ChatGPT

Spring Internals by Observation – Debugging Practice

Abstract

Traditional Spring tutorials teach annotations and configuration recipes but often leave developers unaware of how the container actually behaves at run time. This report documents an **observation-driven** approach to understanding Spring's internals. By instrumenting the container via lifecycle hooks, we examine how beans are created, initialized, proxied, invoked and destroyed. The method enables deterministic debugging of **Spring AOP**, **dependency injection**, **transactions**, **security** and other subsystems and yields a repeatable checklist for diagnosing common problems.

Introduction

Spring is built around an **Inversion of Control (IoC) container** that manages the lifecycle and wiring of application objects. When things go wrong—transactions silently failing, aspects not firing, or classes being replaced with proxies—developers often blame "Spring magic." In reality, these behaviours follow well-defined phases and extension points.

The core observation tool in this study is the `BeanPostProcessor` **(BPP)**. According to the official API, a BPP applies custom logic **before** and **after** the initialization callbacks of each bean [1] [2]. During `postProcessBeforeInitialization` the bean instance already exists with injected dependencies, and the returned instance may wrap the original. During `postProcessAfterInitialization`, the bean may be replaced with a proxy that adds cross-cutting behaviour. By logging these phases we can see the container in action.

We also contrast BPPs with `BeanFactoryPostProcessor` **(BFPP)**. A BFPP runs after the container's standard initialization and may modify **bean definitions**, but **never** bean instances; doing so would violate the container semantics [3]. BFPPs therefore operate at the blueprint stage and run before any beans are instantiated [4].

Understanding **proxy creation** is critical. Spring uses JDK dynamic proxies for beans that implement interfaces and falls back to **CGLIB** (a subclass-based proxy) for concrete classes. The official documentation notes that JDK proxies are built into the JDK and are used when the target object implements at least one interface; otherwise a CGLIB proxy is created [5]. Final classes or methods cannot be proxied because they cannot be subclassed [6]. These differences explain many bugs, such as `ClassCastException` or lost advice.

Methodology: Observation-Driven Debugging

Instead of relying on guesswork, the study created a small Spring application and added instrumentation:

1. **Definition-phase logger** – a BFPP that prints each `BeanDefinition` before any beans are created. It shows scope, lazy initialization flags and bean class names. This clarifies which beans will be created eagerly or lazily.
2. **Lifecycle logger** – a BPP that logs `BEFORE_INIT` and `AFTER_INIT` for application beans. It uses `AopUtils.isAopProxy` and `AopUtils.getTargetClass` to detect proxies and prints the proxy type (JDK vs CGLIB). It demonstrates when a bean is replaced with a proxy and when aspects are applied.
3. **Destruction logger** – a `DestructionAwareBeanPostProcessor` that logs beans during shutdown. This shows which beans receive destroy callbacks and highlights that prototype beans are not managed beyond instantiation [7].
4. **Self-invocation drill** – a simple service with `outer()` calling `inner()` inside the same class, and an aspect around `inner()`. Invoking `inner()` directly through the proxy triggers advice, but calling it from `outer()` bypasses the proxy. This demonstrates why self-invocation breaks AOP.

All instrumentation prints to logs rather than interfering with business logic. Developers can enable the logger in a profile to observe the container in staging or tests.

Bean Lifecycle Phases

The complete lifecycle of a Spring bean includes:

1. **BeanDefinition phase** – definitions are loaded, then BFPPs may modify them [3].
2. **Instantiation** – the container constructs the bean and injects dependencies.
3. **Before-initialization callbacks** – registered BPPs run `postProcessBeforeInitialization` [1].
4. **Initialization callbacks** – `@PostConstruct`, `afterPropertiesSet` and custom init-methods run.
5. **After-initialization callbacks** – BPPs run `postProcessAfterInitialization`, potentially replacing the bean with a proxy [2].
6. **Runtime method invocation** – proxies intercept method calls; aspects such as transactions, logging, security and retry run here.
7. **Destruction phase** – when the context closes, `DestructionAwareBeanPostProcessor` callbacks fire and `@PreDestroy` and destroy-methods are invoked.

Bean scopes and lazy initialization

Spring supports multiple scopes, but the report focuses on **singleton** and **prototype**. Singleton beans are created eagerly (unless marked as lazy) and managed throughout the container's life. Prototype beans are created each time they are requested and **their destruction callbacks are not called**; the client must clean up resources [7].

Eager instantiation is often desirable because misconfigurations surface immediately. To defer creation, the `@Lazy` annotation or `lazy-init` attribute instructs the container to create a bean only when first requested [8]. However, if a lazy bean is a dependency of a non-lazy singleton, it will still be created at

startup to satisfy the dependency [9]. Developers must therefore place `@Lazy` on consuming beans or configuration classes to achieve fully lazy behaviour [10].

Proxying Mechanisms and AOP Internals

JDK dynamic proxies vs CGLIB proxies

Spring AOP chooses a proxying strategy automatically. If the target implements at least one interface, Spring uses a JDK dynamic proxy; otherwise it uses CGLIB [5]. The distinction matters:

- **JDK dynamic proxies** create a runtime class implementing the same interfaces as the target. Only interface methods are proxied. You cannot cast the proxy to the concrete implementation.
- **CGLIB proxies** subclass the target class and intercept method calls. They can proxy classes without interfaces but cannot proxy final classes or final methods [6]. Private methods and package-private methods in a different package are also not advised [11].
- Self-invocation: calling a proxied method from within the same class bypasses the proxy and thus the advice. AspectJ compile-time or load-time weaving does not have this issue [12] because it weaves advice directly into bytecode.

Understanding proxy types helps diagnose `ClassCastException`, missing advice on non-interface methods, and other AOP anomalies.

Cross-cutting concerns and advice ordering

During `postProcessAfterInitialization` the container may wrap the bean with a proxy that adds interceptors. The advice chain order is deterministic: it typically follows `@Order` values or the order in which proxies/advisors are registered. For example, one can see invocation identifiers, logging, auditing, metrics, transaction management, security checks and retry logic executed in sequence. Exceptions propagate through the chain: a security exception triggers a transaction rollback, while a runtime exception may cause a retry attempt and then rollback.

Extending the Container: BPP vs BFPP

A **BeanPostProcessor** deals with **instances**; it can validate beans, inject additional collaborators, wrap beans with proxies, or collect metrics. It runs *after* dependencies are injected but *before* initialization callbacks, and again after initialization [1] [2].

A **BeanFactoryPostProcessor** deals with **bean definitions**. It allows modification of property values, scope and other metadata, but cannot access bean instances. Interacting with instances in a BFPP can cause premature instantiation and side effects [3]. BFPPs run before any beans are created, making them ideal for overriding property values or registering additional bean definitions [4].

Understanding this separation prevents misuse of post-processor hooks and clarifies when modifications take effect.

Findings and Debugging Insights

By running the instrumentation against a sample application, several insights emerged:

1. **Target vs Proxy identity** – The container creates both a target object and a proxy. The BPP logs the target during instantiation; dependency injection returns the proxy. This explains why `getClass()` might return an unfamiliar class such as `jdk.proxy2.$Proxy32`.
2. **Aspect beans are not advised** – Aspect classes themselves are plain singletons; they provide advice but are not proxied.
3. **@Configuration classes are CGLIB-enhanced but not AOP proxies** – Configuration classes are subclassed via CGLIB to ensure that `@Bean` methods return singletons, but `AopUtils.isAopProxy` returns false.
4. **Self-invocation bypasses proxies** – A method calling another method on `this` does not go through the proxy. To apply advice, refactor the logic into separate beans or use AspectJ weaving.
5. **Lazy initialization nuance** – Marking a bean as lazy does not prevent its creation if a non-lazy singleton depends on it [9]. Place `@Lazy` on the consuming bean or configuration class for full laziness [10].
6. **Prototype beans are not destroyed** – The container instantiates prototypes but does not invoke destruction callbacks; the client must clean up resources [7].
7. **Ordering matters** – Implement `PriorityOrdered` or `Ordered` to control the sequence of BPPs and BFPPs [13]. Without ordering, subtle behaviour changes may occur when multiple post-processors interact.

These insights allow systematic diagnosis of misbehaving beans, missing advice, unexpected proxies and resource leaks.

Top 10 Questions When Debugging a Spring Application

Developers often ask similar questions when investigating Spring issues. Below is a curated list of **10 high-leverage questions**, each tied to the concepts above.

1. **Was the bean created at startup or lazily?** Check the scope and `@Lazy` annotation. Lazy beans are created only when first requested [8], but they may be instantiated early if injected into a non-lazy singleton [9].
2. **Which BeanPostProcessors are running on this bean?** Inspect the application context for BPP beans and their order. Misconfigured BPPs can wrap or replace beans unexpectedly.
3. **Is this bean a JDK proxy or a CGLIB proxy?** Use `AopUtils.isAopProxy()` and check whether the runtime class name starts with `jdk.proxy` or contains `$$SpringCGLIB`. Interface-based proxies only advise interface methods [5].
4. **Why does my `@Transactional` / `@Secured` / `@Retryable` not work?** Ensure that the proxied bean is injected (not the target), that the method is public, and that calls are not self-invocations. Final classes or final methods cannot be proxied [6].
5. **Are my dependencies correctly injected?** Check for multiple candidates requiring `@Primary` or `@Qualifier`, and verify that the container has created the dependent beans.

6. **Are prototype beans leaking resources?** Remember that the container does not destroy prototype beans [7]. Use a custom cleanup mechanism or convert the prototype to singleton with explicit management.
7. **Is the bean definition modified by a BeanFactoryPostProcessor?** BFPPs can override properties or add definitions before instantiation [3]. Unexpected property values may originate here.
8. **In what order are aspects applied?** Review `@Order` values on aspects; ordering influences security checks, transaction boundaries and logging. Test chains manually to verify expected behaviour.
9. **Are initialization callbacks executed?** Ensure that `@PostConstruct` methods run; BPPs should not replace beans before initialization unless necessary. Initialization issues often manifest as null dependencies or half-initialized beans.
10. **What happens on shutdown?** Confirm that destroy callbacks run for singletons, and use a `DestructionAwareBeanPostProcessor` to log or clean up resources. Make sure the application context is closed properly.

These questions act as a debugging checklist. By tracing the lifecycle and proxying behaviour, developers can quickly locate the cause of misbehaviour.

Conclusion

The **Spring Internals by Observation** approach transforms Spring from a perceived “black box” into a predictable system. By instrumenting the container with post-processors, this study demystifies bean creation, proxy substitution and method interception. Official documentation confirms that BPPs operate on bean instances before and after initialization [1] [2], while BFPPs modify definitions before any beans exist [3]. Proxy selection (JDK versus CGLIB) is governed by whether the target implements interfaces [5], and final classes or methods cannot be proxied [6]. Prototype beans are not destroyed by the container [7], and lazy initialization has limitations when dependencies are non-lazy [9].

By combining these documented facts with runtime observation, developers gain a reliable, multi-perspective debugging model. The **top 10 questions** provide a pragmatic checklist to diagnose common Spring issues. Mastery of these internals enables confident engineering of complex applications, integration of cross-cutting concerns and effective troubleshooting of misbehaviour.

End Note

Understanding Spring’s internals requires patience, experimentation and a willingness to look beyond annotations. Treating mistakes and misconfigurations as academic data points, as done in this report, turns confusion into insight. With this knowledge, developers can move from “magic” to **mechanics**, building robust, maintainable Spring applications.

---

[1] [2] BeanPostProcessor (Spring Framework 7.0.2 API)
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanPostProcessor.html

3 4 13 BeanFactoryPostProcessor (Spring Framework 7.0.2 API)
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanFactoryPostProcessor.html

5 6 11 12 Proxying Mechanisms :: Spring Framework
https://docs.spring.io/spring-framework/reference/core/aop/proxying.html

7 Bean Scopes :: Spring Framework
https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html

8 9 10 Lazy-initialized Beans :: Spring Framework
https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-lazy-init.html