Spring Forward

Navigating the Breaking Changes in Spring Boot 4.0

Steve Poole

herodevs

SPRING-BOOT 3.5 → 4.0

Spring Boot 4.0 Migration Guide

Navigating the Breaking Changes in Spring Boot 4.0

Tier 1 · Won't Build	39 challenges
Tier 2 · Won't Run	27 challenges
Tier 3 · Wrong Results	17 challenges

Estimated total migration effort:
200–500 hours depending on codebase size

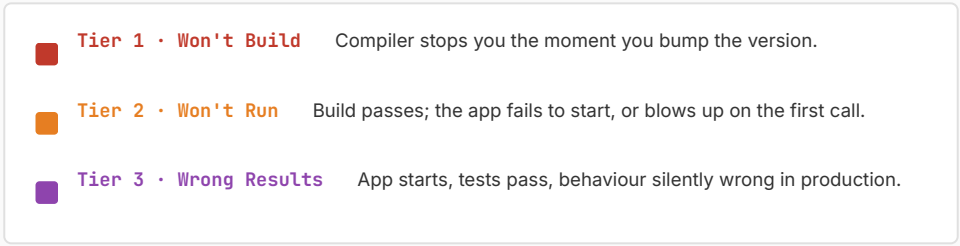
Foreword

Spring Boot 4.0 is not a gentle nudge. The Spring team made deliberately breaking choices — clearing out years of deprecated APIs, reorganising package namespaces, tightening the component model. Right decisions for the long run. Significant work in the short term.

This guide came out of asking one question of every breaking change: **how does this actually affect a team?** The answer organises into three tiers, by how loudly each change fails and how late it surfaces in your pipeline.

Tier 3 is the one to take seriously: a change that silently alters behaviour can sit undetected for months. Use this guide as a checklist, not a reading exercise. Start at Tier 1, work through systematically, and cross-reference against the official Spring Boot 4.0 migration notes.

The effort estimates are honest, not optimistic. A 200–500 hour range looks alarming; for a large, legacy codebase that hasn't kept up with intermediate releases, it probably is that much work. Better to know upfront than to discover it three weeks before a compliance deadline.



Why This Migration Matters

At Spring I/O 2026 we ran a 15-question quiz on Spring Boot 4.0 migration. About a hundred people played: developers interested enough to attend a Spring conference and curious enough to walk up to a booth.

The mean score was 66%.

The average player recognised roughly two-thirds of the breaking changes in 4.0. The other third was a surprise. Booth visitors are almost certainly better informed than the wider population of Spring developers, so the gap on a typical production team is probably larger.

The headline figure: roughly **4,900 hours** of unseen migration work across the cohort. Each wrong answer is a change the player would have walked into during a real migration. Per player, that's around **50 hours** of engineering effort nobody had on their roadmap.

If you're sizing this migration on what your team currently knows, you're probably underestimating it. For a small service that's a couple of weeks of unplanned work. For a sprawling enterprise codebase, it's the kind of slip that turns a comfortable schedule into a tight one.

The Spring team didn't write 4.0 to inconvenience anyone. They cleared a decade of deprecations, modernised the component model, and aligned the dependency tree with current Java. Right calls for the long term. The short-term cost falls on every team running 3.x today, and it falls unevenly: the simpler your codebase, the easier the upgrade.

This guide makes the unseen work visible before it's a problem. Each card describes one breaking change, how to detect it, and how to fix it. The tier system tells you how loudly each change fails. The Migration Playbook tells you how to sequence the work. Use them together.

About the Author

SP

Steve Poole

Developer Advocate · HeroDevs

JAVA CHAMPION ORACLE ACE IBM CHAMPION

Steve Poole has been working in and around Java since the late 1990s, spending many years at IBM before moving into developer advocacy and open-source consulting. He's a Java Champion, Oracle ACE and IBM Champion.

A regular on the conference circuit — JavaOne, Devoxx, JAX, and others — and someone who has spent a lot of time thinking about the things developers tend not to think about until they have to.

At **HeroDevs**, Steve focuses on open-source security, software supply chain risk, and the practical realities of keeping enterprise Java systems running safely after their official support windows close.

His current preoccupation is the gap between **"still running in production"** and **"still receiving security patches"** — a gap that affects more organisations than anyone in the ecosystem likes to admit.

Contents

REFERENCE PAGES

Tier 1 — Won't Build

39 cards

- Maven Plugin Version Alignment for AOT
- OkHttp3 Support Removed
- PropertyMapper.alwaysApplyingWhenNonNull() Removed
- @PropertyMapping Relocated to test.context
- Security DSL Rewrite
- SimpDestinationMessageMatcher Removed
- Spring AMQP RabbitRetryTemplateCustomizer Removed
- spring-jcl Removed
- spring-retry Removed from BOM
- Spring Security Access API Moved to spring-security-access
- Testcontainers 2.0 Package Relocation
- TestRestTemplate Package Relocated
- Tomcat WAR Deployment Requires New Runtime Starter
- Undertow Embedded Server Removed
- webjars-locator-core Removed from BOM

Tier 2 — Won't Run

27 cards

Tier 3 — Wrong Results

17 cards

- spring.jackson.default-property-inclusion Silently Ignored
- Liveness and Readiness Probes Enabled by Default
- Logback Default Charset Changed to UTF-8
- MongoDB Configuration Properties Renamed
- MongoDB UUID and BigDecimal Representations No Longer Defaulted
- @NullMarked Default — IDE/Compiler Null Safety
- Controller/View Spans Silently Disabled
- Path Matching Engine: AntPathMatcher → PathPattern
- maxAttempts Now Means Retries, Not Total Attempts
- @Retryable + @Transactional AOP Ordering Change
- Spring Session Property Prefixes Renamed

Migration Impact Estimates

How we sized the 83 breaking changes — and what the numbers mean.

T-SHIRT SIZES

Each card carries an impact badge — **S**, **M**, or **L** — representing the typical engineering impact per card, for a medium-sized codebase (~50k LOC).

SMALL

2–6 h

Config change, dependency swap, or one-line fix. Low risk of follow-on work.

48 cards

MEDIUM

6–16 h

API migration, code rewrite, or multi-file refactor. May need testing and validation.

30 cards

LARGE

16–40 h

Major subsystem migration. Touches many files, requires security review or arch decisions.

5 cards

HOW WE CALCULATED

Estimates are based on the HeroDevs Spring Boot migration research, validated against the **Spring I/O 2026 quiz** (88 developers, mean score 66%), production migration audits, and per-subsystem breakdowns:

SUBSYSTEM	TYPICAL HOURS	KEY DRIVER
Jackson 3.0	16–40 h	Group ID + class renames across every file that imports Jackson
Spring Security 7	8–16 h	DSL rewrite, OAuth removal, default-deny policy
Hibernate 7	4–8 h	Dialect removal, Session API changes, date type shifts
Testing (JUnit 6+)	16–32 h	@MockBean removal, test slice relocation, Testcontainers package
Build tooling	8–32 h	Java 17 minimum, Gradle 8.x, Maven AOT plugin
Observability	4–8 h	Controller spans disabled, OTLP endpoint flip

Total for a medium app: **120–230+ hours** (weeks, not days). Minor version bumps (3.2→3.3, 3.3→3.4) averaged 6–8 hours each. The 4.0 jump is 15–30× that.

OPENREWRITE COVERAGE

18 of 83 cards (22%) have an OpenRewrite recipe that can automate all or most of the fix. Cards with a recipe are marked ⚙ — run the recipe first, then use the card to verify and handle anything it missed.

Tier 1: 9 of 39 (23%) — best coverage, mostly dependency/import swaps

Tier 2: 8 of 27 (30%) — good coverage for config and security rewrites

Tier 3: 1 of 17 (6%) — behavioural changes are hardest to automate

Key insight: OpenRewrite handles the loud failures well (Tiers 1–2), but Tier 3 "wrong results" bugs — the ones customers find — are almost entirely manual detective work.

SPRING-BREAK DEMO MODULES

Each card's Further Info section references a **spring-break module** by name (e.g. `spring-break module: jackson-group-id`). These are runnable before/after demo projects that reproduce the exact failure described on the card. Find them at github.com/spoole167/spring-break. Clone the repo, `cd` into the named module, and run `mvn verify` to see the error — then apply the fix and confirm it passes. Cards without a spring-break module are noted in their Further Info section.

TIER 1

Won't Build

Compilation & Dependency Failures

Your code won't compile. These are import, dependency, and API changes that produce immediate build failures.

CARD INDEX · 39 CARDS

AOP Starter Renamed to spring-boot-starter-aspectj

spring-boot-starter-aop is no longer in the Spring Boot 4.0 BOM. Replace it with spring-boot-starter-aspectj.

WHAT YOU'LL SEE

```
$ mvn validate
[ERROR] 'dependencies.dependency.version' for
  org.springframework.boot:spring-boot-starter-aop:jar is missing.
```

WHAT CHANGED

`spring-boot-starter-aop` has been removed from the Spring Boot BOM. The replacement starter is `spring-boot-starter-aspectj`. The new starter provides the same AspectJ weaving capabilities under a name that better describes what it does.

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

HOW TO FIX

1. **Replace the starter in pom.xml / build.gradle.** Find `spring-boot-starter-aop` in all build files and replace with `spring-boot-starter-aspectj`. No code changes are needed — the classpath content is equivalent.

✓ Check:
mvn validate — no version is missing error for the AOP starter

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

ApacheDS Embedded LDAP Support Removed

ApacheDSContainer is removed in Spring Security 7.0. Use UnboundIdContainer instead.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/ApacheDsUsage.java:[3,57]
  error: package org.springframework.security.ldap.server does not contain
  ApacheDSContainer
```

WHAT CHANGED

`org.springframework.security.ldap.server.ApacheDSContainer` has been removed from Spring Security. The embedded LDAP server capability is now provided exclusively through `org.springframework.security.ldap.server.UnboundIdContainer`.

```
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-server-jndi</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.unboundid</groupId>
    <artifactId>unboundid-ldapsdk</artifactId>
    <scope>test</scope>
</dependency>
```

```
import org.springframework.security.ldap.server.ApacheDSContainer;
// ...
new ApacheDSContainer("dc=springframework,dc=org", "classpath:users.ldif");
import org.springframework.security.ldap.server.UnboundIdContainer;
// ...
new UnboundIdContainer("dc=springframework,dc=org", "classpath:users.ldif");
```

HOW TO FIX

1. **Replace ApacheDSContainer with UnboundIdContainer.** Swap the dependency in your build file and update the import and instantiation. The constructor signature is the same — base DN and LDIF classpath location. The LDIF format is also compatible.
2. **Use Spring Boot's embedded LDAP auto-configuration.** If you're only using embedded LDAP for tests, add `spring-boot-starter-ldap` and set `spring.ldap.embedded.base-dn` in test properties. Spring Boot will auto-configure UnboundID for you.

✓ Check:
mvn compile — no cannot find symbol for ApacheDSContainer

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Spring Batch ChunkHandler Renamed; setJobLauncher Removed

ChunkHandler renamed to ChunkRequestHandler. JobStep.setJobLauncher replaced by setJobOperator. Both fail to compile on Boot 4.0.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/BatchRenamingUsage.java:[4,65]
  error: cannot find symbol
    symbol: class ChunkHandler
    location: package org.springframework.batch.integration.chunk
[ERROR] /src/main/java/com/example/BatchRenamingUsage.java:[18,19]
  error: cannot find symbol
    symbol:   method setJobLauncher(JobLauncher)
```

WHAT CHANGED

`org.springframework.batch.integration.chunk.ChunkHandler` has been renamed to `ChunkRequestHandler`. `JobStep.setJobLauncher(JobLauncher)` has been replaced by `JobStep.setJobOperator(JobOperator)`.

```
~~import org.springframework.batch.integration.chunk.ChunkHandler;~~
~~// ...~~
~~ChunkHandler handler = ...;~~
import org.springframework.batch.integration.chunk.ChunkRequestHandler;
// ...
ChunkRequestHandler handler = ...;
```

```
~~jobStep.setJobLauncher(jobLauncher);~~
jobStep.setJobOperator(jobOperator);
```

HOW TO FIX

1. **Rename ChunkHandler to ChunkRequestHandler.** Update the import and all references. The interface contract is the same — only the name changed.
2. **Replace setJobLauncher with setJobOperator.** Inject a `JobOperator` bean instead of `JobLauncher` and pass it to `setJobOperator()`. `JobOperator` is available as a Spring-managed bean when Spring Batch autoconfiguration is active.

> **✓ Check:**
> mvn compile — no cannot find symbol for ChunkHandler or setJobLauncher

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Spring Batch JobBuilder(String) Constructor Removed

JobBuilder(String) is removed. JobRepository is now mandatory at construction: new JobBuilder("name", jobRepository).

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/JobBuilderUsage.java:[9,33]
  error: no suitable constructor found for JobBuilder(java.lang.String)
```

WHAT CHANGED

`new JobBuilder(String name)` and the subsequent `.repository(JobRepository)` method have been removed. `JobRepository` is now mandatory at construction time: `new JobBuilder(String name, JobRepository repository)`. The same applies to `StepBuilder` and other builder classes.

```
new JobBuilder("myJob")
    .repository(jobRepository)
    .start(step)
    .build();
```

```
new JobBuilder("myJob", jobRepository)
    .start(step)
    .build();
```

```
@Bean
public Job myJob(Step step) {
    return new JobBuilder("myJob").start(step).build();
}
@Bean
public Job myJob(JobRepository jobRepository, Step step) {
    return new JobBuilder("myJob", jobRepository).start(step).build();
}
```

HOW TO FIX

1. **Inject JobRepository and pass it to the constructor.** Add `JobRepository` as a parameter to your `@Bean` method (Spring will inject it) and pass it as the second argument to `new JobBuilder(...)`. The same change is needed for `StepBuilder` and any other builder that previously accepted only a name.

✓ Check:
mvn compile — no cannot find symbol for JobBuilder(String) constructor

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

Batch Listener Base Classes Removed

Spring Batch 6.0 removed the abstract listener base classes like JobExecutionListenerSupport and StepExecutionListenerSupport.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/batch/JobCompletionListener.java:[4,52]
  error: cannot find symbol
    symbol:   class JobExecutionListenerSupport
    location: package org.springframework.batch.core.listener
[ERROR] /src/main/java/com/example/batch/StepLoggingListener.java:[4,52]
  error: cannot find symbol
    symbol:   class StepExecutionListenerSupport
```

WHAT CHANGED

Spring Batch 6.0 deleted the `*ListenerSupport` abstract classes: `JobExecutionListenerSupport`, `StepExecutionListenerSupport`, `ChunkListenerSupport`, `ItemReadListenerSupport`, and others. Listeners must now implement the interfaces directly, which have had default methods since Batch 5.0.

```
~~import org.springframework.batch.core.listener.JobExecutionListenerSupport;~~

~~public class JobCompletionListener extends JobExecutionListenerSupport {~~
~~    @Override~~
~~    public void afterJob(JobExecution jobExecution) {~~
~~        log.info("Job completed: {}", jobExecution.getStatus());~~
~~    }~~
~~}~~
import org.springframework.batch.core.JobExecutionListener;

public class JobCompletionListener implements JobExecutionListener {
    @Override
    public void afterJob(JobExecution jobExecution) {
        log.info("Job completed: {}", jobExecution.getStatus());
    }
}
```

```
~~import org.springframework.batch.core.listener.StepExecutionListenerSupport;~~

~~public class StepLogger extends StepExecutionListenerSupport {~~
import org.springframework.batch.core.StepExecutionListener;

public class StepLogger implements StepExecutionListener {
```

HOW TO FIX

1. **Change extends to implements.** Replace `extends JobExecutionListenerSupport` with `implements JobExecutionListener`. Same for all other `*ListenerSupport` classes. The interfaces have default methods, so you only need to override the callbacks you use.
2. **Use @BeforeJob / @AfterJob annotations.** Spring Batch also supports annotation-based listeners. Annotate methods with `@BeforeJob`, `@AfterJob`, `@BeforeStep`, `@AfterStep` etc. on any POJO — no interface needed.

✓ **Check:**

mvn compile and Batch job completes with listener callbacks firing

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

Spring Batch Core Package Relocations

Core Spring Batch domain classes (Job, JobExecution, etc.) moved from org.springframework.batch.core to org.springframework.batch.core.job.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/BatchCoreUsage.java:[3,44]
  error: package org.springframework.batch.core does not contain class Job
[ERROR] /src/main/java/com/example/BatchCoreUsage.java:[4,44]
  error: package org.springframework.batch.core does not contain class JobExecution
```

WHAT CHANGED

Core domain classes have moved from `org.springframework.batch.core` to subpackages, primarily `org.springframework.batch.core.job`. Affected classes include `Job`, `JobExecution`, `JobInstance`, and `JobParameters` among others.

```
~~import org.springframework.batch.core.Job;~~
~~import org.springframework.batch.core.JobExecution;~~
~~import org.springframework.batch.core.JobInstance;~~
~~import org.springframework.batch.core.JobParameters;~~
import org.springframework.batch.core.job.Job;
import org.springframework.batch.core.job.JobExecution;
import org.springframework.batch.core.job.JobInstance;
import org.springframework.batch.core.job.JobParameters;
```

HOW TO FIX

1. **Update imports to include the job subpackage.** Add `.job` to the package path for the affected classes. This is a mechanical find-and-replace: `org.springframework.batch.core.Job` becomes `org.springframework.batch.core.job.Job`, and so on. Check the Spring Batch 6.0 Migration Guide for the complete list of moved classes.

✓ Check:
mvn compile — no package does not exist errors for org.springframework.batch.core classes

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

BootstrapRegistry and EnvironmentPostProcessor Relocated

BootstrapRegistry moved to org.springframework.boot.bootstrap and EnvironmentPostProcessor moved to org.springframework.boot. Both old import paths are gone.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/BootstrapRegistryUsage.java:[3,47]
  error: package org.springframework.boot does not contain BootstrapRegistry
[ERROR] /src/main/java/com/example/EnvironmentPostProcessorUsage.java:[3,43]
  error: package org.springframework.boot.env does not contain EnvironmentPostProcessor
```

WHAT CHANGED

`BootstrapRegistry` and `ConfigurableBootstrapContext` moved from `org.springframework.boot` to `org.springframework.boot.bootstrap`. `EnvironmentPostProcessor` moved in the opposite direction — from `org.springframework.boot.env` to `org.springframework.boot`.

```
~~import org.springframework.boot.BootstrapRegistry;~~
import org.springframework.boot.bootstrap.BootstrapRegistry;
```

```
~~import org.springframework.boot.ConfigurableBootstrapContext;~~
import org.springframework.boot.bootstrap.ConfigurableBootstrapContext;
```

```
~~import org.springframework.boot.env.EnvironmentPostProcessor;~~
import org.springframework.boot.EnvironmentPostProcessor;
```

HOW TO FIX

1. **Update imports.** Find all usages of the old package paths and replace with the new ones. Both changes are pure package moves — no method signatures or behaviour changed.
2. **Search spring.factories / AutoConfiguration.imports.** If you register an `EnvironmentPostProcessor` in `META-INF/spring.factories` or `META-INF/spring/org.springframework.boot.env.EnvironmentPostProcessor.imports`, update the key to `org.springframework.boot.EnvironmentPostProcessor`.

> ✓ **Check:**
> mvn compile — no package does not exist errors for BootstrapRegistry or EnvironmentPostProcessor

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Classic Uber-Jar Loader Removed

The `CLASSIC` uber-jar loader option is removed in Boot 4.0. The Spring Boot Maven/Gradle plugin fails the build if `CLASSIC` is configured. Remove the configuration or switch to the default loader.

WHAT YOU'LL SEE

```
<!-- pom.xml with CLASSIC loader -->
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <loaderImplementation>CLASSIC</loaderImplementation>
    </configuration>
</plugin>

// Boot 4.0 build error:
[ERROR] Failed to execute goal org.springframework.boot:spring-boot-maven-plugin
...CLASSIC loader implementation is no longer supported
```

WHAT CHANGED

The `CLASSIC` loader implementation — the original Spring Boot uber-jar format — was removed from the Maven and Gradle plugins. Any build file with `loaderImplementation=CLASSIC` fails at the packaging phase. The default nested-jar loader (introduced in Boot 3.2) is the only supported format.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <loaderImplementation>CLASSIC</loaderImplementation>
    </configuration>
</plugin>
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <!-- No loaderImplementation needed — default is the only option -->
</plugin>
```

```
tasks.named("bootJar") {
    loaderImplementation = LoaderImplementation.CLASSIC
}
// No loaderImplementation configuration needed
```

```
ENTRYPOINT ["java", \
  "org.springframework.boot.loader.JarLauncher"]
ENTRYPOINT ["java", \
  "org.springframework.boot.loader.launch.JarLauncher"]
```

HOW TO FIX

1. **Remove the CLASSIC loader configuration.** Delete `<loaderImplementation>CLASSIC</loaderImplementation>` from the Maven plugin configuration, or remove `loaderImplementation = LoaderImplementation.CLASSIC` from the Gradle boot jar task. The build then defaults to the modern nested-jar loader.

2. **Update Docker ENTRYPOINT if using layered jars.** The new loader lives in the `org.springframework.boot.loader.launch` package. Dockerfiles that use an explicit class ENTRYPOINT (common with the layered-jar extraction pattern) must change from

`org.springframework.boot.loader.JarLauncher` to `org.springframework.boot.loader.launch.JarLauncher`. Deployments using `java -jar app.jar` are unaffected because the manifest is updated by the plugin automatically. Paketo buildpacks and Jib are also unaffected.

✓ Check:
Application jar is built successfully and starts correctly after removing CLASSIC loader configuration from the build file

Elasticsearch RestClient Renamed to Rest5Client

Spring Boot 4.0 auto-configures Rest5Client instead of RestClient for Elasticsearch.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/ElasticsearchUsage.java:[5,35]
  error: cannot find symbol
    symbol:   class RestClient
    location: package org.elasticsearch.client
```

WHAT CHANGED

Spring Boot 4.0 auto-configures `org.elasticsearch.client.Rest5Client` instead of `org.elasticsearch.client.RestClient`. Components that inject `RestClient` directly will fail because the auto-configured bean is now of type `Rest5Client`.

```
~~import org.elasticsearch.client.RestClient;~~
import org.elasticsearch.client.Rest5Client;
```

```
~~@Autowired~~
~~private RestClient restClient;~~
@Autowired
private Rest5Client restClient;
```

```
~~@Bean~~
~~public RestClientBuilderCustomizer customizer() { ... }~~
@Bean
public Rest5ClientBuilderCustomizer customizer() { ... }
```

HOW TO FIX

1. **Rename RestClient to Rest5Client.** Replace `RestClient` with `Rest5Client` in all injection points, customizer beans, and direct usages. The API is broadly compatible but check the Elasticsearch client changelog for any subtle differences.

✓ Check:
mvn compile — no cannot find symbol errors for RestClient auto-config injection

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

@EntityScan Relocated to persistence.autoconfigure

@EntityScan moved from org.springframework.boot.autoconfigure.domain to org.springframework.boot.persistence.autoconfigure. Old import doesn't compile.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/EntityScanApp.java:[3,52]
  error: package org.springframework.boot.autoconfigure.domain does not exist
```

WHAT CHANGED

`@EntityScan` moved from `org.springframework.boot.autoconfigure.domain` to `org.springframework.boot.persistence.autoconfigure`. The annotation's behaviour is unchanged — only the package differs.

```
~~import org.springframework.boot.autoconfigure.domain.EntityScan;~~
import org.springframework.boot.persistence.autoconfigure.EntityScan;
```

HOW TO FIX

1. **Update the import.** Replace `org.springframework.boot.autoconfigure.domain.EntityScan` with `org.springframework.boot.persistence.autoconfigure.EntityScan`. One import change per file — no other modifications needed.

✓ Check:
mvn compile — no package does not exist error for @EntityScan import

GraalVM 25 for Native Image Builds

Spring Boot 4.0 requires GraalVM 25 for native image compilation. Older GraalVM versions produce build failures or runtime crashes.

WHAT YOU'LL SEE

```
$ mvn -Pnative native:compile
[ERROR] Error: Unsupported native-image version: 22.3.5
[ERROR] Spring Boot 4.0 requires GraalVM 25.0 or later.
[ERROR] Please upgrade your GraalVM installation.
[ERROR]
[ERROR] → [Help 1]
[ERROR] org.graalvm.buildtools:native-maven-plugin:0.10.6:compile failed
```

WHAT CHANGED

Spring Boot 4.0 targets GraalVM 25 (based on JDK 25) for native image builds. The `native-maven-plugin` and `org.graalvm.buildtools.native` Gradle plugin must be updated to version 0.10.6+. Older GraalVM versions (22.x, 23.x) are no longer compatible with the reachability metadata shipped in starters.

```
<plugin>
    <groupId>org.graalvm.buildtools</groupId>
    <artifactId>native-maven-plugin</artifactId>
    <version>0.9.28</version>
</plugin>
<plugin>
    <groupId>org.graalvm.buildtools</groupId>
    <artifactId>native-maven-plugin</artifactId>
    <version>0.10.6</version>
</plugin>
```

```
plugins {
    id 'org.graalvm.buildtools.native' version '0.9.28'
}
plugins {
    id 'org.graalvm.buildtools.native' version '0.10.6'
}
```

```
export GRAALVM_HOME=/opt/graalvm-ce-java17-22.3.5
export GRAALVM_HOME=/opt/graalvm-jdk-25+35
```

HOW TO FIX

1. **Install GraalVM 25.** Download GraalVM 25 from graalvm.org and set `GRAALVM_HOME`. If using SDKMAN: `sdk install java 25-graal`.
2. **Update native build tools.** Bump the `native-maven-plugin` or Gradle `org.graalvm.buildtools.native` plugin to 0.10.6+. The parent POM manages this if you inherit from `spring-boot-starter-parent`.

> ✓ **Check:**
> native-image –version shows 25+ and mvn -Pnative package succeeds

Gradle 8.14+ / Gradle 9 Required

Spring Boot 4.0's Gradle plugin requires Gradle 8.14 or later. Older Gradle wrappers fail immediately at configuration time.

WHAT YOU'LL SEE

```
$ ./gradlew build
FAILURE: Build failed with an exception.
* Where:
Build file '/project/build.gradle' line: 3
* What went wrong:
An exception occurred applying plugin request [id: 'org.springframework.boot', version:
'4.0.0']
> Failed to apply plugin 'org.springframework.boot'.
   > Spring Boot plugin requires Gradle 8.14+. Found: 8.5.
```

WHAT CHANGED

The Spring Boot Gradle plugin now requires Gradle 8.14 as a minimum. Gradle 9.0 is fully supported and recommended for new projects. The plugin uses Gradle APIs introduced in 8.14 that don't exist in earlier versions.

```
~~distributionUrl=https\://services.gradle.org/distributions/gradle-8.5-bin.zip~~
distributionUrl=https\://services.gradle.org/distributions/gradle-8.14-bin.zip
```

```
~~distributionUrl=https\://services.gradle.org/distributions/gradle-8.5-bin.zip~~
distributionUrl=https\://services.gradle.org/distributions/gradle-9.0-bin.zip
```

HOW TO FIX

1. **Update the Gradle wrapper.** Run `./gradlew wrapper --gradle-version 8.14` (or `9.0`) to update the wrapper in place. Commit the updated wrapper files.
2. **Check plugin compatibility.** After upgrading Gradle, run a build to verify all other plugins are compatible. The Gradle upgrade guide lists deprecations to address.

✓ Check:
./gradlew –version shows 8.x+ and build succeeds

Hibernate EmptyInterceptor Removed

Hibernate's EmptyInterceptor is gone. Implement Interceptor directly — it has default methods for everything you don't need to override.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/MyHibernateInterceptor.java:[5,43]
  error: cannot find symbol
    symbol: class EmptyInterceptor
    location: package org.hibernate
```

WHAT CHANGED

`org.hibernate.EmptyInterceptor` has been removed. The `org.hibernate.Interceptor` interface now has default (no-op) implementations for all its methods, so the abstract base class serves no purpose.

```
import org.hibernate.EmptyInterceptor;

public class MyInterceptor extends EmptyInterceptor {
    @Override
    public boolean onLoad(Object entity, Object id,
            Object[] state, String[] propertyNames, Type[] types) {
        // custom logic
        return false;
    }
}
```

```
import org.hibernate.Interceptor;

public class MyInterceptor implements Interceptor {
    @Override
    public boolean onLoad(Object entity, Object id,
            Object[] state, String[] propertyNames, Type[] types) {
        // custom logic
        return false;
    }
    // All other methods have default no-op implementations
}
```

HOW TO FIX

1. **Implement Interceptor directly.** Replace `extends EmptyInterceptor` with `implements Interceptor`. Remove the import for `EmptyInterceptor`. You only need to override the methods you actually use — all others have default no-op implementations on the interface itself.

✓ Check:
mvn compile — no cannot find symbol for EmptyInterceptor

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

hibernate-jpamodelgen Renamed

Hibernate renamed its annotation processor artifact from hibernate-jpamodelgen to hibernate-processor. Metamodel generation breaks.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[WARNING] The POM for org.hibernate.orm:hibernate-jpamodelgen:jar:7.0.0 is missing, no
dependency
  information available
[ERROR] Failed to execute goal on project my-service: Could not resolve dependencies:
  Could not find artifact org.hibernate.orm:hibernate-jpamodelgen:jar:7.0.0
  in central (https://repo.maven.apache.org/maven2)
```

WHAT CHANGED

Hibernate 7.0 (shipped with Spring Boot 4.0) renamed the annotation processor artifact from `org.hibernate.orm:hibernate-jpamodelgen` to `org.hibernate.orm:hibernate-processor`. The processor now handles both JPA static metamodel generation and Hibernate's new query validation.

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-processor</artifactId>
    <scope>provided</scope>
</dependency>
```

```
annotationProcessor 'org.hibernate.orm:hibernate-jpamodelgen'
annotationProcessor 'org.hibernate.orm:hibernate-processor'
```

```
<path>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
</path>
<path>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-processor</artifactId>
</path>
```

HOW TO FIX

1. **Rename the artifact.** Replace `hibernate-jpamodelgen` with `hibernate-processor` in your POM or Gradle build file. The version is managed by the Spring Boot BOM — just change the artifact ID.
2. **Check compiler plugin configuration.** If you declared the processor in `maven-compiler-plugin`'s `annotationProcessorPaths`, update it there too. Both the dependency and the processor path must use the new name.

> ✓ **Check:**
> mvn compile uses hibernate-processor and metamodel generates

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Hibernate SelectionQuery.setOrder() Removed

Hibernate's incubating SelectionQuery.setOrder() was removed in 7.0. Use an ORDER BY clause in HQL instead.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/HibernateQueryUsage.java:[14,15]
  error: cannot find symbol
    symbol:   method setOrder(java.util.List)
    location: interface SelectionQuery<Product>
```

WHAT CHANGED

The `@Incubating` `setOrder(List<Order>)` method on `SelectionQuery` (and `Query`) was removed in Hibernate 7.0. It had been added in Hibernate 6.x as part of Jakarta Data integration work but was never promoted to stable API.

```
SelectionQuery<Product> query =
    session.createSelectionQuery("from Product", Product.class);
query.setOrder(List.of(Order.asc(Product.class, "name")));
```

```
SelectionQuery<Product> query =
    session.createSelectionQuery("from Product order by name asc", Product.class);
```

HOW TO FIX

1. **Move ordering into the HQL query string.** Add an `ORDER BY` clause directly to the HQL/JPQL string. This is the most portable approach and works on both 3.5 and 4.0.
2. **Use CriteriaQuery with Order.** For fully programmatic ordering, use the JPA Criteria API: `criteriaQuery.orderBy(builder.asc(root.get("name")))`. This is the stable alternative.

✓ Check:
mvn compile — no cannot find symbol for setOrder on SelectionQuery

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

Session.delete() Removed

Hibernate 7.0 removed Session.delete() — use Session.remove() instead. Any direct Hibernate Session usage for deleting entities won't compile.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/repo/CustomRepoImpl.java:[28,16]
  error: cannot find symbol
    symbol:   method delete(com.example.model.Order)
    location: variable session of type org.hibernate.Session
```

WHAT CHANGED

Hibernate 7.0 removed the legacy `Session.delete()` method. The JPA-standard `EntityManager.remove()` (which maps to `Session.remove()`) is the only way to delete entities. Similarly, `Session.save()`, `Session.update()`, and `Session.saveOrUpdate()` were also removed — use `persist()` and `merge()` instead.

```
~~session.delete(order);~~
session.remove(order);
```

```
~~session.save(newOrder);~~
session.persist(newOrder);
```

```
~~session.update(existingOrder);~~
session.merge(existingOrder);
```

```
~~session.saveOrUpdate(order);~~
session.merge(order);
```

HOW TO FIX

1. **Replace with JPA standard methods.** `delete()` becomes `remove()`, `save()` becomes `persist()`, `update()` becomes `merge()`, `saveOrUpdate()` becomes `merge()`.
2. **Prefer EntityManager over Session.** If you're unwrapping the `Session` just to call these methods, consider using `EntityManager` directly. It has the same `persist()`, `merge()`, and `remove()` methods.

✓ **Check:**
mvn compile — no Session.delete() symbol errors

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Hibernate @Where and @OrderBy Removed

Hibernate's @Where and @OrderBy are removed. Replace with @SQLRestriction and @SQLOrder.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/User.java:[5,39]
  error: cannot find symbol
    symbol: class Where
    location: package org.hibernate.annotations
[ERROR] /src/main/java/com/example/User.java:[6,39]
  error: cannot find symbol
    symbol: class OrderBy
    location: package org.hibernate.annotations
```

WHAT CHANGED

`@org.hibernate.annotations.Where` and `@org.hibernate.annotations.OrderBy` have been removed. The replacements are `@org.hibernate.annotations.SQLRestriction` and `@org.hibernate.annotations.SQLOrder`, which take the same SQL fragment arguments.

```
~~import org.hibernate.annotations.Where;~~
~~import org.hibernate.annotations.OrderBy;~~

~~@OneToMany~~
~~@Where(clause = "deleted = false")~~
~~@OrderBy(clause = "name desc")~~
~~private List<Post> posts;~~
import org.hibernate.annotations.SQLRestriction;
import org.hibernate.annotations.SQLOrder;

@OneToMany
@SQLRestriction("deleted = false")
@SQLOrder("name desc")
private List<Post> posts;
```

HOW TO FIX

1. **Replace @Where with @SQLRestriction.** Change the annotation name and attribute: `@Where(clause = "...")` becomes `@SQLRestriction("...")`. The SQL fragment is the same.
2. **Replace @OrderBy with @SQLOrder.** Change the annotation name and attribute: `@OrderBy(clause = "...")` becomes `@SQLOrder("...")`. Do not confuse with JPA's `@jakarta.persistence.OrderBy`, which is a separate annotation taking attribute names and is unaffected.

✓ Check:
mvn compile — no cannot find symbol for @Where or @OrderBy from Hibernate

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

HttpHeaders No Longer Implements MultiValueMap

HttpHeaders no longer implements MultiValueMap. Code that treats headers as a general-purpose map fails to compile.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/HttpHeadersUsage.java:[8,24]
  error: incompatible types: HttpHeaders cannot be converted to MultiValueMap
[ERROR] /src/main/java/com/example/HttpHeadersUsage.java:[12,16]
  error: cannot find symbol
    symbol: method containsKey(String)
```

WHAT CHANGED

`HttpHeaders` no longer implements `MultiValueMap<String, String>` or `Map<String, List<String>>`. Map-style methods like `containsKey()`, `keySet()`, and `entrySet()` are gone from the type. Header-specific methods replace them.

```
~~headers.containsKey("Content-Type")~~
headers.containsHeader("Content-Type")
```

```
~~Set<String> names = headers.keySet();~~
Set<String> names = headers.headerNames();
```

```
~~Set<Map.Entry<String, List<String>>> entries = headers.entrySet();~~
Set<Map.Entry<String, List<String>>> entries = headers.headerSet();
```

```
~~void process(MultiValueMap<String, String> map) { ... }~~
~~process(headers);~~
void process(HttpHeaders headers) { ... }
// or, if the method signature can't change:
process(headers.toMultiValueMap());
```

HOW TO FIX

1. **Replace Map methods with HttpHeaders methods.** `containsKey(k)` → `containsHeader(k)`, `keySet()` → `headerNames()`, `entrySet()` → `headerSet()`. These replacements compile on both 3.5 and 4.0.
2. **For method signatures that accept MultiValueMap.** If you have a utility method that accepts `MultiValueMap<String, String>` and you pass `HttpHeaders` into it, either change the parameter type to `HttpHeaders`, or call `headers.toMultiValueMap()` at the call site.

✓ Check:
mvn compile — no incompatible types or cannot find symbol errors on HttpHeaders usage

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

Jackson Class Renames

Jackson 3.0 renamed core classes like JsonSerializer, JsonDeserializer, and SerializerProvider. Every custom serialiser is now broken.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/DateSerializer.java:[4,44]
  error: cannot find symbol
    symbol:   class JsonSerializer
    location: package tools.jackson.databind
[ERROR] /src/main/java/com/example/DateSerializer.java:[12,47]
  error: cannot find symbol
    symbol:   class SerializerProvider
    location: package tools.jackson.databind
```

WHAT CHANGED

Jackson 3.0 renamed several core classes: `JsonSerializer` became `ValueSerializer`, `JsonDeserializer` became `ValueDeserializer`, `SerializerProvider` became `SerializationContext`, `DeserializationContext` kept its name but moved packages. Method signatures on these classes also changed.

```
~~import com.fasterxml.jackson.databind.JsonSerializer;~~
~~import com.fasterxml.jackson.databind.SerializerProvider;~~

~~public class DateSerializer extends JsonSerializer<LocalDate> {~~
~~    @Override~~
~~    public void serialize(LocalDate value, JsonGenerator gen,~~
~~            SerializerProvider provider) throws IOException {~~
import tools.jackson.databind.ser.ValueSerializer;
import tools.jackson.databind.SerializationContext;

public class DateSerializer extends ValueSerializer<LocalDate> {
    @Override
    public void serialize(LocalDate value, JsonGenerator gen,
            SerializationContext ctxt) throws IOException {
```

```
~~import com.fasterxml.jackson.databind.JsonDeserializer;~~

~~public class DateDeserializer extends JsonDeserializer<LocalDate> {~~
import tools.jackson.databind.deser.ValueDeserializer;

public class DateDeserializer extends ValueDeserializer<LocalDate> {
```

```
~~import com.fasterxml.jackson.databind.JsonMappingException;~~
import tools.jackson.databind.DatabindException;
```

HOW TO FIX

1. **OpenRewrite (recommended).** Run the Jackson 3 type changes recipe. It handles class renames, method signature updates, and import changes in one pass.
2. **Manual migration.** Replace class names and imports file by file. The full mapping is in the Jackson 3 migration guide. Don't forget to update method parameter types in overridden methods.

✓ **Check:**

mvn compile — no JsonSerializer/SerializerProvider errors

@JsonComponent and @JsonMixin Renamed to @JacksonComponent and @JacksonMixin

`@JsonComponent` and `@JsonMixin` are removed in Boot 4.0. Replace with `@JacksonComponent` and `@JacksonMixin`. Code that uses the old names fails to compile.

WHAT YOU'LL SEE

```
@JsonComponent
public class MoneySerializer extends JsonSerializer<Money> { ... }

// Boot 4.0 compile error:
error: cannot find symbol
  symbol: @JsonComponent
  location: class MoneySerializer
```

WHAT CHANGED

Spring Boot's custom serializer registration annotations were renamed to align with Jackson 3's branding. `@JsonComponent` becomes `@JacksonComponent` and `@JsonMixin` becomes `@JacksonMixin`. The functionality is identical — only the annotation names changed.

```
~~import org.springframework.boot.jackson.JsonComponent;~~

~~@JsonComponent~~
~~public class MoneySerializer extends JsonSerializer<Money> { ... }~~
import org.springframework.boot.jackson.JacksonComponent;

@JacksonComponent
public class MoneySerializer extends JsonSerializer<Money> { ... }
```

```
~~import org.springframework.boot.jackson.JsonMixin;~~

~~@JsonMixin(Money.class)~~
~~public abstract class MoneyMixin { ... }~~
import org.springframework.boot.jackson.JacksonMixin;

@JacksonMixin(Money.class)
public abstract class MoneyMixin { ... }
```

HOW TO FIX

1. **Rename the annotations.** Replace all `@JsonComponent` with `@JacksonComponent` and all `@JsonMixin` with `@JacksonMixin`. Update the imports accordingly. The behaviour is identical.

✓ Check:
Custom serializers and mixins annotated with the new names register correctly with the auto-configured ObjectMapper

Jackson 3.0 Group ID Change

Jackson 3.0 moved from com.fasterxml.jackson to tools.jackson. Every Maven coordinate and every import path is now wrong.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/JacksonConfig.java:[3,32]
  package com.fasterxml.jackson.databind does not exist
[ERROR] /src/main/java/com/example/JacksonConfig.java:[8,5]
  cannot find symbol
    symbol: class ObjectMapper
```

WHAT CHANGED

Jackson 3.0 moved from the `com.fasterxml.jackson` Maven group to `tools.jackson`. Spring Boot 4.0's managed dependencies point to Jackson 3.0, so the old group ID no longer resolves. Every import statement using `com.fasterxml.jackson` breaks. See also: jackson-class-renames (for the class-level renames that come with the same upgrade), jackson-exception-hierarchy.

```
~~<dependency>~~
~~    <groupId>com.fasterxml.jackson.core</groupId>~~
~~    <artifactId>jackson-databind</artifactId>~~
~~</dependency>~~
<dependency>
    <groupId>tools.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

```
~~import com.fasterxml.jackson.databind.ObjectMapper;~~
~~import com.fasterxml.jackson.core.JsonProcessingException;~~
import tools.jackson.databind.ObjectMapper;
import tools.jackson.core.JacksonException;
```

HOW TO FIX

1. **OpenRewrite (recommended).** Run the Jackson 3 migration recipe. Handles group ID changes and import updates across the entire codebase in one pass.
2. **Manual: update Maven coordinates first.** Replace `com.fasterxml.jackson` with `tools.jackson` in all POM or Gradle files. Then fix imports file by file. Note that the group ID fix alone is not enough — class names also changed. See: jackson-class-renames.

> ✓ **Check:**
> mvn compile succeeds with no com.fasterxml errors

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Java 17 Minimum Requirement

Spring Boot 4.0 requires Java 17 as the minimum language level, with 21 recommended. Projects still on Java 11 or earlier won't compile at all.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-
plugin:3.13.0:compile
  on project my-service: Fatal error compiling: error: release version 17 not supported
[ERROR]
[ERROR] → [Help 1] Check that your JAVA_HOME points to a JDK 17+ installation.
```

WHAT CHANGED

Spring Boot 4.0 sets `java.version` to `17` as the baseline. The framework bytecode and all starters are compiled at the Java 17 language level. Java 11 and earlier are no longer supported.

```
<java.version>11</java.version>
<java.version>17</java.version>
```

```
java {
    sourceCompatibility = JavaVersion.VERSION_11
}
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}
```

```
FROM eclipse-temurin:11-jre
FROM eclipse-temurin:17-jre
```

HOW TO FIX

1. **Upgrade your JDK.** Install JDK 17 or (recommended) JDK 21 and set `JAVA_HOME` accordingly. Update `java.version` in your POM or `toolchain` block in Gradle.
2. **Update CI and container base images.** Change your CI pipeline and Dockerfiles to use a 17+ base image. `eclipse-temurin:21-jre` is a good default.

> ✓ **Check:**
> java -version shows 17+ and mvn compile succeeds

Kafka StreamsBuilderFactoryBeanCustomizer Removed

Spring Boot's StreamsBuilderFactoryBeanCustomizer is gone. Replace it with Spring Kafka's StreamsBuilderFactoryBeanConfigurer.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/KafkaCustomizerUsage.java:[3,60]
  error: package org.springframework.boot.autoconfigure.kafka does not contain
  StreamsBuilderFactoryBeanCustomizer
```

WHAT CHANGED

`org.springframework.boot.autoconfigure.kafka.StreamsBuilderFactoryBeanCustomizer` has been removed. The replacement is `org.springframework.kafka.config.StreamsBuilderFactoryBeanConfigurer` from Spring Kafka itself.

```
~~import org.springframework.boot.autoconfigure.kafka.StreamsBuilderFactoryBeanCustomizer;~~
import org.springframework.kafka.config.StreamsBuilderFactoryBeanConfigurer;
```

```
~~@Bean~~
~~public StreamsBuilderFactoryBeanCustomizer customizer() { ... }~~
@Bean
public StreamsBuilderFactoryBeanConfigurer configurer() { ... }
```

```
~~public class MyCustomizer implements StreamsBuilderFactoryBeanCustomizer {~~
public class MyCustomizer implements StreamsBuilderFactoryBeanConfigurer {
```

HOW TO FIX

1. **Switch to StreamsBuilderFactoryBeanConfigurer.** Replace the import and the implements / return type declaration. The interface method signature is compatible — the method you override is `configure(StreamsBuilderFactoryBean)` in both cases.

✓ Check:
mvn compile — no cannot find symbol for StreamsBuilderFactoryBeanCustomizer

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

ListenableFuture Removed

Spring Framework removed ListenableFuture entirely. All async code using it must switch to CompletableFuture.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/service/AsyncService.java:[4,49]
  error: cannot find symbol
    symbol:   class ListenableFuture
    location: package org.springframework.util.concurrent
[ERROR] /src/main/java/com/example/service/AsyncService.java:[5,49]
  error: cannot find symbol
    symbol:   class ListenableFutureCallback
    location: package org.springframework.util.concurrent
```

WHAT CHANGED

`ListenableFuture`, `ListenableFutureCallback`, `ListenableFutureTask`, and related classes in `org.springframework.util.concurrent` were deleted from Spring Framework 7.0. All Spring APIs that previously returned `ListenableFuture` now return `java.util.concurrent.CompletableFuture`.

```
~~import org.springframework.util.concurrent.ListenableFuture;~~

~~@Async~~
~~public ListenableFuture<String> fetchData() {~~
~~    return AsyncResult.forValue(doWork());~~
~~}~~
import java.util.concurrent.CompletableFuture;

@Async
public CompletableFuture<String> fetchData() {
    return CompletableFuture.completedFuture(doWork());
}
```

```
~~ListenableFuture<String> future = asyncService.fetchData();~~
~~future.addCallback(~~
~~    result -> log.info("Success: {}", result),~~
~~    ex -> log.error("Failed", ex)~~
~~);~~
CompletableFuture<String> future = asyncService.fetchData();
future.whenComplete((result, ex) -> {
    if (ex != null) log.error("Failed", ex);
    else log.info("Success: {}", result);
});
```

HOW TO FIX

1. **Replace with CompletableFuture.** Change return types from `ListenableFuture<T>` to `CompletableFuture<T>`. Replace `AsyncResult.forValue()` with `CompletableFuture.completedFuture()`. Replace `addCallback()` with `whenComplete()` or `thenAccept()`.
2. **Use OpenRewrite.** The `org.openrewrite.java.spring.framework.MigrateSpringAssert` recipe set includes `ListenableFuture` migration.

✓ Check:

mvn compile — no ListenableFuture symbol errors

Maven Plugin Version Alignment for AOT

The Spring Boot Maven plugin's AOT processing goals changed in 4.0. Mismatched plugin versions or stale AOT configuration breaks the build.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn spring-boot:process-aot
[ERROR] Failed to execute goal org.springframework.boot:spring-boot-maven-
plugin:4.0.0:process-aot
  on project my-service: Execution default of goal
  org.springframework.boot:spring-boot-maven-plugin:4.0.0:process-aot failed:
  An API incompatibility was encountered while executing
  org.springframework.boot:spring-boot-maven-plugin:4.0.0:process-aot:
  java.lang.NoSuchMethodError:
    'void
org.springframework.aot.generate.GenerationContext.<init>(java.lang.ClassLoader)'
```

WHAT CHANGED

The `spring-boot-maven-plugin` must match the Spring Boot version exactly. In 4.0 the AOT processing API changed — the `process-aot` and `process-test-aot` goals use new generation context methods. A plugin version mismatch causes `NoSuchMethodError` at build time.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>3.4.1</version>
</plugin>
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>4.0.0</version>
</plugin>
```

```
<execution>
    <id>process-aot</id>
    <goals><goal>process-aot</goal></goals>
</execution>
<!— AOT goals are now configured automatically by the parent POM —>
```

HOW TO FIX

1. **Align the plugin version with Spring Boot.** If you inherit from `spring-boot-starter-parent`, the plugin version is managed automatically — remove any explicit `<version>` tag. If you use a BOM without the parent POM, set the plugin version to `4.0.0` explicitly.
2. **Remove manual AOT execution blocks.** The parent POM now binds the AOT goals automatically. Keeping an explicit `<execution>` block can cause the goal to run twice or with stale configuration.

> ✓ **Check:**
> mvn spring-boot:aot-generate runs without plugin errors

OkHttp3 Support Removed

Spring Boot 4.0 dropped the OkHttp3 client factory and BOM entry. Projects using OkHttp3RestClient or the managed dependency won't compile.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/config/RestConfig.java:[5,48]
  package org.springframework.http.client.OkHttp3ClientHttpRequestFactory does not exist
[ERROR] /src/main/java/com/example/config/RestConfig.java:[14,16]
  cannot find symbol
    symbol:   class OkHttp3ClientHttpRequestFactory
    location: class com.example.config.RestConfig
```

WHAT CHANGED

The `OkHttp3ClientHttpRequestFactory` class was removed from Spring Framework 7.0. Spring Boot 4.0 also removed `com.squareup.okhttp3` from its managed dependency BOM. Any code creating a `RestClient` or `RestTemplate` with the OkHttp3 factory won't compile.

```
~~import org.springframework.http.client.OkHttp3ClientHttpRequestFactory;~~

~~@Bean~~
~~public RestClient restClient() {~~
~~    return RestClient.builder()~~
~~        .requestFactory(new OkHttp3ClientHttpRequestFactory())~~
~~        .build();~~
~~}~~
import org.springframework.http.client.JdkClientHttpRequestFactory;

@Bean
public RestClient restClient() {
    return RestClient.builder()
        .requestFactory(new JdkClientHttpRequestFactory())
        .build();
}
```

```
~~<dependency>~~
~~    <groupId>com.squareup.okhttp3</groupId>~~
~~    <artifactId>okhttp</artifactId>~~
~~</dependency>~~
<!— JdkClientHttpRequestFactory uses java.net.http — no extra dependency needed —>
```

HOW TO FIX

1. **Switch to JdkClientHttpRequestFactory (simplest).** Replace `OkHttp3ClientHttpRequestFactory` with `JdkClientHttpRequestFactory`. It uses the JDK's built-in HTTP client and requires no additional dependencies.
2. **Switch to Apache HttpClient 5.** If you need connection pooling or advanced proxy configuration, use `HttpComponentsClientHttpRequestFactory` with Apache HttpClient 5. Add `org.apache.httpcomponents.client5:httpclient5` to your POM.

> **✓ Check:**
> mvn dependency:tree shows no okhttp3 and build compiles

PropertyMapper.alwaysApplyingWhenNonNull() Removed

PropertyMapper.alwaysApplyingWhenNonNull() is removed because skipping null values is now the default. Just delete the call.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/PropertyMapperUsage.java:[8,47]
  error: cannot find symbol
    symbol:   method alwaysApplyingWhenNonNull()
    location: class PropertyMapper
```

WHAT CHANGED

`PropertyMapper.alwaysApplyingWhenNonNull()` has been removed because its behaviour — skipping mappings when the source value is `null` — is now the default behaviour of `PropertyMapper`. Calling it was a no-op from 4.0 onward, so the method was removed.

```
~~PropertyMapper map = PropertyMapper.get().alwaysApplyingWhenNonNull();~~
PropertyMapper map = PropertyMapper.get();
```

HOW TO FIX

1. **Delete the alwaysApplyingWhenNonNull() call.** Simply remove `.alwaysApplyingWhenNonNull()` from the chain. The mapper behaves identically on Boot 4.0 without it. If you explicitly want to map null values, use `.always()` on individual mappings.

✓ Check:
mvn compile — no cannot find symbol for alwaysApplyingWhenNonNull

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

@PropertyMapping Relocated to test.context

@PropertyMapping moved from org.springframework.boot.test.autoconfigure.properties to org.springframework.boot.test.context. Old import doesn't compile.

WHAT YOU'LL SEE

```
$ mvn test-compile
[ERROR] /src/test/java/com/example/CustomTestAnnotation.java:[3,62]
  error: package org.springframework.boot.test.autoconfigure.properties does not exist
```

WHAT CHANGED

`@PropertyMapping` moved from `org.springframework.boot.test.autoconfigure.properties` to `org.springframework.boot.test.context`. Behaviour is unchanged.

```
~~import org.springframework.boot.test.autoconfigure.properties.PropertyMapping;~~
import org.springframework.boot.test.context.PropertyMapping;
```

HOW TO FIX

1. **Update the import.** Replace `org.springframework.boot.test.autoconfigure.properties.PropertyMapping` with `org.springframework.boot.test.context.PropertyMapping`. This typically appears only in custom meta-annotations for test slices.

✓ Check:
mvn test-compile — no package does not exist error for @PropertyMapping import

TIER 1 · WON'T BUILD IMPACT: L spring-boot 3.5 → 4.0

Security DSL Rewrite

Spring Security removed authorizeRequests(), antMatchers(), and the .and() chaining method. The entire HTTP security DSL must be rewritten.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/SecurityConfig.java:[22,14]
  error: cannot find symbol
    symbol:   method authorizeRequests()
    location: variable http of type HttpSecurity
[ERROR] /src/main/java/com/example/SecurityConfig.java:[23,18]
  error: cannot find symbol
    symbol:   method antMatchers(java.lang.String)
[ERROR] /src/main/java/com/example/SecurityConfig.java:[28,10]
  error: cannot find symbol
    symbol:   method and()
```

WHAT CHANGED

Spring Security 7.0 (shipped with Boot 4.0) removed the deprecated `authorizeRequests()` method — use `authorizeHttpRequests()` instead. `antMatchers()`, `mvcMatchers()`, and `regexMatchers()` are replaced by `requestMatchers()`. The `.and()` chaining method is gone — use lambda DSL with separate method calls instead.

```
~~http~~
~~    .authorizeRequests()~~
~~        .antMatchers("/public/**").permitAll()~~
~~        .antMatchers("/admin/**").hasRole("ADMIN")~~
~~        .anyRequest().authenticated()~~
~~        .and()~~
~~    .formLogin()~~
~~        .loginPage("/login")~~
~~        .and()~~
~~    .logout()~~
~~        .logoutSuccessUrl("/");~~
http
    .authorizeHttpRequests(auth → auth
        .requestMatchers("/public/**").permitAll()
        .requestMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated()
    )
    .formLogin(form → form
        .loginPage("/login")
    )
    .logout(logout → logout
        .logoutSuccessUrl("/")
    );
```

```
~~http.csrf().disable();~~
http.csrf(csrf → csrf.disable());
```

HOW TO FIX

1. **Rewrite to lambda DSL.** Replace every `.and()` chain with a lambda-based configuration block. Each security concern (authorization, form login, logout, CSRF) gets its own lambda. See the Spring Security 7 migration guide.

2. **Replace matchers.** Change `antMatchers()` and `mvcMatchers()` to `requestMatchers()`. The new method auto-selects the matching strategy. `regexMatchers()` becomes `requestMatchers(new RegexRequestMatcher(...))`.

> ✓ **Check:**
> App starts and security filter chain initialises without errors

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

SimpDestinationMessageMatcher Removed

SimpDestinationMessageMatcher is gone in Spring Security 7.0. WebSocket security now uses the AuthorizationManager pattern.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/SimpDestUsage.java:[3,66]
  error: package org.springframework.security.messaging.util.matcher
  does not contain SimpDestinationMessageMatcher
```

WHAT CHANGED

`org.springframework.security.messaging.util.matcher.SimpDestinationMessageMatcher` has been removed. Spring Security 7.0 replaces the old `AbstractSecurityWebSocketMessageBrokerConfigurer` / matcher-based approach with `MessageMatcherDelegatingAuthorizationManager`.

```
~~import~~
~~org.springframework.security.messaging.util.matcher.SimpDestinationMessageMatcher;~~
~~// ...~~
~~new SimpDestinationMessageMatcher("/topic/**");~~
```

```
import
org.springframework.security.messaging.access.intercept.MessageMatcherDelegatingAuthori
zationManager;
// ...
MessageMatcherDelegatingAuthorizationManager.builder()
    .simpDestMatchers("/topic/**").authenticated()
    .anyMessage().denyAll()
    .build();
```

HOW TO FIX

1. **Migrate to MessageMatcherDelegatingAuthorizationManager.** Replace any use of `SimpDestinationMessageMatcher` and `AbstractSecurityWebSocketMessageBrokerConfigurer` with a `MessageMatcherDelegatingAuthorizationManager` bean registered on your message broker. See the Spring Security 7 messaging migration guide for the full configuration pattern.

✓ Check:
mvn compile — no cannot find symbol for SimpDestinationMessageMatcher

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Spring AMQP RabbitRetryTemplateCustomizer Removed

`RabbitRetryTemplateCustomizer` is removed in Boot 4.0. Replace with `RabbitTemplateRetrySettingsCustomizer` (publisher) or `RabbitListenerRetrySettingsCustomizer` (consumer). Code using the old interface fails to compile.

WHAT YOU'LL SEE

```
@Bean
public RabbitRetryTemplateCustomizer retryCustomizer() {
    return (target, retryTemplate) -> { ... };
}

// Boot 4.0 compile error:
error: cannot find symbol
  symbol: class RabbitRetryTemplateCustomizer
```

WHAT CHANGED

Spring AMQP 4.0 dropped its dependency on Spring Retry and now uses Spring Framework's retry abstraction. The customizer API was redesigned to separate publisher and consumer retry configuration. `RabbitRetryTemplateCustomizer` is replaced by `RabbitTemplateRetrySettingsCustomizer` and `RabbitListenerRetrySettingsCustomizer`.

```
~~import org.springframework.amqp.rabbit.retry.RabbitRetryTemplateCustomizer;~~

~~@Bean~~
~~public RabbitRetryTemplateCustomizer retryCustomizer() {~~
~~    return (target, retryTemplate) -> {~~
~~        retryTemplate.setRetryPolicy(...);~~
~~    };~~
~~}~~
import
org.springframework.boot.autoconfigure.amqp.RabbitTemplateRetrySettingsCustomizer;

@Bean
public RabbitTemplateRetrySettingsCustomizer retryCustomizer() {
    return settings -> {
        settings.setMaxAttempts(3);
    };
}
```

```
~~// Previously both publisher and consumer shared RabbitRetryTemplateCustomizer~~
import
org.springframework.boot.autoconfigure.amqp.RabbitListenerRetrySettingsCustomizer;

@Bean
public RabbitListenerRetrySettingsCustomizer listenerRetryCustomizer() {
    return settings -> {
        settings.setMaxAttempts(5);
    };
}
```

HOW TO FIX

1. **Replace with the split customizer interfaces.** Replace `RabbitRetryTemplateCustomizer` beans with `RabbitTemplateRetrySettingsCustomizer` for publisher retry and `RabbitListenerRetrySettingsCustomizer` for consumer retry. The new interfaces configure settings directly rather than manipulating a `RetryTemplate`.

✓ Check:

RabbitTemplate and RabbitListener retry settings are configured via the new customizer interfaces without compile errors

spring-jcl Removed

Spring Framework replaced its spring-jcl logging bridge with a direct dependency on Commons Logging 1.3.0. Manual spring-jcl exclusions break.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] Failed to execute goal on project my-service:
  Could not resolve dependencies:
  The following artifacts could not be resolved:
    org.springframework:spring-jcl:jar:7.0.0 (not found in central)
```

WHAT CHANGED

The `org.springframework:spring-jcl` module was removed in Spring Framework 7.0. Spring now depends directly on `commons-logging:commons-logging:1.3.0`, which gained native SLF4J and `java.util.logging` bridge support. If your POM explicitly references `spring-jcl` — or excludes `commons-logging` in favour of `spring-jcl` — the build breaks.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <!-- Commons Logging 1.3.0 is pulled in transitively — no exclusion needed -->
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jcl</artifactId>
</dependency>
<!-- spring-jcl no longer exists. Commons Logging 1.3.0 bridges to SLF4J natively. -->
```

HOW TO FIX

1. **Remove spring-jcl references.** Delete any explicit `spring-jcl` dependency declarations. Remove any `commons-logging` exclusions on Spring modules — those were needed to avoid the old Commons Logging / spring-jcl conflict. Commons Logging 1.3.0 bridges to SLF4J out of the box.
2. **Remove jcl-over-slf4j.** If you had `org.slf4j:jcl-over-slf4j` on the classpath, remove it. Commons Logging 1.3.0 provides the same bridge natively. Having both will cause a classpath conflict.

✓ Check:
mvn compile — no spring-jcl dependency errors

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

spring-retry Removed from BOM

Spring Boot 4.0 removed spring-retry from its managed dependency BOM. Projects that relied on version management must now declare the version explicitly.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] 'dependencies.dependency.version' for org.springframework.retry:spring-retry:jar
  is missing. @ line 42, column 17
[ERROR] Failed to execute goal on project my-service:
  Could not resolve dependencies: Failed to collect dependencies at
  org.springframework.retry:spring-retry:jar:RELEASE (no version)
```

WHAT CHANGED

`org.springframework.retry:spring-retry` is no longer version-managed by the Spring Boot BOM. If you declared the dependency without an explicit `<version>` tag, the build fails because Maven/Gradle can't resolve it. The library itself still works — it just isn't managed anymore.

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
    <version>2.0.11</version>
</dependency>
```

```
implementation 'org.springframework.retry:spring-retry'
implementation 'org.springframework.retry:spring-retry:2.0.11'
```

HOW TO FIX

1. **Add an explicit version.** Add `<version>2.0.11</version>` (or the latest compatible release) to your spring-retry dependency declaration. Check Maven Central for the latest version.
2. **Import Spring Retry's own BOM.** If multiple modules use spring-retry, add its BOM to your `<dependencyManagement>` section so versions are managed in one place.

> ✓ **Check:**
> mvn dependency:tree shows aligned Spring Retry version and retry works

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

Spring Security Access API Moved to spring-security-access

AccessDecisionManager and related legacy authorisation classes moved to a separate spring-security-access module. Add it or migrate to AuthorizationManager.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/main/java/com/example/AccessApiUsage.java:[3,48]
  error: package org.springframework.security.access does not exist
```

WHAT CHANGED

`AccessDecisionManager`, `AccessDecisionVoter`, `@EnableGlobalMethodSecurity`, and related legacy authorisation classes have been moved from `spring-security-core` to a new standalone `spring-security-access` artifact.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-access</artifactId>
</dependency>
```

```
~~import org.springframework.security.access.AccessDecisionManager;~~
~~// implements AccessDecisionManager~~
import org.springframework.security.authorization.AuthorizationManager;
// implements AuthorizationManager<RequestAuthorizationContext>
```

HOW TO FIX

1. **Add spring-security-access (short-term fix).** Add `spring-security-access` as an explicit dependency. This restores compilation without any logic changes and buys time to do the full migration.
2. **Migrate to AuthorizationManager (recommended).** Replace `AccessDecisionManager` / `AccessDecisionVoter` implementations with `AuthorizationManager`. Replace `@EnableGlobalMethodSecurity` with `@EnableMethodSecurity`. The Spring Security migration guide has step-by-step instructions.

✓ Check:
mvn compile — no cannot find symbol for AccessDecisionManager or related classes

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Testcontainers 2.0 Package Relocation

Testcontainers 2.0 moved database and middleware containers out of the shared org.testcontainers.containers package into module-specific sub-packages (e.g. org.testcontainers.postgresql). The Maven group ID stays org.testcontainers.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] /src/test/java/com/example/IntegrationTest.java:[4,52]
  error: cannot find symbol
    symbol:   class PostgreSQLContainer
    location: package org.testcontainers.containers
```

WHAT CHANGED

Testcontainers 2.0 resolved JPMS split-package problems by moving specialised containers into module-specific packages. `PostgreSQLContainer` moved from `org.testcontainers.containers` to `org.testcontainers.postgresql`. Similar moves apply to all database and middleware modules (Kafka, Redis, RabbitMQ, etc.). `GenericContainer` remains in `org.testcontainers.containers`. The Maven group ID (`org.testcontainers`) is unchanged.

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers-bom</artifactId>
    <version>1.19.8</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
<!-- Managed by Spring Boot 4.0 parent POM — remove explicit BOM if using starter-parent
-->
```

```
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.containers.KafkaContainer;
import org.testcontainers.postgresql.PostgreSQLContainer;
import org.testcontainers.kafka.KafkaContainer;
```

```
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.containers.GenericContainer;  // unchanged
```

HOW TO FIX

1. **Update Java imports for specialised containers.** For each database or middleware container, change the import from `org.testcontainers.containers.XxxContainer` to the module-specific sub-package, e.g. `org.testcontainers.postgresql.PostgreSQLContainer`, `org.testcontainers.kafka.KafkaContainer`, `org.testcontainers.redis.RedisContainer`. Check the Testcontainers 2.0 migration guide for the full mapping.
2. **Remove explicit BOM version if using Spring Boot parent.** Spring Boot 4.0's parent POM manages the Testcontainers 2.0 version. If you previously imported the Testcontainers BOM explicitly with a 1.x version, remove the version override to avoid conflicts.

✓ Check:
mvn test — Testcontainers start and tests pass

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

TestRestTemplate Package Relocated

TestRestTemplate moved to org.springframework.boot.resttestclient. Every test that imports the old package fails to compile.

WHAT YOU'LL SEE

```
$ mvn test-compile
[ERROR] /src/test/java/com/example/MyControllerTest.java:[3,56]
  error: package org.springframework.boot.test.web.client does not exist
[ERROR] /src/test/java/com/example/MyControllerTest.java:[15,5]
  error: cannot find symbol
    symbol: class TestRestTemplate
```

WHAT CHANGED

`TestRestTemplate` was removed from `org.springframework.boot.test.web.client` and relocated to `org.springframework.boot.resttestclient` as part of a new dedicated module for REST test clients.

```
~~import org.springframework.boot.test.web.client.TestRestTemplate;~~
import org.springframework.boot.resttestclient.TestRestTemplate;
```

HOW TO FIX

1. **Update the import.** Replace `org.springframework.boot.test.web.client.TestRestTemplate` with `org.springframework.boot.resttestclient.TestRestTemplate`. The class itself is functionally equivalent.
2. **Consider RestTestClient.** The new `RestTestClient` provides a more modern, fluent API for REST testing and is the preferred approach in Boot 4.0. If you are refactoring tests anyway, this is a good time to migrate.

> ✓ **Check:**
> mvn test-compile — no package does not exist for org.springframework.boot.test.web.client

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

Tomcat WAR Deployment Requires New Runtime Starter

Boot 4.0 adds `spring-boot-starter-tomcat-runtime` for WAR deployment. Using the old `spring-boot-starter-tomcat` (provided scope) pattern may cause classpath conflicts with the external container.

WHAT YOU'LL SEE

```
// Deploying WAR to external Tomcat with Boot 3.5 dependency pattern:
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>

// Boot 4.0: may result in:
java.lang.ClassNotFoundException: jakarta.servlet.FilterRegistration
// Or: version conflicts between embedded and container Tomcat classes
```

WHAT CHANGED

A new `spring-boot-starter-tomcat-runtime` artifact was introduced specifically for WAR deployment scenarios. It provides the runtime bridge classes needed to run a Boot application inside an external Tomcat without bringing in the full embedded Tomcat server dependencies. The `spring-boot-starter-tomcat` artifact is now exclusively for embedded server use.

```
~~<dependency>~~
~~    <groupId>org.springframework.boot</groupId>~~
~~    <artifactId>spring-boot-starter-tomcat</artifactId>~~
~~    <scope>provided</scope>~~
~~</dependency>~~
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat-runtime</artifactId>
    <scope>provided</scope>
</dependency>
```

HOW TO FIX

1. **Replace spring-boot-starter-tomcat (provided) with spring-boot-starter-tomcat-runtime.** In your WAR project's `pom.xml`, replace the provided-scope `spring-boot-starter-tomcat` dependency with `spring-boot-starter-tomcat-runtime`. Keep the `provided` scope — the external container supplies Tomcat at runtime.

> **✓ Check:**
> WAR file deploys to external Tomcat without ClassNotFoundException or dependency conflicts after swapping to spring-boot-starter-tomcat-runtime

TIER 1 · WON'T BUILD IMPACT: M spring-boot 3.5 → 4.0

Undertow Embedded Server Removed

Spring Boot 4.0 dropped Undertow as an embedded server option. Projects depending on spring-boot-starter-undertow won't resolve.

WHAT YOU'LL SEE

```
$ mvn compile
[ERROR] Failed to execute goal on project my-service:
  Could not resolve dependencies for project com.example:my-service:jar:1.0.0:
  The following artifacts could not be resolved:
    org.springframework.boot:spring-boot-starter-undertow:jar:4.0.0 (not found)
```

WHAT CHANGED

The `spring-boot-starter-undertow` module was removed from Spring Boot 4.0. Undertow is no longer a supported embedded servlet container. The supported options are Tomcat (default), Jetty, and Netty (for reactive).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!— Uses Tomcat by default. For Jetty, add spring-boot-starter-jetty
     and exclude spring-boot-starter-tomcat —>
```

```
server:
  undertow:
    io-threads: 8
    worker-threads: 64
    buffer-size: 1024
server:
  tomcat:
    threads:
      max: 200
      min-spare: 10
```

HOW TO FIX

1. **Switch to Tomcat (default).** Remove the `spring-boot-starter-undertow` dependency and the Tomcat exclusion. Tomcat is the default — just use `spring-boot-starter-web` without modifications.
2. **Switch to Jetty.** If you need a non-Tomcat server, use `spring-boot-starter-jetty` instead. Exclude the Tomcat starter and add the Jetty starter.

✓ Check:

App starts on Tomcat/Jetty with no Undertow class errors

TIER 1 · WON'T BUILD IMPACT: S spring-boot 3.5 → 4.0

webjars-locator-core Removed from BOM

Spring Boot 4.0 removes `webjars-locator-core` from the BOM. WebJar version-agnostic URLs (`/webjars/jquery/jquery.min.js`) stop resolving. Switch to `webjars-locator-lite`.

WHAT YOU'LL SEE

```
// pom.xml — dependency with no version (relies on BOM)
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator-core</artifactId>
</dependency>

// Boot 3.5: BOM provides version 0.59 — resolves fine.

// Boot 4.0: BOM no longer manages webjars-locator-core.
[ERROR] 'dependencies.dependency.version' for
  org.webjars:webjars-locator-core:jar is missing.
[ERROR] BUILD FAILURE

// Or if version is pinned in pom.xml — compiles but WebJarAssetLocator
// is no longer registered by Boot's auto-configuration:
java.lang.ClassNotFoundException: org.webjars.WebJarAssetLocator
```

WHAT CHANGED

`org.webjars:webjars-locator-core` was the original WebJar asset locator, using full classpath scanning to find versioned WebJar paths. Spring Boot 4.0 removes it from the BOM and its auto-configuration wiring, replacing it with `org.webjars:webjars-locator-lite`, which uses a pre-generated manifest file (`webjars-requirejs.js` or `META-INF/resources/webjars/`) to locate assets without scanning.

```
~~<dependency>~~
~~    <groupId>org.webjars</groupId>~~
~~    <artifactId>webjars-locator-core</artifactId>~~
~~</dependency>~~
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator-lite</artifactId>
</dependency>
```

```
~~import org.webjars.WebJarAssetLocator;~~
~~WebJarAssetLocator locator = new WebJarAssetLocator();~~
~~String path = locator.getFullPath("jquery", "jquery.min.js");~~
import org.webjars.lite.WebJarAssetLocator;
WebJarAssetLocator locator = new WebJarAssetLocator();
String path = locator.getFullPath("jquery", "jquery.min.js");
```

HOW TO FIX

1. **Replace webjars-locator-core with webjars-locator-lite.** Update `pom.xml` or `build.gradle` to declare `org.webjars:webjars-locator-lite`. The API is compatible for common use cases — Spring MVC's WebJar resource handler works with both.
2. **Update direct WebJarAssetLocator imports.** If your code imports `org.webjars.WebJarAssetLocator` directly, change the import to the `org.webjars.lite` package. The method signatures for `getFullPath()` are compatible.

✓ Check:

WebJar assets resolve correctly via the webjars-locator-lite replacement

TIER 2

Won't Run

Runtime & Startup Failures

Your code compiles but crashes at startup or runtime. Configuration, security, and framework changes that break execution.

CARD INDEX · 27 CARDS

Actuator Endpoint @Nullable: org.springframework.lang Replaced by JSpecify

Actuator endpoint parameters must use org.jspecify.annotations.Nullable, not org.springframework.lang.Nullable.

WHAT YOU'LL SEE

```
$ mvn compile  (or runtime binding failure)
Actuator endpoint parameter annotated with @org.springframework.lang.Nullable
is no longer recognised — optional parameter treated as required,
causing MissingServletRequestParameterException at runtime.
```

WHAT CHANGED

Spring Boot 4.0 no longer recognises `org.springframework.lang.Nullable` on Actuator `@ReadOperation` / `@WriteOperation` parameters. The supported annotation is now `org.jspecify.annotations.Nullable` from the JSpecify library, which is on the classpath transitively via Spring Framework 7.0.

```
~~import org.springframework.lang.Nullable;~~

~~@Endpoint(id = "my-endpoint")~~
~~public class MyEndpoint {~~
~~    @ReadOperation~~
~~    public String get(@Nullable String name) { ... }~~
~~}~~
import org.jspecify.annotations.Nullable;

@Endpoint(id = "my-endpoint")
public class MyEndpoint {
    @ReadOperation
    public String get(@Nullable String name) { ... }
}
```

HOW TO FIX

1. **Replace org.springframework.lang.Nullable with org.jspecify.annotations.Nullable.** Update the import on all Actuator endpoint parameters. The annotation name is the same — only the package changes. No logic changes required. JSpecify is available on the classpath via Spring Framework 7.0.

✓ Check:
mvn compile — no cannot find symbol or binding failure for @Nullable on endpoint params

TIER 2 · WON'T RUN | IMPACT: S | spring-boot 3.5 → 4.0

AspectJ Weaving Required for @Observed

Spring Boot 4.0 requires AspectJ-based weaving for the @Observed annotation to produce spans and metrics. Without the aspect configured, @Observed silently does nothing.

WHAT YOU'LL SEE

```
# No startup error — this is a silent behaviour change.
# Detected when @Observed methods produce no observations:

@Observed(name = "order.process")
public Order processOrder(OrderRequest request) { ... }

# Expected in metrics:
$ curl -s http://localhost:8080/actuator/metrics/order.process
{"name":"order.process","measurements":[{"statistic":"COUNT","value":42.0}]}

# Actual after upgrade:
$ curl -s http://localhost:8080/actuator/metrics/order.process
{"status":404,"error":"Not Found"}

# No metric registered. No spans created. Method executes normally
# but observability is completely missing.
```

WHAT CHANGED

In Spring Boot 3.x, `@Observed` worked via Spring AOP proxies with auto-configuration that registered the `ObservedAspect` bean automatically. In Spring Boot 4.0, the `ObservedAspect` auto-configuration requires explicit AspectJ weaving support. If your project does not have the AspectJ weaving dependency and configuration, the `@Observed` annotation is silently ignored.

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing</artifactId>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

```
// ObservedAspect was auto-configured in Boot 3.x
@Configuration
public class ObservabilityConfig {
    @Bean
    public ObservedAspect observedAspect(ObservationRegistry registry) {
        return new ObservedAspect(registry);
    }
}
```

```
# proxy-based AOP was sufficient
spring.aop.auto=true
```

HOW TO FIX

1. **Add AspectJ weaver dependency.** Add `org.aspectj:aspectjweaver` to your dependencies. Spring Boot's dependency management provides the version. This enables the AspectJ-based processing of `@Observed`.
2. **Register ObservedAspect bean explicitly.** If the auto-configuration does not pick up the aspect, register an `ObservedAspect` bean manually in a `@Configuration` class. Pass the `ObservationRegistry` to its constructor.
3. **Verify AOP is enabled.** Ensure `spring.aop.auto=true` (the default) and that you have not disabled AspectJ auto-proxying. Check for `@EnableAspectJAutoProxy` if you have custom AOP configuration.

> ✓ **Check:**
> Custom @Observed aspects fire and metrics appear in /actuator/metrics

Spring Batch Now Uses In-Memory Job Repository by Default

Spring Batch defaults to an in-memory job repository in Boot 4.0. The `BATCH_JOB_EXECUTION` and related tables are no longer created automatically. Add `spring-boot-starter-batch-jdbc` to restore database persistence.

WHAT YOU'LL SEE

```
// Boot 3.5: BATCH_JOB_EXECUTION table created, job persisted.

// Boot 4.0 with only spring-boot-starter-batch:
// No schema created. Query against batch tables fails:
org.springframework.jdbc.BadSqlGrammarException:
  PreparedStatementCallback; bad SQL grammar
  [SELECT JOB_INSTANCE_ID, JOB_NAME from BATCH_JOB_INSTANCE ...];
  Table "BATCH_JOB_INSTANCE" not found

// Or job restart logic silently ignores execution history.
```

WHAT CHANGED

Spring Boot 4.0 introduced `spring-boot-starter-batch-jdbc` as the explicit opt-in for database-backed job execution persistence. `spring-boot-starter-batch` now configures an in-memory `JobRepository` backed by a `ConcurrentHashMap`. The Batch schema DDL is no longer applied automatically unless the JDBC starter is on the classpath.

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

HOW TO FIX

1. **Add spring-boot-starter-batch-jdbc for database-backed persistence.** If your application relies on job execution history, restartability, or the Batch admin tables, add `spring-boot-starter-batch-jdbc` alongside `spring-boot-starter-batch`. This restores the Boot 3.5 behaviour.
2. **Embrace the in-memory default if persistence isn't needed.** If your jobs are stateless or ephemeral — one-shot CLI jobs, test fixtures, data imports — the in-memory repository is correct and removes the need for a datasource entirely. No changes required beyond removing schema initialisation configuration.

> **✓ Check:**
> Batch jobs persist execution history to the database when spring-boot-starter-batch-jdbc is declared

Spring Batch 6 Listener Support Classes Removed (Runtime)

`JobExecutionListenerSupport`, `StepExecutionListenerSupport`, and `ChunkListenerSupport` are removed in Spring Batch 6.0. Dynamic loading via reflection throws `ClassNotFoundException` on Boot 4.0.

WHAT YOU'LL SEE

```
// Dynamic class loading — e.g. framework, XML config, or legacy reflection
Class.forName("org.springframework.batch.core.listener.JobExecutionListenerSupport");

// Boot 3.5 (Batch 5.x): class present, deprecated since 5.0 — loads fine.

// Boot 4.0 (Batch 6.0): class removed.
java.lang.ClassNotFoundException:
  org.springframework.batch.core.listener.JobExecutionListenerSupport
```

WHAT CHANGED

Spring Batch 6.0 (shipped with Spring Boot 4.0) removed three convenience base classes that had been deprecated since Batch 5.0: `JobExecutionListenerSupport`, `StepExecutionListenerSupport`, and `ChunkListenerSupport`. These classes provided empty default implementations of listener methods, a pattern made redundant by Java 8 interface default methods.

```
~~public class MyJobListener extends JobExecutionListenerSupport {~~
~~    @Override~~
~~    public void afterJob(JobExecution jobExecution) {~~
~~        // only override what you need~~
~~    }~~
~~}~~
public class MyJobListener implements JobExecutionListener {
    @Override
    public void afterJob(JobExecution jobExecution) {
        // implement the interface directly
    }
}
```

HOW TO FIX

1. **Implement the listener interface directly.** Replace `extends JobExecutionListenerSupport` with `implements JobExecutionListener`. Do the same for `StepExecutionListenerSupport` → `StepExecutionListener` and `ChunkListenerSupport` → `ChunkListener`. You only need to implement the methods you actually use — all others have default no-op implementations on the interface.

✓ Check:
JobExecutionListenerSupport, StepExecutionListenerSupport, and ChunkListenerSupport are on the classpath and loadable at runtime

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

Spring Batch 6 Schema Sequence Rename

Spring Batch 6 renamed its database sequences (e.g., BATCH_JOB_SEQ became BATCH_JOB_INSTANCE_SEQ). Existing databases using the old names fail when Batch tries to generate IDs.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
org.springframework.dao.InvalidDataAccessResourceUsageException:
  could not fetch next sequence value
...
Caused by: java.sql.SQLSyntaxErrorException:
  sequence "BATCH_JOB_INSTANCE_SEQ" does not exist
 at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(...)
 ...
org.springframework.batch.core.launch.JobLaunchException:
  Could not launch job 'importCustomersJob'.
  Failed to create new JobInstance for job [importCustomersJob]
```

WHAT CHANGED

Spring Batch 6 renamed the database sequences used for generating IDs: `BATCH_JOB_SEQ` became `BATCH_JOB_INSTANCE_SEQ`, `BATCH_JOB_EXECUTION_SEQ` stayed the same, and `BATCH_STEP_EXECUTION_SEQ` stayed the same. The metadata table structures also changed. Applications with existing Batch schema from Spring Batch 5 will fail when the framework tries to use the new sequence names.

```
-- Old sequences from Spring Batch 5
-- Rename sequences to match Spring Batch 6 expectations
ALTER SEQUENCE BATCH_JOB_SEQ RENAME TO BATCH_JOB_INSTANCE_SEQ;
```

```
-- Old sequences from Spring Batch 5
-- MySQL uses tables for sequence emulation
RENAME TABLE BATCH_JOB_SEQ TO BATCH_JOB_INSTANCE_SEQ;
```

```
# no Batch schema configuration
# Point Batch to legacy sequence names while you plan migration
spring.batch.jdbc.isolation-level-for-create=ISOLATION_DEFAULT
```

HOW TO FIX

1. **Run a schema migration script.** Create a Flyway or Liquibase migration that renames the sequences. For PostgreSQL: `ALTER SEQUENCE BATCH_JOB_SEQ RENAME TO BATCH_JOB_INSTANCE_SEQ;`. For MySQL, rename the emulation tables. Apply this before deploying the upgraded application.
2. **Let Spring Batch recreate the schema.** If your Batch job data is disposable (e.g., dev/staging), set `spring.batch.jdbc.initialize-schema=always` to let Spring Batch drop and recreate the metadata tables with the new names. Do not use this in production.
3. **Use the migration SQL from Spring Batch.** The Spring Batch 6 migration guide provides official SQL scripts for each database platform. Use those as the basis for your migration rather than writing them by hand.

✓ Check:
Spring Batch job launches with no BATCH_ table not found errors

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

Controller & View Spans Disabled by Default

Spring Boot 4.0 disables controller and view rendering spans by default in Micrometer tracing. Dashboards, alerts, and SLOs relying on these spans silently lose data.

WHAT YOU'LL SEE

```
# No startup error — this is a silent behaviour change.
# Detected when spans are missing in your observability platform:

$ curl -s http://localhost:16686/api/traces?service=my-app | jq
'.data[0].spans[].operationName'
"HTTP GET /api/orders"
# Previously also showed:
# "OrderController#getOrders"
# "orders :: view rendering"
# These spans are now GONE.

# Grafana alert fires:
[FIRING] No controller-level spans for service my-app in last 15m
# Or SLO dashboard shows 0% data for controller latency percentiles.
```

WHAT CHANGED

Spring Boot 4.0 changed the default for `management.observations.http.server.requests.name` and disabled the creation of child spans for Spring MVC controller method invocations and view rendering. Previously, Micrometer Tracing created spans for both the HTTP request and the controller method execution. Now only the HTTP server request span is created by default.

```
# default: controller spans were enabled
management.observations.http.server.requests.controller.enabled=true
management.observations.http.server.requests.view.enabled=true
```

```
# default: controller spans were enabled
management:
  observations:
    http:
      server:
        requests:
          controller:
            enabled: true
          view:
            enabled: true
```

HOW TO FIX

1. **Re-enable controller spans.** Add `management.observations.http.server.requests.controller.enabled=true` to your `application.properties`. Do the same for `view.enabled` if you need view rendering spans. This restores the Spring Boot 3.x behaviour.
2. **Update dashboards and alerts.** If you decide to accept the new default (no controller spans), update your Grafana dashboards, alert rules, and SLO definitions to use the HTTP server request span instead of controller-level spans. The HTTP span still contains the handler method in its attributes.
3. **Use @Observed for targeted spans.** Instead of blanket controller spans, annotate specific controller methods with `@Observed` to get spans only where you need them. This is more efficient than re-enabling all controller spans globally.

✔ **Check:**
Trace dashboard shows controller-level spans after re-enabling

Deprecated RestTemplateBuilder APIs Removed

Deprecated `RestTemplateBuilder` methods removed in Boot 4.0. Code that called deprecated builder methods compiles on 3.5 (with warnings) and fails to compile on 4.0. Migrate to `RestClient`.

WHAT YOU'LL SEE

```
// Calling a deprecated builder method
RestTemplate rt = builder
    .additionalMessageConverters(new MappingJackson2HttpMessageConverter())
    .build();

// Boot 3.5: compiles with deprecation warning, runs fine.

// Boot 4.0: compile error.
error: cannot find symbol
  symbol: method additionalMessageConverters(MappingJackson2HttpMessageConverter)
```

WHAT CHANGED

Spring Boot 3.x deprecated several `RestTemplateBuilder` methods to signal the migration path to `RestClient` (introduced in Spring Framework 6.1). Boot 4.0 removes these deprecated methods entirely, following the standard deprecation-then-removal cycle.

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder
        .additionalMessageConverters(new MappingJackson2HttpMessageConverter())
        .build();
}
@Bean
public RestClient restClient() {
    return RestClient.builder()
        .messageConverters(converters →
            converters.add(new MappingJackson2HttpMessageConverter()))
        .build();
}
```

HOW TO FIX

1. **Migrate to RestClient.** `RestClient` is the modern replacement. It offers a similar fluent builder API and is available from Spring Framework 6.1 onward (Spring Boot 3.2+). Migrating removes the deprecated call and future-proofs the HTTP client code.
2. **If RestTemplate must be kept, remove the deprecated builder calls.** Construct a `RestTemplate` directly (via constructor or with a plain `RestTemplateBuilder.build()`) and configure message converters via `restTemplate.getMessageConverters().add(...)` after construction.

✓ Check:
Code using deprecated RestTemplateBuilder methods fails to compile on Boot 4.0

TIER 2 · WON'T RUN IMPACT: M spring-boot 3.5 → 4.0

CascadeType.SAVE_UPDATE Removed in Hibernate 7

Hibernate 7 removed the proprietary CascadeType.SAVE_UPDATE. Entities using this cascade type fail at startup or throw MappingException when the SessionFactory is built.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
org.springframework.beans.factory.BeanCreationException:
  Error creating bean with name 'entityManagerFactory'
...
Caused by: org.hibernate.MappingException:
  Unrecognized legacy cascade style: save-update
 at org.hibernate.boot.model.internal.EntityBinder.bindEntityAnnotation(...)
 at org.hibernate.boot.model.internal.AnnotationBinder.bindClass(...)
 ...
APPLICATION FAILED TO START
---
Description:
Failed to initialize JPA EntityManagerFactory:
Unrecognized legacy cascade style: save-update
```

WHAT CHANGED

`org.hibernate.annotations.CascadeType.SAVE_UPDATE` was a Hibernate-specific extension to JPA cascade types. It triggered cascade behaviour on Hibernate's proprietary `Session.saveOrUpdate()` method. Hibernate 7 removed both `saveOrUpdate()` and the corresponding cascade type. Code using the Hibernate-specific `@Cascade` annotation with `SAVE_UPDATE` breaks at entity mapping time.

```
~~import org.hibernate.annotations.Cascade;~~
~~import org.hibernate.annotations.CascadeType;~~

~~@OneToMany(mappedBy = "order")~~
~~@Cascade(CascadeType.SAVE_UPDATE)~~
~~private List<OrderItem> items;~~
@OneToMany(mappedBy = "order", cascade = {
    javax.persistence.CascadeType.PERSIST,
    javax.persistence.CascadeType.MERGE
})
private List<OrderItem> items;
```

```
~~@OneToMany(mappedBy = "parent")~~
~~@Cascade({CascadeType.SAVE_UPDATE, CascadeType.DELETE})~~
~~private Set<Child> children;~~
@OneToMany(mappedBy = "parent", cascade = {
    CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE
})
private Set<Child> children;
```

HOW TO FIX

1. **Replace with JPA standard cascades.** Swap `CascadeType.SAVE_UPDATE` for the combination of `CascadeType.PERSIST` and `CascadeType.MERGE`. These two JPA-standard cascades cover the same persist-or-update behaviour that `SAVE_UPDATE` provided.
2. **Remove @Cascade if redundant.** If your entity already has `cascade = CascadeType.ALL` on the JPA annotation, the Hibernate `@Cascade` is redundant. Delete it entirely.

✓ **Check:**

mvn compile — no CascadeType.SAVE_UPDATE errors

Version-Specific Hibernate Dialects Removed

Hibernate 7 removed version-specific dialect classes like MySQL57Dialect and PostgreSQL95Dialect. Applications that explicitly configure a dialect fail at startup with ClassNotFoundException.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
org.springframework.beans.factory.BeanCreationException:
  Error creating bean with name 'entityManagerFactory'
...
Caused by: java.lang.ClassNotFoundException:
  org.hibernate.dialect.MySQL57Dialect
 at java.base/
jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:641)
 at java.base/
jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:188)
 ...
APPLICATION FAILED TO START
---
Description:
Failed to configure a DataSource: Hibernate dialect
'org.hibernate.dialect.MySQL57Dialect' could not be found.
```

WHAT CHANGED

Hibernate 7 removed all version-specific dialect classes (`MySQL57Dialect`, `MySQL8Dialect`, `PostgreSQL95Dialect`, `PostgreSQL10Dialect`, `Oracle12cDialect`, `SQLServer2016Dialect`, etc.). Only the base dialect classes remain (`MySQLDialect`, `PostgreSQLDialect`, `OracleDialect`, `SQLServerDialect`). Hibernate now auto-detects the database version from the JDBC connection and configures features accordingly.

```
~~spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57Dialect~~
# Remove explicit dialect — Hibernate auto-detects from JDBC connection
```

```
~~spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL95Dialect~~
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

```
~~<property name="hibernate.dialect" value="org.hibernate.dialect.Oracle12cDialect"/>~~
<property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect"/>
```

HOW TO FIX

1. **Remove explicit dialect (recommended).** Delete the `spring.jpa.properties.hibernate.dialect` property entirely. Hibernate 7 auto-detects the correct dialect from the JDBC connection metadata. This is the cleanest fix and the one the Hibernate team recommends.
2. **Switch to base dialect class.** If you need an explicit dialect for testing or CI (e.g., H2 in-memory), use the unversioned base class: `MySQLDialect`, `PostgreSQLDialect`, `OracleDialect`, or `SQLServerDialect`.

> ✓ **Check:**
> App starts with no DialectResolutionInfoMissingException

TIER 2 · WON'T RUN | IMPACT: S | spring-boot 3.5 → 4.0

Hibernate Untyped Join Query Rejected at Runtime

Hibernate 7 rejects untyped join queries with no explicit SELECT clause. A query that silently returned Object[] on Boot 3.5 throws SemanticException at runtime on Boot 4.0.

WHAT YOU'LL SEE

```
// Query without explicit result type
session.createQuery("from Product p join p.category c").getResultList();

// Boot 3.5 (Hibernate 6.x): runs fine, returns List<Object[]>

// Boot 4.0 (Hibernate 7.x): throws at runtime
org.hibernate.query.SemanticException:
  Query has no explicit SELECT and multiple query roots are defined.
  An explicit SELECT clause is required when multiple query roots exist
  or when the query contains a JOIN.
```

WHAT CHANGED

Hibernate 7.0 tightened validation of HQL queries. A query like `from Product p join p.category c` has two implicit query roots (Product and the joined Category). Without an explicit `SELECT` clause or result type, the return type is ambiguous. Hibernate 6.x silently resolved this as `Object[]`; Hibernate 7.0 rejects it at parse time with a `SemanticException`.

```
~~session.createQuery("from Product p join p.category c")~~
~~        .getResultList();~~
session.createQuery(
    "SELECT p FROM Product p JOIN p.category c",
    Product.class
).getResultList();
```

```
~~session.createQuery("from Product p join p.category c")~~
~~        .getResultList();~~
session.createQuery(
    "SELECT p, c FROM Product p JOIN p.category c",
    Object[].class
).getResultList();
```

HOW TO FIX

1. **Add explicit SELECT clause and result type (recommended).** All `Session.createQuery(String)` calls with join queries must include an explicit `SELECT` clause. Pass the expected result class as the second argument: `createQuery(hql, Entity.class)`.
2. **Search for bare 'from' queries.** Grep for `createQuery("from` and `createQuery('from` in your codebase. Any query that starts with `from` and contains a join is at risk.

> ✓ **Check:**
> HQL join queries without explicit SELECT or result type execute without SemanticException

TIER 2 · WON'T RUN IMPACT: M spring-boot 3.5 → 4.0

JacksonException No Longer Extends IOException

Jackson 3.0 changed the exception hierarchy so JacksonException extends RuntimeException instead of IOException. Existing catch blocks silently miss Jackson errors, and throws clauses become lies.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
com.example.OrderController: Unhandled exception in /api/orders
tools.jackson.core.exc.StreamReadException: Unexpected character ('x' (code 120)):
  expected a valid value (JSON String, Number, Array, Object or token)
 at [Source: (String)"x]"; line: 1, column: 2]
 at tools.jackson.core.json.JsonParser._reportUnexpectedChar(JsonParser.java:...)
 ...
Caused by: tools.jackson.core.JacksonException (not an IOException!)
---
HTTP 500 Internal Server Error
{"timestamp":"2026-04-15T10:22:03","status":500,
 "error":"Internal Server Error","path":"/api/orders"}
```

WHAT CHANGED

In Jackson 2.x, `JacksonException` extended `java.io.IOException`, so any `catch (IOException e)` block would also catch malformed-JSON errors. In Jackson 3.0, `JacksonException` extends `RuntimeException`. Catch blocks for `IOException` no longer intercept Jackson parsing failures, and methods that declared `throws IOException` for Jackson work now throw unchecked exceptions instead.

```
try {
    Order order = objectMapper.readValue(body, Order.class);
} catch (IOException e) {
    return ResponseEntity.badRequest().body("Invalid JSON");
}
try {
    Order order = objectMapper.readValue(body, Order.class);
} catch (JacksonException e) {
    return ResponseEntity.badRequest().body("Invalid JSON");
}
```

```
@ExceptionHandler(IOException.class)
public ResponseEntity<String> handleJsonError(IOException ex) {
@ExceptionHandler(JacksonException.class)
public ResponseEntity<String> handleJsonError(JacksonException ex) {
```

```
public Order parseOrder(String json) throws IOException {
public Order parseOrder(String json) {
```

HOW TO FIX

1. **Search for catch blocks.** Grep for `catch (IOException` and `catch (JsonProcessingException`. Replace with `catch (JacksonException` where the intent is to handle Jackson parse failures. Keep `IOException` catches for genuine I/O work (file reads, sockets).

2. **Update @ExceptionHandler advice.** If your `@RestControllerAdvice` catches `IOException` to return 400 on bad JSON, add a separate handler for `JacksonException`. Otherwise those errors now surface as unhandled 500s.

3. **Clean up throws clauses.** Methods that declared `throws IOException` only for Jackson can drop the clause entirely. The compiler won't complain, but the stale declaration misleads readers.

> **✓ Check:**
> mvn compile — no JsonMappingException symbol errors

TIER 2 · WON'T RUN | IMPACT: S | spring-boot 3.5 → 4.0

javax.annotation Removed

Spring Framework 7.0 stopped processing javax.annotation lifecycle annotations. @PostConstruct and @PreDestroy methods silently never fire. The code compiles. Nothing tells you something is wrong.

OpenRewrite

WHAT YOU'LL SEE

```
// No compile error — javax.annotation-api is still downloadable.
// Failure is silent at runtime:

@PostConstruct
public void init() {
    this.ready = true;  // never called on Boot 4.0
}

// Later in a test or endpoint:
java.lang.NullPointerException: Cannot invoke method because 'this.cache' is null
// or simply: assertions on initialisation state fail unexpectedly
```

WHAT CHANGED

The `javax.annotation:javax.annotation-api` dependency is no longer on the Spring Boot classpath. Annotations `@javax.annotation.PostConstruct`, `@javax.annotation.PreDestroy`, and `@javax.annotation.Resource` are unavailable. The Jakarta equivalent is `jakarta.annotation:jakarta.annotation-api`, managed by the Boot BOM. See also: javax-inject-removed.

```
~~import javax.annotation.PostConstruct;~~
~~import javax.annotation.PreDestroy;~~
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
```

```
~~<dependency>~~
~~    <groupId>javax.annotation</groupId>~~
~~    <artifactId>javax.annotation-api</artifactId>~~
~~    <version>1.3.2</version>~~
~~</dependency>~~
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
</dependency>
```

HOW TO FIX

1. **Switch to jakarta.annotation.** Replace `javax.annotation` imports with `jakarta.annotation`. The annotations are identical — only the package name changed. The `jakarta.annotation-api` artifact is managed by the Spring Boot BOM so no version is needed.
2. **Remove explicit dependency version.** If you declared `javax.annotation-api` explicitly in your POM, replace it with `jakarta.annotation-api` and omit the version — Boot's BOM manages it.

> ✓ **Check:**
> App starts and @PostConstruct methods fire — confirm via log output or assert that initialisation state is set after context loads

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

javax.inject Removed

Spring Framework 7.0 stopped processing javax.inject annotations. @Inject fields silently stay null. @Named beans may not be registered. The code compiles and the app starts. Nothing tells you it's broken.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
// No compile error — javax.inject:javax.inject still downloadable.
// Failure is silent at runtime:

@Named("orderService")
public class OrderService {
    @Inject
    private OrderRepository repo;  // null on Boot 4.0 — never injected
}

// At call time:
java.lang.NullPointerException: Cannot invoke
  "com.example.OrderRepository.findById(Long)"
  because "this.repo" is null
```

WHAT CHANGED

The `javax.inject:javax.inject` dependency is no longer on the Spring Boot classpath. Annotations like `@javax.inject.Inject`, `@javax.inject.Named`, and `@javax.inject.Singleton` are unavailable. The Jakarta equivalent is `jakarta.inject:jakarta.inject-api`.

```
~~import javax.inject.Inject;~~
~~import javax.inject.Named;~~
import jakarta.inject.Inject;
import jakarta.inject.Named;
```

```
~~<dependency>~~
~~    <groupId>javax.inject</groupId>~~
~~    <artifactId>javax.inject</artifactId>~~
~~    <version>1</version>~~
~~</dependency>~~
<dependency>
    <groupId>jakarta.inject</groupId>
    <artifactId>jakarta.inject-api</artifactId>
</dependency>
```

```
~~@Named("orderService")~~
~~public class OrderService {~~
~~    @Inject~~
~~    private OrderRepository repo;~~
@Service("orderService")
public class OrderService {
    @Autowired
    private OrderRepository repo;
```

HOW TO FIX

1. **Switch to jakarta.inject.** Replace `javax.inject` imports with `jakarta.inject`. The annotations are identical — only the package name changed. The `jakarta.inject-api` artifact is managed by the Spring Boot BOM.

2. **Switch to Spring annotations (recommended).** Replace `@Inject` with `@Autowired` and `@Named` with `@Component` / `@Service` / `@Qualifier`. This removes the dependency on the inject API entirely.

> **✓ Check:**
> App starts and @Inject fields are non-null — confirm via a test that autowires a bean registered with @Named

TIER 2 · WON'T RUN IMPACT: M spring-boot 3.5 → 4.0

@MockBean / @SpyBean Removed

Spring Boot 4.0 removed the @MockBean and @SpyBean annotations. Tests using them fail at runtime with annotation-not-found errors. Replace with @MockitoBean and @MockitoSpyBean.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
java.lang.NoClassDefFoundError:
  org/springframework/boot/test/mock/mockito/MockBean
 at com.example.OrderServiceTest.<clinit>(OrderServiceTest.java:18)
Caused by: java.lang.ClassNotFoundException:
  org.springframework.boot.test.mock.mockito.MockBean
---
java.lang.IllegalStateException: Failed to load ApplicationContext for
  [MergedContextConfiguration@4a5905...
  testClass = com.example.OrderServiceTest]
...
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0
```

WHAT CHANGED

The `@MockBean` and `@SpyBean` annotations from `org.springframework.boot.test.mock.mockito` were deprecated in Spring Boot 3.4 and removed in 4.0. They are replaced by `@MockitoBean` and `@MockitoSpyBean` from `org.springframework.test.context.bean.override.mockito`, which live in Spring Framework 7 itself rather than Spring Boot.

```
~~import org.springframework.boot.test.mock.mockito.MockBean;~~
import org.springframework.test.context.bean.override.mockito.MockitoBean;
```

```
~~import org.springframework.boot.test.mock.mockito.SpyBean;~~
import org.springframework.test.context.bean.override.mockito.MockitoSpyBean;
```

```
~~@MockBean~~
~~private OrderRepository orderRepository;~~

~~@SpyBean~~
~~private NotificationService notificationService;~~
@MockitoBean
private OrderRepository orderRepository;

@MockitoSpyBean
private NotificationService notificationService;
```

HOW TO FIX

1. **Find-and-replace annotations.** Replace `@MockBean` with `@MockitoBean` and `@SpyBean` with `@MockitoSpyBean`. Update the imports from `org.springframework.boot.test.mock.mockito` to `org.springframework.test.context.bean.override.mockito`.
2. **OpenRewrite recipe.** The Spring Boot 4.0 OpenRewrite migration recipe handles this rename automatically across the entire test codebase.
3. **Check for @MockBean attributes.** If you used `@MockBean(name = "...")` or `@MockBean(classes = ...)`, verify the equivalent attributes exist on `@MockitoBean`. The attribute names may differ slightly.

✓ **Check:**

mvn test — no @MockBean compile errors

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

MockitoTestExecutionListener Removed

Spring Boot 4.0 removes MockitoTestExecutionListener. @Mock fields in @SpringBootTest tests stay null and throw NullPointerException unless @ExtendWith(MockitoExtension.class) is added.

WHAT YOU'LL SEE

```
@SpringBootTest
class MyTest {
    @Mock
    private MyService myService;

    @Test
    void test() {
        // Boot 3.5: myService is initialised → passes
        // Boot 4.0: myService is null → NullPointerException
        when(myService.greet()).thenReturn("Hello");
    }
}

java.lang.NullPointerException: Cannot invoke method on null object
  at com.example.MyTest.test(MyTest.java:15)
```

WHAT CHANGED

`MockitoTestExecutionListener` was a Spring Boot-specific bridge that triggered Mockito's annotation processing inside Spring's test lifecycle. Spring Boot 4.0 removed this listener, aligning with the standard JUnit 5 approach of using `@ExtendWith(MockitoExtension.class)` or `MockitoAnnotations.openMocks(this)` in a `@BeforeEach` setup method.

```
~~@SpringBootTest~~
~~class MyTest {~~
~~    @Mock~~
~~    private MyService myService;~~
~~}~~
@SpringBootTest
@ExtendWith(MockitoExtension.class)
class MyTest {
    @Mock
    private MyService myService;
}
```

```
~~@SpringBootTest~~
~~class MyTest {~~
~~    @Mock~~
~~    private MyService myService;~~
~~}~~
@SpringBootTest
class MyTest {
    @MockitoBean
    private MyService myService;  // registered in the Spring context
}
```

HOW TO FIX

1. **Add @ExtendWith(MockitoExtension.class) to the test class.** This is the standard JUnit 5 way to enable Mockito annotation processing. It works alongside `@SpringBootTest` and requires no additional dependency — Mockito's JUnit 5 extension is already on the test classpath via `spring-boot-starter-test`.

2. **Use @MockitoBean instead of @Mock (if the mock belongs in the context).** If the mock is being injected into a Spring bean under test, use `@MockitoBean` (Boot 4.0's replacement for `@MockBean`). This registers the mock in the Spring application context rather than just the test instance.

> ✓ **Check:**
> @Mock fields in @SpringBootTest classes are initialised correctly

TIER 2 · WON'T RUN IMPACT: M spring-boot 3.5 → 4.0

Modular Auto-Configuration

Spring Boot 4.0 split the monolithic auto-configuration jar into modules. Beans you got for free (ObjectMapper, Validator) are now missing unless you add the explicit starter. The app starts. NoSuchBeanDefinitionException hits at first use.

WHAT YOU'LL SEE

```
// App compiles and starts. Failure at first injection or endpoint call:

org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type
  'com.fasterxml.jackson.databind.ObjectMapper' available

// Or from a @RestController trying to serialise a response:
org.springframework.http.converter.HttpMessageNotWritableException:
  No converter for [class com.example.OrderDto]
```

WHAT CHANGED

The `spring-boot-autoconfigure` jar was a single monolithic module that bundled auto-configuration for every Spring Boot feature. In Boot 4.0 it is split into focused modules — one per feature area. Each module's auto-configuration only activates when its corresponding starter is explicitly present. Adding `spring-boot-starter-web` no longer implicitly enables Jackson, validation, or actuator auto-configuration.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!— Now required explicitly — no longer implicit via web starter —>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

HOW TO FIX

1. **Audit and add missing starters.** Run `mvn dependency:tree` and compare what was transitive on Boot 3.5 with what your Boot 4.0 build pulls in. Add an explicit `spring-boot-starter-*` dependency for every feature your application uses: `spring-boot-starter-json` for Jackson, `spring-boot-starter-validation` for Bean Validation, `spring-boot-starter-actuator` for Micrometer metrics.

2. **Watch for silent failures in @RestController endpoints.** If a controller returns an object that Jackson used to serialise automatically, the endpoint now returns `406 Not Acceptable` or throws `HttpMessageNotWritableException` rather than an obvious startup error. Add `spring-boot-starter-json` to restore the message converter.

✓ Check:

App starts and ObjectMapper, Validator, and other auto-configured beans are present — confirm with a test that autowires each one

TIER 2 · WON'T RUN IMPACT: L spring-boot 3.5 → 4.0

OAuth 2.0 Password Grant Completely Removed

Spring Security 7 removed all support for the OAuth 2.0 Resource Owner Password Credentials grant. Applications that use password grant for authentication fail at startup or return 401s at runtime.

WHAT YOU'LL SEE

```
org.springframework.beans.factory.BeanCreationException:
  Error creating bean with name 'securityFilterChain'
...
Caused by: java.lang.IllegalArgumentException:
  The grant type 'password' is not supported.
  Supported grant types are: [authorization_code, client_credentials,
  refresh_token, urn:ietf:params:oauth:grant-type:device_code,
  urn:ietf:params:oauth:grant-type:token-exchange]
 at
org.springframework.security.oauth2.client.registration.ClientRegistration$Builder.build
(...)
---
APPLICATION FAILED TO START
```

WHAT CHANGED

The OAuth 2.0 Resource Owner Password Credentials grant (password grant) was deprecated in Spring Security 5.x and has been fully removed in Spring Security 7 (shipped with Spring Boot 4.0). The `ClientRegistration` builder rejects `authorization-grant-type: password`, and the `ResourceOwnerPasswordResourceDetails` class no longer exists.

```
~~spring:~~
~~  security:~~
~~    oauth2:~~
~~      client:~~
~~        registration:~~
~~          my-api:~~
~~            authorization-grant-type: password~~
~~            client-id: my-client~~
~~            client-secret: secret~~
~~        provider:~~
~~          my-api:~~
~~            token-uri: https://auth.example.com/oauth/token~~
spring:
  security:
    oauth2:
      client:
        registration:
          my-api:
            authorization-grant-type: authorization_code
            client-id: my-client
            client-secret: secret
            scope: openid,profile
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
        provider:
          my-api:
            issuer-uri: https://auth.example.com
```

```
~~ClientRegistration.withRegistrationId("legacy")~~
~~    .authorizationGrantType(AuthorizationGrantType.PASSWORD)~~
~~    .tokenUri("https://auth.example.com/oauth/token")~~
~~    .clientId("my-client")~~
~~    .clientSecret("secret")~~
```

```
    .build();
ClientRegistration.withRegistrationId("legacy")
    .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
    .authorizationUri("https://auth.example.com/authorize")
    .tokenUri("https://auth.example.com/oauth/token")
    .redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")
    .clientId("my-client")
    .clientSecret("secret")
    .build();
```

HOW TO FIX

1. **Migrate to Authorization Code + PKCE.** Replace the password grant with Authorization Code flow. For browser-based apps, use Authorization Code with PKCE. This is the recommended approach and requires changes to both the client application and possibly the authorization server configuration.
2. **Use Client Credentials for service-to-service.** If the password grant was used for machine-to-machine communication (no actual user), switch to the Client Credentials grant. This does not require user interaction.
3. **Custom token exchange as a last resort.** If you must support legacy systems that send username/password, implement a custom token exchange endpoint on your authorization server that accepts credentials and issues tokens. This keeps the anti-pattern in one controlled place while you plan a proper migration.

> ✓ **Check:**
> Token endpoint returns 200 for auth code flow (not password grant)

OpenSAML 4 Support Removed

Spring Security 7 dropped OpenSAML 4 support and requires OpenSAML 5. Applications using SAML 2.0 login fail at startup with ClassNotFoundException for OpenSAML 4 classes.

WHAT YOU'LL SEE

```
org.springframework.beans.factory.BeanCreationException:
  Error creating bean with name 'securityFilterChain'
...
Caused by: java.lang.NoClassDefFoundError:
  org/opensaml/saml/saml2/core/impl/AuthnRequestMarshaller
 at org.springframework.security.saml2.provider.service.registration
  .OpenSamlRelyingPartyRegistrationBuilderFactory.<clinit>(...)
Caused by: java.lang.ClassNotFoundException:
  org.opensaml.saml.saml2.core.impl.AuthnRequestMarshaller
 at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(...)
---
APPLICATION FAILED TO START
```

WHAT CHANGED

Spring Security 7 removed all OpenSAML 4 compatibility classes and now requires OpenSAML 5 exclusively. The internal SAML support classes (`OpenSaml4AuthenticationProvider`, `OpenSaml4AuthenticationRequestResolver`, etc.) were replaced with OpenSAML 5 equivalents. OpenSAML 5 changed package structures and removed the marshaller/unmarshaller split.

```
~~<dependency>~~
~~    <groupId>org.opensaml</groupId>~~
~~    <artifactId>opensaml-saml-api</artifactId>~~
~~    <version>4.3.2</version>~~
~~</dependency>~~
~~<dependency>~~
~~    <groupId>org.opensaml</groupId>~~
~~    <artifactId>opensaml-saml-impl</artifactId>~~
~~    <version>4.3.2</version>~~
~~</dependency>~~
<dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-saml-api</artifactId>
    <version>5.1.2</version>
</dependency>
<dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-saml-impl</artifactId>
    <version>5.1.2</version>
</dependency>
```

```
~~import org.springframework.security.saml2.provider.service.authentication~~
~~    .OpenSaml4AuthenticationProvider;~~

~~OpenSaml4AuthenticationProvider provider =~~
~~    new OpenSaml4AuthenticationProvider();~~
import org.springframework.security.saml2.provider.service.authentication
    .OpenSaml5AuthenticationProvider;

OpenSaml5AuthenticationProvider provider =
    new OpenSaml5AuthenticationProvider();
```

```
~~OpenSaml4AuthenticationProvider provider =~~
~~    new OpenSaml4AuthenticationProvider();~~
~~provider.setResponseValidator(responseToken -> {~~
~~    Saml2ResponseValidatorResult result =~~
~~        OpenSaml4AuthenticationProvider~~
~~            .createDefaultResponseValidator()~~
~~            .convert(responseToken);~~
OpenSaml5AuthenticationProvider provider =
    new OpenSaml5AuthenticationProvider();
provider.setResponseValidator(responseToken -> {
    Saml2ResponseValidatorResult result =
        OpenSaml5AuthenticationProvider
            .createDefaultResponseValidator()
            .convert(responseToken);
```

HOW TO FIX

1. **Upgrade OpenSAML dependencies to 5.x.** Update your OpenSAML dependencies to version 5.1.x. If you use Spring Boot's dependency management, the managed version will already be OpenSAML 5 in Spring Boot 4.0. Remove any explicit version overrides pinning to 4.x.
2. **Rename Spring Security SAML classes.** Replace all references to `OpenSaml4*` classes with their `OpenSaml5*` equivalents. The main classes are `OpenSaml5AuthenticationProvider` and `OpenSaml5AuthenticationRequestResolver`.
3. **Review custom SAML processing code.** If you directly use OpenSAML APIs (marshallers, unmarshallers, builders), review the OpenSAML 5 migration guide. The marshaller/unmarshaller split was removed and the object provider model changed.

> ✓ **Check:**
> SAML login flow completes with OpenSAML 5 libraries

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

OTLP/HTTP Now Primary Export Protocol

Spring Boot 4.0 changed the default OTLP transport from gRPC to HTTP/protobuf. Applications exporting to gRPC-only collectors fail silently or get connection refused errors at runtime.

WHAT YOU'LL SEE

```
WARN [otel.exporter] io.opentelemetry.exporter.otlp.http.OtlpHttpSpanExporter:
  Failed to export spans. Server responded with HTTP status 404.
  url=http://otel-collector:4318/v1/traces
---
# Or if collector only exposes gRPC port 4317:
WARN [otel.exporter] io.opentelemetry.exporter.otlp.http.OtlpHttpSpanExporter:
  Failed to export spans.
  java.net.ConnectException: Connection refused
  target=http://otel-collector:4318/v1/traces
---
# Metrics and traces silently stop arriving in your backend.
# Application continues running but observability data is lost.
```

WHAT CHANGED

Spring Boot 4.0 changed the default OTLP export transport from gRPC (port 4317) to HTTP/protobuf (port 4318). The default endpoint changed from `http://localhost:4317` to `http://localhost:4318/v1/traces` (and `/v1/metrics`, `/v1/logs`). Applications that relied on the default gRPC transport now send data to the wrong port and protocol.

```
# default was gRPC on port 4317
management.otlp.tracing.transport=grpc
management.otlp.tracing.endpoint=http://otel-collector:4317
management.otlp.metrics.export.transport=grpc
management.otlp.metrics.export.endpoint=http://otel-collector:4317
```

```
management.otlp.tracing.endpoint=http://otel-collector:4317
management.otlp.tracing.endpoint=http://otel-collector:4318/v1/traces
```

```
otel-collector:
  image: otel/opentelemetry-collector:latest
  ports:
    - "4317:4317"   # gRPC
otel-collector:
  image: otel/opentelemetry-collector:latest
  ports:
    - "4317:4317"   # gRPC
    - "4318:4318"   # HTTP/protobuf
```

HOW TO FIX

1. **Accept HTTP/protobuf (recommended).** Update your OTLP endpoint to use port 4318 and the HTTP path format. Ensure your collector exposes the HTTP receiver. Most modern collectors (OpenTelemetry Collector, Grafana Agent, Datadog Agent) support both protocols.
2. **Explicitly set gRPC transport.** If your collector only supports gRPC, add `management.otlp.tracing.transport=grpc` and set the endpoint to the gRPC port. This preserves the old behaviour.
3. **Update collector configuration.** If using the OpenTelemetry Collector, ensure both the `otlp/grpc` and `otlp/http` receivers are enabled in your collector config. This lets you support both old and new applications during migration.

✔ **Check:**
OTLP exporter sends traces to collector (check collector logs)

TIER 2 · WON'T RUN IMPACT: M spring-boot 3.5 → 4.0

PKCE Mandatory for Confidential OAuth Clients

Spring Security 7 enforces PKCE for all OAuth 2.0 clients, including confidential clients. Authorization requests without a code_challenge are rejected by the authorization server.

WHAT YOU'LL SEE

```
[oauth2-client] o.s.s.oauth2.client.web.OAuth2LoginAuthenticationFilter:
  Authentication request failed:
org.springframework.security.oauth2.core.OAuth2AuthenticationException:
  [invalid_request] PKCE code_challenge is required for this client.
 at org.springframework.security.oauth2.client.oidc.authentication
  .OidcAuthorizationCodeAuthenticationProvider.authenticate(...)
---
HTTP 302 → /login?error=invalid_request
---
Browser shows: "OAuth2 Error: [invalid_request]
PKCE code_challenge is required for this client."
```

WHAT CHANGED

Spring Security 7 now sends PKCE parameters (`code_challenge` and `code_challenge_method`) on every Authorization Code request, for both public and confidential clients. If the authorization server requires PKCE but the client was not sending it (pre-upgrade behaviour for confidential clients), or if the authorization server rejects unexpected PKCE parameters, the flow breaks.

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-app:
            authorization-grant-type: authorization_code
            client-id: my-confidential-client
            client-secret: ${CLIENT_SECRET}
            scope: openid,profile
# Client config stays the same — Spring Security 7 adds PKCE automatically.
# But your authorization server MUST accept PKCE parameters.
spring:
  security:
    oauth2:
      client:
        registration:
          my-app:
            authorization-grant-type: authorization_code
            client-id: my-confidential-client
            client-secret: ${CLIENT_SECRET}
            scope: openid,profile
```

```
"clientId": "my-confidential-client",
"publicClient": false,
"pkceCodeChallengeMethod": ""
"clientId": "my-confidential-client",
"publicClient": false,
"pkceCodeChallengeMethod": "S256"
```

```
// default PKCE behaviour
@Bean
public OAuth2AuthorizationRequestResolver pkceResolver(
        ClientRegistrationRepository repo) {
```

```
    DefaultOAuth2AuthorizationRequestResolver resolver =
        new DefaultOAuth2AuthorizationRequestResolver(
            repo, "/oauth2/authorization");
    // Temporary: remove when auth server supports PKCE
    resolver.setAuthorizationRequestCustomizer(
        OAuth2AuthorizationRequestCustomizers.withoutPkce());
    return resolver;
}
```

HOW TO FIX

1. **Enable PKCE on your authorization server.** Configure your authorization server (Keycloak, Auth0, Okta, Azure AD) to accept PKCE `code_challenge` parameters for confidential clients. Most modern providers already support this — it may just need to be enabled per client. Use `S256` as the challenge method.
2. **Verify no auth server rejects extra parameters.** Some older or custom authorization servers reject unknown parameters. If yours rejects `code_challenge`, you need to update the auth server first, or temporarily disable PKCE enforcement on the Spring Security side using a custom resolver.
3. **Disable PKCE temporarily (not recommended).** If you cannot update the authorization server immediately, register a custom `OAuth2AuthorizationRequestResolver` that strips PKCE parameters. Treat this as technical debt and plan the auth server update.

> ✓ **Check:**
> OAuth2 login completes with PKCE challenge in the auth request

Spring Pulsar Reactive Auto-Configuration Removed

Spring Boot 4.0 dropped reactive Pulsar auto-configuration. `ReactivePulsarTemplate` and `ReactivePulsarClient` beans are no longer available. The app starts but fails on first use.

WHAT YOU'LL SEE

```
// Boot 4.0 startup or runtime failure:
org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type
  'org.springframework.pulsar.reactive.core.ReactivePulsarTemplate' available

// Or auto-wiring failure:
Field reactivePulsarTemplate required a bean of type
  'ReactivePulsarTemplate' that could not be found.
```

WHAT CHANGED

`spring-boot-starter-pulsar-reactive` and the corresponding auto-configuration module were removed from Spring Boot 4.0. The reactive Pulsar client management — including `ReactivePulsarClient`, `ReactivePulsarTemplate`, and `ReactiveMessageListenerContainer` auto-configuration — no longer exists in Boot's auto-configure artifact.

```
<!-- Boot 3.5: auto-configured reactive Pulsar client -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-pulsar-reactive</artifactId>
</dependency>
<!-- Boot 4.0: use imperative starter or configure reactive beans manually -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-pulsar</artifactId>
</dependency>
```

HOW TO FIX

1. **Switch to the imperative Pulsar integration or configure reactive beans manually.** If the reactive API is required, declare the reactive Pulsar beans explicitly in a `@Configuration` class. If reactive messaging is not critical, switch to the imperative `spring-boot-starter-pulsar` which remains auto-configured in Boot 4.0.

> **✓ Check:**
> Reactive Pulsar producers and consumers initialise correctly after removing the dependency on Boot's auto-configuration

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

RestTemplate Auto-Configuration Removed

Spring Boot 4.0 removed RestTemplateAutoConfiguration. Tests and production code that relied on an auto-configured RestTemplateBuilder or RestTemplate bean get NoSuchBeanDefinitionException at startup.

WHAT YOU'LL SEE

```
org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type
  'org.springframework.boot.web.client.RestTemplateBuilder' available:
  expected at least 1 bean which qualifies as autowire candidate.
Dependency annotations:
  {@org.springframework.beans.factory.annotation.Autowired(required=true)}
 at org.springframework.beans.factory.support
  .DefaultListableBeanFactory.raiseNoMatchingBeanException(...)
---
APPLICATION FAILED TO START
---
Description:
Parameter 0 of constructor in com.example.OrderClient required a bean
of type 'org.springframework.boot.web.client.RestTemplateBuilder' that
could not be found.
```

WHAT CHANGED

Spring Boot 4.0 removed `RestTemplateAutoConfiguration`, which previously provided a `RestTemplateBuilder` bean and customizers. The auto-configuration for `RestClient` (the modern replacement) remains. Code that injected `RestTemplateBuilder` or relied on auto-configured `RestTemplate` beans no longer finds them.

```
@Service
public class OrderClient {
    private final RestTemplate restTemplate;

    public OrderClient(RestTemplateBuilder builder) {
        this.restTemplate = builder
            .rootUri("https://api.example.com")
            .build();
    }

    public Order getOrder(Long id) {
        return restTemplate.getForObject(
            "/orders/{id}", Order.class, id);
    }
}
@Service
public class OrderClient {
    private final RestClient restClient;

    public OrderClient(RestClient.Builder builder) {
        this.restClient = builder
            .baseUrl("https://api.example.com")
            .build();
    }

    public Order getOrder(Long id) {
        return restClient.get()
            .uri("/orders/{id}", id)
            .retrieve()
            .body(Order.class);
```

```
    }
}
```

```
@RestClientTest(OrderClient.class)
class OrderClientTest {
    @Autowired
    private MockRestServiceServer server;
    @Autowired
    private OrderClient client;
@RestClientTest(OrderClient.class)
class OrderClientTest {
    @Autowired
    private MockRestServiceServer server;
    @Autowired
    private OrderClient client;
    // Works the same — @RestClientTest supports RestClient
```

HOW TO FIX

1. **Migrate to RestClient (recommended).** Replace `RestTemplate` usage with `RestClient`. Inject `RestClient.Builder` (auto-configured by Spring Boot 4) instead of `RestTemplateBuilder`. The API is fluent and supports the same customizers.
2. **Define your own RestTemplateBuilder bean.** If migration is too large to do immediately, register your own `RestTemplateBuilder` bean in a `@Configuration` class. This restores the old behaviour while you plan the migration.
3. **Use WebClient for reactive code.** If your application is reactive, use `WebClient` instead. It remains fully auto-configured in Spring Boot 4.0.

> **✓ Check:**
> App starts with explicit RestTemplate bean and HTTP calls succeed

TIER 2 · WON'T RUN IMPACT: L spring-boot 3.5 → 4.0

Spring Boot Spock Integration Removed

Spock does not support Groovy 5. Spring Boot 4.0 requires Groovy 5. Spock-based `@SpringBootTest` specifications stop running. Migrate to JUnit 5 or wait for Spock's Groovy 5 release.

WHAT YOU'LL SEE

```
// Spock specification in a Boot 4.0 project:
@SpringBootTest
class MyServiceSpec extends Specification {
    @Autowired MyService service

    def "should do something"() {
        expect: service.doThing() == "result"
    }
}

// Fails at compile or runtime — Spock and Groovy 5 are incompatible.
// Exact error depends on Groovy/Spock versions on the classpath.
```

WHAT CHANGED

Spring Boot 4.0 upgraded its Groovy baseline to Groovy 5. Spock Framework (as of this writing) requires Groovy 4.x or earlier. Spring Boot removed the Spock integration rather than block on Spock's Groovy 5 support. Projects using Spock for Spring-context-dependent tests must either migrate to JUnit 5 or defer the Boot 4.0 upgrade until Spock adds Groovy 5 support.

```
// MyServiceSpec.groovy
@SpringBootTest
class MyServiceSpec extends Specification {
    @Autowired MyService service

    def "should return result"() {
        expect: service.doThing() == "result"
    }
}
// MyServiceTest.java
@SpringBootTest
class MyServiceTest {
    @Autowired MyService service;

    @Test
    void shouldReturnResult() {
        assertThat(service.doThing()).isEqualTo("result");
    }
}
```

HOW TO FIX

1. **Migrate Spock specifications to JUnit 5.** Rewrite Spock-based Spring integration tests using JUnit 5 and AssertJ. The `given/when/then` blocks map to JUnit 5 methods and AssertJ assertions. Data-driven tests (`where:` blocks) translate to `@ParameterizedTest` with `@MethodSource` or `@CsvSource`.

2. **Wait for Spock's Groovy 5 support if migration is impractical.** If the volume of Spock tests makes immediate migration impractical, track the Spock project's Groovy 5 roadmap. Unit tests written with Spock that do not require the Spring context are unaffected by this break and can remain as-is in the meantime.

✓ **Check:**
Spock specifications that use @SpringBootTest or @SpringRunner pass after migrating to JUnit 5 or waiting for Spock's Groovy 5 support

Spring Session Hazelcast Auto-Configuration Removed

Spring Boot 4.0 dropped built-in Hazelcast session auto-configuration. The app compiles but fails at startup with `NoSuchBeanDefinitionException`. Migrate to the Hazelcast-team-maintained Spring Session integration.

WHAT YOU'LL SEE

```
// Boot 4.0 startup failure:
org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type
  'org.springframework.session.SessionRepository' available

// Or context load failure mentioning missing HazelcastIndexedSessionRepository
```

WHAT CHANGED

Spring Boot's auto-configuration for Hazelcast-backed Spring Session was removed from `spring-boot-autoconfigure`. The classes `HazelcastSessionConfiguration` and `HazelcastSessionProperties` no longer exist in the Spring Boot artifact. The Hazelcast team took ownership of the integration.

```
<!-- Boot 3.5: built-in, no extra dependency needed beyond spring-session-hazelcast -->
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-hazelcast</artifactId>
</dependency>
<!-- Boot 4.0: use Hazelcast team's own integration artifact -->
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast-spring-session</artifactId>
    <version><!-- check Hazelcast release --></version>
</dependency>
```

HOW TO FIX

1. **Migrate to the Hazelcast-maintained Spring Session integration.** Replace `spring-session-hazelcast` with the artifact published by the Hazelcast team. Refer to Hazelcast's own documentation for configuration — the property keys and bean names may differ from the Boot-managed version.

✓ Check:
Hazelcast-backed session store initialises correctly after migrating to the Hazelcast-team-maintained integration

Spring Session MongoDB Auto-Configuration Removed

Spring Boot 4.0 dropped built-in MongoDB session auto-configuration. The app compiles but fails at startup with `NoSuchBeanDefinitionException`. Migrate to the MongoDB-team-maintained Spring Session integration in Spring Data MongoDB.

WHAT YOU'LL SEE

```
// Boot 4.0 startup failure:
org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type
  'org.springframework.session.SessionRepository' available

// Or context load failure mentioning missing MongoIndexedSessionRepository
```

WHAT CHANGED

Spring Boot's auto-configuration for MongoDB-backed Spring Session was removed from `spring-boot-autoconfigure`. The `MongoSessionConfiguration` and related auto-configuration classes no longer exist in the Spring Boot artifact. Spring Data MongoDB took ownership of the integration.

```
<!-- Boot 3.5: built-in auto-config via spring-session-data-mongodb -->
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-mongodb</artifactId>
</dependency>
<!-- Boot 4.0: session support is part of spring-boot-starter-data-mongodb -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<!-- Refer to Spring Data MongoDB docs for explicit session config -->
```

HOW TO FIX

1. **Migrate to the Spring Data MongoDB session integration.** The MongoDB team's Spring Session support is bundled with Spring Data MongoDB. Add `spring-boot-starter-data-mongodb` and configure the session repository according to Spring Data MongoDB's documentation. The `spring.session.store-type=mongodb` property alone is no longer sufficient.

> ✓ **Check:**
> MongoDB-backed session store initialises correctly after migrating to the MongoDB-team-maintained integration

@SpringBootTest No Longer Auto-Configures MockMvc

`@SpringBootTest` no longer auto-configures `MockMvc` in Boot 4.0. Add `@AutoConfigureMockMvc` to your test class or the `MockMvc` bean won't be available.

WHAT YOU'LL SEE

```
@SpringBootTest
class MyControllerTest {
    @Autowired
    MockMvc mockMvc;  // null in Boot 4.0

    @Test
    void test() throws Exception {
        mockMvc.perform(get("/api/hello"))  // NullPointerException
               .andExpect(status().isOk());
    }
}

// Boot 4.0 failure:
org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type
  'org.springframework.test.web.servlet.MockMvc' available
```

WHAT CHANGED

MockMvc auto-configuration was decoupled from `@SpringBootTest`. The MOCK webEnvironment still creates a mock servlet context, but the `MockMvc` bean is no longer registered in that context automatically. It must be opted into explicitly with `@AutoConfigureMockMvc`.

```
~~@SpringBootTest~~
~~class MyControllerTest {~~
~~    @Autowired~~
~~    MockMvc mockMvc;~~
~~}~~
@SpringBootTest
@AutoConfigureMockMvc
class MyControllerTest {
    @Autowired
    MockMvc mockMvc;
}
```

```
// @WebMvcTest already includes MockMvc auto-configuration.
// Only @SpringBootTest is affected.
```

HOW TO FIX

1. **Add @AutoConfigureMockMvc to the test class.** This is a one-annotation fix. Add `@AutoConfigureMockMvc` alongside `@SpringBootTest` on any test class that autowires `MockMvc`.

✓ Check:
MockMvc is available in @SpringBootTest tests after adding @AutoConfigureMockMvc

TIER 2 · WON'T RUN IMPACT: S spring-boot 3.5 → 4.0

Test-Slice Annotation Starters Relocated

Spring Boot 4.0 relocated test-slice annotations and their auto-configuration metadata. Tests using custom test slices or referencing old packages fail with ClassNotFoundException or silently load the wrong context.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
java.lang.IllegalStateException: Failed to load ApplicationContext for
  [MergedContextConfiguration@6f2b5e...
  testClass = com.example.web.ProductControllerTest]
Caused by: java.lang.ClassNotFoundException:
  org.springframework.boot.test.autoconfigure.web.servlet
  .WebMvcTypeExcludeFilter
 at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(...)
---
Or silently:
---
com.example.web.ProductControllerTest > shouldReturnProduct() FAILED
  org.springframework.beans.factory.NoSuchBeanDefinitionException:
  No qualifying bean of type 'com.example.web.ProductController'
  (expected in thin web-slice context but full context was loaded instead)
```

WHAT CHANGED

Spring Boot 4.0 reorganised the test auto-configuration packages. Several `TypeExcludeFilter` classes and test-slice support classes moved or were renamed. Custom test-slice annotations that referenced the old filter classes by fully qualified name break at runtime. Standard annotations like `@WebMvcTest`, `@DataJpaTest`, and `@JsonTest` still work, but custom slices that mirrored their structure need updating.

```
~~@Target(ElementType.TYPE)~~
~~@Retention(RetentionPolicy.RUNTIME)~~
~~@BootstrapWith(SpringBootTestContextBootstrapper.class)~~
~~@OverrideAutoConfiguration(enabled = false)~~
~~@TypeExcludeFilters(CustomSliceTypeExcludeFilter.class)~~
~~@AutoConfigureCache~~
~~@AutoConfigureCustomService~~
~~@ImportAutoConfiguration~~
~~public @interface CustomServiceTest {~~
~~}~~
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@BootstrapWith(SpringBootTestContextBootstrapper.class)
@OverrideAutoConfiguration(enabled = false)
@TypeExcludeFilters(CustomSliceTypeExcludeFilter.class)
@AutoConfigureCache
@AutoConfigureCustomService
@ImportAutoConfiguration
public @interface CustomServiceTest {
    // Annotation structure unchanged, but verify your
    // TypeExcludeFilter extends the correct base class
    // and spring.factories / AutoConfiguration.imports are updated
}
```

```
~~org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureWebMvc=\~~
~~  com.example.test.CustomWebMvcAutoConfiguration~~
# Moved to META-INF/spring/AutoConfiguration.imports
# or updated key in spring.factories
```

```
org.springframework.boot.autoconfigure.AutoConfiguration.imports=\
  com.example.test.CustomWebMvcAutoConfiguration
```

HOW TO FIX

1. **Update custom test-slice annotations.** If you wrote custom test-slice annotations (like `@CustomServiceTest`), verify that the `TypeExcludeFilter`, bootstrapper, and auto-configuration imports still resolve. Update fully qualified class references to match the new package locations.
2. **Migrate spring.factories to AutoConfiguration.imports.** Move test auto-configuration entries from `META-INF/spring.factories` to `META-INF/spring/AutoConfiguration.imports`. This was deprecated in Boot 3.x and is now enforced.
3. **Re-test standard slices.** Even if you only use standard annotations like `@WebMvcTest`, run your full test suite to verify. The context filtering may behave differently and load or exclude different beans.

> **✓ Check:**
> mvn test — all @DataJpaTest/@WebMvcTest slices load correctly

TIER 3

Wrong Results

Subtle Behavioural Changes

Your code runs but produces wrong output. Subtle behavioural changes in serialisation, queries, and observability.

CARD INDEX · 17 CARDS

DevTools Live Reload Disabled by Default

Boot 4.0 disables DevTools Live Reload by default. The browser no longer refreshes automatically after code changes. Set `spring.devtools.livereload.enabled=true` in your dev profile to restore it.

WHAT YOU'LL SEE

```
# Boot 3.5: live reload works automatically with devtools on classpath.
# Boot 4.0: live reload silently disabled. Browser does not refresh.
# No error — the server restarts but the browser stays on the old page.
```

WHAT CHANGED

The `spring.devtools.livereload.enabled` property now defaults to `false`. The Live Reload server is not started unless this property is explicitly set to `true`. The DevTools restart trigger (watching for classpath changes) is unaffected — only the browser push notification is disabled.

```
# Boot 3.5: live reload on by default, no property needed
# Boot 4.0: must opt in
spring.devtools.livereload.enabled=true
```

HOW TO FIX

1. **Add spring.devtools.livereload.enabled=true to your dev profile.** Add the property to `application-dev.properties` or your development-time configuration. Do not add it to production profiles — DevTools should not be on the classpath in production anyway, but explicit opt-in makes the intent clear.

> ✓ **Check:**
> Browser refreshes automatically after code changes when spring.devtools.livereload.enabled=true is set in development profile

TIER 3 · WRONG RESULTS IMPACT: M spring-boot 3.5 → 4.0

Hibernate Native Query Date Return Types Changed

Native SQL queries now return `java.time.LocalDate` instead of `java.sql.Date`. Code casting to the old types throws ClassCastException or silently loses time components.

WHAT YOU'LL SEE

```
// Native query — same code, different result type
Object result = em.createNativeQuery("SELECT created_date FROM orders WHERE id = 1")
                  .getSingleResult();

// Before (Spring Boot 3.5)
result.getClass() → java.sql.Date

// After (Spring Boot 4.0)
result.getClass() → java.time.LocalDate

// ClassCastException at runtime
java.lang.ClassCastException:
  java.time.LocalDate cannot be cast to java.sql.Date

// Or subtler: time component silently lost
java.sql.Timestamp ts = (java.sql.Timestamp) result;  // was Timestamp
// Now: java.time.LocalDateTime — no getTime() method
```

WHAT CHANGED

Hibernate 7 (used by Spring Boot 4.0) changed the default Java types returned for SQL `DATE`, `TIME`, and `TIMESTAMP` columns in native queries. `java.sql.Date` became `java.time.LocalDate`, `java.sql.Time` became `java.time.LocalTime`, and `java.sql.Timestamp` became `java.time.LocalDateTime`.

```
~~java.sql.Date date = (java.sql.Date) row[2];~~
~~long millis = date.getTime();~~
LocalDate date = (LocalDate) row[2];
long millis = date.atStartOfDay(ZoneOffset.UTC)
                  .toInstant().toEpochMilli();
```

```
~~@Query(value = "SELECT created_date FROM orders", nativeQuery = true)~~
~~List<java.sql.Date> findAllDates();~~
@Query(value = "SELECT created_date FROM orders", nativeQuery = true)
List<java.time.LocalDate> findAllDates();
```

```
~~java.sql.Timestamp ts = (java.sql.Timestamp) row[3];~~
LocalDateTime ts = (LocalDateTime) row[3];
```

HOW TO FIX

1. **Update result type casts to java.time (recommended).** Replace all `java.sql.Date` casts with `java.time.LocalDate`, `java.sql.Time` with `java.time.LocalTime`, and `java.sql.Timestamp` with `java.time.LocalDateTime`. This is the correct modern approach.
2. **Force old types with explicit addScalar().** For legacy code that can't be updated, use Hibernate's `addScalar("col", StandardBasicTypes.DATE)` to force the old `java.sql` return types on specific queries.

> ✓ **Check:**
> SELECT queries return correct date/time types (check ResultSet)

Jackson Date Serialisation Flip

Jackson 3.0 defaults to ISO-8601 date strings instead of numeric timestamps. Every API response containing a date silently changes shape.

WHAT YOU'LL SEE

```
// API response — before (Spring Boot 3.5)
{"created": 1714089600000}

// API response — after (Spring Boot 4.0)
{"created": "2025-04-26T00:00:00Z"}

// Front-end JavaScript blows up
TypeError: date.getTime is not a function
// or: contract test fails
Expected: <1714089600000> (Long)
  Actual: <"2025-04-26T00:00:00Z"> (String)
```

WHAT CHANGED

Jackson 3.0 changed the default value of `SerializationFeature.WRITE_DATES_AS_TIMESTAMPS` from `true` to `false`. Dates, times, and durations now serialise as ISO-8601 strings by default instead of numeric epoch milliseconds.

```
# (no explicit setting needed in Boot 3.5)
spring.jackson.serialization.write-dates-as-timestamps=true
```

```
const ts = new Date(response.created);  // was a number
const ts = new Date(response.created);  // now parses ISO string
```

```
// Jackson 2 default: timestamps enabled
ObjectMapper mapper = new ObjectMapper();
mapper.registerModule(new JavaTimeModule());
// Jackson 3 default: ISO strings
ObjectMapper mapper = new ObjectMapper();
mapper.registerModule(new JavaTimeModule());
// only if you need the old numeric format:
mapper.enable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
```

HOW TO FIX

1. **Opt in to the new ISO-8601 default (recommended).** Update your API clients and contract tests to expect ISO-8601 strings. This is the better long-term format. Coordinate the change with front-end teams so they parse the string, not a number.
2. **Restore numeric timestamps.** Add `spring.jackson.serialization.write-dates-as-timestamps=true` to `application.properties`. This gives you the old behaviour while you plan the migration.

> ✓ **Check:**
> JSON responses show dates in expected format (check API output)

write-dates-as-timestamps Config Silently Not Applied

The `spring.jackson.serialization.write-dates-as-timestamps` property is silently ignored in Boot 4.0. Your explicit timestamp configuration stops working.

WHAT YOU'LL SEE

```
# application.properties (still present, no warning)
spring.jackson.serialization.write-dates-as-timestamps=true

// API response — before (Spring Boot 3.5, property honoured)
{"timestamp": 1714089600000, "expiry": 1714176000000}

// API response — after (Spring Boot 4.0, property ignored)
{"timestamp": "2025-04-26T00:00:00Z", "expiry": "2025-04-27T00:00:00Z"}

// Mobile app crash log
java.lang.NumberFormatException: For input string: "2025-04-26T00:00:00Z"
```

WHAT CHANGED

Spring Boot 4.0's Jackson auto-configuration no longer applies the `spring.jackson.serialization.*` feature toggles to the Jackson 3 ObjectMapper. The property remains in your config with no startup warning, but the serialisation feature is never enabled. Jackson 3's own default (ISO-8601 strings) takes effect instead.

```
~~spring.jackson.serialization.write-dates-as-timestamps=true~~
# Removed — configure via ObjectMapper customiser instead
```

```
~~# (relied on spring.jackson.serialization.write-dates-as-timestamps)~~
@Bean
public Jackson2ObjectMapperBuilderCustomizer timestampDates() {
    return builder -> builder.featuresToEnable(
        SerializationFeature.WRITE_DATES_AS_TIMESTAMPS
    );
}
```

```
~~private Instant timestamp;~~
@JsonFormat(shape = JsonFormat.Shape.NUMBER)
private Instant timestamp;
```

HOW TO FIX

1. **Register an ObjectMapper customiser bean.** Create a `Jackson2ObjectMapperBuilderCustomizer` that explicitly enables `WRITE_DATES_AS_TIMESTAMPS`. This replaces the dead properties-based config.
2. **Migrate clients to ISO-8601 (recommended).** If you can coordinate with API consumers, drop the timestamp format entirely and adopt ISO-8601 strings. Remove the old property and let Jackson 3's new default take effect.

> **✓ Check:**
> Date fields serialize as ISO-8601 strings, not numeric timestamps

Jackson Locale Format Change (BCP 47)

Jackson 3.0 serialises `Locale` using IETF BCP 47 format (zh-CN) instead of Java's underscore format (zh_CN). Locale-matching logic silently breaks.

WHAT YOU'LL SEE

```
// API response — before (Spring Boot 3.5)
{"userLocale": "zh_CN"}

// API response — after (Spring Boot 4.0)
{"userLocale": "zh-CN"}

// Downstream lookup fails silently
Map<String, String> labels = i18nMap.get(user.getLocale());
// returns null — map keys use "zh_CN", response now sends "zh-CN"

// Test failure
org.opentest4j.AssertionFailedError:
Expected: "zh_CN"
  Actual: "zh-CN"
```

WHAT CHANGED

Jackson 3.0 switched `Locale` serialisation from Java's `toString()` format (`zh_CN`) to the IETF BCP 47 language tag format (`zh-CN`). Underscores become hyphens and the structure follows RFC 5646.

```
~~// No custom serialiser needed in Boot 3.5~~
@JsonSerialize(using = JavaLocaleSerializer.class)
private Locale userLocale;
```

```
~~String localeKey = user.getLocale(); // "zh_CN"~~
String localeKey = Locale.forLanguageTag(user.getLocale()).toString(); // normalise
```

```
~~i18nMap.put("zh_CN", chineseLabels);~~
i18nMap.put("zh-CN", chineseLabels);
```

HOW TO FIX

1. **Adopt BCP 47 throughout (recommended).** Update your i18n maps, database locale columns, and downstream consumers to use hyphenated BCP 47 tags. This aligns with HTTP headers and browser APIs. Run a data migration for persisted locale strings.
2. **Write a custom Locale serialiser.** Register a custom Jackson serialiser that calls `locale.toString()` instead of `locale.toLanguageTag()` . This preserves the old underscore format while you plan the broader migration.

> ✓ **Check:**
> Locale-sensitive fields render correctly in all target locales

Jackson 3 Auto-Discovers All Modules on Classpath

Jackson 3 registers every module it finds on the classpath. Modules you added as transitive dependencies may now change your JSON output silently. Disable auto-discovery with `spring.jackson.find-and-add-modules=false`.

WHAT YOU'LL SEE

```
// Boot 3.5: only well-known modules registered. Output is predictable.
{"amount": 150.00, "currency": "EUR"}

// Boot 4.0: a transitive Jackson module on the classpath changes
// BigDecimal serialization:
{"amount": "150.00", "currency": "EUR"}

// No error, no warning. The JSON schema changed silently.
```

WHAT CHANGED

Jackson 3 uses the Java ServiceLoader mechanism to find and register all `com.fasterxml.jackson.databind.Module` implementations on the classpath. Spring Boot no longer controls which modules are registered — every module present is activated. In Boot 3.5, Spring Boot explicitly registered only well-known modules.

```
# Boot 3.5: no property needed, only well-known modules registered
# Boot 4.0: disable auto-discovery to prevent unexpected module activation
spring.jackson.find-and-add-modules=false
```

HOW TO FIX

1. **Audit classpath modules and disable auto-discovery if needed.** Run `mvn dependency:tree` and look for any artifacts ending in `-module` or containing `jackson`. For each one, verify its effect on your serialization output. If unexpected modules are activating, set `spring.jackson.find-and-add-modules=false` and register only the modules you need explicitly.
2. **Explicitly register required modules.** After disabling auto-discovery, register required modules via a `Jackson2ObjectMapperBuilderCustomizer` (now renamed `JsonMapperBuilderCustomizer`) bean that calls `modules(new JavaTimeModule(), ...)`.

> ✓ **Check:**
> JSON serialization output is identical before and after the upgrade, or unexpected module behaviour is controlled with spring.jackson.find-and-add-modules

spring.jackson.default-property-inclusion Silently Ignored

The `spring.jackson.default-property-inclusion` property no longer configures Jackson 3's ObjectMapper. Null fields reappear in your API responses without warning.

WHAT YOU'LL SEE

```
# application.properties (still present, no startup warning)
spring.jackson.default-property-inclusion=non_null

// API response — before (Spring Boot 3.5)
{"name": "Alice", "age": 30}

// API response — after (Spring Boot 4.0)
{"name": "Alice", "age": 30, "middleName": null, "nickname": null}

// Contract test failure
org.opentest4j.AssertionFailedError:
Expected JSON keys: [name, age]
  Actual JSON keys: [name, age, middleName, nickname]
```

WHAT CHANGED

Spring Boot 4.0's auto-configuration for Jackson 3 no longer reads the `spring.jackson.default-property-inclusion` property. The property sits in your config file doing nothing — no deprecation warning, no startup error. The ObjectMapper reverts to Jackson's default inclusion policy: `ALWAYS` (include everything, including nulls).

```
~~spring.jackson.default-property-inclusion=non_null~~
# Removed — configure via ObjectMapper bean instead
```

```
~~# (relied on spring.jackson.default-property-inclusion)~~
@Bean
public Jackson2ObjectMapperBuilderCustomizer nonNullInclusion() {
    return builder → builder
        .serializationInclusion(JsonInclude.Include.NON_NULL);
}
```

```
~~public class UserDto {~~
@JsonInclude(JsonInclude.Include.NON_NULL)
public class UserDto {
```

HOW TO FIX

1. **Register an ObjectMapper customiser bean (recommended).** Create a `Jackson2ObjectMapperBuilderCustomizer` bean that calls `serializationInclusion(NON_NULL)`. This replaces the property-based configuration and works with Jackson 3.
2. **Annotate DTOs individually.** Add `@JsonInclude(JsonInclude.Include.NON_NULL)` to each DTO class. More verbose, but gives you per-class control and makes the contract explicit in the code.

> **✓ Check:**
> JSON output excludes null fields where expected (compare before/after)

Liveness and Readiness Probes Enabled by Default

Boot 4.0 enables liveness and readiness probes for all apps, not just those in Kubernetes. The `/actuator/health` response changes shape. Downstream parsers and security rules may break.

WHAT YOU'LL SEE

```
// Boot 3.5 (outside Kubernetes): /actuator/health response
{"status":"UP"}

// Boot 4.0: /actuator/health response includes probe groups
{
  "status": "UP",
  "groups": ["liveness", "readiness"]
}

// Consumer code that checked for exact JSON equality or did not
// expect the "groups" key now fails.
```

WHAT CHANGED

The `management.endpoint.health.probes.enabled` property now defaults to `true` instead of being driven by Kubernetes detection. Both `/actuator/health/liveness` and `/actuator/health/readiness` are always exposed, and the main health response always lists them in the `groups` field.

```
# Boot 3.5: probes off by default outside Kubernetes
# management.endpoint.health.probes.enabled=true  # required to enable
# Boot 4.0: probes on by default; disable explicitly if not needed
management.endpoint.health.probes.enabled=false
```

HOW TO FIX

1. **Disable probes if your deployment does not use them.** Set `management.endpoint.health.probes.enabled=false` in `application.properties` to restore Boot 3.5 behaviour for non-Kubernetes deployments. Alternatively, update downstream health consumers to expect the `groups` field.
2. **Update security rules if probe endpoints were unintentionally exposed.** Review your actuator security configuration. If `/actuator/health/liveness` and `/actuator/health/readiness` were not expected to be publicly accessible, add them to your security exclusions or require authentication.

> ✓ **Check:**
> Health endpoint responds correctly and liveness/readiness groups are present or absent as expected after reviewing probe configuration

Logback Default Charset Changed to UTF-8

Boot 4.0 forces Logback file appenders to UTF-8. Logs containing non-ASCII characters may appear garbled to tools or systems that expected the platform's default encoding. No error — wrong characters.

WHAT YOU'LL SEE

```
# Boot 3.5 (platform encoding = Windows-1252):
2025-01-01 INFO  Order confirmed: €150.00 for Müller

# Boot 4.0 (UTF-8 forced, consumed by tool expecting Windows-1252):
2025-01-01 INFO  Order confirmed: â,¬150.00 for MÃ¼ller
# Or if the terminal/tool can't decode UTF-8 sequences:
2025-01-01 INFO  Order confirmed: ?150.00 for M?ller
```

WHAT CHANGED

Spring Boot's default Logback configuration changed the charset for file appenders to UTF-8. Console appenders now use `Console#charset()` (the JVM's console charset, Java 17+) rather than the platform default. The change affects the default Logback XML provided by Spring Boot; custom Logback configurations that already specify an explicit charset are unaffected.

```
<!-- Boot 3.5: no charset specified, platform default used for file appender -->
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>app.log</file>
    <encoder>
        <pattern>%d{yyyy-MM-dd} %-5level %msg%n</pattern>
    </encoder>
</appender>
<!-- Boot 4.0: UTF-8 is default; specify charset explicitly if you need platform
encoding -->
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>app.log</file>
    <encoder>
        <charset>UTF-8</charset>
        <pattern>%d{yyyy-MM-dd} %-5level %msg%n</pattern>
    </encoder>
</appender>
```

HOW TO FIX

1. **Update log consumers to expect UTF-8.** If log files are consumed by external tools, log aggregators, or scripts, ensure those consumers are configured to read UTF-8. This is the correct long-term fix.
2. **Restore platform encoding if needed.** If UTF-8 is not acceptable for your environment, add an explicit `<charset>` to your file appenders in `logback-spring.xml`. This is a short-term workaround; prefer updating consumers to UTF-8.

✓ Check:
Log output contains correctly encoded characters (especially non-ASCII) in both console and file appenders after the charset change

MongoDB Configuration Properties Renamed

`spring.data.mongodb.*` properties are silently ignored in Boot 4.0. The new prefix is `spring.mongodb.*`. The app starts but connects with defaults — wrong host, no auth, wrong database.

⚙ OpenRewrite

WHAT YOU'LL SEE

```
# application.properties (Boot 3.5 style — silently ignored on 4.0):
spring.data.mongodb.uri=mongodb://user:pass@prod-host:27017/mydb
spring.data.mongodb.database=mydb

# Boot 4.0: app starts, connects to localhost:27017 with no auth.
# No error — wrong database, wrong host. Data operations succeed
# against the wrong instance.
```

WHAT CHANGED

All MongoDB connection and configuration properties moved from the `spring.data.mongodb` namespace to `spring.mongodb`. Examples: `spring.data.mongodb.uri` → `spring.mongodb.uri`, `spring.data.mongodb.database` → `spring.mongodb.database`. Actuator management properties changed from `management.health.mongo.*` to `management.health.mongodb.*`.

```
~~spring.data.mongodb.uri=mongodb://user:pass@host:27017/mydb~~
~~spring.data.mongodb.database=mydb~~
~~spring.data.mongodb.auto-index-creation=true~~
spring.mongodb.uri=mongodb://user:pass@host:27017/mydb
spring.mongodb.database=mydb
spring.mongodb.auto-index-creation=true
```

```
~~management.health.mongo.enabled=false~~
management.health.mongodb.enabled=false
```

HOW TO FIX

1. **Rename all spring.data.mongodb. properties to spring.mongodb..** Do a project-wide search-and-replace: `spring.data.mongodb.` → `spring.mongodb.`. Check all `application.properties`, `application.yml`, and any profile-specific variants. OpenRewrite has a migration recipe that handles this automatically.

✓ Check:
Application connects to MongoDB and all configuration (URI, auth, pool size, etc.) applies correctly after renaming properties

MongoDB UUID and BigDecimal Representations No Longer Defaulted

Boot 4.0 removes the default UUID and BigDecimal BSON representations. Existing documents written with Boot 3.5 defaults may deserialize incorrectly or fail. Explicit configuration is now required.

WHAT YOU'LL SEE

```
// Boot 3.5: UUID stored as STANDARD binary subtype 4
// Boot 4.0 default: UUID stored as JAVA_LEGACY binary subtype 3

// Reading a Boot 3.5 document in a Boot 4.0 app:
org.bson.codecs.configuration.CodecConfigurationException:
  No codec found for UUID with representation JAVA_LEGACY

// Or: UUID field deserializes to null or wrong value silently
```

WHAT CHANGED

The `spring.mongodb.representation.uuid` and `spring.data.mongodb.representation.big-decimal` properties must now be configured explicitly. Spring Boot 4.0 no longer injects defaults for these representations. Applications must declare the representations they use — the same values Boot 3.5 set automatically, if compatibility with existing data is required.

```
# Boot 3.5: these were set automatically (no configuration needed)
# Boot 4.0: must be explicit to match what Boot 3.5 used
spring.mongodb.representation.uuid=standard
# For BigDecimal — check what representation your Boot 3.5 data was written with:
# spring.data.mongodb.representation.big-decimal=decimal128
```

HOW TO FIX

1. **Declare the representations that match your existing data.** Set `spring.mongodb.representation.uuid` to `standard` if your documents were written with Boot 3.5 defaults. For BigDecimal, check which BSON type your existing documents use (inspect with mongosh) and configure `spring.data.mongodb.representation.big-decimal` accordingly.

2. **Migrate existing data if changing representations.** If you want to adopt different representations going forward, you must write a migration script to convert existing documents. Do not change the representation without migrating data or you will lose the ability to read old documents.

> **✓ Check:**
> UUID and BigDecimal fields round-trip correctly and existing documents with the old representation are still readable after explicit configuration

TIER 3 · WRONG RESULTS IMPACT: M spring-boot 3.5 → 4.0

@NullMarked Default — IDE/Compiler Null Safety

Spring Framework 7 applies `@NullMarked` to all packages. IDEs and null-checking tools now flag your existing null-passing code as warnings or errors — even though the runtime behaviour is unchanged.

WHAT YOU'LL SEE

```
// Your existing code — unchanged, worked fine in Boot 3.5
ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
String name = ctx.getEnvironment().getProperty("app.name");
int length = name.length();  // ← IDE warning: name may be null

// IntelliJ inspection results after upgrade
WARNING: Method invocation 'length' may produce NullPointerException
  at MyService.java:42
WARNING: Passing 'null' argument to parameter annotated as @NonNull
  at MyConfig.java:18

// If using -Werror or NullAway/ErrorProne
$ mvn compile
[ERROR] MyService.java:[42,24] error: [NullAway] dereferencing expression
  name, which is @Nullable
[ERROR] MyConfig.java:[18,35] error: [NullAway] passing @Nullable parameter
  where @NonNull is required
```

WHAT CHANGED

Spring Framework 7 added `@NullMarked` (from JSpecify) at the package level across the entire framework. This means all method parameters and return types are assumed `@NonNull` by default unless explicitly annotated `@Nullable`. IDEs, static analysis tools, and compile-time null checkers now see Spring's null contracts and flag violations.

```
~~String name = env.getProperty("app.name");~~
~~return name.toUpperCase();~~
String name = env.getProperty("app.name");
if (name == null) {
    throw new IllegalStateException("app.name not configured");
}
return name.toUpperCase();
```

```
~~String name = env.getProperty("app.name");~~
String name = env.getRequiredProperty("app.name");
```

```
~~// (no null warnings in Boot 3.5)~~
// Temporarily suppress in build.gradle
tasks.withType(JavaCompile) {
    options.compilerArgs += ['-Xlint:-nullness']
}
```

HOW TO FIX

1. **Fix null handling in your code (recommended).** Follow the IDE warnings and add null checks, use `Optional`, or switch to non-null API variants like `getRequiredProperty()`. These are real potential NPE bugs that were always there — the framework is now telling you about them.
2. **Suppress during migration.** If the volume of warnings is too high to fix at once, suppress null-checking warnings at the compiler level and create a backlog to address them incrementally. Don't leave the suppression in place permanently.

✓ Check:

No new NullPointerException at boundaries where nulls were OK before

TIER 3 · WRONG RESULTS IMPACT: M spring-boot 3.5 → 4.0

Controller/View Spans Silently Disabled

Spring Boot 4.0 disables automatic controller and view rendering spans by default. Observability dashboards show gaps where traces used to appear — but the app itself works fine.

WHAT YOU'LL SEE

```
// Grafana / Jaeger trace — before (Spring Boot 3.5)
── HTTP GET /api/orders ──────────────────────
  └── controller: OrderController.getOrders  [12ms]
      └── view: orders/list                  [3ms]
          └── jdbc: SELECT * FROM orders     [8ms]

// Grafana / Jaeger trace — after (Spring Boot 4.0)
── HTTP GET /api/orders ──────────────────────
  └── jdbc: SELECT * FROM orders             [8ms]
// controller and view spans MISSING

// Dashboard alert
"No data" for panel "Controller Response Time p99"
// SLO breach: "99th percentile latency unknown"
```

WHAT CHANGED

Spring Boot 4.0 disabled the automatic instrumentation for Spring MVC controller method spans and view rendering spans by default. The properties `management.observations.http.server.requests.enabled` and related `spring.mvc.observation` settings changed their defaults. The traces still record HTTP and JDBC spans, but the controller-level granularity is gone.

```
# (default in Boot 3.5: controller spans enabled)
management.tracing.enabled=true
spring.mvc.observation.auto-configuration.enabled=true
```

```
# (auto-instrumentation was on by default)
@Bean
public ObservationRegistryCustomizer<ObservationRegistry> serverSpans() {
    return registry -> registry.observationConfig()
        .observationHandler(new DefaultMeterObservationHandler(meterRegistry));
}
```

```
# (no explicit config needed in Boot 3.5)
management.observations.http.server.requests.enabled=true
management.observations.http.client.requests.enabled=true
```

HOW TO FIX

1. **Re-enable controller observation spans.** Add `spring.mvc.observation.auto-configuration.enabled=true` and `management.observations.http.server.requests.enabled=true` to your `application.properties`. This restores the controller-level spans in your traces.
2. **Update dashboards to use HTTP spans.** If the overhead matters, update your Grafana/Datadog dashboards to use the HTTP server span (`http.server.requests`) instead of the controller span. This span is still present and includes the handler information as attributes.

> ✓ **Check:**
> All Grafana/Jaeger panels show data — no ‘No data’ gaps

TIER 3 · WRONG RESULTS IMPACT: M spring-boot 3.5 → 4.0

Path Matching Engine: AntPathMatcher → PathPattern

Spring Security now uses `PathPattern` instead of `AntPathMatcher`. Some wildcard patterns match differently, causing silent authorisation gaps or false denials.

WHAT YOU'LL SEE

```
// Security configuration — unchanged
http.authorizeHttpRequests(auth → auth
    .requestMatchers("/api/**/admin").hasRole("ADMIN")
);

// Before (AntPathMatcher — Spring Boot 3.5)
GET /api/v1/admin         → 403 (matched, requires ADMIN)
GET /api/v1/users/admin   → 403 (matched, ** crosses segments)

// After (PathPattern — Spring Boot 4.0)
GET /api/v1/admin         → 403 (still matched)
GET /api/v1/users/admin   → 200 (NOT matched — pattern behaves differently)

// Security test failure
Expected: request denied (403)
  Actual: request allowed (200)
// Potential security hole: endpoint exposed without authorisation
```

WHAT CHANGED

Spring Boot 4.0 switched the default URL path matching engine from `AntPathMatcher` to `PathPatternParser` for both MVC request mapping and Security request matchers. While most patterns work identically, edge cases around `**` in the middle of patterns, trailing slashes, and encoded path segments differ.

```
~~.requestMatchers("/api/**/admin").hasRole("ADMIN")~~
.requestMatchers("/api/{*path}/admin").hasRole("ADMIN")
```

```
~~.requestMatchers("/api/users").authenticated()~~
.requestMatchers("/api/users", "/api/users/").authenticated()
```

```
~~// using default PathPatternParser~~
@Bean
public WebSecurityCustomizer legacyMatching() {
    return web → web.httpFirewall(new StrictHttpFirewall());
}
// In security config:
.requestMatchers(new AntPathRequestMatcher("/api/**/admin"))
    .hasRole("ADMIN")
```

HOW TO FIX

1. **Audit all path patterns in security config.** Review every `requestMatchers()` pattern for uses of `**` in the middle (not at the end), trailing slashes, and URL-encoded segments. Test each pattern with PathPattern semantics. Use `{*varName}` for capturing multi-segment path variables.
2. **Use AntPathRequestMatcher for legacy patterns.** For patterns that can't be rewritten, wrap them in `new AntPathRequestMatcher(pattern)` to use the old matching engine on a per-rule basis.

✓ Check:

All API endpoints respond (especially wildcard/regex patterns)

TIER 3 · WRONG RESULTS IMPACT: M spring-boot 3.5 → 4.0

maxAttempts Now Means Retries, Not Total Attempts

`@Retryable(maxAttempts = 3)` now performs 3 retries (4 total calls) instead of 3 total attempts. Your retry budget silently doubles.

WHAT YOU'LL SEE

```
// Before (Spring Boot 3.5): 3 total attempts
INFO  Calling payment service (attempt 1/3)
WARN  Payment failed, retrying...
INFO  Calling payment service (attempt 2/3)
WARN  Payment failed, retrying...
INFO  Calling payment service (attempt 3/3)
ERROR Payment failed after 3 attempts

// After (Spring Boot 4.0): 1 initial + 3 retries = 4 total
INFO  Calling payment service (attempt 1/4)
WARN  Payment failed, retrying...
INFO  Calling payment service (attempt 2/4)
WARN  Payment failed, retrying...
INFO  Calling payment service (attempt 3/4)
WARN  Payment failed, retrying...
INFO  Calling payment service (attempt 4/4)
ERROR Payment failed after 4 attempts

// Integration test failure
Expected invocation count: 3
  Actual invocation count: 4
```

WHAT CHANGED

The `maxAttempts` parameter in `@Retryable` changed semantics. In Spring Retry 1.x (Boot 3.5), it meant total attempts including the initial call. In the new version (Boot 4.0), it means the number of retries after the initial call. So `maxAttempts = 3` now results in 4 total calls.

```
~~@Retryable(maxAttempts = 3)~~
@Retryable(maxAttempts = 2)  // 1 initial + 2 retries = 3 total
```

```
~~@Retryable(maxAttempts = 3)~~
@Retryable(retries = 2)  // clearer: 2 retries after initial call
```

```
~~RetryTemplate template = RetryTemplate.builder()~~
~~    .maxAttempts(3)~~
~~    .build();~~
RetryTemplate template = RetryTemplate.builder()
    .maxAttempts(2)  // was 3 — now means retries, not total
    .build();
```

HOW TO FIX

1. **Subtract 1 from every maxAttempts value.** Search for all `@Retryable` annotations and `RetryTemplate` configurations. Reduce each `maxAttempts` value by 1 to preserve the original total attempt count.
2. **Audit retry budgets.** The extra attempt may be acceptable or even desirable. Review each retry site and decide whether the new count is correct. Document the intended behaviour either way.

✓ Check:

Retried operations use correct backoff timing and max attempts

TIER 3 · WRONG RESULTS IMPACT: M spring-boot 3.5 → 4.0

@Retryable + @Transactional AOP Ordering Change

The default AOP advice ordering flipped: retry now wraps outside the transaction. A retried method gets a new transaction each attempt instead of retrying inside the same one.

WHAT YOU'LL SEE

```
// Before (Spring Boot 3.5) — retry inside transaction
DEBUG Opening transaction
DEBUG  Attempt 1: inserting order
WARN   Optimistic lock failure, retrying...
DEBUG  Attempt 2: inserting order
DEBUG  Attempt 2: success
DEBUG Committing transaction (1 commit total)

// After (Spring Boot 4.0) — retry outside transaction
DEBUG Opening transaction #1
DEBUG  Attempt 1: inserting order
WARN   Optimistic lock failure
DEBUG Rolling back transaction #1
DEBUG Opening transaction #2
DEBUG  Attempt 2: inserting order
DEBUG  Attempt 2: success
DEBUG Committing transaction #2

// Symptom: partial writes visible between retries
// Symptom: retry sees stale data from previous transaction
```

WHAT CHANGED

The default ordering of Spring AOP advice changed so that `@Retryable` has a higher precedence (runs first) than `@Transactional`. This means the retry logic wraps the transaction, not the other way around. Each retry attempt runs in its own transaction.

```
@Retryable(maxAttempts = 3)
@Transactional
public void placeOrder(Order order) {
@Retryable(maxAttempts = 3)
@Transactional
@Order(Ordered.HIGHEST_PRECEDENCE)  // force tx to wrap retry
public void placeOrder(Order order) {
```

```
# (default ordering in Boot 3.5: tx outside, retry inside)
@EnableRetry(order = Ordered.LOWEST_PRECEDENCE)
// Ensures retry runs inside the transaction
```

```
@Retryable @Transactional
public void placeOrder(Order order) { ... }
// Outer bean: handles retry
@Retryable(maxAttempts = 3)
public void placeOrderWithRetry(Order order) {
    orderService.placeOrder(order);
}
// Inner bean: handles transaction
@Transactional
public void placeOrder(Order order) { ... }
```

HOW TO FIX

1. **Separate retry and transaction into different beans (recommended).** Move `@Retryable` to an outer service and `@Transactional` to an inner service. This makes the ordering explicit, avoids AOP precedence surprises, and works the same regardless of framework defaults.

2. **Set explicit advice ordering.** Use `@EnableRetry(order = ...)` and `@EnableTransactionManagement(order = ...)` to explicitly control which advice runs first. Document the intended order.

> ✓ **Check:**
> Retried transactional methods execute retries outside the transaction

Spring Session Property Prefixes Renamed

`spring.session.redis.*` and `spring.session.mongodb.*` are silently ignored in Boot 4.0. The new prefixes are `spring.session.data.redis.*` and `spring.session.data.mongodb.*`.

WHAT YOU'LL SEE

```
# application.properties (Boot 3.5 style — silently ignored on 4.0):
spring.session.redis.namespace=myapp
spring.session.redis.flush-mode=on-save
spring.session.mongodb.collection-name=sessions

# Boot 4.0: app starts, sessions stored under default namespace.
# No error. Session isolation between apps sharing the same Redis
# instance is broken.
```

WHAT CHANGED

Spring Session's Redis backend properties moved from `spring.session.redis.*` to `spring.session.data.redis.*`. MongoDB backend properties moved from `spring.session.mongodb.*` to `spring.session.data.mongodb.*`. This mirrors the MongoDB connection property rename pattern (`spring.data.mongodb` → `spring.mongodb`) and aligns with the session backends being tied to Spring Data modules.

```
~~spring.session.redis.namespace=myapp:session~~
~~spring.session.redis.flush-mode=on-save~~
~~spring.session.redis.save-mode=on-set-attribute~~
~~spring.session.redis.cleanup-cron=0 * * * * *~~
spring.session.data.redis.namespace=myapp:session
spring.session.data.redis.flush-mode=on-save
spring.session.data.redis.save-mode=on-set-attribute
spring.session.data.redis.cleanup-cron=0 * * * * *
```

```
~~spring.session.mongodb.collection-name=sessions~~
spring.session.data.mongodb.collection-name=sessions
```

HOW TO FIX

1. **Rename spring.session.redis. to spring.session.data.redis..** Do a project-wide search-and-replace: `spring.session.redis.` → `spring.session.data.redis.` and `spring.session.mongodb.` → `spring.session.data.mongodb.`. Check all property files, YAML files, and profile variants.

> ✓ **Check:**
> Session store connects correctly with the renamed property prefixes — confirm sessions persist to Redis or MongoDB with the new configuration

Migration Playbook

There is no clean order to do this work in. There's a small set of changes that gate everything (do them first, by codemod), a larger set of subsystem work that's parallel-safe (do whichever you actually use), and a set of behavioural changes that you can't sequence at all — they ship the moment you upgrade, so the work is to catch them, not to do them. Decide which path you're on before any of that.

0

Decide Whether to Upgrade Now

Not every team should migrate in 2026. Spring Boot 3.5 has commercial extended support available, and a deliberate plan beats a rushed migration. Pick your path before opening any code.

Stay on 3.5 with NES | Pilot one module first | Full upgrade now

A

Gates — Compile-Blockers (do first, by codemod)

Without these, nothing compiles. Run OpenRewrite on a clean branch, accept the diff as a single reviewable PR, then get a green build before anyone reviews anything else. This is the only part of the migration that's genuinely sequential.

BUILD & RUNTIME	Java 17 Minimum · Gradle 8.14+ / Gradle 9 · Maven AOT Plugin · GraalVM 25 (native only) · Classic Uber-Jar Loader Removed · Tomcat WAR Runtime Starter
CODEMOD-HANDLED	⚙ Jackson 3.0 Group ID · ⚙ Jackson Class Renames · ⚙ javax.annotation Removed · ⚙ javax.inject Removed · ⚙ Security DSL Rewrite · Hibernate Processor Rename
COMPILE-BREAKERS	spring-jcl Removed · OkHttp3 Support Removed · ListenableFuture Removed · HttpHeaders / MultiValueMap · spring-retry BOM removal · webjars-locator-core BOM removal

B

Subsystem Work — Parallel-Safe (do whichever you use)

Each row is a separate PR and only matters if your codebase uses it. Order doesn't matter globally; merge in whatever sequence keeps trunk green. Test infrastructure travels with the production code that depends on it — don't defer it to a "testing phase," your build won't go green if you do.

HIBERNATE / JPA	EmptyInterceptor Removed · Session.delete() Removed · SelectionQuery.setOrder() Removed · @Where / @OrderBy Removed · Version Dialects Removed · CascadeType.SAVE_UPDATE Removed · Untyped Join Rejected · Native Query Date Types
SPRING SECURITY	OAuth Password Grant Removed · OpenSAML 4 → 5 · PKCE Mandatory · Access API Relocated · ApacheDS LDAP Removed
BATCH & MESSAGING	Batch: all 7 cards · Kafka StreamsBuilderFactoryBean · AMQP RabbitRetryTemplate · STOMP SimpDestinationMatcher · Pulsar Reactive Removed
DATA & SESSION	MongoDB property renames · Elasticsearch Rest5Client · Spring Session changes · Undertow Removed
TEST INFRASTRUCTURE	@MockBean / @SpyBean Removed · MockitoTestExecutionListener · @SpringBootTest + MockMvc · Test Slice Starters Relocated · Testcontainers 2.0 Packages · TestRestTemplate Relocated · @PropertyMapping Relocated · Spock Integration Removed

C

Behavioural Risks — Catch in Test, Don't Sequence

These ship the moment you deploy. You can't schedule them; you can only test for them. Treat them as monitoring and QA targets, not work items. Roll out behind feature flags or canary where the change is observable. See the Harder Than It Looks page for the highest-risk items.

CROSS-CUTTING	⚠ Path Matching: security audit required · Jackson date serialisation (×3) · Locale format (zh_CN → zh-CN) · ⚠ @Retryable + @Transactional order · maxAttempts semantics · @NullMarked defaults

OBSERVABILITY

⚠ AspectJ Weaving for @Observed
⚠ Controller Spans Disabled
⚠ OTLP/HTTP Primary (port change)
Liveness/Readiness by Default
Modular Auto-Config

RUN OPENREWRITE FIRST

18 cards have a recipe. Run the full set on a clean branch before any manual changes — it handles most of Category A automatically. Commit its output as one reviewable diff so humans only review the parts that need human judgement.

ONE SUBSYSTEM PER PR

Hibernate, Security, Batch and Test infrastructure each touch overlapping code paths. Merging them separately keeps reviews tractable and makes rollback possible if one subsystem hides a regression.

CATCH CATEGORY C IN STAGING

Path matching, Jackson dates, retry order, OTLP port — these change under your feet on first deploy. Run the full upgraded build in staging with production-like traffic before promoting. Feature-flag what you can.

SECURITY ANNEX

Security Changes

Spring Security 7 · Spring Boot 4.0 · 7 breaking changes

All security-related breaking changes in one place for security engineers and architects. Three changes require significant rework (L); all others are targeted substitutions. One change has an OpenRewrite recipe (⚙). Review each entry against your authentication and authorisation surface before starting the migration.

TIER	IMPACT	CHANGE	REQUIRED ACTION
T1	L	**Security DSL Rewrite** ⚙ `authorizeRequests()`, `antMatchers()`, and `.and()` chaining removed. Entire HTTP security configuration API replaced.	Replace all `authorizeRequests()` → `authorizeHttpRequests()`, `antMatchers()` → `requestMatchers()`, and remove all `.and()` chains. Run the OpenRewrite recipe first; expect manual follow-up on complex custom DSL configurations.
T1	M	**Spring Security Access API Moved** `AccessDecisionManager` and legacy authorisation classes relocated to a separate `spring-security-access` module.	Add `spring-security-access` dependency if using `AccessDecisionManager`, `AccessDecisionVoter`, or `AffirmativeBased`. Consider migrating to `AuthorizationManager` instead.
T1	M	**ApacheDS Embedded LDAP Removed** `ApacheDSContainer` removed. Embedded LDAP test support replaced.	Replace `ApacheDSContainer` with `UnboundIdContainer` in test configuration. Add `com.unboundid:unboundid-ldapsdk` to test scope.
T2	L	**OAuth 2.0 Password Grant Removed** Resource Owner Password Credentials grant fully removed from the OAuth 2.0 client and authorization server support.	No migration path within Spring Security. Replace with Authorization Code + PKCE for interactive flows, or Client Credentials for machine-to-machine. Co-ordinate with identity provider and all consuming clients.
T2	M	**PKCE Mandatory for Confidential Clients** PKCE enforced for all OAuth 2.0 clients, including confidential clients. Auth requests without a code challenge are rejected.	Verify your authorization server supports PKCE. Remove any `pkceRequired(false)` opt-outs. Test all OAuth client integrations, particularly service-to-service flows that previously skipped PKCE.

TIER	IMPACT	CHANGE	REQUIRED ACTION
T2	L	**OpenSAML 4 Support Removed** SAML 2.0 login now requires OpenSAML 5. Applications with `opensaml:opensaml-saml-impl:4.x` fail at startup.	Upgrade to `opensaml:opensaml-saml-impl:5.x` and `opensaml:opensaml-saml-api:5.x`. Review the OpenSAML 5 migration guide for API changes. Retest all SAML SSO flows end-to-end with your identity provider.
T3	M	**Path Matching: AntPathMatcher → PathPattern** Security path matching switched to `PathPattern`. Wildcard behaviour differences can cause routes to match or not match unexpectedly.	Audit all `requestMatchers()` patterns for wildcard differences: `/**` and `/*` behave differently under PathPattern. Write integration tests covering your security rules before migrating. Pay attention to patterns protecting admin and actuator endpoints.

Impact key: L Architectural change, plan a sprint M Targeted change, hours to a day ⚙ OpenRewrite recipe available — run it first, then verify manually. See individual tier cards for diffs, error output, and detailed fix instructions.

Harder Than It Looks

These changes carry an S or M impact badge — but each has hidden complexity that can blow up the estimate. Read these before you plan the sprint.

HIDDEN RISK — IMPACT LIKELY UNDERSTATED

CARD	TIER	IMPACT	RISK	WHY IT'S WORSE THAN IT LOOKS
AspectJ @Observed	T2	S	HIGH	**Completely silent.** Observability via `@Observed` stops working with no errors or warnings. You find out when dashboards go blank in production.
OTLP HTTP Primary	T2	S	HIGH	**Silent production failure.** Traces and metrics vanish. Requires network policy changes (port 4317→4318) and possibly different vendor endpoints.
Batch Schema Rename	T2	S	HIGH	**Not just a rename.** Column types and table structures changed. Rolling deploys break. Tests pass on H2 while production fails on real databases.
spring-jcl Removed	T1	S	HIGH	**Enterprise POM cascade.** The `commons-logging` exclusion pattern used for a decade is now harmful. Corporate parent POMs silently break downstream modules.
Hibernate Processor	T1	S	HIGH	**Unbounded compile failures.** The new processor validates HQL at build time. A one-line GAV change surfaces every broken query in the entire codebase at once.
Jackson Locale Format	T3	S	HIGH	**Silent data corruption.** Locale strings flip from `zh_CN` to `zh-CN`. Database equality checks silently fail in i18n-heavy applications.
Path Matching Engine	T3	M	HIGH	**Security hole.** URL patterns that silently stop matching can expose previously-protected endpoints. Not just a broken feature — a potential vulnerability.
Retryable + Txn Order	T3	M	HIGH	**Near-impossible to unit-test.** AOP ordering change causes rolled-back data before retry. Subtle data consistency issues that only appear under concurrent load.
Gradle Version	T1	S	MOD	**Multi-step upgrade.** Can't jump straight to 8.14. Must stage through major boundaries; Gradle 9 enables configuration cache by default — breaks many plugins.

Jackson Dates/Timestamps	T3	M	MOD	**Compounding changes.** Interacts with the separate date-format change card. Two Jackson date shifts at once makes root-cause debugging doubly confusing.
Test Slice Relocated	T2	S	MOD	**Silent test degradation.** Tests may load the full application context instead of a thin slice. Tests pass but are slower and less precise — until they time out in CI.

> **Pattern:** The most dangerous changes are Tier 3 "Wrong Results" and Tier 2 silent failures. They don't break your build — they break your production data, your observability, or your security posture. Budget extra time for these even when the per-card effort badge is Small.

After the Migration

If you've worked through this guide and your build is green, your tests pass, and your application is running on Spring Boot 4.0, the obvious thing to do is celebrate. Do that. I expect you found more problems than you expected.

Then come back and read the Tier 3 cards one more time, with production traffic in front of you. Behavioural changes don't show up against a synthetic test fixture. They show up when real users do real things, often in the boring corners of the application that nobody pays attention to until something breaks. The migration isn't finished when the build is green. It's finished when you've watched the upgraded system serve its actual workload for a couple of weeks and nothing has surprised you.

The Spring team will keep moving. There's a 4.1, a 4.2, eventually a 5.0. Most of those will be small. Some will not. Subscribe to the Spring Blog, watch the Spring Boot release notes wiki, and bookmark the Further Reading page in this guide as a starting point for the next round. The cheapest migration is the one you saw coming.

This guide stops at the catalogue of breaking changes. The actual upgrade plan is something you have to write yourself, with your team's context, your codebase's quirks, and your roadmap's constraints in mind. The Migration Playbook is a starting framework. Adapt it. If you find a better one, let me know.

That's the stance behind everything in this book. Move forward. Don't break the past. If you're still on Spring Boot 3.5 and the timing isn't right yet, that's a defensible position too. Staying on a maintained 3.x until your team has the bandwidth to do this properly is often the better engineering call. The About the Sponsor page covers the option of extended support if it applies.

About the Sponsor

HeroDevs

HeroDevs

Never-Ending Support for end-of-life open-source software

This guide was produced with support from **HeroDevs**. Their backing made it possible to go through every breaking change in Spring Boot 4.0 systematically and produce something more useful than a changelog summary.

HeroDevs was founded on a straightforward premise: a lot of critical software runs on open-source libraries that have officially reached end-of-life, and most organisations don't have a credible plan for what happens next. Their **Never-Ending Support (NES)** product provides security patches, CVE fixes, and compliance documentation for frameworks and runtimes after their official support windows close.

SPRING BOOT NES

If migration to Spring Boot 4.0 isn't on the table right now — whether due to timeline, cost, or codebase complexity — HeroDevs NES for Spring Boot provides continued security coverage for 3.x after end-of-life. Same API, same behaviour, ongoing CVE fixes and compliance reports.

Their support of this guide reflects a shared interest in helping teams

---

WEBSITE
herodevs.com

SPRING NES
herodevs.com/support/spring-nes

CONTACT
spring-nes@herodevs.com

What's At Stake

39	27	17
BUILD FAILURES	RUNTIME CRASHES	SILENT BUGS

Spring Boot 3.5 reaches end of support in June 2026. After that date, no security patches, no CVE fixes, no bug fixes. Every unpatched vulnerability is a compliance risk.

Option 1: Migrate Now

Start the migration while you still have time. Use this guide and OpenRewrite to move systematically.

Option 2: Migrate Incrementally

Use OpenRewrite recipes to automate what you can. Fix the rest module by module. Track progress against this checklist.

Option 3: Stay on 3.5 Safely

HeroDevs Never-Ending Support (NES) provides security patches, CVE fixes, and compliance certifications for Spring Boot after end-of-life. Same SLAs, same API — no migration required.

Talk to HeroDevs

herodevs.com/support/spring-nes

spring-nes@herodevs.com

Free migration assessment available — we'll estimate your specific migration effort.