Find More: www.javapro.io

13 BUILDING A SELF-SERVICE PRICE INTELLIGENCE AGENT WITH SPRING AI 2.0 & MCP: AN EVENT-DRIVEN APPROACH FOR SMALL RETAILERS

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

Edition 9 / 2026

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

#JAVAPRO

MODERN JAVA IN PRACTICE

29 RESILIENCE4J IN PRODUCTION: THE FAILURE PATTERNS THAT ACTUALLY MATTER

38 THE NEXT CHAPTER OF KUBERNETES & JAVA: JAVA APPS & DYNAMIC RESOURCE ALLOCATION

58 DON'T TRUST THAT PACKAGE: APPLYING ZERO TRUST TO OPEN SOURCE

74 POLYGLOT PROGRAMMING WITH GRAALVM: CALLING PYTHON, JAVASCRIPT, RUBY, & R FROM JAVA

84 BEYOND COLD STARTS: OPERATING RESTORED JAVA SERVICES WITH OPENJ9 CRIU SUPPORT

JCON USA 2026
at IBM TechXchange
Oct 26 - 29, 2026
www.jcon.one

Editorial

Modern Java in Practice

Modern Java is not defined by a single language feature, framework, or architectural trend. Its real strength becomes visible when technology meets reality: when applications have to scale, recover, remain observable, integrate with other ecosystems, and stay secure under constantly changing conditions. That is what this issue is about.

The articles in this edition look beyond what modern Java can do and focus on how it actually works in practice. That distinction matters. A technology may look convincing in documentation or in a proof of concept, but production environments introduce their own rules. Resources change, dependencies fail, execution moves across threads, security assumptions become outdated, and architectural choices that once seemed obvious deserve to be questioned again.

Several contributions explore exactly these operational realities. Restoring warmed Java applications with OpenJ9 and CRIU, for example, can dramatically change startup behavior, but operating a restored JVM also introduces new questions around diagnostics, tuning, and reliability. Kubernetes is evolving as well, opening the door to more dynamic resource allocation for Java applications instead of treating CPU and memory as purely static boundaries.

Resilience and observability reveal another side of the same challenge. Patterns such as retries, circuit breakers, and bulkheads are easy to name but much harder to apply correctly when real failures occur. Reactive applications create similar surprises: a trace ID that appears reliable in a traditional request model can suddenly disappear when execution crosses asynchronous and reactive boundaries.

Modern Java development also means reconsidering established assumptions. Do we really need a single-page application for every modern web interface? Server-driven approaches and hypermedia show that rich user experiences do not automatically require another layer of complexity. GraalVM takes the idea of choosing the appropriate tool even further, allowing Java applications to work directly with Python, JavaScript, Ruby, and R when another language is better suited to a particular task.

And modern systems cannot be discussed without security. Post-quantum cryptography is beginning to influence the foundations of secure communication, while software supply chains require us to reconsider what it means to trust a dependency. A familiar package name, a popular repository, or a clean vulnerability scan is no longer sufficient evidence on its own. Trust increasingly has to be verified and continuously reassessed.

In the end, modern Java is not defined by what is possible in theory, but by what works in practice.

We hope this issue gives you new ideas, practical insights, and a few good reasons to reconsider how you build and operate your own Java systems.

My very special thanks go to all of the fantastic authors of this issue.

Enjoy this edition of JAVAPRO.

Author: Markus Kett

Markus Kett and his teams have been working on IDE tools for Java and database development, as well as various open-source projects for 20 years. Markus is CEO and co-founder of MicroStream, the company behind the Eclipse open-source projects EclipseStore, Eclipse Serializer, and RapidClipse IDE. He is also the editor-in-chief for the free JAVAPRO magazine in Germany and the founder and co-organizer of the Java community conference series JCON. He is an independent editor for several magazines, and a speaker at numerous international developer conferences, user groups, and meetups.



LinkedIN

Contents

Editorial **2**
Modern Java in Practice . . . 2

JCON *Richard Fichtner*

Come for Java. Get a Whole Lot More. **7**
From Orlando to Atlanta . . . 8
Three Ways to Experience JCON USA . . . 8
Session Highlights . . . 9
A Complete Technical Conference Around JCON . . . 10
A Home for the Java Community . . . 10
Event Information . . . 10

AI & ML *Soham Sengupta*

Building a Self-Service Price Intelligence Agent with Spring AI 2.0 and MCP **13**
Generic Use Case: . . . 14
A Specific Use Case: . . . 14
System Analysis: Why This Should Be Event-Driven, Not Real-Time . . . 15
A Note on Scope . . . 21

Architecture & Microservices *Suleyman Yildirim*

Why Your Trace IDs Disappear in Reactive Java Application (And How To Fix It) **23**
Problem Description . . . 24
What Is MDC? . . . 24
Why It Works in Spring MVC? . . . 25
Why It Breaks in Project Reactor? . . . 25
Solution . . . 26
Lessons Learned . . . 28

Architecture & Microservices *Yasin Baytürk*

Resilience4j in Production: The Failure Patterns That Actually Matter **29**
Why Resilience4j . . . 30
The Four Patterns . . . 30
Circuit Breaker . . . 31
Retry . . . 33
Bulkhead . . . 33
Rate Limiter . . . 35
The Order of Decoration . . . 35
Observability with Micrometer . . . 36
Production Anti-Patterns . . . 36
Recommended Defaults . . . 36
Conclusion . . . 37

Cloud *Abdel Sghiouar*

The Next Chapter of Kubernetes and Java: Java Apps and Dynamic Resource Allocation **38**
Under the Hood: How In-Place Resizing Works . . . 39
The resizePolicy: Fine-Grained Control . . . 40
CPU Resizing . . . 40
Memory Resizing . . . 41
VPA CPU Startup Boost . . . 41

Operating with the New Patterns . . . 42
Conclusion . . . 43

Web Development *Hyyan Abo Fakher*

The Server-Driven UI Renaissance **44**
What "Server-Driven" Actually Means: Three Architectures, Not One . . . 45
Why the Pendulum Swung Back to Server-Driven UI . . . 46
The Four Hard Problems Every Stateful Server Tree Must Answer . . . 47
What a Java Team Actually Chooses Between . . . 48
When Not to Reach for This . . . 48
Where Server-Driven UI Genuinely Wins . . . 48

Web Development *Frederik Hahne*

Joyful Web Development with Hypermedia-Driven Applications and HTMX **50**
The Problem with Complexity . . . 51
What Is Hypermedia? . . . 51
HTML as Hypermedia . . . 52
HTMX – HTML Extended . . . 52
JTE – A Modern Template Engine . . . 52
HTMX Patterns . . . 53
Further Features . . . 55
Conclusion . . . 56
Related Links . . . 56

Security *Catherine Edelveis*

Don't Trust That Package: Applying Zero Trust to Open Source **58**
What Is Zero Trust . . . 59
Why 5K Stars on GitHub Is Not Grounds for Blind Trust in OSS . . . 59
Applying Zero Trust Principles to Open Source Software . . . 60
Implementing Controls Throughout the Software Lifecycle . . . 60
Security Signals and What They Prove . . . 62
Guidelines You Should Start With . . . 62
Possible Gradual Introduction of Controls . . . 63
Before You Leave: Open-Source Sustainability as a Security Control . . . 63

Security *Babneet Singh*

Java 27's Post-Quantum TLS Handshake: How JEP 527 Brings Hybrid Key Exchange to JSSE **66**
The TLS Traffic Recorded Today May Matter Tomorrow . . . 67
What Java 27 Changes, and Why It Is Hybrid . . . 67
The Centrepiece: Two Key Shares in the First ClientHello . . . 68
What Happens on the Hybrid Path . . . 69
What the JDK Actually Built . . . 69
Post-Quantum Security Has a Wire Cost . . . 70
What Java Applications Get Automatically . . . 70
How Configuration Can Silently Turn It Off . . . 71
Verify the Handshake, Not the Source Code . . . 71
What About OpenJ9 and Eclipse Semeru? . . . 72
Production Rollout . . . 72
The Real Takeaway . . . 72
References . . . 73

JVM Languages — *Atiq Amjad*

Polyglot Programming with GraalVM: Calling Python, JavaScript, Ruby, and R from Java **74**
Why Polyglot Programming? . . . 75
GraalVM's Polyglot Architecture . . . 75
The Polyglot API: Core Concepts . . . 76
Prerequisites & Dependencies . . . 77
Building the Polyglot Application . . . 78
Data Exchange and Interoperability . . . 80
Security and Sandboxing . . . 81
Comparison with Traditional Approaches . . . 82
Enterprise Use Cases . . . 82
When Polyglot Is (and Isn't) the Right Choice . . . 83
Conclusion . . . 83

Performance — *Babneet Singh and Tobi Ajila*

Beyond Cold Starts: Operating Restored Java Services with OpenJ9 CRIU Support **84**
A Checkpoint Image Is a Baseline, Not a Finished Deployment . . . 85
Restore-Time Controls in OpenJ9 . . . 86
Debugging a Restored JVM . . . 88
Conclusion: Fast Startup with Operational Control . . . 89
References . . . 89

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors







Gold Sponsors





Silver Sponsors

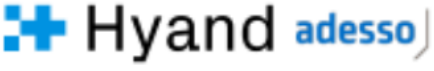








Bronze Sponsors











JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92681 Erbendorf
Germany

E-Mail: info@javapro.io
Website: http://www.javapro.io

Editor in Chief:
Markus Kett (V.i.S.d.P.)

Editorial:
info@javapro.io

Design, Layout & Print:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
Germany

Copyright (c) 2026
Impuls Mediengruppe GmbH

All rights reserved.

Java(TM) is a registered trademark of Oracle Corporation.

Javapro is an independent magazine and is not sponsored by Oracle Corporation.

Articles marked with a name do not necessarily reflect the opinion of the editors.

The images featured in this issue are sourced from royalty-free platforms such as Pixabay and Unsplash, contributed by our authors, or creatively generated with the help of artificial intelligence.





Come for Java. Get a Whole Lot More.

Community **# Conference** **# Java**

Author: Richard Fichtner

Richard Fichtner is the Principal Software Architect at XDEV Software GmbH, with over 20 years of experience in the software industry. He works at the intersection of business and technology, contributing to the open-source community through projects like RapidClipse (https://www.rapidclipse.com) and advocating for best practices in Apache Maven. Richard is passionate about sharing knowledge on Java technologies, making him a frequent conference speaker. He leads the Java User Group Oberpfalz, co-organizes the JCON conferences (https://jcon.one), and has been recognized as a Java Champion, Oracle ACE Pro, IBM Champion and Vaadin Champion. With a Master of Science in Applied Computer Science, his expertise spans Java, clean code, cloud technologies, and pragmatic development practices, focusing on improving developer productivity through cloud solutions. Richard enjoys relaxing by a campfire and listening to/playing country music in his spare time.



LinkedIN

JCON USA returns inside IBM TechXchange 2026 with two rooms of deep Java content, a dedicated community home, and direct access to one of the year's biggest technical conferences.

From Orlando to Atlanta

Last year, we tried something new. We brought JCON USA into IBM TechXchange and created a Java community conference inside a much larger technology event. The result showed us that these two worlds fit together remarkably well. JCON brought the independent, developer-to-developer spirit of the Java community. TechXchange provided the scale, infrastructure, learning opportunities, and access to experts that only a major international conference can offer.

In 2026, we are taking the next step. JCON USA comes to Atlanta from October 26 to 29 at the Georgia World Congress Center. JCON USA is the Java community conference within IBM TechXchange 2026.



The Georgia World Congress Center in Atlanta, where JCON USA takes place inside IBM TechXchange.

For me, this creates a pretty cool combination. You can spend your morning in a deep Java session, join a hands-on lab afterwards, explore the Sandbox, talk with an IBM engineer, meet people working in AI, data, infrastructure, security, or hybrid cloud, and then return to the JCON Community Hub to continue the Java conversation. You do not have to choose between JCON and TechXchange. You get both.

Three Ways to Experience JCON USA

The structured technical program is the backbone of the event. From Tuesday through Thursday, two dedicated JCON rooms will host 23 breakout sessions. Each session has 45 minutes, including Q&A. The speakers include Java Champions, Java User Group leaders, experienced developers, architects, and other respected voices from across the ecosystem. The focus is practical: Java, Jakarta EE, architecture, AI, cloud native development, open source, developer productivity, and the technologies shaping our daily work.

But a good conference is more than a sequence of presentations. That is why the JCON Community Hub matters so much. It is our home base inside TechXchange and the place where the Java crowd comes together throughout the conference. You can meet speakers, Java Champions, JUG leaders, fellow developers, and the engineers behind IBM's Java technologies. There will be Tech Talks, Meet the Experts sessions, office hours, and informal discussions. You can also simply come over, sit down, and talk. Even with thousands of technologists around you, you will always know where to find the Java community.



Attendees on the conference floor.

The third part is the JCON Mentorship Hub. Sometimes you do not need another presentation. You need a conversation. The Mentorship Hub gives you direct access to experienced people who are willing to discuss a technical problem, an architectural decision, open source, career development, leadership, community work, or another topic that matters to you. There is no sign-up and no complicated process. Check the schedule, find a mentor or topic, walk up, and join the conversation.

Session Highlights

The 2026 program reflects a Java ecosystem that is modernizing without losing its engineering foundations. AI is an important part of that story, but the sessions do not treat it as a collection of demos or vague promises. They ask what it takes to make AI useful, maintainable, observable, and safe in real Java systems.

Emily Jiang and Ivar Grimstad bring the perspective of people who help shape enterprise Java standards. Both are Java Champions. Emily is a long-time MicroProfile and Jakarta EE specification leader; Ivar is Jakarta EE Developer Advocate and EE4J PMC Lead. In "AI in Java: No Python Required," they connect that standards experience with a practical question: how can teams add AI through Jakarta EE, MicroProfile, and langchain4j-cdi while staying within familiar Java patterns?

Spencer Gibb brings a similarly direct connection to his subject. As co-founder and lead of Spring Cloud, he has helped build the tools teams use for distributed systems. His session "Of Microservices and Monoliths" draws on that experience to explore when distribution is worth the complexity, and when an application and its team are better served by a monolith.

Open source itself is also part of the conversation. Brian Fox, Sonatype co-founder and CTO and former chair of the Apache Maven project, uses Maven Central as a real-world case study in "Free Isn't Free: Moving From Open Source Abuse to Alignment." The session examines how everyday build and cloud practices can create enormous hidden load for shared infrastructure, and what responsible engineering teams can do about it.

These highlights reflect the JCON approach: useful technical content from people who build, operate, and improve the technologies we depend on. The full breakout and Tech Talk program offers many more opportunities to learn from that experience and continue the discussion in person.

A Complete Technical Conference Around JCON

JCON USA is included in the IBM TechXchange conference pass. There is no separate JCON pass. Attendees can join hands-on labs, technical breakouts, live demos, Meet the Experts sessions, Tech Talks, certification testing, and activities in the Sandbox Expo. The conference spans topics well beyond Java, including AI, data, hybrid cloud, infrastructure, security, and governance. This wider context is valuable because Java rarely exists in isolation in real organizations.

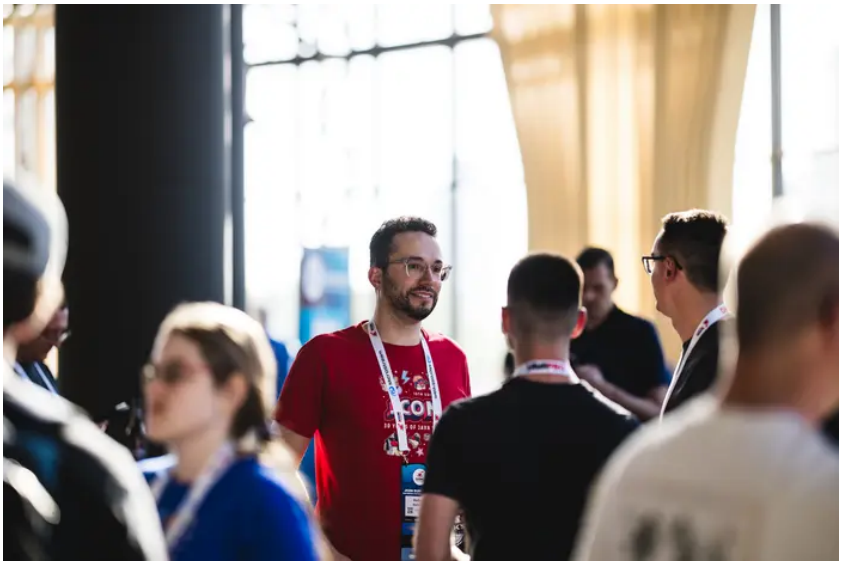
The IBM TechXchange conference pass includes two complimentary onsite certification exam attempts. Breakfast and lunch are provided during the conference, and the shared program includes the Opening Night Block Party in the Sandbox on Monday evening and the TechXchange Networking Event on Wednesday evening.

JCON's community kickoff is the Block Party on Monday, October 26. The two JCON rooms and Hub program begin on Tuesday and continue through Thursday. This rhythm gives attendees a natural way to meet first, then spend three days learning and exchanging ideas.

A Home for the Java Community

Large conferences create opportunities that smaller events cannot. You meet people from different fields, gain access to a much wider program, and discover technologies you did not expect to encounter. But scale can also feel anonymous. JCON USA solves that problem by giving Java developers and architects a clear home inside the larger event.

That home is not defined only by rooms, stages, or signs. It is defined by the people who gather there. JCON is organized by the community, for the community. The sessions are important, but so are the questions after a talk, the discussion over lunch, the introduction to an open-source maintainer, and the honest conversation with someone who has already faced the problem you are working on.



Between the talks: the conversations are half of what a conference is for.

Last year proved that the Java community can bring its own energy into a major technology conference without losing what makes it special. In Atlanta, we want to build on that foundation with more Java content, a stronger home base, and more opportunities for direct exchange.

If you work with Java, Jakarta EE, Spring, Quarkus, the JVM, cloud native systems, open source, or AI-assisted development, join us in Atlanta. Come for Java. Get a whole lot more.

Event Information

Event: JCON USA @ IBM TechXchange 2026

Dates: October 26-29, 2026

Venue: Georgia World Congress Center, Atlanta, Georgia

Website: https://2026.usa.jcon.one/

Sessions: https://2026.usa.jcon.one/sessions

Registration: https://reg.tools.ibm.com/flow/ibm/techxchange26/reg (Link)

› **Back to the Table of Contents** ‹



JCON goes USA

OCT
26-29



JCON USA 2026

at IBM TechXchange

JCON USA 2026

in Atlanta, Georgia

@ IBM TechXchange



Building a Self-Service Price Intelligence Agent with Spring AI 2.0 and MCP

SpringAI # MCP # EventDriven

Author: Soham Sengupta

Soham would like to portray himself as a passionate learner working and delivering to heterogeneous disciplines. Once an avid coder, he loves to think Computers in Physics and Mathematics. He encourages people to become stack neutral and read between the protocols, how systems talk to each other, and then code. The ability to use a programming language and its different frameworks is no more than one's speaking a foreign language fluently—the longer one stays somewhere or interacts with people there, the more fluent one should become. However, fluency might drift once one travels back. Technology, however, is like one's native language, literally and eternal, exhibiting its presence in whichever programming language it is that one currently works on. Still, his bias for Java is for its admirably strong rationale in the language architecture, backward compatibility, and catching up whenever someone doubted it. He teaches, researches, other than practising engineering. And he loves his family the best.



LinkedIN

Small and mid-sized retailers often need to track competitor prices but cannot justify the cost or legal risk of building a full-scale scraping infrastructure. This article presents a lightweight, event-driven price intelligence system built entirely on the Spring AI 2.0 stack. The system reads a retailer's own product catalog page, extracts a list of entities to track, and then uses an agentic search workflow — powered by the Model Context Protocol (MCP) — to look up comparable pricing information from the wider web. Because the workflow depends on external search results and periodic catalog changes rather than continuous data streams, the system is designed to be event-driven rather than real-time, triggered by catalog updates or scheduled intervals. The architecture exposes its capabilities as MCP tools using `spring-ai-agent-utils`, allowing the same logic to be invoked either as a standalone Spring Boot service or as a tool available to external AI assistants such as Claude. The article includes a working code structure — POJOs, a service layer, an MCP tool definition, and configuration — that a developer can extend for their own catalog and search provider.

Generic Use Case:

Any business that sells products online — or even offline with listed prices — periodically needs to know how its prices compare to the wider market. This need typically falls into three recurring scenarios:

1. **Competitive benchmarking.** A seller wants to know whether their price for a given product is higher, lower, or in line with what other retailers charge for the same or similar item.
2. **Dynamic repricing input.** Before adjusting a price during a sale, seasonal event, or clearance, a business wants a quick read of what the market currently looks like, rather than manually checking five or six websites by hand.
3. **Catalogue health monitoring.** As new products are added to a business's own catalogue, there is a recurring need to check where each new item stands relative to existing market prices, without repeating this manually for every SKU.

In all three scenarios, the starting point is the same: a business already has a list of things it sells — a catalog page, a spreadsheet, a product feed. The goal is to take that list of entities and, for each one, find out how it compares to what is available elsewhere on the web. Critically, this does not require scraping competitor websites directly. It only requires:

- Reading the business's **own** data (which it fully owns and controls), and
- Performing a **search** for each entity across the web, using an authorized search interface rather than a custom scraper aimed at specific competitor sites.

This generic pattern — *read your own list → search the web for each item → summarize the comparison* — is reusable across industries: electronics, fashion, grocery, even services and courses. It does not depend on any single marketplace, and it does not require continuous, real-time access to any external system.

A Specific Use Case:

Consider *Masha*, who runs a small electronics retail business selling mobile accessories, chargers, and small home appliances through her own website and a modest physical store. She lists around 80 products on her website, updated a few times a month as stock changes.

Masha does not have a data engineering team, and she has no interest in running a large scraping operation or reselling price data to anyone else. Her need is simple and entirely internal: **once or twice a day, she wants to know how her prices compare to the general market for the same products**, so she can decide whether to adjust her own pricing before a sale, a festival season push, or a new stock arrival.

Her requirements, translated into a system design, look like this:

- **Input**: her own product catalog page (a URL she controls), listing product names, SKUs, and her current prices.
- **Process**: for each product in that list, find out what similar listings elsewhere on the web are currently priced at, using a general web search rather than targeting any single competitor's site directly.
- **Output**: a simple dashboard or report showing, per product, her price alongside a market price range and a short note (e.g., "your price is slightly above the median online price for this category").
- **Frequency**: this does not need to run continuously. Running it once or twice a day — or whenever her catalog changes — is more than sufficient for her decision-making needs.
- **Ownership boundary**: she only scrapes her own page. Everything else comes from a search interface designed for this kind of lookup, not from scripts written against competitor storefronts.

This is a good example because it is small enough to be genuinely low-risk, well within what one person can build and maintain, and yet structurally identical to what a larger business might need — which is exactly why it is worth building properly rather than as a one-off script.

System Analysis: Why This Should Be Event-Driven, Not Real-Time

The workflow consists of four discrete stages: the catalog page is scraped, a price comparison is run for each entity, the result is passed to a business engine responsible for price adjustment, and that engine applies the change to the database — typically subject to Masha's explicit approval rather than automatically. Each stage is triggered by the completion of the previous one, not by continuous polling, which makes an event-driven design the natural fit.

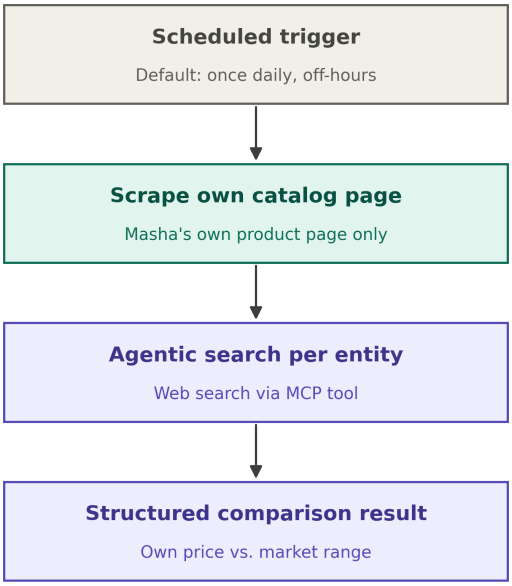
Real-time processing is unwarranted here for three reasons. First, the catalog itself changes only a few times a month, leaving nothing to react to continuously. Second, each comparison carries a measurable cost: a search-API call per entity, plus an LLM call to structure the result, billed per token. Running this continuously across Masha's roughly 80 products would multiply both costs with no gain, since retail prices do not move on a sub-daily timescale. Third, because price adjustments require Masha's approval, there is an inherent human-in-the-loop delay that makes real-time execution moot regardless of data freshness.

A once-daily sweep, scheduled during off-business hours, is a sensible default — though the frequency should remain configurable, since Masha bears the resulting cost and is best placed to balance it against how often she wants fresh visibility.

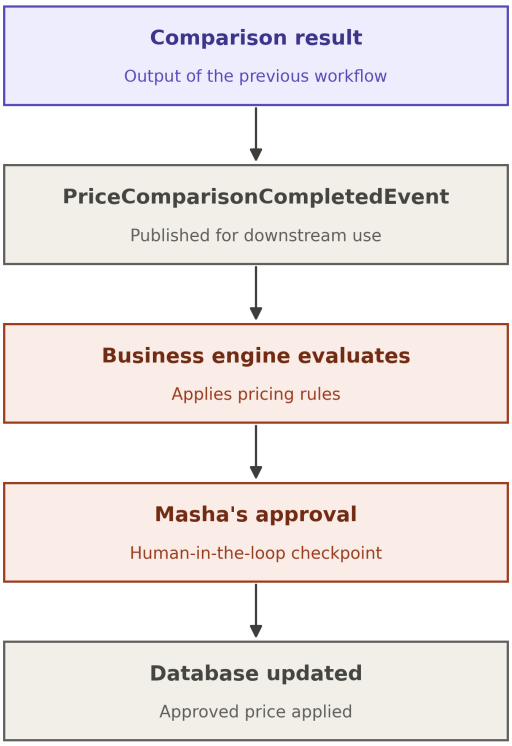
This maps cleanly onto Spring's event and scheduling model:

- A `@Scheduled` job triggers the daily sweep (interval externalized to configuration).
- The **scraping and comparison** run per entity, asynchronously, so one slow lookup doesn't block the batch.
- On completion, a `PriceComparisonCompletedEvent` is published, carrying the recommended adjustments.
- A **business engine** listens for this event, applies its own rules, and — pending Masha's approval — writes the accepted changes to the database.

This keeps the system cheap to run, easy to reason about, and safe by construction: nothing touches the price database without going through the approval step.



The daily sweep: a scheduled job scrapes the shop's own catalogue page, searches the web for each entity through an MCP tool, and ends in a structured comparison of the own price against the market range.



What follows the comparison: the result is published as an event, the business engine applies the pricing rules, and the new price reaches the database only after Masha has approved it.

Architecture and Tooling: Spring AI 2.0, `spring-ai-agent-utils`, and MCP

The workflow described in Figures 1 and 2 is implemented as a Spring Boot application built on Spring AI 2.0, the current generation of Spring's AI integration framework, released as a general-availability version in mid-2026 on the Spring Boot 4.0 platform. Three architectural decisions underpin the design: the use of Spring AI's tool-calling abstraction to structure the agentic search step, the use of `spring-ai-agent-utils` to simplify the orchestration of multi-step agent behaviour, and the exposure of the system's capabilities through the Model Context Protocol (MCP), such that the same logic that serves Masha's internal dashboard can equally be invoked as a tool by an external AI assistant such as Claude.

Spring AI 2.0 provides the foundational abstractions used throughout the system: the `ChatClient` interface for invoking a language model, structured output support for converting a model's response directly into a typed Java object, and a tool-calling mechanism by which a method annotated appropriately becomes callable by the model during reasoning. These abstractions are used, respectively, to interpret search results for a given entity, to produce the `EntityComparison` object referenced in the daily-sweep workflow above, and to expose both the catalog-reading step and the web-search step as callable tools.

`spring-ai-agent-utils` sits above these primitives and addresses a recurring concern in agentic systems: the need to iterate a multi-step reasoning process — read an entity, search for it, structure the result — across a list of items, with appropriate handling of retries, partial failures, and per-item cost tracking, without the developer having to write this orchestration logic by hand for every new workflow. In this system, it is used to drive the per-entity loop shown in that same workflow, ensuring that a single failed or delayed search does not interrupt the processing of the remaining catalog.

The Model Context Protocol (MCP) is the mechanism by which this system's capabilities are exposed beyond the boundaries of the Spring Boot application itself. Rather than building a bespoke API for Masha's dashboard alone, the catalog-reading and price-comparison capabilities are each defined as MCP tools. This has a direct and practical consequence: any MCP-compatible client — including Claude Desktop, Claude Code, or another Spring AI application — can discover and invoke these same tools without additional integration work. Spring AI 2.0 supports this natively, using Streamable HTTP as its default transport, which allows the MCP server to be deployed and reached over a standard web endpoint rather than requiring a local process connection.

The remainder of this section is organized as a set of code provisions, each corresponding to a layer of the implementation: the Maven build definition, the domain objects (POJOs) representing an entity and its comparison result, the service layer implementing the workflow of Figures 1 and 2, the MCP tool definitions that expose this service externally, and the application configuration governing scheduling, endpoints, and credentials. Concrete code is provided in the sections that follow.

- **`pom.xml`** — Maven dependencies for Spring Boot 4, Spring AI 2.0, `spring-ai-agent-utils`, and the MCP server starter.
- **POJOs** — `Entity`, `SearchHit`, and `EntityComparison`, representing the data exchanged between stages of the workflow.
- **Service** — `PriceComparisonService`, implementing the sequence: scrape, search per entity, structure result, publish event.
- **MCP Tool** — the `@McpTool`-annotated methods exposing catalogue reading and entity search to external MCP clients such as Claude.
- **`application.properties`** — configuration for the scheduled refresh interval, off-hours scheduling window, and search API credentials.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>4.0.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>dev.soham.spring.ai</groupId>
    <artifactId>first-agent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>first-agent</name>
    <description>first-agent</description>
    <url/>
    <licenses>
        <license/>
    </licenses>
    <developers>
        <developer/>
    </developers>
    <scm>
        <connection/>
        <developerConnection/>
        <tag/>
        <url/>
    </scm>
    <properties>
        <java.version>25</java.version>
        <spring-ai.version>2.0.0-M2</spring-ai.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
<!-- Mistral is used in this demo  -->
        <dependency>
            <groupId>org.springframework.ai</groupId>
            <artifactId>spring-ai-starter-model-mistral-ai</artifactId>
```

```
        </dependency>
        <!-- We Use Spring AI Agent Utils -->
        <dependency>
            <groupId>org.springaicommunity</groupId>
            <artifactId>spring-ai-agent-utils</artifactId>
            <version>0.4.2</version>
        </dependency>

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

        <dependency>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
            <version>1.18.3</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>

<dependency>
            <groupId>org.springframework.ai</groupId>
            <artifactId>spring-ai-starter-mcp-server-webmvc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springaicommunity</groupId>
            <artifactId>mcp-server-security</artifactId>
            <version>0.1.6</version>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.ai</groupId>
                <artifactId>spring-ai-bom</artifactId>
                <version>${spring-ai.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
```

```
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <executions>
                    <execution>
                        <id>default-compile</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                        <configuration>
                            <annotationProcessorPaths>
                                <path>
                                    <groupId>org.projectlombok</groupId>
                                    <artifactId>lombok</artifactId>
                                </path>
                            </annotationProcessorPaths>
                        </configuration>
                    </execution>
                    <execution>
                        <id>default-testCompile</id>
                        <phase>test-compile</phase>
                        <goals>
                            <goal>testCompile</goal>
                        </goals>
                        <configuration>
                            <annotationProcessorPaths>
                                <path>
                                    <groupId>org.projectlombok</groupId>
                                    <artifactId>lombok</artifactId>
                                </path>
                            </annotationProcessorPaths>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <releases>
                <enabled>false</enabled>
            </releases>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <releases>
```

```
                <enabled>false</enabled>
            </releases>
        </pluginRepository>
    </pluginRepositories>

</project>
```

```
// The Product Info
package dev.soham.spring.ai.firstagent;

public record Product(String productName,String brand, String price) {
}
```

```
// scraper service
@Service
@Slf4j
public class WebScraperService {

    public String scrapeWebPage(String targetUrl) throws IOException {
        var cleanWebpageText = Jsoup.connect(targetUrl)
                .userAgent("Mozilla/5.0") // Good practice to prevent basic bot blocks
                .get()
                .text();

        log.info("WebScraperService :: scrapeWebPage::  {}", cleanWebpageText);
        return cleanWebpageText;
    }

    @Retryable(
            maxRetries = 5,
            delay = 30, multiplier = 2.5, includes = {NonTransientAiException.class},
            maxDelay = 60
            , timeUnit = TimeUnit.SECONDS)
    @Nullable
    public List<PriceComparisonRowItem> getPriceComparisonList(Product product,
        ChatClient chatClient) {
        return chatClient.prompt()
                .system(prompt -> prompt.text(PriceComparisonService.
                    PROMPT_PRODUCT_COMPARISON)
                        .param("product", product))
                .user("Find all products")
                .toolContext(Map.of(PriceComparisonService.SESSION_ID, "user-session-123"
                    )) // dummy id
                .call()
                .entity(new ParameterizedTypeReference<List<PriceComparisonRowItem>>() {
                });
    }
```

```
  @Bean
  public ChatClient agentChatClient(ChatClient.Builder chatClientBuilder) {

         return chatClientBuilder
                .defaultSystem("Always use the available skills to assist the user  in
                    their requests.")
                .defaultToolCallbacks(SkillsTool.builder()
                        .addSkillsDirectory("src/main/resources/skills/")
                        .build()
                        )
                .defaultTools(ileSystemTools.builder().build()) // We don't use it //
                    here
                .defaultAdvisors(new SimpleLoggerAdvisor()) // for coarse logging
                .build();

  }
```

```
spring.application.name=masha-product-comaprsion-agent

# ========================
# Mistral AI Configuration
# ========================
spring.ai.mistralai.api-key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

spring.ai.mistralai.chat.options.model=mistral-large-latest
logging.level.org.springframework.ai=DEBUG

# Retry configuration for Spring AI
spring.ai.retry.max-attempts=5
spring.ai.retry.backoff.delay=2000
spring.ai.retry.backoff.multiplier=2
spring.ai.retry.backoff.max-delay=30000
```

A Note on Scope

The MCP server in this architecture is secured with password-based authentication in a full deployment, following the OAuth 2.0 or API-key patterns supported natively by Spring AI 2.0's MCP integration. That security configuration is not covered here — it is left for a future article. The intent of this piece was to keep the focus strictly on the AI integration itself: how the catalog is read, how the agentic search is structured, and how the result flows through to an approval step. The architecture was discussed only to give a use-case-level overview of how these pieces fit together, not as a production deployment guide.

› Back to the Table of Contents ‹

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

www.javapro.io



Why Your Trace IDs Disappear in Reactive Java Application (And How To Fix It)

Reactive # Tracing # Logging

Author: Suleyman Yildirim

Senior Software Engineer with 15 years of experience in backend development, cloud-native systems, and software architecture. His primary focus areas include Java (version 17 and above), Reactive Programming, Spring Ecosystem, AWS, Microservices, and DevOps practices. He has delivered projects in diverse industries, including banking, public sector, media, and SaaS. Key projects include work with ING Hubs, Vodafone UK, HMRC, and West Midlands Police. His recent roles at ING, BeatStars, and Smartsheet has allowed him to lead end-to-end development initiatives, CI/CD pipeline implementations, and architectural improvements.



LinkedIN

A deep dive into ThreadLocal, MDC, and context propagation in Project Reactor

Problem Description

While working with Project Reactor, we encountered an observability issue in production: **trace IDs were disappearing from logs at certain points**. This significantly impacted our ability to trace requests end-to-end and made debugging much harder.

Despite multiple investigations by different engineers, the root cause remained unresolved for quite some time. The issue required deep analysis and collaborative effort across the team before we were finally able to identify and fix it.

We observed this issue in two common scenarios: **New Thread Creation** and **Reactive Thread Switching**.

1. New Thread Creation

When executing logic asynchronously (e.g., via `CompletableFuture.runAsync`), the code runs on a **different thread**. Since MDC context is not propagated, the trace ID is lost.

```
private CompletableFuture<Void> runAsyncProcess(String id) {

    // Assume traceId exists in the current context
    Runnable task = () -> {
        log.info("Starting async processing for id: {}", id);
        callExternalService(id);
    };

    // TraceId is lost here because execution switches to a new thread
    return CompletableFuture.runAsync(task);
}
```

2. Reactive Thread Switching

In Project Reactor, operators like `publishOn` and `subscribeOn` explicitly move execution to different schedulers (threads). These transitions **break MDC continuity**, causing logs downstream in the reactive chain to miss the trace ID.

```
public void asyncMethod() {
  return repository.deleteById(id)
         .flatMap(result ->
             doSomeAction(id)
           .doOnSuccess(success -> log.info("Some action is successful for id: {}", id))
            // Execution switches to another thread from this point
           .publishOn(Schedulers.boundedElastic())
           .doOnSuccess(res ->
                // Runs on a different thread → traceId is already lost here
               runAsyncProcess(id)
          )
     );
}
```

What Is MDC?

For those new to the concept, MDC (Mapped Diagnostic Context) is a feature provided by logging frameworks, mainly by log4j and logback. It lets you attach key-value pairs to the current thread's context, such as `traceId`. Therefore, they automatically appear in every log line without you having to pass them around manually.

```java
// Set once at the start of a request
MDC.put("traceId", "abc-123");

// Now every log statement in this thread will include traceId=abc-123
log.info("Processing order"); // logs: Processing order traceId=abc-123
log.info("Fetching user");    // logs: Fetching user    traceId=abc-123

// Clean up when done
MDC.remove("traceId");
```

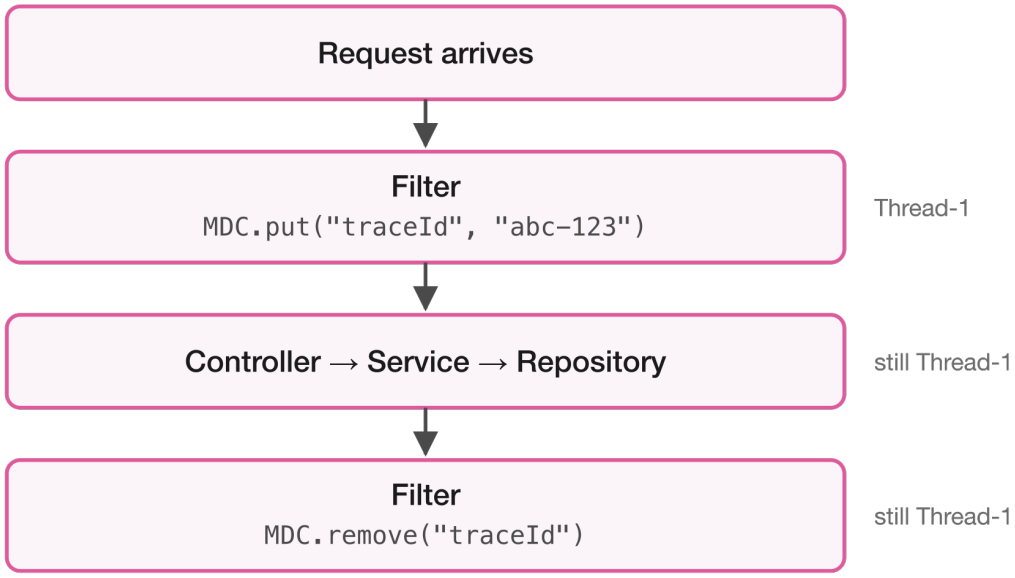
This works seamlessly in traditional Spring MVC, but can break silently in reactive applications.

Why It Works in Spring MVC?

MDC is backed by `ThreadLocal`. A `ThreadLocal` is a Java mechanism that gives each thread its own isolated copy of a variable. When you call `MDC.put("traceId", "abc-123")`, it stores that value in a map that is local to the *current thread*.

```
Thread-1: MDC["traceId"] = "abc-123"
Thread-2: MDC["traceId"] = "xyz-789"
Thread-3: MDC["traceId"] = null
```

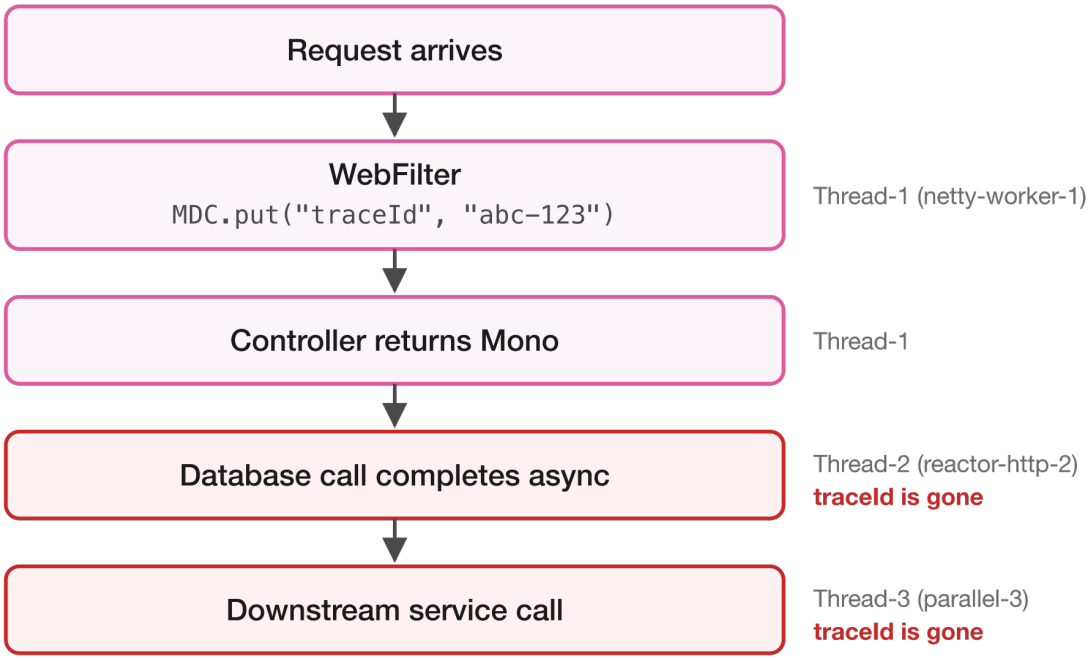
In Spring MVC, the threading model is simple: one request, one thread, for the entire lifecycle of that request. A servlet filter puts the trace ID into MDC at the start of the request, the thread handles all the business logic, and the filter cleans up MDC at the end.



In Spring MVC the request stays on one thread, so the trace ID placed in the MDC survives until the filter removes it.

Why It Breaks in Project Reactor?

Project Reactor does not follow the one-request-one-thread model. It is built on a small pool of non-blocking threads, and a single request can hop between threads as it moves through the reactive pipeline. Here's what that looks like in practice:



In Project Reactor the execution moves to other threads, and the trace ID is gone from the moment the database call completes asynchronously.

When execution resumes on Thread-2 or Thread-3, their `ThreadLocal` maps are empty. The MDC does not know what happened on Thread-1. The trace ID is lost, and your logs look like this:

```
10:21:03 [reactor-http-1] traceId=abc-123 - Received request
10:21:03 [reactor-http-2] traceId=      - Fetching user from DB ←      lost!
10:21:04 [parallel-3]     traceId=      - Calling inventory service ←   lost!
10:21:04 [parallel-3]     traceId=      - Returning response ←          lost!
```

Only the first log line has a trace ID. Everything after the first async boundary is untraceable. In a busy system, this makes debugging nearly impossible.

Solution

To address trace ID loss, we focused on a core problem: **context is not automatically preserved when execution switches threads**. Instead of relying solely on MDC, we introduced a solution that **explicitly propagates trace context across reactive and asynchronous boundaries**.

We adopted a strategy that ensures trace context survives thread switching. This approach allowed us to maintain a consistent trace ID, even when execution moved across multiple threads.

1. *Setup Micrometer Context Propagation*
2. *Add ThreadLocal in MDCInterceptor*
3. *Update Blocking/Async Tasks with Mono.fromRunnable*

Setup Micrometer Context Propagation

We added Micrometer's context propagation library to support automatic propagation of ThreadLocal values:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>context-propagation</artifactId>
</dependency>
```

Then enabled it during application startup:

```
Hooks.enableAutomaticContextPropagation();
```

Add ThreadLocal Support in MDC Interceptor

Since MDC is thread-bound, we introduced a **custom ThreadLocal** to store the trace ID. This class sets up **distributed tracing** by generating and managing a unique tracking ID (`traceId`) across different layers of the application. It ensures that every log message printed during a specific HTTP request contains the same `traceId`.

```
public class MdcInterceptor implements HandlerInterceptor {

    public static final String TRACE_ID_KEY = "traceId";
    public static final ThreadLocal<String> THREAD_LOCAL = ThreadLocal.withInitial(() ->
        "");

    // To propagate our custom ThreadLocal, we register a ThreadLocalContextAccessor
    static {
        ContextRegistry.getInstance()
                .registerThreadLocalAccessor(TRACE_ID_KEY, THREAD_LOCAL);
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler) {
        String traceId = // generate traceId here
        MDC.put(TRACE_ID_KEY,traceId);
        THREAD_LOCAL.set(traceId);
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler, @Nullable ModelAndView modelAndView) {
        MDC.remove(TRACE_ID_KEY);
        THREAD_LOCAL.remove();
    }

   @Override
   public void afterCompletion(HttpServletRequest request,  HttpServletResponse response,
        Object handler, @Nullable Exception ex) {
        // This block guarantees cleanup even if the controller crashes
        MDC.remove(TRACE_ID_KEY);
        THREAD_LOCAL.remove();
    }
}
```

Update Blocking/Async Tasks with Mono.fromRunnable

```
Runnable task = () -> {
    log.info("Starting async execution for id: {}", id);

    // TraceId is now available via ThreadLocal
    log.info("runAsyncProcess() traceId: {}", THREAD_LOCAL.get());

    callExternalService(id);
};

Mono.fromRunnable(task).subscribe();
```

Lessons Learned

During this journey, we explored multiple approaches before arriving at a reliable solution. These experiments revealed several important trade-offs and deeper insights about context propagation in reactive systems.

We evaluated several alternative approaches proposed in official documentation and various solutions offered in blog posts to preserve the trace ID. While these approaches looked promising in theory, they did not fully solve the problem in our environment.

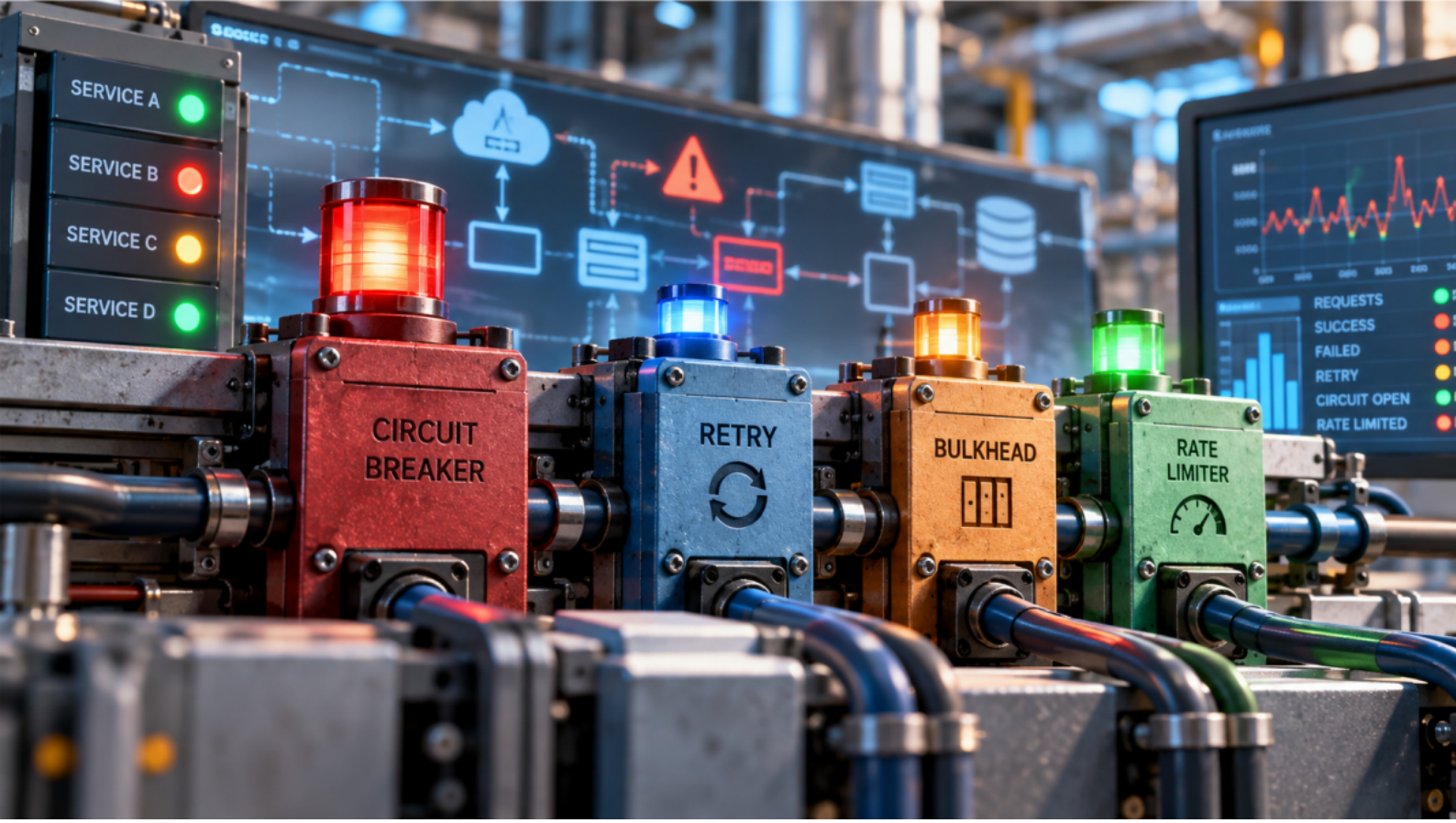
- Attempted to override the **default MDC adapter** used by our logging setup
- Investigated using **Twitter Finagle Futures**, which natively preserve context across async execution
- Experimented with a mix of Reactor and async techniques, such as `doOnEach`, `contextWrite`, wrapping tasks with custom MDC-aware logic.

One of the biggest challenges came from the logging infrastructure itself. There are multiple MDC adapter implementations such as **Logback MDC Adapter** (default in Spring Boot) and **Finagle MDC Adapter**.

In practice, Logback adapter worked correctly and preserved the trace ID. The Finagle adapter failed to propagate context, especially when new threads were created. We attempted to force the application to use the Logback adapter. However;

- It was not possible to exclude the Finagle adapter due to transitive dependencies
- It could not be reliably overridden programmatically due to SLF4J internals

› Back to the Table of Contents ‹



Resilience4j in Production: The Failure Patterns That Actually Matter

Resilience4j **# CircuitBreaker** **# FaultTolerance**

Author: Yasin Baytürk

Yasin Baytürk is a Senior Software Engineer at Stremler AG in Lindau, Germany, specializing in Java and the Spring Boot ecosystem. His work centers on system design, software architecture, and the practical application of design patterns in long-lived enterprise codebases — currently a production planning system (RTT) for the manufacturing industry. He works hands-on with SAP IDoc integration, batch processing, and Vaadin/TypeScript frontends, but most of his focus goes into architectural decisions and optimizing systems for speed and memory under real production load. He holds a Computer Engineering degree from Marmara University and has been working in Germany since 2017. He is especially drawn to Domain-Driven Design, microservices, and AI-assisted development on large legacy codebases. He writes to share lessons from real production systems, not toy examples.



LinkedIN

Circuit breakers, retries, bulkheads, rate limiters, and the mistakes that turn resilience code into outage amplification.

The first failure is rarely the outage. The outage usually starts when every caller reacts to that failure in the same wrong way: retry immediately, wait forever, fill the thread pool, and keep hammering a dependency that is already down.

Resilience4j is a small, modular Java library that gives you the building blocks for handling these failures: circuit breakers, retries, rate limiters, bulkheads, and time limiters. This article walks through the four most useful patterns, how they interact, and the production lessons the documentation does not always make obvious.

Why Resilience4j

Resilience4j is lightweight and modular. Each pattern lives in its own module (`resilience4j-circuitbreaker`, `resilience4j-retry`, `resilience4j-bulkhead`, `resilience4j-ratelimiter`, `resilience4j-timelimiter`) and you pick what you need. The API is functional: every pattern produces a decorator that wraps a `Supplier`, `Function`, or `CompletionStage`. There is also a Spring Boot starter that exposes everything as annotations and `application.yml` configuration.

For context, the older Netflix Hystrix library (in maintenance mode since 2018) used command-level isolation that commonly involved dedicated thread pools, which made it heavier than many simple circuit-breaker use cases needed. Resilience4j was designed from the start for Java 8+ and functional composition, without that overhead.

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
    <version>${resilience4j.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

At the time of writing, `${resilience4j.version}` resolves to 2.4.0. The AOP and Actuator starters are required at runtime: AOP for the annotations to be intercepted, Actuator for metrics and health indicators. Pinning a version property at the top of your POM saves you from chasing version numbers across modules.

The Four Patterns

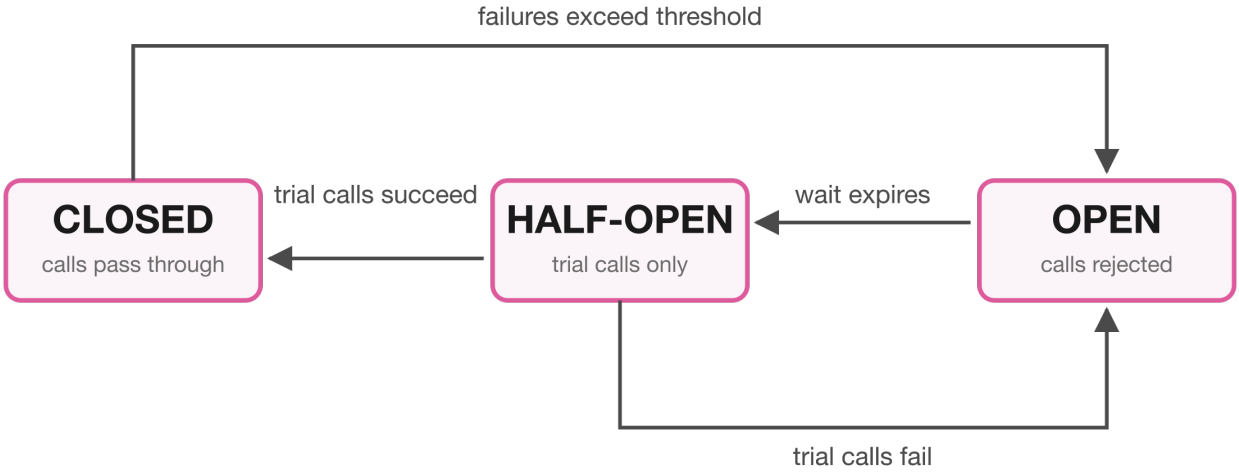
Each pattern answers a different failure question. Circuit breaker asks “should I even try this call right now?”. Retry asks “should I try again if it fails?”. Bulkhead asks “how many concurrent calls am I willing to make?”. Rate limiter asks “how many calls per second am I willing to make?”.

These are not interchangeable. Each addresses a different mode of failure. In production you will usually combine them, and the order in which they wrap your code matters. More on that later.

The examples below use synchronous HTTP calls for clarity. The same principles apply to `RestClient`, `WebClient`, and message-driven integrations, but the timeout and bulkhead choices differ.

Circuit Breaker

A circuit breaker monitors the calls flowing through it and opens (refusing all subsequent calls) when failures exceed a threshold. The breaker has three states:



The three states of a circuit breaker and the transitions between them: failures open it, the wait duration moves it to half-open, and the trial calls decide whether it closes again.

Closed is normal operation: calls pass through and outcomes (success, failure, slow) are recorded in a sliding window. **Open** rejects calls immediately without invoking the protected operation. After a configured wait duration, the breaker transitions to half-open. **Half-open** allows a limited number of trial calls through. If they succeed, the breaker closes. If they fail, it opens again.

The point is twofold: protect the downstream system from being hammered while it is already struggling, and protect the upstream system from queueing up requests that are doomed anyway.

Configuration That Matters

```yaml
resilience4j:
  circuitbreaker:
    instances:
      paymentApi:
        sliding-window-type: COUNT_BASED
        sliding-window-size: 20
        minimum-number-of-calls: 10
        failure-rate-threshold: 50
        slow-call-rate-threshold: 80
        slow-call-duration-threshold: 2s
        wait-duration-in-open-state: 30s
        permitted-number-of-calls-in-half-open-state: 3
        automatic-transition-from-open-to-half-open-enabled: true
        record-exceptions:
          - java.io.IOException
          - java.util.concurrent.TimeoutException
        ignore-exceptions:
          - com.example.payment.InvalidCardException
```

Most teams misconfigure circuit breakers in a small number of recurring ways.

`minimum-number-of-calls` is too low. If the breaker evaluates after only 5 calls and 3 of them fail, the failure rate is 60% and the breaker opens. But 5 calls is statistical noise. Set this high enough that the failure rate is meaningful for your traffic. For a service doing 1,000 requests per second, 50 is reasonable. For one that does 10 per minute, 5 might be all you can afford.

`record-exceptions` includes business exceptions. If `InvalidCardException` is recorded as a failure, then a flood of legitimately invalid cards will open the breaker and stop your valid cards from being processed. The breaker should only count infrastructure failures: timeouts, connection errors, 5xx responses. Move business exceptions into `ignore-exceptions`.

`slow-call-rate-threshold` is unset. A downstream service that is up but responding in 30 seconds will not register as a failure, because the calls eventually succeed. But your upstream service is still dying, because every request is now holding a thread for 30 seconds. The slow-call configuration treats responses above a duration threshold as failures even when they technically succeed. Always set it.

Sliding Window: Count-Based vs. Time-Based

Option	Window content	Best for	Drawback
`COUNT_BASED`	Last N calls	Steady traffic, predictable behaviour	A low-traffic service may take a long time to fill the window
`TIME_BASED`	All calls from the last N seconds	Variable or bursty traffic	More memory; sensitive to traffic shape (a quiet minute followed by 100 failures can open the breaker instantly)

Default to count-based unless you have measured a reason to switch.

Half-Open Is Where Most Incidents Happen

The transition out of `OPEN` is risky. When the breaker decides to test recovery, it lets `permitted-number-of-calls-in-half-open-state` calls through. If those calls happen to coincide with a slow recovery, they will hang or fail, and the breaker will close back to `OPEN`. Set this number low (3 to 5) so you do not amplify a partial outage. If your downstream is still struggling, you do not need 50 simultaneous probes.

Annotation Usage

```
@Service
public class PaymentClient {

    @CircuitBreaker(name = "paymentApi", fallbackMethod = "paymentFallback")
    public PaymentResult charge(PaymentRequest request) {
        return restClient.post()
                         .uri(url)
                         .body(request)
                         .retrieve()
                         .body(PaymentResult.class);
    }

    private PaymentResult paymentFallback(PaymentRequest request, Throwable t) {
        log.warn("Payment circuit open, falling back. cause={}", t.toString());
        return PaymentResult.deferred(request.id());
    }
}
```

Two things about fallbacks. The fallback method must have the same return type as the protected method, plus a `Throwable` parameter at the end. If the signature does not match, fallback resolution fails at runtime, usually at exactly the moment you need it most. Also, the fallback must not call the same protected method. It is a recursion trap that will exhaust your stack the first time the breaker opens.

Retry

Retry is the most over-applied pattern in this library. The instinct to "just try again" is correct about half the time, and the wrong half causes more outages than the failure it was trying to mitigate.

```yaml
resilience4j:
  retry:
    instances:
      paymentApi:
        max-attempts: 3
        wait-duration: 200ms
        enable-exponential-backoff: true
        exponential-backoff-multiplier: 2
        enable-randomized-wait: true
        randomized-wait-factor: 0.5
        retry-exceptions:
          - java.net.SocketTimeoutException
          - java.io.IOException
        ignore-exceptions:
          - com.example.payment.InvalidCardException
```

When Not to Retry

The first question to ask before adding `@Retry` to anything: is this operation idempotent?

Reading data is fine; retry. Sending an email is probably not idempotent (two emails are worse than zero). Charging a credit card without an idempotency key is definitely not idempotent. You will charge customers twice during partial network failures. Writing to a queue with at-least-once semantics is also a no, because the queue is your retry mechanism.

If the operation is not idempotent, do not add a retry. Make it idempotent first, by attaching an idempotency key, by making the write a CAS operation, or by checking before writing. Then retry.

Backoff Matters

A retry without backoff is a livelock generator. If five replicas of your service are all retrying a downstream service that is overloaded, immediate retries will keep the downstream pinned at 100% CPU and prevent it from ever recovering. Always enable exponential backoff. Always enable randomised jitter. The arithmetic is:

> `delay = base × multiplier^(attempt-1) × randomFactor`

With `wait-duration: 200ms`, `multiplier: 2`, `randomized-wait-factor: 0.5`, your three attempts wait roughly 100 to 300ms, then 200 to 600ms. The downstream gets time to breathe and your replicas desynchronise.

Retry Budgets

Cap your total retries, not just per-call retries. If 1% of your calls fail and you retry each three times, your downstream sees 3% extra load during normal operation. During an outage it sees 4× load, which is exactly when you can least afford it. Combine retry with a circuit breaker (the breaker will open and stop the retries from making the outage worse) and with a bulkhead (capping concurrent retries).

Bulkhead

The bulkhead pattern isolates concurrent resource usage so that a slow dependency cannot consume your entire thread pool. Resilience4j offers two implementations.

SemaphoreBulkhead

A counting semaphore. Simple, low overhead, runs on the calling thread.

```yaml
resilience4j:
  bulkhead:
    instances:
      paymentApi:
        max-concurrent-calls: 25
        max-wait-duration: 50ms
```

When 25 calls are already in flight, the 26th waits up to 50ms for a permit. If none becomes available, it fails fast with `BulkheadFullException`. The fail-fast behaviour is the point. You want the upstream caller to know immediately that you are saturated, rather than queueing it up to time out 30 seconds later.

ThreadPoolBulkhead

A separate executor with its own queue. Useful when you want to isolate slow calls from the calling thread's lifecycle.

```yaml
resilience4j:
  thread-pool-bulkhead:
    instances:
      paymentApi:
        max-thread-pool-size: 10
        core-thread-pool-size: 5
        queue-capacity: 20
        keep-alive-duration: 20ms
```

ThreadPoolBulkhead returns a `CompletionStage`, not the original return type. Your caller has to be prepared to handle this asynchronously.

Bulkheads and Virtual Threads

Virtual threads have been stable since Java 21, and JDK 24 improved their interaction with **`synchronized`** blocks through JEP 491. That changes the bulkhead conversation, but it does not remove the need for bulkheads.

The classic argument for ThreadPoolBulkhead, that you do not want a slow downstream blocking your servlet container's thread pool, is weaker when each request runs on its own virtual thread. The carrier threads are not blocked, because virtual threads on blocking I/O unmount from their carriers. With virtual threads in the picture, ThreadPoolBulkhead is close to retired for most synchronous use cases.

But the resource you are protecting was never really the threads. It was concurrent connections to the downstream, downstream-side capacity, and your own application's heap. Bulkheads still matter for those. With virtual threads, prefer SemaphoreBulkhead. It is cheaper, it does not require an executor, and its only job is to count concurrent calls.

One caveat. On JDK versions before JEP 491, blocking inside **`synchronized`** sections could pin virtual threads. On JDK 24 and later, that specific limitation is largely addressed, but native calls and a few remaining cases (class initialisation, certain VM frames) can still pin. Use `ReentrantLock` on any blocking path you care about regardless of JDK version. It is also clearer about intent.

Rate Limiter

Rate limiter restricts how many calls a window of time admits. The mental model is a token bucket: every `limit-refresh-period`, the bucket gets `limit-for-period` fresh tokens; each call consumes one. When the bucket is empty, new calls wait up to `timeout-duration` for tokens, then fail.

```
resilience4j:
  ratelimiter:
    instances:
      paymentApi:
        limit-for-period: 100
        limit-refresh-period: 1s
        timeout-duration: 50ms
```

Rate limiters are useful in three scenarios. The first is respecting a downstream's published quota. Your provider's contract says 100 requests per second; set the limiter at 90 to give yourself headroom for clock skew and burst tolerance. The second is protecting an internal service. A batch job suddenly producing 10,000 requests per second can be throttled at the client side before it reaches the network. The third is cost control. Some downstreams (LLM APIs, third-party data providers) bill per call. A rate limiter caps your worst-case spend.

Rate limiter is not a substitute for a circuit breaker. The breaker says "stop trying because this is broken". The limiter says "slow down because you are sending too much". You may want both.

The Order of Decoration

This is the detail teams miss most often. When you stack annotations, the order in which they wrap your code matters, and the library's default ordering may not be what you expect.

The Resilience4j documentation gives the aspect order as:

> `Retry ( CircuitBreaker ( RateLimiter ( TimeLimiter ( Bulkhead ( Function ) ) ) ) )`

A retry will retry after the circuit breaker has decided to allow the call, after the rate limiter has admitted it, and after the bulkhead has granted a permit. Each retry attempt re-enters all the inner decorators.

The implications are concrete. A retry rejected by the circuit breaker will not consume retry attempts on a `CallNotPermittedException` if you configure `ignore-exceptions` properly. Each retry attempt re-acquires a bulkhead permit, so if your bulkhead is small and your retries are aggressive, retries can starve other callers. A retry attempt is also rate-limited; if your rate limiter is at saturation, retries wait, which is usually what you want, but if `timeout-duration` is too short you will see `RequestNotPermitted` exceptions failing your retries.

If you need a different order, you can override the aspect order with properties (`resilience4j.retry.retryAspectOrder`, `resilience4j.circuitbreaker.circuitBreakerAspectOrder`, and so on), or build the decorator chain manually with the fluent builder:

```
Supplier<PaymentResult> decorated = Decorators
    .ofSupplier(() -> client.charge(request))
    .withBulkhead(bulkhead)
    .withRateLimiter(rateLimiter)
    .withCircuitBreaker(circuitBreaker)
    .withRetry(retry)
    .withFallback(List.of(CallNotPermittedException.class), ex -> deferred(request))
    .decorate();

PaymentResult result = decorated.get();
```

The fluent builder makes the layering explicit. Reading from outside to inside tells you exactly what happens in what order.

Observability with Micrometer

Resilience4j integrates natively with Micrometer. Once `micrometer-core` is on the classpath and Spring Boot Actuator is enabled, every breaker, retry, bulkhead, and rate limiter publishes metrics:

- `resilience4j_circuitbreaker_state` (gauge per state)
- `resilience4j_circuitbreaker_calls` (counter, tagged by `kind=successful|failed|slow|not_permitted`)
- `resilience4j_circuitbreaker_buffered_calls`
- `resilience4j_retry_calls` (tagged by outcome)
- `resilience4j_bulkhead_available_concurrent_calls`
- `resilience4j_ratelimiter_available_permissions`
- `resilience4j_ratelimiter_waiting_threads`

Three dashboards are worth building. Track circuit breaker state over time: a flapping breaker with frequent transitions between closed and open signals that thresholds are wrong, not that the downstream is broken. Track retry success rate: if `successful_with_retry` is high, retries are doing real work; if `successful_without_retry` dominates, your retries are mostly wasted CPU; if `failed_with_retry` is high, retries are not actually helping and you should investigate. Track bulkhead saturation: `available_concurrent_calls` against `max-concurrent-calls`. Persistent saturation means either the downstream is too slow or your bulkhead is sized too small for your traffic.

Production Anti-Patterns

A short list of mistakes that recur in real systems.

One global circuit breaker for all downstream calls. Different downstreams have different failure modes; a breaker for "external HTTP" treats payment, search, and recommendations as the same dependency. Have one breaker per logical dependency.

Retry inside retry. A library retries; the calling service also retries; the orchestrator retries. By the time a request is genuinely lost, the system has multiplied the load 27× and turned a transient blip into an outage.

Tight retries with no jitter. Five replicas all retrying after exactly 1 second produce a perfectly synchronised hammer on the downstream. Always randomise.

Fallbacks that do hidden work. A fallback that calls another remote service is just another circuit-breakable call. If you must call something else, decorate the fallback too. The simplest fallbacks return cached data or a sentinel.

No `slow-call` configuration. A downstream that responds slowly but successfully will exhaust your threads or virtual-thread carriers without ever opening the breaker.

Identical configuration in dev and prod. A breaker tuned for a 10-rps dev environment will be useless at 10,000 rps. Per-environment overrides are essential.

Recommended Defaults

A starting point, not a prescription. Tune these to your traffic and downstream characteristics.

Pattern	Safe default
Retry	Idempotent operations only; exponential backoff with jitter; 3 attempts max
Circuit breaker	One per logical downstream; `minimum-number-of-calls` scaled to your traffic; slow-call threshold enabled
Bulkhead	SemaphoreBulkhead for synchronous and virtual-thread paths; ThreadPoolBulkhead only for legacy async cases
Rate limiter	Slightly below downstream quota; longer `timeout-duration` than upstream's read timeout

Pattern	Safe default
Fallback	Cached data, sentinel response, or deferred processing; never another remote call without its own breaker

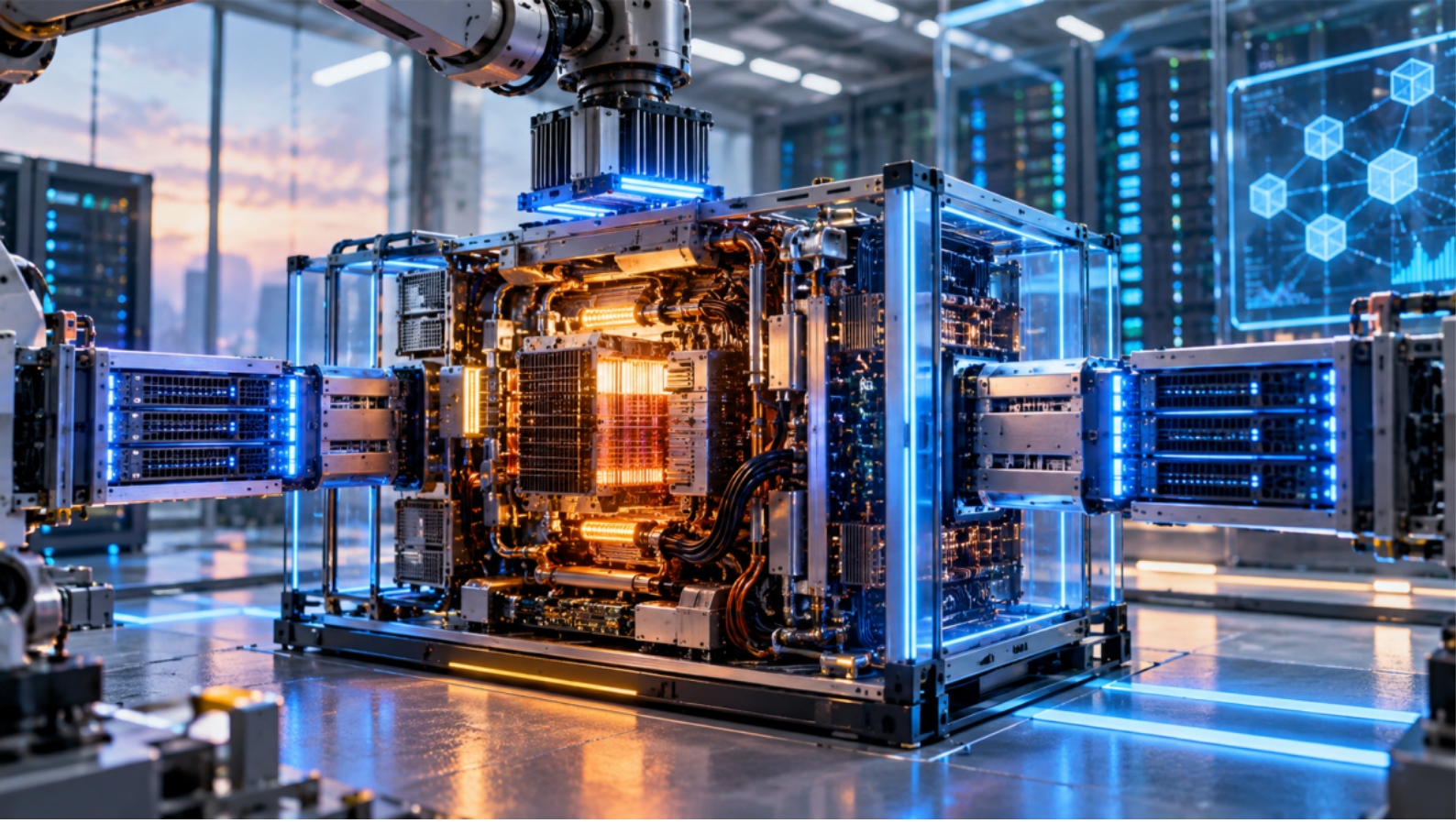
Conclusion

Resilience4j is small, modular, and well documented. The four patterns answer four distinct questions about how to behave when dependencies misbehave. Used together, they prevent the failure modes that turn a single sick service into a regional outage.

The patterns themselves are easy to find. The lessons that matter in production are the ones the documentation does not always emphasise: idempotency before retry, slow-call thresholds before timeouts, jitter before tight loops, the decoration order between annotations, and the observability that tells you whether any of it is actually working. Get those right and your service stops being part of the outage.

› Back to the Table of Contents ‹



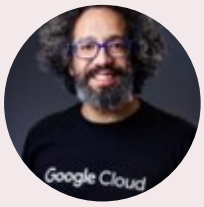


The Next Chapter of Kubernetes and Java: Java Apps and Dynamic Resource Allocation

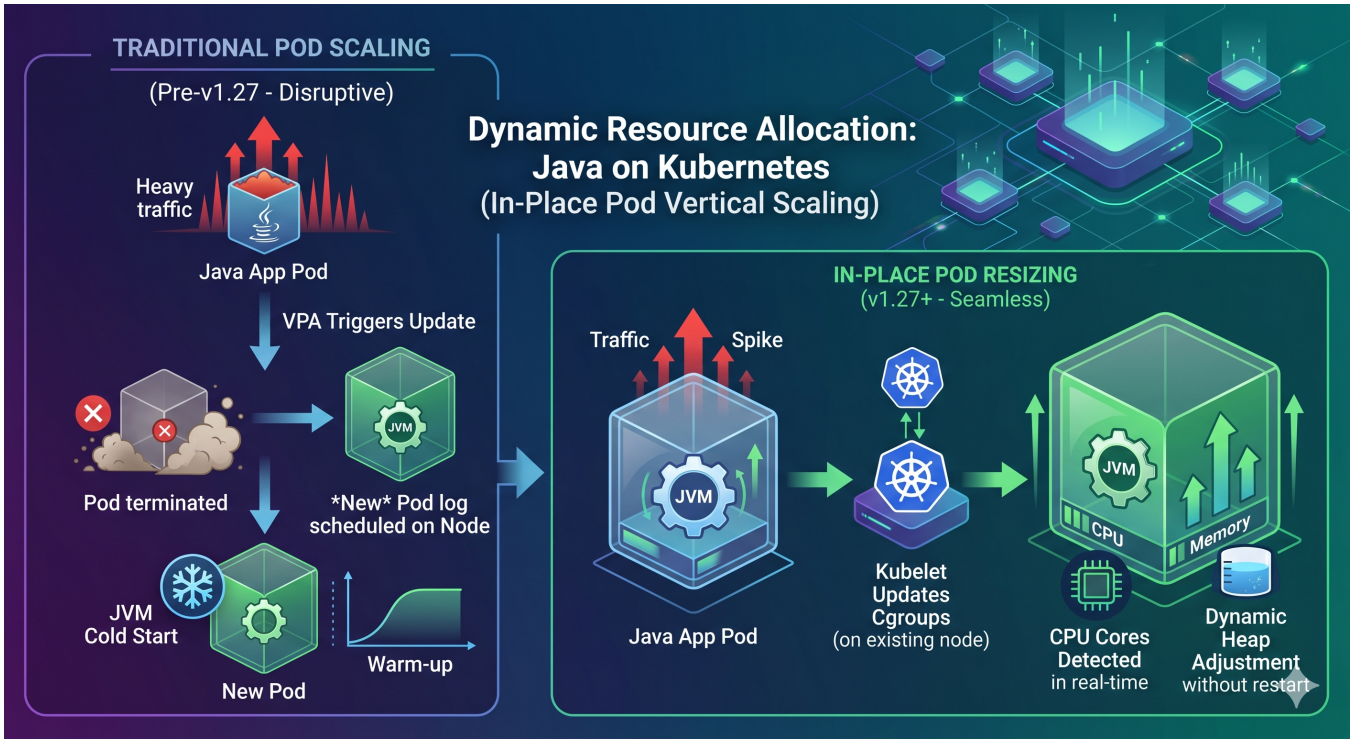
Kubernetes **# DRA** **# CloudNative** **# Container**

Author: Abdel Sghiouar

Abdel Sghiouar is a senior Cloud Developer Advocate @Google Cloud. A co-host of the Kubernetes Podcast by Google and a CNCF Ambassador. His focused areas are GKE/Kubernetes, Service Mesh, and Serverless. Abdel started his career in data centers and infrastructure in Morocco, where he is originally from, before moving to Google's largest EU data center in Belgium. Then in Sweden, he joined Google Cloud Professional Services and spent five years working with Google Cloud customers on architecting and designing large-scale distributed systems before turning to advocacy and community work.



LinkedIN



Before in-place resizing a pod is terminated and replaced, and the new JVM starts cold; from Kubernetes v1.27 the kubelet adjusts the cgroups of the running pod instead.

For years, Java and Kubernetes operated under an unspoken, slightly tense diplomatic truce. Kubernetes was built on the cloud-native philosophy of ephemerality: workloads are cattle, not pets, and if a container needs more power, you kill it and deploy a bigger one. Java, on the other hand, historically behaves like a well-insulated mansion. It takes time to boot, requires a warm-up period for the Just-In-Time (JIT) compiler to optimize bytecode, and builds up internal caches that make sudden evictions highly disruptive and expensive.

If your production Spring Boot or Quarkus application experienced a sudden traffic spike, the traditional Vertical Pod Autoscaler (VPA) or a manual YAML patch would save the day, but at a cost. Kubernetes would gracefully terminate the running Pod and schedule a new one with higher resource limits. During that rolling update, your application lost its warm caches, dropped established connections, and spent its first few minutes consuming significant CPU just to recompile classes.

But friction is over with the graduation of **In-Place Pod Vertical Scaling** (InPlacePodVerticalScaling) to General Availability (GA) in Kubernetes v1.35, and the recent introduction of Pod-level resources in v1.34, we have entered a new era. We can now resize the CPU and memory of running containers on the fly, without a restart.

But what does this mean for the JVM? Let's dive into the mechanics of in-place resizing and see how to architect Java applications to exploit this feature.

Under the Hood: How In-Place Resizing Works

Before this feature, container resources in a Pod specification were completely immutable. To change them, you had to recreate the Pod. Today, the spec.containers[*].resources field is mutable for both CPU and memory.

When an autoscaler (like VPA) or an SRE issues a patch API Call to adjust a running Pod's resources, the sequence avoids the scheduling queue entirely:

1. The Kubernetes API server checks the request against namespace ResourceQuotas and LimitRanges. If valid, it updates the Pod's spec.
2. The kubelet on the hosting node watches for this update. It evaluates whether the node has enough unallocated capacity to satisfy the new request.

3. If the node has room, the kubelet bypasses pod recreation and directly invokes the Container Runtime Interface (CRI) API's UpdateContainerResources() method. The container runtime updates the Linux kernel **cgroups** (Control Groups) for that specific container on the fly.
4. The Pod's status.containerStatuses[*].resources is updated to reflect the newly actuated limits. Additionally, the pod's Status field reflects these events as they occur.

If the node lacks resources, the Pod isn't killed; instead, Kubernetes marks the operation with a PodResizePending condition (status: Deferred or Infeasible), allowing automation tools (such as the Cluster Autoscaler) to either wait or trigger a traditional node scale-out.

The resizePolicy: Fine-Grained Control

Crucial to this workflow is a field introduced in the container specification: resizePolicy. This block allows developers to dictate exactly how a container should react when a resource changes. It supports two strategies:

- NotRequired: Change the cgroup limits dynamically while the container runs.
- RestartContainer: Trigger a local container restart within the existing Pod envelope to safely apply the resource change.

Consider the deployment spec for an example application below.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example
spec:
  replicas:
    template:
    spec:
      containers:
      - name: jvm-app
        image: openjdk:21-jdk-slim
        resources:
          limits:
            cpu: "2"
            memory: "2Gi"
          requests:
            cpu: "1"
            memory: "1Gi"
        resizePolicy:
        - resourceName: "cpu"
          restartPolicy: "NotRequired"
        - resourceName: "memory"
          restartPolicy: "RestartContainer"
```

For Java developers, configuring these two resources requires entirely different approaches because of how the JVM interacts with the operating system.

CPU Resizing

Scaling CPU in-place with restartPolicy: "NotRequired" works for Java applications out of the box.

Modern OpenJDK releases (Java 17, 21, and the recently released Java 25) feature container awareness via -XX:+UseContainerSupport (enabled by default). When the kubelet alters the underlying cgroup's cpu.max quota, the JVM notices this change dynamically.

If you invoke:

```java
int availableProcessors = Runtime.getRuntime().availableProcessors();
```

The JVM will return the updated count reflecting the new cgroup constraints *without* restarting the process.

Why This Matters for Java Runtimes:

- **ForkJoinPool & Parallel Streams:** Common frameworks like Spring Boot, Quarkus, and Micronaut build their underlying thread pools based on availableProcessors(). While standard Java thread pools (like a fixed-size ThreadPoolExecutor) might not dynamically resize their worker arrays automatically without custom code, reactive frameworks and the fork-join pools used by parallel streams query this runtime value to adapt their parallelism thresholds during execution.
- **JIT Compilation:** When traffic spikes, the JIT compiler (C1/C2) works overtime to compile frequently executed code paths into native machine code. Dynamically giving a container 2 extra cores during a spike provides the JIT compiler with immediate execution bandwidth, eliminating CPU throttling without losing the pre-compiled native optimizations already residing in the CodeCache.

Memory Resizing

While CPU scaling is seamless, **Memory Resizing** requires careful consideration.

The Linux kernel handles memory via hard cgroup limits (memory.max). If a container exceeds this limit, the kernel's Out-Of-Memory (OOM) Killer instantly terminates the process. Inside the container, the JVM allocates its memory primarily through the Max Heap Size (-Xmx).

Traditionally, the JVM calculates -Xmx at startup based on a percentage of the container's memory (e.g., -XX:InitialRAMPercentage=50.0 and -XX:MaxRAMPercentage=75.0).

Here is the catch: **The standard JVM does not dynamically recalculate its Max Heap Size (-Xmx) if the underlying cgroup memory limit changes after boot. This leaves us with two options**

Option A: RestartContainer

If your application relies heavily on predictable heap sizing or uses a fixed -Xmx configuration, setting the memory restartPolicy to RestartContainer is the safest option. When the VPA detects memory exhaustion, the kubelet restarts *only the container* inside the existing Pod. This skips the Kubernetes scheduling phase entirely, reducing downtime compared to a full Pod eviction. But still incurs a JVM cold-boot penalty.

Option B: Live Scaling (NotRequired + Dynamic Runtimes)

If you want to scale up memory with absolutely zero downtime, you must configure the resource as NotRequired and handle heap adjustments gracefully:

1. **Leveraging Dynamic GC Ergonomics:** Advanced garbage collectors like ZGC (Z Garbage Collector) and G1 are highly elastic; they naturally return unused memory to the operating system via explicit uncommitting mechanisms (-XX:ZUncommit or -XX:G1PeriodicGCInterval.
2. **Explicit Headroom Configuration:** If you expect a pod to scale from 2Gi to 4Gi in-place, you cannot rely entirely on MaxRAMPercentage calculated against the initial 2Gi. Instead, decouple the container's cgroup limits from the JVM's Max Heap size, or use custom JMX beans and modern operator patterns to programmatically adjust JVM parameters or trigger explicit garbage collections when a Kubernetes downscale is imminent.

Warning: Be incredibly careful with **down-scaling memory** in-place. If the VPA shrinks your container's cgroup memory limit below the JVM's currently *committed* memory pool, the Linux kernel will instantly OOM-kill your Java process. Always ensure your garbage collector aggressively uncommits heap memory to the OS before configuring automated downward memory scaling.

VPA CPU Startup Boost

While in-place resizing resolves the friction of sudden runtime traffic spikes, the open-source community has delivered an ecosystem evolution that targets the JVM's most notorious cloud-native pain point: **startup and warm-up throttling.**

Under Autoscaling Enhancement Proposal AEP-7862, the Vertical Pod Autoscaler introduces the **CPUStartupBoost** feature gate (Currently in Alpha).

Removing the "Warm-Up Tax"

When a Java Pod boots, it executes static initializers, loads thousands of classes, and activates the JIT compiler to compile bytecode into native instructions. This process is intensely CPU-bound. Historically, to prevent startup times from stretching into minutes, platform teams over-provisioned a deployment's initial CPU resources, leaving those cores idle once the app reached a steady state. This resulted in low resource density and inflated cloud bills.

CPUStartupBoost solves this by applying a temporary, high-capacity CPU allocation (the "Boost" phase) at Pod initialization. Once the application stabilizes, VPA leverages the underlying InPlacePodVerticalScaling machinery to scale the CPU back down to runtime levels, completely in place, with no container restarts, preserving the hot JVM CodeCache.

Configuring the VPA CPU Startup Boost

To harness this feature, your cluster must run Kubernetes 1.33+ and VPA 1.7.0+ with the CPUStartupBoost feature gate enabled. Below is a VPA manifest tailored for an elastic Java application:

```yaml
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: example-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: example
  updatePolicy:
    updateMode: "Auto"
  startupBoost:
    cpu:
      type: "Factor"
      factor: 3
      durationSeconds: 10
```

In this configuration, the example deployment requests 1 core of CPU, the VPA Admission Controller will instantly inject a 3 cores at boot time. After the application passes its readiness probes and passes the set durationSeconds configuration, the VPA Updater triggers an in-place patch to gracefully throttle the container back down to its runtime needs.

Operating with the New Patterns

To successfully implement dynamic resource allocation in your enterprise Java platforms, update your deployment manifests and infrastructure strategies to target these three modern patterns:

1. Couple VPA with InPlaceOrRecreate

When configuring your Vertical Pod Autoscaler, leverage the updateMode: InPlaceOrRecreate. This instructs the autoscaler to always attempt a non-disruptive, in-place cgroup resize first. It will only fall back to a disruptive Pod eviction and recreation if the underlying node is resource-choked. Note this is GA in VPA 1.7.0 and requires Kubernetes 1.33+

2. Pay Attention to Thread Pool Inelasticity

Be aware that while Runtime.getRuntime().availableProcessors() updates dynamically; old-school, third-party library thread pools initialized at startup might capture that value in a static final variable. Prefer modern reactive frameworks, or use Java 21+ **Virtual Threads (Project Loom)**, which mount on a carrier pool that natively integrates with the JVM's ForkJoinPool infrastructure.

3. Embrace Pod-Level Budgets

With Kubernetes v1.34 introducing beta support for Pod-level resource updates, you can now define a collective resource pool for your entire Pod rather than micromanaging individual sidecars (such as Logstash, FluentBit, or Istio proxy). If your main Java application container requires a burst of CPU, it can instantly draw from the aggregate Pod allocation without requiring complex per-container YAML configurations.

Conclusion

The evolution of Kubernetes resource management represents a good opportunity for enterprise Java. By transforming CPU and memory limits from static boundaries into fluid, mutable envelopes, Kubernetes has adapted to the architectural realities of the JVM.

By designing your deployments with the correct resizePolicy, writing runtime-aware code, and utilizing modern, elastic garbage collectors like ZGC, you can achieve a truly self-healing, auto-optimizing Java ecosystem that scales instantly to demand—saving cloud spend and infrastructure overhead without sacrificing a single millisecond of application uptime.

› Back to the Table of Contents ‹





The Server-Driven UI Renaissance

ServerDrivenUI **# WebArchitecture** **# Frontend**

Author: Hyyan Abo Fakher

Hyyan Abo Fakher is a lead software architect working on frameworks, UI architecture, and developer tooling across Java and JavaScript. He started out as a full-stack developer and gradually moved toward the platform layer — frameworks, internal tools, and developer experience — where most of the interesting edge cases live. He writes about what he learns along the way: Java, JavaScript, CSS, UI design, and the day-to-day realities of building stable, predictable application



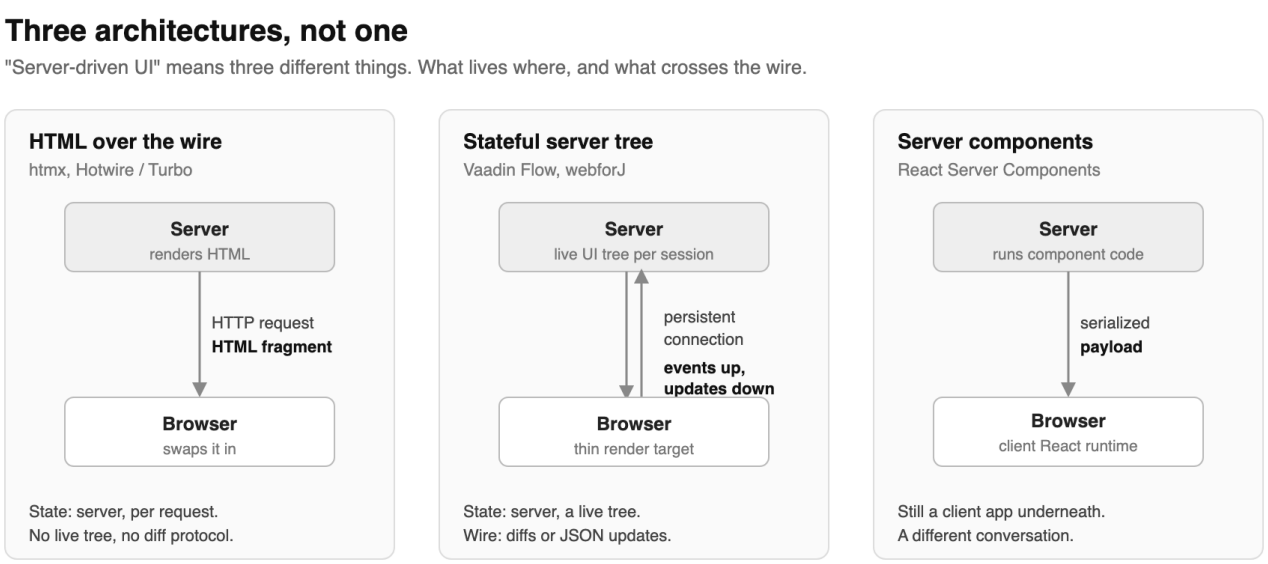
LinkedIN

The single-page app was never the right default. We shipped it everywhere, paid a large complexity tax for it, and rebuilt a chunk of HTTP in JavaScript to make it work. The shift back toward server-driven UI is not nostalgia. It is a correction for the class of apps that never needed a client-side framework in the first place. That is the argument of this piece, and I can make it from the inside, because I build one of these server-driven frameworks for a living and I know exactly where the model bleeds.

So before any of the upside: the costs are real. A server-driven UI puts a round trip between the user's click and the visible result. It keeps per-user state in server memory, which you now have to size and scale. It has a "what happens when the connection drops" story that you cannot wave away. And the moment you need genuinely local interactivity, you are back to shipping JavaScript. Each of these gets its due below. If the model still fits your app after that, it fits for the right reasons.

There is also a vocabulary problem. "Server-driven UI" gets used for at least three different architectures that behave nothing alike. Most of the confusion in this debate is people arguing past each other because they mean different things by the same phrase. Let me fix that first.

What "Server-Driven" Actually Means: Three Architectures, Not One



The three server-driven UI architectures: HTML over the wire, the stateful server tree, and server components, showing where state lives and what crosses the wire.

HTML over the Wire

The cleanest of the three. htmx and Hotwire/Turbo live here. The server renders HTML. The browser asks for more HTML when something happens, and swaps it into the page. There is no long-lived component tree on the server and no diffing protocol. There is just HTTP and markup.

This is worth seeing concretely, because the wire is the whole point. An htmx click is an ordinary HTTP POST. A counter's +1 button carries `hx-post="/increment"`, and the response body is just HTML:

```
<div id="counter">
  <span id="value">1</span>
  <button
     hx-post="/increment"
     hx-target="#counter"
     hx-swap="outerHTML">
    +1
  </button>
</div>
```

That is the whole response: HTML, with the new value already rendered into it. Not JSON, not a patch instruction, not a virtual-DOM diff. htmx takes those bytes and replaces the target element with them. The htmx docs say the quiet part out loud: "on the server side you typically respond with HTML, not JSON." The state, the count lives in the server process. The browser holds none of it.

The Stateful Server Tree

This one is a different beast. The server keeps a live object graph that represents your UI for that one user's session. The browser holds a persistent connection back to it. When you click, the event travels to your server-side handler, your code mutates the tree, and the framework works out what changed and tells the browser to update. You write the whole UI in your backend language. The browser is a thin render target.

In the Java world, two frameworks build directly on this model: Vaadin Flow and webforJ, the framework I work on. Both hold a tree of Java objects on the server, sync it to the browser as JSON, and let you write the whole UI in Java with no templates and no hand-written client. Vaadin's documentation puts the shared mechanism plainly: on an interaction Flow "creates a new JSON object with rendering instructions" returned over an HTTP POST, acting as a "remote control for HTML elements in the browser."

webforJ takes the same shape. A browser event arrives at the server as a typed Java event and runs your handler, an internal bridge turns the raw client event into that Java event, and the wire, like Vaadin's, is JSON: an auto-generated message schema of several hundred message types, each a small JSON object the engine sends to the browser.

This is not a Java-only idea, and that is the point. Other ecosystems have their own take on the same stateful-server-tree model: Laravel's Livewire in PHP and Phoenix LiveView in Elixir, where component state lives on the server and the browser receives diffs. Different languages, one architecture. If anything, Java has the deepest history with it: JavaServer Faces and Apache Wicket were holding component state on the server for the better part of two decades, and the idea carried onto the modern web through frameworks like Vaadin Flow and webforJ.

Server Components

The one people most often confuse with the other two, as in React Server Components. RSC runs component code on the server to produce a serialized payload that the React client renders, and it is still fundamentally a client-side React app underneath, with hydration and a client runtime. It is not a stateful server session holding your UI between clicks. I mention it only to keep it distinct from the stateful server trees. It is a genuinely different architecture, and one this article does not try to cover.

So: HTML-over-the-wire, stateful server tree, and server components. Three answers to "where does the UI live," not one. The rest of this article is about the middle one, the stateful server tree, because that is where the interesting trade-offs live, and where the Java ecosystem has placed most of its bets.

Why the Pendulum Swung Back to Server-Driven UI

The SPA became the default answer to every web UI, including the ones it was bad at. A CRUD admin panel for forty internal users does not need a single-page app, with the build pipeline and the client-side state that come bundled with one. It got one anyway. That was the path of least resistance for a decade.

The tax showed up as duplication. Your data model lives on the server. To put it on screen you describe it a second time for the client: an API contract, the request and response types, a copy of the server's state that has to stay in lockstep with the original. Validation gets written twice. The loading and error and empty states, which a server-rendered page used to give you for nothing, you now wire up by hand, because the client owns the fetch. None of that is wasted effort in a Figma or a Google Maps. In a form-over-data business app, almost all of it is.

For a Java shop the duplication carried a second cost: a second language and a second team. Wanting a modern web UI used to mean standing up a separate JavaScript frontend, a build pipeline to match it, and a REST layer whose whole job was to shuttle your domain model across the gap. One application, run as two codebases by two teams who barely spoke the same language. The stateful-server-tree model collapses that back into the Java you already have. The UI is Java and the services are already Java. Nothing sits between them, because there is nothing left to glue.

The server-driven models delete that duplication by keeping one source of truth. The model and the UI logic both live on the server. There is no API to design as glue between two copies of your state, because there is only one copy. That is the real pull, and it has nothing to do with round-trip speed. A whole category of apps had been paying for a client-side architecture it never used.

The Four Hard Problems Every Stateful Server Tree Must Answer

Picking server-driven UI does not make problems disappear. It moves them. Four of them, specifically, and how a framework answers each is the only comparison that matters.

Latency per Interaction

When the UI tree lives on the server, the authoritative result of a click also lives on the server. So every meaningful interaction is a round trip. On a fast LAN this is invisible. On a phone on a train it is a real, felt delay before the button looks pressed. The mitigations are known: optimistic UI that updates locally before the server confirms, client-side bindings for the trivial stuff, debouncing input so you are not shipping a request per keystroke. But the baseline cost is structural. You are trading the SPA's "instant but duplicated" for "single source of truth but a network hop." Know your users' network before you make that trade.

The mitigation that matters most is not paying for the trivial interactions at all: toggling a panel open, validating a field's format, filtering a list the browser already holds. A framework that keeps those on the client and reserves the round trip for what genuinely needs the server turns latency from a constant tax into an occasional one. That is the line between a server-driven UI that feels native and one that feels like a remote desktop. Test it on a real mobile connection, not just your office LAN, before you decide.

Session State Cost and Scaling

A stateful server tree means one live object graph per user per open tab, sitting in server memory between clicks. That is the bill the SPA does not pay, because the SPA's per-user state sits in the user's browser for free. Now it is your RAM and your scaling problem. It pushes you toward sticky sessions, because the user's UI tree is pinned to the node that holds it, which complicates load balancing and rolling deploys. What one session costs depends on what is in that tree. A login form and a five-thousand-row data grid are not in the same league, so measure your own worst case before you size anything. The trade itself is plain: cheap client memory becomes your server's, and at high concurrency that is the first place it hurts.

Connection Loss and Reconnection

This is the problem the model cannot hide, because the persistent connection is the whole architecture. Networks drop. Laptops sleep. In a Java stateful framework your UI tree is tied to the user's server session: if that session is still alive when the connection returns, you resume where you were. If it expired, or a deploy restarted the node that held it, the tree is gone and the user faces a reload. That is the shape of it, and it is the same in every framework of this kind.

So the questions to ask are concrete. How long does an idle session live before its tree is evicted? When the connection drops and returns inside that window, does the framework re-attach to the existing tree or rebuild from scratch? What happens to a half-filled form? Is it preserved, or does the user retype it? And under a rolling deploy, when the node holding a session goes away, does the user get a clean "session expired, reload" or a broken screen? None of these are edge cases. On a flaky mobile network they are routine. A framework worth adopting documents its answers to all four. Read them before you commit, because this is the one trade-off the model cannot make vanish.

The Escape Hatch

Sometimes you need code that runs in the browser, full stop. A drag interaction that must track the pointer at 60fps. A canvas. A third-party JS widget. A keyboard shortcut that must feel instant. A round trip will not do, and a server-driven framework that has no answer here is a trap. So forget "does it avoid JavaScript." The question that matters is how gracefully it lets you drop into JavaScript when you have to. The frameworks vary a lot here, and a thin escape hatch is a real weakness, not a purity virtue.

What a Java Team Actually Chooses Between

If the model fits your app, you still have three real options on a Java backend, and they are not close substitutes.

On a Java backend	You write	UI state lives	Separate JS frontend?	Best fit
HTML over the wire (htmx + Spring MVC / Thymeleaf)	controllers and HTML templates	server, per request (no live UI tree)	no	making server-rendered pages dynamic, no build step
Stateful Java component framework (Vaadin Flow, webforJ)	the UI in Java, no templates or REST glue	server, a live tree per session	no	data-dense internal tools and forms-over-data
SPA + REST (React or Angular plus a Spring REST API)	a JavaScript app and a Java API	the browser	yes	highly interactive, offline, or public high-traffic UIs

The middle row is the one most Java teams overlooked. You already have the services and the domain model in Java. A stateful component framework puts a UI on them without a second language or a REST layer in between, which is the duplication the first half of this article was about. Vaadin Flow and webforJ are the frameworks built on this model for today's web, and both run as ordinary Spring Boot applications, so you keep your existing services, security, and configuration, and add a UI instead of standing up a second tier. JavaServer Faces and Apache Wicket are the previous generation of the same idea, still running plenty of production systems.

The HTML-over-the-wire row is the lightest touch. If you mostly have server-rendered pages and want to make them dynamic, htmx adds that with a few attributes and a Spring controller that returns a fragment, and nothing else in your stack changes. The SPA row earns its complexity only when the interactivity, offline support, or public scale genuinely demands it. Most internal Java apps never did.

When Not to Reach for This

There are apps server-driven UI is wrong for, and naming them matters as much as naming the wins.

- Do not build an offline-first app on a stateful server tree. The architecture assumes a live connection to where the UI lives. No connection, no UI. If your users work on planes or in the field, this is the wrong model, and no amount of reconnection cleverness changes that.
- An ultra-high-interactivity canvas is the wrong fit too: a drawing tool, a game, a timeline editor where the interaction is the product. The per-interaction round trip is fatal to that feel, and the escape hatch becomes the whole app, at which point you are writing a client-side app with extra steps.
- And keep it away from public, high-traffic, mostly-anonymous pages. Per-session server memory is fine for few hundreds users and ugly for forty thousand drive-by visitors. The SPA's free client-side state was the right trade there. Use it.

Where Server-Driven UI Genuinely Wins

It wins exactly where the SPA was overkill. Internal tools and line-of-business apps, where the user count is bounded and a round trip on click is imperceptible on the corporate network. Data- dense applications where the logic is heavy, the model is the product, and duplicating it across a client/server boundary was pure cost. Teams that are deep in Java and shallow in the JavaScript build ecosystem, who get a real web UI without standing up a separate frontend stack. And security-sensitive UIs, where keeping business logic on the server, never shipped to the browser, is a feature, not an inconvenience.

This was never about "server winning." SPA-by-default was simply the wrong answer for a large class of app, and the industry is catching up to that. Pick the model that matches your app, not the one that matches the last conference talk. For a Figma, ship the client-side app and pay its complexity gladly. For the internal dashboard people use every day, a server-driven UI deletes a pile of accidental complexity you were never getting paid to maintain. The renaissance is just that recognition, arriving a decade late.

› **Back to the Table of Contents** ‹





Joyful Web Development with Hypermedia-Driven Applications and HTMX

HTMX **# Hypermedia** **# WebDev**

Author: Frederik Hahne

Frederik has been developing mainly web applications since 2007, but he has also experience in building desktop applications. He has created JSP-based applications, Wicket web applications and SPAs with angular.js and vue.js. Since 2022, Frederik has been contributing his knowledge as a Software Architect/Senior Software Engineer for adesso SE at the beautiful Paderborn site. in various, mostly Java-based, projects. In his scarce free time, he organises the Java User Group Paderborn to promote the exchange of knowledge in and around Paderborn. He has been part of the JHipster Core Team since 2015 and is responsible for especially the Gradle support and lately the integration with Spring integration with Spring Native and GraalVM.



LinkedIN

Single-page applications, state management, hydration, client-side routing—modern web applications are packed with tools, frameworks, and complexity. Yet the original idea behind the web was actually relatively simple. In this article, we explore the origins of hypermedia and how hypermedia-driven applications can help us rediscover the joy of web development today—without the complexity of typical single-page applications.

The Problem with Complexity

These days, a typical web application usually consists of a server-side component that provides a JSON API.
These APIs are then consumed by a single-page application as the front end.
While the options on the server side are still relatively limited, the possibilities on the front end are virtually endless and continue to grow every day. (Link))

This often leads to

- high(er) complexity, two different stacks need to be mastered
- High entry barriers: It's not easy to become "productive" quickly
- High maintenance burden: security updates, dependency updates, build pipelines, API churn
- Additional sources of errors: API contracts, client-side state, routing

Often, the decision to use an SPA is simply accepted as a given because "that's just what everyone does." Thoughtworks has dedicated a separate category to this phenomenon in its Tech Radar (Link)): "SPA by default."

Thoughtworks argues that the decision to use an SPA is often not questioned, and that the software requirements frequently do not justify the complexity involved—a server-side rendered website is often sufficient.

Users' expectations have naturally risen since the advent of websites, and a minimum level of convenience and interactivity is now expected. Static HTML pages alone are often not enough anymore. Applications like Miro or Figma would be unthinkable without single-page applications and a significant amount of client-side logic.



Two extremes: only static HTML on one end, Miro and Figma on the other. The interesting question is what sits between them.

What Is Hypermedia?

To understand what a "hypermedia-driven application" is, we need to take a brief look back at history.
The terms "hypermedia" and "hypertext" were coined by Ted Nelson in the 1960s, although Vannevar Bush first mentioned them as early as 1945 in his article "As We May Think."

Hypermedia describes a (at the time) new type of media (usually text) characterized by a nonlinear model. In contrast, traditional media such as films (or books) cannot, for example, switch to a documentary about the actresses in the middle of the film, even if the viewer wants them to.

Hypermedia and hypertext were intended to make the world's information more accessible and to clarify the connections between topics.
Nelson's vision of hypermedia was not limited to text and links between texts, but also included audio, video, images, and other sources of information.
The ideas of Nelson and others anticipated many concepts that we today take for granted.

Based on Nelson's ideas about hypermedia and hypertext, Tim Berners-Lee developed the World Wide Web. (Link)
The Internet of the early 1990s was based almost entirely on text and links, allowing users to read the information provided in any order they chose.

Despite these simple foundations, the Web continued to evolve rapidly. The best example of hypermedia is probably Wikipedia.

But what exactly defines a hypermedia-driven application?

A hypermedia-driven application uses (exclusively) hypermedia exchanges to communicate with the server

HTML as Hypermedia

Although HTML has been very successful in forming the foundation of the modern Internet, it has hardly evolved as a hypertext/hypermedia format since the introduction of forms for manipulating data.

The options are indeed very limited: Only <a> elements (links) and <form>
elements are allowed to generate HTTP requests. "click" and "submit" are the only events that can trigger a server request. Only GET or POST requests are allowed, and the entire page is always replaced with the server's response.

This is where HTMX comes into play, extending HTML as hypertext by removing the limitations mentioned above.

HTMX – HTML Extended

HTMX is a JavaScript library that allows you to use features such as AJAX, WebSockets, and Server-Sent Events in a declaritive via custom attributes directly in HTML.
Processing the server response no longer requires replacing the entire page.

HTMX is based on a small number of attributes marked with the prefix "hx-".

- `hx-get|post|put|delete` : Which HTTP verb should be used, and which URL should be accessed?
- `hx-target` : Which element on the page should be modified or replaced with the server's response?
- `hx-trigger` : Which user interaction triggers the request?
- `hx-swap` : How should the old element be replaced with the answer?

Using these basic attributes, it is already possible to build some pretty dynamic applications. The learning curve is relatively gentle here, since there are so few key attributes.

The following example triggers a POST request to /clicked when the button is clicked. The result (partial HTML) replaces the entire element with the ID parent-div.
The so-called "swap" strategy, outerHTML, describes the behavior whereby, after the replacement, the element with the ID parent-div is no longer part of the DOM (the opposite would be innerHTML).

```
<button
    hx-post="/clicked"
    hx-trigger="click"
    hx-target="#parent-div"
    hx-swap="outerHTML">
    Click Me!
</button>
```

JTE – A Modern Template Engine

As Java developers, we're used to doing things with type safety.
There are many great template engines in Java for generating HTML.
However, most of them aren't type-safe, which is a common source of errors. In particular, changing or renaming classes requires a lot of work in the templates.

JTE is a recent template engine that is very fast, simple, and lightweight, and offers a high degree of type safety.

```
@import org.example.Page
@param Page page
<head>
    <title>${page.getTitle()}</title>
</head>
<body>
    <h1>${page.getTitle()}</h1>
</body>
```

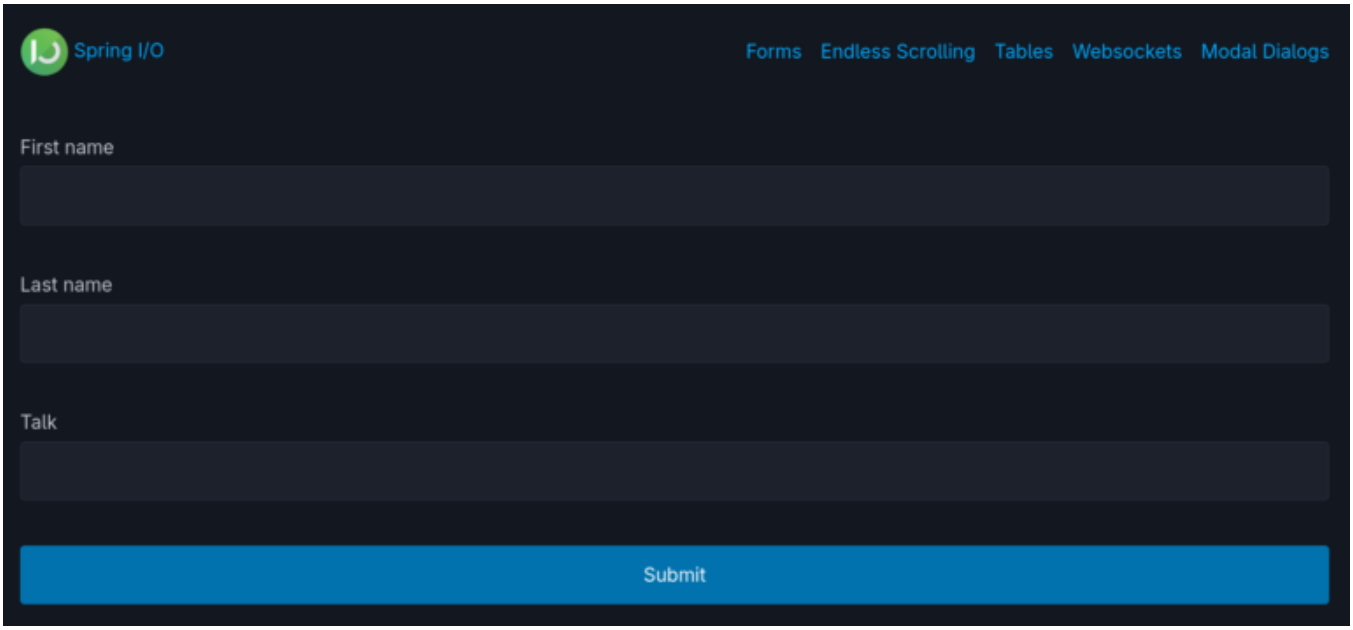
However, it is not necessary to use JTE to build a hypermedia application. Virtually any template engine can be used. This approach is also known as the HOWL stack (“Hypermedia on whatever you like”).

In the following, we’ll implement some typical patterns using HTMX and hypermedia instead of JavaScript/JSON.

HTMX Patterns

In the following, we will build some typical patterns and functions using HTMX as a hypermedia application.

Form Validation



A typical web form.

With HTMX, inline/real-time form validation can be implemented exclusively using hypermedia. A major advantage is that the validation rules are implemented solely on the server side, eliminating the risk of discrepancies between client- and server-side rules.

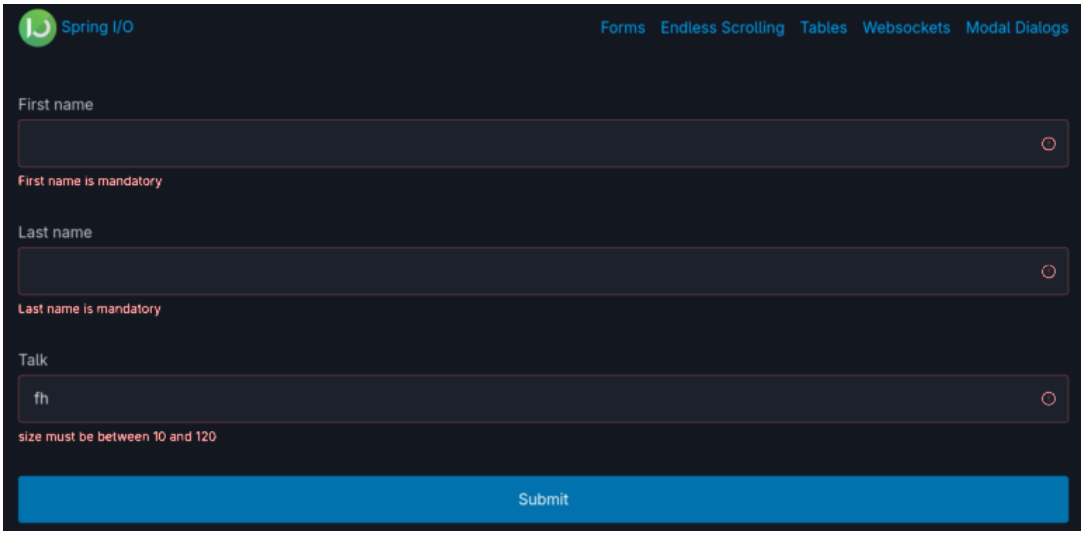
In this example, the hx-include attribute is important: Since the server validates the entire form, the complete form must always be sent. Without the additional hx-include, only the current field would be sent, and validation would fail.
The `hx-swap` strategy `morph` ensures that the form is not simply replaced, but rather that the current state is transferred to the new one using the `morphdom` library. This preserves, for example, the current cursor position.

The selection of the appropriate handler is controlled by special headers.
The @HxRequest annotation is part of the HTMX Starter. Its use is optional; the HX-Request header can also be checked using built-in functionality.

```
<form action="/" method="post" id="hx-form">
    <fieldset>
        <label>Firstname
        <input
                type="text"
                hx-trigger="keyup changed delay:250ms"
                hx-post="/"
                hx-include="closest form"
                hx-target="#hx-form"
                hx-swap="morph">
        </label>
    <fieldset>
    <button type="submit" value="Submit">Submit</button>
</form>
```

```
@HxRequest
@PostMapping("")
public String htmxPost(@Valid FormModel formModel, BindingResult bindingResult, Model
    model){
    model.addAttribute("formModel", formModel);
    model.addAttribute("validationResult", bindingResult);
    return "form";
}

@PostMapping("")
public String postIndex(@Valid FormModel formModel, BindingResult bindingResult, Model
    model){
    // usual form handling with redirect
}
```



The same form with validation errors highlighted.

Endless Scrolling

Infinite scrolling is often used for feeds or timelines. This, too, can be implemented declaratively using HTMX with just a handful of attributes.
Here, the request is triggered not by a click, but when the last element scrolls out of the viewport—and exactly once.
The next "page" is then appended to the end of the list. The new elements, in turn, have the same trigger, but with a different page number in the last element.

```
@for(var personEntry : ForSupport.of(persons.getContent()))
    @if (personEntry.isLast())
        <blockquote
        hx-get="/endless-scrolling?page=${persons.getNumber() + 1}&size=${persons.getSize
            ()}&sort=name,desc"
        hx-trigger="intersect once"
        hx-swap="afterend"
        >${personEntry.get().getQuote()}</blockquote>
    @else
        <blockquote>${personEntry.get().getQuote()}</blockquote>
    @endif
@endfor
```

Modal Dialogs

Modal dialogs are often dynamic (e.g., to confirm a destructive action) and require content from the server that is not yet available on the current page.
Creating all potential modal dialogs in advance would unnecessarily bloat the payload.
With HTMX, the content of a modal dialog can be loaded only when it is actually needed. Alpine.js can further simplify client-side interaction with the modal dialog.

```
<button  hx-get="/modal-dialogs"
         hx-target="#modals-here"
         hx-trigger="click"
         class="btn primary">Open Modal</button>
<div id="modals-here"></div>
```

Further Features

This article can only provide a brief overview. HTMX can be used to implement many other patterns, such as:

- client-side event trigger
- transitions and animation
- server-send-event and websockets

More examples can be found in the sample repository.

Conclusion

The concept of a hypermedia-driven application shown here can eliminate a great deal of complexity.
Learning HTML/HTTP and framework-independent techniques pays off better in the long run than learning a specific JavaScript framework.
Changes to the front end can be made much more quickly, since there's no need to first adjust the API contract, etc.

To return to the original question: Thanks to HTMX, the threshold at which an SPA is absolutely necessary has shifted significantly.



HTMX moves the threshold at which a single-page application becomes unavoidable a good way to the right.

Should we always use hypermedia (and HTMX) to build a web application? Of course, as often, the answer is: "It depends."

When building a typical business application with forms and many CRUD interfaces, hypermedia is a good choice—as it is for an e-commerce application.
In general, a UI that requires well-defined updates is a good use case.
In contrast, an application that requires a very large number of (unrelated) updates and has many dynamic UI elements (e.g., Miro) is generally not well-suited for a hypermedia-driven approach.

However, using the "Dynamic Island" pattern, both approaches can be combined so that SPAs and client-side logic are used only selectively and where necessary.

Related Links

- Hypermedia systems
- Free the web
- A modest critique of HTMX
- Less HTMX is more
- Fun with flags
- More samples

› Back to the Table of Contents ‹

ECLIPSE FOUNDATION

Vector Search with Pure Java

No External Database. Extremely fast. No Compromises.

NEW:

EclipseStore 4 - Java Embedded Vector Database

EclipseStore 4 integrates JVector, turning your app into a Java-native vector database. It eliminates 'Mapping Hell' and the need to manage external database processes - you get elite vector search performance with zero external dependencies.

EclipseStore 4 is just a tiny Java library available on Maven Central. The project is open-source under the Eclipse Public License (EPL 2.0) which allows you to use EclipseStore 4 for any commercial purposes. Check it out and if you like it, please star us on GitHub.

Project: www.eclipsestore.io
Code on GitHub: github.com/eclipse-store/store

NEW:

Eclipse Data Grid 4 - Distributed In-Memory Vector Search

Eclipse Data Grid is a Software-as-Code project that automates complex cluster orchestration, giving you a production-ready environment on your own terms. Built for on-premises and private cloud deployments, it enables you to run EclipseStore 4 as a distributed, in-memory vector search grid with full data sovereignty and zero reliance on public cloud providers.

Air-Gapped Ready: Standardized deployment logic for high-security, disconnected environments.

The project is open-source (EPL 2.0) - check it out on GitHub to bridge the gap between Java code and your private infrastructure, and please give us a star!

Code on GitHub: github.com/eclipse-datagrid/datagrid



Don’t Trust That Package: Applying Zero Trust to Open Source

ZeroTrust # OpenSource # SupplyChain

Author: Catherine Edelveis

Catherine Edelveis is a Developer Advocate at BellSoft and the co-host of CyberJAR, a YouTube channel dedicated to practical, state-of-the-art Java. She is focused on the JVM ecosystem and help teams ship performant and secure cloud-native Java applications using modern OpenJDK capabilities and proven open-source tools. Passionate about sharing knowledge, she writes and records JVM-related tutorials and guides, speaks at conferences, and hosts regular livestreams with leaders from across the Java community.



LinkedIN

Modern software is built from thousands of components that most organizations did not create, do not control, and may not even know they use. Open source is not inherently less trustworthy than proprietary software, but its popularity and a clean CVE report are not enough to justify blind trust.

This article explores how enterprises can apply zero trust to open source software, from analyzing the project health to gathering and continuously monitoring security signals.

What Is Zero Trust

Zero Trust is a software security model whose core idea is that no user, device, or software should be trusted automatically. Traditional security focused on defending the network perimeter ignoring the excessive privileges given to those who are inside. But modern systems are highly complex and don't have a clear inside/outside boundary. Applications now run across multiple cloud services and regions, networks, containers, and Kubernetes clusters, and go through a complex CI/CD pipeline.

So, an attacker doesn't need to break through one central firewall anymore. They can perform an attack by stealing a developer token, compromising a laptop, abusing a cloud identity, hijacking a CI runner, or publishing a malicious dependency.

The zero trust approach to security helps to reduce the potential damage by assuming that an attacker may already be inside the network. So, every request to access a system or resource must be explicitly verified based on the principle of least privilege. In addition, an enterprise must continuously analyze the risks to its business functions and take measures to mitigate these risks. The goal is to limit lateral movement and reduce the blast radius across distributed systems.

Zero trust was originally developed around access to systems and data. But it also fits well into the supply chain security paradigm. Applying it to software supply chain is an extension of the same principle:

Do not trust software merely because of where it came from, who maintains it, or whether it passed a security scanner. Verify the evidence for each artifact and restrict what it is allowed to do.

Putting blind trust in software can be especially risky when dealing with open source.

Why 5K Stars on GitHub Is Not Grounds for Blind Trust in OSS

It is hard to imagine an enterprise project that doesn't use open source software. According to Black Duck's 2026 Open Source Security and Risk Analysis report (Link), the average commercial application contains 1,180 open-source components.

Although zero trust should apply equally to proprietary libraries, SaaS services, CI plugins, compilers, container images, and deployment tools, open source deserves special attention because enterprises depend heavily on open source that they did not build, do not govern, and may not even know they use.

With OSS, an organization cannot directly control who maintains the project, who can publish releases, how well maintainers protect their accounts, how they build and release the package, and how promptly they patch vulnerabilities. That's why the risks posed by open source software cannot be reduced to the number of known CVEs. A library can be clean of critical CVEs, but still expose the organization through:

- Malicious code that can be introduced into open source components by many means, including:
  - Maintainer account takeover,
  - Burned out maintainer who trusted somebody who offered to help (XZ Utils backdoor is one example of such scenario),
  - Compromised package manager through which malicious packages are added,
  - Compromised compiler used by the OSS that adds malicious code,
  - Source-to-binary mismatch,
  - Build environment poisoning;
- Typosquatting that exploits typing errors directing users to malicious software;
- The good OSS component you use adds transitive vulnerable dependencies or dependencies with malicious code;
- The project can be abandoned, so there aren't going to be any patches;
- A maintainer may not be able to deliver patches on time;
- Upstream source code can be removed or taken down.

There is no implication that OSS is a black sheep of the software supply chain. But the usual lack of controls, visibility, and evaluation make it harder to assess the risks and easier for malicious actors to gain access to the system.

Applying Zero Trust Principles to Open Source Software

The original principles of zero trust architecture as defined by the NIST Special Publication 800-207 are not specific to software supply chains, but we can adapt them to govern how we select, verify, and monitor open-source components.

Below is not the official guidance, but rather, a theoretical mapping of traditional zero trust tenets to open source supply chain.

Traditional zero trust tenets applied to software supply chain.

Traditional zero trust	Open source supply chain
Never trust network location	Never trust package origin merely because it came from a familiar registry
Verify identity explicitly	Verify publisher, repository, workflow, and build identity
Least privilege	Restrict build credentials, install scripts, filesystem, and network access
Assume breach	Assume maintainers, accounts, registries, or CI systems may be compromised
Continuous evaluation	Re-evaluate every version, advisory, and project change
Protect resources	Contain dependencies at build time and runtime

Implementing Controls Throughout the Software Lifecycle

To build a zero trust open source supply chain, you need to introduce controls at all the stages of software development and deployment. Here are possible processes that you can introduce to gain more control over the open source your system is using.

Before Adoption

The first step is to determine whether you really need the dependency. Perhaps, another library you already use offers the same functionality? The goal is to reduce the overall number of dependencies to the most essential ones. Then, classify the dependency by business criticality: the more critical the dependency is, the more control it requires.

At this point, you should also analyze what transitive dependencies the library uses because malicious code and CVEs can often hide there. According to the same Black Duck's 2026 OSSRA report quoted earlier, 64% of open source components used in enterprise are transitive dependencies, meaning developers did not select them directly. At this stage, you already require a Software Bill of Materials because it gives a comprehensive data on dependencies, their versions, and licences. An SBOM should be analyzed and used further on in case you decide to keep the dependency.

Finally, verify the project health. You can use OpenSSF Concise Guide for Evaluating Open Source Software (Link) with a list of questions that help define the risk profile of a chosen library. The questions include:

- Is it a widely used project? The more popular it is, the more chances of good security controls in place and community-wide monitoring;
- How often are the updates published?
- What is security policy and vulnerability-response process?

And so on.

After that, you will have a comprehensive understanding of the dependency's necessity, criticality, and health, and will be able to make an informed decision about whether to reject or adopt it.

During Ingestion

After you approve the dependency, the next step is to control how it enters the pipeline.

First, restrict the permitted download mechanisms. Dependencies should be acquired through organization-approved package managers, artifact repositories, registry proxies, or similarly controlled services. All transfers should use secure protocols such as HTTPS with modern TLS configuration and proper certificate validation to protect the connection from interception and tampering.

Dependencies should then be pinned to immutable cryptographic digests. Pinning makes builds predictable and ensures that the artifact reviewed today is the artifact retrieved tomorrow.

Where signatures and provenance are available, they should be verified automatically. A valid signature can confirm the identity associated with an artifact and show that its contents have not changed since signing. Provenance can link the released artifact to its source repository, commit, build process, and build environment.

This leads to an essential rule: trust the release, not merely the project name. A well-established project may have been secure for years and still publish a compromised release after a maintainer account or build workflow is breached. Also, the visible GitHub source is not necessarily what ended up in:

- the npm or pip package;
- the container image;
- the operating-system package.

Provenance helps establish the connection between source and artifact, while reproducible builds allow us to confirm that the same source produces the same result. For especially critical components, an organization may choose to rebuild the artifact from a reviewed source in its own controlled environment.

Container base images require the same scrutiny. Organizations should consider hardened base images that reduce unnecessary components and provide verifiable security metadata. BellSoft Hardened Images, for example, are designed for a reduced attack surface and low-to-zero known CVEs, and are distributed with signatures and SBOMs that can be checked during ingestion.

Before an artifact is admitted, it should be scanned for known vulnerabilities, policy violations, and indicators that it has been identified as malicious. Policies can block known malicious packages automatically and evaluate vulnerable components according to severity, exploitability, reachability, component criticality, and available remediation.

Finally, the ingestion process should retain an approval record containing the component name, exact version or digest, source, approving person or team, supporting evidence, intended use, and reason for acceptance.

As a result of all these actions, an enterprise will have a traceable chain of custody connecting all essential bits together, namely: an approved source, an artifact, verified evidence, and security checks.

During Builds

The build environment processes source code, third-party dependencies, plugins, and scripts that may be compromised. Therefore, it should be treated as a high risk zone with tight controls applied, namely:

- The pipeline should provide a sandboxed environment and automatically run security checks on every commit.
- When credentials are required, they should be short-lived and narrowly scoped.
- Network access should also be restricted to approved repositories and services.
- Generate an SBOM identifying the included components and provenance linking the artifact to its source, commit, workflow, and build environment.
- Sign the final artifact so that its identity and integrity can be verified before deployment.

Together, these controls make the build process isolated, reproducible, traceable, and harder to abuse.

Before Deployment

Before deployment, the release should pass automated policy checks that verify its signatures and provenance generated during the previous stage. Any artifact containing unexpected or policy-violating components should be rejected before it reaches the target environment.

The artifact must also match the reviewed and approved build through a digest. This prevents tampering between the build and deployment stages.

Note that the application should run with the minimum permissions required, including restricted access to the filesystem, network, secrets, operating-system capabilities, and cloud resources. The least privilege approach limits the damage if the artifact is still compromised.

After Deployment

After deployment, you should continuously monitor components for newly disclosed vulnerabilities, malicious-package reports, maintainer compromise, and changes in support status. For critical components, teams should prepare alternatives and be ready to upgrade, replace, or maintain the dependency internally when upstream support is no longer sufficient.

Of course, not every problem requires the same response. Prioritize the risk. You can use reachability analysis and VEX statements to determine whether a vulnerable component is exploitable in the deployed application.

Security Signals and What They Prove

As you gradually implement controls, you will get more valuable data about the software you use. But each of these signals reveals only one part of the overall risk picture. An SBOM, vulnerability scan, signature, or provenance record can answer a specific question, yet none of them proves that a component is fully trustworthy on its own. Consider these signals as puzzle pieces that will give you a clear picture only when gathered together.

The table below shows what each security signal can establish, and — just as importantly, — what it cannot do.

Security signals.

Signal	What it tells you	What it does not prove
SBOM	Which components are reported as present	That they are safe or authentic
CVE scan	Known vulnerabilities match components	Absence of malware, backdoors, or unknown flaws
Signature	Artifact integrity and signing identity	That the signer or source was trustworthy
Provenance	How and where an artifact was built	That the source code was benign
OpenSSF Scorecard	Repository security practices	That a particular release is safe
VEX/reachability	Whether a vulnerability may affect this product	Artifact authenticity or project integrity
Reproducible build	Released artifact corresponds to source	That the source itself is secure

Looking at the limitations mentioned in the table, it is tempting to assume that these signals are optional or ineffective. This is not so. Their value comes from using them together. Each answers a different question, and together, they provide the evidence needed to make informed and risk-based trust decisions.

Guidelines You Should Start With

Secure software development practices as well as requirements to software supply chain security have been under the ramplights of many regulations. Here are some robust sources you can start with if you want to build zero trust for your pipeline:

- NIST SP 800-218: Secure Software Development Framework (Link). This is the core secure-development reference. It pays a lot of attention to protecting software, producing well-secured releases, verifying releases and securing toolchains.

- NIST Special Publication 800-207: Zero Trust Architecture (Link). This one defines the zero-trust principles: no implicit trust based on location or ownership, least privilege, explicit decisions, and continuous evaluation. These principles are applied to system access, but you can use them for inspiration on building a zero trust software supply chain.
- Secure Supply Chain Consumption Framework, S2C2F (Link). This is one of the frameworks directly concerned with open source security. It aims to define how to securely consume OSS dependencies into the developer's workflow, including inventorying, scanning, preventing malicious code, and rebuilding in trusted environments.
- OpenSSF Concise Guide for Evaluating Open Source Software (Link). This one is a highly practical questionnaire that can help you define your stance on a particular OSS component. It covers authenticity, maintenance, security policies, testing, vulnerabilities, suspicious code, indirect dependencies, and sandbox testing.
- OpenSSF Scorecard. This one is a tool. After the installation, it will automatically accumulate project-health signals such as branch protection, dangerous workflows, dependency updates, fuzzing, security policies, maintenance and known vulnerabilities for open source projects used in your repo.
- SLSA. This framework covers build and source tracks and uses provenance and attestations to provide increasing assurance levels. This is a good source to understand how to implement security controls gradually.

Possible Gradual Introduction of Controls

Building a zero trust open source supply chain is not a matter of one day — and that's alright. The existing guidelines recommend introducing changes to the existing workflow gradually and build on what you already have, step by step.

So how might the milestones of the zero trust integration look like based on what we discussed above?

Step 0: Developers download whatever they want and however they want. There's no provenance, no scanning in place, and consequently, no visibility into the supply chain. Risk management is reactive.

Step 1: You know what you use. You implement dependency inventory, generate SBOMs, and have vulnerability scanning in place. Dependency versions are pinned.

Step 2: You control consumption. Developers use only approved sources and download methods. You have dependency policies in place, perform open source project evaluation, and block malicious packages. Internal proxying is implemented.

Step 3: You verify evidence. You use signature and provenance verification, isolated builds, short-lived credentials, and policy-enforced deployment.

Step 4: You make sure that open source components continuously earn trust. You perform continuous reassessment, independent rebuilds where justified, and runtime containment. You have rapid replacement capability in place and invest directly in critical upstream projects.

This model is not an official framework, or substitute for SLSA or other established software supply-chain standards. It is simply a possible blueprint for introducing zero trust controls incrementally and adapting them to an organization's specific environment.

The appropriate pace and depth will depend on the organization's risk profile, resources, regulatory obligations, and the criticality of the software it produces.

Before You Leave: Open-Source Sustainability as a Security Control

Instead of a conclusion, it is important to look at another security control as essential as others — open source sustainability.

Drawing the lessons from the XZ Util backdoor, CISA argues that manufacturers and operators using OSS share responsibility for supporting secure development and sustainable maintainer communities. Companies using open source software to build their solutions should consider funding maintainers or contributing engineering time to help open source projects stay secure and up-to-date.

It is hard to disagree with this point of view. Despite the fact that open source continues driving innovations forward, the situation when an overworked maintainer is the only one keeping a highly popular open source component alive is not a rare one. Open source is about collaboration, not exploitation. Consumers shouldn't demand enterprise-level response from unpaid maintainers while contributing nothing.

So, if you organization uses open source software, consider how you can make this process more reliable and sustainable:

- Fund maintainers directly, sponsor foundations, or pay for security audits and infrastructure;
- Contribute fixes, documentation, testing, vulnerability reports, and engineering time upstream;
- Avoid maintaining large internal forks when improvements could be contributed back to the project;
- Prepare a contingency plan for components that become abandoned, compromised, or impossible to update.

For business-critical components, such as the Java runtime environment, container base images, databases, or core frameworks, consider obtaining commercial support from a vendor. Paid support gives the organization clearer ownership and predictable patching commitments under a strict SLA.

Supporting open-source sustainability is not simply an act of goodwill. It reduces the risk created by exhausted maintainers, abandoned projects, and uneven release processes. So, a mature zero trust strategy should not only verify open-source components at the point of consumption. It should also help preserve the ecosystems on which the organization depends.

› Back to the Table of Contents ‹

$ secure-coding --lang=java v.2026 · hands-on

Secure Coding in Java

// from threat model to hardened production

A code-first workshop across eleven focused modules — supply-chain security, threat modelling, the JDK's own cryptography, and a full web-app hardening deep dive. Led by **Sven Ruppert**.

// curriculum [11]

01 Supply Chain Security

02 Fundamental Security Principles

03 Threat Modelling — STRIDE & DREAD

04 Security Testing — SAST · DAST · IAST · RASP

05 Core Definitions — SBOM · CWE · CVE · EPSS

06 Attack Vectors in the Wild

07 Obfuscation Techniques

08 Selected CWEs

09 Mutation Testing

10 Cryptography in the JDK

11 Hardening a Web App — Deep Dive

add-ons: BouncyCastle · Post-Quantum Cryptography (PQC) for Java developers

two days · in-person or remote
[dates & pricing on request]

> reserve your place

contact@sven-ruppert.com



Java 27's Post-Quantum TLS Handshake: How JEP 527 Brings Hybrid Key Exchange to JSSE

PostQuantum # TLS # JEP527

Author: Babneet Singh

Babneet Singh is a software engineer working on runtime components in IBM Semeru Runtimes, which is based on the Eclipse OpenJ9 and OMR projects. He is a committer on both open-source projects and contributes to JVM runtime development with a focus on correctness, maintainability, and long-term sustainability. He collaborates with engineers across the JVM stack, supports low-level runtime work, and participates in open-source discussions on JVM architecture and design. His interests include JVM internals, runtime concurrency, and building reliable virtual machines for modern Java workloads.



LinkedIN

Hybrid post-quantum key establishment without an application API rewrite, and the compatibility engineering hidden inside Java 27's ClientHello

Most Java developers will never call ML-KEM directly. They may still start using it.

That apparent contradiction is the interesting part of JEP 527, *Post-Quantum Hybrid Key Exchange for TLS 1.3*. Rather than introducing another cryptographic API that applications must explicitly adopt, Java 27 integrates hybrid post-quantum key establishment into SunJSSE, the JDK's TLS implementation.

For an application already using standard Java TLS APIs, the source code may not change at all. What changes is the handshake underneath it. A Java 27 client can offer a hybrid X25519 + ML-KEM exchange and ordinary X25519 at the same time. A server that supports the hybrid can use it immediately. A server that does not can continue with X25519.

That compatibility mechanism matters at least as much as the new cryptography.

The TLS Traffic Recorded Today May Matter Tomorrow

Post-quantum TLS addresses a threat that can begin before cryptographically relevant quantum computers exist.

An adversary can record encrypted TLS traffic today and retain it. If a future quantum computer can break the public-key mechanism that established the session secret, some of that recorded traffic could become decryptable later. This is commonly called *harvest now, decrypt later*.

Hybrid key exchange is a migration strategy for that problem. Instead of replacing a mature classical algorithm with a post-quantum algorithm in one step, the handshake combines both. RFC 9954 defines hybrid key exchange as using multiple mechanisms together with the goal of retaining security if all but one of the components are later defeated. RFC 10024 applies that construction to ML-KEM and elliptic-curve Diffie-Hellman for TLS 1.3.

There is an important boundary: **JEP 527 changes TLS key establishment, not certificate authentication**.

The server certificate, certificate chain, and the signature algorithm used to authenticate the peer remain a separate part of TLS. An RSA- or ECDSA-authenticated connection can therefore use hybrid key establishment, but JEP 527 does not make that certificate signature post-quantum. Calling such a connection "fully quantum-safe" would be misleading.

This is also a JDK security-library change, not a JVM execution feature. The implementation sits in SunJSSE and the JDK security classes, not in HotSpot, OpenJ9, the JIT, GC, or bytecode execution. The JDK 27 CSR classifies the change under `security-libs/javax.net.ssl` and states that applications using JSSE benefit from the new algorithms by default.

What Java 27 Changes, and Why It Is Hybrid

JEP 527 adds three RFC 10024 hybrid TLS groups to SunJSSE:

Named group	Classical component	Post-quantum component
`X25519MLKEM768`	X25519	ML-KEM-768
`SecP256r1MLKEM768`	secp256r1	ML-KEM-768
`SecP384r1MLKEM1024`	secp384r1	ML-KEM-1024

Only `X25519MLKEM768` is added to SunJSSE's default named-group list, ahead of `x25519`; the other two are available for explicit configuration. RFC 10024 likewise marks `X25519MLKEM768` as the recommended group.

Hybrid does not mean double-encrypting application records. TLS still produces one set of traffic keys and protects application data normally. What changes is the material from which those keys are established: `X25519MLKEM768` combines an ML-KEM secret with an X25519 secret.

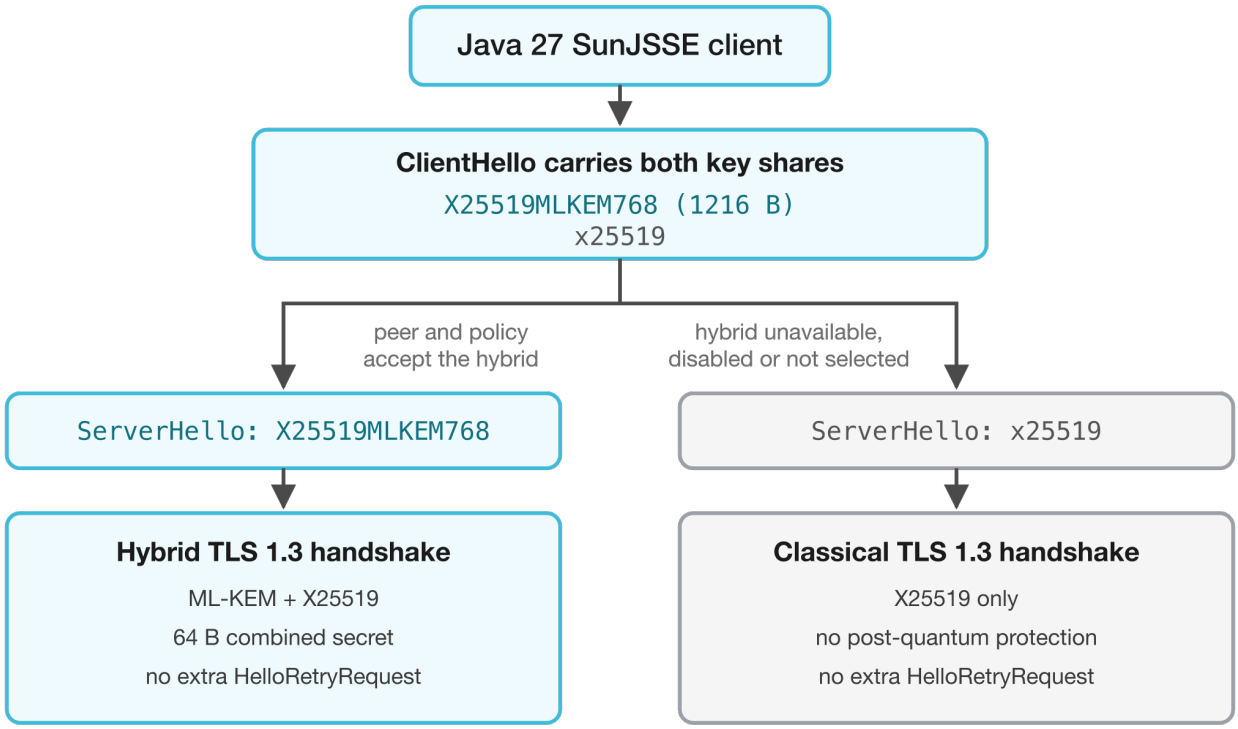
The construction is designed so that the resulting key establishment retains its security if at least one component remains secure. If ML-KEM were later found to have a weakness, X25519 still contributes classical security. If a sufficiently capable quantum computer defeats X25519, ML-KEM supplies the post-quantum component.

The Centrepiece: Two Key Shares in the First ClientHello

TLS 1.3 separates the groups a client says it supports from the actual key shares it sends.

That distinction creates a migration problem. Suppose Java preferred `X25519MLKEM768` but sent only that key share. A TLS 1.3 server that understood X25519 but not the hybrid group could not immediately use the offered key material. It might need a `HelloRetryRequest` (HRR) asking the client for another share.

Java 27 takes a compatibility-first approach. Under the default SunJSSE configuration, the client sends **two key shares** in its initial `ClientHello` — an `X25519MLKEM768` share and an `x25519` share.



Both key shares travel in the first ClientHello, so the server can take the hybrid group or fall back to x25519 without an extra HelloRetryRequest.

A hybrid-capable server can select `X25519MLKEM768` immediately. A server that does not support the hybrid but does support X25519 can select the already-present `x25519` share.

In both common cases, the server does not need an HRR merely to obtain a different share. This is **not TLS 0-RTT**; that term has a specific meaning involving early data. It is simply the normal TLS 1.3 handshake proceeding without an additional HRR round trip.

Fallback is compatible, but it is not post-quantum protection:

Peer and policy	Result	Post-quantum protection
Both accept `X25519MLKEM768`	Hybrid exchange	Yes, under the hybrid security assumptions
Hybrid unavailable, both accept `x25519`	Classical X25519	No
Another advertised group is common but no usable share was sent	HRR and new share	Depends on requested group
No common permitted group	Handshake fails	Not applicable

HRR has therefore not disappeared. JEP 527 avoids an unnecessary retry in the important hybrid-to-X25519 compatibility case; normal TLS negotiation still applies elsewhere.

What Happens on the Hybrid Path

ML-KEM is a key-encapsulation mechanism, so the server side is not identical to ordinary ephemeral Diffie-Hellman.

For `X25519MLKEM768`, the client sends its ML-KEM-768 encapsulation key together with an ephemeral X25519 share. The server encapsulates an ML-KEM shared secret to the client's key and separately completes the X25519 exchange. Its `ServerHello` returns the ML-KEM ciphertext and its own X25519 share.

The client then decapsulates the ML-KEM ciphertext and completes X25519. Both sides arrive at the same two component secrets.

RFC 10024 defines the exact layout:

```
Client hybrid key_exchange

ML-KEM-768 encapsulation key    1184 bytes
X25519 ephemeral share            32 bytes
                                ----------
                                1216 bytes
Server hybrid key_exchange

ML-KEM-768 ciphertext           1088 bytes
X25519 ephemeral share            32 bytes
                                ----------
                                1120 bytes
```

The resulting shared secret is:

```
ML-KEM shared secret      32 bytes
X25519 shared secret      32 bytes
                          --------
                          64 bytes
```

These are standardized wire formats, not OpenJDK-private encodings. RFC 10024 also contains a naming trap: despite the name `X25519MLKEM768`, the ML-KEM material comes first on the wire. The RFC calls this a historical exception to the hybrid-group naming convention.

The same order applies to the component shared secrets: ML-KEM first, X25519 second. That ordering has a separate regulatory context. RFC 10024 discusses NIST SP 800-56C requirements for HKDF-based derivation with multiple shared secrets; this should not be confused with the historical reason for the wire encoding order.

What the JDK Actually Built

JEP 527 makes more sense as the third step in a longer JDK security evolution:

```
Java 21               Java 24                    Java 27
JEP 452               JEP 496                    JEP 527

javax.crypto.KEM  -->  ML-KEM implementation  -->  TLS integration
```

JEP 452 supplied a general API through which security providers can expose key-encapsulation mechanisms. JEP 496 added the NIST-standardized ML-KEM implementation. JEP 527 then connects those primitives to TLS 1.3.

The implementation is interesting because of the abstraction boundary it creates. The integrated OpenJDK change adds or modifies `Hybrid`, `HybridProvider`, `DHasKEM`, `KEMKeyExchange`, `KeyShareExtension`, `KAKeyDerivation`, `NamedGroup`, and related SunJSSE code. It also adds dedicated tests for ClientHello key shares and HRR behaviour, along with JMH benchmarks for KEMs, key-pair generation, and TLS handshakes.

`HybridProvider` is an internal provider used by the TLS implementation to expose the services required to compose the hybrid groups. It is not a new provider that applications are expected to install.

`DHasKEM` is particularly revealing. Its source describes it as a **KEM abstraction over traditional DH-based key exchange**, usable for ECDH/XDH and hybrid TLS exchanges. In effect, it lets JSSE treat X25519 and an actual KEM such as ML-KEM through a sufficiently common internal shape.

`KeyShareExtension` then drives the ClientHello share generation from the configured named-group preferences. With Java 27 defaults, the outcome is the hybrid share plus X25519. That is a default policy, not a protocol invariant: replacing or reordering the configured groups can change what gets emitted.

On the derivation path, `KAKeyDerivation` carries the resulting ordered hybrid secret into the **existing TLS 1.3 HKDF-based key schedule**. JEP 527 does not require applications to know that any of these internal classes exist.

Post-Quantum Security Has a Wire Cost

The compatibility benefit is not free in bytes.

The default client contributes:

```
1216 bytes   X25519MLKEM768
  32 bytes   x25519 fallback
----------
1248 bytes   key-exchange bodies alone
```

That is before the `key_share` framing, other TLS extensions, SNI, ALPN, cipher suites, and the rest of the `ClientHello`.

A common Ethernet MTU is 1500 bytes, but that does **not** mean every Java 27 ClientHello splits into a fixed number of packets. TCP MSS, IPv4 versus IPv6, extensions, tunnelling, and the actual network path all matter. The useful conclusion is more modest: the larger first flight may span additional TCP segments.

That makes real infrastructure worth testing. TLS inspection appliances, older proxies, service meshes, load balancers, custom TLS parsers, and lossy networks now see a substantially larger initial handshake. Losing part of a larger first flight can delay progress while TCP retransmits missing data. QUIC uses the same TLS handshake concepts but has its own datagram-size and amplification considerations.

There is additional computation as well. The client constructs the hybrid share before knowing whether the server will select it. But quoting a generic latency or CPU percentage would be misleading. The OpenJDK patch includes primitive and `SSLEngine` handshake benchmarks; those are tools for measuring the implementation, not proof of a universal production cost.

Measure complete handshakes through the real network path, and separate fresh connections from resumed ones.

FIPS nuance: RFC 10024 explains how the component-secret ordering can be used with NIST-approved key-derivation constructions. That does not make a stock JDK, provider, or deployment FIPS 140-3 certified. Certification applies to the implementation and operating configuration, not merely to use of an RFC-defined group.

What Java Applications Get Automatically

For ordinary Java code, the attraction of JEP 527 is how little there is to do:

```
var client = HttpClient.newHttpClient();

var request = HttpRequest.newBuilder(
        URI.create("https://example.com"))
    .build();

client.send(request, HttpResponse.BodyHandlers.discarding());
```

There is no:

```
enablePostQuantumTLS();
```

Applications do not create ML-KEM keys, call `KEM`, combine secrets, or modify the TLS key schedule themselves.

"Automatic" still has boundaries. The connection must use TLS 1.3; the active JSSE/provider must implement the group; the peer must support and accept it; and configuration must not have removed it. OpenJDK's JDK 27 guidance confirms that standard `javax.net.ssl` applications can benefit by default when named groups have not been overridden.

A framework using OpenSSL, BoringSSL, or another native TLS stack does not gain SunJSSE behaviour simply because its application runs on Java 27.

How Configuration Can Silently Turn It Off

Production Java installations often customize TLS, and that can quietly bypass the new default.

Consider:

```
-Djdk.tls.namedGroups=x25519,secp256r1
```

`jdk.tls.namedGroups` is a Java system property containing the enabled named groups in preference order. A non-empty explicit list overrides the provider defaults rather than appending to them. Since `X25519MLKEM768` is missing here, the hybrid group will not be offered.

A deliberately minimal hybrid-plus-fallback policy would be:

```
-Djdk.tls.namedGroups=X25519MLKEM768,x25519
```

There is an important empty-value distinction. If the system property is absent or empty, SunJSSE uses its implementation defaults.

Applications can also configure groups through `SSLParameters`:

```
SSLParameters parameters = socket.getSSLParameters();

parameters.setNamedGroups(new String[] {
    "X25519MLKEM768",
    "x25519"
});

socket.setSSLParameters(parameters);
```

This API predates Java 27; `setNamedGroups()` and `getNamedGroups()` have existed since Java 20. A null named-group configuration allows provider defaults, while an empty array turns named-group negotiation off. A non-empty array overrides the provider-specific default groups.

`getNamedGroups()` is therefore useful configuration evidence, but it is not proof of what a particular connection negotiated.

Verify the Handshake, Not the Source Code

For a controlled test environment, JSSE diagnostics remain useful:

```
-Djavax.net.debug=ssl,handshake
```

Do not look for a fabricated universal log line; the exact debug text is implementation- and build-dependent. Instead, answer concrete questions:

- Did the ClientHello offer `X25519MLKEM768`?
- Was `x25519` also present?
- Which group did the server select?
- Was an HRR required?
- Does the result change when the connection crosses the real proxy, ingress, mesh, load balancer, or TLS terminator?

Packet captures or controlled test servers can provide more stable automated evidence than parsing diagnostic prose.

The important operational distinction is this:

Handshake availability and cryptographic posture are different metrics.

If the server selects `x25519`, the TLS connection succeeded. From an availability perspective, that is good news. From a post-quantum migration perspective, it is a miss.

Track hybrid selection separately from TLS success whenever the observability stack permits it. A rollout can be perfectly healthy by error-rate metrics while unexpectedly sending much of its traffic through the classical fallback.

What About OpenJ9 and Eclipse Semeru?

JEP 527 does not require a separate OpenJ9 VM implementation.

IBM Semeru Runtimes combine OpenJDK class libraries with the Eclipse OpenJ9 VM. IBM's own release documentation describes those as distinct runtime components. Because JEP 527 lives in the JSSE/OpenJDK security-library layer, the architecture does not require OpenJ9 to implement another TLS handshake in the VM.

That architectural fact should not be stretched into an unconditional product-support claim. Before deployment, verify the exact Java 27 Semeru distribution and build, its class-library level, provider configuration, and the behaviour of an actual handshake.

Production Rollout

A practical Java 27 rollout should answer a short set of questions:

- **Which TLS stack is actually in use?** SunJSSE, a third-party JSSE provider, or native TLS?
- **Are there named-group overrides?** Search launch scripts, container images, framework configuration, and programmatic `SSLParameters`.
- **What does the client actually send?** Confirm both hybrid and classical shares under the intended defaults.
- **What do representative peers select?** Test hybrid-capable and classical TLS 1.3 servers.
- **Does HRR behaviour match expectations?** Include cases where no initially supplied share is acceptable.
- **Does the real network path tolerate the larger handshake?** Test through the same proxies, meshes, firewalls, and termination points used in production.
- **What does it cost?** Measure fresh-handshake latency, CPU, allocations, bytes, failure rates, and packet-loss sensitivity.
- **Can you see fallback?** Treat `x25519` as a compatibility success but a classical-security outcome.

The last point is the one most likely to be missed. A configuration that "works everywhere" is not necessarily a configuration that is negotiating hybrid key establishment where expected.

The Real Takeaway

JEP 527 adds post-quantum cryptography to Java, but the most interesting engineering choice is not ML-KEM itself. It is the migration strategy around it.

Java 27 does not ask application developers to adopt a new TLS API. It does not require every server on the other end of a connection to become post-quantum before clients can upgrade. And it does not discard X25519 interoperability to move the ecosystem forward.

Instead, SunJSSE puts the hybrid and classical options side by side in the first TLS 1.3 handshake. A capable peer can select `X25519MLKEM768`. An existing X25519 peer can keep working. The fallback remains classical, and production systems should make that distinction visible.

That is what makes JEP 527 more interesting than simply saying that Java 27 "supports post-quantum TLS": **the cryptography is new, but the migration is deliberately incremental.**

References

- JEP 527 – Post-Quantum Hybrid Key Exchange for TLS 1.3 (Link)
- JDK-8314323 – Implement JEP 527: TLS 1.3 Hybrid Key Exchange (Link)
- JDK-8369848 – Implement JEP 527: TLS 1.3 Hybrid Key Exchange (CSR) (Link)
- OpenJDK PR #27614 – Implement JEP 527: TLS 1.3 Hybrid Key Exchange (Link)
- OpenJDK integrated implementation of JEP 527 (Link)
- RFC 9954 – Hybrid Key Exchange in TLS 1.3 (Link)
- RFC 10024 – Post-Quantum Traditional (PQ/T) Hybrid Key Agreement Mechanisms for TLS 1.3 (Link)
- JEP 452 – Key Encapsulation Mechanism API (Link)
- JEP 496 – Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism (Link)
- NIST FIPS 203 – Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) (Link)
- Java SE 26 SSLParameters API – Named-group configuration (Link)
- Java SE 26 JSSE Reference Guide – `jdk.tls.namedGroups` system property (Link)
- IBM Semeru Runtimes Release Notes – OpenJDK class libraries and Eclipse OpenJ9 VM (Link)

› Back to the Table of Contents ‹





Polyglot Programming with GraalVM: Calling Python, JavaScript, Ruby, and R from Java

GraalVM # Polyglot # Truffle

Author: Atiq Amjad

He´s a technology leader with a strong background in Java and software architecture. Atiq Amjad began his journey as a Java developer, and his passion for the Java ecosystem and open source has stayed with him throughout his career.



LinkedIN

GraalVM's polyglot capabilities allow Java applications to seamlessly integrate with Python, JavaScript, Ruby, and R, enabling developers to leverage each language's strengths within a single runtime. GraalVM lets guest languages execute within the same runtime, sharing the same heap and JIT optimizations via the Truffle Language Implementation Framework. In this article, we demonstrate practical, production-oriented example, performance benefits, interoperability, and real-world scenarios where combining multiple languages produces cleaner, faster, and more maintainable enterprise solutions.

Why Polyglot Programming?

Modern enterprise systems rarely depend on just one programming language. Data science teams prefer Python, frontend developers work with JavaScript, statisticians use R, and DevOps scripts are typically written in Ruby. Since each language is optimized for particular tasks, these systems often require the best of multiple worlds.

Language	Best suited for
Java	Performance, scalability, and robust enterprise features
Python	Data science, AI, machine learning, and scientific computing
JavaScript	Dynamic web interactions and business rules
Ruby	Metaprogramming, DSLs, and developer productivity
R	Statistical analysis and visualization

Traditionally, these languages communicate using:

- REST APIs
- JNI
- JSR-223
- External processes
- Message queues

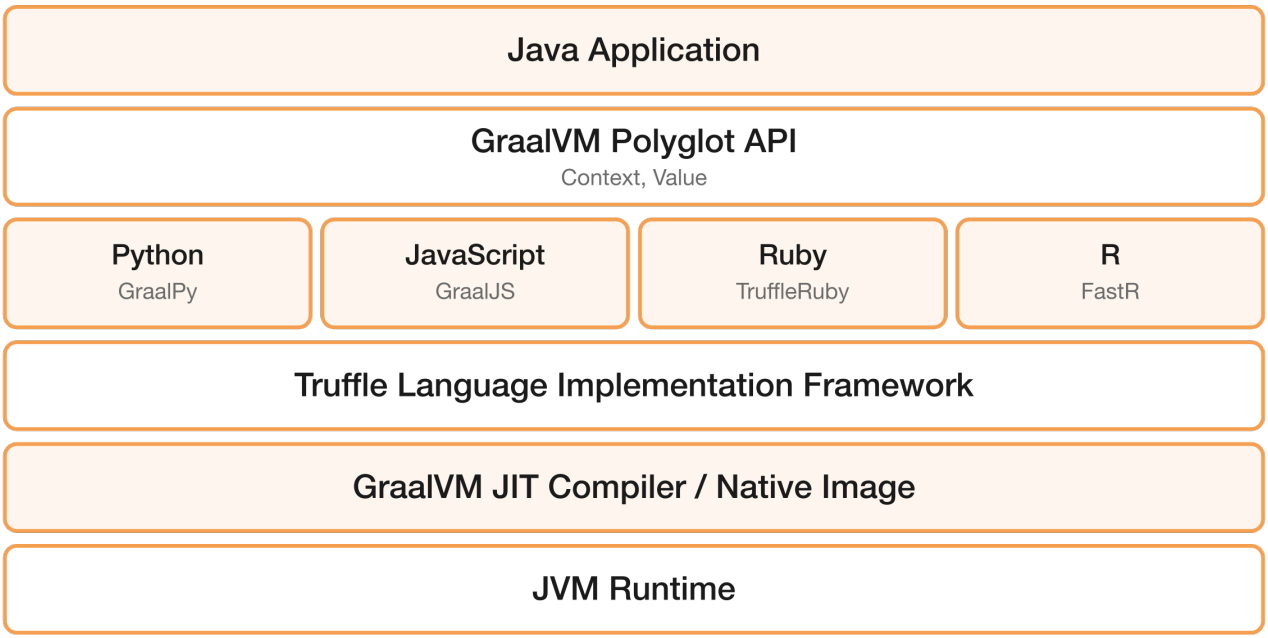
Each approach introduces serialization, latency, deployment complexity, and operational overhead.

GraalVM's polyglot runtime eliminates the need for clunky inter-process communication (IPC), REST APIs, or JNI bridges, enabling direct, high-performance calls across languages.

GraalVM's Polyglot Architecture

The secret to GraalVM's unified execution is Truffle, an open-source framework for building AST-interpreting language implementations that all compile to the same intermediate representation. Because *Python (GraalPy), JavaScript (GraalJS), Ruby (TruffleRuby),* and *R (FastR)* are all Truffle language implementations, they share:

- A common **Polyglot Context** (`org.graalvm.polyglot.Context`) that hosts one or more language engines.
- A unified **object model** (`Value`) so a Java `Map`, a Python `dict`, and a JS `object` can be exchanged with zero-copy semantics where possible.
- **Partial evaluation:** Truffle specializes each language's interpreter into optimized native machine code at the same tier as compiled Java.



The Truffle stack: one polyglot context above the language implementations, all compiled by the same GraalVM JIT.

The Polyglot API: Core Concepts

The **Polyglot API** allows Java to:

- Evaluate scripts or expressions in other languages.
- Call functions defined in other languages.
- Exchange data (primitives, objects, arrays, and functions).
- Manage contexts (isolation, resource cleanup).

Language IDs

A unique identifier represents each guest programming language.

Language identifiers.

Language	ID
Python	python
JavaScript	js
Ruby	ruby
R	R (uppercase)

Context

A `Context` is an isolated execution environment for polyglot code. You must explicitly permit each programming language you intend to use:

```
try (Context context = Context.newBuilder("python", "js", "ruby", "R")
        .allowAllAccess(true) // For demo; restrict in production
        .build()) {
    // Rest of the code
}
```

Value

The `Value` class is the universal handle for guest language data (e.g., a Python list, a JavaScript function). It provides type-safe accessors:

```
try (Context context = Context.newBuilder()
        .allowAllAccess(true)
        .build()) {

    Value result = context.eval("js", "21 + 21");
    System.out.println(result.asInt()); // 42
}
```

Prerequisites & Dependencies

GraalVM Installation

- Download GraalVM Community Edition (or Enterprise) from Oracle GraalVM.
- Set `JAVA_HOME` to the GraalVM installation directory.

To execute multiple guest languages inside Java, include the `org.graalvm.polyglot` artifacts along with the target language implementations in your build configuration.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>polyglot-demo</artifactId>
    <version>1.0</version>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
        <graalvm.version>25.0.0</graalvm.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>polyglot</artifactId>
            <version>${graalvm.version}</version>
        </dependency>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>python</artifactId>
            <version>${graalvm.version}</version>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
```

```
            <artifactId>js</artifactId>
            <version>${graalvm.version}</version>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>ruby</artifactId>
            <version>${graalvm.version}</version>
            <type>pom</type>
        </dependency>
    </dependencies>
</project>
```

Note: *FastR (Oracle's implementation of R on GraalVM)* is not yet available as a Maven artifact. For R, install GraalVM R via `gu install R` and use the GraalVM JDK directly. If you don't want to use Maven or Gradle for other guest languages, you can also use the following command to install the additional languages.

```
 gu install python js ruby R
```

Maven Configuration (pom.xml)

```
<dependencies>
    <dependency>
        <groupId>org.graalvm.polyglot</groupId>
        <artifactId>polyglot</artifactId>
        <version>25.0.0</version>
    </dependency>
</dependencies>
```

Gradle Configuration (gradle.build)

```
dependencies {
    implementation 'org.graalvm.polyglot:polyglot:25.0.0'
}
```

Building the Polyglot Application

Complete Runnable Example

We are building a simple Java risk scoring program `LoanApproval` where:

- **Java** handles orchestration, domain objects, and final output.
- **Python** runs a scoring function.
- **JavaScript** validates business rules.
- **R** computes a statistical confidence interval.
- **Ruby** formats a human-readable report.

```
package com.example;

public class LoanApproval {

    public static void main(String[] args) {
        // 1. Java: build financial data in Java
        Map<String, Object> financialData = new HashMap<>();
        financialData.put("income", 65000.0);
        financialData.put("debt", 15000.0);
        financialData.put("creditHistoryYears", 6);

        // Use try-with-resources to ensure Context is properly closed
        try (Context context = Context.newBuilder("python", "js", "ruby", "R")
                // R requires all access for this demonstration
                .allowAllAccess(true) // start strict, upgrade selectively
```

```
                .out(System.out)
                .err(System.err)
                .build()) {

            // 2. Python: compute scores from financial data
            double score = scoreWithPython(context, financialData);

            // 3. JavaScript: validate score and debt to determine approval
            boolean approved = validateWithJavaScript(context, score, financialData);

            // 4. R: compute a confidence interval for the score
            double[] interval = confidenceIntervalWithR(context, score);

            // 5. Ruby: compute a report with the results
            String report = formatWithRuby(context, financialData, score, approved,
                interval);

            // 6. Java: print the report to the console
            System.out.println(report);

        } catch (PolyglotException e) {
            // Guest-language exceptions surface here with source location
            System.err.println("Polyglot error in " + e.getGuestObject()
                    + ": " + e.getMessage());
        }
    }

    static double scoreWithPython(Context context, Map<String, Object> financialData) {
        Value pyFunction = context.eval("python", """
                def score(income, debt, history):
                    dti = debt / income
                    base = 100 - (dti * 120)
                    return max(0, min(100, base + history * 1.5))
                score
                """);
        // Execute the Python function with Java objects and return the result as a
        // double
        Value result = pyFunction.execute(
                financialData.get("income"), financialData.get("debt"),
                financialData.get("creditHistoryYears"));
        return result.asDouble();
    }

    static boolean validateWithJavaScript(Context context, double score,
            Map<String, Object> financialData) {
        // Return a JavaScript function from eval
        Value jsFunction = context.eval("js", """
                (score, debt) => score >= 55 && debt < 25000
                """);
        // Execute the function with Java object and return the result as a boolean
        return jsFunction.execute(score, financialData.get("debt")).asBoolean();
    }

    static double[] confidenceIntervalWithR(Context context, double score) {
        // R requires allowAllAccess(true) to run
        Value rFunction = context.eval("R", """
                function(s) {
                se <- 4.2
                c(s - 1.96 * se, s + 1.96 * se)
                }
                """);
        // Execute the R function and return the result as a double array
        Value vec = rFunction.execute(score);
        return new double[] { vec.getArrayElement(0).asDouble(),
                vec.getArrayElement(1).asDouble() };
```

```
    }

    static String formatWithRuby(Context context, Map<String, Object> financialData,
            double score, boolean approved, double[] confidenceInterval) {
        Value rubyFunction = context.eval("ruby", """
                ->(income, debt, score, approved, lo, hi) {
                  status = approved ? "APPROVED" : "DECLINED"
                  "Risk Report\\n" +
                  "  Income: $#{income}  Debt: $#{debt}\\n" +
                  "  Score: %.1f (95%% CI %.1f - %.1f)\\n" % [score, lo, hi] +
                  "  Decision: #{status}"
                }
                """);
        // Execute the Ruby function and return the result as a string
        return rubyFunction.execute(financialData.get("income"), financialData.get("debt"),
                score, approved, confidenceInterval[0], confidenceInterval[1]).asString();
    }
}
```

Expected output

```
Risk Report
  Income: $65000.0  Debt: $15000.0
  Score: 81.3 (95% CI 73.1 - 89.5)
  Decision: APPROVED
```

Data Exchange and Interoperability

Data Exchange Between Languages

Data exchange details.

Java	Python	JavaScript	Ruby	R
int, long	int	number	Integer	integer
double	float	number	Float	numeric
String	str	string	String	character
boolean	bool	boolean	TrueClass/FalseClass	logical
List	list	Array	Array	list
Map	dict	Object	Hash	list (named)
function	callable	function	Proc/lambda	function

Handling Objects and Collections

- **Primitives:** Automatically converted (e.g., Java `int` ↔ Python `int`).
- **Collections:** Java `List` ↔ Python `list`, Java `Map` ↔ JavaScript `Object`.
- **Functions:** Passed as `Value` and invoked with `execute()`.

Error Handling

```
try {
    Value result = context.eval("python", "1 / 0");
} catch (PolyglotException e) {
    System.err.println("Error: " + e.getMessage());
    // Handle Python ZeroDivisionError
}
```

Guest exceptions are surfaced as `PolyglotException`. You can distinguish syntax errors, guest stack traces, and host errors via its API and log full guest stack traces in production.

Security and Sandboxing

Sandboxing

GraalVM provides a comprehensive sandboxing framework for executing untrusted guest code safely. To ensure robust security when running guest code, follow these best practices:

- Always set `allowAllAccess(false)` as the default to restrict guest language access to only necessary APIs.
- Explicitly grant only the required permissions using HostAccess and language-specific options. Avoid enabling broad or blanket access unless strictly needed.
- Deny or limit file system and network access using `allowIO(false)` and related options, especially when executing code from external or untrusted sources.
- Prevent guest code from loading arbitrary Java classes by setting `allowHostClassLoading(false)`.
- Disallow native access for guest languages with `allowNativeAccess(false)`, which stops guest code from accessing unsafe native memory or loading C extensions.
- Disable thread creation inside the guest environment using `allowCreateThread(false)` to limit resource usage and potential attacks involving concurrency.
- Always validate and sanitize any input data passed into guest code. Never directly expose sensitive Java objects.
- Log and monitor guest exceptions. Capture stack traces and unexpected behavior for forensic review.
- Use code timeouts and resource limits to prevent denial-of-service attacks from long-running or resource-hungry scripts.

```
Context context = Context.newBuilder("js", "python")
    .option("js.strict", "true")       // Engine-specific flags
    .allowIO(false)                    // Deny file system access
    .allowNativeAccess(false)          // Deny unsafe memory/C-extensions
    .allowCreateThread(false)          // Deny thread spawns
    .allowHostClassLoading(false)      // Deny arbitrary Java class loading
    .build();
```

Sandbox Policies

You can choose a `SandboxPolicy` based on your use case and the acceptable level of security risk.

```
try (Context context = Context.newBuilder("js")
                              .sandbox(SandboxPolicy.TRUSTED)
                              .build();) {
    context.eval("js", "print('This guest code is entirely trusted!');");
}
```

Sandbox policies.

Policy	Use Case	Isolation Level
TRUSTED	Default; Fully trusted code, full host access	None
CONSTRAINED	Trusted but buggy code, regulated access	Shared heap
ISOLATED	Security vulnerabilities possible	Separate VM isolate
UNTRUSTED	Malicious code possible	Maximum hardening

Common Pitfalls

Common pitfalls include accidentally running with allowAllAccess(true) in production, exposing sensitive or internal objects to guest code, or forgetting to restrict I/O and network access. Always review your context configuration before deploying polyglot features to production environments.

Another common mistake developers make is failing to close contexts, which leads to leaks and causes Truffle compiler threads and native memory to accumulate over time.

Comparison with Traditional Approaches

Comparison with other approaches.

Approach	GraalVM Polyglot	JNI	JSR-223	REST API	External Process
Latency	Sub-microsecond call overhead	Low	Moderate	Network-bound (ms)	High
Type Safety	Strong via `Value` wrappers	Weak (manual marshaling)	Weak	Contract-based (schema)	None
Isolation	Configurable, in-process	None	None	Strongest	Strong
Ops Complexity	Single artifact/runtime	High (native build per platform)	Low	High (deployment, networking)	Low per-call, high orchestration
Shared Heap	Yes	No	Partial	No	No
Best Fit	Tight, high-frequency cross-language logic	Wrapping existing C/C++ libraries	Legacy Nashorn Groovy scripting	Independently scaled, team-owned services	Isolated, untrusted or legacy binaries

Enterprise Use Cases

- **Rules engines:** Business analysts write rules in Python/JS; Java hosts execution with audit logging and transactional guarantees.
- **Data science bridges:** Java Spring Boot services call R/Python statistical models without spinning up a separate model-serving process.
- **Extensible SaaS platforms:** Customer-defined scripts (pricing, workflow logic) run in a sandbox within the host application, avoiding per-tenant containers.
- **Data transformation pipelines:** Java orchestrates I/O and concurrency, while Python/JS perform specialized transformations supported by R, which provides domain-specific statistics similar to the example above.

- **Dynamic Plugin Architectures:** Expose safe Python/JS script interfaces to end users with fine-grained CPU and IO sandboxing guarantees.

When Polyglot Is (and Isn't) the Right Choice

Use GraalVM Polyglot When:

- You need low-latency, high-throughput collaboration of multiple languages in a single service.
- You want domain experts to contribute in familiar languages (Python/JS/Ruby/R) while keeping a Java-centric architecture.
- You're building microservices and want to reduce inter-service communication.
- You own the whole stack and can standardize on GraalVM as the JDK.

Avoid When:

- All functionality is already available in Java with acceptable performance.
- If languages must run in separate processes (e.g., security constraints).
- You rely heavily on native-heavy (ctypes or custom C) libraries not fully supported on GraalVM's Python/Ruby/R.

Conclusion

GraalVM's Polyglot API revolutionizes multi-language development by enabling direct, high-performance interoperability on the JVM. For enterprise applications, it offers:

- Simplified integration (no IPC, REST, or JNI).
- Performance (near-native speed after warm-up).
- Flexibility (mix and match languages for their strengths).

By combining Java's enterprise ecosystem, Python's data science tooling, Ruby's expressive syntax, and JavaScript's scripting capabilities within a single shared execution engine, developers can choose the right tool for each sub-problem without taking on microservices latency, serialization costs, or security compromises.

However, GraalVM's Polyglot Programming is not a silver bullet. Evaluate your use case against startup time, isolation requirements, and language support before adopting it. For most polyglot microservices, data pipelines, and scripting tasks, GraalVM is a game-changer.

› Back to the Table of Contents ‹



Beyond Cold Starts: Operating Restored Java Services with OpenJ9 CRIU Support

CRIU **# OpenJ9** **# Checkpoint**

Authors: Babneet Singh and Tobi Ajila

Babneet Singh is a software engineer working on runtime components in IBM Semeru Runtimes, which is based on the Eclipse OpenJ9 and OMR projects. He is a committer on both open-source projects and contributes to JVM runtime development with a focus on correctness, maintainability, and long-term sustainability. He collaborates with engineers across the JVM stack, supports low-level runtime work, and participates in open-source discussions on JVM architecture and design. His interests include JVM internals, runtime concurrency, and building reliable virtual machines for modern Java workloads.



LinkedIN

Tobi is a Java Runtime developer for the OpenJ9 VM team in Ottawa, Canada. In the past he has worked on Interpreter optimizations, JVMTI enhancments, Modularity, JSR 335 and more. Currently, his main focus is on investigating checkpoint/restore technology at the JVM level and Project Panama where he collaborates with other developers in the Valhalla expert group.



LinkedIN

Checkpoint/restore can do more than reduce startup and warm-up time. OpenJ9 CRIU support lets Java application teams use a warmed JVM as a reusable runtime baseline while retaining standard JVM capabilities for diagnostics and incident response after restore.

Checkpoint/restore is often presented as a cold-start optimization: initialize a Java service once, capture its state, then restore ready-to-run instances when demand arrives. That is valuable for elastic workloads, but it is only part of the operational story.

A restored JVM is an application resumed from a known point in its lifecycle. It may already have loaded classes, initialized its framework, populated caches, and compiled hot methods. The relevant operational questions therefore change: can an operator enable additional diagnostics for one restore? Can a warmed instance be investigated without repeating a cold start?

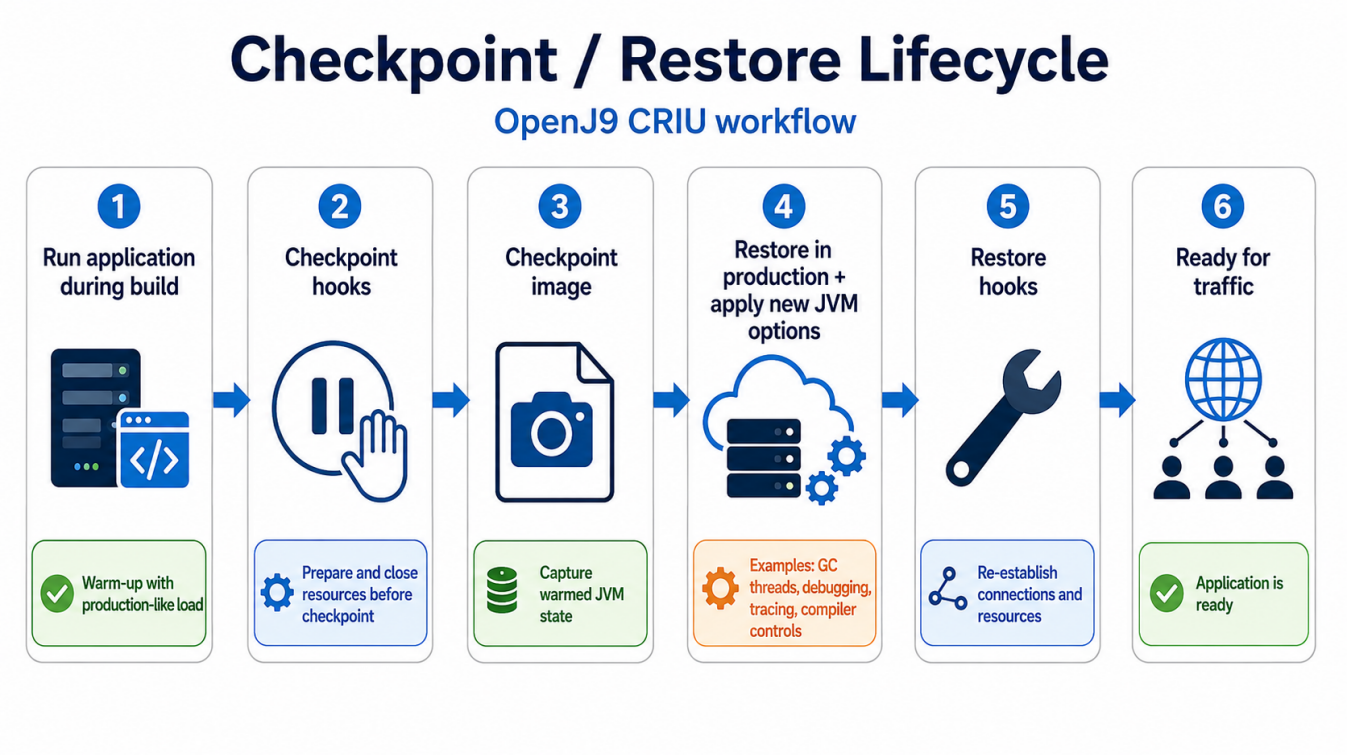
OpenJ9 CRIU support is the JVM-level checkpoint/restore capability discussed in this article. It is exposed through the `CRIUSupport` API and underpins InstantOn-style deployment workflows, including Open Liberty InstantOn. This article focuses on the OpenJ9 CRIU API and restore-time JVM behaviour. `CRIUSupport` does not turn a checkpoint image into a universally portable binary, and it does not remove the need to manage application-owned resources. Instead, it offers a more useful model: a warmed Java runtime baseline whose restore-time behaviour can still be controlled responsibly.

A Checkpoint Image Is a Baseline, Not a Finished Deployment

An image created in one environment is not automatically ready for every other environment. The restore environment might use a different diagnostics profile, container limit, shutdown model, or incident-response policy.

The useful mental model is therefore:

> **Warm once; restore with deliberate operational controls.**



The OpenJ9 CRIU lifecycle: the application is warmed up during the build and checkpointed, then restored in production, where restore-time options and hooks supply what the image itself cannot carry.

Checkpoint/restore lifecycle. During the build phase, the application is initialized and ideally exercised with production-like load. Checkpoint hooks run before the warmed JVM state is captured. In production, OpenJ9 restores the image, reads any supported restore-time options, runs the restore hooks, and resumes the application so that it can become ready for traffic.

The checkpoint image contains reusable runtime state. Restore-time inputs and restore hooks determine how that state should operate in the target environment.

Restore-Time Controls in OpenJ9

The application registers only the paths where OpenJ9 should look for restore-time inputs. The referenced files do not have to exist when the checkpoint is taken; they can be created, replaced, or updated in the target environment before the checkpoint image is restored.

The application can register:

- An **options file** containing supported restore-time JVM options.
- An **environment file** containing environment variables to add after restore. The special `OPENJ9_RESTORE_JAVA_OPTIONS` variable can also supply supported JVM options.

```
CRIUSupport criu = CRIUSupport.getCRIUSupport()
    .setImageDir(Path.of("/opt/checkpoint"))
    .registerRestoreOptionsFile(Path.of("/opt/runtime/restore.options"))
    .registerRestoreEnvFile(Path.of("/opt/runtime/restore.env"));

criu.checkpointJVM();
```

Applications do not necessarily need code changes to register these paths. OpenJ9 also recognizes system properties for the restore-options and environment files:

```
-Dopenj9.internal.criu.optionsFile=/opt/runtime/restore.options \
-Dopenj9.internal.criu.envFile=/opt/runtime/restore.env
```

Whether the paths are supplied through the `CRIUSupport` API or through system properties, they tell OpenJ9 where to look at restore time. The referenced files can be absent when the checkpoint is created and supplied later in the restore environment.

```
# /opt/runtime/restore.options
-Xshareclasses:disableOnRestore
-Xtrace:print={j9vm}

# /opt/runtime/restore.env
APP_CONFIG_FILE=/etc/myapp/region-a.properties
OPENJ9_RESTORE_JAVA_OPTIONS=-Xshareclasses:disableOnRestore -Xtrace:print={j9vm}
```

Environment-file boundary: `registerRestoreEnvFile` is additive. Environment variables captured in the checkpointed JVM are retained. Entries in the restore environment file are added only when they do not modify an existing variable, so the file is not a general environment-override mechanism.

Only a defined subset of JVM options can be supplied after restore. Options that establish fundamental VM state, such as the GC policy, compressed-reference mode, heap-region layout, or arbitrary new `-Xms` and `-Xmx` values, cannot generally be changed after that state has been captured.

Which Controls Are Useful After Restore?

Operational need	Relevant control	Restore-time behaviour and boundary
Stop further shared-class-cache use	`-Xshareclasses:disableOnRestore`	Disables further use of the shared classes cache after restore. It does not remove classes or compiled code already captured in the checkpoint image.

Operational need	Relevant control	Restore-time behaviour and boundary
Adjust signal handling	`-Xrs:onRestore` `-Xrs:syncOnRestore`	Reduces JVM signal handling after restore. Because signal handlers or related state might already be present in the image, validate the behaviour with the container runtime, health probes, diagnostics, and shutdown process.
Enable verbose GC logging	`-Xverbosegclog`	Enables verbose GC logging for the restored JVM, allowing GC activity to be observed in the target environment.
Add or modify tracing	`-Xtrace`	Applies tracing settings after restore. A trace or JIT log file configured before checkpoint must still exist and remain unchanged at restore. Method tracing might also be inconsistent for methods compiled before checkpoint.
Configure dump agents	`-Xdump`	Applies dump-agent configuration after restore. Dump events triggered by exception throws or catches cannot be enabled at this stage.
Change GC parallelism	`-Xgcthreads`	The GC-thread count captured at checkpoint is the minimum. A lower restore-time value is ignored; a higher value can increase GC parallelism for the target environment.
Adjust JIT compilation concurrency	`-XcompilationThreads`	Changes the number of JIT compilation threads used after restore, allowing compilation activity to be aligned with the CPU capacity of the target environment.
Limit JIT sampling duration	`-XsamplingExpirationTime`	Controls when the JIT sampling thread stops. After the sampling thread has been disabled, it cannot be restarted.
Report code-cache use	`-XX:+PrintCodeCache` `-XX:-PrintCodeCache`	Controls whether code-cache usage is printed when the restored JVM exits.
Refine JIT and AOT behaviour	Supported `-Xjit` and `-Xaot` parameters	Supported parameters include `count`, `limit`, `limitFile`, `exclude`, AOT loading filters, `verbose`, and `vlog`. Some affect only future compilation or AOT loading, while selected filters can invalidate compiled methods that no longer match.
Disable future AOT activity	`-Xnoaot`	Prevents future AOT compilation and loading. It does not affect compiled methods already captured in the checkpoint and does not prevent JIT compilation.
Disable future JIT compilation	`-Xnojit`	Invalidates existing compiled methods and prevents future JIT compilation. It does not prevent AOT compilation or loading.

Operational need	Relevant control	Restore-time behaviour and boundary
Debug a restored JVM	JDWP, `suspendOnRestore`, and `-XX:+DebugOnRestore`	JDWP can be enabled before checkpoint, with `suspendOnRestore` controlling suspension after restore. Alternatively, a JVM started with `-XX:+DebugOnRestore` can accept JDWP configuration through the restore-time options.

The JIT compiler cannot always be re-enabled after restore. If the JVM was checkpointed with `-Xint`, or with both JIT and AOT disabled, restore-time options do not recreate the compiler state that was absent from the checkpoint. Similarly, supplying `-Xaot` after restore does not re-enable AOT if the checkpoint was created with `-Xnoaot`.

GC and Heap Controls Selected Before Checkpoint

Some controls influence the restored JVM but cannot be newly supplied through the restore-options file. They must be selected when the checkpoint image is created.

Operational need	**Checkpoint-time control**	**Effect after restore**
Reduce GC-thread restoration overhead	`-XX:CheckpointGCThreads=<number>`	Reduces the number of GC threads retained when the checkpoint image is created. At restore, OpenJ9 can increase the count using a restore-time `-Xgcthreads` value or the default appropriate for the target environment, but it cannot reduce the count below the checkpoint value.
Adapt heap limits to available memory	`-XdynamicHeapAdjustment`	Allows OpenJ9 to adjust the effective minimum and maximum heap sizes so that they remain within the physical-memory limits of the checkpoint and restore environments. The option must be enabled at checkpoint and applies to both checkpoint and restore.

Without dynamic heap adjustment, the heap size and heap regions established at checkpoint remain part of the restored JVM state. For example, an image created in a container without a memory limit does not automatically adopt a smaller limit when it is restored in another container.

`-XdynamicHeapAdjustment` provides an opt-in mechanism for checkpoint images that might be restored into environments with different memory capacities. It does not turn restore into a second JVM startup or permit arbitrary replacement of every heap and GC setting.

This distinction is important: restore-time configuration is a controlled adjustment of an already-running JVM, not a second JVM command line.

Debugging a Restored JVM

A fault might appear only after the application is warm, a cache has been populated, or a particular restore environment is active. OpenJ9 allows a debugger to be attached after the checkpoint image is restored.

One approach is to configure JDWP before checkpoint and use the OpenJ9-specific `suspendOnRestore` parameter to control whether the restored application waits for the debugger:

```
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=127.0.0.1:5005,suspendOnRestore=
    y
```

Here, `suspend=n` allows the application to run normally before checkpoint, while `suspendOnRestore=y` suspends it after restore until the debugger is connected and ready.

Alternatively, when the JVM is started with `-XX:+DebugOnRestore`, the JDWP configuration can be supplied through the restore-time options so that debugging is enabled for the restored JVM.

After restore, the debugger is attached through the standard Java JDWP workflow. As with any JDWP configuration, expose the debug connection only in an appropriately secured environment.

Conclusion: Fast Startup with Operational Control

Checkpoint/restore changes the unit of deployment. Instead of deploying only application code and a cold JVM, teams can deploy a warmed runtime baseline together with controlled restore-time configuration.

OpenJ9 CRIU support is therefore not only about reaching readiness sooner. It retains selected JVM tuning, diagnostic, and debugging capabilities when a warmed Java process becomes a reusable deployment artifact.

References

1. OpenJ9 CRIU support
2. OpenJ9 JDK 26 CRIUSupport API
3. OpenJ9 -Xrunjdwp and suspendOnRestore
4. OpenJ9 compiler support for Debug On Restore (Link)
5. OpenJ9 -XX:CheckpointGCThreads
6. OpenJ9 -XdynamicHeapAdjustment

› **Back to the Table of Contents** ‹

For your IT projects you don't need a know-it-all.

You need a {BUDDY}.



Richard Fichter
CEO @ XDEV

Java Champions

Outdated software? Rising maintenance costs? Security risks? We help you to make Java applications fit for the future - with a clear concept and at eye level. In addition to modern tools, we offer premium support for your **Java modernization.**

- **Not a know-it-all - a buddy:** We support your team with pragmatic methods without playing the wise guy.
- **Modernization with strategy:** agile methods & proof of concept for a secure update.
- **Robust solutions:** We rely on proven technologies and practical concepts - without unnecessary complexity.

Let us move your Java project forward together!

trusted by

ATU · EMPORIO ARMANI · CONRAD · Fraport · Red Bull · Allianz

BEST {BUDDYS} IN CODE

Arrange a free discovery call here!

www.xdev.software

XDEV