista; la vieja sale cuando ya nadie le manda nada.

Migraciones que no cortan: expandir y contraer

Durante un despliegue sin cortes, **la versión vieja y la nueva corren a la vez** contra la misma base. Si la migración de la nueva cambia algo que la vieja usa, la vieja empieza a fallar. Para renombrar la columna **`cliente`** a **`nombre_cliente`**, lo directo es esto:

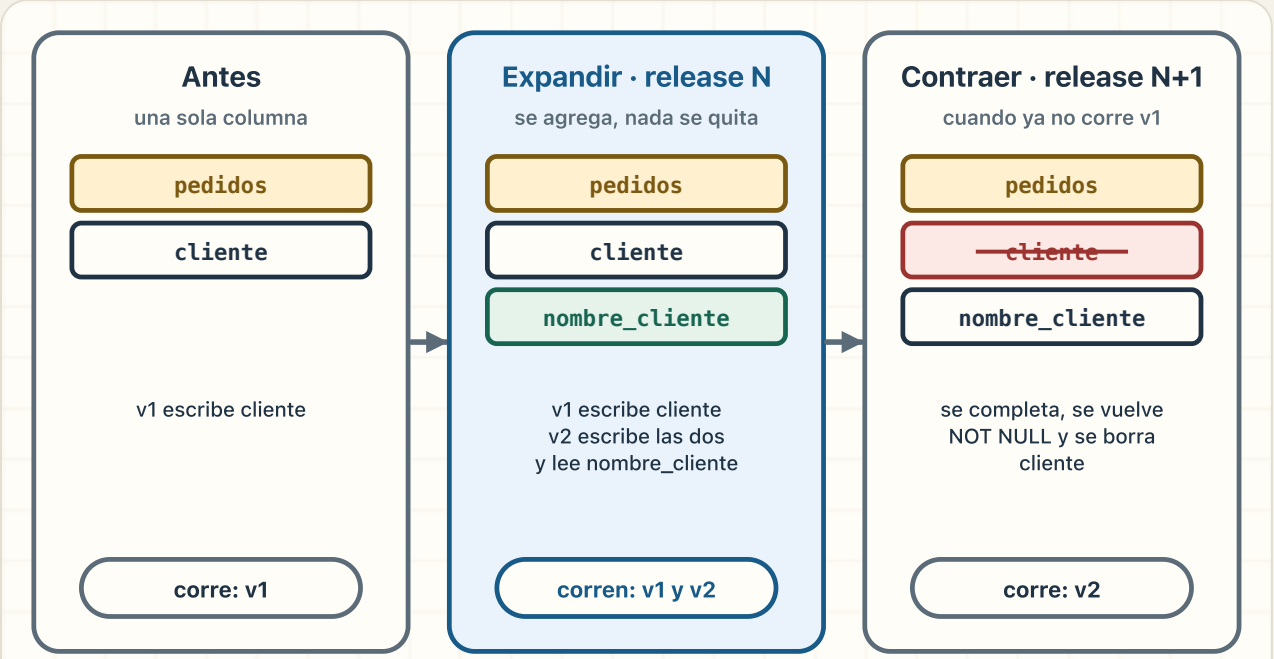
V2__renombrar_cliente.sql

```
ALTER TABLE pedidos RENAME COLUMN cliente TO nombre_cliente;
```

Apenas corre, cada **`INSERT`** de la versión vieja, que todavía atiende tráfico, falla porque la columna **`cliente`** no existe. La alternativa es partir el cambio en dos despliegues: **expandir** y **contraer**.

V2__expandir_nombre_cliente.sql

```
-- Release N: se agrega la columna nueva; la vieja sigue ahí.
ALTER TABLE pedidos ADD COLUMN nombre_cliente VARCHAR(120);
UPDATE pedidos SET nombre_cliente = cliente;
```



Primero se agrega sin quitar nada; se quita recién cuando nadie usa lo viejo.

En la release N, la migración solo **agrega**. La versión nueva escribe en las dos columnas y lee la nueva; la vieja sigue con la suya sin enterarse. En la release N+1, cuando ya no corre ninguna instancia vieja, se **contrae**:

V3__contraer_cliente.sql

```
-- Release N+1, cuando ya no corre ninguna instancia de la versión vieja.
UPDATE pedidos SET nombre_cliente = cliente WHERE nombre_cliente IS NULL;
ALTER TABLE pedidos ALTER COLUMN nombre_cliente SET NOT NULL;
ALTER TABLE pedidos DROP COLUMN cliente;
```

El primer **`UPDATE`** rescata las filas que la versión vieja escribió durante la convivencia. Probé las tres situaciones con tests: renombrar de golpe rompe a la versión vieja, con la expansión conviven las dos, y contraer antes de tiempo también la rompe.

> **Nota:** En este sitio, el despliegue a producción no reconstruye nada: toma el jar que el CI ya probó para ese commit, lo envuelve en una imagen mínima (**`eclipse-temurin:21-jre`**, que solo copia el jar y lo corre con **`java -jar`**, sin capas), lo sube a Railway y espera a que producción sirva ese commit. Lo que corre en producción tiene que ser exactamente lo que pasó los tests.

> **Lo que debes llevarte:** el apagado ordenado ya viene activado; súmale una pausa antes del apagado. Y en un despliegue sin cortes, las migraciones **expanden** primero y **contraen** en la release siguiente.

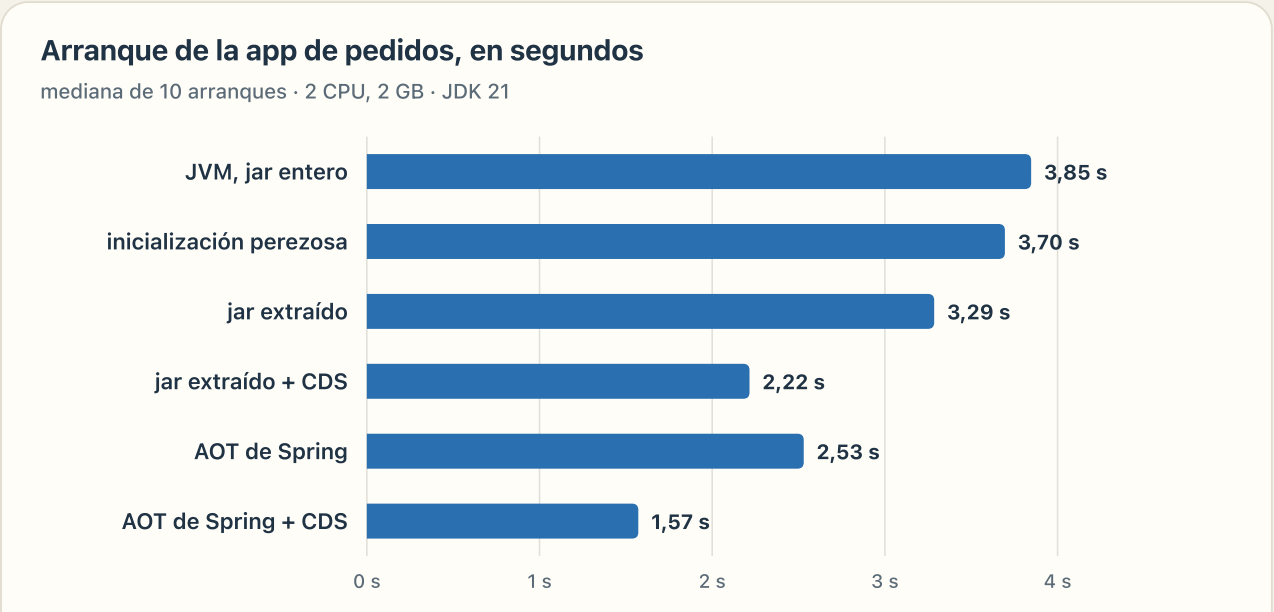
Arranque y memoria

Unos segundos de arranque casi siempre dan igual. Importan cuando escalas por demanda y cada instancia nueva tiene que atender cuanto antes, o cuando pagas por segundo de cómputo. Hay cuatro técnicas, y cada una cobra algo distinto:

- **Inicialización perezosa** (`spring.main.lazy-initialization=true`): los beans se crean cuando alguien los pide, no al arrancar.
- **CDS** (*Class Data Sharing*): una corrida de entrenamiento guarda en un archivo las clases ya cargadas y verificadas, y las siguientes arrancan desde ese archivo. Es una función de la JVM, no de Spring.
- **AOT de Spring** (*ahead-of-time*): en el build, Spring analiza tu configuración y genera código Java que registra los beans directamente, sin escanear ni evaluar condiciones al arrancar.
- **Imagen nativa** con GraalVM: compila la app entera a un ejecutable del sistema operativo, sin JVM.

Lo que medí

Medí las tres primeras en la app de esta parte. Cada variante arrancó 10 veces en un contenedor con JDK 21 (Eclipse Temurin), 2 CPU y 2 GB, contra un Postgres en otro contenedor, en una MacBook Pro con Apple M3 Pro. El valor es el "process running for" que escribe Boot al terminar de arrancar: desde que arranca la JVM hasta que la app está lista, con Flyway, Hibernate y Tomcat. Muestro la mediana.



Mediana de 10 arranques por variante, en un contenedor con 2 CPU y 2 GB. Medido en esta app, no en la tuya: repite la medición antes de decidir.

Variante	Mediana	Mínimo	Máximo
JVM, jar entero (`java -jar`)	3,85 s	3,81 s	4,42 s
Inicialización perezosa	3,70 s	3,58 s	3,78 s
Jar extraído	3,29 s	3,23 s	3,39 s
Jar extraído + CDS	2,22 s	2,19 s	2,70 s
AOT de Spring	2,53 s	2,47 s	3,00 s
AOT de Spring + CDS	1,57 s	1,51 s	2,15 s

Tres cosas para leer en esos números:

- **La perezosa casi no mueve la aguja** en esta app: de 3,85 a 3,70 s. Casi todo el arranque es Flyway, Hibernate y Tomcat, que se preparan igual. Ayuda más en apps con muchos beans que nadie usa al arrancar.
- **Extraer el jar y sumar CDS** baja a 2,22 s, un 42 % menos, sin tocar el código: solo cambia cómo se arma la imagen y cómo se lanza el proceso.
- **AOT de Spring y CDS se suman**, porque atacan cosas distintas: AOT ahorra el trabajo de Spring al decidir qué beans crear; CDS, el de la JVM al cargar clases. Juntos, 1,57 s, un 59 % menos. El equipo de Spring midió lo mismo con su app de ejemplo: AOT más CDS, cerca del doble de rápido (blog de Spring, 2024).

La imagen nativa no está en el gráfico. Spring Boot 4.1 la compila con GraalVM 25 o posterior (Spring Boot Reference, 4.1). Lo intenté con GraalVM CE 25.0.2 en un contenedor con 6 CPU y 7 GB, lo más que deja Docker en esta máquina, y la compilación se quedó sin memoria. Eso dice algo del costo de la técnica, pero no es un tiempo de arranque, y no muestro un número que no medí.

Lo que pagas por cada una

- **Perezosa**: un error de configuración ya no aparece al arrancar, sino cuando alguien usa el bean roto, quizá a las tres de la mañana (Spring Boot Reference, 4.1). Lo comprobé con un test:

InicializacionPerezosaTest.java

```
@Test
void sinPerezaElErrorApareceAlArrancar() {
    assertThatThrownBy(() ->
app(false).run()).isInstanceOf(BeanCreationException.class);
}

@Test
void conPerezaArrancaYElErrorEsperaAlPrimerUso() {
    try (var contexto = app(true).run()) {
        assertThat(contexto.isRunning()).isTrue();
```

```
        assertThatThrownBy(() ->
contexto.getBean(ClienteDePagos.class)).isInstanceOf(BeanCreationException.class);
    }
}
```

- **CDS**: hay que hacer una corrida de entrenamiento en el build y usar el jar extraído; el archivo no sirve con el jar entero (Spring Boot Reference, 4.1). Con Java 25 o posterior, Boot soporta además el *AOT cache* de la JVM, el sucesor de CDS; con Java 21, como esta app, lo que hay es CDS.
- **AOT de Spring**: el classpath y los beans quedan fijos en el build. Un **`@Profile`** o un **`@ConditionalOnProperty`** se evalúan al compilar, no al arrancar, así que no puedes activar un perfil que cambie qué beans existen al desplegar (Spring Boot Reference, 4.1).
- **Imagen nativa**: lo mismo que AOT y más. GraalVM asume un mundo cerrado: todo lo dinámico (reflexión, proxies, recursos) tiene que declararse en el build (Spring Boot Reference, 4.1), y, como viste, el build pide mucha memoria. Pagas en flexibilidad y en tiempo de build.

> **Lo que debes llevarte:** mide antes de elegir. CDS es la ganancia más barata si ya construyes una imagen, y ninguna técnica arregla una app que hace demasiado al arrancar.

Tu propia autoconfiguración

En la Parte 2 viste que la autoconfiguración de Boot es una lista de clases **`@Configuration`** con condiciones. Escribir una es la prueba de que no hay magia.

El caso: varios servicios de la empresa quieren registrar acciones de auditoría. En vez de copiar la clase en cada uno, armas un módulo que, con solo estar en el classpath, le da a cada app un **`Auditor`** listo, y que se hace a un lado si la app ya tiene el suyo. Primero, sus propiedades:

AuditoriaProperties.java

```
@ConfigurationProperties("auditoria")
public record AuditoriaProperties(@DefaultValue("[auditoria]") String prefijo) {}
```

Después, la autoconfiguración:

AuditoriaAutoConfiguration.java

```
@AutoConfiguration
@ConditionalOnBooleanProperty(name = "auditoria.habilitada", matchIfMissing = true)
@EnableConfigurationProperties(AuditoriaProperties.class)
public class AuditoriaAutoConfiguration {

    @Bean
```

```
    @ConditionalOnMissingBean
    Auditor auditor(AuditoriaProperties propiedades) {
        return new AuditorEnLog(propiedades.prefijo());
    }
}
```

- **`@AutoConfiguration`** es una **`@Configuration`** que Boot carga como autoconfiguración, después de la configuración de tu app.
- **`@ConditionalOnBooleanProperty`** la apaga entera con **`auditoria.habilitada=false`**.
- **`@ConditionalOnMissingBean`** es la cortesía de toda autoconfiguración: si la app ya definió un **`Auditor`**, este no se crea.

La pieza que suele faltar: Boot no escanea los paquetes de tus dependencias. Una autoconfiguración se registra en un archivo de texto con un nombre exacto, **`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`**, una clase por línea (Spring Boot Reference, 4.1):

AutoConfiguration.imports

```
com.ejemplo.auditoria.AuditoriaAutoConfiguration
```

En mi ejemplo, la autoconfiguración vive en **`com.ejemplo.auditoria`**, **fuera** del paquete de la app (**`com.ejemplo.pedidos`**), así que el escaneo de componentes no la ve: si el **`Auditor`** aparece en el contexto, es porque Boot leyó el archivo de imports, y un test lo comprueba. En un proyecto real, ese paquete sería un módulo Maven aparte, el que se publica como biblioteca.

Probarla sin levantar la app

`ApplicationContextRunner` arma un contexto mínimo con la autoconfiguración que le pidas, en milisegundos, y prueba cada condición:

AuditoriaAutoConfigurationTest.java

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

@Test
void sinNadaConfiguradoTraeUnAuditorEnElLog() {
    contexto.run(app -> assertThat(app)
            .getBean(Auditor.class)
            .isInstanceOfSatisfying(AuditorEnLog.class, a ->
assertThat(a.prefijo()).isEqualTo("[auditoria]")));
}

@Test
void elPrefijoSeCambiaConUnaPropiedad() {
```

```
    contexto.withPropertyValues("auditoria.prefijo=[tienda]")
            .run(app ->
assertThat(app.getBean(AuditorEnLog.class).prefijo()).isEqualTo("[tienda]"));
}

@Test
void siLaAppDefineSuAuditorLaAutoconfiguracionSeHaceAUnLado() {
    contexto.withUserConfiguration(AuditorPropio.class)
            .run(app ->
assertThat(app).hasSingleBean(Auditor.class).getBean(Auditor.class).isSameAs(AuditorProp
io.UNICO));
}

@Test
void conLaPropiedadEnFalseNoHayAuditor() {
    contexto.withPropertyValues("auditoria.habilitada=false")
            .run(app -> assertThat(app).doesNotHaveBean(Auditor.class));
}
```

Esto es todo lo que hay detrás de "Boot lo configura solo". Cada starter de la serie es lo mismo, con más condiciones: **`DataSourceAutoConfiguration`**, **`SecurityAutoConfiguration`**, la de Caffeine.

> **El error típico:** dejar la autoconfiguración dentro del paquete que la app escanea. El escaneo la carga como configuración normal, y **`@ConditionalOnMissingBean`** puede evaluarse antes de que existan los beans que debería respetar. Una autoconfiguración se carga solo desde el archivo de imports y nunca debe quedar al alcance del escaneo (Spring Boot Reference, 4.1).

> – Para profundizar
>
> el nombre de un starter propio no debe empezar con **`spring-boot`**, aunque uses otro **`groupId`**. El ejemplo de la referencia es **`acme-spring-boot`** para el módulo de autoconfiguración y **`acme-spring-boot-starter`** para el starter (Spring Boot Reference, 4.1). El orden entre autoconfiguraciones se controla con **`@AutoConfiguration(before = …, after = …)`**.

Arquitectura: paquetes y módulos

Las clases de una app se reparten en paquetes **por capa** (**`controllers`**, **`services`**, **`repositories`**) o **por funcionalidad** (**`pedidos`**, **`catalogo`**, **`notificaciones`**, cada uno con su controlador, servicio y repositorio). Por capa se ve ordenado al principio, pero todo termina **`public`** y nada impide que notificaciones use el

repositorio de pedidos. Por funcionalidad, lo de pedidos vive junto, y lo que no hace falta mostrar queda con visibilidad de paquete. Así está organizada la tienda.

Pero Java no tiene forma de decir "este paquete es privado de este módulo". Eso lo agrega **Spring Modulith**: trata cada subpaquete directo del paquete principal como un **módulo**, y los subpaquetes de un módulo como internos, que los otros módulos no pueden usar (Spring Modulith Reference, 2.1). La línea 2.1 es la que corresponde a Spring Boot 4.1 (spring.io).

pom.xml

```
<dependency>
    <groupId>org.springframework.modulith</groupId>
    <artifactId>spring-modulith-bom</artifactId>
    <version>${spring-modulith.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

pom.xml

```
<dependency>
    <groupId>org.springframework.modulith</groupId>
    <artifactId>spring-modulith-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Con eso, un test verifica los límites:

ModulosTest.java

```
@Test
void losModulosDeLaTiendaRespetanSusLimites() {
    ApplicationModules.of(PedidosApplication.class).verify();
}
```

`verify()` falla si hay un ciclo entre módulos o si un módulo usa algo interno de otro. En la tienda, **`notificaciones`** depende de **`pedidos`** y no al revés: el evento **`PedidoPagado`** es lo que permite que la dependencia vaya en un solo sentido. El antes es una tienda enredada, donde cada módulo mete la mano en el otro:

ServicioDePedidos.java

```
public class ServicioDePedidos {

    public String confirmar(long id) {
        return new PlantillaDeCorreo().pagoRecibido(id); // entra a un paquete interno
de otro módulo
```

```
    }
}
```

AvisoDePedido.java

```
public class AvisoDePedido {

    private final ServicioDePedidos pedidos = new ServicioDePedidos(); // y
notificaciones depende de pedidos

    public String avisar(long id) {
        return pedidos.confirmar(id);
    }
}
```

ModulosTest.java

```
@Test
void unaTiendaEnredadaNoPasaLaVerificacion() {
    var violaciones = ApplicationModules.of(TiendaEnredada.class, new
ImportOption.OnlyIncludeTests())
            .detectViolations();

    assertThat(violaciones.getMessage())
            .contains("Cycle detected")
            .contains("non-exposed type
com.ejemplo.enredado.notificaciones.internal.PlantillaDeCorreo");
}
```

La verificación encuentra las dos cosas:

```
- Cycle detected: Slice notificaciones ->
                Slice pedidos ->
                Slice notificaciones
- Module 'pedidos' depends on non-exposed type
com.ejemplo.enredado.notificaciones.internal.PlantillaDeCorreo within module
'notificaciones'!
```

> **Lo que debes llevarte:** organiza por funcionalidad y deja que un test cuide los límites: un límite que solo está en la cabeza del equipo dura hasta la primera entrega con prisa.

> **– Para profundizar**

> un monolito modular bien verificado es el paso previo más barato a separar servicios: si algún día un módulo tiene que irse, sus dependencias ya están a la vista y van en una sola dirección. Modulith también puede generar la documentación de los módulos y probar uno solo, con **`@ApplicationModuleTest`**.

Spring Boot 4 hoy y cómo migrar

Spring Boot 4.0 salió el 20 de noviembre de 2025, y la 4.1 el 10 de junio de 2026 (blog de Spring; releases en GitHub). Frente a 3.x, lo que más se nota, según la guía de migración y el anuncio:

- **Modularización.** Módulos chicos por tecnología, con starters a juego: **`spring-boot-starter-webmvc`** en vez de **`spring-boot-starter-web`**, que queda deprecado; **`spring-boot-starter-flyway`**; y starters de test como **`spring-boot-starter-webmvc-test`**.
- **JSpecify.** Las APIs de Spring declaran qué puede ser **`null`**, y tu IDE puede avisarte.
- **Versionado de API.** Spring Framework 7 sabe resolver la versión de una API desde una cabecera, un parámetro o la ruta, y Boot lo configura con **`spring.mvc.apiversion.*`**.
- **Jackson 3.** Es la biblioteca JSON por defecto, y sus paquetes pasaron de **`com.fasterxml.jackson`** a **`tools.jackson`** (salvo las anotaciones). Jackson 2 sigue soportado, deprecado.
- **La base:** Java 17 como mínimo (la 4.1.2 es compatible hasta Java 27), Jakarta EE 11 y Servlet 6.1. Undertow ya no está soportado.

Cuánto dura el soporte de cada versión

Al 4 de noviembre de 2026, según spring.io:

Línea	Salió	Soporte gratuito hasta	Soporte comercial hasta
3.5.x	2025-05	2026-06 (ya terminó)	2032-06
4.0.x	2025-11	2026-12	2027-12
4.1.x	2026-06	2027-07	2028-07

La política de Spring da al menos 13 meses de soporte gratuito a cada versión menor, y la última menor de cada mayor, como la 3.5, recibe cinco años adicionales de soporte comercial (spring.io, política de soporte). Si tu app sigue en 3.5, ya no recibe parches gratuitos.

Cómo migrar

- **Primero, a la última 3.5.x**, como pide la guía de migración.

- **Una versión menor a la vez**: cada una puede quitar lo que se deprecó en las anteriores, y de a una cada error tiene un solo origen posible.
- **`spring-boot-properties-migrator`**. Agregado como dependencia de **`runtime`**, al arrancar te dice qué propiedades cambiaron de nombre y las traduce temporalmente. Cuando terminas, se quita (guía de migración a Boot 4.0).
- **OpenRewrite**, para automatizar cambios de código: la receta **`org.openrewrite.java.spring.boot4.UpgradeSpringBoot_4_0`** tiene licencia Moderne Source Available, que te permite correrla sobre tu propio código; la de 4.1 es de Moderne y propietaria. Revisa el diff como si lo hubiera escrito otra persona.

Cuándo no elegir Spring Boot

Spring hace en tiempo de ejecución gran parte de su trabajo: escanear, evaluar condiciones, crear proxies. La propia documentación de Boot dice que las aplicaciones típicas de Spring Boot son bastante dinámicas y se configuran en tiempo de ejecución (Spring Boot Reference, 4.1). Esa es la fuente de su flexibilidad, y de su costo de arranque.

Dos alternativas parten de la decisión contraria. **Quarkus** dice que su idea central es hacer en el build lo que los frameworks tradicionales hacen en tiempo de ejecución, y que su inyección de dependencias resuelve el grafo en el build, sin reflexión (quarkus.io). **Micronaut** se presenta con inyección de dependencias en tiempo de compilación, sin reflexión en tiempo de ejecución (micronaut.io). Si tu problema principal es el arranque o la memoria, por ejemplo en funciones serverless, vale la pena medirlas contra Boot con AOT o imagen nativa. Si es construir y mantener una app con mucho ecosistema alrededor, el de Spring es difícil de igualar. Lo honesto es medir.

Lo que solo enseña la experiencia

- **Un `@Cacheable`, un `@Async` o un `@PreAuthorize` que "no funcionan"** casi siempre son una llamada desde la misma clase: el proxy de la Parte 1, con otro sombrero.
- **La actualización perdida no da error**, y el correo de un pago que no existió tampoco. Los dos aparecen en producción, nunca en una demo con un solo usuario.
- **La tarea que corre tres veces** aparece el día que escalas a tres instancias: ShedLock, o una sola instancia dedicada a las tareas.
- **La base en el chequeo de liveness** convierte un problema de la base en una caída total.
- **La migración que renombra una columna** rompe a la versión vieja en pleno despliegue.

El cierre de la serie

Empezamos con un **`new`** que ataba una clase a su notificador, y terminamos con una app que se autentica con tokens firmados, resiste escrituras concurrentes, corre en varias instancias y se despliega sin cortes.

Me quedo con tres ideas:

- **Core antes que Boot.** Cada trampa de esta parte es un proxy, una transacción o un bean que se crea o no se crea. Quien entiende el contenedor de la Parte 1 diagnostica en minutos lo que a otros les lleva días.
- **Lee el código de la autoconfiguración.** Ya escribiste una. Cuando Boot haga algo que no esperabas, abre la clase **`…AutoConfiguration`** que corresponde y lee sus condiciones: ahí está la respuesta.
- **No pelees con el framework.** Si te encuentras desactivando la mitad de una autoconfiguración para obligarla a hacer otra cosa, probablemente hay una propiedad o un bean que Boot espera que definas. Boot está hecho para hacerse a un lado: déjalo.

¿Cuál de las trampas de esta parte ya te encontraste en producción? Y la pregunta más útil: ¿cuál crees que está esperando en tu proyecto, y qué test escribirías hoy para comprobarlo?

Referencias

- Spring Boot Reference, 4.1.2: Spring Security, Caching, Task Execution and Scheduling, SpringApplication (hilos virtuales e inicialización perezosa), Actuator Endpoints, Logging, Tracing, Graceful Shutdown, Deploying to the Cloud, Dockerfiles, AOT Cache y CDS, AOT con la JVM, GraalVM Native Images, Creating Your Own Auto-configuration, System Requirements.
- Spring Boot Maven Plugin, 4.1.2: Packaging OCI Images.
- Spring Boot en GitHub: 4.0 Migration Guide, 4.0 Release Notes, 4.1 Release Notes.
- spring.io: Spring Boot 4.0.0 available now (2025), Spring Boot CDS support and Project Leyden anticipation (2024), Spring Boot, soporte por versión, Support Policy.
- Spring Security Reference, 7.1.2: Architecture, CSRF, CORS, OAuth 2.0 Resource Server JWT, Method Security.
- Spring Framework Reference, 7.0.10: Transaction-bound Events, Declarative Annotation-based Caching, Spring WebFlux, Overview.
- Spring Framework API, 7.0.10: **`ObjectOptimisticLockingFailureException`**, **`TransactionSynchronization`**.
- Spring Modulith Reference, 2.1: Fundamentals, Working with Application Events.
- PostgreSQL Documentation: How Connections Are Established.
- RFC 9110: HTTP Semantics (**`If-Match`**, 409 y 412).
- JDK 21: The java Command.
- ShedLock y HikariCP, sus README.
- OpenRewrite: la receta de Spring Boot 4.0 y su licencia.
- Quarkus: Container First y Micronaut, su propia presentación.