

17 **ACHIEVING MICROSECOND
LATENCIES WITH JAVA**

52 **JAVA 26 +
SPRING BOOT 4**

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

Edition 2 / 2026

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY #JAVAPRO

Engineering Intelligence Building Systems in the AI Age

12 **BUILDING LOW-LATENCY EVENT PIPELINES
WITH JAVA 25**

43 **FROM TABLES TO JSON: HANDLING UNSTRUCTURED
DATA IN RDBMS SYSTEMS WITH SPRING BOOT**

61 **FROM CRON TO CARBON & COST-AWARE:
MODERNIZING JAVA BACKGROUND JOBS**

72 **THE JAKARTA CLIFF:
WHY MODERN JAVA LEAVES STRUTS 1 BEHIND**

76 **AUTOMATING JVM THREAD DUMP
ANALYSIS WITH AI**

83 **ENTERPRISE-GRADE AI FOR JAVA DEVELOPERS:
BEHIND THE BAR OF THE AI REVOLUTION**

136 **ANCHORED IN THE CHAOS:
MANAGING COGNITIVE LOAD, AI UNCERTAINTY,
AND WELL-BEING AS A DEVELOPER**



JCON
EUROPE
COLOGNE
2026

April 20 - 23, 2026
www.jcon.one



SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

The Great Shift: From Syntax to Semantics

Since the first Java magazines reached developers in the mid-1990s, our industry has undergone repeated tectonic shifts: the rise of Java EE, the emergence of Spring's dependency injection, the microservices movement, and the transition to cloud-native architectures. Each wave introduced new abstractions to manage growing complexity. Yet these tools remained instruments—we translated human intent into machine-executable code. We were the authors; the compiler was our editor.

Today, the situation is fundamentally different. We have entered the AI Age. Artificial Intelligence is no longer confined to research; it shapes our daily workflow. It operates within our IDEs, generates boilerplate, reviews pull requests, and increasingly participates at runtime. Amid the enthusiasm for “AI-driven development,” a more demanding question emerges: What does it mean to build systems when writing code is no longer the constraint? This issue explores the answer: the rise of Engineering Intelligence.

The Productivity Illusion: Why Code Is Cheap, but Judgment Is Expensive

There is a persistent misconception that faster code generation automatically results in better systems. In practice, the opposite is often true. A Large Language Model can produce hundreds of lines of syntactically correct Java within seconds—implementing a REST controller or data mapping layer with remarkable speed. Yet it does not understand the system. It detects patterns, but it cannot reason about intent. It cannot evaluate trade-offs such as low latency versus eventual consistency. It generates output, but it cannot ensure that this output is robust, maintainable, or secure within the constraints of a legacy enterprise environment.

As code generation accelerates, the bottleneck in software engineering shifts. Writing is no longer the constraint—judgment is. Critical evaluation, architectural integration, and long-term ownership now determine whether a system succeeds or accumulates catastrophic technical debt. Generated code is not a solution; it is a liability. It becomes part of a complex system that must remain coherent for years. This is where true engineering begins. And when the system fails under peak load, responsibility still rests with us—the developers and architects.

Architecture as a Stability Mechanism in the Storm

In an era where code can be summoned with a prompt, software architecture becomes more critical than ever. Architecture must act as a living system of constraints that stabilizes applications while implementation speed increases. Whether building ultra-low-latency services or handling unstructured data in relational systems with Spring Boot, structural decisions determine longevity.

Modern Java systems are distributed, layered, and interconnected. If every developer—or every AI—contributes code without alignment, architectural entropy emerges. This form of debt is more dangerous than isolated defects because it raises

the cognitive load for everyone who maintains the system. Discipline becomes essential. If AI accelerates production, architecture must introduce the necessary friction to preserve coherence. We must move from “Move Fast and Break Things” to “Move Fast with Stable Structures.”

Testing as a Strategy for Continuity

Acceleration without safeguards leads to instability. This is why testing must be reconsidered. In the AI Age, tests are not merely verification artifacts—they are continuity mechanisms. They define what must remain invariant while implementations evolve.

When machines draft code, validation must focus on externally observable behavior, not internal structure. As we approach Java 26 and Spring Boot 4, production systems demand more than certification-level knowledge; they require resilient test strategies that secure long-term evolution. If implementations can be generated rapidly, the decisive capability becomes confident refactoring. High velocity is only sustainable when protected by durable, behavior-driven safeguards.

Observability and the Human Constraint

Complex systems fail in non-linear ways. Distributed services under load exhibit emergent behavior that static code cannot reveal. If we are to remain accountable for the systems we design, runtime transparency must be intentional. In this issue, we explore automated JVM thread dump analysis with AI—an example of applying *AI for the engineer* to achieve actionable observability on platforms such as Amazon ECS and EKS.

At the same time, we must confront human cognitive limits. Modern development demands navigating expanding stacks and constant information flow. Structural inconsistency and weak conventions act as attention drains. Engineering intelligence therefore includes designing for human clarity—actively managing cognitive load and developer well-being. Reducing unnecessary complexity is not merely stylistic; it is a strategic architectural choice and an act of responsibility toward our colleagues and our future selves.

Conclusion: Sharpening the Craft

The AI Age does not replace software engineering—it intensifies its demands. As routine tasks fade, what remains are the essential challenges of system design. In this issue, we examine these dimensions across the Java ecosystem: low-latency event pipelines with Java 25, microsecond architectures, the debate around Struts 1.3 in 2026 and the “Jakarta Cliff,” carbon-aware background processing, and the creative convergence of developers and designers in “Vibe Coding.”

These are not isolated themes. They converge on a central question: How do we build systems that remain reliable, comprehensible, and maintainable when generating code is no longer the constraint? This is the defining challenge of our time. It is also an opportunity—our role evolves from writing code to designing systems and curating intent.

Welcome to **Engineering Intelligence!**

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 RapidEclipse 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

JAVAPRO

JAVAPRO

Engineering Intelligence Live: JAVAPRO Benefits for JCON 2026	9
A Conference Where the Java Community Actually Connects	10
NEW: Premium Track AI — AI Done Properly	10
Workshops: Hands-On Start on Monday	10
Free Tickets for JAVAPRO Readers	11
Engineering Intelligence: From the Page to the Stage	11

Core Java

Bhupendra Mishra

Building Low-Latency Event Pipelines with Java 25	12
The Challenges of Event-Driven Systems	13
Virtual Threads for Efficient Event Processing	13
Structured Concurrency for Coordinated Event Workflows	14
Improved Garbage Collection	14
Improved Event Modeling and Handling	14
Safer Event Pipelines with Records and Sealed Types	15
Simplifying Event Driven System Implementation	15
Readability, Maintainability, and Performance Benefits	15
Practical Use in Event-Driven Systems	15
Conclusion	16

Core Java

Stefan Angelov

Achieving Microsecond Latencies with Java: Practical Techniques for Building Ultra-Fast Systems	17
How the JIT Compiler Enables Microsecond Latencies	18
Off-Heap Memory - Bypassing the Garbage Collector	20
Why Single-Threaded Shards Beat Multi-Threading	22
Object Pooling - Zero Allocations, Zero GC Pauses	24
Conclusion: The “Zero” Philosophy	24

API and Frameworks

Buhake Sindi

API Development: Implementing an HTTP POST/SSE Event in Jakarta REST	26
What Is Server-Sent Events (SSE)?	27
Server-Sent Events API in Jakarta REST	28
Consuming HTTP POST SSE Streams Using a Modern Java JDK HttpClient.	33
Conclusion	35
Links/Resources	35

API and Frameworks

Robetti Rogerio

Open J Proxy: A Modern, Open-Source Approach to Database Connectivity for the Java Ecosystem	36
Historical Context on JDBC	37
The Underlying Problem: Modern Systems, Ancient Assumptions	37
Why Existing Connection Pools Cannot Solve This	38
The OJP JDBC Driver — A Type 3 JDBC Driver	39
Bringing It All Together	40
Backpressure: The Safety Valve Databases Have Needed for Years	40
Bulkheads and Slow Query Segregation: Architectural Resilience for SQL Workloads	41
A Unified, Multi-Database Access Layer	41
Performance Effects: Stability, Not Just Speed	41
Integration That Feels Almost Too Simple	41

The Broader Impact of OJP	42
Final Thoughts	42

API and Frameworks Kush Sharma

From Tables to JSON: Handling Unstructured Data in RDBMS Systems with Spring Boot	43
Let's CODE JSON in POSTGRESQL!	44

API and Frameworks Vauban Vincent

Java 26 + Spring Boot 4: From Certification Skills to Production-Ready Apps	52
“Freestyle” Learning vs. Guided Learning	53
Certification as a Map: You Don't Conquer Territory Without One	53
The Syllabus Is Not “Tricks”: It Mirrors Business Needs	54
The Inevitable Lag: Certifications Trail Releases, and That's Normal	54
Domain-by-Domain: From Exam Skill to Production-Ready Microservice	55
Where Java 26 Helps Spring Boot 4 Microservices	58
Putting It Together: The “Foundation + Delta” Learning Loop	59
Why This Changes Consulting Confidence (and Team Trust)	59
Closing: Client Satisfaction Is Still the Ultimate Metric	59

API and Frameworks Ronald Dehuysser

From Cron to Carbon & Cost-Aware: Modernizing Java Background Jobs	61
Introduction	62
Planning One-Time and Cron Jobs	62
From Cron to Carbon Aware Job Processing	63
Keeping It Cool if Things Go Wrong	64
Scaling Up with Ease	65
Conclusion	66
References	66
Abstract	67

API and Frameworks Steve Poole

Why Struts 1.3 Still Matters in 2026	68
Rediscovery Day: When Struts Finds You	69
A Framework Out of Time	69
Tooling Moves On, Even If the Code Doesn't	69
The Platform Marches Forward, With or Without You	70
The Modernisation Myth: It's Not Instant, It's Not Easy	70
Buying Time: The Unsung Part of Modernisation	70
Walking the Corridor, Not the Cliff Edge	70

API and Frameworks Steve Poole

The Jakarta Cliff: Why Modern Java Leaves Struts 1 Behind	72
When an Upgrade Isn't Really an Upgrade	73
The Namespace Shift That Changed Everything	73
Where Struts 1 and Modern Containers Diverge	73
The Breaking Point Usually Isn't Technical	74
Why There's No “Compatibility Layer” That Fixes This	74
Living Safely on the Older Side of the Divide	74
Modernisation Doesn't Have to Be Sudden	74
Older Software and Modern Nightmares	74
Buying Time Is Sometimes the Smartest Move	75

The Cliff Is Real, But It Doesn't Have to Be Steep	75
--	----

AI and ML Sascha Möllering

Automating JVM Thread Dump Analysis with AI: Practical Observability for Java on Amazon ECS and EKS	76
The Challenge of Troubleshooting Java Inside Containers	77
Overview of the Automated Analysis Architecture	77
Deploying the Base Infrastructure	78
Configuring Observability in a Java Application	78
Collecting Thread Dumps Automatically	79
AI-Powered Analysis of Thread Dumps	79
Example workflow and analysis	80
Cleaning Up the Environment	82
Conclusion	82

AI and ML Maarten Vandepierre & Camille Nigon

Enterprise-Grade AI for Java Developers	83
The Three Friends at the Bar: Understanding AI Architectures	84
Why AI Projects Stumble: The World Around the Model	85
Nothing Works Without Data: The Bartender Needs Orders	85
An AI System Needs a Well Run Bar: Platform Engineering	86
Where Should the Model Live: The Importance of Optionality	86
The Real Secret: It Is Not About the Drink, It Is About the Bar	86

IDE and Tools Tim Schraepen

Data-Driven Projections with Kotlin Notebooks	88
Why We Need Better Projections	89
Overview	89
Monte Carlo: You Only Need In-Production Data to Get Started	89
What Monte Carlo Is Good For	90
Why this Matters	91
Common Questions	91
Goodhart's Law	91
Practice Time!	92
Aging Charts: What They Are & How to Build Them	92
Why this Matters	93
More Practice Time!	93
Bringing It Together: From Data to Insight to Action	93
Why Kotlin Notebooks?	94
Tips & Pitfalls from Experience	95
Conclusion	96
Resources	96
Getting help	96

Project Management Monschau Andreas

Avoiding Unsustainable Responsibility	97
Technical Debt	98
Poor Team Atmosphere	98
Poor Onboarding	99
Organizational Problems	99
Unclear Responsibilities	99

Project Management

Andres Sacco

Where Is the Logic of My Application?	100
The Problem: Inconsistency Across Applications	101
Solution Part 1: Defining an Architectural Guide	101
Solution Part 2: Validating Architecture Automatically	102
What's next?	109
Conclusion	109

Web Development

Csilla Szántó

When Developers Meet Designers: Lessons from Vibe Coding and the Art of Creative Expression	110
What Is Creative Coding?	111
Why Should a Software Engineer Care?	112
Implementing Tools	112
How to Do Creative Coding?	113
Comparing the Results	114
Interpreting the Results	115
Is AI Replacing Professionals?	115
Combining Approaches: Giving You Power	115
A Final Thought	116

JVM Language

Haoyi Li

Scripting on the JVM with Java, Scala, and Kotlin	117
Example Java Single-File Script	118
Benefits of Scripting on the JVM	119
Challenges of Small-Scale Java Programs	120
Improving the JVM Scripting Experience	122
Parallelizing Our Scripts	126
Conclusion	127

Testing and Quality

Wouter Bauweraerts

From Fragile to Fearless	128
Different Types of Tests	129
What Is the Unit in a Unit Test?	130
Sociable Test or Integration Test	131
What Is a Robust Unit Test?	132
Going even Further	133
Wrap It Up!	135

Human Factors

Simon Verhoeven

Anchored in the Chaos: Managing Cognitive Load, AI Uncertainty, and Well-Being as a Developer	136
The Growing Pressure on Developers	137
Cognitive Overload	137
Navigating Uncertainty Around AI	139
Protecting Personal Well-Being as a Necessity	141
Conclusion: Building an Anchor in the Chaos	142
Helpful Resources	143

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors



JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92637 Erbenhof
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 Erbenhof
Germany

Copyright (c) 2025
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.



International
Java Community Conference
JCON2026 www.jcon.one

Engineering Intelligence Live: JAVAPRO Benefits for JCON 2026

JAVAPRO # JCON # AI # Tickets

Author: JAVAPRO

JAVAPRO is the world's first free print magazine for Java, now also available as a free eBook. JAVAPRO is a project of the Java community for the Java community. At JAVAPRO, renowned authors and speakers from the international Java community write about the latest developments in Java, trends, tools, and best practices for professional application development with Java. In addition to articles, we regularly publish webinars, training sessions and videos on YouTube – for developers who want to share their expertise as well as for those looking to deepen their technical skills. Contributions from the community are always welcome – becoming an author is easy via javapro.io/become-an-author. JAVAPRO was founded in 2017 as a free print and online magazine in Germany. Today, we reach developers worldwide through javapro.io and actively support the community as the organizer of the Java conference JCON.



LinkedIn

JCON EUROPE 2026 is coming back to Cologne — and this year brings two major highlights right from the start: a brand-new **Premium Track AI focused on production-ready AI for Java teams**, and **free conference tickets for JAVAPRO readers and authors**.

From **April 20–23, 2026**, the European Java community will once again gather at the **Cinedom multiplex cinema**, turning one of Germany's largest movie theaters into a four-day Java experience on the big screen.

A Conference Where the Java Community Actually Connects

JCON is not designed as a classic “sit-and-listen” conference. The event is built around real interaction: developers, architects, and speakers meeting face-to-face, exchanging practical knowledge, and creating connections that last beyond the sessions.

With more than **1,000 participants expected**, JCON EUROPE 2026 is one of the most community-driven Java gatherings in Europe.

The format encourages exchange through:

- spontaneous hallway-track discussions
- **1:1 meetings with speakers**, bookable for personal Q&A
- a dedicated **Mentorship Hub** across all conference days (including Bruno Souza and others)
- Slido-powered live Q&A during sessions
- expo area, demos, and hands-on conversations

Happy Hour events on Tuesday and Wednesday add even more space for informal networking, alongside an exclusive VIP gathering on Wednesday evening.

NEW: Premium Track AI — AI Done Properly

AI is everywhere — but integrating it responsibly into real-world Java systems remains a complex challenge.

This is why JCON EUROPE 2026 is introducing the new **Premium Track AI**. It is designed for developers who want to move beyond experimental prototypes and focus on maintainable, enterprise-grade solutions — **true Engineering Intelligence for production environments**.

The Premium Track runs on:

- **Tuesday & Wednesday (April 21–22)**
- Cinema 7
- Accessible via Premium/VIP tickets or upgrade options

Featured sessions include:

- *Seven Habits of Highly Effective AI Java Coding* — Jonathan Vila
- *Green AI with JobRunr* — Ronald Dehuysser
- *Building Intelligent Enterprises with LangChain4j & CDI* — Emily Jiang

A **20% discount** for [selected tickets](#) is available with:

Voucher Code: JAVAPRO-EARLYBIRD

Workshops: Hands-On Start on Monday

The conference week begins with a dedicated Workshop Day on **Monday, April 20**, starting at 13:00.

All attendees can [book expert-led workshops](#) for just:

€19 incl. VAT per workshop
(seats are limited)

Free Tickets for JAVAPRO Readers

JCON EUROPE 2026 remains strongly community-first: participation is free for many members of the Java ecosystem, including:

- JAVAPRO subscribers, authors, and JCON alumni
- Java User Group members

JAVAPRO readers [can book](#) a free **3-Day Full Conference Pass** for the main conference program using:

Ticket Code: JAVAPRO

Please note: workshops and Premium Track AI access require an additional upgrade.

For commercial company participants, regular pricing applies:

- Full Conference Pass (3 Days): **699 EUR** (excl. VAT)
- or [use](#) the 20% Voucher Code: JAVAPRO-EARLYBIRD

Engineering Intelligence: From the Page to the Stage

The core themes of our current issue—**Engineering Intelligence**—find their practical continuation at JCON EUROPE 2026. While the magazine lays the strategic and theoretical foundation for intelligent software systems, the conference in Cologne provides the space to deepen these concepts through live coding and direct exchange with industry experts.

For all readers who want to experience **Engineering Intelligence** beyond the printed page and see it in action within the Java ecosystem, JCON in April 2026 is the must-attend event on the calendar. We look forward to welcoming the JAVAPRO community at the Cinedom.

More information: www.jcon.one Tickets: <https://2026.europe.jcon.one/tickets>

› [Back to the Table of Contents](#) ‹

APR
20-23



www.jcon.one

JCON2026

JCON EUROPE 2026
International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



Building Low-Latency Event Pipelines with Java 25

CoreJava

EventDriven

Author: Bhupendra Mishra

Bhupendra is a seasoned software engineer with nearly 20 years of experience building scalable systems across industries. He's led mobile development, system architecture, and microservice transformations. Outside of work, he's a passionate marathon runner who values balance and brings focus and endurance to both tech and life.



LinkedIn

The modern distributed system relies heavily on event driven system due to obvious reasons. They are better scalable and fault tolerant compared to monoliths. Whether it is real-time analytics or data streaming, these systems work asynchronously and react to events in near real time. However these systems brings their own set of challenges. First of all implementing an efficient event driven pipeline in not trivial. These systems must handle unpredictable traffic spikes, high concurrency and strict SLAs.

While Java has been a preferred choice to develop stable backend systems, event driven architectures has been relied on third party frameworks and complex asynchronous programming models. Java 25's JVM capabilities allows event driven programming to be the first class citizen. Virtual Threads and Structured Concurrency enables developers to build highly concurrent, low-latency event-driven systems using simple and readable code. Improvements in startup performance and garbage collection pushes the envelope further.

This article explores how Java 25 helps implement better event-driven systems. It is a major step forward for a lot of legacy systems developed using earlier versions of Java.

The Challenges of Event-Driven Systems

Event driven systems are designed to handle large amounts of events efficiently. The source of these events could be kafka, jms, kinesis or event buses. These systems are expected to react to these events in real time. They could be performing validation, enrichment, aggregation and/or routing to other systems. Common challenges with event driven systems include:

- Handling event spike on predictable or unpredictable basis
- Managing blocking I/O without wasting threads
- Coordinating multiple downstream calls per event
- Keeping latency predictable under load
- Maintaining readable and maintainable code

Traditional Java concurrency models relied on a limited number of platform threads, making it expensive to block on I/O. This left teams with two choices, either use limited OS threads to develop somewhat lower latency systems. Or adopt a external reactive programming framework, which often introduced additional complexity. Java 25 changes this equation.

Virtual Threads for Efficient Event Processing

Unlike platform threads, virtual threads are not tied to a specific OS thread. They are lightweight threads managed by the JVM. When a virtual thread performs a blocking I/O operation, the Java runtime suspends the virtual thread and releases the underlying carrier thread to execute other work. This mechanism allows the JVM to efficiently support millions of concurrent threads. In a typical event driven system it could be:

- Consuming an event from a stream
- Calling one or more external REST endpoints
- Writing results to a database or another stream

This is where virtual threads stand out. Virtual Threads enable each event to be processed using a simple, synchronous programming model. Now expensive and limited OS threads are used efficiently. This enables:

- Higher throughput with fewer resources
- Simpler code compared to callback-based or reactive styles
- Better utilization of CPU in I/O-heavy workloads

Virtual Threads significantly reduce the operational cost of concurrency. The result, a higher throughput event processing system.

```
1 // Start a virtual thread to run a task.
2 Thread thread = Thread.ofVirtual().start(runnable);
3
4 // A ThreadFactory that creates virtual threads
5 ThreadFactory factory = Thread.ofVirtual().factory();
```

Structured Concurrency for Coordinated Event Workflows

Parallel processing is quite common for every workflow, let alone an event's lifecycle. For example, an event processor may need to call multiple services or aggregate data from different sources during processing of an event.

A group of related tasks running in different virtual threads can be considered as a single unit of work using structured concurrency. This enhances error handling mechanisms and improves reliability and observability of the system. Key benefits of structured concurrency include:

- Automatic cancellation of related tasks on failure
- Clear scoping of concurrent operations
- Improved error propagation and handling

In case of an event failure, the system can respond consistently without leaking threads or leaving partial work behind. This leads to predictable orchestration logic and more robust processing which is essential for high-throughput systems.

```

1  try (var scope = StructuredTaskScope.open ()) {
2
3      Subtask<String> subtask1 = scope.fork(() -> query (left));
4      Subtask<Integer> subtask2 = scope.fork(() -> query (right)) ;
5
6      // throws if either subtask fails
7      scope.join ();
8
9      // both subtasks completed successfully
10     return new MyResult (subtask1.get(), subtask2.get()) ;
11 } // close

```

Improved Garbage Collection

For most real time systems, low latency is a very important requirement. Higher latencies can cause anything from poor user experience to unpredictable workflow behaviors.

Java has been working consistently to improve garbage collection which is often seen as a double edge sword. Java 25 introduces mature pauseless and near pauseless garbage collectors. These garbage collection technology allows java to compete with low latency languages like C++ and Rust.

ZGC as a low latency garbage collector was introduced in Java 11, is Generational ZGC in Java 25. It archives significant performance boost by implementing separate young and old generation. It offers pause time under 1 millisecond irrespective of heap size. Event-driven systems often create many short-lived objects, making GC behavior critical. Java 25's improvements help:

- Reduce latency spikes caused by GC pauses
- Maintain consistent throughput under load
- Optimize memory usage

These improvements help design and maintain more stable event pipelines.

Improved Event Modeling and Handling

Event modeling and processing is another integral part of event driven systems which benefits from various Java 25 enhancements. Best event-driven architectures should rely on polymorphic event types flowing through pipelines. Typically handling these events/objects in Java required verbose instanceof checks, explicit casting, and defensive programming. Advance pattern matching with Java 25's changes this dynamic.

Enhanced pattern matching, particularly when used with switch expressions, allows event handlers to clearly and safely dispatch logic based on the concrete event type. Intent can be expressed declaratively, instead of long conditional chains. This allows mapping each event type to its corresponding behavior. As a result, event-processing code is cleaner, readable and easier to evolve as new event types are introduced.

Safer Event Pipelines with Records and Sealed Types

Events are commonly modeled as immutable data transfer objects. Java 25's record patterns records a natural fit. Record patterns allow handlers to deconstruct events directly within a switch. Required data can be accessed without boilerplate getters. This is especially effective when working with structured event payloads coming from Kafka, cloud event buses, or APIs.

Another best companion event received in Java 25 is Sealed Type. Pattern matching becomes more powerful when combined with sealed interfaces or classes. By declaring a sealed event hierarchy, the compiler is aware of all possible event variants. This ensures event handlers are explicitly handling every case at compile time. This completely removes an entire class of runtime bugs caused by unhandled events. In high-volume, distributed systems, this level of safety is invaluable.

Simplifying Event Driven System Implementation

These language improvements are not just syntactic sugar. They enable teams to move faster and implement better event driven systems without relying on third party libraries. Teams no longer need to choose between a limited java feature set and third party libraries which may bring another level of complexity.

Blocking I/O becomes safe and scalable and parallel event processing is better structured with virtual threads and structured concurrency.

With records and advanced pattern matching, clear event dispatch logic improves maintainability in systems. Event schemas evolve frequently and multiple teams contribute to the same pipelines without causing conflicts. Whether its diagnosing production issues or extending domain logic, teams can reason about event flows more easily.

This makes Java 25 especially appealing for teams building complex event-driven systems where clarity and correctness matter as much as performance.

Readability, Maintainability, and Performance Benefits

These language improvements are not just syntactic sugar. Clear event dispatch logic improves maintainability in systems where event schemas evolve frequently and multiple teams contribute to the same pipelines. Developers can reason about event flows more easily, which is critical when diagnosing production issues or extending domain logic.

When pattern matching is combined with sealed classes, it results in an optimized java compiler. While event-driven systems are often I/O-bound, reducing overhead in the critical event processing path helps maintain predictable latency under load.

Practical Use in Event-Driven Systems

In real-world event-driven systems, Java 25's features come together most effectively when applied end to end. Assuming a kafka centric system, consumers typically process large volumes of immutable events. Then application performs validation, I/O heavy data enrichment and eventually producing new events to downstream topic. This flow can be implemented with java native expressive and efficient code using Java 25.

Kafka consumers can read messages as immutable records. When message types are organized into a sealed event hierarchy, teams explicitly define all supported message types within a topic. Workflow becomes more concise and type safe using Pattern matching in switch expressions based on the event type. For example OrderCreated and PaymentFailed events are handled by respective handlers without instanceof check or manual casting. The compiler enforces exhaustiveness which ensures that new event types are handled explicitly rather than failing silently at runtime.

Once a record is deserialized and classified, Virtual Threads can be used for each Kafka record to be processed independently. With a simpler synchronous model any blocking operations e.g. calling downstream services, accessing databases, or writing to external systems is handled efficiently. The JVM automatically suspends virtual threads during blocking I/O and frees up platform threads to continue processing other records. This enables high consumer throughput and efficient scaling irrespective of traffic spikes on kafka partitions.

Typical parallel work required while processing a record is simplified using Structured Concurrency. For e.g. enrichment calls to multiple services or validations against different data sources can be executed concurrently and treated as a single unit

of work. If any step fails, remaining tasks are cancelled automatically and partial processing is prevented. This structured approach makes it easier to reason about success, failure, and retries which are essential for reliable Kafka consumers.

Kafka workflow typically generates many short-lived objects during deserialization, transformation, and serialization. Java 25's garbage collection improvements help keep memory footprint to optimal. Modern garbage collectors efficiently reclaim short-lived objects while avoiding long pauses. This keeps event processing smooth and prevents latency spikes under load.

With all these features, teams can build java native event driven systems which are scalable, maintainable and performant. Java 25 makes it possible to write clear domain-oriented code while meeting the demanding performance and concurrency requirements of modern streaming architectures.

Conclusion

Event-driven systems thrive on concurrency and responsiveness. With Java 25, these qualities are a core part of Java. Lightweight virtual threads with safer parallel execution and improved runtime performance ensure application provides low latency. Code is becoming more readable and maintainable with sealed and patterns. Overall performance of applications gets better when the system is designed and developed with the latest Java features. No matter which application is being developed, streaming platform or asynchronous microservices, Java 25 provides the tools needed to implement modern event-driven systems efficiently and confidently.

› [Back to the Table of Contents](#) ‹



Achieving Microsecond Latencies with Java: Practical Techniques for Building Ultra-Fast Systems

CoreJava

LowLatency

Author: Stefan Angelov

Stefan Angelov is an Engineering Manager at Tradu, where he leads teams building high-performance trading systems. With over a decade of experience in the Java ecosystem, Stefan has architected low-latency solutions for some of the most demanding fintech platforms, including Nexo and Trading 212, where he designed exchange and trading infrastructure processing massive volumes of real-time market data. Beyond his corporate work, Stefan is passionate about education and community building. He is the founder of CodeBefore.eu and co-founder of Hacker4e.com—an academy that teaches children programming through games, Scratch, and web technologies.



LinkedIn

Four years ago, I joined a crypto trading company. First week, I'm sitting in a meeting and someone casually mentions we need to process "north of 10 gigabytes per second of market data." I'm nodding along like this is normal. It's not normal.

Our Java application was a disaster. Memory usage looked like a hockey stick. Latency? All over the map—2ms one second, 50ms the next when the GC decided to wake up. In crypto trading, 50ms might as well be 50 years. By the time your order hits the exchange, someone else already took the price.

Then management drops the bomb: "**Let's build our own exchange.**"

That conversation broke my brain. Everything I thought I knew about writing Java—immutable objects, dependency injection, clean abstractions—suddenly didn't matter. What mattered was keeping the garbage collector from wrecking your latency at the worst possible moment.

There's no magic to solving this. We had to realize that the JIT compiler is essentially a black box that makes you look like a genius if you feed it right. But it's fickle—one tiny change in your method size can push you over the inlining threshold and suddenly your 2ns hot path is now 20ns. You learn to work with it, not against it.

This isn't theoretical. The open-source **exchange-core** project proves every technique in this article works in production, maintaining sub-microsecond latency for the critical path even under massive load. The codebase is available on GitHub for you to study, benchmark, and learn from.

Here's what worked for us: complete code examples from production trading systems based on actual benchmarks, not theoretical projections. We had to stop writing "Good Java" and start writing "Hardware-Aware Java."

How the JIT Compiler Enables Microsecond Latencies

Most developers think Java is interpreted bytecode. Maybe in 1995. Not anymore.

The JIT compiler is constantly watching your code run. It profiles everything. After about 10,000 invocations of a hot method—which happens in milliseconds for our trading loops—it compiles that method with C2, the aggressive optimizing compiler. At that point, you're running native machine code that's been optimized for your exact workload.

Here's what the JIT knows that you don't: Which branches you actually take. Not the ones you could theoretically take. The ones you do take, 99.9% of the time. It optimizes for that. It also knows which objects never escape a method scope. If it can prove that 'Order temp = new Order()' never leaves the method, it doesn't allocate. Just eliminates the whole thing. Scalar replacement. Suddenly your code that looks like it's creating objects is just shuffling primitives on the stack.

But the JIT is fickle. I spent two days debugging why our order validation suddenly got 10x slower after I extracted a helper method. Turns out I crossed the `MaxInlineSize` threshold. The JIT won't inline methods bigger than 35 bytes of bytecode by default. My "clean" refactoring added 8 bytes of bytecode to the caller, pushed it over 35, killed inlining. 10x slower. **I reverted the refactoring.**

The Optimization Tiers:

Code Execution Journey:

```
1 Code Execution Journey: _____
2
3 0-1,500 calls:      Interpreter (collecting profiling data)
4 1,500-10,000:      C1 Compiler (fast compilation, basic optimizations)
5 10,000+:           C2 Compiler (aggressive optimization based on profiles)
6
7 Result: After 10,000 invocations (milliseconds!), code runs at
8               full native speed with all optimizations applied.
```

In production systems, hot paths hit 10,000 invocations within seconds of startup. After that, they're **fully optimized for the lifetime of the process**. This is why HFT systems always warm up before live trading.

Method Inlining: The Mother of All Optimizations

The JIT's most impactful optimization is method inlining—literally copying method bodies into callers:

Before C2 Optimization:

```

1 public boolean validateOrder(Order order) {
2     return validatePrice(order) &&      // Method call
3         validateQuantity(order) &&      // Method call
4         validateSymbol(order);         // Method call
5 }
6
7 private boolean validatePrice(Order order) {
8     return order.price > 0 && order.price < MAX_PRICE;
9 }
10
11 private boolean validateQuantity(Order order) {
12     return order.quantity > 0 && order.quantity < MAX_QUANTITY;
13 }
14
15 private boolean validateSymbol(Order order) {
16     return order.symbol != null && order.symbol.length() > 0;
17 }

```

After C2 Optimization (what actually executes)

```

1 public boolean validateOrder(Order order) {
2     // All method calls inlined - direct field access!
3     return (order.price > 0 && order.price < MAX_PRICE) &&
4         (order.quantity > 0 && order.quantity < MAX_QUANTITY) &&
5         (order.symbol != null && order.symbol.length() > 0);
6 }

```

Performance:

- * Before: 15-20 ns (three method calls, stack frames, argument passing)
- * After: 2-3 ns (direct field access, zero call overhead)

Improvement: 7-10x faster. The JIT **knows** the actual call sites from profiling data and can inline aggressively. It doesn't need to be conservative about virtual methods or separate compilation units—it has complete runtime visibility.

Escape Analysis: Eliminating Allocations

The JIT's escape analysis can **eliminate allocations entirely** based on runtime behavior:

```

1 // Looks like allocation, but JIT eliminates it!
2 public void processOrder(long orderId, double price, int quantity) {
3     Order temp = new Order(); // Allocation? NOPE!
4     temp.setOrderId(orderId);
5     temp.setPrice(price);
6     temp.setQuantity(quantity);
7
8     double totalValue = calculateValue(temp.price, temp.quantity);
9     sendToRiskEngine(totalValue);
10    // temp never escapes - JIT optimizes it away!

```

After JIT Optimization:

```

1 public void processOrder(long orderId, double price, int quantity) {
2     // Object replaced with scalar values!
3     double totalValue = price * quantity; // Inlined, zero allocation
4     sendToRiskEngine(totalValue);

```

Impact: Object creation, method calls, and field accesses become a simple multiplication. **Zero allocations, zero GC pressure.**

The Warm-Up Requirement

In high-frequency trading (HFT) systems, the warm-up phase is a mandatory production requirement rather than an optional optimization, as it ensures the JIT compiler has fully optimized the critical path before the first live order is ever processed.

```
1 // Startup warm-up phase (60 seconds)
2 for (int i = 0; i < 1_000_000; i++) {
3     processSyntheticOrder(...); // Trigger JIT optimization
4 }
5
6 // Now processing live orders with fully optimized code
7 while (true) {
8     processLiveOrder(...); // Running at peak performance
9 }
```

Performance Difference:

- **Cold start:** Interpreted/C1 code
- **Fully warmed:** C2 optimized code
- **Improvement: 10-20x faster**

Production Practice: exchange-core runs a 60-second warm-up phase before processing live market data. Without this, the first few seconds would have 10-20x worse latency.

Method inlining is the big one. When the JIT inlines your methods, it eliminates the call overhead entirely—just copies the method body directly into the caller. I’ve seen 7-10x speedups from inlining alone. But you have to keep your methods small. Under 35 bytes of bytecode for hot paths, or the JIT won’t inline them.

Then there’s escape analysis. If an object never leaves the method—doesn’t get stored in a field, doesn’t get passed to another method that might store it, doesn’t get returned—the JIT can optimize it away completely. What looks like ‘new Order()’ in your source code becomes three local variables in the actual execution. Zero allocation. Zero GC pressure.

But none of this happens instantly. You need to warm up.

In high-frequency trading, a comprehensive warm-up phase is mandatory to ensure the JIT compiler triggers C2 compilation across all hot paths—such as order validation and matching—before the system goes live. This process ensures that by the time real orders arrive, the code is already fully optimized with the necessary inlining and escape analysis applied.

Neglecting this can lead to misleading performance data; for instance, a cold-start benchmark might show a 200-microsecond latency, while the same system warmed up drops to 0.8 microseconds. The trap is spending days profiling what is essentially just the overhead of the interpreter. To get an accurate picture of a system’s true potential, you must benchmark the optimized machine code, not the transitional state.

Off-Heap Memory - Bypassing the Garbage Collector

When I first started working on low-latency Java, I followed the standard advice: let the JVM manage memory for you. Trust the garbage collector. Don’t worry about allocations because “**the JVM is really good at this.**”

That advice is great for web apps serving 100 requests per second. For microsecond-latency systems? It’s a death sentence.

At 100,000 messages per second, we were creating 12 MB of garbage every second. The young generation would fill up, the GC would wake up, and our latency would spike from 10 microseconds to 50 milliseconds. In crypto trading, 50ms means someone else already took the arbitrage opportunity. Your order is stale. You lost money.

The On-Heap Problem

Let me show you the code that was killing our latency targets when I first joined. We were processing market data like this:

```

1 // Traditional on-heap approach - multiple allocations!
2 public void processMarketData(Socket socket) throws IOException {
3     byte[] networkBytes = new byte[256];           // Heap allocation #1
4     int bytesRead = socket.getInputStream().read(networkBytes);
5
6     ByteBuffer buffer = ByteBuffer.wrap(networkBytes); // Heap allocation #2
7
8     // Extract fields
9     long orderId = buffer.getLong();
10    double price = buffer.getDouble();
11    int quantity = buffer.getInt();
12
13    // Create domain object
14    Order order = new Order(orderId, price, quantity); // Heap allocation #3
15
16    processOrder(order);
17 }

```

What's happening under the hood:

1. **Three heap allocations per message** (byte array, buffer, order)
2. **Garbage collector tracks all three objects** (overhead on every allocation)
3. **Eventually triggers GC** when young generation fills

At scale:

- Processing **100,000 messages/second**
- 3 allocations × 100K = **300,000 heap objects/second**
- Average object size: ~40 bytes
- **Total garbage created: 12 MB/second**
- Result:** Young generation (typically 100-200MB) fills in 10-15 seconds, triggering a GC pause.

GC pause duration: 10-50ms

Impact: In a system with a 10 microsecond latency budget, a 10ms GC pause is **1,000x your entire budget**. Unacceptable.

The Off-Heap Solution: Zero GC, Zero Pauses

Think of the garbage collector as a helicopter parent. It wants to know where every single byte is at all times. It tracks every object, checks if you're still using it, moves things around when you're not looking. Very helpful for normal applications. Absolutely devastating when you need consistent microsecond latency.

By moving our data off-heap, we basically emancipated our memory. If the GC doesn't know the memory exists, it can't try to "help" you by pausing your threads for 50ms while it cleans up. It's a bit more dangerous—you have to manage the lifecycle yourself, no safety net—but the silence from the GC logs is worth the risk.

Here's how we rewrote that market data processing with Agrona's off-heap buffers:

```

1 public class MarketDataProcessor {
2     // Allocate off-heap buffer ONCE at startup
3     private final UnsafeBuffer buffer;
4     public MarketDataProcessor() {
5         ByteBuffer directBuffer = ByteBuffer.allocateDirect(1024);
6         this.buffer = new UnsafeBuffer(directBuffer);
7     }
8     // Zero-copy message processing
9     public void processMarketData(UnsafeBuffer networkBuffer, int offset) {
10        // Read primitives directly from off-heap memory
11        long orderId = networkBuffer.getLong(offset);
12        offset += 8;
13        double price = networkBuffer.getDouble(offset);
14        offset += 8;
15        int quantity = networkBuffer.getInt(offset);

```



```

16         offset += 4;
17         // Process inline - no object allocation!
18         updateOrderBook(orderId, price, quantity);
19         // Zero heap allocations = zero garbage = zero GC pressure
20     }
21     private void updateOrderBook(long orderId, double price, int quantity) {
22         // Your business logic here
23     }
24 }

```

What changed:

1. **Zero heap allocations during processing** (buffer allocated once at startup)
2. **Direct memory access** (read primitives straight from network buffer)
3. **Zero-copy design** (no intermediate byte arrays or objects)
4. **GC never sees this memory** (completely off its radar)

Moving to an off-heap approach transforms performance from a game of chance into a predictable science, shifting message processing latency from 200 nanoseconds down to a mere 20 nanoseconds. While the 10x speedup in average processing time is significant, the true victory lies in the elimination of tail latency; by removing the three heap allocations per message that previously generated 12 MB of garbage every second, the system bypasses the 50-millisecond spikes caused by the garbage collector entirely. This transition ensures that latency remains a flat line even during periods of extreme market volatility, though achieving this requires a total commitment to the architecture. Mixing on-heap and off-heap logic in the same hot path is a recipe for failure, combining the overhead of manual memory management with the very GC pauses you set out to avoid. The fundamental reality is that every byte left on the heap represents a future pause, and only by committing to a zero-allocation, off-heap design can a system maintain microsecond-level consistency under load.

Why Single-Threaded Shards Beat Multi-Threading

For years, I wrote Java code the “right” way: use multiple threads to maximize CPU utilization. Use ConcurrentHashMap for thread-safe collections. Synchronize when you need to coordinate. Then I benchmarked it properly. Adding threads made our account service 800x slower. Not 2x, not 10x—800 times slower. It sounds impossible, but that’s the reality of lock contention in Java.

The Multi-Threading Trap

Let’s start with the “obvious” solution for concurrent account updates:

```

1 // Traditional multi-threaded approach
2 public class MultiThreadedAccountService {
3     private final Map<Long, Account> accounts = new ConcurrentHashMap<>();
4
5     public void updateBalance(long userId, long amount) {
6         Account account = accounts.get(userId);
7
8         synchronized(account) { // ALL THREADS WAIT HERE!
9             account.balance += amount;
10        }
11    }
12 }

```

Benchmarking the impact of lock contention reveals a startling reality: while a single thread can process an update in a mere 0.5 nanoseconds, adding even a second thread with synchronization can balloon that time to 150 nanoseconds. As more threads are introduced, the overhead of coordinating access via synchronized blocks and concurrent collections destroys cache coherence, leading to performance that can be 800 to 2,400 times slower than the single-threaded baseline. This exponential degradation proves that the CPU is spending its cycles managing lock contention and context switching rather than performing actual work, highlighting the hardest lesson in concurrent programming: locks don’t scale, they suffocate.

The Single-Threaded Solution: Sharding

Instead of: Multiple threads sharing data with locks

Use: Multiple single-threaded shards, each owning exclusive data

```

1  public class ShardedAccountService {
2      private final int shardId;          // Which shard am I? (0, 1, 2, 3)
3      private final long shardMask;       // Fast modulo mask (numShards - 1)
4
5      // This thread OWNS these accounts - NO sharing with other threads!
6      private final LongObjectHashMap<Account> myAccounts;
7
8      public ShardedAccountService(int shardId, int numShards) {
9          this.shardId = shardId;
10         this.shardMask = numShards - 1;
11         this.myAccounts = new LongObjectHashMap<>();
12     }
13
14     // Determine if this user belongs to MY shard
15     private boolean isMyUser(long userId) {
16         return (userId & shardMask) == shardId;
17     }
18
19     public void updateBalance(long userId, long amount) {
20         // Not my user? Ignore (different shard handles it)
21         if (!isMyUser(userId)) {
22             return;
23         }
24
25         // MY USER! I own this account exclusively
26         // NO LOCKS NEEDED - no other thread can touch this account!
27         Account account = myAccounts.get(userId);
28         account.balance += amount; // Direct access, ~0.5 ns!
29     }
30 }

```

Why this works:

1. **Each thread owns its data** - No other thread can access it
2. **No locks needed** - Exclusive ownership eliminates synchronization
3. **Perfect cache locality** - User 100's data always in Core 0's L1 cache
4. **Zero context switching** - Dedicated thread per shard, pinned to CPU core
5. **Linear scaling** - 4 cores = 4x throughput (not sub-linear!)

The shift from traditional locking to sharded, isolated threads reveals a massive performance gap: while adding locks and multiple threads can degrade latency by 800x and push p99s from 2 to 5,000 nanoseconds, switching to single-threaded shards restores the 0.5-nanosecond baseline with perfect linear scaling. By assigning each thread exclusive ownership of its data, CPU utilization jumps from 60% spent on coordination to 95% spent on actual work.

However, even with locks removed, the Linux scheduler can cause random 50-microsecond spikes by migrating threads between cores and flushing the L1/L2 caches. Eliminating these blips requires core isolation via the `isolcpus` boot parameter and pinning threads to specific cores using libraries like OpenHFT's Java-Thread-Affinity, ensuring that cache remains hot and execution stays uninterrupted.

```

1  import net.openhft.affinity.Affinity;
2
3  // Pin shard 0 to CPU core 1 (isolated)
4  Affinity.setAffinity(1);
5
6  // Now this thread is locked to core 1
7  // Linux scheduler can't migrate it anymore
8  processOrdersForShard0();

```

The scheduler can't touch our threads anymore. Cache stays hot. p99 spikes vanished.

Object Pooling - Zero Allocations, Zero GC Pauses

Allocation is cheap in Java. That’s what they teach you. And it’s true—allocating one object takes maybe 10-20 nanoseconds. Basically free, right?

Wrong. At 100,000 allocations per second, you’re creating 240 MB of garbage every minute. And garbage means GC pauses. Those “cheap” allocations add up to expensive pauses.

The solution feels wrong if you learned Java the traditional way: pre-allocate everything at startup and reuse it forever. You’re essentially writing C-style memory management in Java. But it works.

Pre-Allocate Everything, Reuse Forever

Instead of allocating on-demand, **pre-allocate at startup and reuse circularly**:

```

1 public class RingBufferObjectPool<T> {
2     private final T[] pool;
3     private final AtomicLong cursor = new AtomicLong(0);
4     private final int mask;
5
6     @SuppressWarnings("unchecked")
7     public RingBufferObjectPool(Supplier<T> factory, int size) {
8         // Size must be power of 2 for fast modulo
9         if ((size & (size - 1)) != 0) {
10             throw new IllegalArgumentException("Size must be power of 2");
11         }
12
13         this.pool = (T[]) new Object[size];
14         this.mask = size - 1;
15
16         // Pre-allocate ALL objects at startup (one-time cost)
17         for (int i = 0; i < size; i++) {
18             pool[i] = factory.get();
19         }
20     }
21
22     public T acquire() {
23         // Fast modulo using bitwise AND instead of expensive %
24         long index = cursor.getAndIncrement();
25         return pool[(int)(index & mask)];
26     }
27
28     // No release method needed!
29     // Objects are reused circularly: 0 → 1 → 2 → ... → 1023 → 0 → 1 → ...
30 }

```

Transitioning to object pooling can reduce p99 latency by over 20x—dropping from 250 microseconds to just 12—by replacing millions of per-order allocations with a small set of objects pre-allocated at startup. This approach completely eliminates mid-trading GC pauses, though it requires sizing pools for 3-4x peak throughput to handle sudden market volatility and accepting a longer initial startup time to pre-load the necessary domain objects. By trading memory and startup speed for runtime consistency, the system moves from 240 MB of garbage per minute to zero, ensuring that performance remains rock-solid when it matters most.

Conclusion: The “Zero” Philosophy

The real-world success of systems like **exchange-core** proves that these techniques are not merely theoretical. By combining off-heap memory, single-threaded sharding, and object pooling, you move beyond “optimizing” and toward “eliminating” overhead entirely. Zero allocations means zero GC pauses; zero locks means zero contention; zero thread migration means zero cache misses.

When you are chasing microseconds, the pattern is consistent: you must stop fighting the JVM and start feeding it exactly what it needs to optimize for the underlying hardware. Understanding the JIT, bypassing the heap, and isolating your CPU cores aren't just tricks—they are the requirements for building ultra-fast systems.

Getting Started Checklist

If you're ready to move your system into the microsecond range, start here:

- **Measure First:** Use `-Xlog:gc*:file=gc.log` and `async-profiler` to find your true bottlenecks.
- **Warm Up:** Never benchmark or go live with a “cold” JVM; run synthetic load for 60 seconds to trigger C2 compilation.
- **Isolate:** Use `isolcpus` and thread affinity to keep your L1/L2 caches hot.
- **JVM Tuning:** Start with a fixed heap and pre-touched memory:
`java -Xms16g -Xmx16g -XX:+AlwaysPreTouch -XX:+UseLargePages YourApp`

Four years ago, if you'd told me Java could achieve sub-microsecond median latency, I would have laughed. Today, I know the language isn't the limitation—our understanding of the platform is. Study the code, run the benchmarks, and start writing hardware-aware Java.

› [Back to the Table of Contents](#) ‹

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



API Development: Implementing an HTTP POST/SSE Event in Jakarta REST

API # Frameworks # ServersideJava # SSE

Author: Buhake Sindi

Buhake Sindi brings over 19 years of experience as a Senior Solutions Architect & Developer, specialising in Java Enterprise technology stacks. His extensive background in coding, design, and problem-solving is complemented by his role as a skilled DevOps Engineer, with 10+ years of hands-on experience in optimising and automating deployments. Buhake excels in translating project requirements into technical specifications and delivering robust, scalable solutions. His passion for technology and ability to communicate complex concepts clearly have made him a sought-after technical speaker and writer. Buhake's dedication to mentoring and guiding developers positions him as an influential advocate for cutting-edge development practices in corporate environments. He founded Devox4Kids South Africa, a multi-award-winning, registered Non-Profit Organisation that introduces and inspires young people into STEM, particularly programming, in a fun and engaging way. Buhake Sindi is also a LangChain4J-CDI committer. Outside of work, he occasionally creates music, produces podcast episodes, writes blogs, and is an avid runner.

Want to Dive Deeper?

Buhake Sindi is a speaker at JCON (April 20-23)! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn



In this article, we'll learn how to send messages to a client using Server-Sent Events (SSE) initiated by a client's persistent HTTP POST connection.

What Is Server-Sent Events (SSE)?

Server-Sent Events (SSE) is a web push technology, based on the HTTP protocol, that allows clients to subscribe to a stream of events, pushed by the servers over a persistent HTTP connection. The server sends chunks of data (as data event streams) asynchronously to the client, once the client connection is established.

How It Works

- **Client initiates:** The client establishes and opens a connection to a specific server endpoint using the `Accept: text/event-stream` HTTP header.
- **Server keeps connection open:** The server responds with a `Content-Type: text/event-stream` header and keeps the connection alive (setting the `Connection: keep-alive` header).
- **Server pushes data:** Whenever new data is available, the server sends it as text messages, called events (ending with a blank line) directly to the client, via a long-running HTTP connection.
- **Client receives:** The client automatically receives these messages sent back by the server as events, and processes the messages without closing the connection. In case of lost connection, clients can continue retrieving the previous streams of events by setting a `Last-Event-ID` HTTP header in a new request.

So, SSE is a one-way (unidirectional) data flow from the server to the client, and the server reuses the same connection to publish new messages to the client when its available.

SSE stream has `text/event-stream` media type and contains multiple SSE events. SSE event is a data structure encoded with UTF-8 and contains fields and comments. The field can be `event`, `data`, `id`, `retry`, or `comment`. Any other fields will be ignored.

The SSE Protocol Diagram

Here is a sequence diagram that visually shows the SSE protocol flow between client and server.

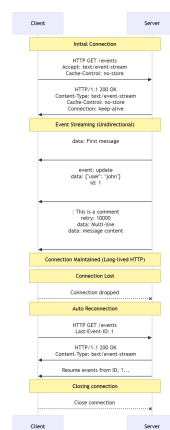


Abb. 1: The SSE protocol sequence diagram

The SSE Message Structure

The SSE message consists of a series of name/value pairs, each name/value pair is separated by a `<EOL>`, while each SSE message sent is separated by `<EOL><EOL>` (where `<EOL>` = `CRLF` or `CR` or `LF`).

A single example of an SSE message typically looks as follows:

```

1 : An example comment for the SSE stream
2 id: 1
3 event: update
4 data: {"message": "Hello, world!"}
5 retry: 5000

```

(Note: A required blank line is appended at the end of the event message at the end in order to terminate the event).

Key Fields in an SSE Message

- **data:** (Mandatory) The payload of the message. If a message has multiple **data:** lines, they are concatenated with a newline character between them.
- **event:** (Optional) The type of event. This allows clients to register specific event listeners (e.g., **event: user_login**). If this event type is not provided, it defaults to “message”.
- **id:** (Optional) A unique identifier for the event, used to track the message sequence. If the connection drops, the client sends this ID back to the server in the **Last-Event-ID** header to resume the stream.
- **retry:** (Optional) The reconnection time in milliseconds. It tells the browser how long to wait before trying to re-connect if the connection is lost.
- **:** (Comment) Lines starting with a colon are ignored and often used as a heartbeat to keep the connection alive.

Now, let us investigate the possible ways of building an SSE server in Jakarta REST.

Server-Sent Events API in Jakarta REST

Jakarta REST provides built-in APIs to handle Server-Sent Events (SSE) within the `jakarta.ws.rs.sse` package. It is used to accept a connection from the client and send events to one or more clients.

Current Server-Side SSE Implementation.

To create an SSE server, a resource method **must** be annotated with a `@GET` annotation, produce the media type as `MediaType.SERVER_SENT_EVENTS` (or `"text/event-stream"`) and inject an `SseEventSink` to the resource method parameter. When all 3 conditions are met, the resource method is an SSE resource method.

The following example accepts SSE connections and sends a new SSE event immediately to the client, via the `OutboundSseEvent` object, before closing the connection.

```

1 @Path("events")
2 public class SseResource {
3
4     @GET
5     @Produces(MediaType.SERVER_SENT_EVENTS)
6     public void listenToEvents(@Context SseEventSink sink, @Context Sse sse) {
7         // Example of sending an event immediately (usually done in a background task)
8         OutboundSseEvent event = sse.newEvent("My first event data");
9         sink.send(event);
10    }
11 }

```

The Key Interfaces for Server-Side implementation:

- The `SseEventSink` instance, injected into the resource method, represents the incoming connection from the client and is used to send the events.
- The `Sse` instance is injected to serve as a factory for creating `OutboundSseEvent` instances and an `SseBroadcaster` (which we will view later).
- The `OutboundSseEvent` instance represents a single outbound Server-Sent Event (SSE) message being sent from the server to a client. You can obtain an object via one of the `Sse` factory method `newEvent(...)`, or via its Builder retrieved by the `Sse.newEventBuilder()` method.

- The application may need to send SSE event messages to multiple clients simultaneously. In Jakarta REST, multiple `SseEventSink`'s can be registered to a single `SseBroadcaster`, which can be obtained from the injected `Sse`'s `newBroadcaster()` method. The `OutboundSseEvent` instance can, then, be broadcasted to all event sinks by calling the `SseBroadcaster.broadcast()` method.

The following example accepts an SSE connection as an `SseEventSink`, registers it to an `SseBroadcaster` and broadcasts the SSE event to all registered clients.

```

1  @Path("events")
2  @ApplicationScoped
3  public class SseResource {
4
5      @Context
6      private Sse sse;
7      private volatile SseBroadcaster sseBroadcaster;
8
9      @PostConstruct
10     public init() {
11         this.sseBroadcaster = sse.newBroadcaster();
12     }
13
14     @GET
15     @Produces(MediaType.SERVER_SENT_EVENTS)
16     public void listenToEvents(@Context SseEventSink sink) {
17         sseBroadcaster.register(sink);
18         OutboundSseEvent event = sse.newEvent("My first event data");
19         sseBroadcaster.broadcast(event); //Broadcast to every client registered.
20     }
21 }

```

By now, you'll realise that an `SseEventSink` is injected only on `@GET` SSE resource method. But, what if your APIs require you to handle client requests and respond with SSE events only on HTTP POST calls?

In Jakarta REST, replacing the `@GET` to a `@POST` on an SSE resource method will result in the container injecting a `null` when injecting an `SseEventSink`.

Consuming SSE Streams Using the Jakarta REST Client API

Reading Server-Sent Events (SSE) using the Jakarta REST client API is accomplished using the `SseEventSource` interface. This interface is to simulate the JavaScript `EventSource` object. This client-side component allows you to connect to an SSE endpoint and register consumers to handle incoming events asynchronously.

Steps to Read SSE Events

The following steps outline how to set up and use the `SseEventSource`:

1. **Obtain a `Client` Instance:** Start by creating a standard `Client` instance using `ClientBuilder`.
2. **Define the Target URL:** Configure a `WebTarget` for the SSE endpoint.
3. **Build the `SseEventSource`:** Use the `SseEventSource.target(target).build()` factory method to create an event source instance. You can also customise the reconnect delay if needed using `reconnectingEvery()`.
4. **Register Event Consumers:** Register consumers (lambda expressions or method references) to process different aspects of the event stream:
 - **onEvent:** A `Consumer<InboundSseEvent>` invoked for each received event.
 - **onError:** A `Consumer<Throwable>` invoked if an unrecoverable error occurs.
 - **onComplete:** A `Runnable` invoked when the stream naturally completes (e.g., server sends a 204 No Content status).
5. **Open the Connection:** Start the connection to the server and begin processing incoming events using `open()`. The `SseEventSource` runs in its own internal thread and handles automatic reconnections by default.
6. **Manage the Lifecycle:** The `SseEventSource` implements `AutoCloseable`, so it is best practice to use it within a `try-with-resources` block to ensure proper closure, or explicitly call `eventSource.close()` when you are finished receiving events.

The following example showcases the consummation of the SSE event `InboundSseEvent` message via the `SseEventSource` using the Jakarta REST client API.

```

1 public class SSEConsumer {
2
3     public static void main(String... args) {
4         Client client = ClientBuilder.newClient();
5         WebTarget target = client.target("http://localhost:8080/yourapp/events");
6         try (SseEventSource eventSource = SseEventSource.target(target).build()) {
7             Consumer<InboundSseEvent> onEvent = event -> {
8                 // Read the event data and process it
9                 System.out.println("Received event: " + event.readData(String.class));
10            };
11            eventSource.register(onEvent,
12                               error -> error.printStackTrace(),
13                               () -> System.out.println("Connection closed"));
14            eventSource.open();
15        } //The SseEventSource implements AutoCloseable, so it is best practice to use it
           within a try-with-resources block to ensure proper closure.
16    }
17 }

```

Note that the `SseEventSource` will connect and open to the HTTP SSE server via an HTTP GET connection.

Key Interfaces

- **`SseEventSource`**: The primary client-side entry point for consuming SSE streams, managing the connection and event handling.
- **`InboundSseEvent`**: Represents a single incoming event from the server. It provides methods like `readData()` to access the event's data payload and metadata like event ID, name, or retry interval.
- **`Client` and `WebTarget`**: Standard Jakarta REST client API components used to configure the base connection details.

But, what if your API provides an HTTP POST SSE stream response? How do we provide a POST SSE event resource call and send event messages if the `SseEventSink` is not available during the POST?

Server-Side HTTP POST SSE Implementation.

The `SseEventSink.send()` and the `SseBroadcaster.broadcast()` both return a `CompletionStage` as a way to provide a *handle* to the action of asynchronously sending a message to a client. Thus, an SSE resource method is an asynchronous method, so future SSE events can be sent even long after the method has been returned.

We need to apply the same procedure to implement an HTTP POST SSE resource method. Luckily, Jakarta REST provides other ways to provide asynchronous processing of resources.

The `jakarta.ws.rs.context.AsyncResponse` Interface.

An `AsyncResponse`, when injected to a resource method, provides a means for asynchronous response processing. The resource method returns but the response is not readily available, until it is processed further. Thus, the server keeps the client HTTP connection long-running and active until it receives the response from the server.

Key Concepts:

- **Non-Blocking Operations**: When a resource method is injected with `AsyncResponse` using the `@Suspended` annotation, the standard server thread is immediately returned to the container's thread pool to handle new incoming requests.
- **Decoupled Processing**: The actual, time-consuming work (e.g., complex computations, database queries, external API calls) is executed on a separate thread (which you manage, for example, using a `ManagedExecutorService` in a Jakarta EE environment or a custom `ExecutorService`).
- **Resuming the Request**: Once the long-running operation is complete, the application calls one of the `resume()` methods on the `AsyncResponse` instance. This action sends the final HTTP response back to the client and terminates the request.

The following example showcases an HTTP POST SSE event that sends an SSE data attribute to the client, and the request terminates.

```
1 @Path("events")
2 @ApplicationScoped
3 public class SseResource {
4     @POST
5     @Produces(MediaType.SERVER_SENT_EVENTS)
6     public void listenToEvents(@Suspended AsyncResponse asyncResponse) {
7         executor.submit(() -> {
8             //Long running operation goes here....
9             //Resume
10            asyncResponse.resume("data: Hello world!\n\n");
11        });
12    }
13 }
```

The `AsyncResponse.resume()` method resumes the suspended `AsyncResponse` instance. The object passed to the `resume()` method is the same object you would have returned if the resource method was a synchronous call (it can be an entity object, a primitive type or Jakarta REST `Response` object).

But what if you have a large data set that you want to stream as SSE data events without loading the entire data set into memory?

The `jakarta.ws.rs.core.StreamingOutput` Interface.

The `StreamingOutput` interface is a type that is used as a return value from a resource method, or as an entity of a `Response` when you want to stream the output. It's an alternative to `MessageBodyWriter`.

The advantage of `StreamingOutput` is that it defines a single method: `void write(OutputStream output) throws IOException, WebApplicationException`, which allows an application to write directly to an `OutputStream` as a message body, which is then sent to the client.

When a resource method returns a `StreamingOutput` instance (usually wrapped in a `Response` object), the Jakarta REST runtime calls the `write` method, providing the actual HTTP response output stream. Your implementation is responsible for writing all data to this stream and flushing it.

For SSE, we can write the SSE event data using the `StreamingOutput` and feed the instance directly into the `AsyncResponse` directly. That way, we still manage to write a lightweight payload to the message body that is sent to the client, as SSE event streams, even if we do have a large data set to process.

Putting It All Together (the Implementation).

To create an SSE POST server, a resource method **must** be annotated with a `@POST` annotation, produce the media type as `MediaType.SERVER_SENT_EVENTS` (or `"text/event-stream"`) and inject an suspended `AsyncResponse` to the resource method parameter. The resource method, typically, returns a `void`.

Since we cannot use the `OutboundSseEvent` instance to send it as an SSE message, we will have to write the SSE event stream directly to the writer, according to the SSE stream format specification (see link of the SSE stream in the resources section).

The example below showcases the use of `StreamingOutput` to process large datasets, and returns as a `Response` entity, which is then resumed as a response by `AsyncResponse`.

```

1  @Path("events")
2  @ApplicationScoped
3  public class SseResource {
4      @POST
5      @Path("stream")
6      @Consumes(MediaType.APPLICATION_JSON)
7      @Produces(MediaType.SERVER_SENT_EVENTS)
8      public void sendMessageStream(@Suspended AsyncResponse asyncResponse) {
9
10         StreamingOutput stream = new StreamingOutput() {
11             @Override
12             public void write(OutputStream out) throws IOException,
13                 WebApplicationException {
14                 Writer writer = new BufferedWriter(new OutputStreamWriter(out));
15                 // Simulate writing a large dataset
16                 for (int i = 0; i < 10000000; i++) {
17                     writer.write("data: {id: " + i + "}\n\n");
18                 }
19                 writer.flush(); // Crucial to flush the stream to ensure data is sent
20             };
21         ResponseBuilder responseBuilder = Response.ok()
22             .type(MediaType.SERVER_SENT_EVENTS_TYPE)
23             .header("Cache-Control", "no-cache, no-transform")
24             .header("Connection", "keep-alive")
25             .entity(stream);
26         asyncResponse.resume(responseBuilder.build());
27     }
28 }

```

Note: Please ensure that you flush the `Writer` in order for the data to be sent.

Note: This is an oversimplification of a solution case. It's up to the developer to ensure that the operations are safely executed in an asynchronous model, but the gist of the solution is provided for better understanding.

Alternative Option: Using the `jakarta.servlet.AsyncContext` Interface.

A final approach in publishing SSE data to the client is relying on old-daddy faithful, the `HttpServletRequest`. Since Servlet 3.0, the API provides a mechanism to detach from the synchronous request/response context and send the response later via asynchronous processing, provided the client connection is still alive. The `AsyncContext` interface represents the execution context for an asynchronous operation that was initiated on a `ServletRequest`.

To obtain the `AsyncContext`, one can start asynchronous processing by calling the `request.startAsync()`, where `request` is an instance of `ServletRequest`.

In Jakarta REST, one can inject the `HttpServletRequest` and we can simply use the `AsyncContext` in the same manner as one would in a `Servlet` environment.

This example shows how an `HttpServletRequest` is injected in order to establish an asynchronous processing context to be used to send SSE events to the response, using the `AsyncContext` instance.

```

1  @Path("events")
2  @ApplicationScoped
3  public class SseResource {
4
5      @POST
6      @Path("stream")
7      @Consumes(MediaType.APPLICATION_JSON)
8      @Produces(MediaType.SERVER_SENT_EVENTS)
9      public void sendMessageStream(@Context HttpServletRequest request) {
10
11          AsyncContext asyncContext = request.startAsync();
12          asyncContext.setTimeout(0); //Infinite timeout
13          HttpServletResponse response = (HttpServletResponse) asyncContext.getResponse();
14          response.setContentType("text/event-stream");
15          response.setCharacterEncoding("UTF-8");
16          response.setHeader("Cache-Control", "no-cache");
17          response.setHeader("Connection", "keep-alive");
18
19          PrintWriter writer = response.getWriter();
20
21          for (int i = 0; i < 1000000; i++) {
22              writer.write("data: {id: " + i + "}\n\n");
23          }
24          asyncContext.complete(); //Completes the asynchronous operation that was started
                                   //on the request that was used to initialize this AsyncContext, closing the
                                   //response that was used to initialize this AsyncContext.
25      }
26  }

```

Here's a conceptual breakdown:

1. **Resource Setup:** Create a RESTful resource method to handle the initial POST request, which will set up the SSE stream.
2. **Start Async:** Inside the POST method, call `request.startAsync()` to get an `AsyncContext`, freeing the request thread.
3. **Event Listener:** Register an `AsyncListener` to handle events like timeout or completion.
4. **Background Processing:** Use an `ExecutorService` (e.g., from CDI or `CompletableFuture`) to process the POST data (e.g., save to DB) off the main thread.
5. **Complete/Dispatch:** Call `asyncContext.complete()` when done, or `asyncContext.dispatch()` to forward to another resource, potentially for cleanup or further processing.

Consuming HTTP POST SSE Streams Using a Modern Java JDK `HttpClient`.

To read a POST Server-Sent Event (SSE) stream using the modern JDK `HttpClient`, you can send the request with a `BodyHandler.ofLines()` and then process the resulting `Stream<String>` as events.

The JDK `HttpClient` does not have built-in, first-class support for the full SSE protocol (e.g., automatic reconnection or event type parsing), but you can implement a basic client using its streaming capabilities.

Implementation Steps:

1. **Create an `HttpClient` instance.**
2. **Build an `HttpRequest`** for the POST method, including the body and the `Accept: text/event-stream` header.
3. **Send the request using `BodyHandlers.ofLines()`** to get a `Stream<String>` which can then be processed line by line.
4. **Process the lines** to extract the SSE data.

Here is an example of how to create a basic, synchronous SSE client for a POST request:

```

1 public class SsePostClient {
2
3     public static void main(String[] args) {
4         // Define the target URL and the POST body
5         String url = "http://localhost:5001/api/extra/generate/stream"; // Replace with
        your SSE endpoint
6         String requestBody = "{\"someKey\": \"someValue\"}"; // Replace with your POST
        data
7         HttpClient client = HttpClient.newBuilder()
8             .version(HttpClient.Version.HTTP_2)
9             .connectTimeout(Duration.ofSeconds(10))
10            .build();
11        HttpRequest request = HttpRequest.newBuilder()
12            .uri(URI.create(url))
13            .header("Accept", "text/event-stream") // Standard SSE MIME type
14            .header("Content-Type", "application/json") // Adjust content type as
            needed
15            .POST(BodyPublishers.ofString(requestBody))
16            .build();
17        System.out.println("Sending POST request and connecting to SSE stream...");
18        try {
19            // Send the request and get a stream of lines
20            HttpResponse<Stream<String>> response = client.send(request, BodyHandlers.
                ofLines());
21            if (response.statusCode() == 200) {
22                System.out.println("Connection established. Processing events:");
23                // Process each line in the response body stream
24                response.body().forEach(line -> {
25                    // Basic processing: check if the line starts with "data:" and print
                    the content
26                    if (line.startsWith("data:")) {
27                        String data = line.substring(5).trim();
28                        System.out.println("Received event data: " + data);
29                    } else if (line.startsWith("event:")) {
30                        String eventType = line.substring(6).trim();
31                        System.out.println("Received event type: " + eventType);
32                    }
33                    // Handle 'id:' or 'retry:' lines for a more robust client (not shown
                    here for brevity)
34                });
35            } else {
36                System.err.println("Failed to establish connection. Status code: " +
                    response.statusCode());
37            }
38        } catch (Exception e) {
39            System.err.println("An error occurred: " + e.getMessage());
40            e.printStackTrace();
41        }
42    }
43 }

```

Note:

1. The standard JDK `HttpClient` does not automatically handle the SSE spec's `retry` field or implement automatic reconnection. For a production-ready client with these features, you may need a dedicated library.
2. I've implemented an `SSEEventSubscriber` (which is a `Flow.Subscriber<String>`) that you can use instead of writing your own SSE stream processor. You can send the request using the `BodyHandlers.fromLineSubscriber`, passing the `SSEEventSubscriber` instance (which also requires an `EventHandler` that will receive SSE `Event` for client processing). The implementation implements the SSE specification (see the resource link below). The full source code is available on [GitHub](#) and you can use it in your project by adding this GAV to your Maven project `za.co.sindi:sindi-commons:<latest-version>`.

Conclusion

Jakarta REST provides various solutions to handle client/server request and response both for synchronous and asynchronous processing. We've showcased various approaches to Server-Sent Events processing, especially outside of the usual HTTP GET operation (as specified on the SSE specification).

Maybe, in future, they should introduce `SseEventSink` to be injected on methods outside of `@GET`?

Links/Resources

1. [Jakarta RESTful Web Services](#) (version 4.0.0 specification)
2. [Server-Sent Events](#) (specification)

› [Back to the Table of Contents](#) ‹



Open J Proxy: A Modern, Open-Source Approach to Database Connectivity for the Java Ecosystem

API

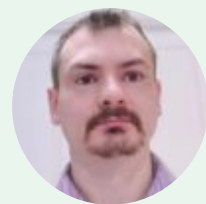
Frameworks

JDBC

OJP

Author: Robetti Rogerio

Rogerio Robetti is a software engineer with many years of experience, an enthusiast and advocate of modern approaches to software engineering. Has a soft spot for resource usage optimization and automation. Have worked in very heterogeneous environments and different roles from developer to architect and from startup size companies to global organizations, mostly producing enterprise grade web applications but also having experience in R&D, mobile and mainframe applications. Currently lives in Dublin, Republic of Ireland.



LinkedIn

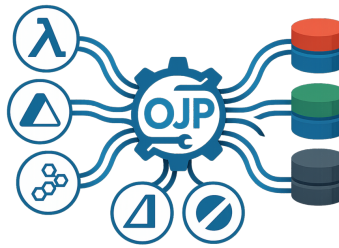


Abb. 2: OJP as database connectivity centralizer

Historical Context on JDBC

The Java ecosystem has never stood still. It began its life powering thick client applications, grew into enterprise-grade monolithic servers, reinvented itself with dependency injection and lightweight frameworks, and then marched decisively into the cloud-native world with microservices, containers, and distributed tracing. But through all these evolutions, one constant has remained surprisingly unchanged: **the way Java applications connect to databases.**

For more than twenty-five years, [JDBC](#) has been the bedrock of relational data access in Java. And for good reason. JDBC has been *battle-tested* across generations of systems, hardened by millions of deployments, and supported by every major relational database vendor. [Oracle](#), [PostgreSQL](#), [SQL Server](#), [MySQL](#), [DB2](#), [MariaDB](#), and, more recently, [CockroachDB](#) — all have deeply optimized JDBC drivers with decades of tuning. Entire enterprise architectures depend on their stability. It has earned its longevity.

But its strength is also its limitation. JDBC was designed for a world where applications were fewer, heavier, and mostly static. The assumption behind JDBC’s programming model is that you get a connection, work with it synchronously, and return it. It assumes that acquiring a connection is cheap and predictable — an assumption that made perfect sense in the 1990s when most systems were vertically scaled and rarely had more than a handful of database clients.

Fast forward to today’s architectures, where a single company may run hundreds of JVMs across many [microservices](#), each auto-scaling independently, each maintaining its own pool of tens of connections. Modern systems multiply concurrency horizontally, while relational databases remain limited by physics: CPU cores, memory, network bandwidth, and the finite overhead they can dedicate to maintaining open connections.

This dissonance — *infinitely elastic compute vs. strictly bounded databases* — has become one of the most understated and underestimated issues in the industry. And it gets amplified by some of JDBC’s fundamental assumptions.

The Underlying Problem: Modern Systems, Ancient Assumptions

To understand why, let’s examine the most common, almost ritualistic pattern in JDBC:

```
1 Connection conn = dataSource.getConnection();
2 PreparedStatement ps = conn.prepareStatement(SQL);
3 ps.setString(1, value);
4 ResultSet rs = ps.executeQuery();
```

Every Java developer has typed these lines a thousand times. But hidden inside this sequence is an architectural assumption from a bygone era.

The connection is acquired *before* the prepared statement is created, and long before the SQL is actually executed.****

This API pattern implies that the application must “claim” a connection to the database even while it is still building SQL parameters, performing JSON transformations, determining business rules, or performing other CPU-bound work. The database connection sits idle until the very last step.

At this point, some readers may object: “*But the time spent between getting the connection and executing the query is tiny—milliseconds at most.*” And that is often true for a single request on a single service instance. However, modern systems rarely operate at that scale. When you have dozens of microservices, each running multiple replicas, each processing hundreds or thousands of requests per second, those “tiny gaps” compound dramatically. A few milliseconds of wasted hold time

multiplied across 80 services can add up to seconds of unnecessary connection occupancy per second of workload. **At scale, small inefficiencies become systemic bottlenecks.**

Another common argument is: “Modern drivers don’t hand out a real database connection immediately anyway—they use lazy acquisition or intermediate pooling tricks.” And some indeed do. But JDBC never standardized that behavior. The specification still treats the call to `getConnection()` as obtaining a real connection, and different drivers implement this differently. Some allocate a real connection immediately; others delay it; others mix strategies based on state. That variability means developers can’t rely on lazy acquisition as a guaranteed cross-driver feature—leaving room for unexpected behavior, especially in mixed-vendor environments.

In OJP’s model, when the developer asks the driver for a “connection,” the driver returns a **virtual handle**. Nothing physical is acquired yet. The proxy does not establish a real database connection until a SQL statement is about to *execute*. All the preparatory steps — constructing a `PreparedStatement`, setting parameters, performing logic — happen without consuming any real database slot.

Only when `.executeQuery()` or `.executeUpdate()` is called does OJP request a physical connection from its centralized pool. That increases reuse efficiency and reduces idle time at scale. The result is fewer real connections and more throughput per connection.

This addresses one of JDBC’s oldest inefficiencies with a simple but powerful idea:

Don’t **take a connection until you’re truly ready to use it.**

Why Existing Connection Pools Cannot Solve This

Classic JDBC pools, such as HikariCP, do an incredible job *inside* a single JVM. They are speedy, predictable, and well-engineered. But they were never intended to coordinate across dozens of JVMs. Every pool optimizes locally. None has global visibility. None can sense that another service is about to spike traffic. None can cap total connection usage across microservices. And none can delay actual connection enlistment because JDBC itself makes that problematic.

Which leads to modern systems repeatedly running into the same wall:

Too many microservices, each opening too many connections, all fighting over the same finite database.

Open J Proxy steps in exactly at this junction — not replacing JDBC’s strengths, but complementing them with the distributed coordination that modern environments demand.

What OJP Actually Is

OJP is a lightweight, cloud-oriented relational database proxy that sits between your Java applications and your database, acting as an intelligent coordination layer. Instead of each application independently opening and managing its own pool of database connections, OJP centralizes that responsibility, enforcing fairness across services, applying backpressure when the database is under load, and shaping workloads in a predictable, controlled manner. All of this happens transparently — frameworks like Spring Boot, Quarkus, Micronaut, or even vanilla JDBC code treat OJP exactly as they would treat any normal JDBC driver.

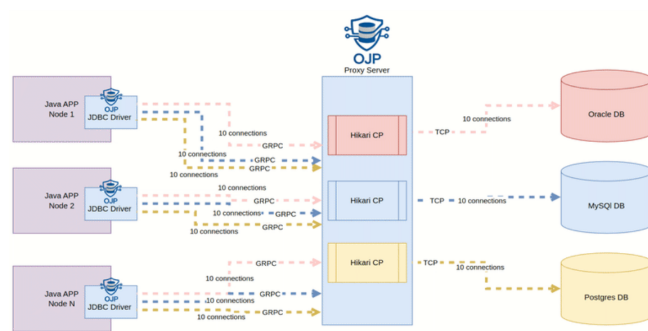


Abb. 3: OJP high level design

At a high level, OJP is composed of two core components:

The OJP Server — A Layer-7 Database Proxy

The OJP Server runs as a standalone process and functions as a **Layer 7 (Application Layer) database proxy**. In simple terms, this means OJP understands SQL-level semantics and manages database access at the same conceptual level as your application logic, not merely as a network packet forwarder.

The server is implemented as a high-performance **gRPC service**, which provides several benefits:

gRPC

- A modern, efficient RPC protocol built on HTTP/2
- Enables streaming, flow control, and multiplexing
- Ideal for low-latency interactions between the driver and the proxy
- Makes OJP cloud-friendly and compatible with service meshes, mTLS, and modern distributed systems patterns

HikariCP

Inside the server, OJP uses [HikariCP](#), the industry's fastest and most reliable JDBC connection pool. By running HikariCP centrally rather than in each microservice, OJP reduces connection duplication, lowers total connection count, and ensures uniform pool behaviour across all services.

Native JDBC Drivers

OJP does not reinvent database protocols. It uses the **native JDBC drivers** (PostgreSQL, MySQL, Oracle, SQL Server, etc.) to communicate with the actual database.

This provides:

- decades-proven reliability,
- full feature support,
- mature protocol handling,
- and complete compliance with database vendor expectations.

By combining [gRPC](#), [HikariCP](#), and **native JDBC drivers**, the server becomes a robust database gateway that can safely and efficiently mediate access from dozens or hundreds of applications.

The OJP JDBC Driver — A Type 3 JDBC Driver

On the application side, developers use the **OJP JDBC Driver**, which replaces the native driver in the application's classpath. The OJP driver is the **first fully open source Type 3 JDBC driver in the world**, also known as a "Network Protocol Driver."

This means:

- It does *not* connect to the database directly.
- Instead, it speaks to the OJP Server using gRPC.
- The server then uses the native JDBC driver to talk to the database.

This architecture brings several advantages:

Virtual Connections

When your application calls `DriverManager.getConnection()`, the OJP driver returns a **virtual connection** rather than a real one.

A real database connection is only requested from the OJP server once a statement is actually executed (`executeQuery()`, `executeUpdate()`, etc.).

This eliminates wasted idle time and dramatically improves connection reuse.

Load Balancing

Because the driver can be configured with multiple OJP servers, it can automatically:

- distribute SQL workloads across them,
- choose the least-loaded node,
- and keep sessions sticky when required for transaction correctness.

Automatic Failover

From version 0.3.1-beta onwards, if the driver detects that an OJP server is unavailable, it automatically switches to another node in the list.

This gives applications:

- high availability,
- fast recovery,
- and resilience is entirely invisible to the developer.

Cross-Language Support via Protobuf

Since both the driver and the server speak Protocol Buffers, the same RPC interface can be used by non-Java applications. Python, Go, Kotlin, Node.js, Rust — any language with Protobuf + gRPC support can communicate with OJP as if it were native.

Bringing It All Together

The result of this architecture is a clean separation of responsibilities:

- **Applications** use a minimal, efficient driver that hands off work to OJP.
- **The OJP Server** uses proven JDBC drivers and HikariCP to centrally manage real connections.
- **gRPC** provides the connective tissue, enabling fast, structured communication with strong typing and low overhead.
- **Virtual connections, load balancing, and failover** make OJP not just a proxy, but a fully managed connectivity layer.

This design allows modern Java applications to overcome longstanding JDBC limitations while maintaining compatibility with the entire Java ecosystem.

As Java Champion Bruno Bossola well summarized: *“The only open-source JDBC Type 3 driver globally, this project introduces a transparent Quality-of-Service layer that decouples application performance from database bottlenecks. It’s a must-try for any team struggling with data access contention, offering easy-to-implement back-pressure and pooling management.”* (Bruno Bossola - Java Champion and CTO @ [Meterian.io](https://meterian.io))

Backpressure: The Safety Valve Databases Have Needed for Years

Backpressure is one of the most potent concepts in distributed systems. It refers to an upstream system’s ability to slow incoming work when downstream resources are saturated.

Relational databases are susceptible to overload. When they receive more requests than they can handle, they do not fail gracefully—they start to accumulate wait events, pile up I/O queues, thrash caches, or stall threads; latency skyrockets. Connection attempts time out. Entire application clusters begin failing health checks.

Traditional JDBC pools have no global view, so they cannot collectively apply backpressure. One JVM may happily open 30 connections while the database is already drowning under the weight of another service.

OJP, by contrast, serves as a centralized choke point. It monitors the database’s health — real concurrency, pool size, and query average latency — and uses that insight to regulate how quickly requests from all services are forwarded downstream. When necessary, OJP will slow down requests just enough to preserve stability. This is the database equivalent of an anti-lock braking system: instead of skidding out of control, the system maintains traction.

The goal of OJP's backpressure is not to reject work, but to prevent the database from spiralling into an unrecoverable state under mounting pressure.

Bulkheads and Slow Query Segregation: Architectural Resilience for SQL Workloads

In modern architectures, “bulkheads” refer to the strategy of isolating workloads so that failures in one domain do not cascade into others. In maritime engineering, bulkheads prevent water from flooding one compartment and sinking an entire ship. In system design, they avoid a misbehaving subsystem from dragging the rest of the system down.

OJP applies the bulkhead principle to SQL execution. Instead of allowing slow or heavy queries to block the same execution lanes used by fast, OLTP-style queries, OJP dynamically detects slow queries and routes them to separate execution slots.

But OJP goes further than classical bulkheads. It employs a **lightweight adaptive runtime labelling** mechanism that continuously observes query duration. Queries aren't pre-labelled; OJP learns their behaviour as they run. This avoids the common flaw of static “rules-based” approaches that misclassify workloads.

Once a slow query lane is established, OJP also applies the **Steal Work pattern**. If a slow lane has idle capacity — say, a long-running lane with four slots, but only 1 is active — OJP can temporarily lend those idle slots to fast lanes during periods of pressure or vice versa, idle fast slots can be filled by slow queries. This ensures the system stays efficient even when slow workloads are not active.

This combination — **Bulkheads + Adaptive Labelling + Steal Work** — forms the foundation of OJP's Slow Query Segregation Strategy. The net effect is that fast workloads always remain responsive, slow workloads remain isolated, and unused capacity is never wasted. It is a dynamic, self-balancing system designed to offer predictable performance even when the workload mix is complex and volatile.

A Unified, Multi-Database Access Layer

One of OJP's unique strengths is its support for a wide range of relational engines: PostgreSQL, MySQL, MariaDB, Oracle, SQL Server, DB2, CockroachDB, and even H2 for local development and theoretically any JDBC-compatible database. Organizations that maintain heterogeneous data environments — common in large enterprises — can standardize operational patterns through a single proxy.

This is not typical. Most existing database proxies are tied to one specific engine or cloud. OJP aims for inclusivity, allowing teams to unify their connectivity strategy without switching tools or rewriting code.

This also reduces vendor lock-in. As teams migrate from one database to another, the connection layer remains consistent and predictable.

Performance Effects: Stability, Not Just Speed

What's striking in early benchmarks is not simply that OJP maintains lower latency under load, but *how stable* the system becomes.

With direct JDBC, latency rises linearly with traffic. Mixed workloads cause pooling contention. Boot storms overwhelm the database. Under high concurrency, tail latencies stretch into seconds.

With OJP, latency remains flat and predictable even at high concurrency. Real database connections stay within bounded limits. Slow queries no longer paralyse fast paths. “Rowdy neighbours” are contained. And boot storms lose their impact because connection enlistment occurs only at execution time, not at pool initialization.

The goal is not just speed — it is **architectural serenity**.

Integration That Feels Almost Too Simple

OJP's ease of integration is one of its strongest advantages. A typical Spring Boot application requires a change to its JDBC URL:


```
1 spring.datasource.url=jdbc:ojp[ojp-server:1059]_postgresql://db:5432/mydb
2 spring.datasource.driver-class-name=org.openjproxy.jdbc.Driver
```

The same applies to Quarkus, Micronaut, and Helidon: modify the URL, set the driver, and the configuration is done.

Add OJP JDBC dependency:

```
1 <dependency>
2   <groupId>org.openjproxy</groupId>
3   <artifactId>ojp-jdbc-driver</artifactId>
4   <version>0.3.1-beta</version>
5 </dependency>
```

Crucial last step: **disable the application's local connection pool**.

Because OJP handles pooling centrally, the local pool becomes unnecessary—and, in fact, counterproductive— and a source of issues as the local and OJP pools start to step on each other's toes.

Once configured, the application treats OJP as a regular database driver. Transactions, prepared statements, batch operations, and metadata calls all flow naturally through the proxy.

This is intentional. Developers should not have to *think* about OJP for it to deliver value.

The Broader Impact of OJP

What OJP offers is not merely a performance enhancement, but an architectural evolution. It bridges the fast-moving, highly dynamic behaviour of cloud-native Java systems with the steadier, capacity-limited nature of relational databases. It preserves JDBC's reliability and decades of proven stability while adding capabilities it was never designed to provide. By doing so, OJP respects the strengths of the existing ecosystem while extending it into a model better suited for today's distributed environments.

OJP gives:

- **developers** to achieve smoother performance without changing their code,
- **managers** improved stability and reduced infrastructure cost,
- **devops engineers** a safety valve that protects the database,
- **architects** a powerful new tool that addresses systemic issues at the root,
- **Data teams'** isolation from unpredictable operational workloads.

It is a rare win-for-everyone technology.

Final Thoughts

Open J Proxy is still maturing, but it already represents a fresh rethinking of how Java systems interface with relational databases. By combining decades of proven JDBC reliability with modern distributed-systems engineering — connection deferral, global pooling, adaptive slow-query segmentation, and intelligent backpressure — it fills a long-standing gap in how microservices share and coordinate access to the most precious resource they have: the database.

OJP is fully open source, Apache 2.0-licensed, and actively welcoming contributors. If your team experiences connection storms, slow query cascades, unpredictable boot times, or database saturation under load, Open J Proxy may be the missing architectural link you've been searching for.

Learn more:

GitHub:<https://github.com/Open-J-Proxy/ojp> LinkedIn: <https://www.linkedin.com/company/open-j-proxy/> Website: <https://openjproxy.com/>

› [Back to the Table of Contents](#) ‹



Most of us have built numerous Spring Boot applications that perform standard CRUD (Create, Read, Update, Delete) operations on structured data stored in relational databases such as PostgreSQL. These systems typically rely on fixed schemas and well-defined table structures. But, what happens when your use case requires storing and manipulating unstructured, schema-less JSON documents inside the RDBMS tables?

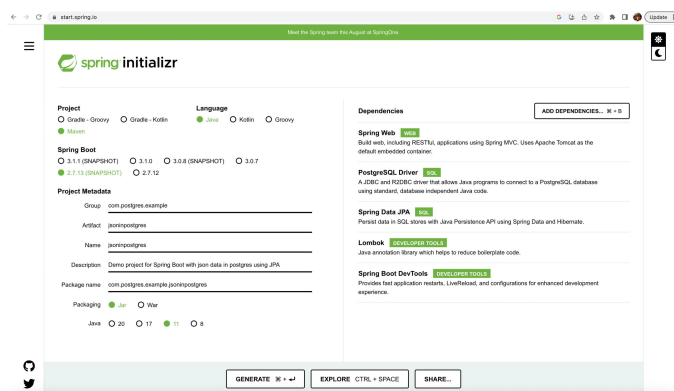
I have been exploring effective strategies to store, update, retrieve and delete unstructured, schema-less JSON data in PostgreSQL tables using Spring Boot. Today, I'm excited to share these insights with you. This feature was introduced by PostgreSQL when NoSQL databases were getting a lot of popularity over traditional RDBMS systems. While many developers might be aware of the fact that PostgreSQL supports JSON and JSONB types, the nuances—such as partial updates using functions like `jsonb_set()`—are less commonly implemented.

By the end of this article, we will have a fully functional REST controller with CRUD endpoints capable of Storing, Reading, Updating and Deleting JSON data in PostgreSQL database using Spring Boot. This includes updating specific key-value pair of JSON, adding a new key-value pair into the JSON, and removing a key-value pair from JSON etc using functions like `to_json()` and `json_set()` for a fine-grained JSON manipulation.

Here is the [link](#) to a Spring Boot application that I wrote.

Let's CODE JSON in POSTGRESQL!

1. Navigate to <https://start.spring.io/> and create a Spring Boot project with the below-mentioned dependencies.



2. Spin up a PostgreSQL docker container to connect with our Spring Boot application by following steps 1–4 from [here](#).
3. Update the application.properties as mentioned below. Because our goal is to persist JSON data into the PostgreSQL database, hence, we specify “stringtype=unspecified” in spring.datasource.url field.

```
1 spring.datasource.url = jdbc:postgresql://localhost:5432/postgres?stringtype=unspecified
2 spring.datasource.username = postgres
3 spring.datasource.password = password
4 spring.datasource.platform = postgresql
```

Below line creates a Student table using JPA entity and updates it if it already exists. (Doesn't reset it on next time run)

```
1 spring.jpa.hibernate.ddl-auto = update
2 spring.jpa.properties.hibernate.show_sql=true
3 server.port=8081
```

4. Create entity classes as below. Student table has a jsonb type column favs. You can read [more](#) about jsonb datatype on the web, but it's the same as json with slight improvements. One of the main advantages of using jsonb over json in our case is that jsonb type data can make use of functions like `jsonb_set()` available in PostgreSQL which helps us in updating the key-value pairs in json (not entire json). If you wish, you can make use of JSON type column along with the `json_set()` function in MySQL.

```

1  @Entity
2  @Getter
3  @Setter
4  @ToString
5  // @TypeDefs({
6  //      @TypeDef(name = "jsonb", typeClass = JsonBinaryType.class)
7  // })
8  public class Student implements Serializable {
9
10     private int id;
11     private String name;
12     private int age;
13
14     // Below will map the city property of the Address class to city column in the Student
15     // database table. Its not needed, if names are same.
16     @EmbeddedId
17     @AttributeOverrides({
18         @AttributeOverride(name = "city", column = @Column(name = "city"))
19     })
20     private Address address;
21     private boolean adult;
22     // @Type(type = "jsonb")
23     @Column(columnDefinition = "jsonb")
24     private String favs;
25 }

```

Address has 2 variables: City and State. Both, City and State have been marked as composite Primary Key for the Student table using Embeddable and EmbeddedId

```

1  @Setter
2  @Getter
3  @Embeddable
4  @ToString
5  @AllArgsConstructor
6  @NoArgsConstructor
7  public class Address implements Serializable {
8
9     private String city;
10    private String state;
11 }

```

5. Create the JPA Repository class as mentioned below. Try to spend some time and analyse the custom queries that I have written. If required, use this PostgreSQL [documentation](#) in order to understand the functions like to_json() and json_set().

```

1  @Repository
2  public interface StudentJpaRepository extends JpaRepository<Student, Address> {
3
4      List<Student> findByName(String name);
5
6      @Query(value = "select * from student where favs->>?1 = ?2", nativeQuery = true)
7      List<Student> findByFavs(String key, String value);
8
9
10     // String query1 = "update student set favs= jsonb_set(favs , '{color}' , '\"darkgreen\"' ) where id=1";
11     // We dont need single quotes in java in case we want to fill values dynamically using @Param.
12     // jsonb_set takes column_name, key, value and boolean as arguments. Boolean signifies whether to create a new field if it already doesnt exist.
13     String query1 = "update student set favs= jsonb_set( favs , CAST(CONCAT('{', CAST(:key AS text), '}') as text[]), to_jsonb(CAST(:value AS text)), true) where id=:id";
14     @Modifying
15     @Query(value = query1, nativeQuery = true)
16     void updateStudentFavsKeyValueOrCreateIfNotExists(@Param("key") String key, @Param("value") String value, @Param("id") int id);
17 }

```

```

17
18
19 //String query2 = "update student set favs= jsonb_set(favs , '{attempts}' , '25')
    where id=1";
20 String query2 = "update student set favs= jsonb_set( favs , CAST(CONCAT('{', CAST(:
    key AS text), '}') as text[]), to_jsonb(CAST(:value AS int)), true) where id=:id"
    ;
21 @Modifying
22 @Query(value = query2, nativeQuery = true)
23 void updateStudentFavsKeyValueOrCreateIfNotExistsIntType(String key, int value, int
    id);
24
25 @Modifying
26 @Query(value = "update student set favs= ?1 where id=?2", nativeQuery = true)
27 void updateStudentFavs(String favs, int id);
28
29 int deleteByName(String name);
30
31 @Modifying
32 @Query(value = "UPDATE student SET favs = favs - CAST(?1 as text) WHERE id = CAST(?2
    as int);", nativeQuery = true)
33 void deleteFavsKeyValuePair(String keyName, int id);
34 }

```

6. Develop the service interface and implementation to encapsulate business logic. Here is a glimpse of it:

```

1 public interface StudentService {
2
3     Student addStudent(Student student) throws JsonProcessingException;
4
5     List<StudentRequest> getAllStudents() throws JsonProcessingException;
6
7     List<StudentRequest> getStudentsByName(String name) throws JsonProcessingException;
8
9     StudentRequest getStudentsById(String city, String state) throws
        JsonProcessingException;
10
11     List<StudentRequest> getStudentsByFavs(String key, String value) throws
        JsonProcessingException;
12
13     void updateStudentFavsKeyValueOrCreateIfNotExists(String key, String value, int id);
14
15     void updateStudentFavsKeyValueOrCreateIfNotExistsIntType(String key, int value, int
        id);
16
17     void updateStudentFavs(String favs, int id);
18
19     int deleteStudent(String name);
20
21     void deleteStudentFavsKeyValue(String keyName, int id);
22 }

```

```

1 @Service
2 @Slf4j
3 public class StudentServiceImpl implements StudentService {
4     private StudentJpaRepository studentJpaRepository;
5
6     @Autowired
7     StudentServiceImpl( StudentJpaRepository studentJpaRepository){
8         this.studentJpaRepository = studentJpaRepository;
9     }
10
11     @Override
12     public Student addStudent(Student student) {
13         return studentJpaRepository.save(student);

```



```

14     }
15
16     @Override
17     public List<StudentRequest> getAllStudents() throws JsonProcessingException {
18         List<Student> studentList = studentJpaRepository.findAll();
19         return getStudentRequestList(studentList);
20     }
21
22     @Override
23     public List<StudentRequest> getStudentsByName(String name) throws
24         JsonProcessingException {
25         List<Student> studentList = studentJpaRepository.findByName(name);
26         return getStudentRequestList(studentList);
27     }
28
29     @Override
30     public StudentRequest getStudentsById(String city, String state) throws
31         JsonProcessingException {
32         Optional<Student> studentOptional = studentJpaRepository.findById(new Address(
33             city, state));
34         if(!studentOptional.isPresent()){
35             return null;
36         }
37         Student s = studentOptional.get();
38         return getStudentRequestList(Collections.singletonList(s)).get(0);
39     }
40
41     @Override
42     public List<StudentRequest> getStudentsByFavs(String key, String value) throws
43         JsonProcessingException {
44         List<Student> studentList = studentJpaRepository.findByFavs(key, value);
45         if(studentList == null || studentList.isEmpty()){
46             return null;
47         }
48         List<StudentRequest> studentRequestList = getStudentRequestList(studentList);
49         return studentRequestList;
50     }
51
52     @Transactional
53     @Override
54     public void updateStudentFavsKeyValueOrCreateIfNotExists(String key, String value,
55         int id) {
56         log.info("Update request received by StudentServiceImpl for updating: {} to {}
57             for ID={}", key, value, id);
58         studentJpaRepository.updateStudentFavsKeyValueOrCreateIfNotExists(key, value, id)
59             ;
60     }
61
62     @Transactional
63     @Override
64     public void updateStudentFavsKeyValueOrCreateIfNotExistsIntType(String key, int value
65         , int id) {
66         log.info("Update request received by StudentServiceImpl for updating: {} to {}
67             for ID={}", key, value, id);
68         studentJpaRepository.updateStudentFavsKeyValueOrCreateIfNotExistsIntType(key,
69             value, id);
70     }
71
72     @Transactional
73     @Override
74     public void updateStudentFavs(String favs, int id) {
75         studentJpaRepository.updateStudentFavs(favs, id);
76     }
77
78     @Transactional

```

```

69     @Override
70     public int deleteStudent(String name) {
71         return studentJpaRepository.deleteByName(name);
72     }
73
74     @Transactional
75     @Override
76     public void deleteStudentFavsKeyValue(String keyName, int id) {
77         studentJpaRepository.deleteFavsKeyValuePair(keyName, id);
78     }
79
80     private List<StudentRequest> getStudentRequestList(List<Student> studentList) throws
      JsonProcessingException {
81         List<StudentRequest> studentRequestList = new ArrayList<>();
82         for (Student student: studentList) {
83             ObjectMapper objectMapper = new ObjectMapper();
84             StudentRequest studentRequest = new StudentRequest();
85             studentRequest.setId(student.getId());
86             studentRequest.setName(student.getName());
87             studentRequest.setAge(student.getAge());
88             studentRequest.setCity(student.getAddress().getCity());
89             studentRequest.setState(student.getAddress().getState());
90             studentRequest.setAdult(student.isAdult());
91             studentRequest.setFavs(objectMapper.readValue(student.getFavs(), HashMap.
              class));
92             studentRequestList.add(studentRequest);
93         }
94         return studentRequestList;
95     }
96 }

```

7. Lastly, Expose REST endpoints for CRUD operations using a Controller class. Here is a glimpse of it:

```

1  @RestController
2  @Slf4j
3  public class StudentController {
4      private RestTemplate restTemplate = new RestTemplate();
5      private StudentService studentService;
6
7      @Autowired
8      public StudentController(StudentService studentService) {
9          this.studentService = studentService;
10     }
11
12     @PostMapping("/student")
13     private Student addStudent(@RequestBody StudentRequest studentReq) throws
      JsonProcessingException {
14         log.info("Student to be added is {}", studentReq);
15
16         Student student = getStudent(studentReq);
17         ObjectMapper objectMapper = new ObjectMapper();
18         student.setFavs(objectMapper.writeValueAsString(studentReq.getFavs()));
19         return studentService.addStudent(student);
20     }
21
22     @GetMapping("/student")
23     private List<StudentRequest> getAllStudent() throws Exception {
24         return studentService.getAllStudents();
25     }
26
27     //Name is one of the columns in our table. So we can use it to query.
28     @GetMapping("/student/filterByName/{name}")
29     private List<StudentRequest> getStudentsByName(@PathVariable String name) throws
      JsonProcessingException {
30         return studentService.getStudentsByName(name);

```

```

31     }
32
33     //Our table has a Primary Key of City, State. So we need to pass both city and state
    to get the student by Id.
34     @GetMapping("/student/filterById")
35     private StudentRequest getStudentsById(@RequestParam String city, @RequestParam
    String state) throws JsonProcessingException {
36         return studentService.getStudentsById(city, state);
37     }
38
39     //favs is a jsonb column in our table. We may want to filter favs by a specific key
    and value.
40     @GetMapping("/student/filterByJsonColumnKeyValue/{key}/{value}")
41     private List<StudentRequest> getStudentsByFavs(@PathVariable String key,
    @PathVariable String value) throws JsonProcessingException {
42         return studentService.getStudentsByFavs(key, value);
43     }
44
45     //update a specific key in json column favs to a new String type value. If key doesnt
    exist, create it.
46     @PatchMapping("/student/updateFavs/{key}/{value}/{id}")
47     private void updateStudentFavsKeyValueOrCreateIfNotExists(@PathVariable String key,
    @PathVariable String value, @PathVariable int id) {
48         studentService.updateStudentFavsKeyValueOrCreateIfNotExists(key, value, id);
49     }
50
51     //update a specific key in json column favs to a new Int type value. If key doesnt
    exist, create it.
52     @PatchMapping("/student/updateFavsIntType/{key}/{value}/{id}")
53     private void updateStudentFavsKeyValueOrCreateIfNotExistsIntType(@PathVariable String
    key, @PathVariable int value, @PathVariable int id) {
54         studentService.updateStudentFavsKeyValueOrCreateIfNotExistsIntType(key, value, id
    );
55     }
56
57     //Update the json column favs entry entirely. This will replace the existing favs
    column entry with the new one for specified ID.
58     @PatchMapping("/student")
59     private void updateStudent(@RequestBody StudentRequest studentReq) throws
    JsonProcessingException {
60         Student student = getStudent(studentReq);
61         ObjectMapper objectMapper = new ObjectMapper();
62         student.setFavs(objectMapper.writeValueAsString(studentReq.getFavs()));
63         studentService.updateStudentFavs(student.getFavs(), student.getId());
64     }
65
66     //Delete a student record by name.
67     @DeleteMapping("/student/{name}")
68     private int deleteStudent(@PathVariable String name) {
69         return studentService.deleteStudent(name);
70     }
71
72     //Our table has json column. We can remove specific key-value pair from the json
    column.
73     @DeleteMapping("/student/keyValue/{key}/{id}")
74     private void deleteStudentFavsKeyValue(@PathVariable String key, @PathVariable int id
    ) {
75         studentService.deleteStudentFavsKeyValue(key, id);
76     }
77
78     private static Student getStudent(StudentRequest studentReq) {
79         Student student = new Student();
80         student.setName(studentReq.getName());
81         student.setAge(studentReq.getAge());
82         Address address = new Address();

```

```
83     address.setCity(studentReq.getCity());
84     address.setState(studentReq.getState());
85
86     student.setAddress(address);
87
88     student.setId(studentReq.getId());
89     student.setAdult(studentReq.isAdult());
90     return student;
91 }
92
93 }
```

8. In order to take requests from user and submit the request to Controller, we have to create a Domain class called StudentRequest. Let's Define a `StudentRequest` class that maps incoming payloads to our domain model.

```
1  @Getter
2  @Setter
3  @ToString
4  @JsonIgnoreProperties(ignoreUnknown = true)
5  public class StudentRequest {
6
7      private int id;
8      private String name;
9      private int age;
10     private String city;
11     private String state;
12     private boolean adult;
13     private Map<String, Object> favs;
14 }
```

That's it. We are all set.

Running and Testing the Application

Now, Build the Spring Boot application:

```
mvn clean install
```

Then, Run the Spring Boot application:

```
mvn spring-boot:run
```

With both PostgreSQL and the Spring Boot app running, we can now test the APIs using Postman.

Download the Postman collection JSON file from my [Github](#). Open Postman and import the Postman collection named "JPAwithJSON.postman_collection.json".

Try to hit Create a new Student API, Get all Students API, etc

This demonstrates that modern RDBMS platforms like PostgreSQL are more than capable of managing semi-structured JSON data without forcing a move to NoSQL solutions.

If you found this article helpful, feel free to share it and don't forget to connect with me on [Linkedin](#).

› [Back to the Table of Contents](#) ‹

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



JCON 2026
EUROPE
COLOGNE

📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Java 26 + Spring Boot 4: From Certification Skills to Production-Ready Apps

Microservices

SpringBoot

Java26

Author: Vauban Vincent

Vincent Vauban is a Fullstack software developer with over 15 years of experience in Java, Spring, JavaScript, React, and Angular. As a vExpert and Udemy instructor, he brings his expertise to a variety of industries, including banking, insurance, retail, and the public sector.



LinkedIn

If you already feel “fluent” in Java and Spring, this article won’t teach you how to write your first controller. Its real goal is different: show how a certification-style syllabus can act as a *practical* learning map that turns day-to-day “it works on my machine” development into production-ready engineering habits.

In other words: certification prep is not about “tricks and traps”. It’s a structured path through the concepts that repeatedly matter in real systems; core container mechanics, data access, web APIs, testing discipline, security defaults, and operational visibility. Then, once you have that foundation, you top it up with focused micro-learning for the newest platform features (today: Spring Boot 4 and Java 26).

Spring Boot 4.0.0 (released **November 20, 2025**) is positioned as a “new generation” built on Spring Framework 7, emphasizing modularization, null-safety with JSpecify, and new first-class capabilities such as API Versioning and HTTP Service Clients. It also offers first-class support for Java 25 while retaining Java 17 compatibility.

Java 26 is scheduled for **General Availability on March 17, 2026**, and (as of its rampdown) targets a concrete set of platform improvements: from HTTP/3 support and G1 throughput changes to structured concurrency (preview) and the Vector API (incubator).

What follows is a guided bridge between “exam domains” (Core, Data, MVC, Testing, Security, Spring Boot) and the production concerns you face on microservices: correctness, evolvability, performance, security, and operability.

“Freestyle” Learning vs. Guided Learning

Many developers learn a new framework by poking at it randomly:

- copy a snippet from a blog,
- tweak until green,
- ship,
- repeat until something breaks.

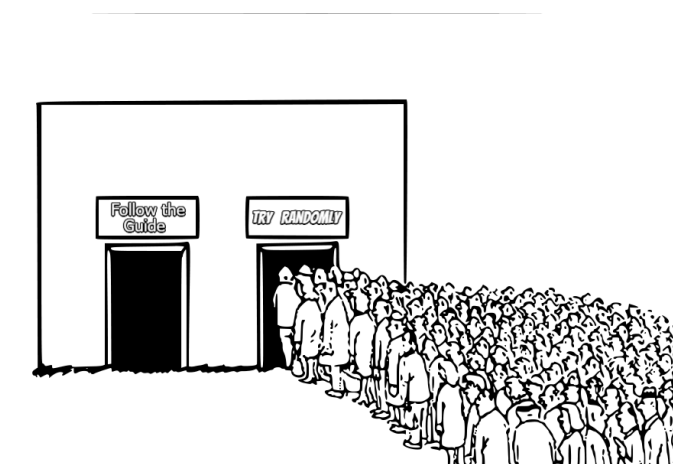


Abb. 4: Following the guide (certification) versus trying randomly

That *works*, but it’s noisy. You end up with blind spots in areas you don’t naturally encounter during feature development (observability, transaction boundaries, edge-case serialization, security defaults, startup behavior, etc.).

A certification syllabus acts like an opinionated checklist of “things that reliably matter.” Even if you never take the exam, preparing *as if you would* creates a disciplined pass over the foundation.

The point is not that the mapped path is “easy”. It’s that you know *why* you’re studying something, and you can see what you’re skipping.

Certification as a Map: You Don’t Conquer Territory Without One

Think of a modern Spring microservice. The hardest parts are rarely “how to create a bean”. They’re usually:

- diagnosing production behavior,
- evolving APIs without breaking clients,
- keeping security tight while still delivering features,
- handling data consistency under concurrency,
- and keeping the system observable at 2 a.m.

A typical “lambda developer” (fast, pragmatic, shipping features) can miss advanced but essential topics simply because they don’t show up in the happy path.

A classic example: **observability**. Spring Boot’s Actuator and its ecosystem (metrics, health, tracing) are not “optional fancy extras”; they’re a maintenance strategy. Certification domains often force you to study these pieces, which translates directly into production confidence.

Spring Boot 4 doubles down on this “operability as a first-class feature”, for example with an OpenTelemetry starter that brings dependencies and auto-configures the OpenTelemetry SDK for exporting metrics/traces over OTLP.

The Syllabus Is Not “Tricks”: It Mirrors Business Needs

A good syllabus is a compressed version of real-world requirements.

Java Certification-Style Domains

- Object-oriented concepts (design, encapsulation, polymorphism)
- Arrays, collections, generics
- I/O and serialization boundaries
- Packaging and deployment basics
- Streams and lambdas (data transformations, readability, performance tradeoffs)
- Date/time and formatting (bugs love timezones)
- Localization (even if you “never need it”... until you do)
- Exceptions (stability under failure)
- Concurrency (correctness under load)

Spring Certification-Style Domains

- Spring Core and configuration
- Testing
- Spring AOP (cross-cutting concerns done safely)
- Data access / transactions
- MVC (web APIs, validation, error contracts)
- Security
- Actuator / monitoring / operations

This list is not a museum of obscure features. It’s the *shelf* of reliable tools that reduce risk:

- fewer home-grown frameworks,
- fewer hidden edge cases,
- fewer inconsistent patterns between services.

And it changes how you think. After a structured pass, you become “tech-wise”: you can look at a requirement and pick the right feature *from the shelf* instead of reinventing it under deadline pressure.

The Inevitable Lag: Certifications Trail Releases, and That’s Normal

Certifications are snapshots. New versions ship faster than exam updates, because question writing and validation take time.

The good news: once you have the foundation, catching up is cheap. You do targeted micro-learning: read a release note, build a tiny lab, and connect it to what you already know.

Java: From a Stable Foundation to Java 25 and Java 26

Oracle's Java language changes summary for Java 25 highlights features such as Module Import Declarations, Compact Source Files and Instance `main` Methods, and Flexible Constructor Bodies, along with ongoing pattern-matching evolution.

Then Java 26 (GA planned March 17, 2026) focuses on concrete improvements and previews such as:

- **JEP 500** Prepare to Make Final Mean Final
- **JEP 516** Ahead-of-Time Object Caching with Any GC
- **JEP 517** HTTP/3 for the HTTP Client API
- **JEP 522** G1 GC throughput improvements
- **JEP 525** Structured Concurrency (preview)
- **JEP 526** Lazy Constants (preview)
- **JEP 529** Vector API (incubator)
- ...and more.

Spring Boot 4: Focus Areas That Matter in Production

Spring Boot 4.0.0's announcement calls out: modularization, JSpecify null-safety improvements, first-class Java 25 support (while keeping Java 17 compatibility), and new first-class support for API Versioning and HTTP Service Clients.

The release notes add operationally relevant changes, including an OpenTelemetry starter and more.

The strategy is simple:

1. Build the mental model via the “map” (syllabus).
2. Add a short “delta” lab for the newest releases.

Now let's map domains to production.

Domain-by-Domain: From Exam Skill to Production-Ready Microservice

Spring Core: The Container Is Your Runtime Contract

Certification-style skills

- Bean lifecycle, scopes, proxies
- Dependency injection patterns
- Configuration: `@ConfigurationProperties`, profiles, conditional beans
- Resource loading, environment, property precedence

Production payoff

- predictable startup and wiring (no surprise beans in prod)
- consistent configuration strategy (safe defaults, overridable per environment)
- “feature flags” via conditions without spaghetti

Boot 4 angle: modularization changes how you *ship* the container

A modularized Boot codebase means smaller, more focused jars and clearer dependency surfaces. That's not just “neat”; it makes it easier to reason about what's on your classpath and why, which reduces upgrade friction and accidental coupling.

Practical checklist

- Prefer `@ConfigurationProperties` for structured config (and validation)
- Keep conditional wiring explicit: `@ConditionalOnProperty`, `@Profile`
- Make startup failures loud: fail fast on missing critical config

Data: Transactions, Consistency, and the “Don’t DIY JDBC Under Pressure” Rule

Certification-style skills

- Repository patterns
- Transaction boundaries ([@Transactional](#))
- Isolation/propagation basics
- Mapping: JPA, JDBC, or Spring Data abstractions

Production payoff

- fewer “it worked in tests” data bugs
- consistent retry and idempotency strategies
- easier migrations and observability at the data layer

The key idea: production reliability often comes from choosing well-tested abstractions instead of rolling your own ad-hoc data access. Not because DIY is impossible, but because *you will forget a corner case at 6 p.m. on a Friday.*

Practical checklist

- Define transaction boundaries at service methods, not randomly in repositories
- Make writes idempotent where possible
- Log slow queries and expose DB health as an SLO input

MVC / Web: API Contracts, Evolvability, and Versioning

Certification-style skills

- Controller design, content negotiation
- Validation, error handling, [@ControllerAdvice](#)
- Serialization concerns (Jackson)
- REST semantics and status codes

Production payoff

- stable client contracts
- controlled evolution of endpoints
- fewer “mystery 500” incidents

Boot 4 angle: built-in API Versioning

Spring Boot 4 adds auto-configuration for API Versioning for Spring MVC and WebFlux, configurable via [spring.mvc.apiversion.*](#) / [spring.webflux.apiversion.*](#) and extensible with resolver/parser beans.

That matters because versioning is often implemented as “random conventions” across services. When the framework gives you a standard mechanism, your teams can converge on one approach.

Boot 4 angle: HTTP Service Clients

Boot 4 includes auto-configuration support for HTTP Service Clients: annotate plain Java interfaces with [@HttpExchange](#) and let Spring generate implementations.

This reduces “stringly-typed HTTP” scattered across codebases and encourages typed, testable client contracts; exactly what you want in microservice-to-microservice calls.

Mini example (conceptual)

- Version your controller paths or headers via the API versioning facilities.
- Define typed outbound clients via [@HttpExchange](#) interfaces.
- In tests, mock the interface rather than mocking low-level HTTP.

Testing: The Discipline That Pays Rent Every Day

Certification-style skills

- Unit tests with Spring support where needed
- Slice tests (web, data)
- Context caching and test configuration
- Mocking boundaries, test profiles

Production payoff

- upgrades without fear
- faster incident fixes (reproduce + regression)
- less “works locally” drift

The certification mindset helps because it forces breadth:

- you test controllers the Spring way,
- you test repositories the Spring way,
- you test configuration the Spring way.

You stop guessing what the framework is doing.

Practical checklist

- Keep most tests “thin Spring” (unit tests) and a smaller set “real Spring”
- Test error contracts and validation paths (not just 200 OK)
- Write one test that verifies observability wiring (Actuator endpoints / tracing presence)

Security: Defaults, Not Add-Ons

Certification-style skills

- Authentication vs authorization
- Filter chain basics
- Method security
- OAuth2 / JWT concepts
- CSRF and session vs stateless

Production payoff

- fewer accidental exposures
- consistent policy enforcement
- easier audits and incident response

A common anti-pattern is adding security “at the end”. Certification prep makes you study the model early, so you design endpoints and error handling with security in mind.

Practical checklist

- Secure by default: deny unless explicitly allowed
- Separate “public” and “internal” endpoints
- Treat configuration and secrets as part of the threat model
- Add security tests: unauthenticated, unauthorized, and role boundary checks

Spring Boot: Operations, Visibility, and the Difference Between “Running” and “Operable”

Certification-style skills

- Auto-configuration mental model
- Externalized configuration

- Actuator: health, metrics, info
- Logging, tracing basics
- Packaging and deployment basics

Production payoff

- predictable deployments
- actionable metrics and health checks
- faster diagnosis when things go wrong

Boot 4 angle: OpenTelemetry starter

The Boot 4 release notes introduce `spring-boot-starter-opentelemetry`, providing dependencies to export metrics/traces over OTLP and auto-configuring the OpenTelemetry SDK.

This is the kind of topic many developers skip until the first production incident. But it's exactly the sort of capability a certification-aligned study path makes you notice early.

Practical checklist

- Define readiness/liveness checks aligned with dependencies
- Treat metrics as part of the API of the service
- Standardize logging fields (trace ids, request ids)
- Make your service observable before it becomes critical

Where Java 26 Helps Spring Boot 4 Microservices

Spring Boot is not “just a framework”; it's a runtime hosted by the JVM. JVM-level improvements often translate directly into operational wins.

Here are a few Java 26 items (as targeted for JDK 26) and why they matter in microservices.

HTTP/3 for the HTTP Client API (JEP 517)

If your service uses the JDK HTTP Client for outbound calls, HTTP/3 support can improve latency characteristics and behavior on lossy networks. Even if you rely on other clients today, the platform evolving matters: it shapes the ecosystem and the defaults teams gravitate toward.

G1 GC Throughput Improvements (JEP 522) + AOT Object Caching (JEP 516)

Latency and throughput are not abstract concerns when your autoscaler watches p95 and CPU. GC and startup/runtime optimizations can translate into lower tail latency, fewer pods, and reduced cloud cost; without you touching application code.

Structured Concurrency (Preview) (JEP 525)

Many microservice operations are “fan-out/fan-in”: call two downstream services, combine, return. Structured concurrency pushes toward safer cancellation and lifetime management of concurrent tasks. Even as a preview, it's a signal of where robust concurrent code is heading.

Vector API (Incubator) (JEP 529)

Not every service needs SIMD acceleration; but when you do heavy data processing (recommendations, analytics, ML-adjacent workloads), using the platform's vector support can deliver speedups while staying in Java.

Prepare to Make Final Mean Final (JEP 500)

This one is about semantics and safety. Microservices are long-lived systems; you want the language and JVM to be strict about correctness assumptions. Tighter guarantees reduce the chance that clever hacks become production landmines.

Putting It Together: The “Foundation + Delta” Learning Loop

A realistic plan for a busy developer looks like this:

1. **Foundation pass (the map):** follow a syllabus-like structure across domains.
 - Core container + configuration
 - Data + transactions
 - Web + validation + errors
 - Testing strategy
 - Security model
 - Boot/Actuator/operations
2. **Delta pass (the newest releases):**
 - Read Spring Boot 4 highlights and pick one feature to lab (API versioning, HTTP service clients, OTel starter).
 - Skim Java language changes (e.g., Java 25 changes) and then review JDK 26 targets to understand upcoming platform shifts.
3. **Production alignment:**
 - Use the new features to reduce risk:
 - standard versioning instead of ad-hoc conventions,
 - typed HTTP clients instead of scattered request builders,
 - standard telemetry wiring instead of custom glue.

This loop keeps you current without turning your life into an endless upgrade chase.

Why This Changes Consulting Confidence (and Team Trust)

There’s a moment many developers recognize: a business requirement arrives fuzzy, the constraints are unclear, and you feel uncertainty about the “best” approach. In those moments, a structured study pass acts like a stabilizer:

- you know what options exist in the platform,
- you know which tradeoffs you’re accepting,
- and you can explain your choice.

Even when you move fast, you’re not doing freestyle. Your teammates sense that difference immediately; because decisions become consistent, and solutions become easier to review and maintain.

And this applies beyond tech stacks. The same “map” effect exists in platform and process certifications too: Kubernetes, Kafka, Scrum... different domains, same outcome: reduced improvisation under pressure.

Closing: Client Satisfaction Is Still the Ultimate Metric

At the end of the day, production-ready engineering is not about being able to recite annotations or memorize APIs. It’s about delivering systems that:

- stay secure as CVEs appear,
- get faster and cheaper as platforms improve,
- remain maintainable for the next team,
- and are diagnosable when reality disagrees with your assumptions.

Certifications are not magic. But a certification-aligned syllabus is a surprisingly effective way to build a reliable foundation; and that foundation makes it easier to absorb the newest platform improvements, like Spring Boot 4's operational focus and Java 26's JVM-level advances.

If the result is a team that ships with confidence and a client who feels the difference, that's not a side effect.

That's the point.

› [Back to the Table of Contents](#) ‹

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



From Cron to Carbon & Cost-Aware: Modernizing Java Background Jobs

API

Frameworks

JobRunr

Author: Ronald Dehuysser

He is a software architect from Belgium and the creator of JobRunr. Seamlessly integrating with Spring Boot, Micronaut, and Quarkus, JobRunr is my solution for efficient distributed background job processing. When he is not working on open-source software, you can find him either rock climbing or sipping a fine Belgian Duvel beer.



LinkedIn

Introduction

Thanks to the ever-evolving JVM ecosystem, executing a simple fire-and-forget job in a separate (virtual or native) thread is not a big deal nowadays. Keeping tabs on the job, executing twenty of them at the same time to split up work only to merge the results and then fan out with a second subtask is an entirely different matter. Factor in resistance to failures and you're in over your head before you know it.

Luckily, a few battle-tested libraries are floating out there to help wrangle & master the job scheduling problem, of which Quartz is no doubt the most famous—but also one of the most verbose. Of course, there is no such thing as an ideal library. But with the arrival of Java 8's lambdas, wouldn't it be great to schedule a job simply by creating a lambda and feeding that into the scheduling system to let it do its thing? And wouldn't it be even better if that library wasn't picky about your database, automatically handled retries when things break, and came with a built-in dashboard so you could actually see what's going on?

That is exactly how JobRunr came to be.

Planning One-Time and Cron Jobs

The simplest possible way to enqueue a job is by simply passing in a lambda to the `BackgroundJob`:

```
1 import org.jobrunr.scheduling.BackgroundJob;
2 BackgroundJob.enqueue(() -> pdfService.generate("id-1234"));
```

All you need is the JobRunr JAR and your preferred JSON serializer, such as Jackson.

JobRunr handles the configuration automatically, so you can simply add the dependency to your favorite build tool using groupid: `org.jobrunr`, artifactId: `jobrunr`, version 8.4.1. (which is the latest at the time of writing).

JobRunr analyzes the lambda and its contents and persists the metadata to the SQL or NoSQL database for any background job server to pick up, meaning this is a non-blocking call and arguments get serialized as well. As the job details (class, method and arguments) are stored in the database, the job can easily be picked up again in case of a system failure.

Of course, most real-life examples are a bit more complex compared to the basic fire-and-forget background job above. With JobRunr, you can also schedule jobs in the future or plan recurring jobs using the typical cron notation using the `schedule` methods:

```
1 BackgroundJob.schedule(now().plusHours(1), () -> pdfService.generate());
2 BackgroundJob.scheduleRecurrently("0 0 1 * *", () -> myService.doOnceAMonth());
```

To help with testability and maintainability of your code, you can also inject a `JobScheduler` instead of relying on the default background job provider using your favourite Dependency Injection (DI) framework. JobRunr fully integrates with Spring, Quarkus, and Micronaut:

```
1 public class InvoiceController {
2     @Inject
3     private JobScheduler scheduler;
4     @Inject
5     private PDFService pdfService;
6
7     @Job(name = "PDF generation for invoice %0", retries = 10)
8     public void generatePDF(String id) {
9         scheduler.enqueue(() -> pdfService.generate(id));
10    }
11
12    @Job(name = "PDF generation monthly overview", retries = 2)
13    @Recurring(id = "monthly-recurring-overview", cron = "0 0 1 * *")
14    public void generateMonthlyOverview() {
15        scheduler.enqueue(() -> pdfService.generateOverview());
16    }
17 }
```

The `@Job` annotation allows for easy job property manipulation. Alternatively, you can rely on the `JobBuilder` class to set them in accordance with the well-known builder pattern. Automatic integration in Spring-managed transactions is also

possible in JobRunr. We have prepared example repositories for each DI framework: see [1] for Spring and Quarkus getting-started projects. As you can see in the code above, scheduling jobs recurrently in these frameworks is as easy as using any other annotation.

As for where exactly JobRunr stores information on the jobs it processes, this is done by configuring a [StorageProvider](#) that acts as the bridge between JobRunr and the database. All major SQL and NoSQL databases are supported, from PostgreSQL and MariaDB to MongoDB and even Amazon's DocumentDB. During the bootstrapping procedure of JobRunr, it is possible to configure your database:

```
1 JobRunr.configure()
2   .useStorageProvider(new DefaultSqlStorageProvider(dataSource))
3   .useBackgroundJobServer()
4   .initialize();
```

By default, JobRunr will create the necessary tables needed for job metadata information itself, but in case you have a custom database migration system already in place, it is of course possible to generate and apply the scripts yourself. Note that the background server configuration can be expanded upon, for example to specify worker counts, the polling interval, server tags (see below), and the thread type. If you're rocking Java 21, JobRunr will select virtual threads for its job worker pools.

From Cron to Carbon Aware Job Processing

The previous examples showcase easy planning of jobs using a cron expression. Such an expression is a great way to postpone CPU-intensive work to a later moment, usually during the night, to avoid clogging up threads that reduce the HTTP responsiveness.

But we can do better. Instead of just planning a job at 3 AM using `0 3 * * *`, why don't we plan the job during a given interval? That nightly job kicking PDF generation in gear is not bound to exactly 3 AM; as long as the generation is done before the next working day at say 9 AM we're good. In JobRunr, a custom extended cron notation easily solves this problem: plan at `0 3 * * * [PT1H/PT4H]`. This means "schedule this job somewhere between 1 hour before 3 AM and 4 hours after 3 AM".

But how do you pinpoint the optimal execution window? The goal is to run jobs precisely when grid carbon intensity bottoms out, ensuring the least amount of CO2 is generated.

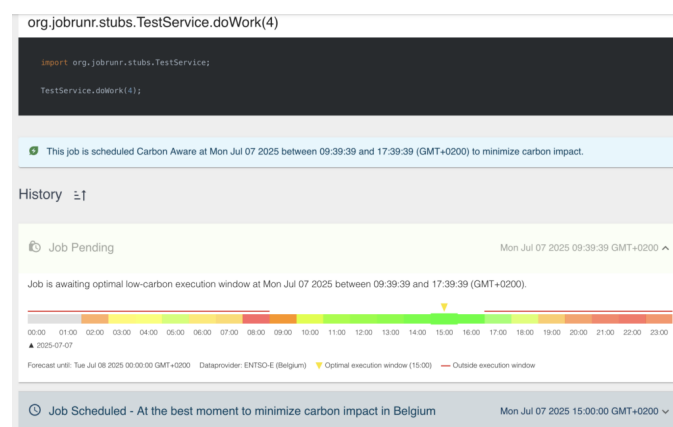


Abb. 5: An example carbon aware daily recurring job with a margin between 9h and 16h, with the local time being 9h. The job was scheduled at 15h to minimize carbon impact.

This is exactly what JobRunr's Carbon Aware Job Processing capability does. JobRunr consults energy forecast services such as ENTSO-E in Europe to accurately predict when the energy prices will be the lowest. The strong relationship between energy pricing and carbon intensity means relying on JobRunr's Carbon Aware system effectively decreases your systems' carbon footprint!

Configuration is extremely simple: (1) tell JobRunr which region your servers are in and (2) provide a margin in your cron expression as stated above.

Here's an example how to enable Carbon Aware Job Processing using the Fluent API:

```
1 JobRunr
2     .configure()
3     .useBackgroundJobServer(
4         usingStandardBackgroundJobServerConfiguration()
5         .andCarbonAwareJobProcessingConfiguration(
6             usingStandardCarbonAwareJobProcessingConfiguration()
7             .andAreaCode("DE"))
8     .initialize();
```

Users of frameworks such as Spring Boot will be happy to hear that for them configuration is even easier using just two properties:

```
1 jobrunr.background-job-server.carbon-aware-job-processing.enabled=true
2 jobrunr.background-job-server.carbon-aware-job-processing.area-code=BE
```

After that it's only a matter of changing the cron expressions of non-time-critical jobs to include a margin. The bigger the margin, the bigger the potential environmental and financial gain. Fire-and-forget jobs can also be fired "somewhere in the near future" by providing a margin using the simple helper class [CarbonAware](#):

```
1
2 BackgroundJob.schedule(CarbonAware.between(now, now.plus(5, HOURS)),
3 () -> myService.sendNewlyRegisteredEmail());
```

Another way of creating Carbon Aware jobs is to use `.before()` to tell JobRunr to schedule the job before a certain point in time *in the future*. No need to worry if the provided interval is too small: in that case, the job will still be scheduled, either at the preferred time or immediately.

Keeping It Cool if Things Go Wrong

As you might have noticed in the previous section, JobRunr comes with a few bells and whistles to automatically retry jobs that failed for some reason.

For example, consider the following method: `notifyService.sendToSlack()`.

What happens if the HTTP connection times out? Then JobRunr will retry the job, according to built-in exponential backoff settings, or according to a fully customisable retry policy (in Pro).

But what happens if the job fails after 10 tries and 10 is the maximum amount of retries? For example, if the internet connection fails or the URL is misconfigured. In that case, the job is not lost to a black hole. Instead, it will be given the **FAILED** state. You can inspect all failed jobs in the integrated UI dashboard. The dashboard allows for an easy overview of all scheduled, processing, successfully completed and failed jobs where you can manually re-trigger them if you wish to do so. Job metadata is also visible there, including detailed information as to why exactly a specific job has failed.

Even if you already have a comprehensive observability stack, the JobRunr dashboard remains a vital tool for operations, specifically for requeuing failed jobs once you've resolved the root cause (like a restored internet connection).

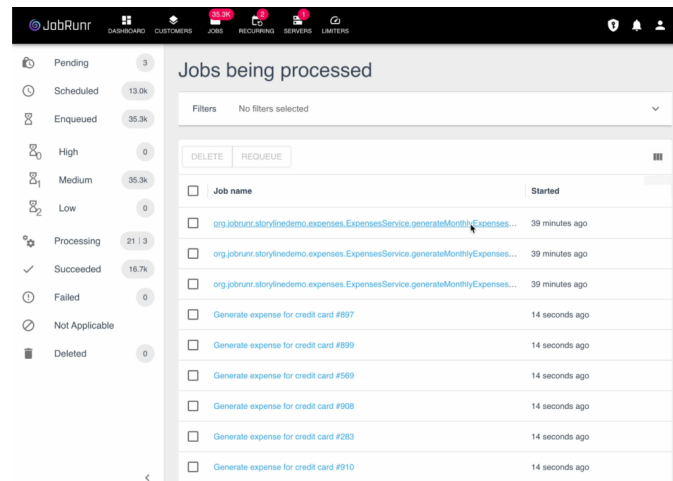


Abb. 6: In your JobRunr dashboard you have an overview of all your pending, scheduled, processing, successful and failed jobs.

For monitoring and alerting, JobRunr integrates seamlessly with Micrometer, while JobRunr Pro adds support for platforms like Jaeger, Honeycomb, and New Relic. By exporting metrics such as job throughput and average duration to tools like Prometheus, linking job failures to your existing alerting system becomes peanuts.

Configuring all this may sound daunting but really is not, especially when using a DI framework.

Just add these lines to your `application.properties`:

```
1 # jaeger integration
2 jobrunr.dashboard.integrations.observability.jaeger.root-url=http://localhost:16686/
3 # MicroMeter integration
4 jobrunr.jobs.metrics.micrometer-observability.enabled=true
5 # OpenTelemetry integration
6 jobrunr.jobs.metrics.otel-observability.enabled=true
```

Re-triggering failed jobs might still get you into trouble if they are not written in an idempotent way and are not what we call “re-entrant”: a job should be able to be rerun after being interrupted without this resulting in inconsistent or duplicate state.

For instance, suppose you want to charge a customer through an external API. In your job, before calling `api.charge()`, a guard should be added: `if(api.hasNotChargedFor(x))`. That way, if the job crashes after charging, it will not charge the customer again.

Unfortunately, not all external APIs offer these kinds of idempotency safeguard checks. In that case, you can rely on JobRunr’s metadata feature to save your progress:

```
1 public void processOrder(Long userId, Long orderId, JobContext jobContext) {
2     var user = userService.getUser(userId);
3     var order = orderService.getOrder(orderId);
4
5     jobContext.runStepOnce("charge-order", () -> api.charge(orderId, order.getTotalAmount()));
6
7     // no guard needed: only this can go wrong and will be retried
8     emailService.send(user.getEmail(), renderOrder(orderId));
9 }
```

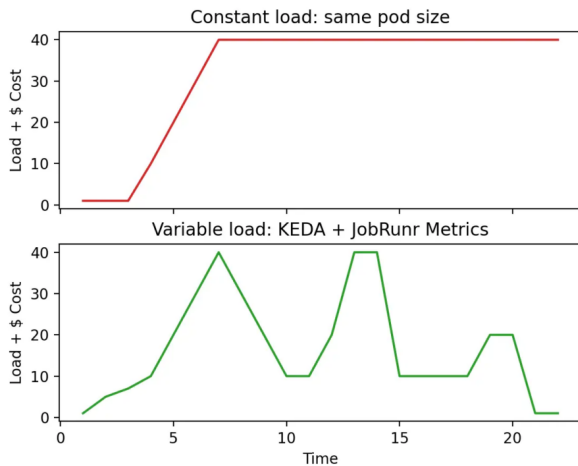
You can read more about idempotency and JobRunr in [2].

Scaling Up with Ease

What if you have a hundred jobs coming in and clogging the server, gradually diminishing the hope of finishing all jobs on time? Horizontal scaling is as easy as spinning up more instances of your application configured to be a background job

server. Since the database is the shared interface, proper optimistic locking mechanisms ensure none of the clusters are processing the same job.

Background servers are aware of each other (again, through the database as the shared instance) and elect a new master in case the previous one goes down. The master server is the one responsible for managing recurring jobs and other various cleanup tasks.



By combining tools such as KEDA with JobRunr metric output, it is possible to automatically scale up your Kubernetes-driven application. A few pitfalls such as graceful termination of long-running jobs need to be taken into account. More information and configuration tips on how to realise this can be found in our guide at [3].

In case horizontal scaling does not suit your needs, you can make use of JobRunr Pro's many queueing configurations. That way, you can queue up jobs in up to five different priority queues making sure there's always a "fast line" available for high priority ones that need to get through. For multi-tenant applications, a load balancing system called *dynamic queues* allows for round robin or weighted round robin dynamic queues ensures that jobs from all tenants are processed, even if one is much more spammy than the other.

Conclusion

In this article, we have showcased the capabilities of JobRunr, a modern asynchronous job scheduling library that, compared to golden standards such as Quartz, manages to be both easy to use and fully-featured.

With JobRunr, creating a job is just a matter of passing in a lambda, whether it is a fire-and-forget job, a job in the near future, or a recurring job. On top of that, JobRunr integrates well into many of the Java frameworks and toolkits you probably already rely on, such as Spring Boot and Quarkus. As it does not require a new system component in your infrastructure, it is also really easy to have approval of your infrastructure team.

JobRunr's Carbon Aware Job Processing system trivializes the scheduling of a job in such a way that your systems' carbon footprint will be reduced. If things go wrong, the dashboard can visualise and help pinpoint the problem in no-time, and if things get hot, JobRunr comes with capabilities to easily scale up (or down, for that matter). We hope that anyone working on the JVM will enjoy working with JobRunr as much as we enjoyed making it.

You can find more information on JobRunr at <https://www.jobrunr.io/>.

References

- [1]: JobRunr example projects at GitHub: [example-spring](#) and [example-quarkus](#)
- [2]: [JobRunr Kubernetes autoscaling](#)
- [3]: Idempotence in [Java Job Scheduling](#)

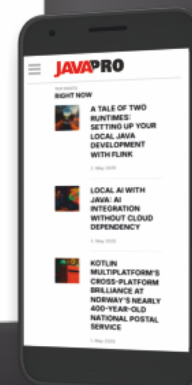
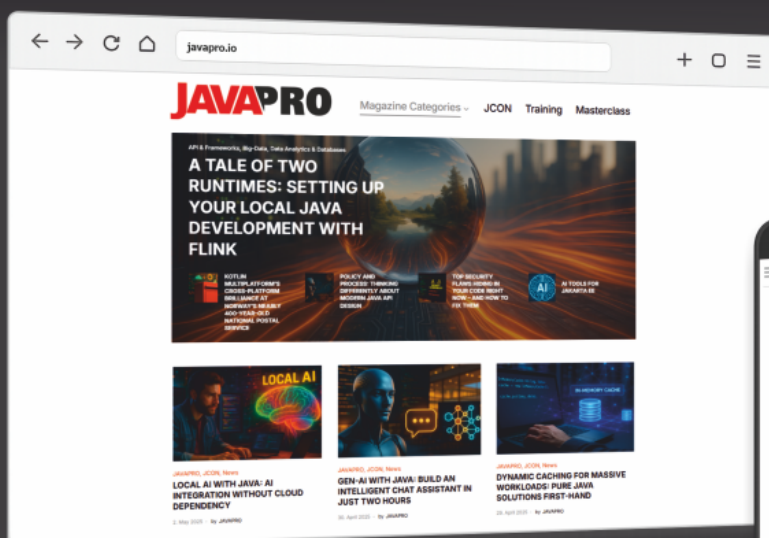
Abstract

JobRunr is a modern free and open source library to monitor and schedule (recurring) asynchronous jobs. It leverages Java 8's lambdas to ease the pain of quickly firing off jobs and comes with a UI dashboard to follow up and manually reschedule those jobs that seem to take longer than expected. On top of that, JobRunr is able to optimize the scheduling of jobs to generate the least amount of CO2. In this article, let's investigate how this all works and how to get started.

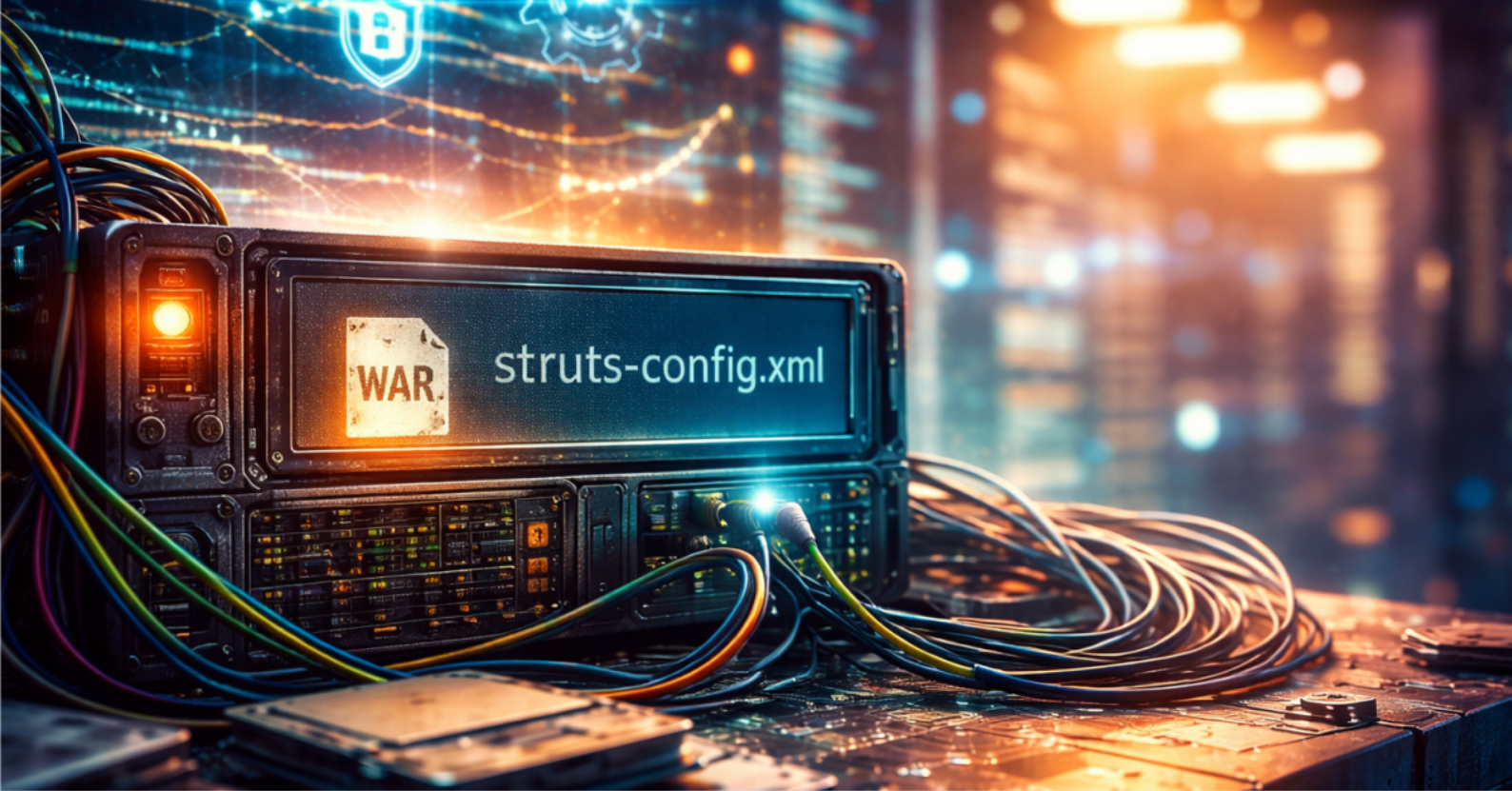
› [Back to the Table of Contents](#) ‹

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



Why Struts 1.3 Still Matters in 2026

EOL # Moderisation # Struts

Author: Steve Poole

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.

Want to Dive Deeper?

Steve Poole is a speaker at JCON (April 20-23)! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn



What You'll Learn

This article is the first in a series that explores why Struts 1 still shows up (quietly but persistently) in enterprise systems in 2026, and what that means for teams trying to keep the lights on.

We look at:

- **The Discovery Effect:** How stable, forgotten Struts apps resurface during audits, migrations, and security scans.
- **Where the Friction Really Is:** The talent shortage, the tooling disconnect, and the Jakarta EE shift: all colliding to make legacy upkeep harder than it should be.
- **Modernisation Without the Fantasy:** Why the rewrite-or-risk framing misleads teams, and how approaches like embedding Struts in Spring Boot create workable transition paths.
- **Buying Breathing Room:** How extended support gives organisations time to avoid crisis-driven decision-making.

Rediscovery Day: When Struts Finds You

There's a moment every long-established engineering organisation experiences eventually. Someone in security or infrastructure runs an inventory sweep, or tries to update a container image, and stumbles across a WAR file from another lifetime. It boots. It works. And sitting right in the middle of it is a `struts-config.xml`, exactly where you hoped it wouldn't be.

This is how Struts 1 is often found. Not because anyone was looking for it, but because it had simply faded into the background, quietly ferrying user requests for a decade or more. It's remarkable how often these systems still run perfectly well. Struts 1 powered the early enterprise web. It introduced structure where there was none and helped teams escape JSP soup and ad-hoc MVC patterns.

Struts 1 has earned its longevity.

A Framework Out of Time

But frameworks don't live in isolation. They live inside platforms, and platforms change. Over time, Struts 1 became less of a framework and more of a geological layer in the architecture. Important for sure, functional, but increasingly out of sync with the world around it.

Part of that misalignment is talent. In 2007, you could post a job for Struts developers and get a stack of CVs by the end of the day.

In 2026, you'd be lucky to find someone who has ever extended a `RequestProcessor` or can explain the lifecycle of an `ActionForm` without Googling it.

Most of the engineers who built these systems have long since moved on to newer frameworks, new companies, or new careers. Struts hasn't been taught in years.

Your discovery hasn't just exposed old code; you've just inherited a way of development, of design, even a development culture that no longer exists.

Tooling Moves On, Even If the Code Doesn't

Modern IDEs, static analysis engines, security scanners, and observability tools are built around annotation-driven frameworks. They expect explicit mappings and clear call graphs. Code paths that can be traced without having to mentally index a decade of XML configurations.

A Struts 1 application predates those norms. It's not hostile to modern tooling, but it lives at its edges. Things that are intuitive in Spring feel obscure in Struts, not because Struts is poorly designed, but because it predates the conveniences we now take for granted.

Security follows the same pattern. The framework itself hasn't seen updates since 2013, but the libraries it depends on have continued their own journeys.

Some patched, some abandoned, some replaced, some left exactly as they were.

In a modern environment, you're expected to patch your dependencies, keep your runtime current, and update your container images. Struts 1 lives in an ecosystem where many of those things aren't possible. You can mitigate, isolate, inspect, and contain, but you can't fix the framework itself.

The Platform Marches Forward, With or Without You

The world has moved on. Now we talk about Jakarta EE, MicroProfile, and Spring Boot. Containers no longer ship the old `javax.*` API that Struts relies on.

You can keep your environment pinned to older Java and Tomcat versions, but you can't move forward without directly addressing the compatibility gap.

This is what makes Struts 1 feel less like a stable legacy system and more like something that resists the natural motion of everything around it. It works, but only if the world agrees not to change.

All of this creates a very particular kind of pressure. Teams know the system needs to be modernised. Security knows the framework can't be easily patched. Infrastructure knows the container stack can't be upgraded. Leadership knows the business relies on workflows buried deep inside Actions and Form beans nobody wants to touch. It's not an urgent crisis, but it's a quiet tension. A sense that the world outside the framework is accelerating while the application itself stands still.

The Modernisation Myth: It's Not Instant, It's Not Easy

This is also where the conversation can easily become unhelpfully binary: either rewrite the system from scratch, or accept the risks.

Reality is far more nuanced. Modernisation is absolutely possible (and often necessary), but it isn't instantaneous. Running Struts 1 inside Spring Boot, sharing sessions between old and new components, and migrating features by feature is a practical approach, but it requires engineering effort. It means handling classloaders carefully, navigating the `javax/jakarta` divide, and building foundations for the new system without destabilising the old one.

That's the part often overlooked when people talk about transformation: even with the right strategy, modernisation takes time.

Modernisation is complex: from initial analysis to building scaffolds to support strangler-fig patterns, to the thorny aspect of migrating safely, including how to replace the business logic no one has touched since 2011. It all takes time. Because ultimately this isn't a big switch, it's an attempt to retire old workflows gracefully with the least impact possible.

Buying Time: The Unsung Part of Modernisation

And this is where companies like HeroDevs enter the conversation, not as a shortcut, but as a pressure valve. Sometimes the most important thing an organisation needs isn't a migration plan, but breathing room. A way to keep the system secure enough while the business prepares for its next move. The question becomes more about buying a year rather than a month, and about stabilising a legacy application so it can be replaced on the organisation's schedule rather than the platform's.

Modernisation carries risk, and not every system needs to be modernised immediately. Some need to run safely until they reach their natural end of life.

Supporting that reality isn't avoiding progress; it's enabling it.

Walking the Corridor, Not the Cliff Edge

So yes. Struts 1 is a product of another era. It carries assumptions the modern ecosystem has outgrown, and it sits on a foundation that the platform has moved away from. But this isn't a story about disaster or panic. It's a story about alignment. About understanding where the framework fits in a world that has evolved dramatically around it. And about recognising that the path forward isn't a cliff edge; it's a corridor. One you can walk steadily, carefully, and, if needed, with support.

The following articles explore the most visible part of that corridor: the Jakarta transition and why modern containers no longer speak the same language as Struts 1.

Hold on - it's a bumpy ride.

› [Back to the Table of Contents](#) ‹

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



The Jakarta Cliff: Why Modern Java Leaves Struts 1 Behind

EOL # Moderisation # Struts

Author: Steve Poole

Steve Poole is an experienced JVM and Java Developer, Developer Advocate, DevOps Leader, and Security Champion with expertise in software supply chain security, AI, public speaking, education, and writing. An open-source contributor (Apache, Eclipse, OpenJDK) and developer relations expert. Regular presenter at international conferences on technical topics. Formerly with IBM and RedHat, with extensive experience from operating systems to JVMs to AI. Sci-fi lover, robot builder, and occasional mad scientist. Working with Java since its early days.

Want to Dive Deeper?

Steve Poole is a speaker at JCON (April 20-23)! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn



This is the second in a series of articles exploring the realities of modernisation. In this article, you'll learn about:

- **The Modern Java Breakpoint:** Why the shift from `javax` to `jakarta` creates an unfixable compatibility break for Struts 1 on modern containers like Tomcat 10 and Jetty 11.
- **The External Pressure Cooker:** How platform baselines, security policies, and cloud migrations force the issue, even if the application code itself hasn't changed.
- **Why “Shims” Fail:** A look at why attempting to patch compatibility with shims is not a viable long-term solution for production environments.
- **Safe Operation Strategies:** What it practically means to run Struts securely on older containers without exposing the rest of your infrastructure to risk.
- **The Path to Modernisation:** How to use incremental migration patterns and extended support to modernise on your own terms, avoiding risky “big-bang” rewrites.

When an Upgrade Isn't Really an Upgrade

In engineering teams, upgrading infrastructure is supposed to feel routine. Treat your system like cattle, not pets, remember? Updating a JVM, rolling out a new Tomcat image, aligning with the company's Java baseline: normal steps in keeping a platform healthy. But the moment a Struts 1 application enters the conversation, the word “upgrade” suddenly ceases to mean what people expect.

Engineers try to move an application from Tomcat 9 to Tomcat 10. Or they push for consistency across the estate by adopting Jetty 11. Or they want to standardise on Java 21 because the rest of the organisation already has. And then the build breaks. The server refuses to start. The logs fill with errors about missing classes. What seemed like a straightforward platform refresh becomes a multi-month discussion.

The problem isn't a misconfiguration. It's not a library version clash. It's not even something you can debug. It's a deeper compatibility break that sits beneath Struts 1 itself.

The Namespace Shift That Changed Everything

For nearly two decades, enterprise Java lived in the `javax.*` namespace. Every servlet, every filter, every listener, every part of the web stack assumed that convention. When Oracle handed Java EE over to the Eclipse Foundation, the specification could move... but the namespace couldn't. Legally, `javax.*` stayed where it was. The new home for enterprise Java needed a new name. It became `jakarta.*`.

This looks like a branding change. But in practice, it's a line in the sand. When the containers (Tomcat, Jetty, Undertow, and the full Jakarta EE servers) adopted the new namespace, they removed the old one. And as soon as they did, a considerable portion of the Java ecosystem had to decide whether to evolve or remain forever pinned to older platforms.

Struts 1 never had the chance to make that decision. It had already reached its end of life years earlier. No maintainers. No updates. No migration plan. Its entire understanding of the web tier is tied to `javax.servlet.*`, and that dependency is now stranded on the far side of a one-way transition.

The EE train moved on, and Struts 1 stayed at the station.

Where Struts 1 and Modern Containers Diverge

This is why running Struts 1 on Tomcat 10 or Jetty 11 doesn't fail gracefully. It often fails absolutely. The container loads, but the framework doesn't. The classes it depends on have vanished. There is nothing to map requests to. No way to load an `ActionServlet`. The framework is looking for an API that no longer exists.

It isn't because Tomcat “removed support for older code.” It's because the industry had to choose a single namespace for the future, and the old one couldn't be carried over. With that choice made, legacy frameworks are effectively frozen in place.

And that freeze becomes an architectural constraint. You can't move the server forward without leaving the application behind. You can't rely on the security, TLS, memory, or performance improvements that modern runtimes offer. Even when the Struts application itself is stable, the platform underneath it grows older by the year, and at some point, organisations begin to feel the tension.

The Breaking Point Usually Isn't Technical

The teams that first feel this are rarely the developers. It's usually platform engineering or cloud migration teams who encounter the incompatibility when they try to consolidate the estate. They realise that every other service can run in a modern container except this one. Or security notices that an approved JDK baseline can't be applied because one application is locked to an older servlet stack. Or operations teams discover that a vendor they rely on no longer supports pre-Jakarta containers.

Most organisations don't hit the Jakarta Cliff because Struts stops working. They hit it because the environment around Struts keeps moving forward.

And at some point, "keeping things as they are" stops being a neutral choice and becomes an obstacle.

Why There's No "Compatibility Layer" That Fixes This

It's natural to ask whether there's some form of compatibility layer: a shim, a transformer, a bytecode trick, that could translate old servlet calls into new ones. A few experiments exist, most created by curious engineers rather than production environments. They work up to a point, but they introduce as many risks as they solve.

Bridging two namespaces at the heart of the servlet lifecycle isn't like adapting a library call one layer up. It touches the core of how request processing works. It affects classloading, security, session handling, filters, and dispatching: everything Struts depends on. And when you begin to modify that profoundly, you aren't extending Struts anymore. You're replacing vast chunks of what made it Struts in the first place.

Living Safely on the Older Side of the Divide

This doesn't mean organisations need to panic. Struts 1 can still run reliably on the platforms it expects. Many businesses today are effectively running a "dual-speed" architecture: modern apps on modern containers, legacy apps on carefully maintained older ones.

But the longer this pattern continues, the more these systems become outliers. Their infrastructure takes special handling. Their patching cycles diverge. Their operational requirements grow distinct. Eventually, it becomes clear that the Struts application is no longer merely *old*: it has become different in kind from everything else the organisation runs.

At this point, the question isn't "how do we keep running Struts?" It's "how do we transition without disrupting the business?"

Modernisation Doesn't Have to Be Sudden

There is a temptation to frame modernisation as a binary moment: today you're running Struts; tomorrow you're running Spring Boot; the day after that you're cloud native. But real engineering work doesn't follow that rhythm. Systems evolve in layers, not leaps.

You can wrap Struts inside a Spring Boot application. You can route legacy URLs through one servlet while gradually introducing new controllers alongside it. You can build a new login path while keeping the old one intact during the transition. These approaches allow change without jeopardising uptime, but they require deliberate engineering. They are a bridge, not a toggle.

This is the nuance that organisations often need to hear: modernisation is possible, but not instant. And that time: the months or years it takes to replace what matters most, is usually the part that's most difficult to secure.

Older Software and Modern Nightmares

Do you have time to execute an accelerated rewrite? So often, the driver for change is an out-of-the-blue moment that leaves you suddenly exposed. A new high-severity CVE in an unsupported old library. While that's always a bad dream, with more scrutiny on compliance than ever before, those dreams are now nightmares. Forcing upgrades (if you can), rewriting parts

of the application to avoid the compromised library, and arguing how to disable the pesky functionality - oh and let's have another firewall in there somewhere.

When you're faced with no time to act, you behave accordingly and attempt things you know are unwise and often downright foolish.

Buying Time Is Sometimes the Smartest Move

What you need is time, and that is often available (for a price). Getting time to plan, time to migrate safely, time to let the application time out, and just time to avoid hasty decisions that introduce more risk than they remove.

Extended support from 3rd parties isn't an admission of defeat; it's a recognition that legacy systems carry real business value, and that modernisation should happen on the organisation's schedule rather than being forced by the platform's constraints.

A Struts 1 application may be approaching retirement, but it should retire on your terms, not because an infrastructure upgrade forced your hand.

The Cliff Is Real, But It Doesn't Have to Be Steep

Struts 1 belongs to a different era of Java, and the Jakarta transition made that undeniable. But the presence of this cliff doesn't mean organisations have to leap blindly or freeze forever. It means they need clarity. They need a realistic plan. They need the breathing room to execute it safely.

The purpose of this article is not to amplify urgency, but to explain why modern containers and Struts 1 can no longer share the same road and why that is a technical fact, not a strategic failure.

In the following article, we'll move from platform concerns to the framework's internals themselves. Struts 1 introduced architectural patterns that shaped a generation of Java developers, but those patterns don't map cleanly onto the assumptions of modern frameworks.

Understanding that mismatch is essential to knowing how to migrate without losing the business logic that still matters.

› [Back to the Table of Contents](#) ‹



Automating JVM Thread Dump Analysis with AI: Practical Observability for Java on Amazon ECS and EKS

Amazon

JVM

Observability

Author: Sascha Möllering

Sascha Möllering is a Principal Specialist Solutions Architect for Containers at AWS, where he has been helping customers architect and optimize their cloud-native applications for nearly a decade. With a deep passion for container technologies, serverless computing, and Java development, he works with organizations across the globe to design and implement scalable, resilient solutions on AWS. A regular speaker at industry conferences and technical events, Sascha shares his expertise through hands-on workshops, blog posts, and technical content. His practical experience and technical depth have made him a trusted voice in the cloud-native community. He is particularly known for his work in containerization strategies and Java application modernization on AWS. When not architecting solutions or speaking at events, Sascha actively contributes to the technical community through writing, mentoring, and developing educational content that helps organizations navigate their cloud transform

Want to Dive Deeper?

Sascha Möllering is a speaker at JCON (April 20-23)! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn



Imagine the following scenario: a Java service that ran flawlessly yesterday suddenly starts consuming 90% CPU, barely responding to user requests. Users encounter timeouts, and the operations team is under immediate pressure to triage the incident. In these situations — whether the application appears stuck or CPU saturation has occurred — one of the most powerful diagnostic assets available is the thread dump. A thread dump captures the state of all JVM threads at a specific point in time, showing execution states, stack traces, and lock contention — the raw material needed to understand what the runtime is *actually doing*.

However, conventional thread-dump based troubleshooting is not without pain. Interpreting these snapshots requires deep JVM knowledge — expertise that typically only a few engineers in a team possess. Manual analysis is slow and error-prone, especially when hundreds of threads are involved, and teams often only trigger thread capture *after* latency and throughput have already degraded — meaning insights arrive too late and customer impact has already materialized.

Generative AI can make this process faster and more accessible. In this article, we show how to build an automated analysis pipeline that combines traditional observability components such as [Prometheus](#) and [Grafana](#) with an AI layer powered by [Amazon Bedrock](#). The system continuously monitors Java workloads on [Amazon Elastic Container Service \(Amazon ECS\)](#) and [Amazon Elastic Kubernetes Service \(Amazon EKS\)](#), collects thread dumps when abnormal thread activity is detected, and automatically produces structured diagnostic reports within seconds.

The Challenge of Troubleshooting Java Inside Containers

Every Java developer knows the power and complexity of the JVM. It abstracts memory management, threading, and I/O across platforms, but this abstraction also makes performance issues harder to diagnose. In container environments, the situation becomes even more complicated. Containers are short-lived, IP addresses change dynamically, and horizontal scaling creates multiple JVM instances whose behavior can differ slightly under load. By the time an engineer connects to a pod or task to gather diagnostic data, the container may already have been restarted.

Manually inspecting each dump also does not scale. Some enterprise systems generate hundreds of threads; others run many applications across several clusters. In both cases, the human bottleneck is clear: teams need a solution that automates thread dump capture and interpretation while retaining the deep JVM insight that experts rely on.

Overview of the Automated Analysis Architecture

The solution described here is deployed as Infrastructure as Code (IaC) using [AWS CloudFormation](#). It brings together three layers of functionality. Prometheus and Grafana form the monitoring and alerting layer, collecting and visualizing JVM metrics from [Spring Boot](#) applications. [AWS Lambda](#) provides the orchestration layer that captures and processes thread dumps on demand. Finally, Amazon Bedrock supplies the AI analysis engine that transforms raw thread dumps into actionable findings.

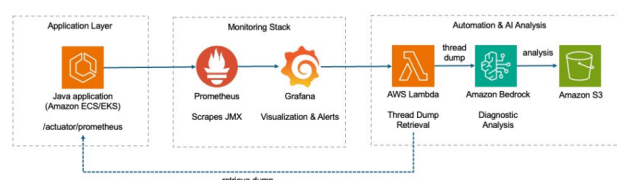


Abb. 7: Observability data flow

The monitoring infrastructure continuously scrapes JVM metrics (in our case `jvm_threads_live_threads`) from each containerized application. When thread counts exceed a predefined threshold, Grafana triggers an alert through a webhook. The alert invokes a Lambda function that automatically determines whether the affected service runs on Amazon ECS or Amazon EKS. Depending on the environment, it executes the appropriate collection method. The system stores

thread dumps and analysis reports in [Amazon Simple Storage Service \(Amazon S3\)](#) for later inspection or trend analysis. The solution code and deployment instructions are available through [Java on AWS Immersion Day](#) Workshop and on [GitHub](#).

Deploying the Base Infrastructure

To illustrate the setup, we use a simple Java application called [UnicornStore](#), built with Spring Boot and exposing standard actuator endpoints. The monitoring environment can be deployed directly from [AWS CloudShell](#) with a few commands.

To follow this walkthrough, you need:

- An AWS account with permissions to create ECS/EKS clusters, Lambda functions, and S3 buckets
- AWS CLI configured with appropriate credentials or AWS CloudShell
- Basic familiarity with Java Spring Boot applications
- Understanding of container orchestration concepts

The following snippet creates an S3 bucket for deployment artifacts and provisions the entire infrastructure through CloudFormation.

```
1 curl https://raw.githubusercontent.com/aws-samples/java-on-aws/main/infrastructure/cfn/
   unicornstore-stack.yaml > unicornstore-stack.yaml
2 CFN_S3=cfn-$(uuidgen | tr -d - | tr '[:upper:]' '[:lower:]')
3 aws s3 mb s3://$CFN_S3
4 aws cloudformation deploy --stack-name unicornstore-stack \
5   --template-file ./unicornstore-stack.yaml \
6   --s3-bucket $CFN_S3 \
7   --capabilities CAPABILITY_NAMED_IAM
```

After the stack completes, the CloudFormation output provides the URL and password for the integrated development environment used in the workshop. At this point, the basic application, monitoring stack, and IAM permissions are ready for further configuration.

Configuring Observability in a Java Application

Our sample application uses [Spring Boot Actuator](#) and [Micrometer](#) to expose detailed metrics from the JVM. In `application.config`, enabling JMX and Prometheus integration looks as follows:

```
1 spring.jmx.enabled=true
2 management.endpoints.web.exposure.include=threaddump,prometheus,health,info
3 management.endpoint.health.probes.enabled=true
4 management.endpoint.prometheus.access=unrestricted
5 management.endpoint.health.group.liveness.include=livenessState
6 management.endpoint.health.group.readiness.include=readinessState
7
8 management.metrics.enable.all=true
9 management.metrics.tags.application=unicorn-store-jmx
```

The Prometheus registry collects data such as heap usage, garbage collection pauses, and live thread counts. Within the application, the `MonitoringConfig` class enriches these metrics with contextual tags identifying the cluster, container name, and network information. These tags help the Lambda function later decide how to capture the appropriate dump:

```
1 registry.config().commonTags(
2     "cluster", cluster,
3     "cluster_type", clusterType,
4     "container_name", containerName,
5     "task_pod_id", taskOrPodId,
6     "instance", ipAddress,
7     "container_ip", ipAddress
8 );
```

This configuration ensures that each metric can be traced back to its source container and orchestration system.

Collecting Thread Dumps Automatically

Once the monitoring infrastructure is active, Grafana continuously evaluates metrics against defined thresholds. When the number of live threads exceeds the limit, the alerting rule triggers a webhook to the Lambda function. The function parses the incoming payload and extracts metadata such as the cluster type, namespace, container name, and IP address. Based on this information, it uses different strategies for ECS and EKS.

For Amazon ECS, the function directly calls the Spring Boot Actuator endpoint:

```
1 response = requests.get(f"http://{container_ip}:8080/actuator/threaddump", timeout=10)
2 response.raise_for_status()
```

For Amazon EKS, it uses the Kubernetes API to execute native JVM diagnostic commands within the affected pod:

```
1
2 # Execute command to capture JVM thread dump from a running container
3 exec_command = [
4     '/bin/sh',
5     '-c',
6     ('if command -v jcmd >/dev/null 2>&1; then '
7      'PID=$(jcmd | grep -v jcmd | cut -d" " -f1); '
8      'jcmd $PID Thread.print; '
9      'elif command -v jstack >/dev/null 2>&1; then '
10     'PID=$(ps -ef | grep java | grep -v grep | awk \'{print $2}\'}'; '
11     'jstack $PID; '
12     'else echo "Neither jcmd nor jstack found"; '
13     'fi')
14 ]
15
16 logger.info(f"Executing thread dump command in pod {pod_name}, container {container_name}"
17            ")
18 resp = stream(
19     self.core_v1_api.connect_get_namespaced_pod_exec,
20     pod_name,
21     namespace,
22     container=container_name,
23     command=exec_command,
24     stderr=True,
25     stdin=False,
26     stdout=True,
27     tty=False
28 )
```

The captured dump is then uploaded to Amazon S3 together with the relevant metadata. From there, the function triggers the AI analysis stage.

AI-Powered Analysis of Thread Dumps

Traditionally, analyzing a large thread dump requires extensive JVM expertise. Engineers must distinguish between blocked, waiting, runnable, and timed-waiting threads, understand which locks are contended, and detect possible deadlocks. The integration with Amazon Bedrock automates much of this reasoning. The Lambda function sends the dump content to Bedrock with a structured prompt that instructs the model to produce a multi-section report including an executive summary, identified issues, and optimization recommendations.

```

1  **Thread Dump Input**:
2  {thread_dump}
3  """
4      payload = json.dumps({
5          "anthropic_version": "bedrock-2023-05-31",
6          "max_tokens": 8192,
7          "messages": [{"role": "user", "content": prompt}],
8          "temperature": 0.7
9      })
10
11     # Implement exponential backoff with jitter to handle
12     # Bedrock throttling during high-volume scenarios
13     # This is critical when multiple alerts trigger simultaneously
14
15     max_attempts = 5
16     base_delay = 10
17
18     # Cap backoff to avoid over-waiting
19     max_delay = 30
20
21     for attempt in range(1, max_attempts + 1):
22         logger.info(f"Invoking Bedrock with payload ... ")
23         try:
24             response = bedrock.invoke_model(
25                 modelId="global.anthropic.claude-sonnet-4-20250514-v1:0",
26                 body=payload
27             )
28             body = json.loads(response.get("body").read())
29             return body.get("content")[0].get("text")

```

The response returned by Bedrock includes detailed commentary such as which threads are blocked, which pools are saturated, and where synchronization overhead occurs. Instead of manually searching for “BLOCKED” entries or waiting for a senior JVM specialist, developers now receive structured guidance within seconds of an alert.

Example workflow and analysis

Once the environment is deployed, it is easy to reproduce a scenario that triggers the automation. Using Grafana’s “Explore” view, query the metric `jvm_threads_live_threads` for the UnicornStore application.

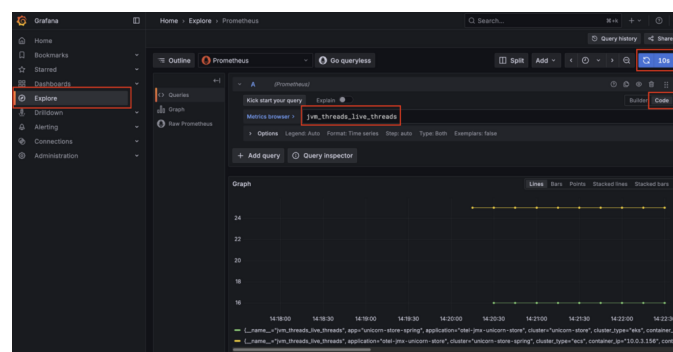


Abb. 8: JVM live threads in Grafana

The dashboard will display the number of live threads along with metadata such as cluster name, namespace, and task or pod ID.

To generate load, create 500 threads in the application by sending a POST request to its REST endpoint. For Amazon ECS, obtain the application endpoint via the load balancer:

```

1  SVC_URL=$(aws elbv2 describe-load-balancers --names unicorn-store-spring --query "
    LoadBalancers[0].DNSName" --output text)

```

For Amazon EKS, use the Kubernetes ingress hostname:

```
1 SVC_URL=http://$(kubectl get ingress unicorn-store-spring -n unicorn-store-spring -o
  jsonpath='{.status.loadBalancer.ingress[0].hostname}')
```

Start the threads and check the count:

```
1 curl --location --request POST $SVC_URL'/api/threads/start' --header 'Content-Type:
  application/json'
2 sleep 2
3 curl --location --request GET $SVC_URL'/api/threads/count' --header 'Content-Type:
  application/json'
```

The corresponding API endpoint is backed by the [ThreadGeneratorService](#), a dedicated Spring component whose sole purpose is to generate synthetic CPU pressure. For this demo we let the service spawn a large number of worker threads that execute meaningless computation loops — effectively simulating a heavy CPU workload. In our scenario, we deliberately create 500 of these threads. Once the load is active, we can switch over to the dashboards in Grafana — *Dashboards* → *Unicorn Store Dashboards* → *JVM Metrics – EKS & ECS* — and observe how the JVM behaves under stress in real time.

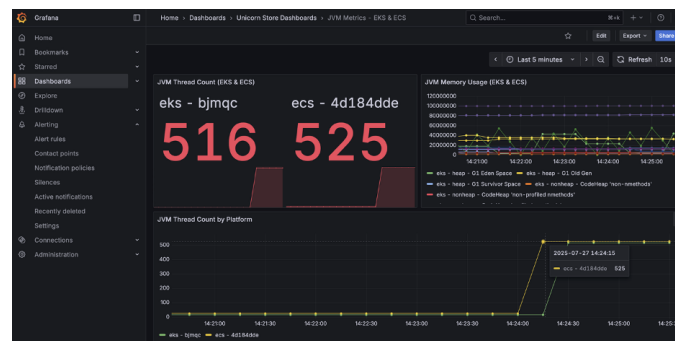


Abb. 9: Grafana alert triggering

When the Grafana alert triggers, it invokes our thread-dump Lambda function via a webhook — with a 60-second delay configured after the alert fires.

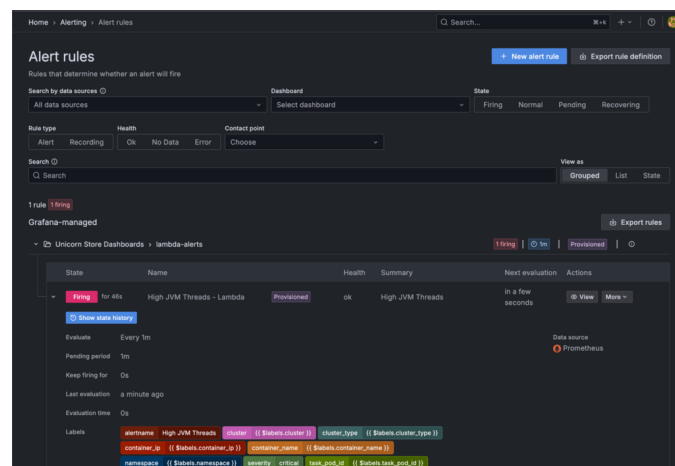


Abb. 10: Grafana alert rules

Roughly a minute later, we can inspect our Amazon S3 bucket: it now contains both the captured thread dumps and the corresponding analysis files. Below you can see an example output from the Amazon Bedrock inference step — it highlights relevant JVM state observations and includes concrete improvement suggestions for the application.

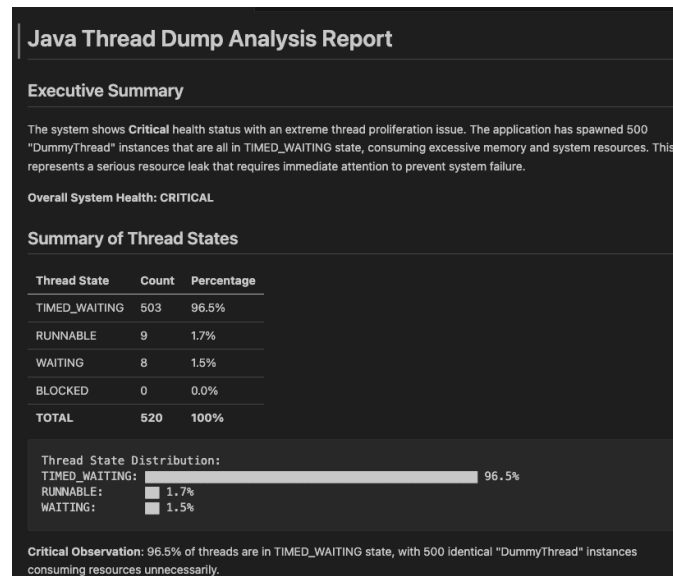


Abb. 11: Thread dump analysis report

Cleaning Up the Environment

To remove all resources created during this walkthrough, execute the cleanup scripts provided in the workshop repository. The commands below delete the EKS add-ons, S3 buckets, and CloudFormation stacks created earlier:

```
1 ~/java-on-aws/infrastructure/scripts/cleanup/eks.sh
2 aws cloudformation delete-stack --stack-name eksctl-unicorn-store-addon-iamserviceaccount
  -kube-system-s3-csi-driver-sa
3 aws s3 rm s3://$S3PROFILING --recursive
4 aws s3 rb s3://$S3PROFILING
5 aws cloudformation delete-stack --stack-name unicornstore-stack
```

Always verify that no residual resources remain, especially S3 buckets used for deployment artifacts or dump storage.

Conclusion

Automating thread dump analysis brings significant efficiency to Java performance troubleshooting, reducing diagnostic time from hours to minutes. Instead of relying on manual inspection and deep JVM expertise for every incident, developers can now receive diagnostic insights automatically within moments of an alert. The integration of Prometheus, Grafana, AWS Lambda, and Amazon Bedrock transforms thread dump interpretation from a reactive, time-consuming process into an automated, repeatable workflow.

Beyond accelerating root-cause analysis, the stored thread dumps and their corresponding AI reports in Amazon S3 create a valuable historical record. Over time, this data reveals recurring patterns such as thread leaks or periodic contention in specific code paths, enabling proactive optimization before issues impact users. The same architecture can easily be extended to other diagnostic artifacts such as heap dumps or application logs, or integrated with incident management systems like PagerDuty and OpsGenie for automated ticket enrichment.

For Java teams operating in containerized environments, this approach closes the gap between observability and understanding. By combining traditional JVM metrics with AI-driven interpretation, engineers gain deeper visibility into their applications' behavior and can act on performance issues long before they affect users. Deploy the complete solution from the Java on AWS GitHub repository and start building your diagnostic intelligence today.

› [Back to the Table of Contents](#) ‹



Enterprise-Grade AI for Java Developers

EnterpriseAI

AIArchitecture

PlatformEngineering

Authors: Maarten Vandepierre & Camille Nigon

Maarten Vandepierre is an experienced software professional and AppDev & AI Specialized Solutions Architect at Red Hat, focusing on application development and AI with OpenShift. Passionate about clean architecture, he emphasizes modular, maintainable, and scalable software design while aligning infrastructure solutions to support these principles. With deep expertise in integration technologies like API Gateways, Keycloak, Kafka, service mesh, and Camel, he ensures seamless system connectivity. Committed to continuous learning and innovation, Maarten strives to bridge technical complexities with business objectives, delivering cutting-edge solutions that help organizations navigate the evolving digital landscape.



LinkedIn

Camille Nigon is an AI Solutions Architect at Red Hat, focused on delivering open-source AI solutions with real-world impact. With a background in data science, she applies AI to complex challenges while championing collaboration and open innovation. She is also deeply committed to diversity and inclusion, actively supporting initiatives that make technology more accessible to all.



LinkedIn

Want to Dive Deeper? They are speaker at JCON (April 20-23)! If you can't attend live, the [session videos](#) will be available after the conference – it's worth checking out!



There is a moment many Java developers experience today. After years spent building reliable systems, modernizing monoliths, shaping APIs, orchestrating containers, and keeping distributed applications alive, someone suddenly asks whether you can “add a bit of AI.” At first it sounds innocent. You try a model endpoint, run a quick experiment in Quarkus, watch a few responses appear in your terminal, and feel the spark of possibility. It looks easy. Almost too easy.

Then reality arrives, quietly at first and then all at once. More teams hear about your small prototype and want their own version. A colleague suggests plugging AI into customer support. Someone else wants to classify documents. Another team wants predictive insights for scheduling. And the CIO wants an innovation roadmap on their desk by next week. Very quickly, your calm workspace starts to resemble the bar of a crowded pub on a Friday evening. Everyone wants a drink, nobody knows how it should be mixed, half of the ingredients are missing, and most requests contradict each other.

That is the situation many developers wake up in today. AI is inspiring and powerful, yet [without the right foundation](#), it can create pressure and confusion faster than any technology wave before it. To understand how to avoid this, it helps to look at AI the way a bartender looks at customers. Different customers want different things. They behave differently, they need different ingredients, and the bartender must understand those differences before attempting to serve them.

The Three Friends at the Bar: Understanding AI Architectures

Imagine three friends walking into your bar. Each of them represents the kind of AI system we see in the real world.



The first friend knows absolutely nothing but can find almost anything. Ask him what is good and he will search reviews, old blog posts, recommendations from friends and whatever the internet has to offer. He pieces everything together until he has something that resembles an informed answer. This friend represents retrieval augmented generation. The model itself does not contain your domain knowledge. Your application performs the search, retrieves relevant documents from internal systems or vector embeddings, and the model turns that context into an answer. This approach is ideal for companies whose knowledge changes frequently or where accuracy and traceability matter. In a Java world this fits naturally inside a Quarkus service that orchestrates searches across databases or APIs. You can even log which documents were used for each answer. When your compliance team asks how a conclusion was produced, you can simply show them the context that was provided.

The second friend always orders the same drink. Not similar, not almost the same, but exactly the same. He does not read the menu, does not experiment and does not change his routine. This is a fine tuned model. You take a base model and adapt it to your domain through training. The result is extremely comfortable when your context is narrow and stable. Think of invoice classification, fraud detection or routing rules for banking documents. The model becomes fast and predictable. But when your data evolves, the model must evolve as well. Suddenly you are retraining, validating, comparing versions and reviewing behaviour. Fine tuning (e.g., through [InstructLab](#)) is wonderful for the right use case but painful when the business shifts every few weeks.

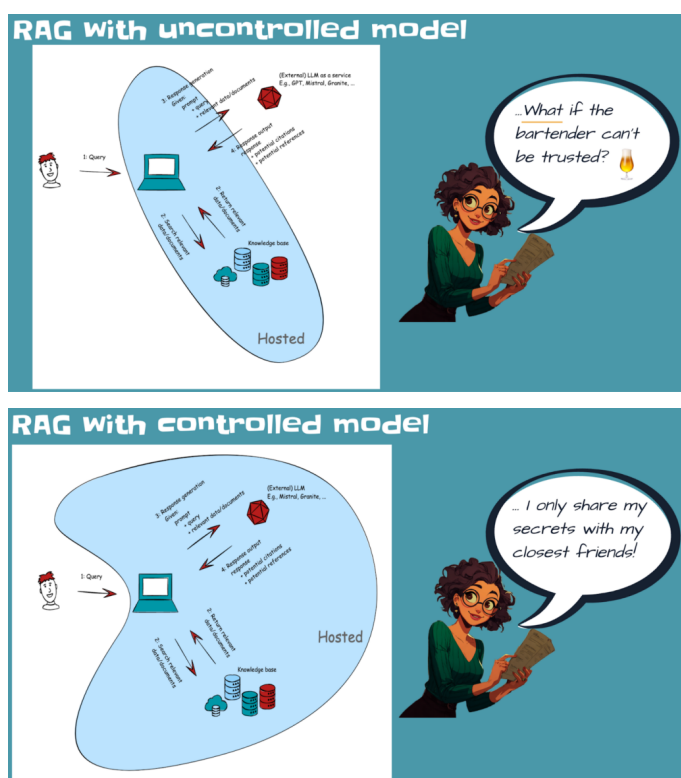
The third friend behaves very differently. Before you even sit down, she has already spoken to the bartender, checked whether the kitchen is open, verified whether you can move to a quieter table, asked for sparkling water, and possibly ordered a small appetizer because she guessed you might be hungry. This friend stands for agentic AI systems. The model becomes part of an orchestrated workflow. It calls APIs, interacts with rules engines, reads schedules, books meetings and manipulates data. It can be extremely powerful in a Java environment where you [combine the model's reasoning with Camel routes or Kafka events](#). It can also spiral into unpredictable behaviour if left uncontrolled, which is why these systems require strong guardrails, monitoring and domain rules.

These three characters are more than metaphors. They offer a way to think about AI architectures without falling in love with one approach. The question is never which one is the cleverest. The question is which one matches your use case, your data, your regulatory environment and your ability to maintain it for years.

Why AI Projects Stumble: The World Around the Model

Teams often get excited by the model and forget the rest. A model is interesting, sometimes even impressive, but it never lives alone. Real systems live inside a web of data sources, service boundaries, network dependencies, approvals, security controls and architectural constraints. This is where many initiatives begin to crack.

Consider what happens when a Java application sends customer data to a model running externally. You need answers to very practical questions. Should the prompts be stored or deleted. Is the provider allowed to retrain on your content. Are you crossing jurisdictions by accident. How do you guarantee that personally identifiable information is not leaking (even with enterprise contracts). And how do you explain behaviour to your auditors when the model suddenly begins to make unexpected decisions. Most teams only discover these questions once the first incident occurs.



Even simple questions like authentication become complex. Your model endpoint needs identity. The embedding service needs entitlements. Your storage needs controls. Your observers need logs. And once you add asynchronous behaviour or streaming, the number of moving parts grows quickly. The problem was never the model. It was everything around it.

Nothing Works Without Data: The Bartender Needs Orders

Before mixing a drink, a bartender needs a clear order. If they receive five different orders from five different people at five different times, the bar descends into chaos. AI behaves no differently. If your data is inconsistent, stale or delivered too late, the model produces unpredictable results.

This is where structured data flows become critical. Event pipelines, [change data capture](#), carefully designed schemas and stable contracts help ensure that everything the model sees is fresh and coherent. A bank that wants AI to generate customer overviews cannot use slightly outdated records. A hospital cannot summarise patient histories based on twenty minute old information. A logistics company cannot classify incoming orders if half of the information arrives through delayed batch processes. Java systems connected through Kafka or Debezium pipelines solve precisely this problem by ensuring the AI layer behaves like a calm bartender receiving clean, real time orders rather than chasing people around the bar.

When the data flows correctly, AI becomes stable. When the data is chaotic, even the best model becomes unreliable.

An AI System Needs a Well Run Bar: Platform Engineering

As soon as an AI system becomes more than a prototype, developers discover that the real challenge is not building the model based feature. The real challenge is running it. You suddenly need versioning, rollbacks, automated deployments, secure model storage, test environments, resource limits and consistent infrastructure. You need metrics showing how often prompts fail. You need traces that show how long inference took. You need logs that reveal what the model saw. You need policies that restrict which services can send what information to which model. These are areas where e.g., [service meshes can bring added value](#).

Without a [platform](#) these responsibilities scatter across teams who reinvent solutions constantly. With a platform you create golden paths that handle these concerns for every team. [Developer portals](#) guide users through template driven creation of AI services. GitOps ensures everything is declarative and observable. Service meshes enforce security and routing without manual configuration. Observability stacks reveal behaviour and drift. Suddenly the bar is not chaotic at all. It becomes a calm, predictable environment where even complex cocktails can be served repeatedly without surprises.

This discipline is not glamorous, yet it determines whether AI becomes a sustainable capability or an experiment that collapses under real world pressure.

Where Should the Model Live: The Importance of Optionality

Every organisation eventually confronts a crucial question. Should the model run locally, inside containers managed by something like Podman or Kubernetes. Should it run as a fully self hosted deployment on OpenShift AI. Or should it rely on hosted APIs in the cloud.

The correct answer changes over time. Local models offer privacy and full control. Hosted models offer convenience and rapid experimentation. Hybrid models blend both. The mistake is committing too early. The landscape of AI frameworks, model families and licensing shifts constantly. What seems affordable now may become costly. What seems accurate now may become outdated. What seems legally safe today may be regulated tomorrow.

A sound Java architecture treats the location of the model as an implementation detail rather than a foundation. Quarkus applications that speak through clean abstraction layers can swap out endpoints without a system wide rewrite. That flexibility is a form of long term resilience. It ensures the bar remains open even when suppliers change their catalogues.

The Real Secret: It Is Not About the Drink, It Is About the Bar

When you watch a professional bartender on a busy night, you notice that the magic is rarely in the individual drink. It is in the rhythm. The organisation. The clean tools. The quiet coordination. The confidence that nothing will fall apart when more people walk in.

Enterprise AI is exactly the same. Anyone can mix a quick demo. But serving hundreds of customers consistently is something else entirely. It requires an architecture that respects data, protects privacy, manages complexity, enforces boundaries and provides clarity for every developer involved.

This is where Java developers have a unique advantage. Our ecosystem has always been about reliability, clarity, modularity and strong engineering discipline. When those traditions meet the creative potential of AI, we gain systems that are not only clever but also stable.

The future will belong to the teams that run their AI systems the way experienced bartenders run their bars. Calmly. Predictably. Responsibly. And with room to adapt when new ingredients arrive.

Anyone can pour a drink once.

Not everyone can keep serving when the rush hits.

The teams who understand this distinction will be the ones who turn AI from a trend into a capability that lasts.

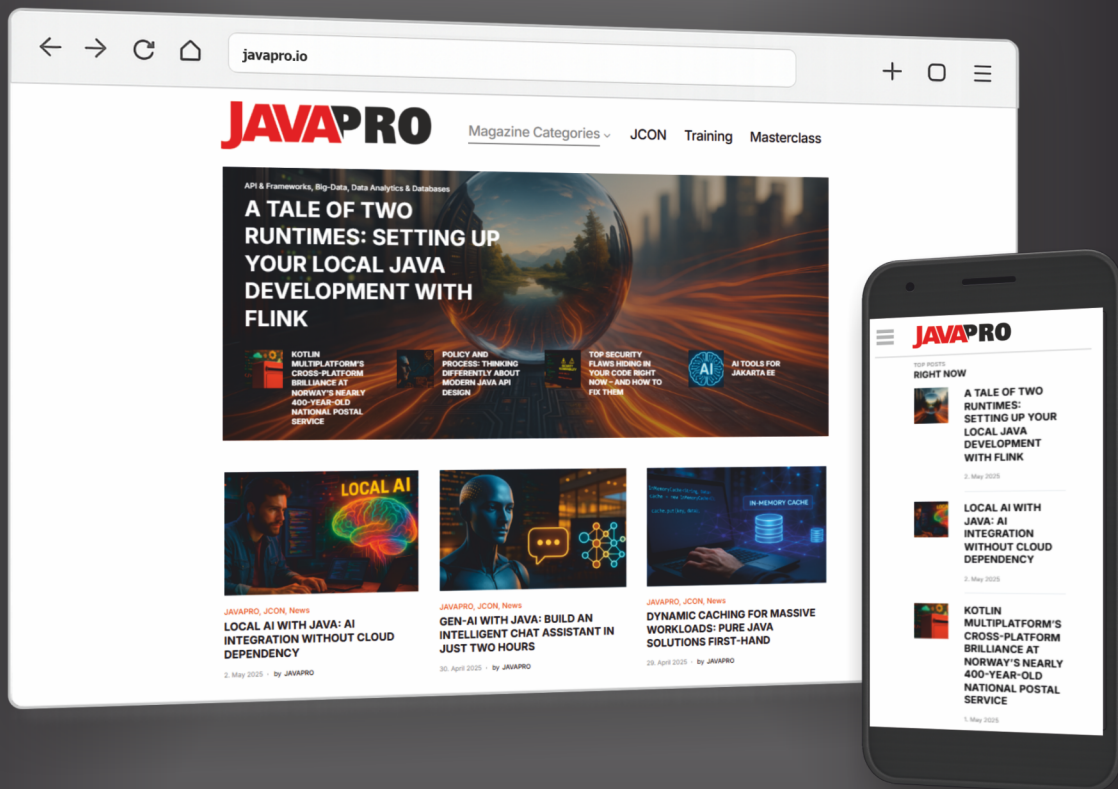
› [Back to the Table of Contents](#) ‹

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



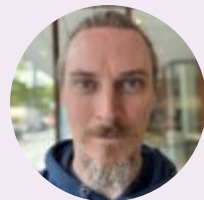
Data-Driven Projections with Kotlin Notebooks

Kotlin

KotlinNotebooks

Author: Tim Schraepen

Tim Schraepen is an enthusiastic software crafter with a big heart for his co-workers. With his 20 years of experience, he's gained some knowledge on how to build well-crafted applications and shares this whenever he can. His love for Kotlin is a natural extension for a person that cried many tears in `NullPointerException` and has come to adore strong typed languages. Outside of work Tim hosts MeetUps for the Belgian Software Crafting & Testing community (SocratesBE), plays board- and videogames, goes on walks with his trusty dog companion and tries to keep his DuoLingo streak going in learning Norwegian.



LinkedIn

Data-Driven projections help developers who organise work, estimate features and want a smarter way to see into the future. All the while staying inside their favourite IDE.

Why We Need Better Projections

In the before-fore times I was working in a team that stared at the same old backlog of work items, asking “Will we able to finish by next quarter?”.

We’d say “If everything goes well... maybe?” And then, surprise surprise: things didn’t go well. We’d have to drop scope while spending even more time in re-estimation meetings. New risks emerged. The cycle repeated.

This felt like a vicious cycle caused by “guesstimation”: vague dates -> faint hopes -> surprise late deliveries -> fatigue -> even fuzzier future dates.

Today, I know there’s a way out: we capture the data we already have, run simulations to see likely outcomes, visualise what’s actually going on, and build team confidence from what we *know* rather than what we *hope*.

Enter the realm of Monte Carlo simulations and Aging Charts.

These aren’t exotic maths for people with PhDs, they’re simply tools you can start using today, instead of doing estimation meetings.

With minimal overhead, we can turn “Will we finish on time?” into “There’s an 80% likelihood of finishing this batch of work by this date”.

In the rest of this article, I’ll share code snippets from a hands-on Kotlin Notebook, included in the accompanying [GitHub repo](#), that reads in-production dates (when a work item was moved to done), runs simulations, and produces aging charts. We’ll walk through the why, the how, and the results, with ready to use Kotlin Notebooks to adapt to your situation.

Overview

We’ll start with explaining MonteCarlo simulations, which will help you replace estimation meetings.

After concluding with the Kotlin Notebook example for those simulations, we’ll explain Aging Charts and how you can use those.

Again concluding with a Kotlin Notebook example for generating Aging Charts.

Then we’ll go over some tips ’n tricks from my personal experience, and a conclusion that sets you on the path to better predictability for you and your team.

Strap yourselves in!

Monte Carlo: You Only Need In-Production Data to Get Started

You don’t need perfect estimates or crystal-ball planning tools. What you do have, typically, is a history of when work items actually got done. If your team tracks work items (say, in Jira, YouTrack, Miro, or whatever) and records the “done” date, then you already have raw data for a forecast model.

By feeding that historical data into simulations that produce a probability distribution (a Monte Carlo simulation), you can project future outcomes for **batches** of work. The only “input” you need is this: for each work item its done date, and optionally its start date (giving you the cycle time). If you have that, you can ask: “Given our historical throughput, what’s the likely date by which the next N work items will be done?”.

It’s important to realise that it’s about the next **batch** of work items, not just an individual work item.

For many teams this is huge. You stop relying on gut-feels and optimistic guesses, and instead work with distributions and chance levels: “We’re 85% likely to finish these 20 work items by May 14th”. That gives you the language of risk, not hope. Best thing is that you don’t need to do estimation meetings anymore that keep interrupting your productivity and feel like ritualistic and meaningless ceremonies.

What Monte Carlo Is Good For

Let's pause for a moment on Monte Carlo. If you've heard of it, you might relate it to "mathy", "finance" or "physics" territory. But at its heart, it's quite simple: you model many possible futures by sampling from your historical distribution of outcomes. Then you summarise what those futures look like.

For example: if your team historically finishes 5 to 10 work items per week (average 7, but sometimes 5, sometimes 10), a Monte Carlo simulation will run many "weeks" sampled from that distribution, and say: in 80% of the runs you'll finish 20 work items by X date.

In effect: You define your "unit" (could be work items per week, days per work item). You sample many runs (say 1,000 or 10,000) of how long it takes to finish the backlog you want to have a prediction for. You pick a percentile (e.g. 85% or 90%) as your safe forecast.

Quick note: the unit you choose will define the granularity of your forecast.

In the example notebook and in my current team, we're using number of work items finished per day.

Most of the days we'll have 0 work items finished, but we'd rather answer "when will it be done" than "how many weeks will it take to be done".

In the repo you'll find a Kotlin notebook that does exactly this. Here's the data-cell that defines the MonteCarlo simulation:

```
1 data class Trial(val totalDays: Int)
2 fun monteCarloFromThroughput(
3     storiesInNextMilestone: Int = 13,
4     trials: Int = 10_000,
5 ): DataFrame<Trial> {
6     val values = historicalThroughput.storiesDelivered.values().toList()
7     val runs = List(trials) {
8         var done = 0
9         var days = 0
10        while (done < storiesInNextMilestone) {
11            done += values[Random.nextInt(values.size)] //some days we had delivered 2
12                stories, other days 1, most days 0
13            days += 1
14        }
15        Trial(days)
16    }
17    return runs.toDataFrame()
```

You'll find more polished code in the notebook: filtering out zero-throughput days, selecting more relevant time windows, etc.

```
1 val distributionBarChart = plot(frequencyPerTotal) {
2     layout { size = 2000 to 900 }
3     bars {
4         x(totalDays)
5         y(frequency)
6         alpha = 0.6
7     }
8     // vertical markers
9     vLine {
10        xIntercept.constant(p50)
11        type = LineType.DASHED
12    }
13    vLine {
14        xIntercept.constant(p85)
15        type = LineType.DASHED
16    }
17 }
18 distributionBarChart
```

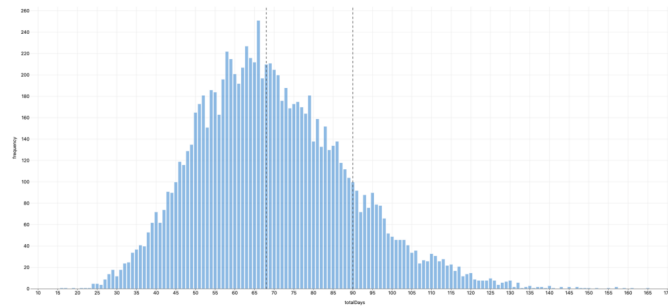


Abb. 12: The distribution barchart of the monte carlo simulation, looking like a bell-curve with some outliers.

Why this Matters

You get probabilistic forecasts, not a single deterministic “We’ll be done on June 1”. Which enables you to visualise risk (e.g. at 50% you finish by May 1; at 90% by June 15). Based on data! Yay!

Teams that use this language stop committing to “we’ll finish by X” and start saying “We are likely to finish by X unless something changes”. This changes behaviour, expectations and trust.

Furthermore, it’s very cheap to get going: just your historical done-dates, Kotlin notebook, maybe a CSV import. You don’t have to deal with “getting experienced in doing estimation meetings”, or phrases like “can you re-estimate this because I’ll have a hard time selling this to our customer”, or some other similar nonsense.

Common Questions

Q: Don’t you include weekends and holidays in your simulations?

A: Good observation. That is indeed the point. We want realistic simulations. In reality, we also don’t work weekends or during holidays, right?

Q: We’re only starting and have a very limited amount of data. Will this be useful to us?

A: Without going into too much detail, you can get meaningful predictions from data of having completed 5 work items. Read “When will it be done” for proper info.

Q: We just hired 2 new junior members. Won’t this impact our predictability?

A: It sure will! Good thing you can easily select data that is relevant to your situation. ;)

Q: Do all our work items need to have the same size?

A: No, but I will say that your prediction accuracy gets a boost when they are similar in size.

Actually, this last part could use a bit more of an explanation.

Goodhart’s Law

“When a measure becomes a target, it ceases to be a good measure”

Charles Goodhart

Applied to work items having the same size that law could look like this:

We know this work item will take longer than the others, so we’ll split it up into 2 work items. That way, “we’ll get more work done and our metrics will look better”.

If you split up the bigger work item into 2, and those 2 work items still manage to deliver value individually, then this is great! You’ve now managed to focus more on finishing WIP and delivering value more quickly.

If they don't deliver value individually, then it's indeed more likely Goodhart's Law in practice. "Deliver value" is a pretty broad spectrum, however:

- Learning about technical issues you face: value++
- Learning about how your users interact with an experimental feature: value+++
- A new partial feature already drives up sales: value++++

So, try to initially define work items that are similar in size. It's ok if they're not, and it's better if you split them up when they end up delivering value individually.

Pro-tip: keep track of how many times work items were split up, and use this data in your MonteCarlo model to get even better predictions!

Practice Time!

If you're following along with the accompanying repository, now is a good time to check out and play around with 01_Data-Exploration.ipynb, 02_MonteCarlo.ipynb and 03_When_Will_It_Be_Done.ipynb.

There's a tricky bit you "just need to know" when working with Kotlin Notebooks:

Type inference on dataframes happens when a data-cell has run. This is why it'll be important to put your transformations into one data-cell and then continue working with them in a next data-cell.

That's also the reason why sometimes it's better to interrupt chaining calls on your Dataframe. The classic example is filtering out null values, and still having to deal with the type being a nullable type in the next chain call.

Come back to this article when you're done playing around to learn about another important component of working with your data: getting fast feedback to adjust your process using Aging Charts!

Aging Charts: What They Are & How to Build Them

If Monte Carlo gives you "when will we finish X amount of work items", then Aging Charts tell you how old the finished work typically gets.

They answer: "Given the current Work In Progress, which items are getting old?".

It's the equivalent to burn-down charts when you're working in sprints: "Do we need to do something to finish work quicker?".

In practice, an aging chart is a scatter plot of the age of each active work item versus time. Over time, items that are not closing will drift up (age increases). If you overlay your typical cycle time, you can see which items are "stuck".

To be able to run useful Aging Charts, in addition to the "done date" you needed for MonteCarlo's, you will now also need the "start date".

If you want an even more detailed overview you can add all the dates of "the value stream": basically all the steps a work item goes through in your team. For example when we pull in a work item in our backlog, it'll go through analysis (where we know when this started and finished), development, testing, when it was deployed to acceptance and when to production, or if it was abandoned (no longer relevant, we chose a different option, ...).

In the repo you'll find a Kotlin notebook that loads your work items, computes cycle times, and plots the distribution over time showing outliers more easily. The relevant snippet that calculates the *Cycle Times* looks like this:

```
1 val offsetWhenFinishedSameDay = 1
2 val valueStream = cleaned
3     .add("AnalysisTime") {
4         `In Analysis`.date.daysUntil(Analyzed.date) +
5             offsetWhenFinishedSameDay
6     }
7     .add("CycleTime") {
8         `In Analysis`.date.daysUntil(`In Production`.date) +
9             offsetWhenFinishedSameDay
10    }
11    .select("StoryId", "AnalysisTime", "CycleTime")
```

Just like the MonteCarlo notebooks, you'll see it uses DataFrame to do data exploration, filtering and transformation, along with Kandy, a simple chart library to visualise what the mean (50 percentile) and 85th percentiles are for work items that are being analysed (actively in analysis) and work items that are being developed.

```

1 //The datacell that generates the scatterplot below:
2 val agingChart = plot(valueStream) {
3     layout { size = 2000 to 900 }
4     points {
5         y(CycleTime)
6         x(StoryId)
7     }
8     // vertical markers
9     hLine {
10         yIntercept.constant(percentiles["p50CycleTime"])
11         type = LineType.DASHED
12     }
13     hLine {
14         yIntercept.constant(percentiles["p85CycleTime"])
15         type = LineType.DASHED
16     }
17 }
18 agingChart

```

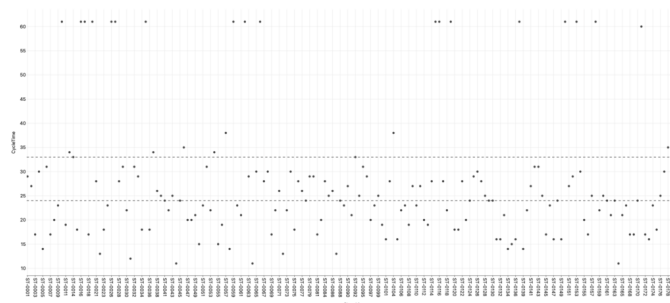


Abb. 13: The generated scatterplot, showing p50 and p85 cycletime dotted vertical lines, and all of the ages (cycletimes) of the work items in the dataset.

Why this Matters

It's the tool that gives you the earliest possible feedback for on-going work

More specifically it provides early warning for “drift”: It'll become obvious which work items need more attention to progress on.

It helps you wake the team up: when you show “Today most of our open work items are 45 days old, up from 30 last quarter”, it hits different.

It ties nicely into Monte Carlo: if you have a bunch of very old items, your future projections should probably need to account for that latent work (a.k.a. *drag*; work in progress that significantly impacts today's projections).

More Practice Zime!

Time to explore! This time you should figure out 04_AreWeGoodBusy.ipynb.

Bringing It Together: From Data to Insight to Action

Putting it all in your toolkit means you can run a short “forecasting ritual” each iteration, backlog refinement meetings, or at roadmap reviews. It might go like this:

Export your list of done-dates for the last N weeks (e.g., last 12–24 weeks).

Load it into your Kotlin notebook that’s supposed to live in your projects’ code repository.

Run a Monte Carlo simulation for “X work items remaining” in order to get finish date distribution.

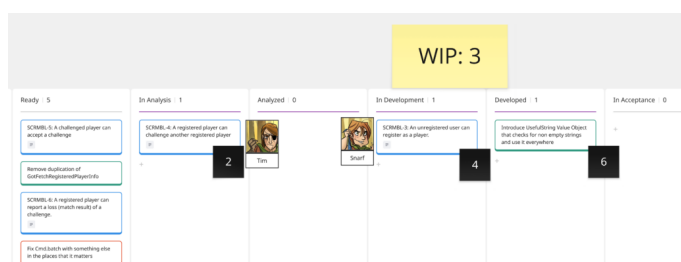
Run aging charts to visualise age distribution, so you can highlight “old” items in your work item dashboard (e.g. “danger zone”, “about to retire”, ...).

Interpret results in team meeting: “If everything remains the same, we have an 80% chance of finishing by June 10.” Or during daily stand-ups: “We have 1 item that’s nearing the 50th percentile. What options can we come up with to do something about it?”

As a team, decide action: maybe stop starting new work items, focus on clearing old ones (by swarming on it), adjust expectations with stakeholders, update roadmap, ...

Assign a different team member to follow up these metrics for next meetings, so that knowledge spreads inside the team.

Here’s what a simple Kanban board could look like in Miro, where the aging of work items is displayed on the black post-its. If you use these in combination with your Aging Charts, you’ll be able to act more easily on items that are getting “too old”.



Why Kotlin Notebooks?

Many teams use spreadsheets, or maybe built-in work-tracking tools for forecasting. So why use Kotlin notebooks?

The existing work-tracking tools typically don’t offer transparency on how they got these charts and predictions. If you build it yourself, you also own the process yourself, because you understand the why and how.

The language is familiar to you as a developer. No new platform to learn.

Kotlin notebooks let you mix code, comments, charts in one place. The narrative stays with the code.

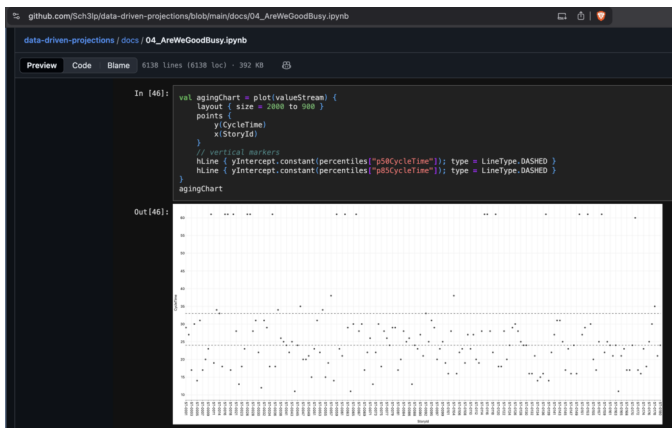
Since it’s code, it’s easier for developers to start caring about process. ;)

You can version-control your notebook with your git repo. The commits will tell a story about the evolution of your codebase AND your development process.

You can customise: connect to your own backlog system, fetch data automatically (with minimal API usage), then run the notebook.

It supports re-use: once set up, each time you need a new prediction you just click a few cells, refresh your data, regenerate charts. Minimal. Backend. Cost. You’re welcome.

Kotlin notebooks are readable in your remote repository (i.e. GitHub or GitLab), allowing for integration into slides or management dashboards.



Tips & Pitfalls from Experience

Read “When will it be done” by Daniel Vacanti to learn much more about data-driven projections than the surface I barely scratched in this article.

[Little’s Law](#) indicates the only thing you have control over is *Work In Progress*, so minimise it as much as you can!

That means don’t pick up new work when you already have too much WIP; improve your skills instead.

Data hygiene matters: remove entries that aren’t “production done” (e.g., closed across QA but not released). Use consistent definitions.

Exclude outliers: if you had one 120-day work item, it will skew your distribution too much without being relevant. Consider filtering or grouping by item size.

Use stable units: Pick throughput units that make sense (work items per week or per day). Be consistent!

Beware “scope creep” if you picked work items per week as the unit: If you keep adding work mid-sprint, your historical data may under-represent actual throughput. Monte Carlo’s usefulness requires that the “same process” continues.

Visualise the tail: Often the 90-95th percentile is what matters for risk communication (“we finish by X only after two standard deviations”).

Communicate well: Replace “we’ll finish by June 1” with “we’re 80% likely to finish by June 1, 90% by June 20”. Stakeholders appreciate transparency. If you have to dumb it down, just give ranges: “We will likely be done between June 1st and June 20th”.

Label your work items with how close they are to the 50th and 85th Aging percentiles, so that it’s super obvious in your visualisation of your work (i.e. a KanBan or Scrum-board) which work items need attention.

Make it a habit: Treat the notebook as part of your iteration start or backlog-triage ritual. Don’t let it sit idle.

And finally a Kotlin Notebooks caveat: `kotlinx:dataframe` is still in development at the time of writing, but the development team is nearing a stable 1.0.0 release. And much like every library dependency, you might want to lock down the current version, or keep on running on the bleeding edge (and report bugs to the dev team).

Conclusion

If you've ever felt the pain of over-optimistic planning, surprise delays, and frustrated stakeholders, you know how easily estimation can become a vicious cycle.

By adopting a simple data-driven approach (using historical done-dates), running Monte Carlo simulations in a Kotlin notebook, and visualising backlog drift with Aging Charts, you can switch to a virtuous cycle in which you measure, visualise, decide, act and improve.

The repository “data-driven-projections” gives you a low-cost, high-leverage toolkit. Grab it, adapt it, run it.

The future doesn't have to be guesswork, it can be probabilistic insight. And for a feature team, that kind of insight is gold.

Resources

[The Data-Driven Projections GitHub Repository, containing the Kotlin notebook examples](#)

[When Will It Be Done - Daniel Vacanti](#)

[FlowMetrics and Little's Law](#)

[Kandy Docs](#)

[DataFrame Docs](#)

Getting help

The *#notebooks* channel on the [kotlinlang slack](#) has very helpful people (including the Notebooks development team) who will gladly help you along on your journey!

› [Back to the Table of Contents](#) ‹



Avoiding Unsustainable Responsibility

TechnicalDebt

AgileReality

TeamDynamics

Author: Monschau Andreas

Andreas Monschau works as a Senior IT Consultant at Haeger Consulting in Bonn. He has taken on various roles, ranging from software developer and software architect to test manager, team leader, and requirements manager. In addition, he leads Haeger Consulting's extensive trainee program and supports other companies in establishing similar processes.



LinkedIn

Success or strain – that is often the question when you step into a chaotic software project. People frequently find themselves at one of two extremes: either the project is stabilised and its recovery is widely acknowledged, or the situation proves unsalvageable despite sustained effort, and the engagement ends without the desired outcome.

But it does not have to come to that. You do not have to leave a chaotic project as a martyr. Often, small actions are enough to make a difference. This article provides a brief overview.

In theory, it could be that simple: you join a team in a software project as a new member, and everyone knows what to do. The team understands the technology, knows the requirements, and if problems occur, they are resolved quickly. That sounds great. But how often do projects like this actually happen? Do we not just as often end up in projects where nobody really knows what to do, what the overall purpose is, and nobody understands the technology?

How common are these chaotic projects? Is there reasonably reliable data? There is. For this article, I use figures from the Standish Group Chaos Report from 2020 (<https://www.standishgroup.com/>).

IT leaders were asked globally about the state of their projects. The respondents came from various industries, including finance, insurance, manufacturing, (e-)commerce, healthcare, and the public sector.

The numbers suggest that 31% of all software projects are successful. That means they stay on schedule, stay within budget, and ultimately deliver the expected value for users or customers. 31% is roughly one third - but what about the remaining 69%? 50% of projects, i.e., half, have major issues of various kinds (including budget overruns), and 19% fail completely, meaning they never reach completion.

Let us look at a few common problems and traps that software projects can fall into, and at the same time consider what you can do to address them, even with small steps.

Technical Debt

Technical problems come in many forms. One example is an outdated or poorly maintained codebase that turns every change into an unpredictable risk. Nobody dares to touch the implementation anymore because you never really know what effects a change might have somewhere else. Technical debt becomes an innovation killer, and developers often say things like, “I do not dare to change anything.”

Technical debt often arises under time pressure. Poor decisions can also contribute to an ever-growing pile of debt. Something gets built “quickly,” and eventually it backfires through rising maintenance costs, higher defect rates, and frustration among developers.

What can you do about it?

Technical debt is not inherently wrong. It only becomes dangerous when nobody knows where it is, why it exists, or who is responsible for it. This article cannot solve all technical debt, but it offers a practical guideline: keep technical debt transparent, involve stakeholders, and show what problems arise if you keep it.

Poor Team Atmosphere

As long as not all team members have been replaced by AI, projects still involve people. And people often end up in conflict with other people. Team members have different views on quality, software development, or how they want to work together. Some are outgoing and visible, others prefer to work quietly. There can also be different opinions about organization and the delivery approach. A typical example is a clash between strong advocates of agility and more critical team members who see Scrum as just one approach among many, and not necessarily one that produces good results.

What can you do about it?

Find out where the issues in the team actually are. Is it missing communication, conflicting goals, different expectations, or personal sensitivities? Do not only run retrospectives. Run dedicated workshops to get to the root causes. Follow the Las Vegas principle: what is said in the room stays in the room. Encourage the team to commit to specific actions that improve the team atmosphere.

Poor Onboarding

The success of new team members depends heavily on a good start. If people are welcomed without warmth and key contacts are not reachable, motivation drops quickly. Confusing tasks, tasks that are too easy, or tasks that are overwhelming right at the beginning also harm new joiners. So does the feeling of not being taken seriously. Toxic team traits often become visible very early during onboarding and indicate how dysfunctional collaboration is. That is why poor onboarding often goes hand in hand with a poor team atmosphere.

What can you do about it?

Take the time to create a real plan and get real commitment from the team. Onboarding is not a friendly reception with coffee and a laptop. It is the first productive phase for a new team member in a software project and often determines early on whether someone will become effective long-term or mentally disengage. Identify what is needed to become productive, write it down, name contacts, and describe what to do. Even if it ends up being “just” a checklist, it is still more than many projects manage to provide.

Organizational Problems

The right organizational setup can bring out the best in a project - or the worst. Many projects skip proper risk management and emergency plans because everyone hopes for the best outcome. When problems appear, there is no preparation, people start looking for someone to blame, and heads begin to roll.

Unrealistic time management and missing prioritization are also major factors. If features or delivery dates are promised to customers without aligning with the team, planning becomes wishful thinking instead of realistic effort estimation. Pressure is applied across the team, which leads to poor software quality.

The team is also left confused when prioritization is missing. If everything is labeled “very important,” then eventually nothing is important, and nobody knows where to start. In the end, this leads to activity that has little to do with the actual project goal.

What can you do about it?

Sometimes you have to name the elephant in the room, even if it is uncomfortable. State the risks and define at least simple emergency plans. Commitments to external stakeholders should only be made after internal alignment with the team. Review plans regularly and adjust them instead of forcing them through at all costs. Prioritization is part of this: define one clear prioritization that is understood and accepted by everyone.

Unclear Responsibilities

Unclear responsibilities accelerate problems in projects. Tasks “belong” to everyone and therefore often to nobody. Decisions are postponed because it is unclear who is allowed to make them or who is supposed to make them. Responsibilities within the team are vague, handovers are sloppy, and important topics sit untouched until they finally escalate.

It becomes especially problematic when multiple roles or people implicitly claim the same responsibility or block each other: project manager, product owner, business stakeholders, architects, team leads - everyone has a say, but nobody decides. This is highly demotivating for the team. Initiative is discouraged, decisions become an unpredictable risk, and in the end people avoid upsetting others.

What can you do about it?

For every task, assign a clearly responsible role or person so responsibilities are explicitly separated. In this context, responsibility means decision-making authority. A RACI matrix helps make responsibilities transparent within a project. In addition, each role should regularly reflect on this question: what is my job, and what is not?

This article can only highlight a fraction of the potential traps a project can fall into. There are many more, from cognitive biases to poor communication to micromanagement. My main point is this: do not try to change everything at once through large-scale actionism. Take small steps instead. Try, evaluate, improve. A pragmatic application of inspect and adapt is, in my view, well suited to improving the project situation step by step. I wish you success.



Where Is the Logic of My Application?

ArchUnit

SoftwareArchitecture

ContinuousVerification

Author: Andres Sacco

Andres Sacco has been working as a developer since 2007 in different languages, including Java, PHP, NodeJs, Scala, and Kotlin. His background is mostly in Java and the libraries or frameworks associated with this language. In most of the companies he worked for, he researched new technologies to improve the performance, stability, and quality of the applications of each company. In 2017, he started to find new ways to optimize the transference of data between applications to reduce the cost of infrastructure. He suggested some actions, some applicable in all the manual microservices and others in just a few. All this work concludes with the creation of a series of theoretic-practical projects, which are available on the page Manning.com Recently, he published a book on Apress about the latest version of Scala. He also published a set of theoretical and practical projects about uncommon ways of testing, such as architecture tests and chaos engineering.



LinkedIn

Nowadays, many teams work with microservice-based architectures, where each service encapsulates its own logic and solves a specific problem—whether it’s managing customer data, handling user information, or sending notifications. Over time, a team may become responsible for dozens of these microservices, and not all of them evolve at the same pace. As a result, inconsistencies often appear: different project structures, package naming conventions, or even distinct prefixes for equivalent layers.

These inconsistencies increase the cognitive load for developers who need to maintain or update the services. Every time someone opens a microservice in the IDE, they must first recall—or rediscover—where each part of the application lives. This raises a natural question: *How can we reduce this friction? Is there a tool or library that can automate standardization?*

The Problem: Inconsistency Across Applications

A company—or even an individual team—often makes its own decisions about how to organize an application internally. These decisions include package naming, prefixing specific classes such as controllers or repositories, defining persistence layers, and the rules governing their interactions.

A single team does not always own these decisions. In many organizations, multiple teams are given the freedom to define their own conventions. In other cases, the company may introduce a standard months or even years later, or revise an existing one as the platform evolves. In both scenarios, developers face recurring problems in their day-to-day work, such as increased cognitive load, slower feature development, and hidden technical debt.

The following table is just a simple representation of a possible situation that could happen in a company with multiple microservices and different structures.

Problems	Microservice A	Microservice B
Package	com.api.flights.repository	com.api.hotels.persistence
Loggers	static final Logger LOGGER;	Logger logger;
Controller	CatalogController	ReservationHandleHTTP
DTO	Without business logic	With tons of business logic
Settings	Distribute across the application	Centralized <u>on</u> one package

Abb. 14: Comparison between two different microservices

These problems tend to increase as the platform evolves, especially in legacy microservices that do not undergo migration.

Solution Part 1: Defining an Architectural Guide

One of the main problems with having different structures defined by each team or developer in each microservice is the lack of a single document that contains all the rules. The document does not need to be complex or follow a special format; it just needs to define the rules with some examples, using a tool or webpage that could be accessible to anyone in the company, but only a small group of developers can modify the document.

This document is not a silver bullet that solves all problems. Still, it reduces the risk that someone misunderstands a rule and introduces a mutation into an existing or new application.

Let’s see some examples of possible rules that could be great to have defined in a company:

- **Controllers:** Must use `@GetMapping`, `@PostMapping`, and other dedicated annotations to expose actions when the application uses Spring Boot.
- **Services:** Must reside in classes annotated with `@Service`, located in the service package, and contain all the business logic.
- **Persistence:** Must reside in the `repository` package and be implemented using interfaces annotated with `@Repository`.
- **DTO:** Must be located in a dedicated `dto` package and must not contain business logic.

After defining the rules in a document, it's possible to find a tool that takes these definitions, validates them across different applications, and shows developers if anything is not aligned with them.

Solution Part 2: Validating Architecture Automatically

Over the years, several well-known alternatives have emerged to help detect problems early in the development process. Traditional static analysis tools such as PMD, Checkstyle, and FindBugs focus primarily on code quality, style consistency, and standard programming errors.

More recently, tools like ArchUnit have introduced a different approach by allowing teams to express architectural rules directly in code and execute them as part of the test suite.

The following table compares these alternatives, highlighting their strengths and limitations for validating microservice structure and architecture.

Capability	PMD	Checkstyle	FindBugs	ArchUnit
Runs as unit tests rather than as a static analyzer	✗	✗	✗	✓
Analyzes source code to detect issues	✓	✓	✓	✓
Supports custom and extensible rules	✓	✓	✓	✓
Validates package structure and naming conventions	✗	✗	✗	✓
Detects architectural violations (layering, dependencies)	✗	✗	✗	✓
Enforces field and class naming constraints	✗	✗	✗	✓
Strong community and documentation	✓	✓	✓	✓

Abb. 15: Comparison between the different alternatives to analyze the code

This list of differences should not be interpreted as an argument for replacing static analysis tools with ArchUnit. Static analyzers remain essential for detecting code smells and defects, whereas ArchUnit excels at validating architectural decisions. The real value comes from combining both approaches to cover different layers of quality control.

Introduction to Archunit

ArchUnit emerges as a practical solution for consistently and automatically enforcing architectural decisions. ArchUnit is an open-source library that allows teams to express architectural rules directly in code and execute them as part of the unit test suite. Using a declarative, fluent API enables developers to define explicit constraints on classes, packages, methods, fields, and dependencies.

Example of Use

To use this library, the first step implies adding the dependencies to the POM file in the case of Maven, as shown in the following fragment:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/
        maven-4.0.0.xsd">
5      <properties>
6          <!-- Other properties -->
7          <archunit-junit5.version>1.2.1</archunit-junit5.version>
8      </properties>
9      <dependencies>
10         <!-- Other configurations -->
11         <dependency>
12             <groupId>com.tngtech.archunit</groupId>
13             <artifactId>archunit-junit5</artifactId>
14             <version>${archunit-junit5.version}</version>
15             <scope>test</scope>
16         </dependency>
17
18         <dependency>
19             <groupId>com.tngtech.archunit</groupId>
20             <artifactId>archunit-junit5-api</artifactId>
21             <version>${archunit-junit5.version}</version>
22             <scope>test</scope>
23         </dependency>
24     </dependencies>
25 </project>

```

As a recommendation, check the latest version of this library on the official webpage or a repository like [this](#) regularly.

Let's create a simple rule that verifies that loggers must be private, static, and final, and that the attribute must be named `LOGGER`. To do this implies creating a class in the test folder with the content that appears in the following fragment:

```

1  @AnalyzeClasses(packages = "com.twa.flights.api", importOptions = ImportOption.
    DoNotIncludeTests.class) // #1
2  class GeneralCodingRulesTest {
3      @ArchTest
4      static final ArchRule loggersShouldBePrivateStaticAndFinal
5          = fields().that().haveRawType(Logger.class).should()
6              .bePrivate().andShould().beStatic().andShould().beFinal().andShould().
              haveName("LOGGER")
7              .because("Logger variables should be private, static and final, and it should
                  be named as LOGGER"); // #2
8
9  }

```

Before declaring the class, it is essential to specify which classes this test will analyze using the `@AnalyzeClasses` annotation; it is possible to select entire packages or just the names of the classes. In the example, the tests are excluded from the verification process as a good practice to reduce the execution time.

The annotation `@ArchTest` indicates that the following fragment is an ArchUnit test that needs to be executed; if the annotation does not exist, the test will be ignored.

The library allows you to analyze fields, methods, classes, interfaces, or packages. The first part of the test must specify which items will be analyzed to detect problems; after that, predefined conditions can be used to create a rule, such as `.bePrivate()` or `._beStatic()`. *It's possible combine multiple validations using the method `.andShould()`.*

As a best practice, it's recommended to introduce the `.because()` with a brief explanation of what happened when the rule detects an issue.

The rules work as a standard unit test, so to execute them, it is only necessary to run the command of the application's dependency management tool, which in this case is Maven.

```

1  $ mvn test
2  ....
3  ....
4  [INFO]
5  [INFO] Results:
6  [INFO]
7  [ERROR] Failures:
8  [ERROR]   Architecture Violation [Priority: MEDIUM] - Rule 'methods that are declared in
   classes that reside in a package '..controller..' and are public should not be
   annotated with @ResponseBody, because Controller endpoints should not be annotated
   with @ResponseBody. Not include on the declaration the annotation.' was violated (1
   times):
9  Method <com.twa.flights.api.catalog.controller.DocumentationController.
   redirectToDocumentation(jakarta.servlet.http.HttpServletRequest)> is annotated with
   @ResponseBody in (DocumentationController.java:27)
10 [ERROR]   Architecture Violation [Priority: MEDIUM] - Rule 'fields that have raw type org
   .slf4j.Logger should be private and should be static and should be final and should
   have name 'LOGGER', because Logger variables should be private, static and final, and
   it should be named as LOGGER' was violated (3 times):
11 Field <com.twa.flights.api.catalog.controller.CatalogController.lo> does not have
   modifier FINAL in (CatalogController.java:0)
12 Field <com.twa.flights.api.catalog.controller.CatalogController.lo> does not have
   modifier STATIC in (CatalogController.java:0)
13 Field <com.twa.flights.api.catalog.controller.CatalogController.lo> does not have name '
   LOGGER' in (CatalogController.java:0)
14  ....

```

The console error message is descriptive of where the problem is located and what the root cause is.

This article uses a source from a [GitHub repository](#); feel free to clone it and use it to learn about how to keep the application updated.

Which Types of Rules Exist?

A series of rules can be created with Archunit. Let's see some of the most common:

- **Naming:** It's possible to define the name that fields, classes, interfaces, and packages follow a standard on the prefix, suffix, or a specific regex.

```

1  @ArchTest
2  static final ArchRule finalStaticVariablesInUppercase = fields().that().areStatic().and()
   .areFinal().and()
3      .doNotHaveName("serialVersionUID").and().doNotHaveModifier(JavaModifier.
   SYNTHETIC).should()
4      .haveNameMatching(".*^[A-Z].*")
5      .because("Variables with static and final modifiers should be named in
   uppercase");

```

- **Annotations:** It's possible to validate that fields, methods, or classes use some specific annotations. This could be especially useful for checking the correct use of Spring Boot annotations or those of any other framework.

```

1 @ArchTest
2 static final ArchRule publicMethodsShouldAvoidUseRequestMapping =
3     methods()
4         .that()
5             .areDeclaredInClassesThat()
6                 .resideInAPackage("..controller..")
7         .and()
8             .arePublic()
9         .should()
10            .notBeAnnotatedWith(RequestMapping.class)
11        .because(
12            "Controller endpoints should not be annotated with @RequestMapping. "
13            +
14            "Use @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and
15            @PatchMapping instead."
16        );

```

- **Constructors:** These rules help enforce correct instantiation patterns.

```

1 @ArchTest
2 static final ArchRule controllersShouldHavePublicConstructors =
3     constructors()
4         .that()
5             .areDeclaredInClassesThat()
6                 .resideInAPackage("..controller..")
7         .and()
8             .areDeclaredInClassesThat()
9                 .areNotAnonymousClasses()
10        .should()
11            .bePublic()
12        .because("Classes in the controller package must expose only public
13        constructors.");

```

- **Layers:** It's helpful to define how different parts of the application can interact.

```

1 @ArchTest
2 static final ArchRule architectureStructure =
3     layeredArchitecture()
4         .consideringAllDependencies()
5
6         .layer("Controller").definedBy("..controller..")
7         .layer("Service").definedBy("..service..")
8         .layer("Repository").definedBy("..repository..")
9
10        .whereLayer("Controller")
11            .mayNotBeAccessedByAnyLayer()
12
13        .whereLayer("Service")
14            .mayOnlyBeAccessedByLayers("Controller", "Service")
15
16        .whereLayer("Repository")
17            .mayOnlyBeAccessedByLayers("Service")
18
19        .because("Controllers should not be accessed by other layers, " +
20            "services should only be accessed by controllers or services, " +
21            "and repositories should only be accessed by services");

```

Many other validations can be implemented using this library, as ArchUnit allows teams to define fully custom rules.

How to Extract and Share the Rules?

Defining the rules using ArchUnit is just the first step in the process. It's not a good idea to explain the rules for each microservice, as it leads to problems such as duplicate code, difficulty keeping the rules up to date, and more. So it's relevant find a way to extract all the rules and create a library. To do this, there is a plugin developed by Société Générale that provides the possibility to create a library with all the rules previously defined with the following POM file:

```

1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
4          xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6      <artifactId>flights-architecture-test</artifactId>
7      <version>1.0.6</version>
8      <properties>
9          <main.basedir>${project.basedir}</main.basedir>
10         <!-- General -->
11         <archunit-junit5.version>1.2.1</archunit-junit5.version>
12         <objenesis.version>3.3</objenesis.version>
13         <junit-jupiter.version>5.8.2</junit-jupiter.version>
14         <junit-platform-engine.version>1.9.3</junit-platform-engine.version>
15         <!-- Maven -->
16         <arch-unit-build-plugin-core.version>3.1.1</arch-unit-build-plugin-core.version>
17     </properties>
18     <dependencyManagement>
19         <dependencies>
20             <dependency>
21                 <groupId>org.objenesis</groupId>
22                 <artifactId>objenesis</artifactId>
23                 <version>${objenesis.version}</version>
24             </dependency>
25             <dependency>
26                 <groupId>org.junit.jupiter</groupId>
27                 <artifactId>junit-jupiter</artifactId>
28                 <version>${junit-jupiter.version}</version>
29             </dependency>
30             <dependency>
31                 <groupId>org.junit.platform</groupId>
32                 <artifactId>junit-platform-engine</artifactId>
33                 <version>${junit-platform-engine.version}</version>
34             </dependency>
35             <dependency>
36                 <groupId>com.tngtech.archunit</groupId>
37                 <artifactId>archunit</artifactId>
38                 <version>${archunit-junit5.version}</version>
39             </dependency>
40         </dependencies>
41     </dependencyManagement>
42     <dependencies>
43         <dependency>
44             <groupId>com.societegenerale.commons</groupId>
45             <artifactId>arch-unit-build-plugin-core</artifactId>
46             <version>${arch-unit-build-plugin-core.version}</version>
47             <exclusions>
48                 <exclusion>
49                     <groupId>junit</groupId>
50                     <artifactId>junit</artifactId>
51                 </exclusion>
52                 <exclusion>
53                     <groupId>org.junit.platform</groupId>
54                     <artifactId>junit-platform-commons</artifactId>
55                 </exclusion>
56             </exclusions>
57         </dependency>
58     </dependencies>

```



```

58         <groupId>com.tngtech.archunit</groupId>
59         <artifactId>archunit-junit5</artifactId>
60         <version>${archunit-junit5.version}</version>
61     </dependency>
62 </dependencies>
63 <repositories>
64     <repository>
65         <id>maven2</id>
66         <name>maven2</name>
67         <url>https://repo.maven.apache.org/maven2</url>
68     </repository>
69     <repository>
70         <id>manning-twa-lib-parent</id>
71         <name>Repository of parent pom</name>
72         <url>https://mymavenrepo.com/repo/tJqfV65wRD0LL9UcK4CQ</url>
73     </repository>
74 </repositories>
75 <distributionManagement>
76     <repository>
77         <id>flights-architecture-test</id>
78         <name>Repository of parent pom</name>
79         <url>https://mymavenrepo.com/repo/9lCyx3X47eTPtjxi1FA8</url>
80     </repository>
81 </distributionManagement>
82 </project>

```

As a recommendation, check the latest version of this plugin on the official webpage or a repository like [this regularly](#).

The previously defined rules need minor modifications to be usable by the plugin. The first one needs to reside in the folder that contains the source code; for Maven projects, in the *src/main/java* folder. The second modification requires creating one class per rule, since multiple rules cannot be in a single class.

The following code block shows what a rule looks like that checks the location and the name of the controllers on the application using the approach necessary to externalize the rules:

```

1  public class ControllerShouldBeNameProperlyRuleTest implements ArchRuleTest {
2      @Override
3      public void execute(String packagePath, ScopePathProvider scopePathProvider,
4                          Collection<String> excludedPaths) {
5          classes().that().resideOutsideOfPackage("..controller..").should()
6              .haveSimpleNameNotEndingWith("Controller")
7              .because("Classes in ..controller.. package should be named with
8                  Controller suffix")
9              .check(ArchUtils.importAllClassesInPackage(scopePathProvider.
10                  getMainClassesPath(), packagePath));
11     }
12 }

```

The structure of the test that resides inside the method executed is the same as what is possible to write in Archunit, so the most significant change to externalize the rules is create one class for each rule and nothing more.

The next step after migrating the rules into the library is building and uploading them to a repository accessible to anyone in the company, and introducing the modifications in the applications to use them. On the different project, the changes are reduced to include the following block:

```

1  <plugin>
2      <groupId>com.societegenerale.commons</groupId>
3      <artifactId>arch-unit-maven-plugin</artifactId>
4      <version>${arch-unit-maven-plugin.version}</version>
5      <configuration>
6          <rules>
7              <preConfiguredRules>
8                  <!-- Naming rules -->
9              </preConfiguredRules>
10         </rules>
11     </configuration>
12 </plugin>

```

```

10    <rule>com.twa.flights.architecture.rules.resource.
    ResourceShouldBeNameProperlyRuleTest</rule>
11    </preConfiguredRules>
12  </rules>
13 </configuration>
14 <executions>
15   <execution>
16     <phase>test</phase>
17     <goals>
18       <goal>arch-test</goal>
19     </goals>
20   </execution>
21 </executions>
22 <dependencies>
23   <dependency>
24     <groupId>com.societegenerale.commons</groupId>
25     <artifactId>arch-unit-build-plugin-core</artifactId>
26     <version>${arch-unit-build-plugin-core.version}</version>
27   </dependency>
28   <dependency>
29     <groupId>com.twa.flights.common</groupId>
30     <artifactId>flights-architecture-test</artifactId>
31     <version>${flights-architecture-test.version}</version>
32   </dependency>
33   <dependency>
34     <groupId>com.tngtech.archunit</groupId>
35     <artifactId>archunit-junit5</artifactId>
36     <version>${archunit-junit5.version}</version>
37   </dependency>
38   <dependency>
39     <groupId>com.tngtech.archunit</groupId>
40     <artifactId>archunit-junit5-api</artifactId>
41     <version>${archunit-junit5.version}</version>
42   </dependency>
43 </dependencies>
44 </plugin>

```

It's possible to enable rules one by one or all the rules in the same package.

Challenges and Best Practices

Introducing a new mechanism of validation, as usual, has some challenges that need to be considered, such as:

- **Execution time:** Adding more tests—whether run locally or in a CI pipeline—can increase build times and potentially delay deployments. One way to mitigate this is to introduce a dedicated profile and execute ArchUnit tests only when needed, or to run them in a separate pipeline stage that is not part of the production deployment flow.
- **Code duplication:** Many architectural rules refer to the same packages, layers, or classes, leading to repeated string literals across tests. A standard solution is to centralize these definitions in a shared constants class, making rules easier to maintain and reducing duplication.
- **Legacy code:** Enabling all architectural rules at once in a legacy project can surface many violations and require disruptive changes. A more practical approach is incremental adoption: enable the rules that do not cause immediate issues first, and progressively address the remaining ones. Prioritize rules that require minimal effort while delivering clear architectural improvements.
- **Keeping rules up to date:** Architectural rules are not static; they evolve as platforms and standards change. For this reason, it is essential to notify teams whenever a new version of the rules library is released and to ensure that applications do not continue using outdated versions. Without proper visibility, legacy rule sets can silently persist across the organization. One effective way to address this is to use tools such as Dependency-Track, which enable the identification of the dependencies and versions used by each application. This visibility helps teams detect services that still rely on legacy versions of the architectural rules and plan upgrades in a controlled, consistent manner.

These points represent only a subset of the challenges that may arise in day-to-day development. Still, it is essential to recognize that ArchUnit addresses architectural concerns that other tools are not designed to cover.

What's next?

There are many resources on the quality and structure of applications. Still, just a few tackles, solving or mitigating it without thinking of one particular tool. The following is just a short list of resources:

- [Fundamentals of Software Architecture: A Modern Engineering Approach](#) by Neal Ford
- [Modernizing Enterprise Java: A Concise Cloud Native Guide for Developers](#) by Markus Eisele
- [Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures](#) by Neal Ford
- [Refactoring: Improving the Design of Existing Code](#) by Martin Fowler

Other resources that could be great for understanding some concepts related to good practices and the relevance of the architecture of an application:

- [Clean Code: A Handbook of Agile Software Craftsmanship](#) by Robert C. Martin
- [Good Code, Bad Code](#) by Tom Long
- [Building Evolutionary Architectures: Automated Software Governance](#) by Neal Ford
- [Re-Engineering Legacy Software](#) by C. Birchall
- [SonarQube in Action](#) by G. Dubois, A. Delmas, and M. Rodiere
- [Clean as You Code: Elevating Software Quality through Continuous Code Improvement](#) By Peter McKee

This is just a small list of available resources for defining and validating an architecture. If something is unclear, find another video, book, or resource.

Conclusion

Problems related to an application's structure will inevitably arise because nothing in a software ecosystem remains static—especially in a platform composed of multiple microservices that evolve independently. This is why it is essential to have a mechanism that automatically detects structural issues, without relying on manual checks, at least during the application's verification process. Naturally, someone within the organization must define the architectural rules that every project should follow and express them in a way that ArchUnit can enforce.

Another significant challenge is ensuring that teams are aware when new versions of these rules are released and encouraging everyone to update them. Not all rules can be applied immediately, particularly when dealing with legacy applications, but an iterative approach can work. Teams can gradually enable rules, begin the standardization process, and progressively align the structure of all applications—while preserving the differences that belong only to each service's internal logic.

› [Back to the Table of Contents](#) ‹



When Developers Meet Designers: Lessons from Vibe Coding and the Art of Creative Expression

AI # CreativeCoding # DataVisualization # VibeCoding

Author: Csilla Szántó

Csilla Szántó is a professional software engineer specializing in web application development and data-driven solutions. She has worked in the industry and for consulting companies across various European countries, and brings experience in working for research and public service institutes. Her interests include inspiring AI and visualization projects, as well as engaging with software communities. In her free time, she enjoys hiking in the mountains.



LinkedIn

Have you ever wondered what happens when you mix climate data, artificial intelligence, and a group of creative people? I recently found out at a “**Future Fridays**” workshop, a collaboration between infogr8 and the Data Visualization Society. The goal was to experiment with AI-assisted creative coding to visualize climate data.

It was a fascinating experience in which data creatives and designers—along with a few developers—came together. We wanted to create visuals that do not just display numbers, but actually spark emotion and inspire action. In this article, I’ll share what I learned about bringing data to life and why you, as a developer, should care about AI-assisted coding.

What Is Creative Coding?

So, how can we define creative coding? Simply put, the goal of creative coding is to create an expressive animation that resonates emotionally. It’s not just about function; it’s about feeling. A key characteristic of this approach is the use of **randomness**. But how do you get from a simple idea to a complex animation?

From Idea to Animation

The process usually looks like this:

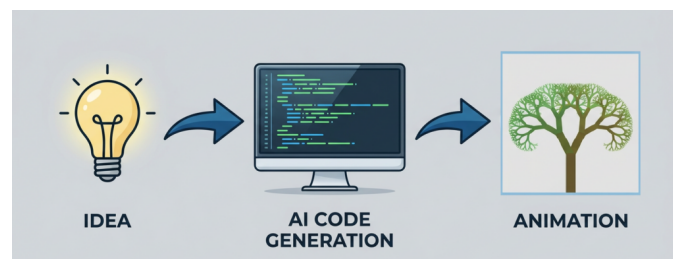


Abb. 16: AI-generated illustration

You start with an idea. Then you want to turn that idea into a simulation or an animation. The “bridge” between your thought and the final result is code generation. Here, programming becomes a method of producing art.

Even if you do not know a specific language, AI models can help you write the code. This is where we get the term “vibe coding.” It means you accept the code the AI generates as it is, purely to get the “vibe” or look you want. In real life, you may want to check the generated code, but we won’t focus on that now. The code in this context is a medium—a tool that creates shapes, text, or images. The use of AI models lowers the barrier to entry significantly.

The Magic of Randomness: Perlin Noise

One of the most important algorithms in creative coding is **Perlin noise**, created by Ken Perlin.

Its magic lies in its ability to transition smoothly between points. It avoids those sharp, unrealistic changes that would result in unnatural textures.

It creates **organic-looking randomness** that feels natural to the human eye. This allows for:

- Complex textures.
- Natural-looking landscapes (rolling hills, mountains).
- Weather effects like clouds, rain, water, smoke, and fire.
- Fluid movements like waves or dust particles.

In texture synthesis, Perlin noise helps you avoid boring repetition. It allows you to use your creativity not just in selecting the right algorithm, but in how you apply it to make something unique.

Defining the Practice

To formulate it another way, creative coding is the practice of using code as an art and design tool. It produces expressive outputs like visuals, music, and interactive experiences.

Here are the key aspects you should know:

- **Artistic Focus:** Unlike traditional programming, which prioritizes functionality, this prioritizes aesthetics, beauty, emotion, and creative expression.
- **Diverse Applications:** It's used in digital art, game development, live visuals, art installations, projection mapping, and advertising.
- **Interactive Experiences:** You can create pieces that respond to sound, data, or user input.
- **Creative Problem-Solving:** It encourages you to find unconventional solutions and develop your creative thinking.

Why Should a Software Engineer Care?

You might be asking, “I’m a back-end developer. I write logic. Why do I need to know about creative coding?” That is a fair question. Here is why it is worth your time:

- **New Perspectives:** It helps you explore new ways people perceive data. You can create designs that do not just function but encourage the audience to reflect and think deeply about a topic. You can bring this understanding of “emotional engagement” and creative coding back to your daily work.
- **Better Front-ends:** If you want to move beyond back-end-heavy apps, this is key. Creative coding teaches you about aesthetics, color theory, and choosing the right typefaces. It helps you make harmonious user interfaces that give your customers an extra artistic touch. It's one step beyond what you learned in computer science school.
- **Broadening Knowledge:** It is important to know what is possible. You do not need to be an expert, but creating a mental map of these tools is useful. When you eventually need a form of creative expression, you will know exactly where to look.

Implementing Tools

While you can use almost any language for this, some frameworks or libraries are built specifically for creative work. Famous examples include Processing, p5.js, and openFrameworks.

Why p5.js?

For our workshop, we focused on p5.js. It is a client-side JavaScript platform for artists, designers, and students. It is based on the core principles of Processing but built for the web.

The Features:

- It is a JavaScript library, meaning it runs directly in the browser.
- It integrates easily with HTML/CSS.
- It allows for real-time collaboration.
- It is very accessible for beginners.

The Weaknesses:

- It has performance limitations for highly complex projects.
- It depends on the browser's capabilities.

Basically, p5.js is an **implementation of the creative coding concept**. You do not need prior knowledge of complex visualization libraries. You can start immediately.

Why it's ideal for beginners:

- It is open source and beginner-friendly.
- It supports randomness and noise (like the Perlin noise we discussed).

- It is perfect for animations.
- You get immediate results via the online editor.

You can find tutorials on [p5.js's website](https://p5js.org/) or inspiring projects on [OpenProcessing](https://openprocessing.org/).

Using AI with p5.js is surprisingly easy. You can instruct an AI model to generate code, paste it into the [p5.js online editor](https://p5js.org/), and immediately preview the animation.

With p5.js you can:

- Build shapes and patterns.
- Create frame-based animations.
- Attach input data (such as a CSV file).
- Add sound or video.
- Play with text and images.

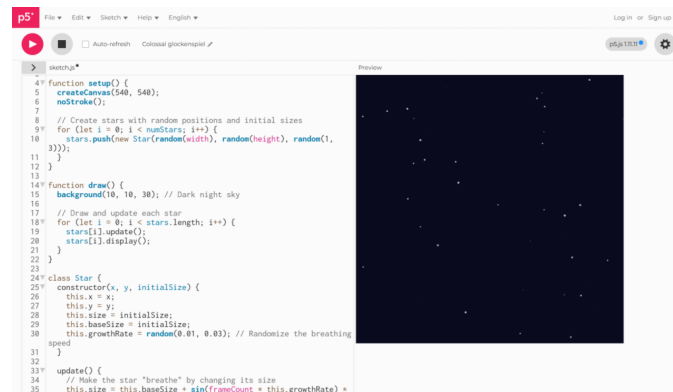


Abb. 17: Screenshot of the p5.js editor

How to Do Creative Coding?

Let's look at how we actually approached the creative process during the "Future Fridays" workshop. The steps were accessible even if you aren't a designer and are quite similar to those that software developers or data scientists are familiar with.

Foundational Work

- **Goal & Audience:** Our goal was to create an engaging animation to make climate data approachable for a broad, non-scientific audience. The target medium was an Instagram Story. We needed a narrative that was simple, clear, emotional, and thought-provoking.
- **Data Source:** We explored various climate datasets. Many participants focused on sea-level rise. I chose snow measurement data from Switzerland (open government data).

My hypothesis was simple: Warming winters lead to shorter, less snowy winters.

My message became: "The snowman is shrinking."

- **Storytelling & Constraints:** The challenge was communicating this idea simply. The animation had to fit Instagram's dimensions and last about 30 seconds to keep the audience engaged. The goal was to leave the audience surprised and reflective.

Cleaning & Analyzing the Data

I started by cleaning the data and creating basic charts. I noticed that high-altitude stations had more variation in snow height, but lower altitudes were more consistent.

However, I couldn't detect a clear trend without additional data necessary to draw a scientifically solid conclusion.

Here is where my process differed from normal development: In a typical project, analysis comes first, then insights. Here, I was driven by my personal experience with nature.

I acknowledged that this project wasn't a scientific paper. I simplified the data processing to fit the narrative, while ensuring I didn't present misleading conclusions. It was a deliberate balance between accuracy and storytelling.

Sketching Designs

- **Mood Boards:** We created mood boards to collect visual inspiration. I learned that a mood board isn't just random images. It defines the "look and feel" of your concept, including font pairings and color palettes that match your emotional tone. We used OpenProcessing to find unconventional shapes and effects.
- **Sketching:** Sketching helps you experiment with different design options. I mapped the median number of snowy days to the size of a snowman. More snow equals a bigger snowman. The selection of winter colors was key to reinforcing the theme.

There are plenty of digital tools available for sketching. I realized that choosing the right typeface is essential: it harmonizes the design.

This process is iterative; every step builds on the previous one.

Creating & Delivering the Visualization

In the final phase, I turned my sketch into a simple animation. This involved vibe coding. I refined AI prompts to get the design I wanted and tested them in the p5.js online editor.

Unlike normal coding, I didn't focus on reading or writing every line myself. The focus was on the results. Many designers did the same. As most of them weren't developers, they couldn't judge the code for correctness. You need to be precise in formulating your requirements to get the right results.

I checked the constraints (Instagram safe zones) and finalized the video. In the real world, you'd typically show your progress to the client regularly to gather feedback. This helps ensure you're on track, adjust the design if needed, and keep the client engaged.

For our Future Fridays workshop, instead of client presentations, we shared the creative process and the final product within the group. The result? A short video animation for Instagram.

Comparing the Results

This was the most interesting part: comparing the work of designers versus developers.

How were the designers' results?

- Most had little to no coding experience.
- They showed incredible imagination and **creativity**. It proved their **deep experience** with the creative process.
- They were fearless: eager to try AI code generation.
- They were **curious**, experimented a lot, and had a lot of fun.
- They had **great patience**, iterating until the result matched their vision exactly.
- Their work captured the emotion and urgency of climate change through color and composition.

How was my result, as a software developer?

- I had a **strong concept**, an appealing visual design, and a thoughtful use of texture.
- I understood the **code** perfectly and could read it easily.
- However, I had less patience with iterations. I delivered a "pragmatic" result due to time constraints (do we developers always leave delivery for the last minute?).
- My data storytelling was less clear than theirs.

The most valuable part for me was seeing the designers' creative process, learning from their thinking, and understanding their decisions.

Interpreting the Results

What does this tell us?

I realized that while vibe coding (prompting AI) wasn't difficult for me, the designers delivered a result with more emotional impact.

Designers mastered the **art of creative expression**. Their experience, creativity, workflow, and affinity for aesthetics gave them an edge. They knew instinctively which colors and shapes would work and what designer tool to choose.

Developers are better at programming and are more pragmatic. However, they think less about the emotions a visualization might provoke.

But my takeaway is that developers can catch up. Our technical foundation is strong. We can learn to make better visualizations by reverse-engineering existing art and learning to see with a designer's eye.

Is AI Replacing Professionals?

Is AI going to replace developers, designers, or data storytellers?

No, at least not at this stage.

AI tools and methods like vibe coding don't replace skilled professionals; they amplify them.

- Designers use AI to execute complex ideas they couldn't code before.
- Developers use AI to bridge the gap in visual creativity.

The emotional richness, taste, and art direction come from you. They are **unique to the individual**. Creativity and the ability to spark emotions are still your privilege. An AI cannot come up with that human spark by itself.

If you are an experienced professional, AI is an indispensable tool to master in order to enhance your work.

If you are a beginner, you need a solid foundation to judge the results AI gives you. Without the basics, you will be left guessing.

Combining Approaches: Giving You Power

I see a lot of potential for creating truly powerful, engaging, and effective visualizations when you combine the "Design Vibe" with "Developer Logic."

We currently have two very strong skill sets:

- **Software engineers' strengths:** We bring analytical thinking, critical code analysis, automated tests, API integration, real-time data delivery knowledge, scalability, and performance. We make things work at scale and sometimes take a pragmatic approach.
- **Designers' strengths:** They bring the "perfect" aesthetic, emotional storytelling, a design that reflects the nature of data, curiosity, and deep creativity. They make things feel right.

Both groups start with a concept, can analyze the data, know their audience, and practice iterative implementation.

Why not let them work together—for instance, when building an interactive web application with advanced graphical requirements.

How to Benefit from This Combination

A Combined Approach for Stronger Visualizations

When designers and developers collaborate, the result is a more effective and engaging visualization. Designers bring expertise in visual storytelling, while developers ensure technical reliability. By working together early, both can test ideas, identify potential issues, and ensure a seamless experience.

How to Work Together

- Designers: Define the emotional journey of the user. How should the experience feel—surprising, reassuring, or empowering? Reflect this in the design and interactions.
- Developers: Ensure that interactions are fast, responsive, and stable, even with large or complex data sets.

Why This Works

- Balanced Results: The final product blends emotional engagement with technical robustness.
- Faster Problem-Solving: Issues with design or performance are detected early, saving time and reducing rework.
- Better Engagement: User interactions—whether hovering, filtering, or exploring data—should feel like a guided journey, enhancing the experience.

This approach keeps your users engaged and in control, benefiting from a smooth and thoughtful experience created by both creative and technical strengths.

Improving as a Developer Without a Designer

If you're working without designers, you can still improve your visualization skills by learning from design principles. Study concepts like color theory, visual channels, hierarchy, and layout, and explore visualizations you admire.

Reverse-engineer the designs: What choices did the designer make? Why use one technique or form over another? Understanding these decisions helps you improve your own work.

Experiment and explore new approaches when time allows. You don't need to become a designer, but having a grasp of fundamental principles will help you create more effective, engaging visualizations.

A Final Thought

The workshop taught me that code is more than just logic; it is a medium for expression. Whether you are building a massive back-end or a small web animation, remembering the human element makes your work better.

So, here is my challenge to you: Next time you have a free afternoon, open up p5.js, prompt an AI to give you some "Perlin noise code," and see what happens. You might just surprise yourself with what you create.

Further sources:

[Snowman animation turned into a mini web application](#)

[Free eBook - my data visualization checklist](#)

› [Back to the Table of Contents](#) ‹



JCON2026

JCON EUROPE 2026
International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO

www.jcon.one



Scripting on the JVM with Java, Scala, and Kotlin

#JVMLanguage

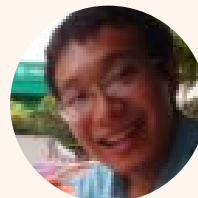
#Java

#Scala

#Kotlin

Author: Haoyi Li

Li Haoyi graduated from MIT with a degree in Computer Science and Engineering, since then has built core infrastructure for high-growth companies like Dropbox and Databricks, and has been a major contributor to the open source community. His projects have over 10,000 stars on Github, and are downloaded over 20,000,000 times a month. Haoyi has deep experience in the JVM and has professionally built distributed backend systems, programming languages, high-performance web applications, and much more.



LinkedIn

This article was originally published on the mill technical blog at <https://mill-build.org/blog/19-scripting-on-the-jvm.html>

Lots of people write code in Java, Scala, or Kotlin, but that code is normally relegated to large enterprise codebases. This article will explore the pros and cons of using JVM languages for small-scale scripting instead, as an alternative to Python, Javascript, or Bash:

- How the performance and compile-time safety of the JVM are advantages over scripting languages
- How verbosity, build tool overhead, and lack of script-focused libraries causes issues

Lastly we will discuss how lightweight languages, tooling, and script-focused libraries is able to smooth over some of those issues, making the JVM truly a world-class scripting environment as robust and convenient as any scripting language out there.

Example Java Single-File Script

Consider this small Java program that someone may write. This program uses [unirest-java](#) and [jackson-databind](#) to fetch the links of a wikipedia page via the wikipedia API, which it then uses to perform a breadth-first traversal of the wikipedia article graph. The user provides the starting input via command-line articles parsed with [picocli](#).

JsonApiClient.java

```

1  @CommandLine.Command(name = "Crawler", mixinStandardHelpOptions = true)
2  public class JsonApiClient implements Callable<Integer> {
3
4      @CommandLine.Option(names = {"--start"}, required = true, description = "Starting title")
5      private String start;
6
7      @CommandLine.Option(names = {"--depth"}, required = true, description = "Depth of crawl")
8      private int depth;
9
10     private static final ObjectMapper mapper = new ObjectMapper();
11
12     public static List<String> fetchLinks(String title) throws Exception {
13         var response = Unirest.get("https://en.wikipedia.org/w/api.php")
14             .queryString("action", "query")
15             .queryString("titles", title)
16             .queryString("prop", "links")
17             .queryString("format", "json")
18             .header("User-Agent", "WikiFetcherBot/1.0 (https://example.com; contact@example.com)")
19             .asString();
20
21         if (!response.isSuccess())
22             throw new IOException("Unexpected code " + response.getStatus());
23
24         var root = mapper.readTree(response.getBody());
25         var pages = root.path("query").path("pages");
26         var links = new ArrayList<String>();
27
28         for (var it = pages.elements(); it.hasNext(); ) {
29             var linkArr = it.next().get("links");
30             if (linkArr != null && linkArr.isArray()) {
31                 for (var link : linkArr) {
32                     var titleNode = link.get("title");
33                     if (titleNode != null) links.add(titleNode.asText());
34                 }
35             }
36         }
37         return links;
38     }
39 }

```

```

40 public Integer call() throws Exception {
41     var seen = new HashSet<>(Set.of(start));
42     var current = new HashSet<>(Set.of(start));
43
44     for (int i = 0; i < depth; i++) {
45         var next = new HashSet<String>();
46         for (var article : current) {
47             for (var link : fetchLinks(article)) {
48                 if (!seen.contains(link)) next.add(link);
49             }
50         }
51         seen.addAll(next);
52         current = next;
53     }
54
55     try (var w = Files.newBufferedWriter(Paths.get("fetched.json"))) {
56         mapper.writerWithDefaultPrettyPrinter().writeValue(w, seen);
57     }
58     return 0;
59 }
60
61 public static void main(String[] args) {
62     System.exit(new CommandLine(new JsonApiClient()).execute(args));
63 }
64 }

```

While this example is a toy program, it is representative of the many kinds of small programs or scripts that people typically use scripting languages for: simple code gluing together various data sources and APIs and performing some simple computation based on them.

Typically these are written in Python, Javascript, or Bash. The example above in contrast is written in Java, which is not very common. Why would someone want to do that, and what are the challenges of doing so?

Benefits of Scripting on the JVM

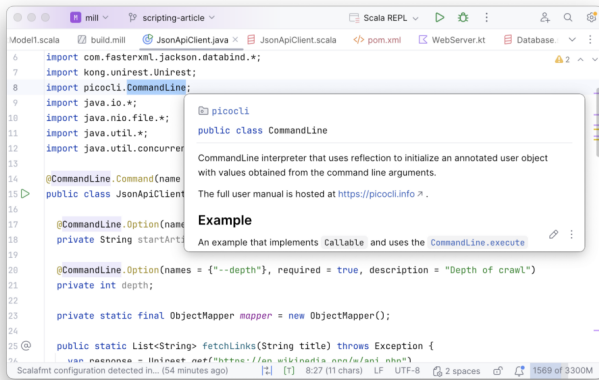
While the JVM isn't traditionally thought of as a scripting platform, in reality there are many benefits from writing and running your scripts on the JVM:

Library Ecosystem

Java libraries are mostly hosted on [Sonatype Central](#), which is one of the largest and most comprehensive set of third-party libraries of any programming language. This includes high-quality well-maintained libraries such as Jackson, Unirest, and PicoCLI that we used above. Other JVM languages like Kotlin or Scala host their libraries on the same platform, which allows them to interoperate and use each other's libraries. If a library exists anywhere in the world, it is likely you will find a Java version of it on Sonatype Central for you to use.

IDE Support

Java is known for having great IDEs such as IntelliJ that make it much easier to navigate, understand, and work with your codebase. This is just as helpful for small scripts as it is for larger applications, as even in scripts you may be working with unfamiliar third-party libraries for which IDE assistance can be especially helpful.



Toolability

Beyond the IDE, the JVM probably has the best tools of any programming ecosystem. From simple things like [jstack](#) to debug stuck processes, to [Yourkit](#) or [JProfiler](#) to investigate tricky performance issues. Many of these come in handy when working with small scripts, which like application code can get stuck or become slow and need investigation and debugging.

Performance

While people often neglect the performance of scripts and small programs, just as often you find people complaining that poorly-optimized scripts are slow and taking longer than they need to. The JVM not only provides excellent single-core performance that's 10-20x faster than traditional scripting languages, it also allows you to easily spawn threads to parallelize your script across multiple cores. This makes it much easier to optimize and fix slow scripts on the JVM than when running on slower single-threaded runtimes like Node.js or Python.

Compile-time Checking

Scripts have a reputation for being fragile and easy to break. This is because they are often poorly tested, for a variety of reasons:

- They are often ad-hoc glue code between different systems, and evolve rapidly as requirements change
- They often interact with external files, processes, or APIs that may be difficult to mock out
- They can run relatively slowly, making automated testing slow and expensive

Writing your scripts in a JVM language like Java, Scala or Kotlin doesn't solve the testing problems above, but it does give you one more tool to lean on: the compiler checking that all your method calls and parameter types are properly wired up. While this won't catch every bug, it at least provides an additional guardrail helping prevent silly mistakes and nudging your scripts towards correctness.

Challenges of Small-Scale Java Programs

While there are many benefits to writing scripts running on the JVM, in practice it can be quite challenging to effectively use Java or other JVM languages to write scripts.

Script Verbosity

The first thing you may notice is: the program above is pretty verbose! Although the Java language has gotten more concise over the years, with features like the `var` keyword that we make good use of in the script above, overall Java code tends to still be significantly more lines than a typical scripting language. While the code above seems *fine*, it might not seem elegant enough that you would choose to write your scripts in Java on the JVM rather than Python.

Tricky installation

The next thing you might deal with writing scripts is how to run them: you need `java`, and you need a build tool like [Maven](#)

- Installing Java can be surprisingly tricky ([link](#)), with a lot of subtle choices and decisions and ways things can go wrong. Installing Maven similarly can take some fiddling
- Installing Java and Maven on all environments your script may run on is more difficult: you need to ensure every laptop, CI work, or production machine has the same exact versions of Java and Maven installed.
- Most difficult is maintaining this installation over time: as laptops get provisioned and retired, Java or Maven versions get upgrades, CI workers and production servers get cycled out, ensuring that every machine has the right version of Java and Maven installed at all times can be quite challenging.

Installing Java and Maven is not a blocker. But it is certainly something that can be a lot more fiddly than expected, especially when you have to ensure consistency over a fleet of heterogeneous machines and ensure it stays consistent over time.

Configuring Maven is Tedious

Once you have Java and Maven installed, you then need to configure Maven to build it. The script above has three third-party dependencies, so the Maven config looks something like this:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
           xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>com.example</groupId>
8      <artifactId>json-api-client</artifactId>
9      <version>1.0.0</version>
10     <packaging>jar</packaging>
11
12     <properties>
13         <maven.compiler.source>17</maven.compiler.source>
14         <maven.compiler.target>17</maven.compiler.target>
15         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
16     </properties>
17
18     <dependencies>
19         <dependency>
20             <groupId>info.picocli</groupId>
21             <artifactId>picocli</artifactId>
22             <version>4.7.6</version>
23         </dependency>
24         <dependency>
25             <groupId>com.konghq</groupId>
26             <artifactId>unirest-java</artifactId>
27             <version>3.14.5</version>
28         </dependency>
29         <dependency>
30             <groupId>com.fasterxml.jackson.core</groupId>
31             <artifactId>jackson-databind</artifactId>
32             <version>2.17.2</version>
33         </dependency>
34     </dependencies>

```



```

1      <build>
2          <plugins>
3              <plugin>
4                  <groupId>org.codehaus.mojo</groupId>
5                  <artifactId>exec-maven-plugin</artifactId>
6                  <version>3.4.1</version>
7                  <configuration>
8                      <mainClass>JsonApiClient</mainClass>
9                  </configuration>
10             </plugin>
11         </plugins>
12     </build>
13 </project>

```

While this Maven config isn't particularly complicated, it is quite a screenful of text we need to write or copy-paste from somewhere in order to get our small script running. Again, not a blocker, but more fiddly stuff to set up and maintain just to write our single-file Java program.

Running Maven is Tedious

Lastly, we need to run our script with Maven. The command to do so is as follows:

```
1 mvn compile exec:java -Dexec.args="--start Functional_programming --depth 2"
```

The single-line command for running your Maven project has a whole bunch of footguns you need to avoid:

- Remember to `compile` first, otherwise your compiled code may be out of date!
- Remember to `exec:java` rather than just `exec`!
- Remember to pass in arguments via `-Dexec.args` rather than passing them indirectly!
- If your arguments themselves need to be quoted, remember to double-escape your nested quotes, or triple-escape them if the arguments themselves contain quotes!

This may seem nitpicky - and it is! - but it's just another papercut in making it annoying to run the `JsonApiClient.java` file we have already written.

None of these issues with Maven or Java are fatal. We've all been writing and running Java programs for decades by now, in my case since Java 4 in 2004. But the sum-total of all these papercuts is that it's not pleasant to write small programs in Java. Even modern AI assistants have limited context windows and benefit from having short, concise code to work with, and don't benefit from digging through verbose boilerplate.

And so small programs and scripts end up being written in Python or Bash or Javascript. Even though in theory Java or Scala or Kotlin could work, in practice nobody uses them for scripting.

Improving the JVM Scripting Experience

While there are many issues with scripting on the JVM, these issues are not insurmountable. Next we'll discuss some of the solutions and workarounds that can mitigate these problems, to provide the streamlined scripting experience that the JVM deserves

Mill as a Lightweight Build Tool

The first thing we can do to simplify our script workflow is to use a lighter weight build tool such as [JBang](#) or [Mill](#). These tools make it much easier to configure some third-party dependencies and compile/run a single file. For example, JBang lets you write [//DEPS header comments](#) and run the `.java` file directly from the command line:

JsonApiClient.java

```

1 //DEPS info.picocli:picocli:4.7.6
2 //DEPS com.konghq:unirest-java:3.14.5
3 //DEPS com.fasterxml.jackson.core:jackson-databind:2.17.2
4 import com.fasterxml.jackson.databind.*;
5 import kong.unirest.Unirest;
6 import picocli.CommandLine;
7 ...

```

```
1 jbang JsonApiClient.java -s Functional_programming -d 2
```

The Mill build tool uses a similar [YAML header syntax](#) that looks a bit different but otherwise works basically the same way:

JsonApiClient.java

```

1 /// mvnDeps:
2 /// - info.picocli:picocli:4.7.6
3 /// - com.konghq:unirest-java:3.14.5
4 /// - com.fasterxml.jackson.core:jackson-databind:2.17.2
5 import com.fasterxml.jackson.databind.*;
6 import kong.unirest.Unirest;
7 import picocli.CommandLine;
8 ...

```

```
1 ./mill JsonApiClient.java -s Functional_programming -d 2
```

Such lightweight build tools make it much easier to run our small Java program:

- Both JBang and Mill make it super easy to configure dependencies and run scripts from the command line by adding a header to the single-file Java program.
- Mill with its `./mill` bootstrap script also solves the tricky to install Java and Maven problem, as `./mill` will automatically download & cache the JVM and build-tool installation and ensure you are using the correct, consistent version

However, even if building and running small scripts written in Java is convenient, writing and maintaining the code itself can be a pain due to the verbosity of the Java language and libraries that makes even simple programs take pages and pages of code. While Java's verbosity may be fine - or even beneficial - for complicated application code, the same verbosity can get in the way of writing small scripts.

But Java isn't the only language on the JVM!

Kotlin as a Lightweight Language

One option here is to write the script in Kotlin. Kotlin is much more syntactically concise than Java, which means much less code to write the same things overall. A version of the [HtmlScraper.java](#) translated to Kotlin is shown below, using the Kotlin [Clikt](#) library rather than PicoCLI:

JsonApiClient.kt

```

1 /// mvnDeps:
2 /// - com.github.ajalt.clikt:clikt:5.0.3
3 /// - com.konghq:unirest-java:3.14.5
4 /// - org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3
5 import com.github.ajalt.clikt.core.CliktCommand
6 import com.github.ajalt.clikt.core.main
7 import com.github.ajalt.clikt.parameters.options.*
8 import com.github.ajalt.clikt.parameters.types.int
9 import kotlinx.serialization.json.*
10 import kong.unirest.Unirest
11 import java.nio.file.*

```

```

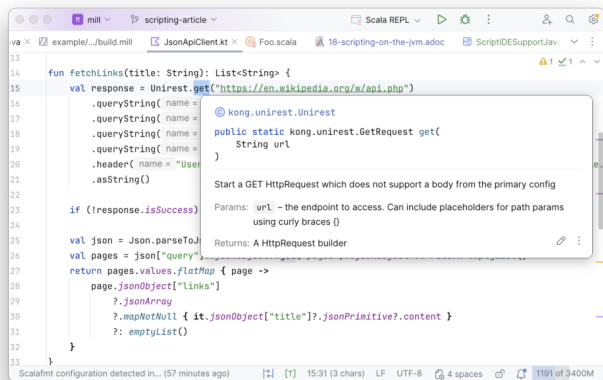
1  fun fetchLinks(title: String): List<String> {
2      val response = Unirest.get("https://en.wikipedia.org/w/api.php")
3          .queryString("action", "query")
4          .queryString("titles", title)
5          .queryString("prop", "links")
6          .queryString("format", "json")
7          .header("User-Agent", "WikiFetcherBot/1.0 (https://example.com; contact@example.com)")
8          .asString()
9
10
11     if (!response.isSuccess) return emptyList()
12
13     val json = Json.parseToJsonElement(response.body).jsonObject
14     val pages = json["query"]?.jsonObject?.get("pages")?.jsonObject ?: return emptyList()
15     return pages.values.flatMap { page ->
16         page.jsonObject["links"]
17             ?.jsonArray
18             ?.mapNotNull { it.jsonObject["title"]?.jsonPrimitive?.content }
19             ?: emptyList()
20     }
21 }
22
23 class Crawler : CliktCommand(name = "wiki-fetcher") {
24     val start by option(help = "Starting Wikipedia article").required()
25     val depth by option(help = "Depth of link traversal").int().required()
26
27     override fun run() {
28         var seen = mutableSetOf(start)
29         var current = mutableSetOf(start)
30
31         repeat(depth) {
32             val next = current.flatMap { fetchLinks(it) }.toSet()
33             current = (next - seen).toMutableSet()
34             seen += current
35         }
36
37         val jsonOut = Json { prettyPrint = true }
38             .encodeToString(JsonElement.serializer(), JsonArray(seen.map { JsonPrimitive(it) })))
39         Files.writeString(Paths.get("fetched.json"), jsonOut)
40     }
41 }
42
43 fun main(args: Array<String>) = Crawler().main(args)

```

```
1 ./mill JsonApiClient.kt --start Functional_programming --depth 2
```

The Kotlin program has about 1/3 fewer lines than the Java equivalent, and overall much less dense. Kotlin features like the properties used to define `val depth` and `val start`, `?`, and `.mapNotNull` simplify the code substantially. This makes it possible to express the same program in a much less verbose syntax.

As Kotlin is also a JVM language, it comes with all the same benefits as writing scripts in Java, e.g. the excellent IDE support provided by editors such as IntelliJ:



This makes using Kotlin a great way to streamline the scripting experience on the JVM. Although the example above uses Mill as the build tool, Kotlin also supports its own scripting [workflows](#), which are also used in Gradle and other projects.

Although scripts in Kotlin are markedly nicer to write and read than scripts written in Java, there is one more step further that we can take:

Scala with its Lightweight Libraries

The last step to simplify scripting on the JVM is to write the script in Scala. Scala is yet another JVM language, like Kotlin. But unlike Kotlin, Scala has many script-focused libraries such as [OS-Lib](#), [MainArgs](#), [Requests-Scala](#), [uPickle](#), or [PPrint](#) that make it very convenient to write small script-like programs in Scala. The above `HtmlScraper.kt` translated to an equivalent `HtmlScraper.scala` is shown below:

HtmlScraper.scala

```
1 def fetchLinks(title: String): Seq[String] = {
2   val resp = requests.get.stream(
3     "https://en.wikipedia.org/w/api.php",
4     params = Seq(
5       "action" -> "query",
6       "titles" -> title,
7       "prop" -> "links",
8       "format" -> "json"
9     )
10  )
11  for {
12    page <- ujson.read(resp)("query")("pages").obj.values.toSeq
13    links <- page.obj.get("links").toSeq
14    link <- links.arr
15  } yield link("title").str
16 }
17
18 def main(start: String, depth: Int) = {
19   var seen = Set(start)
20   var current = Set(start)
21   for (i <- Range(0, depth)) {
22     current = current.flatMap(fetchLinks(_)).filter(!seen.contains(_))
23     seen = seen ++ current
24   }
25
26   pprint.log(seen)
27   os.write(os.pwd / "fetched.json", upickle.stream(seen, indent = 4))
28 }
```

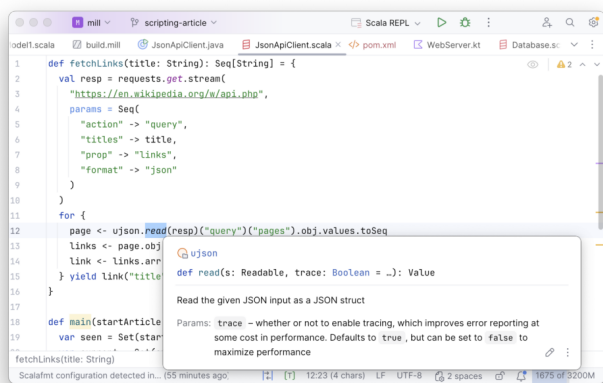
```
1 ./mill HtmlScraper.scala --start Functional_programming --depth 2
```

What is notable about `HtmlScraper.scala` is how much less *stuff* there is to read, with about 1/2 the lines of code as `HtmlScraper.kt` and 1/3 the lines of code as the `HtmlScraper.java`:

- The `requests.get`, `ujson.read`, and `os.write` APIs come from the Mill's bundled [libraries](#), which makes it super easy to interact with the filesystem, subprocess, and JSON APIs over HTTP
- Rather than parsing arguments via annotations or a special class, which is how it's done in PicoCLI or Clikt, `HtmlScraper.scala` uses [MainArgs](#) which lets you simply define a `def main` method and turns the parameter list into the command-line parser

In general, the Scala script we see above looks similar to any scripting language. It has code that specifies clearly the logical steps of accessing the Wikipedia API and performing the breadth-first search, but without the verbose machinery necessary to implement that logic in Kotlin or Java. And although the Scala program is much shorter and more concise than the Java program we started with, overall it still has all the benefits of running on the JVM:

- We can depend on any JVM library via `/// | mvnDeps`. Scala can make use of both Java and Scala-specific libraries, and so you can always find a library to do whatever you need to do
- All other JVM tools work with Scala just as easily as they do with Java: `jstack`, `Yourkit`, `JProfiler`, etc.
- Scala performance is just as good as Java performance, and it makes it even easier to parallelize things using `scala.concurrent.Future` so your scripts can make full use of the multiple cores available on any modern computer.
- We have full support in IDEs like IntelliJ or VSCode:



Parallelizing Our Scripts

Since our `JsonApiClient.scala` script is running on the JVM, we can take advantage of the JVM's support for threading to parallelize our web crawl. The example below does this using the `Future` API in the `scala.concurrent` package, that builds upon Java `Threads` to make each `fetchLinks` call run on a configurable thread pool:

`JsonApiClient.scala`

```
1 import scala.concurrent.*, duration.Duration, ExecutionContext.Implicits.global
2 def fetchLinks(title: String): Seq[String] = {
3   val resp = requests.get.stream(
4     "https://en.wikipedia.org/w/api.php",
5     params = Seq(
6       "action" -> "query",
7       "titles" -> title,
8       "prop" -> "links",
9       "format" -> "json"
10    )
11  )
12  for {
13    page <- ujson.read(resp)("query")("pages").obj.values.toSeq
14    links <- page.obj.get("links").toSeq
15    link <- links.arr
16  } yield link("title").str
17 }
```



```

18
19 def main(start: String, depth: Int) = {
20   var seen = Set(start)
21   var current = Set(start)
22   for (i <- Range(0, depth)) {
23     current = current
24       .map(c => Future{ fetchLinks(c) })
25       .map(Await.result(_, Duration.Inf))
26       .flatten
27       .filter(!seen.contains(_))
28
29     seen = seen ++ current
30   }
31
32   pprint.log(seen)
33   os.write(os.pwd / "fetched.json", upickle.stream(seen, indent = 4))
34 }

```

This kind of ad-hoc parallelism is something that is very difficult or messy to achieve in traditional scripting languages like Python, Bash, or Javascript. But when scripting on the JVM, threads are just a fact of life, and there are many builtin and third-party libraries that make it super easy to parallelize any scripts that may be a bottleneck in your developer workflow.

Conclusion

In this article, we discussed how scripts can benefit from the JVM's library ecosystem, IDE support, tools, and performance, but also how they are hampered by Maven's clunkiness, Java's verbosity, and the lack of script-focused libraries. We explored solutions to these issues, by running our scripts in Mill to simplify the workflow, porting them to Kotlin to cut down on the verbosity, and moving to Scala to take advantage of the many scripting-focused libraries in the Scala ecosystem.

Scripting on the JVM has never been very popular in the past, with projects like [JRuby](#) or [Jython](#) not taking off, and most JVM projects still relying on a mix of Python or Bash for their scripting needs. Even with the solutions discussed above, other issues such as the JVM's slow startup time and large memory footprint remain as challenges

Despite that, scripting on the JVM can definitely be a valuable tool in your toolbox. If you are interested in learning more about scripting in Java, Kotlin and Scala, check out the links to the documentation below:

Java Single-File Scripts	Java Script Use Cases
Kotlin Single-File Scripts	Kotlin Script Use Cases
Scala Single-File Scripts	Scala Script Use Cases

› [Back to the Table of Contents](#) ‹



From Fragile to Fearless

OpenSource

UnitTest

Author: Wouter Bauweraerts

Wouter is a passionate Java developer at The Beehive. As a full stack developer, he excels at solving complex problems efficiently and always focuses on quality. He is known for his creativity and forward-thinking. He enjoys experimenting with new frameworks and technologies to make the best decisions for future projects. Beyond coding, Wouter is dedicated to coaching and mentoring his colleagues. He helps them improve by discussing project challenges and opportunities. In the past, Wouter has also taught programming courses at Karel de Grote Community College in Antwerp. His practical experience and love for learning make him a valuable asset to his team and the IT community.



LinkedIn

I believe that every modern software developer by now realizes that writing tests is an indispensable part of our job. We also know that it's best to start writing our tests before we write the actual implementation. There are already many articles that elaborate on the test-driven development process. We will not investigate how our development process should work. Instead, we will investigate how we can write tests that help us create good software, without adding overhead.

Problem-Solving Assistant

Tests serve multiple purposes while we are working in our codebase. For starters, a test can help us gain more **understanding** of the problem at hand. By writing the test, we think about the small steps we can take to implement the solution step by step. Writing tests can also force us to think about the internal structure of our code in advance. This helps us structure our code so that we will still understand it in several months.

Functional Validation

One of the main reasons we write tests is to **validate** that the code we are writing is correct. By thinking about the expected outcome before we write the code, we know when we have reached our goal. Level up! Write the next test and continue implementing the feature you're working on. Good job!

Finally, our tests can also serve as some kind of **regression** suite. Especially when working in an agile environment, we won't be implementing the complete feature at once. We'll start with the easiest happy path, without adding all the complexities that may be required in the future. We will add additional requirements defined by what the evolving business needs are in the end, during the next iterations.

Let's say we are implementing the checkout process for a web shop. First, we calculate our total price, ignoring discounts and shipping costs. In the next sprints, we can add volume discounts and shipping fees based on destination, total price, and various shipment options. But we still need to be able to ensure that the most basic flow still works as it did before! We will also have to be able to check the correctness of a feature after changing our code structure.

Different Types of Tests

To ensure the quality of our entire system, we often need different types of tests; not every test is equal. They differ in both how we write them and what they are validating. Some tests only cover a small piece of complex business logic. On the other hand, we write tests that mimic an end user by clicking a button that results in adding a new record in our database, returning the persisted data to the browser to show it to our end user. Let's briefly dive into the most common test types that we use daily. Let's order them by starting with the most expensive type!

End-to-End Tests

And the winner is... the end-to-end (E2E) test. The most common use of E2E tests is to **mimic what our end user does** by automatically navigating to the user interface and clicking on elements that are visible. We can write code to simulate the process of filling out a form. Once submitted, we can check that the expected data is displayed on screen.

We also have the possibility to directly query the database or check whether a file has been created or downloaded. We could also try to check for outgoing messages, if they are crucial in our system. I strongly advise focusing on high-level outcomes instead of asserting internal implementation details. E2E tests are hard enough to write as is! If you need to check for a specific outgoing message, it's often easier to validate this with another type of test.

Integration Tests

Next in line are our integration tests, as the name suggests, we are testing some kind of integration. This does not mean that we are testing the integrations between different microservices! No, we are testing that the components of our platform integrate well with the required **infrastructure** and with the **framework** we are using. Writing integration tests has become easier over the past few years with the rise of tools like testcontainers, strongly reducing the overhead of managing the required infrastructure.

What I like to test in an integration test is that my setup of framework features, like security and validation, is correct. I also validate that I can communicate with the required data sources and other external systems, like a message broker, a remote file system, or even external APIs. Even though I want my test to be as realistic as possible, I never integrate with the real instance or an instance being used by a cloud environment. No, I always use an instance specifically created for my tests to avoid failures caused by external factors.

Integration tests don't require a cloud environment to run. We run our integration tests during the build pipeline of our application. The test framework will handle the application context and required infrastructure we need in our tests.

Integration tests are less expensive than E2E tests, because we only need one component of our system instead of a complete environment. This also makes them faster, but they are still slow, which makes them less useful during the actual development process. If I'm trying to find/fix a bug, I don't want to waste my time before knowing that my change fixed the issue!

Unit Tests

That's why we have unit tests. For me, unit tests are the most important type to maintain the functional quality of your codebase. Because unit tests run **extremely fast**, they're the most useful type of test to depend on while writing or changing code. Unit tests are also very easy to write, especially when comparing them to the other tests I've discussed earlier.

Unit tests don't require an application context to be able to run. They don't need to integrate with any of the infrastructure; they focus on testing that your code does what it should be doing. Because we don't have an application context, we need to instantiate the unit under test ourselves, but this isn't a drawback, as unit tests focus on a small piece of our application. We only need to instantiate those parts we need for our tests.

Unit tests are also able to run in isolation; they should never share state, giving them the ability to run in any order. If you are sharing state in your unit tests, you're simply doing it wrong! Modern testing tools like [JUnit](#) help us by allowing us to simply recreate the unit of code we're testing before each test, without having to do this manually in each test.

If our code depends on external systems or infrastructure, we should replace this dependency with a test double. These doubles allow us to control the external behavior, giving us the possibility to test various edge cases without adding complexity.

What Is the Unit in a Unit Test?

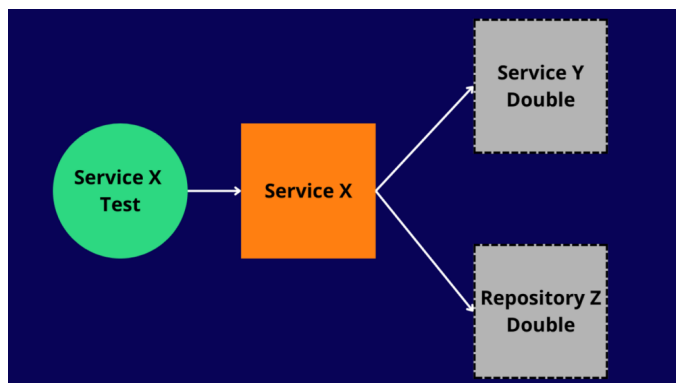
Now that we are aware of the different types of tests and their main purpose, I want to elaborate more on the unit tests. It's not because they are easy to write that we shouldn't think about how we should be writing them. And to know how to write a solid unit test, I think we need to reflect on the meaning of a unit.

A unit can mean anything. If we think beyond code, there are many meanings for the word unit. We could be talking about parts of an air-conditioning system, but also about units of measurement. If we are scrolling through a web shop, the indicated price is the price for one unit of that specific product. For unit testing, the unit can also have multiple interpretations. There's only one thing that every interpretation has in common. A unit defines **the scope of our unit tests**. Which parts of our code are being used during the test, and what parts should be replaced by test doubles?

[Martin Fowler wrote a great article about this](#), where he elaborates on two major subtypes of the unit test. I will only highlight those parts we really need to be able to understand what's up next in this article.

Solitary Unit Test

The first one is the easiest to write, but does that also mean it's the best choice? In this case, we are limiting our unit test to validate only the code of a method within a single class. If the method delegates parts of the behavior to another class, this will be replaced by a **test double**, allowing us to control what's happening. Tests like this are called **solitary tests**.



Sociable Unit Test

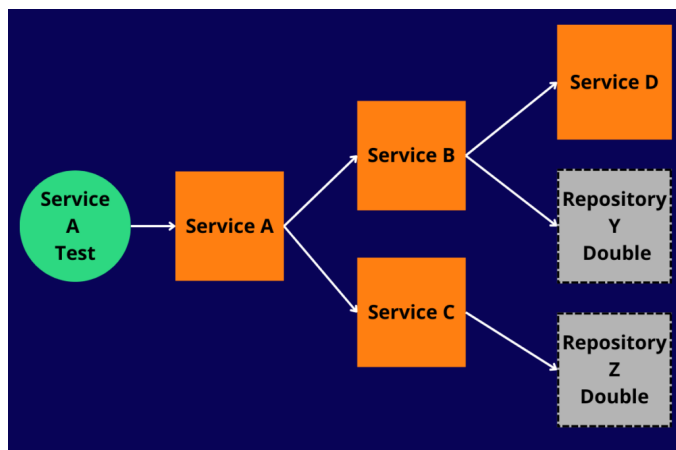
We also have a type of unit test called a sociable unit test, which uses a larger unit. We don't limit our test to the code within a single class; we focus on an isolated behavior. In a **sociable unit test**, we accept some dependencies, while we still mock others.

There are some different opinions on what you should be mocking, but I'll share how I feel you should be doing this. I believe the best way to write a sociable test is to use the real code for everything that's within the bounds of the codebase we're working on. We only mock those parts that we don't control or the dependencies that require an active application context. Some examples of these are remote APIs, communication with application infrastructure, or interactions with a filesystem. So, we will not be using a test container for our database or messaging system, because they would require an application context to read the configuration and to connect with the test instance.

There is one more thing to keep in mind when writing sociable tests. Most developers know that assumptions are the root of all evil. We should never just think something is exactly how we believe it is. With sociable tests, you are allowed to assume one thing, and one thing only! You may assume that all dependencies your test subject has are functionally correct. The simple reason to allow this assumption is that we should have had the discipline to test that new dependency by itself, with its own set of (sociable) tests.

Sociable Test or Integration Test

When I was discussing this with some of my peers, I regularly got the question of whether there is still a difference between a sociable unit test and an integration test besides the presence of the application context. In my opinion, the main purpose of any unit test is to validate the behavior of our code. Do certain inputs result in the expected outcome when we invoke a given method? On the other hand, the purpose of an integration test is to validate that all components of our application work well together. Did we make any mistakes in how we use the features of the framework, or are we injecting the right implementations? Does our code work well with the supporting infrastructure, and do our custom queries behave as expected? This is what I want to validate in an integration test.



A Typical Example

To illustrate this, imagine we are writing a REST API with Spring-Boot. We want to test a POST endpoint that has bean validation and a security constraint. If all goes well, the request should result in a new record being created after gathering some additional information and transforming the data. This record should be created in the expected database table, and the response should contain the expected JSON representation of the data we just added. In such a case, I would not only write an integration test for the happy path, but I would also verify that only users with the expected authorities can call the system. I would also test that an invalid request returns a suitable response, without adding the data to the database. This is easier to test with an integration test, because we are using framework features to add this behavior to our application.

So just to be clear, sociable unit tests are not a replacement for integration tests! Both tests serve their own purpose!

What Is a Robust Unit Test?

We talked about tests, describing how we would write them. This typically happens when we are adding new features or when we extend existing functionality with additional capabilities. If we are writing new tests, we know we are not finished until all tests succeed.

In the introduction, I mentioned that tests should not only serve as a check for the functionality to be correct during implementation. No, our tests should also guarantee that existing functionality will not break when changing the code in the future. This is what we tend to call refactoring the code.

Refactoring

[Martin Fowler once defined refactoring](#) as “a change made to the **internal structure** of software to make it easier to understand and cheaper to modify without changing its **observable behavior**”. This means that we are changing our codebase by moving code around or even by rewriting entire parts to make it more future-proof.

The most important part of refactoring does not lie in changing the code. No, it lies in the part where we do not change the observable behavior. Unchanged behavior means that when we call a method with the same parameters before and after the refactoring, the outcome of this invocation should be identical. But how can we ensure this? Simple, just run your tests!

If you only improved the code within a single method, without changing the delegations to other classes, all your tests should be fine. If not, you made a mistake! But it gets interesting if we start introducing new delegation in our code by extracting pieces of a method to a separate class. If you click run, most of the time your tests will fail! How they'll fail and what you must do to fix them will depend on the type of test you're using.

Solitary Unit Test Behavior

Let's first see how a solitary unit test works. I must begin by saying that we have two possibilities here. If we use the power of *JUnit extensions* combined with [Mockito](#), we could simply set up our test by using annotations like `@Mock_ combined with @InjectMocks_`. The `InjectMocks` annotation defines which class will be instantiated. While instantiating, Mockito will look for mocks or spies you defined to inject them. If the test subject has dependencies that you did not define, by default, *null* will be used.

This is exactly what will happen if you refactor your code without updating the tests. This will result in tests failing with the almighty *NullPointerException*. Add the required mocks to get rid of this awful failure! If you choose to instantiate the test subject yourself in a setup method, your test will not even start. It will not compile, but that's ok. Just inject the required mock to get back on track.

Now the setup of your unit is just fine again, but will your tests pass again? If you try to run your tests, you'll see that your tests are still failing. This is because we didn't define how the new mock(s) should behave yet. To make your tests work again, you'll have to define what the expected outcome is for a given method invocation on one of your test doubles.

Coupling to the Implementation

The fact that we need to stub some method invocations points out a major drawback of solitary tests. Solitary tests tend to be very tightly coupled to the actual **technical implementation** of your class. We can see this because we must define the stubs. We can cause test failures by simply moving code from our test subject to a delegate, hiding this code from the solitary test.

Sometimes we can even test for the wrong things. Let's say we want to test an exception flow, where initially our delegate throws a *RuntimeException*. So, in our test, we define that our test double must throw an exception if the method is invoked with certain parameters. After a while, we decide that the *RuntimeException* isn't good enough. We replace it with a more declarative exception that we added to our codebase. But we forget to update our tests, so our test will still test for the *RuntimeException*. We believe our work is done because we don't have any test failures, but once the application is deployed on production, Murphy's law will haunt us! Our code will crash because we didn't catch the new exception. Angry management and hotfixes incoming!

When you write sociable tests, your test will also fail to compile if you manage your test subject manually. You will have to **update the constructor invocation**, adding all new dependencies. If you did your job right, all you have to do now is run your tests, and everything should be ok. If there are some failures, you did something wrong; you changed the observable behavior of your code!

```

1  @ExtendWith(MockitoExtension.class)
2  class GetCheckupPriceSociableTest {
3      // constants declaration omitted
4      @Mock
5      OwnerRepository ownerRepository;
6      Clock clock;
7      VeterinaryCheckUpConfigProperties configProperties;
8      VeterinaryService service;
9      @BeforeEach
10     void setUp() {
11         clock = Clock.fixed(FIXED_TEST_INSTANT, ZoneId.systemDefault());
12         configProperties = TestConfigProperties.testDefaults();
13         var calculator = new CheckupPriceCalculator(
14             configProperties,
15             new DiscountCalculator(configProperties),
16             new FineCalculatorStrategy(
17                 clock,
18                 new FixedFineCalculator(configProperties),
19                 new PercentageCalculator(configProperties)
20             )
21         );
22         service = new VeterinaryService(
23             ownerRepository,
24             null, null, null,
25             clock,
26             calculator
27         );
28     }
29     // tests omitted
30 }
```

Going even Further

Once you transition from solitary to sociable unit tests, your tests are already a lot more reliable. I can live with it that I have to update my constructor invocations, but I believe that we can make it even easier. The first time I got this idea, I was looking at how dependency injection frameworks handle this. That's where I got the idea for a little experiment. I started thinking about a very lightweight dependency injection tool that creates only the instances that I need within the scope of my test. My inspiration came from how we do this using frameworks and tools like *Spring*, *Mockito*, and even how we test when using the *Angular TestBed*.

UnitSocializer

I started writing my own custom *JUnit* extension that works with some simple annotations, instantiating the classes required to be able to test a given behavior. This is when [UnitSocializer](#) was born. *UnitSocializer* is evolving, but the first version, which supports *JUnit* combined with *Mockito*, is already available. Currently, I'm working on a framework-agnostic implementation, so *UnitSocializer* – unlike *Spring* – is not yet opinionated. There is no default configuration to decide if a given type should be mocked. I will add this later, when I release a specific version for *Spring* and other popular frameworks.



UnitSocializer

The first results are above expectations! Tests run extremely fast! And we can refactor without causing test failures in most cases. Sometimes we'll have to add some additional configuration, especially when we've added new abstractions during our refactoring. This way, we can tell *UnitSocializer* which implementation should be used for a specific test.

Using *UnitSocializer* unlocks the full potential of sociable unit tests, while it feels very familiar because we can set up our test unit by adding some annotations. This triggers the internal magic, and your tests just run! *UnitSocializer* is open source[1], so you can freely experiment, suggest improvements, or new features.

```

1  @SociableTest
2  class GetCheckupPriceUnitSocializerTest {
3      // constants declaration omitted
4      @TestSubject(typeResolvers = {
5          @Resolve(
6              forClass = OwnerMapper.class,
7              use = OwnerMapperImpl.class
8          ),
9          @Resolve(
10             forClass = PetMapper.class,
11             use = PetMapperImpl.class
12         ),
13     })
14     private VeterinaryService service;
15
16     @InjectTestInstance
17     OwnerRepository ownerRepository;
18
19     @Predefined
20     Clock clock = Clock.fixed(
21         FIXED_TIME_INSTANT,
22         ZoneId.systemDefault()
23     );
24     @Predefined
25     VeterinaryConfigProperties config = TestVeterinaryConfig.testDefaults();
26     // tests omitted
27 }

```

Wrap It Up!

Let's conclude by going over the key takeaways once more. The entire article elaborated on our tests, focusing on unit tests. The main goal of our tests is to validate that the code we've written works exactly the way we expect it to. The behavior should not only be checked when writing the code, but we should also check that we did not break any existing functionality.

To ensure this, our tests need to be able to deal with changes. Writing our tests in such a way that they focus on the desired behavior instead of testing the technical implementation can help us achieve this. We need to upgrade our solitary unit tests to the more robust sociable unit tests.

We can easily achieve this by widening the scope of our test unit. Do not focus on a single class in your test; focus on behavior. Accept internal dependencies that fulfil a part of the behavior under test. This will reduce the number of mocks in our tests.

We can write sociable tests without additional dependencies by manually creating our test unit, setting everything up ourselves. In small projects, this is simple, but it can become a hassle when adding more complexity to our project. This is why I started the *UnitSocializer* project, which can help you simplify the setup of your sociable tests! [2]

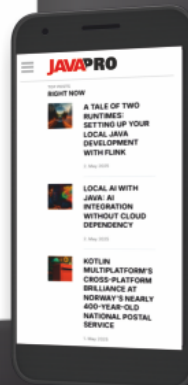
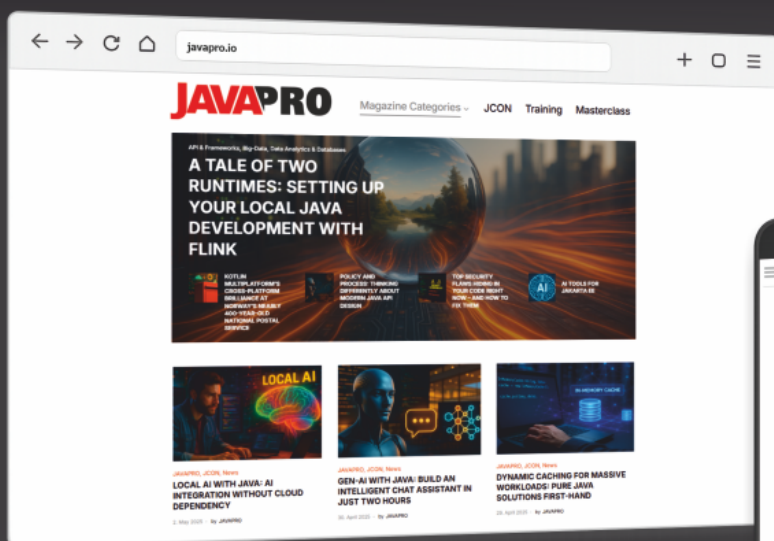
Once you master the art of sociable testing, you'll encounter robust unit tests that still validate the behavior of your code, even when moving things around. This allows us to refactor our code with confidence! You'll boost your productivity and the confidence in your codebase! Join me in the sociable test revolution!

1. UnitSocializer can be found on [GitHub](#).
2. See the [demo project](#) for more test examples.

› [Back to the Table of Contents](#) ‹

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



Anchored in the Chaos: Managing Cognitive Load, AI Uncertainty, and Well-Being as a Developer

DeveloperExperience

MentalHealth

DeepWork

Author: Simon Verhoeven

Simon Verhoeven is a Java software engineer that's interested in sharing experience, and always up for an interesting conversation.



LinkedIn

The Growing Pressure on Developers

If you feel like you're perpetually sprinting just to stay in the same place, you are not alone. Across the industry, from junior developers pushing their first commits to seasoned seniors leading large teams, the professional landscape feels less like a steady ascent and more like a chaotic treadmill set to maximum speed.

AI has certainly contributed to this whiplash-inducing landscape: one day we hear proclamations about the end of our profession, the next we're expected to deliver 10x improvements. For leaders, like Chief Information Security Officers, this is compounded by the tangible challenges of AI's (mis)use.

The ever-evolving nature of our position cannot be denied either. Today's full-stack developer isn't just expected to handle the front-end and back-end. Nowadays, we are often also juggling infrastructure-as-code, complex testing methodologies, cloud deployment, and continuous security monitoring. This widening scope means that being a software engineer today involves much more than just writing clean code.

This has led to a rampant increase in three key areas that I'd like to tackle in this article:

- **Cognitive overload:**
The relentless pace of change, the continuous influx of tools, and the pressure of context-switching across an ever-expanding technical domain can feel utterly exhausting and chaotic.
- **AI uncertainty:**
The rise of powerful AI tools, while exciting, has created palpable anxiety about job security and the future role of human programmers. Expectations are also continually shifting, with some viewing AI as the latest silver bullet.
- **Personal Wellbeing:**
We often treat self-care as a luxury or an afterthought, yet neglecting our mental and physical health makes sustainable, fulfilling work impossible.

I have written this piece to shed light on how this constant pressure manifests, drawing on my own experiences and conversations I've had with others in our community. I aim to move beyond mere recognition and discuss practical, accessible strategies we can use to manage our mental load, reframe our relationship with AI (viewing it as another potential tool in our belt, rather than a source of dread), and firmly establish wellbeing as an essential, non-negotiable component of our professional life.

Cognitive Overload

Let us start with something we've likely all experienced, but maybe haven't quite managed to put a name to.

If you've ever ended a workday feeling utterly exhausted, not because you solved a tough challenge, but because you spent all day juggling Slack threads, tracking down documentation, and switching between three different codebases, that's **cognitive overload**. It's the feeling of your mental hard drive running critically low on space.

Now, a cognitive load itself is not bad; we just need to be able to properly manage it. It is important to differentiate between the three kinds as each can be tackled in its own manner, and not everyone is as susceptible as certain types.

- **Intrinsic load (The hard/fun challenge):**
This is the good kind of challenge. It's the effort needed to solve a genuinely tricky technical issue, designing a complex microservice or figuring out a tough algorithm. You feel focused, maybe frustrated, but ultimately, you are productive.
- **Extraneous load (The friction):**
This is the one that saps our battery. It's the wasted mental energy spent battling the environment: poorly organised code, confusing meetings, documentation that's always outdated, random build failures, or relentlessly being pinged with notifications. **This** is the load we have to fight, because it burns energy without delivering value.
- **Germane load (The learning):**
This is the effort that actually builds skill. It's the time spent integrating an architectural pattern or connecting a new framework to an old concept. This is where real growth happens.

Right now, all too frequently, the **Extraneous Load** is dominating our days. We are continuously swimming in complexity.

For me, this quickly became clear when I became a consultant. When you land on a new client site, the challenge isn't just coding (Intrinsic Load), it's having to rapidly absorb their entire world: their acronyms, their specific environment setup, their internal politics, their application stack with its legacy, good and bad. A lot will, of course, depend on the quality of the onboarding procedure and documentation, if present.

You might also experience a certain **context whiplash** when changing clients, which is pure Extraneous Load. After all, how many organizations operate in the same manner?

This makes it vital to manage this mental friction to ensure your mental well-being.

The Hurdle

The continuous influx of new tools, the unrelenting pace of information, and the pressure of constantly shifting focus between unrelated tasks leave us mentally depleted. Our brain is forced into shallow multitasking rather than deep work.

The Impact

What does this look like in practice?

- **Reduced productivity:** We feel busy, but our output is low. We spend an hour regaining context for every hour we code.
- **More mistakes:** When our working memory is full of clutter, such as **Did I check that email? What was that meeting about?** It is impossible to hold all the necessary variables in our mind. Our mental buffer overflows. Mistakes are inevitable.
- **Feeling Perpetually Behind:** At the end of a full day, we feel like we've achieved nothing, which leads to anxiety and stress.

Practical Techniques

While we can't single-handedly alter how our industry works, we can adapt our workflow to try rebalance things. We can influence our **extraneous load** to a certain extent to better achieve **deep work** (also known as **flow-state**) where time seems to fly, and we feel and are productive.

Owning Our Time

In the past, I have all too frequently had the feeling of: What did I actually achieve today? Where did the time go? There are certain things that have helped me out, such as:

- **Batching communication:** Do not live in your inbox and Slack. Designate specific, short windows, say, 10:00 AM and 2:00 PM, solely for communication. Outside of those times, everything is silenced outside certain channels to keep abreast of time-sensitive matters.
- **Use the pomodoro technique:** It's 25 minutes of intense, single-task focus, followed by a 5-minute break. It trains your brain to sustain attention and gives you built-in rest periods.
- **Timeboxing obstacles:** If you hit a tricky bug, set a timer (e.g., 45 minutes). When the timer goes off, you **must** either ask for help or document the issue and move to a different task. This prevents one small hurdle from eating your whole day and draining your mental reserves.
- **Bullet list:** Starting the day off by creating a bullet list of key goals for the day helps you structure your day and refreshes your memory.

Prioritising Deep Work and Setting Boundaries

Deep work is where we create real value, which is key for both our organization and our own mental well-being. It's where we're at our most productive.

- **Schedule non-negotiable deep work:** Look at your calendar and block out 2-3 hours every day specifically for coding, design, or architecture. Treat this block like a high-stakes client meeting; do not let it be interrupted.
- **Learn to say - No - gracefully:** While it is certainly a good thing to help out **when** you can, make sure that this does not interfere with your key tasks. Every time you agree to an optional meeting or take on a tangential task, you are saying "no" to your ability to focus on your core priorities.
- **Be selective with your learning:** Stop trying to read every tech article or follow every new framework announcement. Focus on becoming an expert in the tools you **use** right now or see yourself using soon, rather than chasing everything new. Just imagine trying to keep up with all the latest Linux variants!

Systemising Knowledge Management (The ‘Second Brain’)

Your brain is a processor, not a file cabinet. Let it process and let an external system handle the storage.

- **Build your second brain:** This could be a bullet journal, a Notion page, or even just a well-organised folder of text files. The point is to have one single, trusted place where you capture all meeting notes, to-dos, code snippets, and random ideas. This is helpful for both you and the organization. Too many of us have joined an organization and received an awkward chuckle when asking about where we can find the documentation.
- **Immediate offload:** The moment a distracting thought pops into your head (e.g., “I need to order more coffee”), immediately capture it in your Second Brain and then return your focus to your task. This allows your working memory to release that thought. I can highly recommend a Fisher Space Pen Bullet and a Field Note notebook.
- **Automate the mundane:** If a task is repeatable, automate it. Use linters, robust testing suites, make, and so on. The less mental energy you spend performing and verifying trivial steps, the more you have for creative problem-solving.

Navigating Uncertainty Around AI

If managing cognitive load is about fighting the present chaos, navigating AI is about managing the anxiety regarding the future.

The moment large language models (LLMs) like GPT-4 hit the mainstream, two extreme, conflicting narratives took hold in the developer community:

1. **The Doomsday Scenario:** AI will replace programmers entirely within five years. The underlying fear is that if your worth is defined solely by **writing lines of code**, you are effectively obsolete because the machine can now do it faster and doesn’t fall ill.
2. **The Silver Bullet:** AI will resolve all our obstacles instantly. We should now be 10x more productive and deliver impossible features overnight.

Both of these fuel anxiety and extraneous load. They force us into a state of comparison and panic, making us believe we must either master every prompt engineering technique immediately or face redundancy. The reality, as always, is far more nuanced and, thankfully, less terrifying.

For many of us, this anxiety and pressure to keep up isn’t just some niggling feeling, but rather a continuous drain. From my own experiences and interactions with others in the development community, it’s clear that the pressure is being manufactured and amplified by external forces. You have **IT Managers and clients** who, after reading a short article or looking at the latest Gartner chart, suddenly see AI as a magic cost-cutting bullet. AI is showcased as something that can generate applications from scratch, without ignoring the prototypical nature of these, and ignoring the domain knowledge aspects. This often forces us to integrate unstable solutions purely because of pressure from above, just because it’s the latest buzzword. Many will be reminded of the low-code/no-code wave a few years ago, and well, here we are once again.

Adding significantly to this **Extraneous Load** are the **Tech Influencers and self-proclaimed gurus** on social media. They preach that if you aren’t using specific technique X or methodology Y, you are falling behind your peers. Luckily, they are offering their \$499 ‘AI Prompt Engineering Masterclass’ at a promotional discount to help uplift others. Their agenda is often transparently self-serving—selling courses and generating clicks. The effect on us is the same: intense, constant stress that we are **failing** if we aren’t chasing the absolute bleeding edge, and prompting managers to ask: **shouldn’t you follow some of those?**

However, this discourse overlooks certain key aspects: AI is only as good as the context and verification we provide. Our enduring value lies in the hard-won skills that AI lacks, namely judgment, architectural insight, understanding ambiguous business context, and taking legal/ethical ownership. If you jot down what your average day looks like, how much do you spend on actual coding versus design, alignment with other parties, analysing what is **actually** needed?

It became clear to me and many others that the real challenge is not mastering the technology, but mastering our response to the sheer volume of external pressure. It requires a fundamental mindset shift to filter out this noise and use AI responsibly and sustainably without losing our heads.

The Core Issue

AI advancements generate palpable anxiety about relevance, job security, and an overwhelming “need to know everything” right now. This pressure often comes from leadership, who may see AI as a simple productivity multiplier without under-

standing the nuances of implementation, security, or verification.

Reframing the Story

We need to shift from a position of anxiety to one of **agency**. AI is not a singularity; it is a complex, often flawed, and fundamentally non-human tool. By viewing it as a **collaborative partner**, a super-smart, tireless, but frequently clueless **copilot** we can redefine our role from ‘code generator’ to ‘system architect and editor.’ As someone who’s a keen proponent of rubber-duck debugging, I view it as having a cyberduck next to me.

- **You are the pilot, AI is the copilot:** The AI can fly the plane on autopilot and handle the routine checks, but the human pilot is still the one setting the destination, checking the calculations, and taking control during turbulence. Your value lies in judgement, context, and verification.
- **The AI produces drafts, you produce quality:** Think of an LLM as a brilliant intern who never sleeps but sometimes hallucinates facts or introduces security flaws. It can give you a starting point in seconds, but your expertise is required to check the logic, ensure best practices, integrate it correctly, and take ownership of the final result.

Practical Strategies

By framing AI as a tool, we can focus on where our unique human skills provide the most worth, essentially protecting our professional ‘Intrinsic Load.’

Focus on Augmentation, Not Replacement

Spot where AI can take over the low-value, repetitive tasks, freeing you up for deep work.

- **High-Quality code yields better AI results:** AI tools operate much more effectively when a codebase is well-documented, follows clear conventions, and has comprehensive tests. This isn’t just a kindness to the AI, it benefits us all as well. Good documentation reduces your **Extraneous Load** and allows the AI to provide more accurate, context-aware suggestions, accelerating your productivity without introducing risk.
- **Automate the mundane:** Focus on AI tasks that help your workflow without touching code directly. For example, generating boilerplate README drafts, summarizing long code comments, or translating a snippet from one language to another to quickly grasp what’s going on. This reduces mental friction and time spent on administrative tasks.
- **Use it as a rubber duck:** Instead of relying solely on it for solutions, use it to explore constraints, test assumptions, or rapidly prototype alternative approaches. It’s an interactive, tireless sounding board. It is a convenient way to help tick off your mental checklist of edge cases.
- **Verification:** While AI can indeed generate code, there are two critical responsibilities it is relatively poor at, namely verifying that code, checking edge cases, and ensuring it adheres to organisational standards and that it actually matches what your business users have a mental picture of. These are high-value skills that AI cannot replicate.

Double Down on Your Core Human Skills

The value of a human programmer increasingly lies in the areas that AI cannot do. You’ll also notice that these happen to be the key skills that are expected once someone wants to move beyond the junior level.

- **Architecture and system design:** AI struggles with abstract, long-term system integrity and cross-domain dependencies. Our ability to design robust, scalable systems that solve **business demands** remains essential.
- **Communication and empathy:** Explaining complex technical concepts clearly to non-technical stakeholders, translating business requirements into technical specifications, extrapolating what someone actually wants and needs rather than what they state, and collaborating with a team are uniquely human skills.
- **Debugging and problem-solving:** While AI is good at suggesting solutions, the core skill of tracing a bug through a complex, unfamiliar system, understanding root cause, and applying lateral thinking remains an area it struggles with.

Adopt a Growth Mindset

View engagement with AI as essential upskilling, not a competitive threat.

- **Experiment Safely:** Dedicate a small, non-critical block of time each week to intentionally experiment with AI tools (Copilot, ChatGPT, et cetera). Treat it as a sandbox for learning their strengths and, more importantly, their weaknesses.
- **Experiment with the interface (prompting):** Learn how to interact with the tool effectively. Understanding how to structure a good prompt is rapidly becoming a core competency for maximising AI's utility.
- **Discuss failures, not just successes:** Be open within your team about instances where AI gave a misleading or flawed answer. By understanding the failure modes, you build a shared, realistic knowledge base and reduce collective anxiety.

Ultimately, the best defence against AI uncertainty is becoming the individual who is able to skillfully wield the tool, define the problem, and deliver verified, high-quality solutions.

Protecting Personal Well-Being as a Necessity

If the first challenge is about managing the input (cognitive load), and the second one is about managing the future (AI), then the third is about maintaining the key piece that processes it all: **you**.

In our industry, we often obsess over the specs of our laptops, the ergonomics of our mechanical keyboards, and the efficiency of our CI/CD pipelines. Yet we frequently ignore the critical component, the central processing unit, namely **ourselves**. We frequently run our own biological hardware on poor fuel, insufficient downtime, and ignored error logs (stress signals).

It's something I've seen happen repeatedly, and something I'm guilty of myself. I've watched brilliant colleagues who could solve complex architectural problems in their sleep suddenly hit a wall. They didn't leave the industry because they couldn't learn the latest framework or because AI took their job. They left because they eventually burned out. They ran out of capacity.

We tend to treat self-care as a luxury, something we'll get around to "after this release" or "once the crunch is over." But to quote Da Vinci: "A work of art is never finished, it is merely abandoned." There is **always** another release. Neglecting our mental and physical health makes sustainable, fulfilling work impossible.

The Situation

While those of us living in Western Europe may be better protected by labour laws and a culture that values time off than our counterparts in the US or Asia, we face a more subtle, yet equally damaging challenge: **intensity and permeability**.

It is rarely about working 16-hour days; it is about the sheer **density** of the hours we do work. We try to squeeze complex coding, architectural decisions, and back-to-back Teams meetings into a condensed window, often skipping breaks to 'power through' or to rush to the next meeting.

Furthermore, hybrid work has eroded the boundaries. The physical commute, which used to serve as a decompression buffer and clear delineator, is gone for many. The result is that while we might physically leave our desks at 5:30 PM, the work follows us via notifications on our personal phones, or the mental urge to "check one last thing" before bed.

This leads to **Health Debt**.

Akin to technical debt, health debt accumulates interest. You might get away with hacking your sleep schedule or skipping movement for a sprint or two, but eventually, the debt comes due. The cost isn't just physical illness; it's a degradation of your cognitive abilities—your focus, your patience, and your ability to solve the very challenges you're paid to solve.

Shifting the Narrative

We need to shift our mindset from "Self-care is a luxury," and "I will get around to it later" to **"Self-care is professional maintenance."**

Think of it this way: You wouldn't run a critical production server at 100% CPU utilization for months without cooling or patching, and then be surprised when it crashes. You are the most vital piece of infrastructure in your career. Maintaining your well-being is not "quiet quitting" or "being lazy" but rather a requirement for high-performance engineering.

- **You are an athlete of the mind:** Professional athletes don't train 24/7. They rest effectively so they are able to perform at their peak. Cognitive work is physically draining (the brain consumes about 20% of the body's energy). You need recovery protocols just as much as a runner does.
- **Resilience requires reserves:** When the "chaos" hits—a production outage, a challenging client, a sudden deadline—then you need mental reserves to handle it. If you are constantly running on fumes, you have no buffer for the unexpected.

Practical Strategies

How do we put this into practice? We don't need a complete lifestyle overhaul; we need patches and configuration changes.

Treat Health as a Non-Negotiable Dependency

If a library is deprecated and causing security flaws, you upgrade it. If your sleep is broken, you need to fix it with the same urgency.

- **Prioritise Sleep:** It is the defragmentation process for your brain. It is when short-term memory is converted to long-term memory and when neurotoxins are cleared. Coding on four hours of sleep is like coding drunk; you might feel productive, but the quality is usually terrible.
- **The Physical API:** Your brain lives in your body. Simple things like hydration, standing up once an hour, and getting sunlight in the morning directly impact your ability to focus on complex logic.

Define "Done" and Disconnect

One of the hardest parts of our job is that the work is never truly "finished". There is always another ticket.

- **The shutdown ritual:** Create a distinct boundary between "work mode" and "home mode." This could be closing all your tabs, writing your to-do list for tomorrow (offloading the cognitive load), and physically leaving your workspace. Personally, I am keen on a forest walk followed by a nice cup of tea.
- **The Right to Disconnect:** Be clear and firm about your availability. If you answer Slack messages at 9 PM "just this once," you teach your colleagues and clients that you are available at 9 PM. Protect your downtime; it is where your creativity regenerates.

Seek Community and Perspective

Isolation is a major contributor to burnout, especially in remote work.

- **Talk about it:** In my experience, admitting "I'm struggling with the workload right now" usually opens the doors for others to say, "Me too." Vulnerability builds trust and reduces the feeling that you are the only one not coping.
- **Hobbies outside of tech:** Have a hobby that involves zero screens. Whether it's cooking, hiking, or, as I mentioned earlier, creative writing using fountain pens and paper. Doing something analogue grounds you in the physical world and gives your visual cortex a much-needed break.

Conclusion: Building an Anchor in the Chaos

We started by looking at three challenges that we as developers encounter all too often: the increasingly challenging **cognitive load**, the uncertainty of **AI**, and our often-neglected necessity of **personal well-being**.

It's important to realise that these challenges are interconnected.

- If you don't manage your **cognitive load**, you won't have the mental space to learn how to leverage new tools such as **AI**.
- If you fear **AI**, you increase your stress, further impacting your **well-being**.
- And if you neglect your **well-being**, you lack the resilience to handle either the load or the uncertainty.

However, none of these are insurmountable. By **systemising** how we handle information, reframing AI as a **collaborative copilot** (pun intended) rather than a replacement, and treating our health as **critical infrastructure**, we will be able to stop flailing in the current and start swimming.

You have the agency to drop an anchor. It doesn't mean the storm stops, but it means you stop drifting. It will enable you to weather the storm, regardless of what the next tech cycle brings.

Helpful Resources

Navigating these challenges might seem daunting, but I've found that a shift in mental mindset is key. Beyond the earlier recommendations of conferences and meetups, certain books have been instrumental for me, and I hope they will be for you too.

It does behove me to say that this is far from an exhaustive list, as there are so many amazing resources. So please do reach out with recommendations or if you're looking for something specific.

1. **The pragmatic programmer** by David Thomas and Andrew Hunt
To me, this book was a lesson in simplicity. It helped me realise that writing adaptable, resilient code isn't just about 'best practice', it is about reducing the mental friction we face every day. Adhering to the KISS principle ('Keep It Simple, Stupid') is an act of kindness to my future self.
2. **The programmer's brain** by Felienne Hermans
Do not let the title of this book deceive you; the content is very much applicable to everyone's brain. Felienne Hermans dives into the cognitive science of how we process information and how we are able to facilitate this both for ourselves and others. It helped me understand why some code is so taxing to read or debug, and gave me practical techniques to work with my own cognitive processes (and quirks).
3. **How to make your brain your best friend** by Rachel Barr
A compassionate and insightful guide to the mind. This book reminded me that sustainable living isn't about fighting my own psychology, but about understanding and nurturing it. It's a foundational read for wellbeing. Small acts of kindness truly matter.
4. **Four thousand weeks: time management for mortals** by Oliver Burkeman
One of those reads that led to quite a paradigm shift in how I viewed productivity and the value of time. It's not one of those traditional self-help books, which just try to cram more into one day. Instead, it offers a liberating philosophy on embracing our finite time. It helped me (somewhat) step off the hamster wheel of efficiency and focus on what's genuinely meaningful.
5. **The bullet journal method** by Ryder Carroll
As developers, we spend our lives on screens, but I found that the best way to clear my head was often analogue. This method gave me a structured way to 'dump' the mental clutter onto paper. It's been quite helpful for quieting the noise and helping me focus on what actually matters, rather than what's just shouting the loudest. My love for fountain pens certainly also helps.
6. **Rewiring your mind for AI** by David A. Wood
With so many loud voices debating AI, this book helped me ground the conversation for myself. This book helped me reframe that anxiety into curiosity. It moves the conversation from fear to collaboration, showing how we should view these tools as powerful teammates rather than replacements.

Remember, investing in these areas isn't a distraction from your usual upskilling. They help make sustained, meaningful progress possible. I wish you the very best on the journey.

› [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



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!

**BEST
{BUDDYS}**
IN CODE

trusted by

ATU

EMPORIO ARMANI

CONRAD

Fraport

Red Bull

Allianz



Arrange a free discovery call here!

www.xdev.software

XDEV