

33 **JAVA RECORDS:
ETCHED IN FINALITY**104 **MICROSERVICES:
LESSONS FROM THE TRENCHES**

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

Special Edition 5 / 2025

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

#JAVAPRO

Java 25 Special Edition

PART 2

10 **CRAFTING NIMBLE JAVA: STRATEGIES
FOR OPTIMAL PERFORMANCE & AGILITY**20 **WRITING READABLE CODE
WITH ALGEBRAIC DATA TYPES & PATTERN MATCHING**26 **JAVA STREAMS EVOLUTION:
FROM JAVA 8 TO TODAY**40 **JAVA 25: GETTING THE MOST OUT OF VIRTUAL THREADS
WITH STRUCTURED TASK SCOPES & SCOPED VALUES**116 **HOW QUARKUS SEAMLESSLY COMBINES
ARCHITECTURE, PERFORMANCE & CLOUD-NATIVE**124 **IMPROVING PLATFORM OBSERVABILITY
WITH DISTRIBUTED TRACING & OPENTELEMETRY**148 **THE OPEN SOURCE, DETERMINISTIC ENGINE
MAINTAINING JAVA'S NEXT 30 YEARS**158 **BRIDGING JAVA & PYTHON FOR AI/ML IN PRODUCTION:
THE CASE FOR GRAALPY ON GRAALVM****JCON**
EUROPE
COLOGNEApril 20 - 23, 2026
www.jcon.one

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

From Virtual Threads to Vector Databases: Why Java 25 is the Foundational LTS for the AI Enterprise

Welcome to part 2 of a very special edition of JAVAPRO, dedicated entirely to the latest Long-Term Support (LTS) release: Java 25. Every LTS release serves as a stable bedrock for the enterprise world for years to come, and this one, arriving precisely on schedule, certainly is no exception. The six-month release cadence continues to deliver, consistently refining and graduating features from the ambitious projects that are reshaping the platform. Java 25 isn't defined by a single seismic feature, but rather by a cohesive, powerful push towards three core pillars: developer productivity, performance, and modern concurrency.

The Value on the Table: Java 25's Core Impact

The features finalised or significantly advanced in Java 25 create genuine, tangible value for every Java developer. The advancements around Structured Concurrency (now in its fifth preview) and the finalisation of Scoped Values (JEP 506) fundamentally change how we write concurrent, high-throughput code. Paired with Virtual Threads (which arrived in Java 21), these tools make handling thread-local data safer, simpler, and far more efficient, directly impacting the scalability and reliability of microservices and complex server-side applications. Furthermore, new language features like Module Import Declarations (JEP 511) and Compact Source Files and Instance Main Methods (JEP 512) directly address decades-old pain points. The former cleans up cluttered import lists for modular applications, while the latter is a huge win for beginners, scripts, and educational content, simplifying the "Hello World" barrier to entry. This LTS release also delivers significant runtime improvements through deep changes under the hood. The promotion of Compact Object Headers (JEP 519) to a product feature, coupled with the introduction of the Generational Shenandoah Garbage Collector (JEP 521), means applications will consume less memory and experience better-managed, more predictable latencies. These are high-impact, low-effort gains that benefit nearly every deployed Java application.

A Look Ahead to Java 26

Following the solid foundation of this LTS, the next non-LTS release, Java 26 (due March 2026), promises further evolution of the platform. We anticipate integrating the new HTTP/3 for the HTTP Client API (JEP 517), which is crucial for modern networking that demands low-latency, high-performance web communication via the QUIC protocol. We expect more runtime optimisation with a push to improve G1 GC throughput by reducing synchronisation (JEP 522), ensuring Java's default garbage collector keeps pace with the demands of highly concurrent systems. Finally, we will likely see the finalisation of current preview features like Structured Concurrency and further evolution of pattern matching with Primitive Types in patterns (JEP 507), proving that the six-month cycle is effectively turning experimentation into standard practice.

Generative AI: The New Frontier for Java Developers

Separate from the core JDK, one area demands the immediate attention of every Java developer: Generative AI (GenAI). While Python still dominates model training, Java's primary role in GenAI is as the enterprise orchestrator, excelling at

production-grade, scalable application development. This dynamic has led to the rapid maturation of libraries that bridge the gap, with Spring AI and LangChain4j emerging as the de facto standards for building GenAI-powered applications, enabling developers to integrate various Large Language Models (LLMs) and vector databases without needing deep Python or data science expertise. The current focus is heavily on patterns like Retrieval-Augmented Generation (RAG), where Java is used to connect LLMs to trusted, internal business data - the real value driver in the enterprise. The next 12 months are critical, and Java developers must immediately prioritise a few key areas. They should learn to communicate effectively with LLMs through prompt engineering, a fundamental development skill. Developers need to master implementing RAG architecture patterns using libraries such as Spring AI to build secure, context-aware applications. A solid understanding of vector databases for semantic search and efficient data retrieval in AI workflows is also essential. Finally, embracing AI-assisted coding tools like GitHub Copilot and Amazon CodeWhisperer is no longer optional, as these are becoming indispensable for boosting personal and team productivity.

No, I don't believe AI will replace developers in the foreseeable future. But AI is radically changing the way developers work right now. And GenAI is now beginning to change enterprise software as well. And those who don't adapt quickly will soon be unable to keep pace in terms of productivity and will very likely be replaced sooner or later. That's why it's never been more critical than today not to get left behind in AI. With our new JCON GenAI workshop series, which we prefer to organise in collaboration with Java User Groups, we want to inspire Java developers to get started with development using AI tools and to develop GenAI use cases. We've secured leading innovators in the Java space as speakers for this. Please take advantage of this opportunity, come to our JCON GenAI workshops and attend the meetups of your local JUGs to exchange ideas and share experiences.

The Verdict on Java 25

Is Java 25 the big deal that dramatically overturns the table? No, it's not an earth-shattering, revolutionary release like the launch of the Module System (Java 9) or Virtual Threads (Java 21). However, it is a solid evolution and an exceptionally well-rounded LTS. Java 25 takes the revolutionary groundwork laid by Project Loom (Virtual Threads) and makes it safer and more practical with Scoped Values and Structured Concurrency. Critically, it simultaneously delivers essential performance improvements and performance gains for the JVM through compact object headers and GC enhancements. Java 25 is less about dazzling new concepts and more about maturity, stability, and pragmatic enterprise value. It's the LTS that shapes the future of Java into a reliable reality today, delivering precisely what the community and the enterprise truly need.

In this issue, you'll learn more about the latest Java features and news about GenAI with Java. My special thanks go to all the fantastic authors of this issue, as well as to all the authors who constantly write on our online platform, without whom JAVAPRO would not be possible. Please recommend us to your colleagues and friends. JAVAPRO is a completely free Java magazine for the international Java community. I hope you enjoy reading this issue.

Author: Markus Kett

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



LinkedIn

Contents

Core Java		<i>Gautham Krishnan</i>
Crafting Nimble Java: Strategies for Optimal Performance and Agility		10
Where Even Your Bugs Have Their Own Little Cubicles: Modular Architecture		11
Because Life's Too Short to Waste on Booting Up: InstantOn		11
Zip and Zing with Shared Class Cache and Dynamic Ahead-of-Time Compilation		11
Microservices Are Even Applied at the JVM Level!!		12
Trying to Fit an Elephant into a Tupperware: Container Conundrum		12
The Unsung Hero of Memory Optimization: Compressed Object References		12
Wrapping It Up: Fast Startup and Small Footprint		12
Core Java		<i>Ties van de Ven</i>
State Management Explained		14
Why State Management Matters		15
Complexity		15
Cognitive Load		15
Tradeoffs		16
Using the Type System		18
Conclusion		18
Core Java		<i>Balkrishna Rawool</i>
Writing Readable Code with Algebraic Data Types and Pattern Matching in Java		20
Algebraic Data Types		21
Pattern Matching		21
Using Them All Together		23
Data Oriented Programming		24
References		25
Core Java		<i>Mihaela Gheorghe-Roman</i>
Java Streams Evolution: From Java 8 to Today		26
Introduction to Java Streams		27
Motivation Behind Streams		27
Stream Pipeline Architecture		27
Intermediate vs. Terminal Operations		28
Evolution from JDK 8 to TODAY		28
Best Practices and Common Pitfalls		31
Summary		31
Core Java		<i>Manoj Nalledathu Palat</i>
Java Records — Etched in Finality		33
The Response		36
The Feature — Records		36
“record” — Restricted Identifier		36
The Parent — java.lang.Record		36
Components		37
Canonical Constructor		37
Compact Constructor		37
What about “normal” Constructors?		37
Record — Expanding the Right Way		37
Errors — A Part of Programs		37

A Disassembled View	38
The Beginning.. . . .	39

Core Java Christian Woerz

Java 25: Getting the Most Out of Virtual Threads with Structured Task Scopes and Scoped Values	40
Structured Task Scopes — the Missing Piece	41
Virtual Threads in Action	41
Concurrent Calls with StructuredTaskScope	42
Joiners — Controlling How Tasks Complete	42
Writing a Custom Joiner	43
Combining Concurrent Scopes	44
Why Structured Task Scopes Matter	44
Scoped Values – Replacing ThreadLocal	44
From ThreadLocal to ScopedValue	44
Why ScopedValue is Better	45
Final Thoughts	45

API and Frameworks Julius Lauterbach

The Blockchain in Your Java Application	47
Short Foreword	48
Blockchain	48
Smart Contracts	49
Web3j Library	49
Integration in Java with Maven	50
Example from Practice	51
Conclusion	51

API and Frameworks Franck Pachot

No Deadlocks in MongoDB: Atomic Documents and Retries with Spring Data MongoDB	53
Avoiding Deadlocks in MongoDB	54
Handling Write Conflicts in MongoDB with Spring Data	54
Troubleshooting	59
In Summary	61

API and Frameworks Luis Majano

BoxLang: A Dynamic Renaissance – From Zero to Release in 21 Months	62
An 18-Year Drought	63
Part I: The Foundation	63
Part II: Project Jericho	64
Part III: Architecture	64
Part IV: The Journey	66
Part V: What Was Built	70
Part VI: The Formula for Success	71
Part VII: What's Next	71
The Complete Package	72
Epilogue: The Shofar Sounds	72

API and Frameworks Julius Mischok

High Speed JPA	74
The JPA Paradox	75
A Brief Look Back	75

The N+1 Problem: Why 1+N Would Be a Better Name	75
Lazy vs. Eager Loading: The Subtle Performance Trap	77
Shorts: Other Pitfalls and Solutions	78
Memory Pitfalls	79
Beyond JPA: The Holistic View	79
High Speed JPA is Possible	79

API and Frameworks

Werner Keil

Jakarta Data and NoSQL – Standardized Data Access for Jakarta EE	81
What are NoSQL Databases?	82
Features of NoSQL Databases	82
Types of NoSQL Databases	82
Key-Value Stores	82
Column-oriented	82
Document-oriented NoSQL	83
Document Stores	83
Graph Databases	83
Time Series	83
Multi-Model	83
SQL vs. NoSQL	83
Jakarta Data	85
Repository	86
Ordering and Sorting	87
Pagination	87
Event Handling	87

AI and ML

Rory Preddy

Sarcasm-as-a-Service: Five Years Later	91
Why Does Sarcasm Matter?	93
Opportunities and Risks	94
Revisiting the Original Demo	95
Prosody and Brain Chemistry	96
Recent Advances and the New Sarcastic Soundboard	96
Conclusion	96
Appendix A – Demo Deep dive	97

Architecture and Microservices

Rafał Maciak

Microservices: Lessons from the Trenches	104
Improper Service Boundary Designs	105
Avoid Overuse of Synchronous Communication	107
Missing or Inappropriate Tools	110
Organizational and Process Changes Are Also Crucial	110
Takeaways	114

Architecture and Microservices

Holger Tiemeyer

How Quarkus Seamlessly Combines Architecture, Performance, and Cloud-Native	116
From Weakness to Strength: How Quarkus Is Redefining Java	118
A Look at the Basic Principles and Structure of Quarkus	119
Reactive Development with Quarkus in Dev Mode	121
The Strategic Advantages of Quarkus	123
Literature & Links	123

Improving Platform Observability with Distributed Tracing and OpenTelemetry	124
Distributed Traces: An Essential Pillar	125
The Architecture	126
Implementation and Data Broadcast	126
Aggregation	127
Storage	129
Display	129
Compatibility	129
Performance	130
Conclusion	130

Debugging In The Dark	131
Data Is Too Less	132
Data Is Too Much	133
Data Is Contradictory	134
Problem Not Reproducible	135
The Problem Is in Our Mind!	136

The Art of Static Code Analysis	138
The Necessity for Static Analysis of Source Code	139
Parser Generators to the Rescue	139
Java Libraries to the Rescue	141

Beyond Version Control: How Git Can Power Smarter Technical Decisions	143
The Untapped Power of Git History	144
What Parts of the Codebase Are the Most Fragile or Risky?	144
Who Are the Key Contributors to Critical Modules?	145
Limitations and Cautions	146
Conclusion: Your Git Repository is a Goldmine	147

The Open Source, Deterministic Engine Maintaining Java's Next 30 Years	148
Why AI Assistants Fall Short	149
The Lossless Semantic Tree (Deep Dive)	150
Recipes: Deterministic Automation for Java	153
AI + Recipes: A Hybrid Model that Scales	154
How AI Is Accelerating the Growth of OpenRewrite	155
Use Cases in Automated Java Modernization	156
Continuously Maintaining Java's Next 30 Years	157

Bridging Java and Python for AI/ML in Production: The Case for GraalPy on GraalVM	158
Introduction	159
The AI ERA	159
Why Combine Java and Python?	160
Overview of the PoC Project	160

Developer Setup and Project Structure	160
Core Code Snippets	161
Running the App & API Demo	163
Proof Is in the Performance (or Was It Pudding)	163
Lessons Learned & Challenges	166
Production Considerations	166
Conclusion	166

Project Management

Erik Pronk

The Michael Jordan Approach to Software Engineering	167
Focus on Fundamentals	168
Work Ethic	168
Embracing Short-Term Pressure	168
Continuous Self-Improvement	169
Teamwork and Leadership	169
Staying Healthy and Looking After Yourself	169
To Sum It All Up	170

Projectmanagement

Ina Bumstein

Conferences. How and Why	171
First Step: Choosing	172
Second Step: Before the Event	172
Step Three: During the Event	172
Step Four: After the Event	173
Final Thoughts: The Why	173

Web-Development

Lucien Immink

Design-first APIs for UI Components	174
Communication	175
APIs	175
UI Components	175
Design first	175
Framework agnostic	176
Web-Components API	176
Custom Elements Manifest (CEM)	176
Skeleton components	176
Create the skeleton component	177
API Viewer/Explorer	178
AAAG	179
AAAG Examples	180
Getting Started	180
Wrapping Up	180

Security

Mihaela Gheorghe-Roman

Security in the Age of Java 25: New Language Tools for Safer Code	181
Introduction: Why Java security features Matters More Than Ever	182
Java Security Features Beyond Cryptography	182
Sealed Classes: Preventing Inconsistent State	182
Pattern Matching: Safer Type Inspections	183
Records: Immutable by Default	184
Case Study 1: Preventing SQL Injection with Safer Modeling	185

Case Study 2: Java Security Features – Microservices Boundaries	185
Case Study 3: Enforcing Business Rules at Compile Time	185
Best Practices for Java Security Features	186
Conclusions	186
REFERENCES	186

JVM-Languages

Peter Verhás

Why I Abandoned my 30-year Open Source Project	187
Introduction	188
Program Enhanced Document a.k.a. PET	188
What is PET	188
Jamal, Perl version	189
Pyama	189
Jamal in Java	190
Jamal Problems	190
PET Problems	190
PET and LLM	191
Recommended Approach	191
Summary	192

JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
Germany

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

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

Editorial:
info@javapro.io

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

Copyright (c) 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.



Crafting Nimble Java: Strategies for Optimal Performance and Agility

Cloud Native

JVM

Modularity

Author: Gautham Krishnan

Gautham Krishnan is a dedicated software professional boasting over 8 years of industry experience. Currently employed as a Software Developer at IBM Software Labs, he focuses on traditional WebSphere application server development. Additionally, Gautham actively contributes to the enhancement and maintenance of Open Liberty server, an esteemed open-source initiative at IBM.



LinkedIn

“Who says you cannot be Big and Nimble?”

Numerous online enterprises rely on application servers, with platforms like [Liberty](#) and [JBoss](#) being prominent examples, all leveraging Java technology. Java presents users with many advantages, including a comprehensive class library, impressive throughput performance, robust debugging and tooling features, and additional benefits.

A typical JVM execution begins with class loading and initialization, followed by initial interpretation of methods until they are compiled by the Just-In-Time (JIT) compiler into machine code. The outcome? Lengthier start-up times and an initial phase of increased memory usage as the system ramps up. In this blog post, we will discuss some of the key strategies to make a nimble Java.

Where Even Your Bugs Have Their Own Little Cubicles: Modular Architecture

By breaking down applications into smaller, reusable modules, developers can achieve greater flexibility, scalability, and maintainability. Embrace Java’s modules and `jlink` to help with modularity, to enforce encapsulation, and to create custom runtime images.

Modules allow you to organise your code into discrete units of functionality. Each module specifies its dependencies on other modules and what packages it makes available to other modules.

`Jlink` takes a set of modules (along with their dependencies) as input and generates a runtime image that contains only the modules necessary to run your application. This helps to reduce the size of your application distribution and improves startup time, as you’re only bundling the required modules.

```
1 jlink --module-path=$JAVA_HOME/jmods:mllib \ --add-modules com.demo.main \ --output  
testRuntimeImage \ --strip-debug \ --no-header-files
```

In the example above, `jlink` will create a custom runtime image with the modules provided, excluding unnecessary debug information and header files from the JDK modules.

For a fully working setup: [My sample code on modules and `jlink`](#).

Because Life’s Too Short to Waste on Booting Up: InstantOn

In a world where streaming services are omnipresent, we have the freedom to pause a video at any moment and can resume from the same position where we left off, even if we switch to a different device.

Consider the prospect of applying a similar concept to our Java applications!

Thanks to [IBM Semeru Runtime](#)’s InstantON feature. IBM Semeru Runtime’s InstantON feature is built on OpenJ9 CRIU Support technology. It makes use of a Linux feature, CRIU (checkpoint — restore in user space).

CRIU support is not enabled by default. You must enable it by specifying the `-XX:+EnableCRIUSupport` command-line option when you start your application.

For a fully working setup: [Getting started with openj9 CRIU support](#)

Zip and Zing with Shared Class Cache and Dynamic Ahead-of-Time Compilation

Shared classes in a cache offer a straightforward and effective method to enhance startup performance. During the initial run of an application, the VM is tasked with loading all necessary classes, a process that consumes time. However, by caching these classes, subsequent runs experience significantly reduced initialization times.

The Shared Class Cache (SCC) technology involves a shared memory area designed to cache entities like `ROMClass` (a read-only representation of a Java class), AOT compiled code, interpreter profiling information, and more. Implementing a shared classes cache not only streamlines startup performance but also minimizes memory usage by enabling multiple VMs to share class data. Furthermore, debugging-related class data remains on disk rather than in memory, contributing to a smaller memory footprint.

Another key technology that improves start-up time is dynamic AOT. It is called dynamic because methods are compiled to a relocatable format at runtime and stored in the shared class cache. Any subsequent JVM instance can use these methods

after a cheap relocation process, which is about 100 times faster than a JIT compilation. With AOT, the JVM can transition faster from the interpreter to machine code, which improves start-up time as a result.

In [Semeru-Xshareclasses](#) command-line option is highly configurable, allowing you to specify where to create the cache, how much space to allocate for AOT code and much more.

Microservices Are Even Applied at the JVM Level!!

[Semeru](#) Cloud Compiler (aka JIT server) technology decouples the JIT compiler from the VM and lets the JIT compiler run remotely in its own process. This mechanism prevents your Java application from suffering possible negative effects due to CPU and memory consumption caused by JIT compilation. With the Semeru Cloud Compiler, the application is not affected by the stability of the compilation; if the Semeru Cloud Compiler goes down, the application continues to run, because the JVM retains the ability to compile locally using its embedded JIT compiler.

Ok, now you will be thinking why dynamic AOT if we have Semeru Cloud Compiler to the rescue. The code quality of the AOT code is about 10% less (in terms of throughput) than the code quality of JIT compiled code, which is tailor-fitted for a particular JVM. This 10% throughput gap is typically addressed by recompiling hot AOT-compiled methods with the regular JIT compiler. Moreover, some methods cannot be subjected to AOT compilations, and they need to be compiled with the regular JIT compiler.

Thus, the AOT and Semeru Cloud Compiler technologies complement each other, rather than competing with one another: The AOT solution can be used to quickly bring the throughput to reasonable levels, while the Semeru Cloud Compiler can be used to reach peak throughput by performing recompilations and compilations not handled by AOT.

Trying to Fit an Elephant into a Tupperware: Container Conundrum

Containers, paired with orchestration tools like Kubernetes, are currently the talk of the town. When it comes to running Java applications within containers, both Alpine Linux and UBI Image offer compelling advantages. Here's a comparison to help you decide which option best suits your needs:

- Alpine Linux: Ideal for resource-constrained environments, lightweight deployments, and security-conscious applications. Well-suited for microservices architectures, DevOps workflows, and cloud-native deployments. Base image can be as small as 3 MB on disk.
- UBI Image: Suitable for enterprise-grade deployments, mission-critical applications, and environments requiring extensive support and certification. Recommended for organizations with existing investments in the Red Hat ecosystem and a preference for standardized, certified container images. At 72 MB, this is one of the larger base images available.

The Unsung Hero of Memory Optimization: Compressed Object References

On 64-bit systems, the VM can use compressed references to decrease the size of Java objects and make better use of the available space in the Java heap. By storing objects in a 32-bit representation, the object size is identical to that in a 32-bit VM, which creates a smaller memory footprint. These 4-byte (32-bit) compressed references are converted to 64-bit values at runtime with minimal overhead. Smaller objects enable larger heap sizes that result in less frequent garbage collection and improve memory cache utilization. Overall, the performance of 64-bit applications that store compressed rather than uncompressed 64-bit object references is significantly improved.

In [OpenJ9](#), compressed references are used by default when the maximum Java heap size is in the range 0–57 GB. To turn off compressed references, use the `-Xnocompressedrefs` command-line option.

Wrapping It Up: Fast Startup and Small Footprint

For latency-sensitive applications, fast start-up is a requirement if one is to use a dynamic scaling approach to provision resources. With [OpenJ9](#) features like InstantOn, Semeru Cloud Compiler, and metadata caching techniques such as the Shared Classes Cache (SCC), alongside dynamic ahead-of-time (AOT) compilation, adopting a modular runtime architecture, and selecting appropriate container images significantly reduces start-up times and will minimize memory footprints.

Write once, debug anywhere!

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

JCON2026
www.jcon.one

JAVAPRO

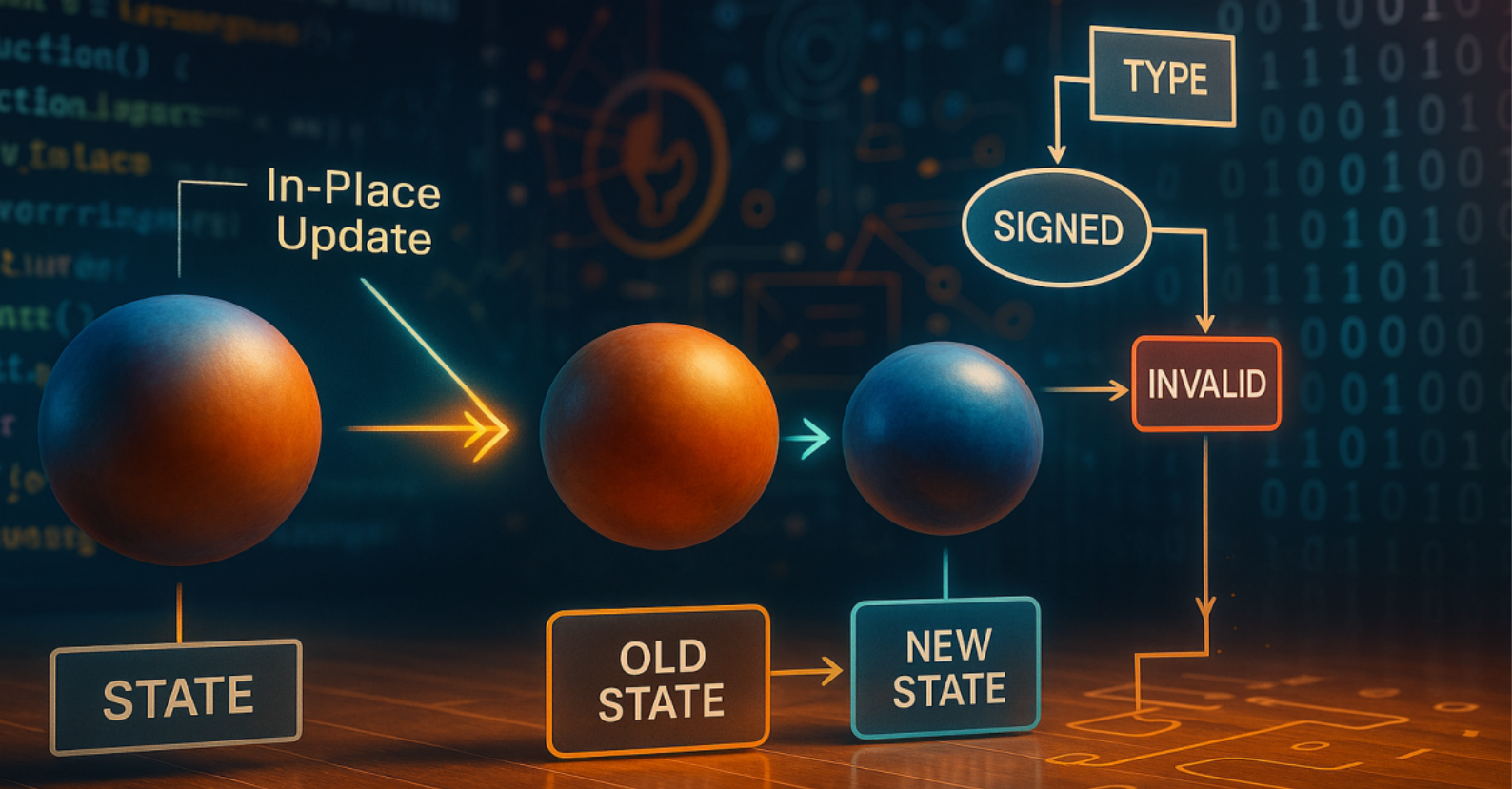
APR
20-23



📍 **CINEDOM COLOGNE**

JCON EUROPE 2026

International Java Community Conference



State Management Explained

#API

#JavaRecords

#TypeSafety

Author: Ties van de Ven

Ties van de Ven is a software engineer with a passion for concepts, software engineering fundamentals and helping others. He combines these passions by doing public speaking, volunteer work for organisations like Devoxx4kids and coding-coach and working as a Software Quality Expert at Alliander.



LinkedIn

State causes a lot of complexity, is the main source of (weird) bugs and introduces high cognitive load to software engineers so it is important to handle it with care. In this article we go into the problems of (mutable) state, how and when to use mutable and immutable state, and how we can use our type system to prevent us from making mistakes.

Why State Management Matters

Most bugs out there are in one way related to state. Some examples are things like race conditions, null pointer exceptions and inconsistent behavior. And let's not forget the famous quote "Have you tried turning it off and on again" that seems to fix most problems. This tends to work because resetting the device or software causes it to start with a new fresh state. Mutable state can also cause headaches when doing concurrent programming, but because most developers do not write multi-threaded software on a day to day basis, let's focus for now on problems that are always present: complexity and high cognitive load.

Complexity

Imagine we have a red car that we would like to turn into a blue car. In the real world we would go to a garage to give it a paint job. Typically Object Oriented Programming mimics this by creating a mutable version of a car with a getter and setter. We can then call the setter to make our red car blue.

```
1 public class Car {
2     private String colour;
3     public String getColour() {
4         return colour;
5     }
6     public void setColour(String colour) {
7         this.colour = colour;
8     }
9 }
```

A more functional programming approach would be to omit the setter, thus creating an immutable version of the same car. The real world equivalent of using this would be to create a second identical car next to the old one but use blue paint instead of red paint, and then destroy the red car. While in the real world this would not be a financially solid decision... in software engineering this does have an advantage. Imagine if I would ask our Object Oriented Programming car what color it is, what would it return? The answer is that you cannot be sure. It depends on when you ask the question, was it before the paint job, or after. Whilst if you ask any of the two functional programming cars what color it is, you will always get back the same answer. And this is why mutable state is always by definition more complex than immutable state. It adds the complexity of time.

value = (state, time)

Cognitive Load

Another example of how mutable state adds to code can be seen when we look at local reasoning; the ability to tell what code does in isolation without having to see the rest of the software. If we have the following function that receives a **mutable** car, will the two print statements print out the same value?

```
1 public void foo(Car redCar) {
2     System.out.println(redCar);
3     paint(redCar);
4     System.out.println(redCar);
5 }
```

The answer is, we don't know. It depends if the paint method paints the car, or makes a painting of a car. The only way to know for sure is to go into the paint method (and down a whole function chain if you are unlucky) and see what happens. This means we cannot apply local reasoning to this bit of code. We need more information from outside of this functions scope. But what if we would create an immutable version of the car, we can do this by using records

```
public record ImmutableCar(String color, String brand){}
```

Record classes provide getters, equals and hashCode out of the box. Because there are no setters it cannot be changed after creation. If we would pass this car into the same function, will the two print statements print the same thing?

```
1 public void foo(ImmutableCar redCar) {
2     System.out.println(redCar);
3     paint(redCar);
4     System.out.println(redCar);
5 }
```

The answer is of-course yes. Because the car is immutable it is now impossible for the print statements to print something different. We can now reason about behavior in this code without having to look into the paint method. But what if we do want to change the state of the car? Since we cannot change an ImmutableCar once it is created the only option is to create a new instance of it with different values. So we could create a function like this to change the color of a car to blue.

```
1 public ImmutableCar paintBlue(ImmutableCar car) {
2     return new ImmutableCar("Blue", car.getBrand());
3 }
```

If we now apply this to our original method it would look something like this:

```
1 public void foo(ImmutableCar redCar) {
2     System.out.println(redCar);
3     ImmutableCar blueCar = paintBlue(redCar);
4     System.out.println(blueCar);
5 }
```

If we would now ask the question, will the two print statements print out the same value, the answer is... probably not. paintBlue could still just return the input variable, so a state change is not guaranteed, but it does make it explicit that paintBlue can return a new state. Further hints can now be given by assigning the new state to a different variable name that would clarify it. So while we cannot get rid of state changes, using immutable state automatically forces state changes to become explicit, which allows for more local reasoning.

Tradeoffs

Everything in software engineering is a tradeoff, there are no silver bullets. Using immutable state is no different, it also has downsides. The two main downsides are easy of state changes and memory usage.

State Changes

Currently the only native Java way to do a state change with immutable objects is creating a new object. This does mean having to set all the fields even though you only want to change one field. One current workaround is to use Lombok. We can use the @Builder annotation here to add a builder to our ImmutableCar.

```
1 @Builder(toBuilder = true)
2 public record ImmutableCar(String color, String brand) { }
```

Using this we can now use the toBuilder() method to create a builder that will have all current fields as default values, thus we can use this to only change one field.

```
1 ImmutableCar car = new ImmutableCar("Red", "Volvo");
2 var blueCar = car.toBuilder().color("Blue").build();
```

This way we can easily create a new immutable object which is a copy except of the old one except for the color attribute. Another option is to use the library RecordBuilder which does something similar. Both are not ideal solutions because they require additional libraries, but the Java team is working hard on JEP 468 to make sure that this becomes a language feature which will look something like this:

```
1 ImmutableCar car = new ImmutableCar("Red", "Volvo");
2 var blueCar = car with { color = "Blue"; };
```

Memory / Performance Issues Having to create a new object every time a state change happens can affect performance. We can see this quite well with the String class. Strings are immutable in Java, which means we have to create a new instance if we want to change a String. In this code we see that we need three String objects to do String concatenation.

```
var c = "a" + "b";
```

An object for “a”, “b” and the result in c. If Strings would be mutable we could have added “b” to “a” and we would not have needed a third object. This means concatenating Strings is an expensive operation. We can see this clearly when we try to do this in a loop:

```
1 public String foo(String input) {
2     var result = input;
3     for (var i = 0; i < 1_000_000; i++) {
4         result += "0";
5     }
6     return result;
7 }
```

This will take a long time to execute because we keep creating new objects. This is the reason the `StringBuilder` was introduced, to limit the amount of objects created, since concatenation only happens at the very end when calling `toString()`.

```
1 public String foo(String input) {
2     var builder = new StringBuilder(input);
3     for(var i = 0; i < 1_000_000; i++) {
4         builder.append("0");
5     }
6     return builder.toString();
7 }
```

So if this is the case, why did they decide to make String immutable. First of all, in a lot of cases it is not a big problem, as long as you are not concatenating Strings in a huge loop you will not notice performance problems. The same is true for other immutable objects, as long as you are not creating thousands of them within a loop you will not really notice the performance costs.

Sometimes immutable objects can actually be beneficial for memory usage. Because these objects are immutable it is safe to share them across different threads. This also means it is safe to have multiple pointers pointing to the same String. This way if a String is reused it is safe to use the same object instead of having to create a duplicate object. We can see that in the next example

```
1 public static void main(String[] args) {
2     System.out.println("a" == "a"); // True
3     System.out.println("a" == a()); // False
4 }
5 public static String a(){
6     return "ab".substring(0,1);
7 }
```

Because the compiler figured out the Strings “a” are all the same String, it reuses this object instead of creating a new String every time. This is why `==` comparison sometimes works on Strings. As demonstrated, the compiler cannot always optimize for this, so it is still better to use `.equals` for comparison.

Best of Both Worlds

We demonstrated the advantages of using immutable state for reducing cognitive load and the advantages of mutable state for performance. They don’t have to be mutually exclusive. As long as the input and output of a function is immutable, we don’t really care about using mutable state within function scope.

```
1 public ImmutableCar foo(ImmutableCar car) {
2     StringBuilder builder = new StringBuilder(car.brand());
3     for (int i = 0; i < 1_000_000; i++) {
4         builder.append(i);
5     }
6     return new ImmutableCar(car.color(), builder.toString());
7 }
```

Here both `builder` and `i` are mutable variables but the input and output of the function is immutable. No mutable state escapes the function scope, therefor the rest of the application would not see the difference between this function implemented this way or if we would use recursion for example to get rid of these mutable variables. As long as the mutable state

stays within function scope it is fully thread safe and the rest of the application can reason in immutable state. This way we have the best of both worlds, we have the performance of mutable state and the reduced cognitive load of immutable state for the rest of the application.

Using the Type System

There is a principle called “Illegal states should not be representable”. This means objects should not be created in an illegal state (we can use the constructor for this) and it should not be able to change to an illegal state (immutability guarantees this). An illegal state in this case is a value that is not supposed to be there or does not fit certain constraints. Or if a value is null when it should not be null. But some values can be null like in the following example:

```
1 public record Person(  
2     String firstName,  
3     @Nullable String middleName,  
4     String lastName  
5 ) {}
```

Here middleName is nullable because it is an optional field. Since not everyone has a middle name this is a true optional field of Person. But not every nullable field is really an optional field. If we take a look at the following class:

```
1 public record Contract(  
2     String text,  
3     @Nullable String signature  
4 ) {}
```

we can see that signature is nullable. It is however not an optional field, instead it represents a state change. When the contract is created there will not be a signature, but at some point in time the signature will be provided and should not be null afterwards. Since the field is nullable we still need to take this into account in the rest of our application, resulting in code like this:

```
1 public void startProject(Contract contract) {  
2     if(contract.signature() == null){  
3         throw new IllegalStateException();  
4     }  
5 }
```

Where in order to start our project we need to check if Contract is in a legal state in order to continue. This can lead to a lot of checks in our code base because we cannot guarantee that Contract is in a legal state at that point in time. An alternative approach is capturing this state change in our type system.

```
1 public record UnsignedContract(String text){}  
2 public record SignedContract(String text, String signature){}
```

By creating a separate class for both states, we now see that both classes only contain non-null fields. This makes sure that we do not need to do any null checks in our code base anymore.

```
1 public void startProject(SignedContract contract) {  
2     // No null check needed  
3 }
```

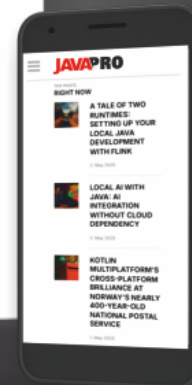
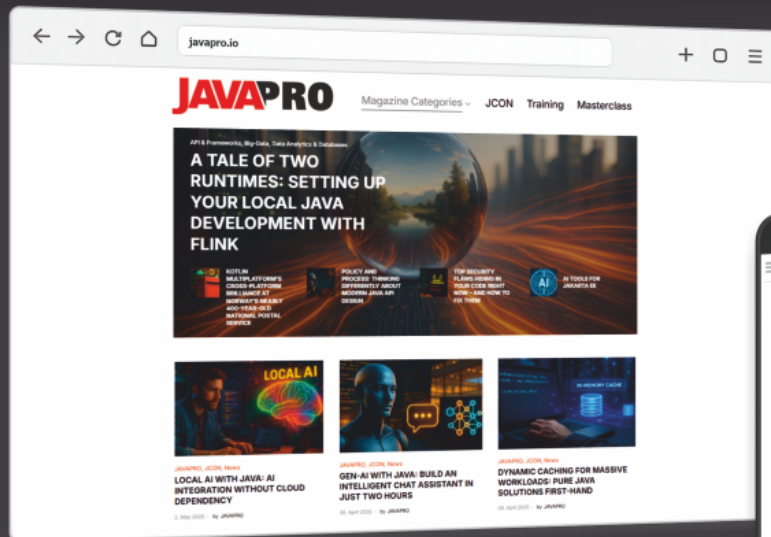
There still needs to be validation to make sure there is a signature, but this can now be moved to the creation of the Signed-Contract class. This means that this validation only needs to take place at class creation and all other parts of the code can rely on a safe, immutable object.

Conclusion

We learned that by using immutable objects we are required to code in certain ways that tend to reduce cognitive load by enabling local reasoning and making state changes explicit. Mutable state is not necessarily bad, but try to keep it within function scope so it does not affect the rest of the application. And by using the type system to represent different state changes in different classes we make sure that classes are always created in a correct and trustworthy state.

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO**www.javapro.io****www.javapro.io**



Writing Readable Code with Algebraic Data Types and Pattern Matching in Java

Algebraic Data Types

Data Oriented Programming

Pattern Matching

Project Amber

Readability

Author: Balkrishna Rawool

Balkrishna Rawool works as an Engineering Lead at ING Bank. He is also a frequent speaker at renowned tech conferences. He has passion for continuous learning and genuine desire to sharing knowledge. Although he has been working with Java for many years, he finds latest developments in Java quite exciting. In his free time, he enjoys playing chess. The analytical thinking and foresight used in chess is something he finds useful in software development.



LinkedIn

Writing readable code is essential for the maintainability of any application. There are many techniques we can use to be able to write readable code. This article discusses one such technique that uses Algebraic Data Types and Pattern Matching, a few of the important features from Project Amber [1]. We will first discuss what Algebraic Data Types are, and then we will discuss some Pattern Matching features from Java. Finally, we will see an example where we can use them to write readable code.

Algebraic Data Types

Algebraic Data Types (ADTs) are data-types that are created by combining existing data-types, mainly by applying algebraic operations to them. There are two main categories of ADTs: Product Types and Sum Types. Let's explore them both.

Product Types

In Product Types, each possible value of the resulting data-type has a value for each of its constituent types. For example, we can create Point type by combining two int types (two coordinates of a point).

In Java, a product-type can be implemented by using record [2].

```
1 record Point(int x, int y) { }
```

Another example is an Address has a houseNumber, a street, a city and a country. In Java this can be implemented as:

```
1 record Address(int houseNumber, String street, String city, String country) { }
```

Sum Types

In Sum Types, each possible value of the resulting data-type is one of the values of its constituent types. For example, let's say a shape can either be a triangle, a quadrilateral or a circle.

In Java, we can implement this by creating sealed interface [3] called Shape with 3 implementations: a Triangle, a Quadrilateral and a Circle. This way, any instance of Triangle, Quadrilateral or Circle is of type Shape. Also, a Shape instance can only be one of these 3 implementation types and nothing else.

```
1 sealed interface Shape {
2     record Triangle(Point p1, Point p2, Point p3) implements Shape { }
3     record Quadrilateral(Point p1, Point p2, Point p3, Point p4) implements Shape { }
4     record Circle(Point p1, int radius) implements Shape { }
5 }
```

Pattern Matching

Pattern Matching is checking for the presence of a pattern in a given sequence of tokens. There are many pattern-matching features in Java introduced as part of Project Amber. Let's discuss some of these features, which are needed for our example later on.

Pattern Matching for Instanceof

instanceof is an operator that you can use to check if an object is an instance of a class or interface. This way you can safely cast the object to the class or interface. Typically, this is followed by using the object and accessing the members of the class or interface.

With this feature, you can now do all that in the same statement. For example:

```
1 if (object instanceof Point) {
2     Point p = (Point) object;
3     double dist = Math.sqrt(p.x * p.x + p.y * p.y);
4 }
```

Here we are checking if object is an instance of Point, and if it is, we create a variable p of type Point (by casting object to Point) and we can use that variable in the subsequent code. All of this can be done with this [4]:

```
1 if (object instanceof Point p) {
2     double dist = Math.sqrt(p.x() * p.x() + p.y() * p.y());
3 }
```

Record Patterns

In the above example, if Point is a record, we could deconstruct [5] it into its constituent members. As shown in the example below, we can access x and y coordinates of the point and calculate the distance of the point from origin.

```
1 if (object instanceof Point(int x, int y)) {
2     double dist = Math.sqrt(x * x + y * y);
3 }
```

We can also apply the deconstruction pattern recursively. For example,

```
1 if (shape instanceof Circle( Point(int x, int y), int radius)) {
2     double dist = Math.sqrt(x * x + y * y) - radius;
3 }
```

Here, we check if the given shape is a circle, and if it is, we deconstruct the circle into a point (center of the circle) and radius. And then we further deconstruct the point into its x and y coordinates.

Pattern Matching for Switch

Record patterns can also be used in switch expressions and statements [6]. For example,

```
1 int sides = switch (shape) {
2     case Triangle(Point p1, Point p2, Point p3) -> 3;
3     case Quadrilateral(Point p1, Point p2, Point p3, Point p4) -> 4;
4     case Circle(Point p1, int radius) -> 0;
5 };
```

In the example above, we calculate the sides of a shape. When the shape is triangle, we return 3. When the shape is quadrilateral, we return 4 and when the shape is circle, we return 0.

Unnamed Variables and Patterns

When a variable is unused, we can be explicit about this by using an underscore as a name of the variable. That way we can inform the compiler (and readers of the code) that we don't plan to use it.

For example, the above example can be rewritten with all variables in record patterns as underscores [7]:

```
1 int sides = switch (shape) {
2     case Triangle(Point _, Point _, Point _) -> 3;
3     case Quadrilateral(Point _, Point _, Point _, Point _) -> 4;
4     case Circle(Point _, int _) -> 0;
5 };
```

We can make it even more concise by not deconstructing the records as below:

```
1 int sides = switch (shape) {
2     case Triangle _ -> 3;
3     case Quadrilateral _ -> 4;
4     case Circle _ -> 0;
5 };
```

Using Them All Together

Let's now take a look at how we can use all these features together.

What we want to do is write a program that can evaluate such expressions, for given set of variables-values:

```
1 4x + 3y + 2z + 1
2 23ab + 34bc + 45ac
```

If we analyse the expressions above, we notice that there are some variables, and these expressions are created by applying some operations to these variables. The variables in the above examples are x, y, z and a, b, c. The operations that are used in above examples are addition and multiplication.

If we think of all possible expressions that we can form by these rules, we see that a valid expression can be a constant, a variable, addition of two valid expressions or multiplication two valid expressions. We can denote this with a grammar like this:

```
1 Expression = a constant | a variable | an expression + an expression | an expression \*
               an expression
```

We can create all valid expressions using this grammar, and any expression created by this grammar will be a valid expression.

This tells us that an expression can be one of four things: a constant, a variable, addition of two expressions and multiplication of two expressions. And that means an expression is a sum-type. So let's implement this using a sealed interface as below:

```
1 sealed interface Expression {
2
3     record ConstExpr(int value) implements Expression { }
4     record VarExpr(String name) implements Expression { }
5     record AdditionExpr(Expression left, Expression right)
6         implements Expression { }
7     record MultiplicationExpr(Expression left, Expression right)
8         implements Expression { }
9 }
```

Here we have four implementations of Expression, viz., ConstExpr (a constant expression), VarExpr (a variable expression), AdditionExpr (an addition expression) and MultiplicationExpr (a multiplication expression). These four are records. ConstExpr can only have a single integer value. VarExpr has a variable name as its member. AdditionExpr and MultiplicationExpr both have two expressions (left and right) as their members. So these four are product-types, and expression is a sum-type. By using Algebraic Data Types for the definition of Expression, we ensure that any instance of Expression will be a valid expression.

Now for the evaluation part, we create a default method evaluate() which takes two arguments: an expression to be evaluated and a Map<String, Integer> of variable-values. We can implement this method as below:

```
1 default int evaluate(Map<String, Integer> values) {
2     return switch (this) {
3         case ConstExpr(var value) -> value;
4         case VarExpr(var name) -> values.get(name);
5         case AdditionExpr(var left, var right) ->
6             left.evaluate(values) + right.evaluate(values);
7         case MultiplicationExpr(var left, var right) ->
8             left.evaluate(values) * right.evaluate(values);
9     };
10 }
```

Here, we use pattern matching for switch expression to check what type of expression it is and depending on the type of the expression, we evaluate it differently. So if the expression is a constant expression, we already know its value, and we can simply return it. If it is a variable expression, we can retrieve the value from the map being passed in the argument and return it. If it is an addition expression, we recursively evaluate the left expression and the right expression and then simply add the two values to get the result and return it. The same is the case for multiplication expression, with the only difference that we return the multiplication of the results of left and right expressions.

Now using this, let's see how we can evaluate the two example expressions. Before that, let's add some methods in Expression to make creating expressions easier:

```

1  static Expression constant(int value) {
2      return new ConstExpr(value);
3  }
4  static Expression varExpr(String name) {
5      return new VarExpr(name);
6  }
7  default Expression plus(Expression other) {
8      return new AdditionExpr(this, other);
9  }
10 default Expression times(Expression other) {
11     return new MultiplicationExpr(this, other);
12 }

```

The first expression can be evaluated as below (for values: x=10, y=15 and z=20):

```

1  // 4x + 3y + 2z + 1
2  static void main() {
3      var expr1 =
4          constant(4).times(varExpr("x"))
5              .plus(
6                  constant(3).times(varExpr("y"))
7              ).plus(
8                  constant(2).times(varExpr("z"))
9              ).plus(
10                 constant(1)
11             );
12
13     var values = Map.of("x", 10, "y", 15, "z", 20);
14     System.out.println(expr1.evaluate(values));
15 }
16
17 // Output:
18 // 126

```

And the second one can be evaluated as this (for values: a=10, b=15 and c=20):

```

1  // 23ab + 34bc + 45ac
2  static void main() {
3      var expr2 =
4          constant(23).times(varExpr("a")).times(varExpr("b"))
5              .plus(
6                  constant(34).times(varExpr("b")).times(varExpr("c"))
7              ).plus(
8                  constant(45).times(varExpr("a")).times(varExpr("c"))
9              );
10
11     var values = Map.of("a", 10, "b", 15, "c", 20);
12     System.out.println(expr2.evaluate(values));
13 }
14
15 // Output
16 // 22650

```

Data Oriented Programming

If you analyse the implementation of the example above, what we notice is that we have divided the implementation into two parts: definition of the data-model (the Expression type) and implementation of the behaviour (the evaluate() method).

To define the data-model, we used Algebraic Data Types, viz., records and sealed-types. This ensures that only valid states

are allowed to exist. To define the behaviour we used powerful pattern-matching features. This makes code easy to read and easy to understand. This pattern is also known as Data Oriented Programming (DOP) [8] [9].

DOP is based on following principles:

- Model data immutably and transparently.
- Model the data, the whole data, and nothing but the data.
- Make illegal states unrepresentable.
- Separate operations from data.

If we follow these principles, we can utilize Data Oriented Programming to greatly increase the readability of Java programs.

References

- [1] <https://openjdk.org/projects/amber/>
- [2] <https://openjdk.org/jeps/395>
- [3] <https://openjdk.org/jeps/409>
- [4] <https://openjdk.org/jeps/394>
- [5] <https://openjdk.org/jeps/440>
- [6] <https://openjdk.org/jeps/441>
- [7] <https://openjdk.org/jeps/456>
- [8] <https://www.infoq.com/articles/data-oriented-programming-java/>
- [9] <https://inside.java/2024/05/23/dop-v1-1-introduction/>

› **Back to the Table of Contents** ‹



Java Streams Evolution: From Java 8 to Today

Architecture

Java 25

Streams

Author: Mihaela Gheorghe-Roman

Mihaela Gheorghe-Roman is a Software Architect at Systematic, Romania. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge at work, at the university, at different national and international IT conferences. In her professional career, spanning over 14+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular. Apart from her passion for programming, she loves parrots, she is enthusiastic about traveling the world and doing a lot of sports, from running at marathons, martial arts and to different aerobic trainings.



LinkedIn

The Java Streams evolution, starting from Java 8, has significantly transformed how we process data collections with a clean, declarative style. What began as straightforward map, filter, and reduce operations has expanded into sophisticated stream pipelines and advanced features like Gatherers. Gaining a solid grasp of this evolution empowers you to write more elegant and efficient Java code.

This article takes you through the journey of Java Streams, tracing their development from Java 8 to the present. Along the way, we'll highlight important enhancements, practical use cases, and best practices to help you leverage Streams to their fullest potential.

Introduction to Java Streams

Java Streams provide a clean, declarative API for processing collections and data pipelines. At its core, a Stream is an abstraction for processing sequences of elements. Instead of manually iterating over collections like Lists, Sets, or arrays with loops, Streams allow us to express operations in a more declarative and concise way.

The Java Streams evolution has reshaped how developers approach data processing, moving from imperative loops to declarative pipelines.

Essentially, Streams enable functional-style operations on data collections. This means we can apply transformations such as `filter`, `map`, `reduce`, and others, focusing on *what* we want to achieve rather than *how* to implement it. The functional nature comes from treating data as immutable and applying pure functions – operations that do not modify the original data but instead return transformed results. With Streams, we avoid writing explicit loops and managing intermediate state, leading to clearer and more maintainable code.

Motivation Behind Streams

Understanding the Java Streams evolution helps us appreciate how Java shifted from verbose, error-prone loops to expressive and efficient data pipelines.

Imperative Limitations

Before Streams, Java developers relied heavily on imperative-style loops – such as `for` or `while` loops – to process collections. These approaches tend to be verbose, error-prone, and harder to maintain, especially as the logic grows more complex. For instance, a simple task like filtering a list of names to print those starting with a specific character often required multiple lines of boilerplate code. Developers had to manually manage the loop, perform condition checks, and handle output, resulting in repetitive and less readable code.

Functional Programming Paradigm

Java has long been an object-oriented language, but the need for a more functional approach to handling collections became increasingly apparent. Functional programming enables developers to focus on *what* they want to achieve with data, rather than *how* to manage each iteration or accumulation step, resulting in clearer and more concise code.

Parallelism

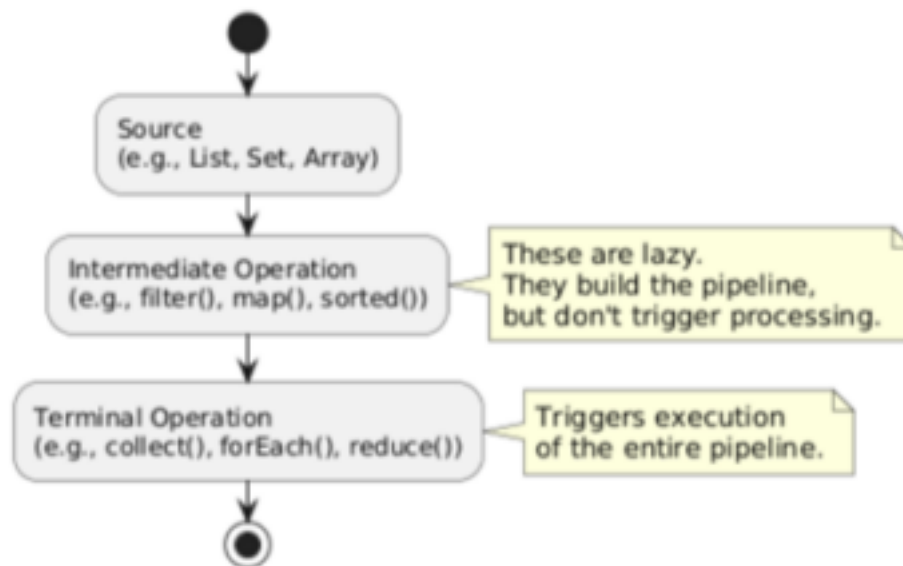
Another major challenge with traditional loops was the complexity of parallelizing operations. Managing threads manually in Java was often error-prone and cumbersome. Streams simplified this by providing built-in parallelism. By simply switching to a parallel stream using `.parallelStream()`, developers can leverage multi-core processors effortlessly, while the framework handles thread management behind the scenes, minimizing the complexity on our end.

Stream Pipeline Architecture

The design of the Stream pipeline is a core part of the **Java Streams evolution**, enabling lazy evaluation and efficient processing. A Stream pipeline consists of three main components:

1. A Stream always begins with a **source**, which can be a collection such as a `List` or `Set`, an array, or even an I/O channel. This source represents the origin of the data to be processed.

2. Once a source is defined, **intermediate operations** can be applied to transform the data. Operations such as `filter()`, `map()`, and `sorted()` fall into this category. These operations are *lazy*, meaning they are not executed immediately. Instead, they are composed into a Stream pipeline and deferred until a terminal operation is invoked. In essence, the Stream builds up a sequence of transformations, but no data is processed until the final step – triggered by a terminal operation.
3. **Terminal operations** consume the Stream and produce a result or a side effect. Methods such as `collect()`, `forEach()`, and `reduce()` are examples of terminal operations. When one of these is invoked, the entire Stream pipeline is executed, and the data is processed according to the intermediate operations defined earlier.



As illustrated in the diagram above, data flows from the source through a sequence of intermediate operations and concludes with a terminal operation. Thanks to lazy evaluation, these intermediate steps are not executed until a terminal operation is invoked. This design ensures that Streams remain efficient and scalable, processing only what's necessary.

Intermediate vs. Terminal Operations

Intermediate operations

Intermediate operations are responsible for building the Stream pipeline. They include methods such as `filter()`, `map()`, and `sorted()`. These operations are *lazy* – they do not perform any processing until the pipeline is triggered by a terminal operation. Each intermediate operation returns a new Stream, making them inherently composable and allowing multiple operations to be chained together in a fluent, readable manner.

Terminal operations

Terminal operations are responsible for consuming the Stream and producing either a result or a side effect, such as printing output or performing transformations. Common terminal operations include `forEach()`, `collect()`, `count()`, and `reduce()` – some of the most frequently used in real-world applications. These operations trigger the execution of the Stream pipeline and finalize the processing of the data.

Evolution from JDK 8 to TODAY

The Java Streams evolution began in JDK 8 with foundational features like `map`, `filter`, and `reduce`. Over subsequent releases, the Java Streams evolution introduced powerful enhancements such as `takeWhile()`, `gatherers`, and `transform()`, broadening the API's expressiveness and performance capabilities.

Java 8: The Beginning of the Java Streams Evolution

Java 8 was a game-changer for the language. By introducing the Stream API, it allowed us to perform functional-style operations on collections of data.

For example, with a simple Stream pipeline, we can process data by applying operations such as `filter()`, `map()`, and `reduce()`. This shift allowed developers to move from imperative code to a more declarative and expressive style. Additionally, Java 8 introduced parallel streams, enabling developers to leverage multi-core processing with minimal effort, eliminating the need for manual thread management and reducing the complexity of concurrent programming.

Here's a basic pipeline that filters, transforms, and consumes elements:

```
1 List<String> list = List.of("a", "b", "c");
2 list.stream()
3     .filter(s -> !s.equals("b"))
4     .map(String::toUpperCase)
5     .forEach(System.out::println); // A, C
```

Java 9: Stream Slicing

Java 9 introduced two important additions to the Stream API – `takeWhile()` and `dropWhile()` – which enhanced the flexibility of stream iteration by allowing more fine-grained control over how elements are processed based on conditions.

With `takeWhile()`, we can take elements from a stream as long as a condition holds true, which is great for conditional processing. In the below code example, `takeWhile()` is used to stop iterating as soon as we encounter an element that doesn't meet the defined criteria – in this case when this is greater than 10. This is particularly useful when working with sorted data.

```
1 Stream<Integer> numbers = Stream.iterate(1, n -> n + 1)
2     .takeWhile(n -> n < 10);
3 numbers.forEach(System.out::println); // 1, 2, ..., 9
```

On the other hand, `dropWhile()` does the reverse – it skips elements from the stream as long as a condition holds true. This is useful when we want to discard elements from the beginning of the stream until a certain condition is met.

```
1 Stream.of(1, 2, 3, 4, 1, 2)
2     .dropWhile(n -> n < 3)
3     .forEach(System.out::println); // 3, 4, 1, 2
```

Java 11: var in Lambda Parameters

While Java 11 didn't introduce major changes to the Stream API itself, it did bring improvements that indirectly enhanced stream usability. One notable addition was the ability to use `var` in lambda parameters, which can improve code readability and support annotations in lambda expressions.

An example can be seen below.

```
1 List<String> list = List.of("a", "b", "c");
2 list.stream()
3     .map((var s) -> s.toUpperCase())
4     .forEach(System.out::println); // A, B, C
```

Java 16: Stable toList()

Java 16 introduced a small but widely appreciated quality-of-life improvement: the addition of the `.toList()` method directly on the `Stream` interface. This made it easier and more intuitive to collect stream elements into an immutable list without the need for `Collectors.toList()`.

Previously, converting a stream into a list required using `collect(Collectors.toList())`, which was a bit verbose. With this new method, we can simply call `.toList()`, resulting in cleaner and more readable code.

```

1 List<String> list = List.of("a", "b", "c");
2 list.stream()
3     .map((var s) -> s.toUpperCase())
4     .toList(); // [A, B, C]

```

Java 21: Enhanced mapMulti

Java 21 introduced a powerful new method, `mapMulti()`, offering a more controlled and efficient alternative to `flatMap()`. This method is particularly useful when mapping a single element to zero, one, or multiple outputs without the overhead of creating intermediate streams.

```

1 Stream.of(1, 2)
2     .mapMulti((n, c) -> { c.accept(n); c.accept(n * 10); })
3     .forEach(System.out::println); // 1, 10, 2, 20

```

Java 22: Gatherers (Preview)

Java 22 introduced a preview feature called Stream Gatherers, which enables the use of the new `gather()` method to collect elements from a Stream into more flexible and customizable data structures.

One of the key enhancements is the introduction of batch processing within Streams, allowing elements to be collected into windows or chunks. For example, in the code below, a list of strings is divided into groups of three using `Gatherers.windowFixed(3)`, enabling more efficient and organized processing of data in batches.

```

1 List<String> groups = List.of("A","B","C","D","E").stream()
2     .gather(Gatherers.windowFixed(3))
3     .toList();

```

This enhancement simplifies working with batched data by eliminating the need to implement complex batching logic manually.

Java 23: Stream Transform

Java 23 introduces several enhancements, including the new `Stream.transform()` method. This addition enables functional transformation of stream elements in a more concise and readable manner, further improving the expressiveness of the Stream API.

```

1 Stream<String> up = Stream.of("a", "b")
2     .transform(s -> s.map(String::toUpperCase));

```

In addition to the `transform()` method, Java 23 brings significant improvements to parallel stream processing. The JVM can now distribute work more efficiently across multiple threads, reducing overhead during the processing of large data sets. As a result, applications can handle larger collections with improved performance and scalability.

Java 24: Flatten to Primitive Streams

Java 24 brings several key enhancements to the Stream API, aimed at simplifying data manipulation and boosting performance.

One of the most notable additions in Java 24 is the introduction of `Stream.flatMapToInt()` and `Stream.flatMapToDouble()` methods. These allow flattening streams of collections into primitive streams of `int` and `double`, respectively. This enhancement is especially valuable when working with large datasets of primitive types, improving both performance and code clarity.

In the example below, we flatten a list of lists into a single stream of integers. This approach is more memory-efficient than using a `Stream<Integer>` because it works directly with the primitive `int` type, reducing boxing overhead.

```
1 List<List<Integer>> nested = List.of(List.of(1,2), List.of(3));
2 IntStream flat = nested.stream().flatMapToInt(l -> l.stream().mapToInt(i -> i));
```

Java Streams Evolution Continues in Java 25

Java 25 is set to finalize the Stream Gatherers API, bringing flexible and efficient batch processing capabilities to the Stream API. This makes it easier to work with windowed or chunked data collections in a more declarative way.

Meanwhile, Java continues to advance features related to concurrency, including Scoped Values and Structured Concurrency, aimed at safer context propagation and improved management of concurrent tasks.

Best Practices and Common Pitfalls

Leveraging Features in the Java Streams Evolution for Better Code:

Use Streams for Simple and Composable Operations

Streams excel at tasks like filtering, mapping, and reducing. Keep stream pipelines focused and avoid embedding complex or imperative logic within them to maintain readability and maintainability.

Take Advantage of Lazy Evaluation

Streams are evaluated lazily, meaning operations are only performed when a terminal operation is invoked. Leveraging this behavior helps prevent unnecessary computations and can improve performance.

Profile before applying `parallelStream()`

While parallel streams offer automatic parallelization, it's important to ensure that the operations are independent and the dataset is sufficiently large to benefit from parallelism. Profiling your code helps determine if using `parallelStream()` will yield meaningful performance improvements.

Common Pitfalls in the Java Streams Evolution to Avoid:

Modifying state inside a stream

Avoid altering external state (such as lists or counters) within a Stream pipeline. Doing so can cause unpredictable behavior and introduces subtle, hard-to-debug errors.

Using side-effects in `map()` or `filter()`.

Stream operations should ideally be free of side effects. Modifying external variables inside methods like `map()` or `filter()` breaks the functional programming principles of Streams and can lead to difficult-to-trace bugs.

Confusing Intermediate and Terminal Operations

Remember that intermediate operations are lazy and only executed when a terminal operation is invoked. Mixing these up can result in unexpected behavior or degraded performance.

Summary

In summary, Java Streams evolution provide a powerful, declarative, and functional paradigm for processing data, significantly improving code readability and maintainability. We have examined how intermediate and terminal operations facilitate clear and efficient data transformations and consumption. While parallel streams offer performance gains through concurrent processing, they must be used judiciously to avoid pitfalls related to shared mutable state. Since their introduction in JDK 8, the Stream API has steadily evolved – adding methods like `takeWhile()`, `dropWhile()`, and `transform()` – to offer enhanced flexibility and scalability. Adhering to best practices, such as leveraging terminal operations and avoiding external flags or mutable states in parallel streams, ultimately results in cleaner, more reliable code.

References:

<https://www.baeldung.com/java-25-features>

https://download.java.net/java/early_access/jdk25/docs/api/

<https://openjdk.org/jeps/506>

<https://www.baeldung.com/java-mapmulti>

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



JAVAPRO

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

record: to set down in writing or the like, as for the purpose of preserving evidence. [dictionary.com]

The Minimalist — A Resolution Working from home has taught some of us to be thinking of being minimalistic. It was no different for our protagonist “Dev” — Dev decided to being minimalistic with respect to some of the dresses — So he decided to keep track of what he is wearing to donate the unused dresses. To keep it simple, he decided to keep track of two items, shirts and shoes, and wrote the following class in Java:

```
1 final class Attire {
2     private final String shirt;
3     private final String shoe;
4     public Attire(String shirt, String shoe) {
5         this.shirt = shirt;
6         this.shoe = shoe;
7     }
8 }
```

Impatient he was, Dev now wanted to populate and print out the values for a basic sanity testing:

```
1 Attire at1 = new Attire("formal", "black");
2 System.out.println(at1);
```

“Whoa”, cried he, when he saw: “Attire@251a69d7”. Not satisfied, he wrote the following code — please spare the opinions on the code — he just wrote a short code to print the values as follows:

```
1 @Override
2 public String toString() {
3     StringBuilder sb = new StringBuilder(this.getClass().getName());
4     sb.append("[");
5     sb.append("shirt=");
6     sb.append(this.shirt);
7     sb.append(",");
8     sb.append("shoe=");
9     sb.append(this.shoe);
10    sb.append("]");
11    return sb.toString();
12 }
```

And he was satisfied with the output: “Attire[shirt=formal,shoe=black]”. And then he realized that he is just filling the values once and forgetting them — notice they are private final fields since Dev doesn’t like anyone to tamper with his shirts and shoes and its a record to keep, he needs to have getters/accessors — and he wanted the accessors to have the same names as the field names, so here’s what the code looked like:

```
1 public String shirt() {
2     return this.shirt;
3 }
4 public String shoe() {
5     return this.shoe;
6 }
```

Before closing up the class, he wanted to make sure that he got a true when he compared records of identical shirt-shoe values and tried out:

```
1 Attire at1 = new Attire("formal", "black");
2 Attire at2 = new Attire("formal", "black");
3 System.out.println(at1);
```

But we know the answer — “false”. So being a determined person, he decided to write the equals() himself:

```

1  @Override
2  public boolean equals(Object obj) {
3      if (!(obj instanceof Attire))
4          return false;
5      Attire other = (Attire) obj;
6      return this.shirt.equals(other.shirt()) &&
7          this.shoe.equals(other.shoe());
8  }

```

Luckily, Dev had read Effective Java by Joshua Bloch and knew that he had to override hashCode() as well. Anyway, with that also out of the way, now Dev can comfortably compare and hash the records and decide what items are used and what are not. Let us look at the complete class definition code that Dev wrote:

```

1  final class Attire {
2      private final String shirt;
3      private final String shoe;
4      // Constructor
5      public Attire(String shirt, String shoe) {
6          this.shirt = shirt;
7          this.shoe = shoe;
8      }
9      //Accessors
10     public String shirt() {
11         return this.shirt;
12     }
13     public String shoe() {
14         return this.shoe;
15     }
16     // Equals and HashCode Definitions
17     @Override
18     public boolean equals(Object obj) {
19         if (!(obj instanceof Attire))
20             return false;
21         Attire other = (Attire) obj;
22         return this.shirt.equals(other.shirt()) &&
23             this.shoe.equals(other.shoe());
24     }
25     @Override
26     public int hashCode() { // can do better here
27         return this.shirt.hashCode() + this.shoe.hashCode();
28     }
29     // Print out
30     @Override
31     public String toString() { // oh please cleanup this code
32         StringBuilder sb = new StringBuilder(this.getClass().getName());
33         sb.append("[");
34         sb.append("shirt=");
35         sb.append(this.shirt);
36         sb.append(",");
37         sb.append("shoe=");
38         sb.append(this.shoe);
39         sb.append("]");
40         return sb.toString();
41     }
42 }

```

Along with us, Dev also looked at his class definition, and that was a problem. He wondered that the his idea to start with was to be minimalistic but probably he would end up being minimalistic in his clothes, but from the code point of view, this is far from being minimalistic and decided to call the Java folks and explain the problem and ask if they have a solution... And he called...

The Response

“What did you smoke last night?” — **No**, that was not the response he received. And to disappoint you, neither am I telling the answer if you had asked me that question :). In fact, the response he received was, “use Java 16 and the entire code you wrote is just one line now”:

```
1 record MyAttire(String shirt, String shoe) {}
```

Not believing this, he used the Disassembler of Eclipse IDE and saw that there indeed were the fields `shirt` and `shoe`, **private final** as he deemed, so were the accessors `shirt()` and `shoe()` along with the rest of methods. We will see the disassembled code a little later. For now, Dev understood that the compiler is doing this magic under the hood to provide this feature to the users.

The Feature — Records

Records provide the [Cartesian Product](#) part of the story of [Algebraic Data Types](#) — It provides the notion of logical And. okay, hold on before you yawn and sign off, let me explain. Every day, we do the cartesian product operation — Take the example of Dev, he selects one shirt out of the **set** of shirts **and** chooses a shoe from a **set** of shoes — whether the set of shoes is finite or not, is apparently a very sensitive topic for some, so let us not go there at all :). Suffice to say that he does this operation, and creates a pair of shirt and shoe — this indeed is equivalent to the cartesian product of two sets and choosing one of the outcomes — and each of these outcomes is an instance of a record.

Records also provide the notion of constant-ness — **private final** fields are just one part, `hashCode()` and `equals()` add to the story where the values of the fields are the same for two records, they will be considered equal as in:

```
1 MyAttire m = new MyAttire("formal", "black");
2 MyAttire m2 = new MyAttire("formal", "black");
3 System.out.println(m.equals(m2));
```

This would give out a “true” since the values are the same. And as the name implies, once it is recorded, we cannot tamper with its values.

Why do we want such a construct? The answer is it’s a convenient way to provide the notion of data classes — some call it *tuples* [[Python world](#)]. Records can be thought of providing itemized tuples with named accessors. And data classes blend themselves nicely into the Pattern Matching [see [here](#) for an informal primer] constructs Java is providing now and in the near future. Since this article is about records in detail, let us delve deeper into records.

“record” — Restricted Identifier

First things first — how do we define a record in source code? Using the `record` “restricted identifier”. What is this “restricted identifier” — if you have used `var` or `yield` you would immediately recognize that `record` also falls into the same bucket — record can be used just like an identifier in almost all places — but with restrictions — you cannot have a type named record. And it acts like a keyword in record declaration — quite a context sensitive phenomenon — this complexity is taken care by the compiler. As a user, we just need to know that `record` is the “text” used for defining a record.

The Parent — java.lang.Record

All of us know that when define `class X{}` internally the compiler interprets this as `class X extends java.lang.Object {}`, in the case of a record definition, a record `R` implicitly **extends** `java.lang.Record`, a new class provided by jdk. Does this Record provide the internal methods — actually no, they are provided by a combination of compiler generated methods working in sync with the bootstrap mechanism akin to that of lambda — we will see more of this when we discuss the byte code or disassembly. Also, please note that a record can of course implement interfaces.

Components

Those “things” that look like the formal parameters of records — `String shirt` and `String shoe` in our example — are called *Components* in the record parlance. We now know that internally they get converted into private final fields. Additionally, these components also result in the definition of accessors of these fields. These components can be individually annotated and there indeed is a new annotation element type `ElementType.RECORD_COMPONENT` — Good thing is the annotations percolate to the fields and the accessors as applicable.

Canonical Constructor

Records provide a convenient way to minimize the source code by having a perfect blend of class definition with constructor syntax to provide concise definition albeit with an internal definition of a “canonical” constructor as opposed to the “default” constructor of classes which we are familiar with. While we know that the default constructor doesn’t take any arguments, the “canonical” constructor indeed takes the same number of arguments identical to record components in that order:

```
1 MyAttire(String shirt, String shoe){ /* init fields */}
```

Compact Constructor

A compact construct is just another simpler way to define a canonical constructor, where the parameters can be omitted:

```
1 MyAttire { /* Hey, I am the compact constructor */}
```

Since record is all about minimizing, this is just a convenient syntax for omitting the parameters since the canonical constructor parameters should match with the record definition and hence its easier and safer to omit to use the compact constructor syntax if you want to define a canonical constructor yourself — in fact there indeed is a specific characteristic of compact constructor which we will see while discussing errors.

What about “normal” Constructors?

This brings us to the question whether the “normal” constructors are allowed? Let us agree that anything other than a canonical constructor is a normal constructor — ie any constructor which doesn’t have parameters in the same number and in the same order is a normal constructor. Of course, they are allowed but with a catch — Each such constructor should call a canonical constructor as the first step.

Record — Expanding the Right Way

We just glanced through the different concepts and parts that make up the records — “record” restricted identifier, `java.lang.Record`, Components, Canonical, Compact and Normal constructors. Idea is that it is very easy to create a record using the default definitions. If it is a minimalistic definition, you are ready to go as soon as you write that small definition. However, as you expand and write your own specifics — say the constructors for example, or maybe provide your own methods, add more fields etc, then more rules come into play to make sure that the “constant-ness” of a record is not tampered with. And errors are part of program analysis — in fact when we implemented records in the Java Compiler in Eclipse (ecj), we ended up checking for more than thirty new errors (thirty-five?) for records alone. Let us briefly sample this in the following section.

Errors — A Part of Programs

Let us pick up a few scenarios where the errors will be thrown and understand the reasoning behind these:

- User declared non-static fields are not permitted in a record — Let us say for now, we permit instance fields — and we can immediately see that this goes against the definition of records — All components are defined and filled up at the creation of instance of records; also, think of the equals and hashCode — they are based on the component and a

user defined non-static field torpedoed their effort — removing all constant-guarantees. We are of course at liberty to define static fields.

- Multiple canonical constructors are not allowed — First of all, how do we define multiple canonical constructors? — we can create a compact constructor and a canonical constructor with the same component order. In the source code it would look different but this would result in a clash ending up as duplicate constructors since internally they will have the same parameters.
- Illegal explicit assignment of a final field shirt in compact constructor — This one is interesting; consider:

```
1 record MyAttire(String shirt, String shoe) {
2     MyAttire { // Compact Constructor
3         this.shirt = "";
4     }
5 }
```

The compact constructor is supposed to be really compact and is expected to be added just for making sanity checks — its not allowed to assign fields explicitly in the source code. Compiler internally adds the assignments while generating the code. And hence the error when we have such an assignment in the source code.

We have barely touched less than ten percent of the errors, but this just gives an idea of the care taken to make sure the definition of record is kept sacrosanct and the majority of error checking and reporting done to make sure that all possible errors are caught in the source. We will now take a look at the byte code — and see how our little code gets expanded.

A Disassembled View

The Source code we have is:

```
1 record MyAttire(String shirt, String shoe) {}
```

In the disassembled version, this expands to:

```
1 final class MyAttire extends java.lang.Record
2 ...
3     private final java.lang.String shirt; // fields
4     private final java.lang.String shoe; // fields
5 ...
6 MyAttire(java.lang.String, java.lang.String); // Constructor
7 ...
8 // Accessors
9     public java.lang.String shirt();
10    public java.lang.String shoe();
11 ...
12 //
13    public final java.lang.String toString();
14    public final int hashCode();
15    public final boolean equals(java.lang.Object);
16 ...
```

Our old fellow, Dev, can immediately relate to this and see that this expanded code looks very similar to what he wrote in the first place. Examining further he sees that the record is in fact generated as a class only in the byte code. Being a learner, he was inquisitive and decided to delve more and took a look at one of the methods — the shirt accessor:

```
1 public java.lang.String shirt();
2 0: aload_0
3 1: getfield        #14    // Field shirt:Ljava/lang/String;
4 4: areturn
```


This self-explanatory code just returns the field of shirt as expected and nothing unexpected here, but then he looks at `hashCode` and he sees something strange:

```
1 public final int hashCode();
2 0: aload_0
3 1: invokedynamic #30, 0 // InvokeDynamic #0:hashCode:(LMyAttire;)I
4 6: ireturn
```

He sees an `invokeDynamic` instruction — he remembers something similar when he used `lambda` — and he doesn't see any calculation inside the hash function. So what happens here? Similar to that of `lambda`, there is a bootstrapping mechanism that comes into the play here. On observing at the end of disassembly, there is this code:

```
1 Bootstrap methods:
2 0 : # 47 invokestatic java/lang/runtime/ObjectMethods.bootstrap:
3 ...
4 Method arguments:
5 #1 MyAttire
6 #48 shirt;shoe
7 #50 REF_getField shirt:Ljava/lang/String;
8 #51 REF_getField shoe:Ljava/lang/String;The End of the Beginning
```

The exact mechanics is beyond the scope of this article, but for now we kinda assume that there is this `ObjectMethods` class in JDK which has a method `bootstrap` which take the record, the `String` names of the fields and getters as parameters along-with a method handle and magically returns a method handle that calculates the hash code, which is then called to get the hash code. A similar mechanism for equals as well. In a nutshell, the code for the calculation comes from the JDK itself with inputs from the record. If sufficient interest is there, a separate article can explore this in detail. For now let us conclude the analysis of the disassembled code with just one more detail.

```
1 Record: #Record
2 Components:
3 // Component descriptor #6 Ljava/lang/String;
4 java.lang.String shirt;
5 // Component descriptor #6 Ljava/lang/String;
6 java.lang.String shoe;
```

A new structure to capture the record components is defined in the virtual machine specification — and in a human readable format, the disassembler provides the view above. Adding this here for the sake of completeness.

The Beginning..

With this we come to the end of the discussion on records. The changes indeed require detailed attention and bigger article(s) if we go deeper into specifics, but for now we have done a broad coverage of the concept of records. A record is just one building block for the patterns especially record patterns. Records work in tandem with other features like sealed types, pattern switches etc to complete the overall story. To make the story short, this is not the end but just the beginning...

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



Java 25: Getting the Most Out of Virtual Threads with Structured Task Scopes and Scoped Values

Java 25

Structured Concurrency

Virtual Threads

Author: Christian Woerz

Christian Woerz is a Software Engineer with over 15 years of development experience. He writes code in Java, Kotlin, JavaScript and TypeScript. In his spare time, he enjoys attending conferences where he shares his knowledge. He also frequently uploads advanced TypeScript videos to his YouTube Channel (Typed Rocks).



LinkedIn

With Java 21, the Java platform completely reinvented how threads work by introducing *virtual threads* alongside traditional *platform threads*.

Virtual threads let us create millions of concurrent tasks without worrying about creation time or memory usage. This makes it possible to handle vastly more requests than before, but it doesn't solve every problem that once led developers to reactive programming.

In this article, we'll explore how *StructuredTaskScope* and *ScopedValue*, two powerful features that complement virtual threads, help us write safer, faster, and more maintainable concurrent code.

Structured Task Scopes — the Missing Piece

Introduced as a preview in Java 21, *StructuredTaskScope* provides a structured way to manage concurrent tasks. The API remained stable through Java 24, but with Java 25 the Java team completely redesigned it. Simplifying its usage and improving performance. Let's see what's new and how to use it effectively.

Virtual Threads in Action

Virtual threads remove the need for most thread-pools.

Here's a simple example in a Spring Boot controller:

```
1 @GetMapping("/talk/{speaker}")
2 Talk getTalk(@PathVariable String speaker) {
3     // ~1s
4     return restClient.retrieveTalk(speaker);
5 }
```

One incoming request triggers one outgoing call. Virtual threads handle this perfectly.

But what if we need to gather multiple pieces of data?

```
1 @GetMapping("/speaker/{speaker}")
2 Speaker getSpeaker(@PathVariable String speaker) {
3     // ~1s
4     final var talk = fakeRestClient.retrieveTalk(speaker);
5     // 3 x ~500ms
6     final var infosGoogle = fakeRestClient.retrieveGoogle(speaker);
7     final var infosLinkedin = fakeRestClient.retrieveLinkedin(speaker);
8     // may throw
9     final var infosFacebook = fakeRestClient.retrieveFacebook(speaker);
10    final var bestInfos = Stream.of(infosFacebook, infosGoogle, infosLinkedin)
11        .max(Comparator.comparingInt(info -> info.infoList().size()))
12        .orElseThrow();
13    return new Speaker(talk, bestInfos);
14 }
```

Sequential calls like these mean a total request time of around 2.5 seconds. And if one request fails, the entire call fails. We can do better.

Concurrent Calls with StructuredTaskScope

We want all requests to run concurrently, and we want to handle errors gracefully after all tasks complete. That's where *StructuredTaskScope* comes in.

```

1  private Info getInfo(String speaker) throws InterruptedException {
2      try (var scope = StructuredTaskScope.open(Joiner.awaitAll())) {
3          var infosGoogle = scope.fork(() -> restClient.retrieveGoogle(speaker));
4          var infosLinkedin = scope.fork(() -> restClient.retrieveLinkedin(speaker));
5          var infosFacebook = scope.fork(() -> restClient.retrieveFacebook(speaker));
6          scope.join();
7          return Stream.of(infosGoogle, infosFacebook, infosLinkedin)
8              .filter(task -> task.state() == Subtask.State.SUCCESS)
9              .map(Subtask::get)
10             .max(Comparator.comparingInt(info -> info.infoList().size()))
11             .orElseThrow();
12      }
13  }

```

How it works

- **(1)** We open a new scope with `StructuredTaskScope.open()`. Because scopes implement `AutoCloseable`, we use try-with-resources.
- **(2)** Each call to `fork()` starts a new virtual thread and returns a `Subtask`, similar to a `Future`.
- **(3)** `scope.join()` waits for all subtasks to complete.
- **(4)** We filter successful results, handle failures ourselves, and pick the best response.

With this approach, the total latency drops from ~2.5 seconds to ~1.5 seconds, since all calls run concurrently. By using the `Joiner.awaitAll()`, errors no longer abort the whole request unless we choose to. *Joiners* are a great tool to configure, how we want our scope to handle results.

Joiners — Controlling How Tasks Complete

The *Joiner* interface defines how subtasks are collected and what happens on failure. Java ships several ready-made joiners:

JOINER	DESCRIPTION
<code>Joiner.awaitAll()</code>	Waits for all subtasks to finish.
<code>Joiner.allSuccessfulOrThrow()</code>	Throws if any subtask fails.
<code>Joiner.anySuccessfulResultOrThrow()</code>	Returns when one task succeeds.
<code>Joiner.allUntil(predicate)</code>	Waits until the given condition is met.

For most cases these built-ins are enough, but sometimes you need custom logic. For example if we want to select the most informative result while ignoring failures.

Writing a Custom Joiner

```

1  public class InfoJoiner implements Joiner<Info, Info> {
2
3      private final Collection<Info> results = new ConcurrentLinkedQueue<>();
4      private final Collection<Throwable> exceptions = new ConcurrentLinkedQueue<>();
5
6      @Override
7      public boolean onComplete(Subtask<? extends Info> subtask) {
8          if (subtask.state() == Subtask.State.SUCCESS) {
9              results.add(subtask.get());
10             }
11             else if (subtask.state() == Subtask.State.FAILED) {
12                 exceptions.add(subtask.exception());
13             }
14             return false; // never cancel
15         }
16
17         @Override
18         public Info result() {
19             if (results.isEmpty()) {
20                 var rte = new RuntimeException("All subtasks failed");
21                 exceptions.forEach(rte::addSuppressed);
22                 throw rte;
23             }
24             return results.stream()
25                 .max(Comparator.comparingInt(info -> info.infoList().size()))
26                 .orElseThrow();
27         }
28     }

```

Let's take a quick look into the two methods we are overriding here:

results()

This gets called when `scope.join()` gets called and is a great place to gather all the information and return the result to the caller. In our case we just check, if there are any successful subtasks in our results collection and if so, we get the one result with the most information. If there wasn't any element in the list, we would just add all the Throwable objects from our exceptions list to a newly created RuntimeException and then throw that exception.

onComplete

`onComplete` will get called each time, a subtask has completed with any kind of result be it `SUCCESS` or `FAILED`. This is the perfect place to check the state of our subtask and either add the successful result to our results collection using `subtask.get()` or to add it to our exceptions-list by using `subtask.exception()`. One more interesting part of this method is the boolean return value. This will tell the scope if it should cancel the scope or not. In our case we never want to cancel the scope after a specific subtask has ended, so we always return false. Other default implementations like `anySuccessfulResultOrThrow` are using that functionality as they don't want to wait for all subtasks to finish, but only need one successful result.

Now after implementing everything needed, we can use it in our controller:

```

1  private Info getInfo(String speaker) throws InterruptedException {
2      try (var scope = StructuredTaskScope.open(new InfoJoiner())) {
3          scope.fork(() -> restClient.retrieveGoogle(speaker));
4          scope.fork(() -> restClient.retrieveFacebook(speaker));
5          scope.fork(() -> restClient.retrieveLinkedin(speaker));
6          return scope.join();
7      }
8  }

```

The method is now concise and efficient. `scope.join()` returns the best result directly from our custom joiner.

Combining Concurrent Scopes

Because retrieving the talk and fetching speaker info are independent, we can run both in parallel:

```
1 @GetMapping("/speaker/{speaker}")
2 Speaker getSpeaker(@PathVariable String speaker) throws InterruptedException {
3     try (var scope = StructuredTaskScope.open()) {
4         var talk = scope.fork(() -> fakeRestClient.retrieveTalk(speaker));
5         var infos = scope.fork(() -> getInfo(speaker));
6         scope.join();
7         return new Speaker(talk.get(), infos.get());
8     }
9 }
```

Two tasks, two virtual threads, all running concurrently.

Why Structured Task Scopes Matter

Easy to create: It's never been simpler to write concurrent Java code.

Predictable error handling: Decide whether to fail fast or collect results first.

Extensible: Build your own `Joiner` for custom aggregation logic.

Testable: Because execution is still synchronous from the caller's point of view, standard unit tests work perfectly.

Scoped Values – Replacing ThreadLocal

Before structured task scopes, Java applications often relied on *ThreadLocal* to store request-specific context. But with multiple virtual threads per request, `ThreadLocal` becomes unreliable. Each virtual thread has its own isolated local state.

Enter *ScopedValue*.

From ThreadLocal to ScopedValue

Old approach:

```
1 public static final ThreadLocal<String> TL = new ThreadLocal<>();
2
3 public static void main() throws InterruptedException {
4     TL.set("my data");
5     try (var scope = StructuredTaskScope.open()) {
6         System.out.println("T1: " + TL.get()); // "my data"
7         scope.fork(() -> System.out.println("T2: " + TL.get())); // null
8         scope.join();
9     }
10 }
```

The child thread prints **null** because the `ThreadLocal` value isn't inherited in the thread created by `scope.fork`.

New approach with ScopedValue:

```

1 public static final ScopedValue<String> SV = ScopedValue.newInstance();
2
3 public static void main() throws InterruptedException {
4     ScopedValue.where(SV, "my data").run(() -> {
5         try (var scope = StructuredTaskScope.open()) {
6             System.out.println(SV.get()); // "my data"
7             scope.fork(() -> System.out.println(SV.get())); // "my data"
8             scope.join();
9         } catch (InterruptedException e) {
10             throw new RuntimeException(e);
11         }
12     });
13 }

```

Now both threads print the same value.

`ScopedValue` binds data to a *scope*, not a thread, ensuring all child tasks share the same context. In our case we are using `run` to start our context, as we don't need the result of our lambda. We could also use `call` which will return the result of our lambda expression instead.

Why ScopedValue is Better

Inheritance: Child scopes automatically inherit values.

```

1 ScopedValue.where(Key1, "1").run(() ->
2     ScopedValue.where(Key2, "2").run(() ->
3         System.out.println(Key1.get() + Key2.get()) // prints "12"
4     )
5 );

```

Immutability: Scoped values can't be changed within a scope. You can override them only in nested scopes, eliminating accidental mutations.

```

1 ScopedValue.where(Key1, "1").run(() -> {
2     Key1.set("2"); // compile error
3     ScopedValue.where(Key1, "2").run(...); // valid
4 });

```

If you use one-thread-per-request, `ThreadLocal` still works. But when adopting virtual threads and structured task scopes, you should migrate to `ScopedValue` for predictable behavior.

Final Thoughts

Java 25 delivers huge steps forward in concurrent programming.

While *StructuredTaskScope* is still a preview feature, its redesign in this release shows the platform's commitment to clarity and correctness. The willingness to overhaul the API even after several iterations is a testament to the Java team's dedication to getting it right.

With *StructuredTaskScope* and *ScopedValue*, Java's approach to concurrency finally feels modern, intuitive, and safe. Once the API stabilizes, likely in Java 26, these tools will be indispensable for anyone building high-throughput applications on the JVM.

› [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



The Blockchain in Your Java Application

Blockchain

Maven

Web3j

Author: Julius Lauterbach

Julius Lauterbach is a Software Developer specializing primarily in Java backend development. Over the past few years, he has increasingly focused on blockchain development and has been working on integrating blockchain technologies into enterprise software solutions.



LinkedIn

Short Foreword

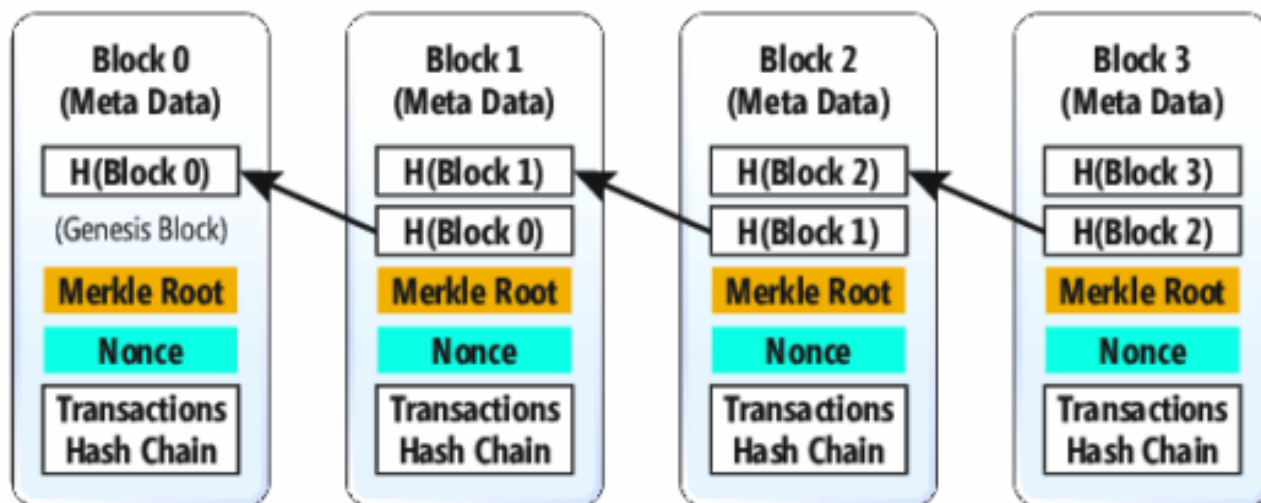
This article discusses the integration of blockchain mechanisms into Java applications. It is important to clarify that the focus is not on prices or individual cryptocurrencies overall. Throughout the article, I mention various blockchain networks, but I want to emphasize that this is **not** investment advice or promotional material for these networks or their cryptocurrencies. Instead, the article provides a technical explanation of how to incorporate blockchain technology into Java applications using Web3j.

Blockchain

The fundamental idea of distributed systems with shared data storage and collaborative management has existed for a very long time. The first white paper on Bitcoin – currently the most prominent and widely known blockchain network – was published on November 1, 2008 via a mailing list under the pseudonym Satoshi Nakamoto and can still [be viewed online today](#). A miner successfully mined the first block, known as the Genesis Block, on January 3, 2009. This event marked the official launch of the Bitcoin network.

Blockchain is a distributed ledger technology managed by a peer-to-peer network. Essentially, it is a decentralized or distributed database that allows all participants to access and verify data, ensuring a trustworthy and transparent exchange of information. This decentralized approach is in contrast to traditional, centralized systems, where control over the database resides with a single entity.

As the term “blockchain” suggests, it consists of the words “block” and “chain”. A block contains various data – most often represented through transactions in most blockchain networks. A block is created when a certain number of transactions, along with specific data such as difficulty level, block number, and block hash, are grouped together. The chain comes into play when each new block references the previous one via its hash, creating a secure, sequential link between blocks (see [following figure](#)).



Much like the internet facilitates the free exchange of information, blockchain enables the almost unrestricted exchange of digital values. However, unlike the internet, access to the blockchain network alone is not enough – it also requires a digital wallet, specifically a private key, to initiate new transactions. Using this private key, a participant can sign a message for the network. This digital signature verifies the legitimacy and origin of the message, allowing others to trust its authenticity.

Blockchain has fundamentally revolutionized and influenced the financial sector by enabling secure, transparent, and efficient transactions. Over time, its application has expanded to numerous other fields, including real estate, supply chain management, voting systems, and intellectual property protection.

Smart Contracts

A smart contract is a self-executing digital agreement that functions as a distinct entity within a blockchain network. It has its own blockchain address and can receive, manage, and send digital assets such as tokens. The execution of a smart contract's functions occurs autonomously across multiple nodes in the network, ensuring that all participants obtain the same, unaltered result. Like standard blockchain transactions, smart contracts are transparent, tamper-proof, and immutable.

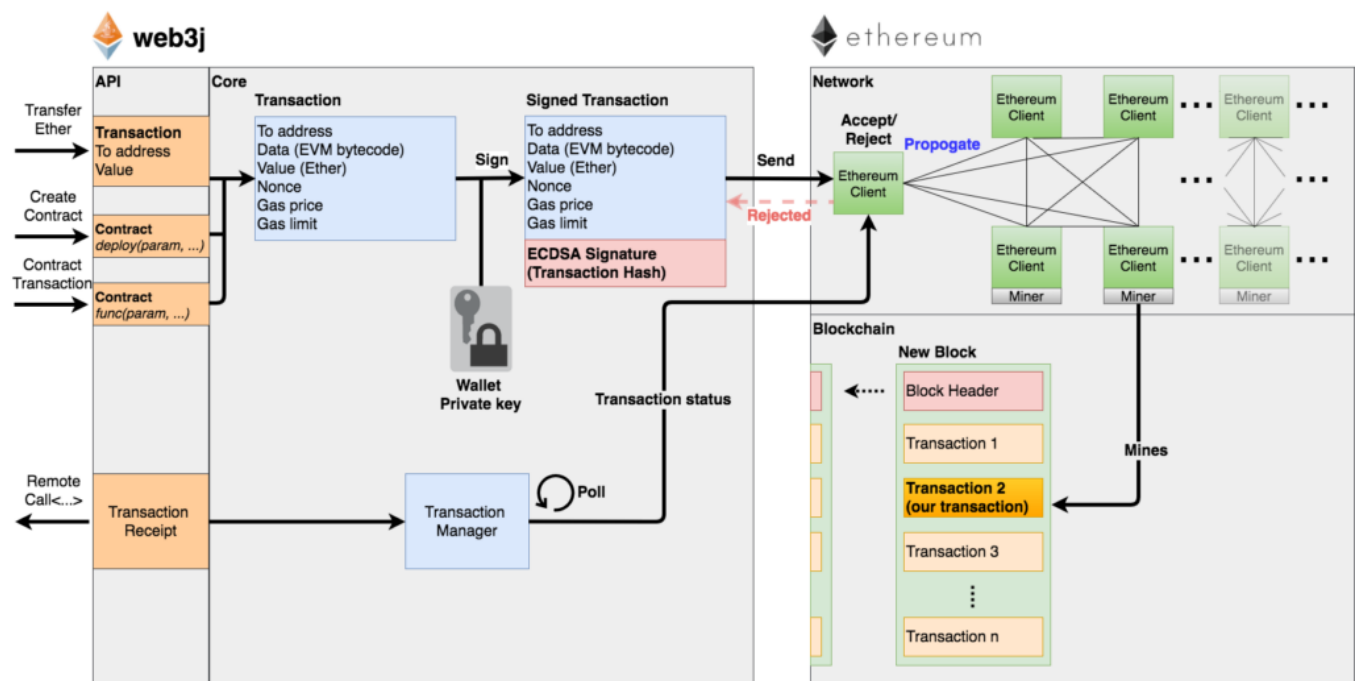
An illustrative example, which I encountered at the Linux Foundation Decentralized Trust initiative, is a vending machine. You deposit something – such as 2 euros or a cryptocurrency share – into the machine, which then dispenses a specific item in return, like a bottle of water or a financial instrument such as a share.

In summary, smart contracts offer an innovative, efficient, and secure method of automating contractual agreements by leveraging the advantages of blockchain technology.

```
1 pragma solidity 0.8.15;
2 contract HelloWorld {
3     function getHelloWorld()
4         public pure returns (string memory) { return "Hello World"; }
5 }
```

Web3j Library

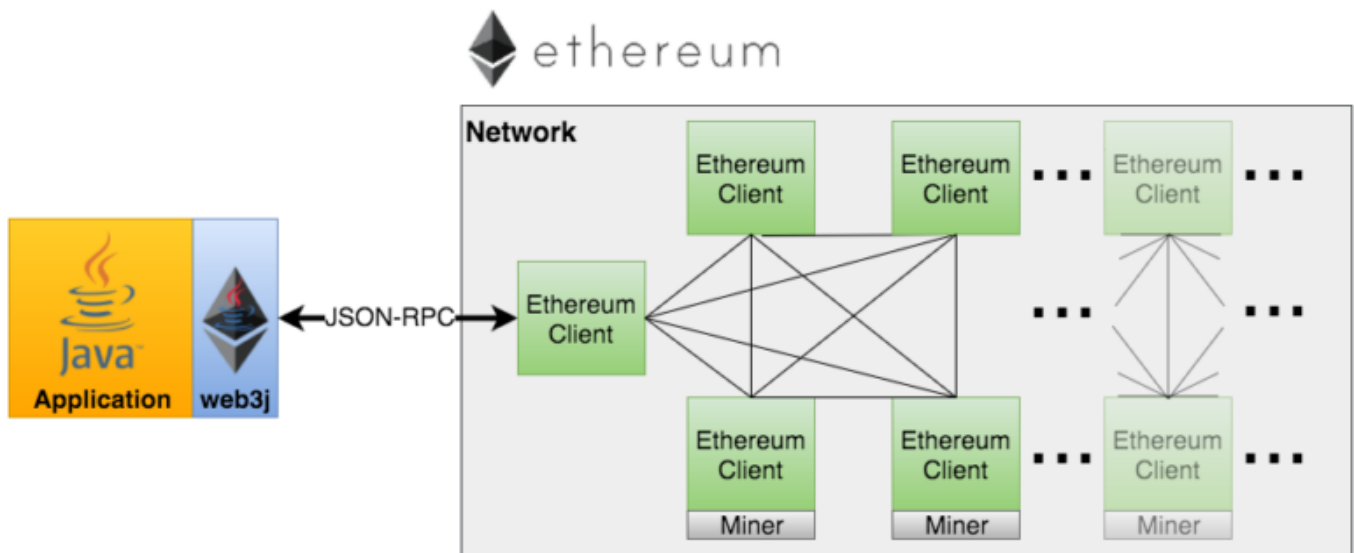
A transaction in the Ethereum network, for example, can appear very complex at first glance. There are the possibilities of a transfer of ether (the currency within Ethereum), a smart contract deployment or a smart contract function call. All of these processes are transactions within the blockchain. The [following illustration](#) shows such a schematic representation of a transaction.



However, when we consider a standard HTTPS call, we encounter different levels and a process involving the exchange of certificates during the SSL handshake. At its core, this idea is quite similar, with only a different implementation.

After compiling a transaction, the sender signs it using their private key. To ensure security, the application should generate this signature **locally**, so that private keys are not transmitted and remain exclusively in the possession of the user. Without using a dedicated library, you must implement the entire process of creating and signing the transaction manually in your Java application.

The Web3j library simplifies this complexity and provides an abstraction. The following figure visualizes this abstraction and illustrates the simplification.



The developer can now focus on building the actual Java application and its use cases. A detailed understanding of smart contracts and transaction execution is not strictly necessary at this stage. The following section also provide a Java example to illustrate this.

Integration in Java with Maven

To include Web3j in your Java project, add the corresponding [Maven repository](#) as a dependency in your pom.xml.

```
1 <dependency>
2   <groupId>org.web3j</groupId>
3   <artifactId>core</artifactId>
4   <version>${web3j-core.version}</version>
5 </dependency>
```

For better integration, include the Web3j Maven plugin as a build plugin in your project's pom.xml

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.web3j</groupId>
5       <artifactId>web3j-maven-plugin</artifactId>
6       <version>${web3j-maven-plugin.version}</version>
7       <configuration>
8         <soliditySourceFiles/>
9       </configuration>
10    </plugin>
11  </plugins>
12 </build>
```

For a detailed description of the parameterization, refer to the [official Web3j Docs page](#). Using this setup, you can generate Java wrapper classes from Smart Contracts. The command to perform the generation is as follows:

```
1 mvn web3j:generate-sources
```

Here is an example of a Java wrapper class generated for the Solidity Smart Contract described above:

```
1 public class HelloWorld extends Contract {
2
3   public RemoteFunctionCall<String> getHelloWorld() {...}
4
5   public static HelloWorld load(...) {...}
6   public static RemoteCall<HelloWorld> deploy(...) {...}
7 }
```


These generated classes always contain the “load” and “deploy” functions and all possible external (*public* or *external*) function calls. The **load** function loads the structure of a smart contract and stores its address. The **deploy** function packages and publishes a new contract on the blockchain – similar to a constructor or an HTTP POST request.

```
1 \<contract name>.\<function name>(\[param1, ..., paramN]).\[send() | sendAsync()];
```

Building on our example from the Smart Contract section, a typical scenario for using the generated Java code is as follows:

```
1 Web3j web3j = Web3j.build(new HttpService("someAddress.de"));
2 Credentials credentials = WalletUtils.loadCredentials("password", keyFile);
3 DefaultGasProvider gasProvider = new DefaultGasProvider();
4
5 HelloWorld helloWorld = HelloWorld.deploy(web3j, credentials, gasProvider).send();
6 helloWorld.getHelloWorld().send();
7
8 > "Hello World"
```

The above code also manages the connection to a blockchain node, handles credentials (address, public key & private key) for signing transactions and sets the transaction fees (GasFee). The “HelloWorld” smart contract is then called up and deployed as a Java object. After deployment, the “getHelloWorld” call follows. This function call returns a string that can be further used and utilized in the Java application.

Difficulties of Integration

That’s how easy it is to integrate the blockchain in Java with Web3j – well, almost. Of course we still need a suitable wallet file, connection to a blockchain node and tokens to be able to pay for our transactions.

Some blockchain systems mitigate these challenges by offering alternative cost models or by enabling other parties to cover transaction fees. Additionally, some service providers offer interfaces that simplify access to the network. But do we still need blockchain if we have to trust the service provider to carry out our transaction as expected? If we want full sovereignty over our transactions, we should create them ourselves. Only then can we be sure that the transactions arrive as intended and act in our best interests. However, there are already ideas to address these issues by verifying signatures within smart contracts and executing transactions only after successful verification.

Example Project

This is another [project from my personal GitHub account](#), linked here. In this project, I combined the integration of the Web3j library with automated Java code generation. It also includes a wallet with sufficient test network currency to experiment with. If there are any problems or difficulties, I’ll be happy to help.

Example from Practice

In practice, many companies brand themselves with terms related to blockchain, NFTs, smart contracts and similar concepts. However, we rarely see the actual underlying technologies behind these claims.

A positive example of this is [an open standard from DZ BANK AG](#), which asserts to have traded financial products via smart contracts in a legally binding manner. One such financial product is the so-called Smart Derivative Contract, for which further publications – including [technical descriptions](#) and initial reference implementations – are available.

Conclusion

Many Java developers are probably wondering at this point whether they could also work on blockchain projects.

Of course, there are hurdles when it comes to integrating blockchain with Java using Web3j. However, Web3j’s abstraction significantly simplifies this process, allowing Java developers to familiarize themselves very well with the topic.

A certain level of prior knowledge is not always mandatory—the specific tasks ultimately determine the learning curve. Nevertheless, interest and enthusiasm—as in many other areas—can make it easier to get started and build a solid understanding of the concepts.

Once familiar, developers can learn a great deal about blockchain technology and its various applications through relevant projects.

A minimal example can be implemented quickly and provides a clear introduction, as demonstrated in the Java example above.

With this in mind, don't be afraid, play around and explore the world of blockchain. Or to quote Simon Seiter in the closing words of a [conference panel discussion](#): “sail the ships [...] leave your island and discover the free world of public chains”

› **Back to the Table of Contents** ‹



No Deadlocks in MongoDB: Atomic Documents and Retries with Spring Data MongoDB

MongoDB

Spring

DeadlockPrevention

Author: Franck Pachot

Franck Pachot is a Developer Advocate at MongoDB. He has extensive experience in database consulting for development and operations teams and is passionate about improving the developer experience, data modeling, and performance troubleshooting. Franck holds several certifications, including Oracle Certified Master and MongoDB Certified Associate Data Modeler, and is a recognized expert in PostgreSQL and YugabyteDB. Amazon acknowledges him as an AWS Data Hero.



LinkedIn

Deadlocks are frequent in SQL databases, as normalization and two-phase locking (2PL) require locking multiple tables or rows during DML operations. Numerous use cases share them, and transactions might access data in a different order, resulting in circular waits and deadlocks. While databases can detect them, they can't prevent or resolve deadlocks, as the application determines what is run and in what order. The best the database can do is cancel one statement or transaction, and let the application rollback and retry, which can reduce performance as systems wait several seconds before getting notified. As Oracle Database makes clear in the error log: *"It is a deadlock due to user error in the design of an application or from issuing incorrect ad-hoc SQL."* Most RDBMS use pessimistic locking to avoid conflicts, but deadlocks are unavoidable, and locking can increase latency. For instance, by default, DB2 waits 10 seconds for deadlock detection, SQL Server waits 5 seconds, Oracle waits 3 seconds, and YugabyteDB and PostgreSQL wait 1 second. These delays impact latency, but deadlock prevention remains the application's responsibility. The only way to entirely prevent deadlocks is by avoiding locks altogether, as in MongoDB, which provides lock-free ACID transactions.

Avoiding Deadlocks in MongoDB

There are two principal strategies to avoid deadlocks and their performance penalties:

1. **Avoid normalization and use a single-statement update**, which the database can transparently restart in the event of a conflict.
2. **When multi-statement transactions are necessary, use an optimistic concurrency control** to detect conflicts immediately and raise errors without waiting.

MongoDB encourages the first approach by leveraging a rich document model that naturally aligns business transactions with a single database document. Any update to a single document is atomic. When the business logic spans multiple documents, MongoDB supports explicit multi-document transactions using optimistic concurrency control, which aborts conflicting transactions immediately, allowing your application to retry rather than waiting for timeouts. For an in-depth look, see my previous post: [Lock-Free Wait-on-Conflict and Fail-on-Conflict in MongoDB](#)

Adding retry logic in your application may sound complex, but you must always handle exceptions, and your development framework can make the retry declarative and straightforward. With [Spring Retry](#), you can define retry policies using `@Retryable` annotation alongside the `@Transactional` annotation.

Handling Write Conflicts in MongoDB with Spring Data

The `Account` class represents a bank account document, identified by `owner` and holding balances for sub-accounts (e.g., checking and savings). I've declared it with Spring Data MongoDB annotations in `src/main/java/com/example/demo/Account.java` as:

```
1 package com.example.demo;
2 //SNIPP imports
3 @Document(collection = "account")
4 public class Account {
5     @Id
6     public String id;
7     public String owner;
8     public Map<String, Double> balances = new HashMap<>();
9
10    public Account() {}
11
12    public Account(String owner, Map<String, Double> balances) {
13        this.owner = owner;
14        this.balances = balances;
15    }
16
17    @Override
18    public String toString() {
19        return String.format("%s: %s", owner, balances);
20    }
21 }
```

Here's an example of documents:

```

1  [
2    {
3      _id: ObjectId('688c7b458a3d8a4f4fba3b5a'),
4      owner: 'alex',
5      balances: { checking: 100, savings: 50 }
6    },
7    {
8      _id: ObjectId('688c7b458a3d8a4f4fba3b5b'),
9      owner: 'kim',
10     balances: { checking: 30, savings: 60 }
11   }
12 ]

```

This document model allows fund transfers between sub-accounts for the same user to be performed as atomic updates to a single document. No transaction is needed, and no deadlock is possible.

Spring Data MongoDB's repository abstraction makes data access both simple and powerful. By extending `MongoRepository` in your repository, Spring Data provides ready-to-use CRUD operations and parses custom finder methods directly from their names, eliminating the need for manual query code. Here is my `src/main/java/com/example/demo/AccountRepository.java`:

```

1  package com.example.demo;
2  //SNIPP imports
3  public interface AccountRepository extends MongoRepository<Account, String> {
4      Optional<Account> findByOwner(String owner);
5  }

```

I compile it with `mvn compile` and the following POM, which configures a Spring Boot 3.3.0 app with MongoDB support, Spring Retry, Java 17, and standard build plugin:

```

1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
2      XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
3      maven.apache.org/xsd/maven-4.0.0.xsd">
4      <modelVersion>4.0.0</modelVersion>
5      <groupId>org.example</groupId>
6      <artifactId>spring-mongo-transfer-demo</artifactId>
7      <version>1.0-SNAPSHOT</version>
8      <properties>
9          <java.version>17</java.version>
10         <spring-boot.version>3.3.0</spring-boot.version>
11     </properties>
12     <dependencyManagement>
13         <dependencies>
14             <dependency>
15                 <groupId>org.springframework.boot</groupId>
16                 <artifactId>spring-boot-dependencies</artifactId>
17                 <version>${spring-boot.version}</version>
18                 <type>pom</type>
19                 <scope>import</scope>
20             </dependency>
21         </dependencies>
22     </dependencyManagement>
23     <dependencies>
24         <dependency>
25             <groupId>org.springframework.boot</groupId>
26             <artifactId>spring-boot-starter</artifactId>
27         </dependency>
28         <dependency>
29             <groupId>org.springframework.boot</groupId>
30             <artifactId>spring-boot-starter-data-mongodb</artifactId>
31         </dependency>

```

```

32     <groupId>org.springframework.retry</groupId>
33     <artifactId>spring-retry</artifactId>
34 </dependency>
35 <dependency>
36     <groupId>org.springframework.boot</groupId>
37     <artifactId>spring-boot-starter-aop</artifactId>
38 </dependency>
39 </dependencies>
40 <build>
41     <plugins>
42         <plugin>
43             <groupId>org.springframework.boot</groupId>
44             <artifactId>spring-boot-maven-plugin</artifactId>
45         </plugin>
46         <plugin>
47             <groupId>org.apache.maven.plugins</groupId>
48             <artifactId>maven-compiler-plugin</artifactId>
49             <version>3.10.1</version>
50             <configuration>
51                 <source>${java.version}</source>
52                 <target>${java.version}</target>
53             </configuration>
54         </plugin>
55     </plugins>
56 </build>
57 </project>

```

Single-Document Operations: No Retry Loop Needed

With MongoDB, atomicity at the single-document level guarantees safe concurrent updates, even under high load. The database's storage engine detects conflicts and transparently retries the operation.

Here's an atomic transfer using Spring Data's lower-level `MongoTemplate` that I declared in `src/main/java/com/example/demo/AccountService.java`:

```

1  package com.example.demo;
2  //SNIPP imports
3  @Service
4  public class AccountService {
5      private final MongoTemplate mongoTemplate;
6
7      AccountService(MongoTemplate mongoTemplate) {
8          this.mongoTemplate = mongoTemplate;
9      }
10     public void transferBetweenSubAccounts(String owner, String from, String to, double
        amount) {
11         Query query = new Query(where("owner").is(owner).and("balances." + from).gte(amount))
12         ;
13         Update update = new Update()
14             .inc("balances." + from, -amount)
15             .inc("balances." + to, amount);
16
17         var res = mongoTemplate.updateFirst(query, update, Account.class);
18         if (res.getModifiedCount() == 0) {
19             throw new RuntimeException("Insufficient funds in " + from + " or account not found
20             ");
21         }
22     }
23 }

```

The query ensures sufficient balance before updating. The entire operation is handled in a single atomic update command in MongoDB. No transaction or application-level retry logic is needed here, and MongoDB guarantees ACID properties.

To test it, I can call it from my Spring Boot application's entry point in `src/main/java/com/example/demo/DeadlockDemoApplication.java`:

```

1  package com.example.demo;
2  //SNIPP imports
3  @SpringBootApplication
4  @EnableRetry
5  public class DeadlockDemoApplication implements CommandLineRunner {
6
7      @Autowired
8      private AccountService accountService;
9
10     @Autowired
11     private AccountRepository repo;
12
13     public static void main(String[] args) {
14         SpringApplication.run(DeadlockDemoApplication.class, args);
15     }
16
17     @Override
18     public void run(String... args) {
19         // Demo: clean up and initialize data
20         repo.deleteAll();
21
22         Account alex = new Account("alex", Map.of("checking", 100.0, "savings", 50.0));
23         Account kim = new Account("kim", Map.of("checking", 30.0, "savings", 60.0));
24         repo.save(alex);
25         repo.save(kim);
26
27         System.out.println("Before:");
28         repo.findAll().forEach(System.out::println);
29
30         System.out.println("Transferring 40 from alex.checking to alex.savings...");
31         accountService.transferBetweenSubAccounts("alex", "checking", "savings", 40);
32         repo.findAll().forEach(System.out::println);
33     }
34 }

```

I run it with `mvn spring-boot:run` and it shows the balances before and after:

```

1
2  .
3  ( ( ) \___| ' _| ' _| ' _ \ / _ ' \ \ \ \
4  \ / _ ___| |_) | | | | | | ( _ | | ) ) ) )
5  ' |___| . _| | | _| | \_, | / / / /
6  =====|_|=====|___/_/_/_/_/
7
8  :: Spring Boot ::      (v3.3.0)
9
10 Before:
11
12 alex: {checking=100.0, savings=50.0}
13 kim: {checking=30.0, savings=60.0}
14
15 Transferring 40 from alex.checking to alex.savings...
16
17 alex: {checking=60.0, savings=90.0}
18 kim: {checking=30.0, savings=60.0}

```

For such operations, debug logs show a single `update` without transaction markers:

```

1 $ mvn spring-boot:run -Dspring-boot.run.arguments="--logging.level.org.mongodb.driver.
  protocol.command=DEBUG"
2 ...
3 Single-document transfer (alex moves 40 from checking to savings):
4
5 2025-08-01T08:28:35.969Z DEBUG ... Command "update" started ... Command: {"update": "
  account", ... "owner": "alex", ...}
6 2025-08-01T08:28:35.982Z DEBUG ... Command "update" succeeded ... Command reply: {"n": 1,
  "nModified": 1, "ok": 1.0, ...}
7 2025-08-01T08:28:35.984Z DEBUG ... Command "find" started ... Command: {"find": "account"
  , ...}
8 ...
9 alex: {checking=60.0, savings=90.0}
10 kim: {checking=30.0, savings=60.0}

```

MongoDB is free for these operations because the concurrency control is at the document level. A concurrent transaction that starts to update a sub-account will detect the conflict and restart transparently. The application experiences only a few milliseconds of latency, with no errors.

Multi-Document Transactions: Handle Write Conflicts with Retries

For operations that require coordination across documents, such as transferring funds between different owners in our example, it is essential to use a MongoDB multi-document transaction. If two transactions conflict, MongoDB will abort one of them and report a `WriteConflict` error to the application, treating it as a transient error that can be retried. Here's an example, adding a method in `src/main/java/com/example/demo/AccountService.java`:

```

1 @Autowired
2 private AccountRepository repo;
3
4 @Transactional
5 @Retryable(
6     value = {
7         org.springframework.dao.TransientDataAccessResourceException.class,
8         org.springframework.data.mongodb.UncategorizedMongoDbException.class
9     },
10    maxAttempts = 15,
11    backoff = @Backoff(delay = 20, multiplier = 2, random = true)
12 )
13 public void transferBetweenAccounts(String fromOwner, String toOwner, double amount) {
14     Account from = repo.findByOwner(fromOwner).orElseThrow();
15     Account to = repo.findByOwner(toOwner).orElseThrow();
16
17     double fromBal = from.balances.getOrDefault("checking", 0.0);
18     if (fromBal < amount) throw new RuntimeException("Insufficient funds for transfer");
19     from.balances.put("checking", fromBal - amount);
20     to.balances.put("checking", to.balances.getOrDefault("checking", 0.0) + amount);
21
22     repo.save(from);
23     repo.save(to);
24 }

```

`@Transactional` participates in MongoDB's ACID transaction system (requires a configured `MongoTransactionManager`). `@Retryable` declares my retry policy, here with exponential backoff. I include both **`TransientDataAccessResourceException`** and **`UncategorizedMongoDbException`** in the value list until all Spring Data versions consistently map write conflicts to the transient type.

I run it by adding the following in **src/main/java/com/example/demo/DeadlockDemoApplication.java**:

```
1 System.out.println("\nMulti-document transfer (40 from alex.checking to kim.checking,
  with retry):");
2 accountService.transferBetweenAccounts("alex", "kim", 40);
3 repo.findAll().forEach(System.out::println);
```

When run, it adds the following to the output:

```
1 Multi-document transfer (40 from alex.checking to kim.checking, with retry):
2 ...
3 alex: {savings=90.0, checking=20.0}
4 kim: {savings=60.0, checking=70.0}
5 ...
6 Multi-document transfer (40 from alex.checking to kim.checking, with retry):
7 2025-08-01T08:31:11.663Z DEBUG ... Command "find" started ... Command: {"find": "account"
  , ... "owner": "alex", ... "startTransaction": true, ...}
8 2025-08-01T08:31:11.675Z DEBUG ... Command "find" started ... Command: {"find": "account"
  , ... "owner": "kim", ...}
9 2025-08-01T08:31:11.689Z DEBUG ... Command "update" started ... Command: {"update": "
  account", ... "owner": "alex", ...}
10 2025-08-01T08:31:11.701Z DEBUG ... Command "update" started ... Command: {"update": "
  account", ... "owner": "kim", ...}
11 2025-08-01T08:31:11.717Z DEBUG ... Command "commitTransaction" started ... Command: {"
  commitTransaction": 1, ...}
12 alex: {checking=20.0, savings=90.0} kim: {checking=70.0, savings=60.0}
```

The presence of `commitTransaction` shows that an explicit transaction was used to run this. If at the same time another instance of the application did update the same documents in a different order, the write conflict would be detected:

```
1 2025-08-01T11:00:57.555Z DEBUG ... Command "update" failed ... Command: {"update": "
  account", ...} com.mongodb.MongoCommandException: Command failed with error 112 (
  WriteConflict): 'Caused by :: Write conflict during plan execution and yielding is
  disabled. :: Please retry your operation or multi-document transaction.' ... "
  errorLabels": ["TransientTransactionError"], "code": 112, "codeName": "WriteConflict"
  , ...}
2 2025-08-01T11:00:57.564Z DEBUG ... Command "abortTransaction" started ... Command: {"
  abortTransaction": 1, ...}
3 2025-08-01T11:00:57.576Z DEBUG ... Command "abortTransaction" failed ... Command failed
  with error 251 (NoSuchTransaction): com.mongodb.MongoCommandException: Command failed
  with error 251 (NoSuchTransaction): 'Transaction with { txnNumber: 4 } has been
  aborted.'
```

Suppose the conflict stays, which is improbable when the transaction does not span user interaction, but it can be simulated with a manual transaction. In that case, the sequence of aborted updates is retried until the conflict is resolved, the maximum number of retries is reached, or the other transaction times out.

The settings for exponential backoff retries are dependent on your specific application and infrastructure. The primary objective is to minimize the number of retries while allowing concurrent transactions to complete without causing significant latency.

Troubleshooting

To build upon the previous code, I have addressed a few potential pitfalls.

Verifying Transactional and Retryable Support

It is essential to ensure that transactions are used, as a race condition could otherwise corrupt data. While developing this demo, I've added:

```
1 System.out.println("Is transaction active? " + org.springframework.transaction.support.
  TransactionSynchronizationManager.isActualTransactionActive());
```

I've also printed the retry attempts:

```
1 if (RetrySynchronizationManager.getContext() != null
2   && RetrySynchronizationManager.getContext().getRetryCount() > 0) {
3   System.out.printf("RETRY #%d at %s%n",
4     RetrySynchronizationManager.getContext().getRetryCount(),
5     java.time.LocalDateTime.now());
6 }
```

Enabling Transaction Support in Spring Data MongoDB

Initially, no transaction was started, and I had to enable the MongoDB transaction manager in the application context, adding this in `src/main/java/com/example/demo/MongoConfig.java`:

```
1 package com.example.demo;
2 //SNIPP imports
3 @Configuration
4 public class MongoConfig {
5     @Bean
6     MongoTransactionManager transactionManager(MongoDatabaseFactory dbFactory) {
7         return new MongoTransactionManager(dbFactory);
8     }
9 }
```

Then, my code displayed the following when I was in an explicit transaction:

```
1
2 Before:
3
4 alex: {checking=100.0, savings=50.0}
5 kim: {checking=30.0, savings=60.0}
6
7 Transferring 40 from alex.checking to alex.savings...
8
9 Is transaction active? false
10
11 alex: {checking=60.0, savings=90.0}
12 kim: {checking=30.0, savings=60.0}
13
14 Multi-document transfer (40 from alex.checking to kim.checking, with retry):
15
16 Is transaction active? true
17
18 alex: {checking=20.0, savings=90.0}
19 kim: {checking=70.0, savings=60.0}
```

Printing this info was good during testing, but I recommend adding such an assertion to raise an error if there's no transaction after the first statement, because the business logic requires an ACID transaction:

```
1 import org.springframework.transaction.support.TransactionSynchronizationManager;
2
3 if (!TransactionSynchronizationManager.isActualTransactionActive()) {
4     throw new IllegalStateException("No transaction active! ACID guarantee not present.");
5 }
```

This is a safeguard to avoid inconsistencies during a race condition, in case the transaction manager was not correctly configured.

Enabling Retry Support in Spring Data

The retryable annotation must be enabled. I added the following to `src/main/java/com/example/demo/DeadlockDemoApplication.java`:

```
1 import org.springframework.retry.annotation.EnableRetry;  
2 @EnableRetry
```

Catching the Right Exception for the Retry Logic

When building this demo, I initially retried only on `TransientDataAccessResourceException`, but the traces show that Spring is throwing `UncategorizedMongoDbException` for this error, because Spring Data MongoDB has no mapping for `TransientTransactionError`. I used the following logging flags when troubleshooting:

```
1 mvn spring-boot:run -Dspring-boot.run.arguments="  
2 --logging.level.org.springframework.data.mongodb.core.MongoTemplate=DEBUG  
3 --logging.level.org.mongodb.driver.protocol.command=DEBUG \  
4 --logging.level.org.springframework.data.mongodb.MongoTransactionManager=DEBUG  
5 "
```

Exponential Backoff

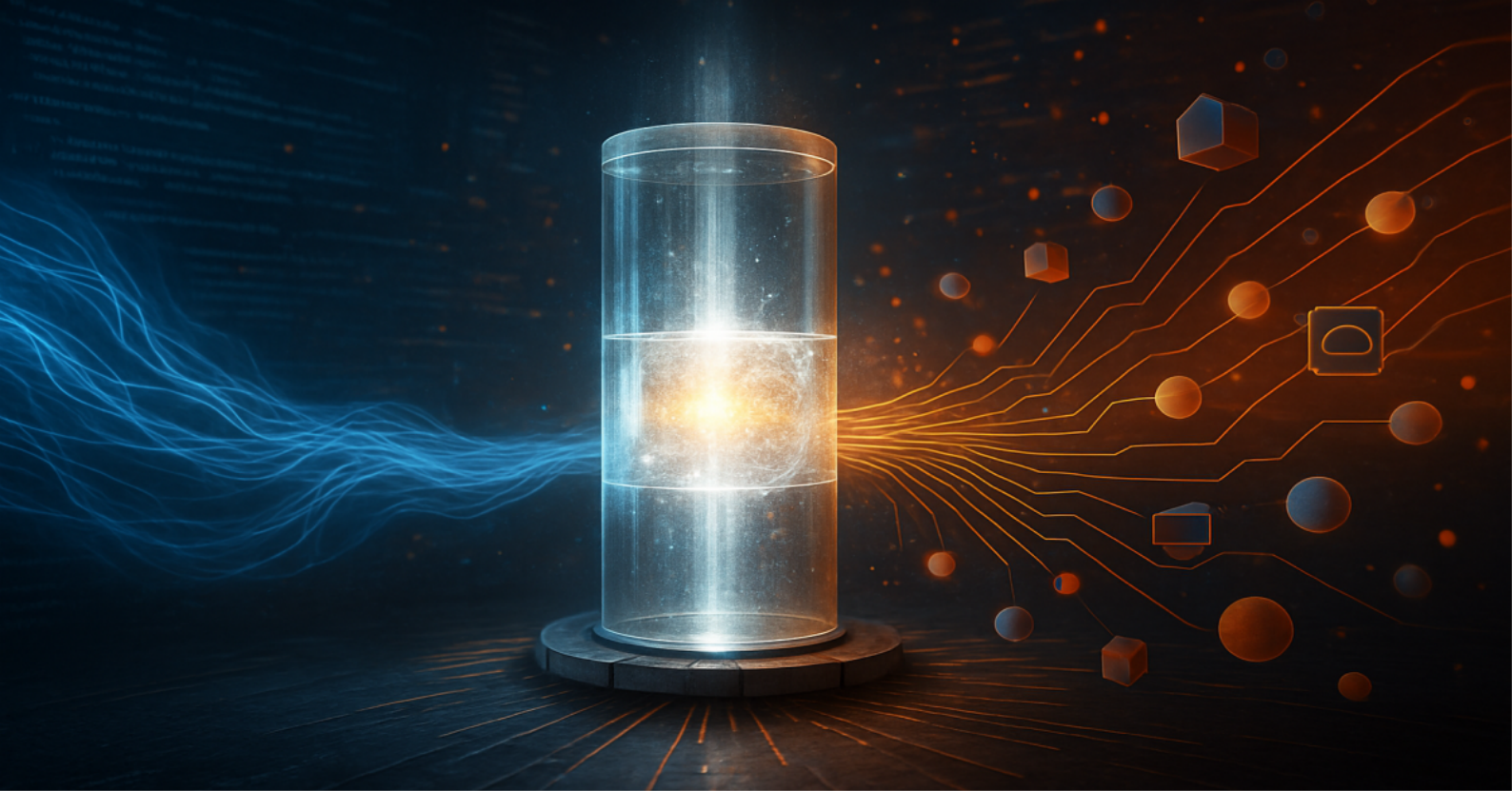
It is vital to implement a waiting period to prevent overwhelming the database during contention on a hotspot. Failing to do so has led to the myth that transactions are inherently slow, as discussed in my previous post [Tx Performance: Retry with Backoff](#). An exponential backoff minimizes retries and increases latency only during high contention, allowing the hotspot to cool down instead of becoming overwhelmed by repeated attempts. An additional randomness, or jitter, prevents many clients from retrying at the exact same moments, which could otherwise result in recurring spikes of contention.

In Summary

MongoDB does not experience deadlocks due to its lock-free concurrency control, which is fully ACID compliant. The WiredTiger storage engine employs optimistic concurrency control (OCC). Still, MongoDB's query layer presents it as wait-on-conflict for single-document updates, allowing these updates to be transparently retried by the query layer. When application-initiated explicit transactions conflict, the application must manage retries, as the database lacks context on other operations the application may have performed. With explicit transactions, MongoDB adopts a fail-on-conflict strategy to provide predictable latency. This differs from traditional RDBMS databases, which often lock and wait, but still require application-level logic to handle deadlocks.

Spring Data MongoDB offers a robust framework for implementing ACID transactions, streamlining retryable exception handling without added complexity. Key considerations include ensuring a transaction is initiated when needed, as the transactional annotation may be ignored if no transaction manager is set up. Additionally, ensure that the retry logic captures the correct exceptions, as not all transient errors are consistently mapped across all Spring Data MongoDB versions. Although MongoDB's lock-free concurrency control proactively prevents deadlocks, incorporating retry logic remains essential when using transactions. This typically involves a loop that catches temporary errors and includes a brief wait before retrying, ensuring quick retries without overwhelming the database.

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



BoxLang: A Dynamic Renaissance – From Zero to Release in 21 Months

JavaEcosystem

ServerlessRuntime

BoxLang

Author: Luis Majano

Luis Majano is a Computer Engineer, published author, and the founder and CEO of Ortus Solutions, Corp, a global consulting firm specializing in professional open-source development, web application architecture, and modern engineering tooling. He is the lead language engineer and creator of BoxLang, a dynamic, modern language for the JVM designed to power serverless, cloud-native, and cross-platform applications. Luis is also the creator of several widely adopted open-source projects, including ColdBox HMVC, ContentBox Modular CMS, Test-Box BDD, and the CommandBox CLI, contributing to a portfolio of over 300 open-source tools and libraries. Since 2000, Luis has been deeply involved in designing and building software architectures and developer platforms. He is passionate about empowering developers through mentorship, education, and open-source innovation, promoting sustainable software practices. He is a frequent international speaker and community advocate, regularly sharing his insights at www.luismajano.com.



LinkedIn

An 18-Year Drought

In the world of programming languages, 18 years is more than an eternity—it’s a generational shift. Yet that’s precisely how long the industry waited for a truly new, pure dynamic language built specifically for the JVM. Since Clojure’s debut in 2007, innovation in dynamic languages on the Java platform had effectively stalled.

Python stabilized. Ruby reached maturity. JavaScript conquered the world. Python rose again—but on the JVM, the world’s most proven and resilient runtime, dynamic language innovation had long gone quiet. The industry had largely decided that static typing was the future, leaving dynamic languages as relics to be maintained, not advanced.

Kotlin reignited that spark—a welcome breath of fresh air on the JVM—but its focus remained mostly on static constructs and semantics.

A small team at Ortus Solutions disagreed. They wanted to end the 18-year drought. They wanted to prove that dynamic languages could be fresh, modern, and blazingly fast. And they wanted to do it in 24 months, with a mostly part-time team, and without a single dollar of venture capital.

They did it in 21 months.

This is the story of **BoxLang**—the first pure dynamic language built specifically for the JVM in 18 years—created by betting everything on professional open source, AI-first development, and nearly two decades of hard-won experience.

Part I: The Foundation

17 Years of Professional Open Source

Luis Majano was born in El Salvador and studied computer engineering at Florida International University in Miami. In 2006, he created **ColdBox**—the first conventions-based MVC framework for ColdFusion. Nineteen years later, it remains strong.

That same year, Majano founded **Ortus Solutions** on a simple principle: *professional open source works*.

Not “open source as marketing” or “open source until acquisition.” Real, sustainable, professional open source—built on biblical principles of stewardship, excellence, **documentation**, and service.

For 17 years, Ortus built frameworks, libraries, and tools while providing professional services. The company maintained a portfolio of over 350 open-source libraries across the Java, JavaScript, and ColdFusion ecosystems. From a Miami startup, Ortus has grown into a company of over 40 professionals, expanding its operations into El Salvador—Majano’s birthplace—and into Europe since 2022, where he now leads from Málaga, Spain.

Those 350+ libraries represented nearly two decades of learning what developers truly need, what patterns work in production, and which architectural decisions pay dividends years later. The company was about to bet everything on that accumulated wisdom.

The Crisis of Possibility

By 2023, Ortus faced a crisis. Not a financial one—the business was thriving. This was a **crisis of possibility**.

The team wanted to build CLI tools. They wanted to create desktop applications, deploy them to embedded systems, and utilize serverless solutions across multiple cloud providers. They wanted their 350+ libraries to run on Android, in the browser via WebAssembly—anywhere developers might need them.

But they couldn’t. They were constrained by vendor decisions beyond their control. The runtime they depended on couldn’t go where they needed to go. They had no control over the full stack—from source code to bytecode. Innovation was stifled.

And then there was the broader landscape: dynamic languages had stagnated on the JVM. Since Clojure’s emergence in 2007—18 years ago—no truly new, pure dynamic language has been built specifically for the Java platform and today’s problems and runtimes. The industry had moved on, convinced that static typing was the only path forward. Dynamic languages were often treated as legacy—maintained, perhaps, but not innovated upon.

The Ortus team disagreed. They believed dynamic languages had a vibrant future on the JVM. But belief wasn’t enough.

The Stakes

The stakes were high. Ortus had 17 years of work, 350+ libraries, a thriving community, and an entire company's future on the line. They could stay constrained and accept the status quo, or they could bet everything on a radical idea: build a completely new dynamic language from scratch—the first in 18 years with just a handful of geo-distributed people, and it had to be done part-time, and in less than 24 months.

Part II: Project Jericho



The Audacious Plan

In early 2023, Ortus finally made the decision. They would design and build BoxLang—a modern, dynamic language for the JVM that could go anywhere, be more than just a language but also a framework, provide comprehensive documentation, world-class tooling, and revive innovation in the dynamic language space.

Here's what they wouldn't do: seek venture capital, take outside investment, pursue other projects, or compromise their professional open-source values for investor returns.

Here's what they would do: stop company growth, reinvest every dollar into Project Jericho, and give themselves exactly 24 months to ship a production-ready 1.0 release, or abandon the project entirely.

Twenty-four months. Make it work or lose everything. No runway extensions. No pivots to please investors. No safety net.

Breaking Down the Walls

The project needed a code name. They called it **Jericho**.

The symbolism was deliberate. Like the ancient city whose walls fell after faithful persistence, the team needed to break down the walls of their predecessors—the constraints, the limitations, the frozen innovation. They needed to create something entirely new from the rubble of those walls.

Ortus Solutions operates on biblical principles. The company believes in faithful stewardship of the resources entrusted to it, in servant leadership, and in building things that genuinely serve its community.

Project Jericho embodied those principles. This wasn't about ego or market domination. It was about faithful service to a developer community that deserved better options. It was about being good stewards of 17 years of accumulated knowledge by putting it to work in the service of something greater.

Part III: Architecture

August 2023: The Team Assembles in Málaga

Majano assembled the initial team of five in Málaga:

- **Brad Wood** – Senior Language Engineer & Architect
- **Jon Clausen** – Senior Language Engineer & Architect

- Two language engineers from a specialized language company will serve as consultants to help design the initial parsers.

August 2023 was the infancy of AI assistance in software development. The team had early access to Gemini and ChatGPT. They didn't know it then, but AI would become as fundamental to their development process as the JVM itself.

They started with a clean slate but 17 years of experience. They analyzed every major static and dynamic language. They studied successes and, more importantly, failures. They examined where Groovy, ColdFusion, and open-source CFML engines had succeeded for over 20 years and where they had been hindered.

The mission was clear: break down the walls. Build something new. End the drought.

The BLAST: A Revolutionary Architecture

The team made their most revolutionary decision during this retreat: the multi-parser architecture.

They created what they lovingly call the BLAST—the BoxLang Abstract Syntax Tree. It's a common intermediary layer that multiple source languages can compile to, which then compiles to JVM bytecode.

This solved their biggest challenge: how to build an entirely new language without abandoning 17 years of source code?

The answer: build a CFML parser that transpiles CFML to the BLAST, which then compiles to bytecode. Ortus's entire catalog of over 300 libraries, written in CFML, could run on BoxLang. They'd have an instant ecosystem from day one. MVC Frameworks, Testing Frameworks, DI, AOP, BDD — you name it, all libraries will be available.

But the BLAST's potential went far beyond CFML. They designed it to be the target for any language parser. Groovy to BLAST to bytecode. PHP to BLAST to bytecode. Even COBOL to BLAST to bytecode if someone wanted to write that parser.

The BLAST became their lingua franca—a way to bring multiple language communities into a shared, modern runtime while preserving their source code investments.

Multi-Runtime from Day One

BoxLang was designed to be **multi-runtime from the start**—one language, many deployment targets. The team knew they'd need a CLI runtime, a servlet runtime, and cloud-native runtimes, all sharing the same language core. This couldn't be bolted on later; it had to be part of the DNA.

Every runtime in BoxLang follows a **hierarchical model**, where contexts build themselves up like an onion—discovering new capabilities based on where they run.

This dynamic layering is made possible by the **JVM's mature, modular foundation**. Decades of classloading, threading, and memory innovation give BoxLang the freedom to adapt, extend, and deploy anywhere—without changing a single line of code.

Documentation from Day One

The BoxLang documentation began on day one. With a clear vision of the road ahead, the team structured the entire knowledge base early and automated processes to keep it **continuously updated and version-aware**—a living reflection of the platform itself. This is what made ColdBox the de facto enterprise MVC framework in CFML: its passion and dedication to excellence in documentation.

Event-Driven Core

From 17 years of building ColdBox, the team knew the power of event-driven architecture. ColdBox's interceptor pattern had been wildly successful—allowing developers to hook into framework lifecycle events and extend behavior without modifying core code.

They brought that philosophy into BoxLang at the language level. Class compilation fires events. Function invocation fires events. Developers and language extenders can hook into these events, implementing before and after AOP advices without any bytecode manipulation or complex wiring.

The language itself is extensible through clean, understandable event hooks.

Modularity Core

Building on the success of ColdBox’s hierarchical modular services, the same approach was envisioned for BoxLang. This would keep the core extremely lightweight and tiny, pushing all innovation to modular packages that could be built by the team or any member of the community. Everything had to be extensible via modules, including global functions, components, events, parsers, and more.

The Dynamic Object Framework

To achieve seamless Java interoperability, the team created the Dynamic Object framework—one of BoxLang’s key innovations.

The Dynamic Object framework makes any Java class interact transparently with BoxLang’s dynamic typing system. It provides:

- **Auto-casting:** Automatic, intelligent type conversion
- **Auto-boxing:** Seamless handling of primitives and objects
- **Type promotions:** Smart type upgrades when needed
- **Type contagion:** Dynamic type propagation through operations
- **Type inference:** Intelligent type detection without explicit declarations
- **Dynamic functions to specific Lambda Interfaces:** Intelligent Lambda Interface proxies coming from typeless functions.

All of this is built on JDK 21’s invoke dynamic capabilities, avoiding reflection overhead while maintaining dynamic flexibility.

The result? **Dynamic BoxLang classes can extend Java classes and implement Java interfaces.** This isn’t just calling Java code—this is true interoperability. BoxLang code can inherit from Java’s type hierarchy and participate in Java’s object model.

Java’s performance and ecosystem with dynamic flexibility. Developers get the best of both worlds without compromise.

AI-First Design

From that first week in August, the team utilized AI as a design partner. They held design sessions with Gemini and ChatGPT, discussing potential service architectures, evaluating approaches, and validating their patterns against the collective knowledge encoded in these models.

This wasn’t about having AI write code for them. It was about utilizing AI to validate thinking, identify potential issues early, and explore design spaces more efficiently than manual analysis allowed.

Part IV: The Journey

December 2023: Turing Complete

By December, just four months in, the team had achieved their first major milestone: a Turing-complete language. They had a basic parser, a functional runtime, and enough language constructs to write real programs. Was it fast? Heck no! But hey, that if statement was beautiful.

The Dynamic Object framework was working. They could call Java code from BoxLang. Basic type conversions were functioning. They had a foundation.

January 2024: The Team Expands

In January 2024, the consultancy company finalized its tenure; it was now up to Ortus. The foundation was solidly established, and it was time for the transition to the broader Ortus team.

The company added a significant portion of its engineering talent to Project Jericho:

Core Engineering Team:

- Jacob Beers: The master mind behind the BoxLang IDE based on VSCode and the ASM Maestro!
- Jorge Reyes (COO, operational leadership)
- Eric Peterson, Grant Copley, Javier Quintero, Jaime Ramirez, Davis Vega (Senior Engineers)
- Alex Ventura, Abilio Posada, Daniel Garcia (Senior Engineers)
- Michael Borne (Senior Engineer & ORM Specialist)
- Alan Quintana, Esme Acevedo, Dan Card (Senior Engineers)

Supporting Roles:

- Cristobal Escobar (Business Development and Marketing)
- Maria Jose Herrera (Marketing)
- Jonathan Lainez (Design Team)

Most continued working part-time on BoxLang while maintaining client responsibilities. This constraint turned out to be an advantage. When you only have limited hours, you ruthlessly prioritize. There's no time for gold-plating, no room for nice-to-haves that don't serve the mission. Client work kept the team grounded in real-world needs.

Jacob Beers continued leading IDE development, ensuring that BoxLang's tooling would be professional-grade from day -60.

Brad Wood immersed himself in ANTLR to attack all possible scenarios of CFML parsing and our new BoxLang vision parser.

Luis and Jon continued development of all the framework capabilities of the runtime.

February 2024: Full Java Interoperability

Two months later, Java interoperability was complete. Not basic—complete. Dynamic classes extending Java classes: working. Interface implementation from dynamic code: working. Auto-casting and type promotions: functional across the board.

This was a critical capability. Without full Java interoperability, BoxLang would have been isolated from the Java ecosystem. With it, they had access to decades of Java libraries and frameworks.

The Chicago Retreat: Innovation Under Pressure

In early 2024, during the Ortus team retreat in Chicago, the team sought to provide developers with an online platform to test BoxLang. They envisioned try.boxlang.io—a website where people could write code, execute it, evaluate the results, and experiment with the language without needing to install anything.

But how would they achieve it? The discussion began with uncertainty. How do you safely execute arbitrary user code? How do you scale it? How do you make it fast and responsive?

After just one hour of intense discussion, the AWS Lambda Runtime was born. By leveraging serverless architecture, they could provide isolated execution environments that scaled automatically and cost-effectively. Each code execution would run in its own Lambda function, providing security through isolation and scalability through AWS's infrastructure.

What started as a feature request became a complete runtime implementation. Today, the AWS Lambda Runtime not only powers try.boxlang.io, enabling developers worldwide to experiment with BoxLang instantly, but it also provides a production-ready deployment target for serverless applications.

Innovation born from necessity, delivered in an hour, serving two purposes. It was characteristic of how the team worked throughout the project.

March 2024: The Runtime Explosion

March was intense. The team was building multiple runtimes simultaneously:

- **CLI Runtime:** For command-line applications and scripting

- **MiniServer:** A lightweight, standalone web server
- **AWS Lambda Runtime:** Born from the Chicago retreat, powering both try.boxlang.io and production serverless deployments
- **CommandBox Servlet Runtime:** Integration with Ortus's Undertow-based engine
- **JSR223 Runtime:** For embedding BoxLang in Java applications
- **Desktop Runtime:** For building desktop applications
- **LSP Runtime:** The Language Server Protocol as a complete runtime

Jacob Beers was driving IDE development forward. The BoxLang LSP was taking shape as a runtime itself. The VSCode extension was becoming feature-complete.

The team also finalized the template language. The .bxm file format was ready. They had studied GSP, JSF, Blade, and CFML, taking the best ideas from each. The template language would have full access to the language, support components, and integrate seamlessly with BoxLang's module system.

April 2024: Documentation and Three Ecosystems

<https://boxlang.ortusbooks.com> went live. The custom Claude model, trained on Ortus standards and the ANTLR grammar, generated consistent, comprehensive documentation for every language feature.

The team also completed the CFML Compat Module. This was their third ecosystem strategy: BoxLang native, Java interoperability, and CFML compatibility. Developers could choose their path.

May 2024: First Beta at Into The Box

At the annual Into The Box conference, Ortus released the first public beta. The performance wasn't optimized yet—parsing was slow, execution had overhead. But it worked. The vision was a reality.

ColdBox, the 17-year-old framework, was running on BoxLang. Multiple runtimes were operational. The IDE was functional. The documentation was live. The CFML compatibility module was parsing real-world code. And try.boxlang.io was live, letting developers experiment online.

In just nine months, the team had gone from zero to a functional, multi-runtime language with professional tooling and three ecosystems.

The community response was electric. This was real. The walls were falling.

June-December 2024: AI-Amplified Refinement

The second half of 2024 was about refinement at scale. The team parsed client code continuously. Certification of all internal libraries and frameworks began. The commitment to release minor updates monthly started.

Ortus had made AI integration systematic:

Mandatory Code Reviews: GitHub Copilot code reviews are now compulsory for every pull request. Not optional, not nice-to-have—mandatory. Every piece of code got AI analysis for patterns, potential bugs, and optimization opportunities.

Integrated Development: VSCode Copilot integration became required for all developers. Real-time problem-solving, inline suggestions, and performance analysis as developers typed. AI assistance became as fundamental as the compiler.

Issue Management: For bug fixes and small updates, the team started assigning GitHub issues directly to Copilot. The AI would propose solutions, and developers would review, refine, and incorporate them. This freed engineers to focus on architecture and complex problems.

Documentation Engine: Mid-2024 brought the real game-changer with Claude. The team trained a custom Claude model on Ortus's coding standards—the Ortus style guides for both CFML and Java, as well as their architectural patterns and even their ANTLR grammar. This model kept [BoxLang.ortusbooks.com](https://boxlang.ortusbooks.com) synchronized with code changes, maintaining consistency across thousands of pages.

Without AI, the team would not have been able to achieve this in 21 months, despite having mostly part-time developers. AI didn't write BoxLang—the team's experience and expertise guided every major decision. However, AI amplified capabilities, enabling a small part-time team to achieve what would have required a much larger, full-time organization.

The full Ortus company, comprising over 40 people, was now engaged. While the core engineering team focused on language features, the entire organization contributed to testing, documentation, community support, client work, and infrastructure development.

February 2025: Tooling Maximized

Sixty days before the planned release, the tooling was ready. The IDE had all the planned features. The LSP was fully functional. Debugging capabilities were professional-grade.

The team had achieved what they set out to do: tooling first, not tooling later. Jacob Beers's leadership on IDE development ensured that BoxLang would have professional tooling from day one:

- Deep syntax highlighting via custom tmLanguage grammars
- Context-aware code completion
- Real-time error detection
- Type inference and intelligent hints
- Line-by-line debugging with thread inspection
- Dynamic variable modification during debugging
- Professional debugging capabilities that many established dynamic languages still lack

April 2025: The Walls Fall

Twenty-one months after starting in Málaga, Ortus launched BoxLang 1.0, followed by 1.1 due to initial bugs discovered on the same day.

Three months early. Three months under their 24-month deadline.

They had built what took other languages 30 years to achieve in 21 months. They had created the first new pure dynamic language for the JVM in 18 years. And they had done it without a single dollar of venture capital, with a mostly part-time team, while still serving their clients.

The shofar had sounded. The walls had fallen. Something new had emerged from the rubble.

Project Jericho had arrived!

Part V: What Was Built



Nine Runtimes: Deploy Anywhere

BoxLang launched with nine production-ready runtimes, each solving real-world deployment needs:

1. **LSP Runtime:** The Language Server Protocol as a complete runtime, powering IDE services
2. **Any OS (JVM):** Runs anywhere Java runs—Windows, macOS, Linux, BSD
3. **Docker:** Containerized deployments for CLI apps, MiniServer, and full servlet applications
4. **CommandBox:** Ortus’s powerful Undertow-based servlet engine, 17 years in development
5. **WAR Deployments:** Standard Java web archives for any servlet container
6. **Desktop:** Native desktop application development
7. **AWS Lambda:** Born from the Chicago retreat, powering both try.boxlang.io and production serverless applications
8. **JSR223:** Embed BoxLang in any Java application as a scripting layer
9. **MiniServer:** Lightweight standalone web server for microservices

Same language, nine different deployment targets. Write code once, deploy it anywhere—from embedded scripting to enterprise servers, from CLI tools to serverless functions, from desktop applications to containerized microservices.

This is “write once, run anywhere” for dynamic languages, finally realized in its fullest sense.

Three Ecosystems: Unprecedented Compatibility

BoxLang Native: Modern syntax, full OO capabilities with the Dynamic Object framework, the .bxm template language, event-driven architecture, and a growing module system. Everything fresh, modern, and fast—designed for today’s development challenges.

Java Ecosystem: Through 100% interoperability, BoxLang developers have instant access to the entire Java ecosystem. Maven Central, Spring Framework, Jakarta EE, Apache Commons—everything. The Dynamic Object framework ensures seamless interoperability—auto-casting and type promotions happen transparently.

CFML Compatibility: The CFML Compat Module provides a complete migration path for existing CFML applications. The CFML parser transpiles CFML source to the BLAST, which then compiles to bytecode.

All of Ortus’s libraries work across all three ecosystems. This is the largest initial ecosystem any new language has ever launched with—not because the team spent years building libraries, but because they built an architecture that could leverage existing code.

Why the JVM Was Essential

The Java Virtual Machine is 25+ years of continuous optimization. It has world-class garbage collection, sophisticated JIT compilation, and enterprise-grade performance characteristics. When you build on the JVM, you inherit decades of work by some of the world’s best virtual machine engineers.

Moreover, the JVM was the only mature platform for building a modern dynamic language with enterprise-grade performance. Its battle-tested production record, cross-platform nature, and massive ecosystem made it irreplaceable.

BoxLang was built on JDK 21, leveraging invoke dynamic for the Dynamic Object framework. By using invoke dynamic instead of reflection, BoxLang achieves dynamic typing performance that approaches that of statically-typed code—avoiding the performance penalties that many dynamic JVM languages suffer.

Part VI: The Formula for Success

What Made 21 Months Possible

Looking back, five factors made this possible:

17 Years of Experience: The team knew what worked because they’d built it before. They knew what to avoid because they’d hit those walls. BoxLang’s architecture reflects nearly two decades of framework development, library maintenance, and production debugging. When they designed the event system, they leveraged ColdBox’s proven interceptor pattern. When they built the module system, they avoided OSGi’s complexity based on real-world observations.

The Java Platform: The team inherited 25+ years of JVM optimization. They got enterprise performance without spending years building a runtime. They gained access to thousands of production-tested libraries. The JVM was the only mature platform that could support their ambitions for a high-performance dynamic language.

AI Integration: Systematic AI integration at every level—design, development, review, documentation—allowed a small team to achieve large-team output. Early access to Gemini and ChatGPT informed architectural decisions. Mandatory GitHub Copilot code reviews caught issues at scale. A custom-trained Claude model maintained documentation consistency. AI didn’t build BoxLang, but it amplified capabilities decisively.

Tooling First: By prioritizing developer experience and building professional tools before launch, the team ensured that developers would actually want to use BoxLang. Jacob Beers’s leadership on IDE development meant that 60 days before release, BoxLang had debugging capabilities and IDE features that some established dynamic languages still lack.

Financial Discipline: No VC funding meant no distractions, no pivots for investors, no pressure to compromise the vision. The 24-month deadline forced ruthless prioritization. Part-time developers focused on what mattered because they had to. Client work kept the team grounded in real-world needs.

Part VII: What’s Next

BoxLang 2.0 and Beyond

Ortus isn’t stopping. BoxLang 2.0 is already in development, with continued JVM optimization, enhanced language features, and an expanding standard library. The team is building a new module to support client and migration needs and focusing on being able to run their entire catalog natively in BoxLang.

The team is also building additional runtimes:

- **Electron-based desktop:** For web-technology-based desktop apps
- **JavaFX-based desktop:** For native desktop applications
- **Azure Web Functions:** Microsoft cloud integration
- **Google Cloud Functions:** GCP serverless support
- **Android Runtime:** Mobile app development
- **WebAssembly:** Browser-based execution

Every runtime they add expands the possibilities for BoxLang and what developers can build with it.

AI-First Development

Ortus launched the BoxLang AI Module in 2024, providing comprehensive AI capabilities with so much more coming soon:

- Encapsulation and abstractions for all major LLMs (OpenAI, Anthropic, Google, and more)
- Local LLM support included
- Memory management for conversational AI
- Chat interfaces and multi-turn dialogues
- Embedding support and vector storage
- RAG (Retrieval Augmented Generation) frameworks
- Agent construction tooling

The API design is fluent and dynamic—building AI agents with BoxLang is remarkably straightforward. An AI-first language with AI-first tooling, enabling developers to build AI applications with minimal friction.

The Complete Package

BoxLang isn't just a scripting language. It's a complete platform:

- Full scripting capabilities for any platform
- Complete OO with dynamic classes extending Java classes and implementing Java interfaces
- A modern template language influenced by the best ideas from GSP, JSF, Blade, and CFML
- The Dynamic Object framework enabling transparent Java interoperability
- Nine deployment runtimes covering everything from embedded scripting to enterprise servers
- Three complete ecosystems: native BoxLang, full Java access, and CFML compatibility
- Professional IDE and debugging tools from day one
- AI-first development tooling including a comprehensive AI module
- Full framework capabilities: events, scheduled tasks, box futures, security, PDF, Image manipulation, encryption, and so much more.

Ortus didn't build a toy language for enthusiasts. They built a professional platform to continue servicing their clients needs and moving the JVM forward.

Epilogue: The Shofar Sounds



The BoxLang logo—a shofar—represents more than just branding. It represents the signal that the battle has begun, that the walls are falling, that something new is emerging.

For 18 years, those walls stood. Dynamic language innovation on the JVM was frozen. Developers were told the future was static, that dynamic languages were legacy.

The shofar has sounded. The walls have fallen. The 18-year drought is over.

BoxLang is here, built on biblical principles of faithful stewardship and servant leadership. It's here because a company was willing to bet everything on serving its community well. It's here because a small team believed that dynamic languages deserved better, and they were willing to march around those walls until they fell.

From an hour-long discussion in Chicago that gave birth to a serverless runtime, to <https://try.boxlang.io> enabling developers worldwide to experiment instantly, to nine runtimes and three ecosystems launching in just 21 months—BoxLang represents what's possible when experience, platform, AI, and discipline align with a clear mission. All in a single JVM platform that unifies the entire world.

The dynamic renaissance has begun. And it's only getting started.

BoxLang is professionally maintained by Ortus Solutions. Documentation, downloads, and community resources are available at boxlang.io and boxlang.ortusbooks.com. The language is open source, with professional subscriptions, and welcomes community contributions.

Luis Majano leads BoxLang development from Málaga, Spain, where he directs Ortus Solutions' worldwide operations. The broader Ortus team continues to expand BoxLang's capabilities and support the growing community.

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



JAVAPRO
THE FREE MAGAZINE FOR THE JAVA COMMUNITY

30 YEARS OF JAVA
SPECIAL EDITION

JOON!
JAVASPEAKERS

30 YEARS OF JAVA: HOW THE LANGUAGE HAS EVOLVED
CONFERENCES DEDICATED TO JAVA PERFORMANCE

A JOURNEY THROUGH 30 YEARS OF AN EVOLUTION
BUILDING APPS & SERVICES IN JAVA
HOW TO GET STARTED WITH AN IDE

30 YEARS OF JAVA: 30 YEARS OF INNOVATION
PLAYING ENGINEER AND SEE FOR JAVA EVOLUTION

JAVAPRO

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



High Speed JPA

#ORMPitfalls

#JPA

#EntityGraph

Author: Julius Mischok

Julius Mischok is founder and CEO of Mischok GmbH. Together with his consulting team, he empowers project partners to implement complex software solutions.



LinkedIn

JPA sometimes carries the stigma of being “too slow for production.” Yet in countless projects, it has quietly powered reliable enterprise systems at scale. The truth is more nuanced: JPA itself is not the bottleneck — but the way we use it can turn convenience into catastrophe. In this article, we’ll explore why green tests can hide critical performance problems, and how a handful of practical techniques can transform JPA from a suspected culprit into a trusted high-speed partner.

The JPA Paradox

In the lab, everything seems perfect. Unit tests pass with flying colors, integration tests run smoothly, and even the first versions on dev or staging environments feel reassuringly fast. Yet the moment the application goes live with a real dataset, performance collapses. Queries that returned instantly on a few dozen rows suddenly take seconds or longer. What happened?

For many Java developers, this paradox feels all too familiar. JPA promises to remove the boilerplate of database access and let us focus on our domain model instead of SQL. And for the most part, they deliver: less code, cleaner APIs, green tests. But hidden behind this convenience lurks a set of performance pitfalls. Defaults that seem harmless in small test environments can trigger catastrophic slowdowns in production.

This article makes one claim: JPA is not slow. The framework itself is not the culprit. What slows down your application are subtle misunderstandings, default settings, and abstraction leaks. The good news is that once you recognize these patterns, you can prevent them — and keep JPA running at high speed.

A Brief Look Back

To understand why JPA behaves the way it does, it helps to revisit the history of object-relational mapping (ORM) in the Java world. In the 1990s, developers wrote SQL by hand, juggling `PreparedStatement` objects and manual mapping of rows into domain objects. The approach was powerful but tedious, and error-prone in larger systems.

By the early 2000s, frameworks like TopLink and Hibernate introduced a new paradigm: declare how your classes map to tables, and let the framework handle the SQL. This drastically reduced boilerplate and made persistence a solved problem for many teams. In 2004, the Java Persistence API (JPA) standardized this approach, and later Spring Data JPA made it trivially easy to integrate into enterprise applications. Today, Hibernate remains the most widely used JPA implementation.

But abstraction always comes at a price. Jeff Atwood famously described ORM as “the Vietnam of Computer Science” — extremely helpful in many situations, but dangerously misleading if treated as a silver bullet [Blogpost](#). The story of JPA reflects this perfectly: for 95% of use cases, it just works. For the remaining 5%, it can become a source of nightmares.

And this is where our journey begins: the infamous N+1 problem, lazy loading traps, and the hidden costs that appear only when the data grows.

The N+1 Problem: Why 1+N Would Be a Better Name

One of the most notorious pitfalls when working with JPA is the so-called N+1 queries problem. In practice, it might be more accurate to call it 1+N: one query to load the initial dataset, and then – in the worst case – one additional query per fetched row. It often sneaks in unnoticed, because at small scales it barely leaves a trace. On a test system with a few dozen rows, you may not even realize it’s happening. But once the dataset grows, the effect becomes devastating.

Let's take a simple example from our demo project [DemoCode](#). We have companies and employees, linked by a @Many-To-One relation in the Employee entity:

```

1  @Entity
2  @Table(name = "company")
3  public class Company {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      @Column(name="id")
7      private Long id;
8
9      @Column(name = "name")
10     private String name;
11
12     @OneToMany(mappedBy = "company")
13     private List<Office> offices;
14
15     @OneToMany(mappedBy = "company")
16     private List<Employee> employees;
17 }

```

```

1  @Entity
2  @Table(name="employee")
3  public class Employee implements WithId {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      @Column(name="id")
7      private Long id;
8
9      @Column(name="first_name")
10     private String firstname;
11
12     @Column(name="last_name")
13     private String lastname;
14
15     @ManyToOne
16     @JoinColumn(name = "company_id")
17     private Company company;
18
19     @Column(name = "age")
20     private Integer age;
21 }

```

For the showcase, we use a very basic Spring Data Rest Repository:

```

1  @Repository
2  public interface EmployeeRepository extends JpaRepository {
3      // ... some custom query methods ...
4  }

```

Running the following standard call seems harmless enough:

```

1  List<Employee> employees = employeeRepository.findAll();

```

You can see the whole test case in [NplusOneTest.java](#). It pretends to execute 1 query but in fact it will result in 4 queries even on the very small dataset in the test. With only a handful of employees, you will not notice performance issues. But if we inspect the SQL log, the problem becomes obvious:

```

1  select e1_0.id, e1_0.age, e1_0.company_id, e1_0.first_name, e1_0.last_name
2  from employee e1_0
3  select c1_0.id, c1_0.name from company c1_0 where c1_0.id=?
4  select c1_0.id, c1_0.name from company c1_0 where c1_0.id=?
5  select c1_0.id, c1_0.name from company c1_0 where c1_0.id=?

```

One query for the employees, and then one query per company reference. With a dataset of ten employees, that's just a

maximum of eleven queries — no big deal. But with 10,000 employees spread across 500 companies, we suddenly face 501 queries. One for all the employees and one per each individual company! If each query would take 10 ms, the request now burns more than five seconds. In production, that's unacceptable.

Fortunately, JPA gives us a straightforward way to prevent this. With `@EntityGraph`, we can tell the persistence provider to fetch related entities in one go:

```
1 @Repository
2 public interface EmployeeRepository extends JpaRepository<Employee, Long> {
3     @EntityGraph(attributePaths = { "company" })
4     List<Employee> readAllBy();
5 }
```

The effect is immediate. Instead of firing one query per company, Hibernate generates a single join, just as you would have done manually in former times:

```
1 select e1_0.id, e1_0.age, c1_0.id, c1_0.name, e1_0.first_name, e1_0.last_name
2 from employee e1_0
3 left join company c1_0 on c1_0.id = e1_0.company_id
```

Now, no matter how many employees or companies exist, the system retrieves them in one efficient query. On large datasets, this optimization makes the difference between a responsive application and an unusable one.

In a recent client project we were facing the issue that a simple synchronization job took a couple of seconds on test data but hours on production. Fortunately, everything was developed test-driven, so that we were able to introduce EntityGraphs at the correct places without destroying the business logic. Two hours of optimization sent the execution time down to about 10 minutes which was completely acceptable for the use case.

The lesson: the N+1 problem is not a flaw of JPA, but of unawareness. Once you recognize it, the fix is simple — but you need to look beyond green test cases and anticipate the scale of production data.

Lazy vs. Eager Loading: The Subtle Performance Trap

If the N+1 problem is the loud warning shot, then lazy and eager loading are its quieter accomplices. They sound simple: either fetch data when it's needed (lazy) or fetch it immediately (eager). In practice, the defaults JPA chooses can be surprising — and costly.

By specification, `@ManyToOne` associations default to EAGER, while `@OneToMany` associations default to LAZY. That means whenever you load an entity with a `@ManyToOne` relation, the related entity is fetched right away. With `@OneToMany`, the related collection is only loaded when you first access it.

The difference becomes clear in this simple example:

```
1 Long employeeCount = allCompanies.stream()
2     .mapToLong(c -> (long) c.getEmployees().size())
3     .sum();
```

At first glance, this looks like harmless Java code. But each call to `c.getEmployees().size()` may trigger a new SQL query. On a dataset of just three companies, the log shows three additional selects. On a production dataset of several hundred companies, this can balloon into hundreds of queries — all caused by a single aggregation line.

```
1 select c1_0.id, c1_0.name from company c1_0
2 select e1_0.id, e1_0.company_id, e1_0.age, e1_0.first_name, e1_0.last_name
3 from employee e1_0 where e1_0.company_id=?
4 select e1_0.id, e1_0.company_id, e1_0.age, e1_0.first_name, e1_0.last_name
5 from employee e1_0 where e1_0.company_id=?
6 select e1_0.id, e1_0.company_id, e1_0.age, e1_0.first_name, e1_0.last_name
7 from employee e1_0 where e1_0.company_id=?
```

The fix? It depends...

One way is to use EntityGraph again. If you are really sure, that you will use a specific field of the entity after the initial load, it may make sense to join it into the first query as we learned above. You can even traverse into embedded entities to extract

the smallest suitable dataset for your needs. Just be aware of who else uses the EntityGraph-improved Repository method: If others won't use the fields just joined, it may impact their performance by transferring a larger result set.

The second way requires a bit more structure. Java's Stream API invites developers to do calculations in code, which could be done more efficient in the relational database. If you want aggregated data, ask the database directly instead of relying on JPA's lazy resolution which is not obvious within a lambda expression. A simple projection query avoids the query storm entirely:

```
1 @Query("select new de.mischok.academy.companydatabase.domain.CompanyAverageAge(e.company,
   avg(e.age))
2 from Employee e group by e.company")
3 List<CompanyAverageAge> getAverageAges();
```

This tells Hibernate to fetch the average age per company in a single query. The collection is never materialized in memory, and the database does the heavy lifting — exactly what it's good at. You'll find a test case illustrating the tweak here: [JpqlTest.java](#).

The opposite trap is eager loading. When JPA eagerly fetches associations, especially in central domain entities, the SQL can quickly grow into massive joins. Suddenly, one query drags in half your schema, with a payload far larger than what the application actually needs. Especially if you have reference cycles in your schema, queries can blow up fastly.

Best practice:

- Prefer setting @ManyToOne explicitly to LAZY, even if JPA defaults to EAGER. But keep an eye on the impact this change has, LazyInitializationExceptions may be missed in transactional tests.
- Use eager loading only when you are certain the data is always required.
- Prefer explicit control via @EntityGraph or tailored queries when performance matters.

The key is not to fear lazy or eager loading — but to understand their tradeoffs. Each query the ORM generates is ultimately your responsibility. And as we've seen, what looks like a neat one-liner in Java may translate into a torrent of SQL in production.

Shorts: Other Pitfalls and Solutions

Not every JPA performance issue deserves a full chapter. Some are smaller, yet they can quietly erode application speed and stability. Here are three categories that regularly cause trouble in real-world projects.

Too Much Information

When exposing entities directly via REST, it's easy to end up with gigantic JSON payloads. A Company entity with embedded Employees, each of which includes its Company again, and so on — the cycle never ends.

Besides bloating payload size, this also leads to unnecessary serialization work and network overhead. The result: slow APIs and frustrated clients. And maybe security flaws as you unintentionally expose fields to the consumer of the API.

Solution:

- Design REST responses carefully, splitting large structures into multiple resources.
- Use pagination for large collections. Spring Data JPA provides everything you need for it.
- Consider Data Transfer Objects (DTOs) to control exactly which fields are exposed.

Flushing

In a transactional context, JPA collects changes in its persistence context. At some point, those changes must be flushed to the database. This process ensures consistency, but it's expensive.

Calling flush() explicitly inside loops or in performance-critical paths can kill throughput. Each flush forces JPA to synchronize all pending changes, often resulting in more SQL than expected.

Solution:

- Let the transaction manager handle flushing by default.
- Call `flush()` only when truly necessary (e.g., when IDs from just-persisted entities are needed immediately).
- Keep in mind: every flush is a round trip to the database. Sometimes the effect of one `flush()` after a loop is the same like a `flush()` in every loop cycle – but performance will be dramatically better!

Memory Pitfalls

Entities are not just POJOs — even if we treat them as such: they carry persistence state. Storing them in session maps, HTTP sessions, or static fields can lead to subtle memory leaks. Garbage collection cannot reclaim them while the persistence context still holds references, causing heap usage to grow uncontrollably.

Solution:

- Keep entities short-lived and stateless, i.e. never reference them in a non-data class like a Spring Bean.
- Detach them when passing outside the persistence layer.
- Avoid storing them in caches or sessions unless you really understand the consequences. Prefer to save ID's and reload the entity if needed.

Each of these problems may seem minor in isolation, but together they can cripple performance. As with the N+1 problem, awareness is half the battle. Once you know what to watch out for, the fixes are often straightforward.

Beyond JPA: The Holistic View

So far, we've looked at specific pitfalls and their solutions. But performance is rarely about a single annotation or query tweak. In practice, it emerges from the bigger picture: data models, access patterns, and architectural decisions. JPA can be part of a high-performance system — but only if it's used with the right mindset.

The first principle is anticipate scale early. Many JPA performance problems are invisible in small test datasets. A query that takes 5 ms on a few dozen rows may take seconds on millions. The only reliable way to detect such issues is to test with production-like data volumes from the start. Teams that integrate realistic test data into their CI pipelines discover bottlenecks long before users do.

Second, remember that JPA is not always the right tool. For most CRUD operations, it's a huge productivity boost. But for complex reporting queries, heavy aggregations, or large-scale batch jobs, the ORM abstractions become a burden. In these cases, dropping down to plain SQL or using a query library like jOOQ can be the pragmatic choice. Mixing approaches is not a failure — it's part of treating persistence as a first-class design concern.

Finally, recognize that performance tuning is both technical and organizational. Developers need the freedom to choose the right persistence approach, and teams need the discipline to question defaults. Optimizing JPA is not about blind annotation tweaking; it's about understanding the interaction between code, data, and infrastructure.

In other words: JPA is a powerful tool, but not a silver bullet. The most successful teams don't try to bend it to every problem. They use it where it shines — and reach for other solutions when the problem demands it.

High Speed JPA is Possible

JPA's reputation for being slow is undeserved. As we've seen, it's not the framework itself that drags down performance, but the way it's used. The infamous N+1 problem, the traps of lazy and eager loading, excessive flushing, or oversized REST payloads — all of these are pitfalls born from defaults and misunderstandings, not from inherent flaws in JPA.

The encouraging part is that the solutions are straightforward once you know them. Use `@EntityGraph` to control fetching. Keep a critical eye on lazy and eager defaults. Avoid cascading surprises. Test against realistic data. These principles are enough to prevent most performance disasters.

But the deeper lesson is that persistence should never be an afterthought. Treating database access as a first-class design concern — just like API design, security, or user experience — makes all the difference. Sometimes the best solution is an optimized JPA query, sometimes a DTO projection, sometimes a direct SQL statement. The point is to remain flexible, pragmatic, and aware of the tradeoffs. And always develop test driven to be prepared for future refactorings.

In the end, the paradox we began with — green tests in development, but poor performance in production — is not inevitable. With the right mindset, you can enjoy the productivity benefits of JPA without falling into its traps. High Speed JPA is not a dream. It's a matter of awareness, design, and a willingness to look beyond the defaults.

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



JAVAPRO
THE FREE MAGAZINE FOR THE JEE8 COMMUNITY

30 YEARS OF JAVA
SPECIAL EDITION

JAVAPRO

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



Jakarta Data and NoSQL – Standardized Data Access for Jakarta EE

Databases # JakartaEE # NoSQL # SQL

Author: Werner Keil

Werner Keil is a Cloud Architect, Jakarta EE and Microservice expert. Helping Global 500 Enterprises across industries and leading IT vendors. He worked for over 30 years as an IT Manager, PM, Coach, SW architect, developer, and consultant in Finance, Mobile, Media, Transport, and the Public sector. Werner develops enterprise systems using Java, Java/Jakarta EE, Oracle, IBM, Spring or Microsoft technologies, JavaScript, Node, Angular, dynamic or functional languages. Besides working for major companies, he runs his own consultancy, CATMedia, and, in his spare time, runs and succeeds in projects and writes song lyrics, novels, screenplays, technical books, and articles. Werner is Committer at Apache Foundation, Eclipse Foundation, Babel Language Champion, UOMo Project Lead and active member of the Java Community Process in JSRs like 321 (Trusted Java), 344 (JSF 2.2), 354 (Money, also Maintenance Lead), 358/364 (JCP.next), 362 (Portlet 3), 363 (Unit-API 1), 365 (CDI 2), 366 (Java EE 8), 375 (Java EE Security), 380 (Bean Validation 2), 385 (Unit-API 2, also Spec Lead) and was the longest serving Individual Member of the Executive Committee for 9 years in a row till 2017. Since its inception, Werner has been the Committer representative in the Jakarta EE Specification Committee.



LinkedIn

The amount of data collected by applications nowadays is growing rapidly. Many of them need to support both relational SQL and non-relational NoSQL databases. Jakarta Data provides an API to allow easy data access. Developers can split the persistence mechanism and the model using common features like the Repository pattern and seamlessly switch between SQL and NoSQL databases or even use both in the same application.

What are NoSQL Databases?

NoSQL databases, also called “not only SQL”, “non-Relational” or “non-SQL”, are a type of database management system, where storage and querying data are performed differently from traditional ways used in relational databases.

Instead of the tabular structure of a relational database with tables and columns, NoSQL databases often use different data structures, like a JSON document or a graph structure. As the non-relational database design does not require a schema, it offers greater flexibility and scalability to manage large and usually unstructured data.

NoSQL also works well for distributed databases, where data is copied and stored on multiple servers, either locally or remotely. This improves availability and reliability of data. If some data is unavailable, the rest of the database can continue to run. While most relational database systems won’t work anymore, if say the list of schemas or tables is missing or corrupted.

Features of NoSQL Databases

While relational databases optimize storage and avoid redundancy, NoSQL databases can handle large amounts of unstructured or semi-structured data, offering scalability and flexibility for modern distributed applications.

The key features of NoSQL databases are:

- Flexible data structures and dynamic schemas instead of fixed schemas
- Horizontal scalability
- Low latency
- Large number of concurrent users and data volumes
- Distributed nature
- Higher performance and speed

Types of NoSQL Databases

The main types of NoSQL databases are:

- Key-Value stores
- Column-oriented
- Document stores
- Graph databases
- Time series

Key-Value Stores

Key-value stores (databases) are the simplest type of NoSQL database. Data is stored in key-value pairs, essentially like a **Map** in the Java Collections API.

Examples of Key-value stores are Amazon DynamoDB, S3, Redis, Hazelcast or Memcached

Column-oriented

Column-oriented databases store data in columns instead of rows, like relational databases do. This type of database is well-suited for mixing structured, semi-structured and unstructured data, offering high concurrency and scalability.

Examples of Column-oriented databases are Apache HBase, Apache Cassandra, Scylla, Clouddata, SimpleDB or Azure Cosmos DB.

Document-oriented NoSQL

Document-oriented NoSQL databases store data in flexible documents, usually in JSON, BSON, or XML format, instead of rigid tables. These documents can have different structures, which is well-suited for applications that require variable data definitions and can enable faster, more agile development cycles.

Examples of document store NoSQL databases include MongoDB, CouchDB, Couchbase Server, Elasticsearch, Google Cloud Firestore, Amazon DocumentDB and DynamoDB.

Document Stores

Document stores allow to store, retrieve and edit documents in different formats like XML or more commonly JSON. Documents can be embedded inside other documents. Document stores are best suited for unstructured data like social media profiles, product sheets or content management.

Popular document stores include MongoDB, Couchbase, Apache CouchDB, Elastic or Oracle NoSQL Database.

Graph Databases

Graph databases allow to store more complex relationships than relational databases. The model graph applies graph theory, data is stored as a network of nodes and edges. Graph databases are well-suited for the “social graph” in social networks, but also AI models, especially Retrieval Augmented Generation (RAG) databases.

Examples for graph databases are Neo4J, ArangoDB, OrientDb, InfoGrid, Sones, HyperGraphDB or JanusGraph.

Time Series

A time series database can be seen as a specialized form of key-value store, with a timestamp as the key, and values associated with that timestamp.

Examples for time series databases are InfluxDB, KairosDB, TimescaleDB, Apache Pinot, Riak-TS or Prometheus.

Multi-Model

Some NoSQL databases combine more than one type, for example:

1. OrientDB (Graph, Document)
2. Couchbase (Key-value, Document)
3. ArangoDB (Document, Graph, Key-value)
4. Elasticsearch (Document, Graph)

SQL vs. NoSQL

These are the the main differences between SQL and NoSQL databases:

Feature	SQL	NoSQL
Schema	Fixed	Dynamic or none
ACID/BASE	ACID	BASE or limited ACID
Data Model	Structured, table-based	Flexible: documents, key-value, graphs
Data Structure	Normalized	Denormalized
Scalability	Vertical scaling	Horizontal scaling
Best Use Cases	Transactional, flow-based applications	Big Data, AI, <u>Real time</u> data analytics, embedded/IoT

Differences between SQL and NoSQL

ACID vs. BASE principles:

ACID



- Atomicity
- Consistency
- Isolation
- Durability

BASE



- Basically Available
- Soft state
- Eventual consistency

Here are the most common keywords used by different database types:

SQL	Key-Value	Column	Document	Graph
Table	Bucket	Column family	Collection	
Row		Column	Document	Vertex
Column		Key/value pair	Key/value pair	Vertex and Edge property
Relationship			Link	Edge
Index				

Keywords used by different database types

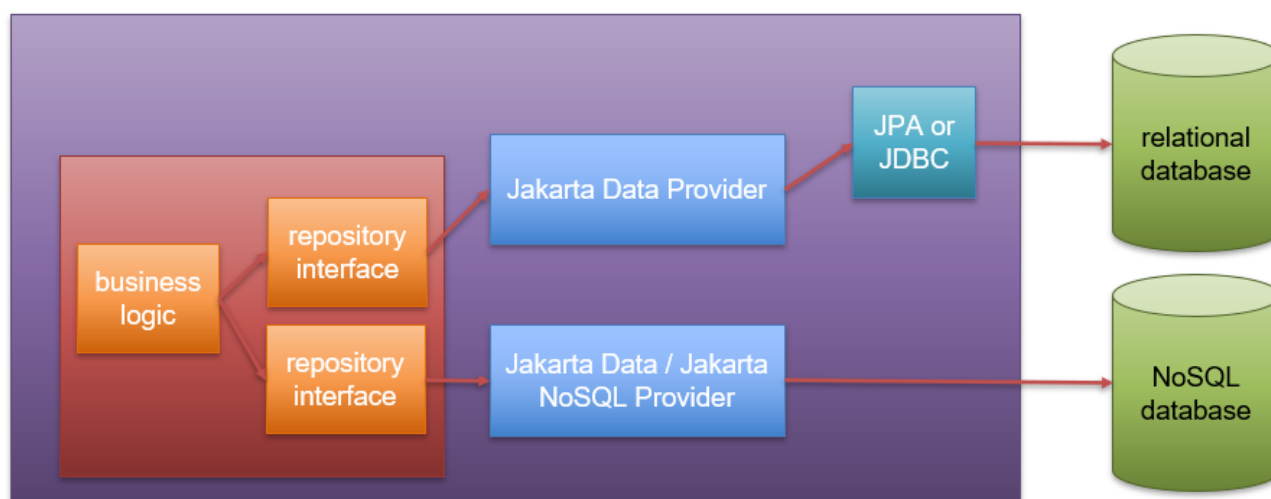
Using JPA for NoSQL databases can be problematic:

1. Async Saves
2. Async Callback
3. Time to Live (TTL)
4. Consistency Level
5. SQL based
6. Diversity of NoSQL

This is where Jakarta Data and Jakarta NoSQL come into play.

Jakarta Data

[Jakarta Data](#) version 1.0 was released as part of Jakarta EE 11. It offers simplified data access to both relational and NoSQL databases. Applying common concepts like repositories and custom queries, known from popular data access libraries like Hibernate, Spring Data or Micronaut Data. Most of them support Jakarta Data or plan to do so in future versions.



Key elements of the Jakarta Data API are:

- Repository
- Order
- Query
- Sort
- Page

Repository

Here is a Repository example:

```
1 @Repository
2 public interface Products extends DataRepository<Product, Long> {
3     List<Product> findByPriceLessThan(float maxPrice);
4     void save(Product product);
5     ...
6 }
```

And how to use it:

```
1 @Inject
2 Products products;
3
4 products.save(new Product(1, "$25 gift card", 25.0f));
5
6 List<Product> found = products.findByPriceLessThan(100.0f);
```

Doing the same with the **CrudRepository**:

```
1 @Repository
2 public interface Products extends CrudRepository<Product, Long> {
3     List<Product> findByPriceLessThan(float maxPrice);
4     ...
5 }
```

Inherits methods like:

```
1 save(E entity);
2 saveAll(Iterable<E> entities);
3 findAll();
4 findById(K id);
5 existsById(K id);
6 delete(E entity);
7 deleteById(K id);
8 ...
```

Allowing a more compact repository implementation.

Ordering and Sorting

An example for Ordering and a dynamic Query:

```

1  @Repository
2  public interface Products {
3      Product findById(long productId);
4
5      @OrderBy("price")
6      List<Product> findByNameContains(String searchFor);
7
8
9      @Query("UPDATE Product o SET o.price = o.price - (?2 * o.price) WHERE o.id = ?1")
10     boolean discount(long productId, float discountRate);
11
12     void save(Product p);
13 }

```

Applying Sort:

```

1  found = products.findByNameContains(namePattern, Sort.desc("price"));

```

Pagination

Jakarta Data supports both offset and cursor pagination.

Here's an example for offset pagination, using a simple numerical offset:

```

1  @Repository
2  public interface BeerRepository extends BasicRepository<Beer, String> { }
3
4  Pageable page = Pageable.ofPage(1).sortBy(Sort.desc("style"));
5  Page<Beer> page1 = repository.findAll(page);
6  System.out.println("The first page:");
7  page1.forEach(System.out::println);
8
9  Pageable secondPage = page.next();
10 Page<Beer> page2 = repository.findAll(secondPage);
11 System.out.println("The second page:");
12 page2.forEach(System.out::println);

```

Cursor-based pagination uses a reference, rather than a simple offset, in this case a cursor to break records into pages:

```

1  @Repository
2  public interface FruitRepository extends BasicRepository<Fruit, String> {
3
4      @Find
5      CursoredPage<Fruit> cursor(PageRequest pageRequest, Sort<Fruit> order);
6
7  }
8
9  var pageRequest = PageRequest.ofSize(size).beforeCursor(PageRequest.Cursor.forKey(before));
10 var page = fruitRepository.cursor(pageRequest, DESC);

```

Event Handling

Jakarta Data further supports persistence life cycle events like:

```

1  void afterUpdate(@Observes PostUpdateEvent<Book> event) {
2      // Perform post-update actions
3  }

```

Offering the same functionality without having to write e.g. database triggers.

Jakarta NoSQL

[Jakarta NoSQL](#), while it started earlier than the factored-out Jakarta Data, released its 1.0 version in spring of 2025. The content of Jakarta EE 11 had already been finalized, so Jakarta NoSQL 1.1 aims for inclusion in Jakarta EE 12.

Jakarta NoSQL offers many annotations familiar to those in JPA, making it also blend-in well with Jakarta Data, especially if you switch between SQL and NoSQL databases:

Annotation	Description
@Entity	Defines a class as a persistable entity
@Id	Marks the primary key of an entity
@Column	Maps a property or field to a database column
@Embeddable, @MappedSuperclass	Enable composition and inheritance
@Convert	Handles type conversion for persistence

Jakarta NoSQL Annotations

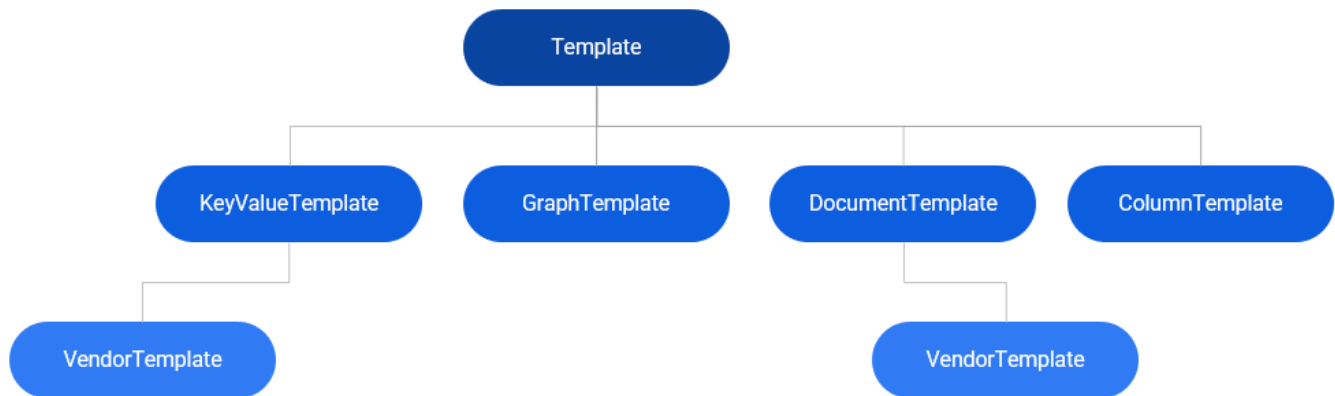
Repository

Using the compatible implementation [Eclipse JNoSQL](#), you can define a very similar BeerRepository we saw earlier, but for a NoSQL database like the Oracle NoSQL Database:

```
1 import jakarta.data.repository.Query;
2 import jakarta.data.repository.Repository;
3 import org.eclipse.jnosql.databases.oracle.mapping.OracleNoSQLRepository;
4
5 @Repository
6 public interface BeerRepository extends OracleNoSQLRepository<Beer, String> {
7
8     Set<Beer> findByStyle(String style);
9
10    @Query("select * from Beer")
11    Set<Beer> query();
12 }
```

Template

In addition to the Repository, Jakarta NoSQL also supports the **Template** interface to connect Java entities with NoSQL databases. Through this interface, you can perform standard CRUD operations and fluent queries with a consistent API.



Let's take an example of a **Car** entity:

```

1  @Entity
2  public class Car {
3
4      @Id
5      private Long id;
6
7      @Column
8      private String name;
9
10     @Column
11     private CarType type;
12
13     // Getters and setters...
14 }

```

And apply some simple CRUD operations using the template:

```

1  @Inject
2  Template template;
3
4  Car car = Car.id(1L)
5              .name("Ferrari")
6              .type(CarType.SPORT);
7
8  template.insert(car);
9
10 Optional<Car> result = template.find(Car.class, 1L);
11 template.delete(Car.class, 1L);

```

In addition to basic persistence, the **Template** interface also offers a typesafe and expressive **fluent query API**:

```

1  List<Car> cars = template.select(Car.class)
2                          .where("type").eq(CarType.SUV)
3                          .orderBy("name").asc()
4                          .result();
5
6  template.delete(Car.class)
7              .where("type").eq(CarType.COUPÉ)
8              .execute();

```

Supported NoSQL Database Types

Eclipse JNoSQL currently supports the three most common types of NoSQL databases: key-value, column-based, document and graph.

Graph Databases

For graph databases JNoSQL builds on top of [Apache TinkerPop](#), by which there are more than 30 supported NoSQL databases as of now.

Custom Annotations

For some NoSQL databases Eclipse JNoSQL offers custom annotations. Here an example for [Apache Cassandra](#).

Entity:

```
1 @Entity("god")
2 public class God {
3
4     @Column
5     private String name;
6
7     @UDT("weapon")
8     @Column
9     private Weapon weapon;
10
11 }
```

Repository:

```
1 @Repository
2 interface GodRepository extends CassandraRepository<God, String> {
3
4     @CQL("select * from God where name = ?")
5     List<God> findByName(String name);
6
7 }
```

Conclusion

Jakarta Data and Jakarta NoSQL streamline the integration of Java applications with both relational and non-relational databases. Based on active and vibrant communities like Eclipse or Apache TinkerPop, and supported by some of the most popular Java persistence libraries including Hibernate, Micronaut or OpenLiberty, both Jakarta EE specs can help improve your data applications into a vendor-neutral future.

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



Sarcasm-as-a-Service: Five Years Later

Accessibility # AI # Azure # TTS

Author: Rory Preddy

Rory Preddy is a Principal Advocate at Microsoft, where he helps professional cloud developers discover and successfully use Microsoft's platforms. He is a seasoned speaker whose talks are both meaningful and humorous, and he speaks around the world, empowering developers to achieve more. Rory has over 25 years of experience in software development, working with various technologies such as Java, AI and Azure.



LinkedIn



Abb. 1: Rory Preddy, Author, Travelling

TLDR; Demo App <https://aka.ms/ttsazure>, Repo: <https://github.com/roryp/ttsazure>

I am 4 foot 1, and I have achondroplasia (dwarfism). Following spinal surgery, I was in a coma for three months and woke up unable to speak. I longed to tell my family that I loved them. That experience changed everything. A voice is not just sound; it is identity, emotion, and memory.

As a Developer Advocate at Microsoft, my role blends equal parts influencer and engineer, with a constant supply of questions from younger generations. When I first presented this talk in 2019, my career took off—audiences laughed at my demos, my jokes, and occasionally at me. Many assumed I was joking when I promised to revisit the project. Yet here I am, in a country where perpetual load shedding inspired this talk’s subtitle: “The power is out again. Fantastic.”

It has been five years since I first delivered my talk Sarcasm-as-a-Service and launched a demo that explored whether machines could be taught to use sarcasm. I would like to say that I now fully understand sarcasm; however, that would be untrue. What I have truly learned is that sarcasm remains messy, emotional, and deeply human. My task continues to be demonstrating how machines might help us experience and convey it.



Abb. 2: Image, “Sarcasm is hard to detect”

Why Does Sarcasm Matter?

Since my first attempt at teaching machines sarcasm, I have spent considerable time working advocating AI Accessibility and reflecting on human communication. Sarcasm is far more than a source of cheap humor; it functions as an encrypted form of communication. Through word choice, tone, and context, sarcasm encodes meaning that only those attuned to it can fully appreciate.



Abb. 3: Inclusive design, <https://accessibility.blog.gov.uk>

For individuals with disabilities—particularly those who struggle to read facial expressions or rely on assistive technologies—sarcasm can be one of the few expressive tools available. In my work, I often share the stories of Ron, an 82-year-old learning technology for the first time; Ashley, a developer who reads code with a screen reader; and Claudia, who depends on a screen magnifier. They represent just a fraction of the billion people worldwide living with disabilities. As a developer of tools that run on 95% of the world’s desktops, I owe them more than token gestures—I owe them technology that speaks their language, including its sarcasm.



Abb. 4: Secret Service Sarcasm Detector

[Secret Service Sarcasm Detector](#)

Sarcasm is hard to grasp, even governments have grappled with sarcasm's complexity. In 2014, the U.S. Secret Service sought a social-media analytics system capable of detecting sarcastic posts. The rationale was clear: misreading sarcasm in a threat could result in catastrophic consequences. If the federal government struggles to distinguish sarcasm from seriousness, imagine the challenge for a smart speaker.

Opportunities and Risks

Sarcasm can be powerful, but it is not always appropriate. In my daily work, I encounter many potential applications: injecting humor into developer advocacy posts, adding wit to customer support systems, designing code review bots with a playful tone, or bringing comic relief to training modules. Yet sarcasm is also easy to misuse. Machines without context can transform support interactions into unintentional insults.



Abb. 5: Tay AI, [https://en.wikipedia.org/wiki/Tay_\(chatbot\)](https://en.wikipedia.org/wiki/Tay_(chatbot))

The most infamous example is Microsoft’s Tay, a Twitter bot that quickly began parroting offensive content. That episode highlighted the critical importance of ethical guardrails in large language models.

Revisiting the Original Demo

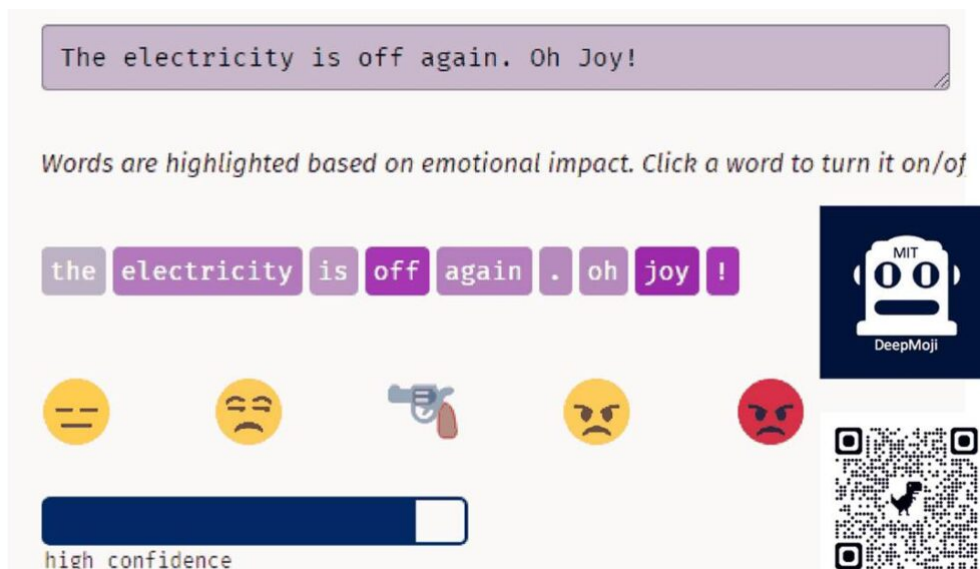


Abb. 6: DeepMoji, <https://www.media.mit.edu/projects/deepmoji/overview/>

In 2019, I built a sarcastic bot using the tools of the time: vector space models, recurrent neural networks (RNNs), and the DeepMoji dataset. In DeepMoji uses a twitter database to pair with emojis to classify sarcasm. The results were more comedic than rigorous.

One enduring example was the phrase: “The electricity is off again. Oh joy!”—a sentiment instantly familiar to South Africans. The dataset paired such statements with eye-rolling emojis, but the machine never understood the underlying despair. Context was absent; sentiment was reduced to emoji counts.

Why is Sarcasm so hard to grasp? It’s because sarcasm is often confused with irony or insult. Irony occurs when outcomes are opposite to expectations—for instance, a billboard promising “Complete home repairs” placed on a derelict shack. Sarcasm is biting irony, derived from the Greek *sarkasmós*, meaning “to tear flesh” or “sneer.” It delivers harsh criticism cloaked in pleasant words.

This could be why machines struggle because sarcasm operates on multiple layers: the literal text, cultural and situational context, and tone. For example, if my wife suggests lunges to stay in shape and I reply, “That’s a big step forward,” the pun is obvious in text, but the sarcasm comes alive through tone—something a screen reader or machine cannot capture.

Prosody and Brain Chemistry

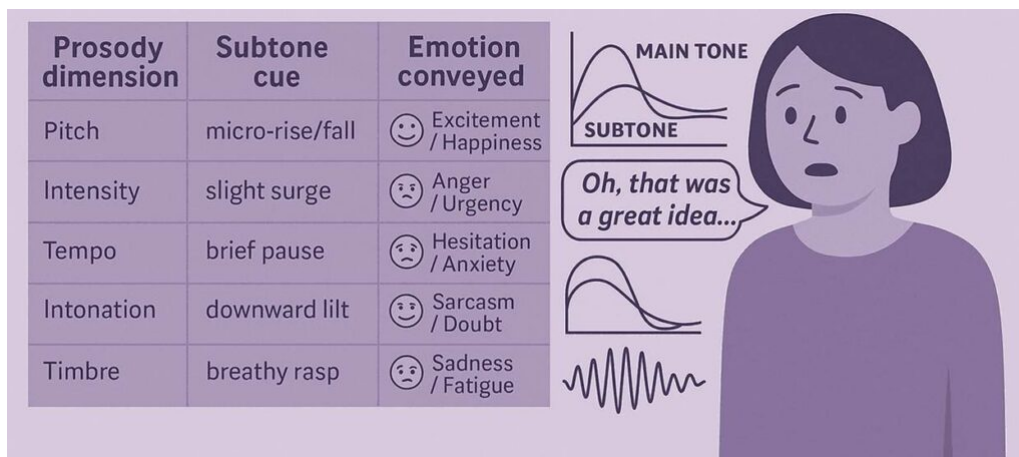


Abb. 7: Prosody dimensions

Capturing sarcasm requires modeling prosody—the “music” of speech. Prosody includes pitch, duration, loudness, and timbre, which together shape intonation and stress. A slight rise in pitch may signal excitement, a pause may convey doubt, and timbre may reflect joy or sadness. These subtle cues help listeners decode sarcasm.

Why does this matter? Because laughter is not merely sound but chemistry. A well-timed sarcastic remark triggers a neurochemical response: reduced cortisol (stress), alongside dopamine, oxytocin, and endorphins that foster trust and bonding. Machines must therefore do more than imitate words; they must convincingly stimulate this human chemistry.

Recent Advances and the New Sarcastic Soundboard

Fast-forward to 2025 and the past five years have brought remarkable progress with large language models and text-to-speech systems. Models such as Azure OpenAI’s GPT-Audio can now generate expressive speech. Building on this, I developed a new sarcastic soundboard using Azure Container Apps → <https://aka.ms/ttsazure> (see [Appendix A](#) for a technical deep dive)

The web app allows users to enter text, choose a “vibe” (Excited, Calm, Sad, Sarcastic), and select from premium voices. The backend, built with Spring Boot on Java 21, integrates with Azure OpenAI’s GPT-Audio model. It supports multiple formats, streaming, and secure authentication through Azure Managed Identity. Safety features include rate limits, logging, and safeguards against misuse.

Conclusion

[Microsoft Pilot with Brian Jeansonne](https://www.youtube.com/watch?v=5FWwM1S8RfE), <https://www.youtube.com/watch?v=5FWwM1S8RfE>

Are we there yet? The answer is both yes and no. Machines still lack true understanding; they miss cultural nuance, context, and the unspoken smirk. Today’s models can detect sarcasm, generate witty retorts, and mimic prosody convincingly enough to trigger human responses. They can restore voices once thought lost.

Microsoft’s pilot with Team Gleason helped Brian Jeansonne, living with ALS, explore what an Azure custom neural voice can do to bring more of his personality into everyday conversations. In one case, hearing Brian’s preserved voice say “Hey, Christy, I love you” moved me to tears. This is not merely technology for humor—it is technology for dignity, connection, and humanity.

As I conclude, the power goes out again—oh joy. Five years later, I remain cautiously optimistic. Machines may never truly be sarcastic, but they can certainly help us communicate and preserve what makes us human. If that isn’t fantastic, I don’t know what is.

Appendix A – Demo Deep dive

Lets dive into my demo → A modern, interactive text-to-speech application built with Spring Boot and Azure OpenAI's GPT-Audio model. It can transform any text into natural-sounding speech with multiple voice options, customizable styles, and advanced tone guidance.

1) Azure Developer CLI (azd) foundation

First lets get everything ready and clone the GitHub repo:

```
1 git clone https://github.com/roryp/ttsazure.git
2 cd ttsazure
```

Next we'll leverage Azure Developer CLI (azd) which orchestrates the demo apps end-to-end deployment from a single command:

```
1 azd up
```

azd reads the project's azure.yaml and coordinates both infrastructure provisioning and app deployment.

2) Infrastructure provisioning (infra/main.bicep)

Digging deeper into azd we see it uses a single Bicep template to create all the cloud resources required by the demo:

```
1 // Core infrastructure components
2 resource managedIdentity 'Microsoft.ManagedIdentity/userAssignedIdentities@2023-01-31'
3 resource openAiAccount 'Microsoft.CognitiveServices/accounts@2024-10-01'
4 resource containerRegistry 'Microsoft.ContainerRegistry/registries@2023-07-01'
5 resource containerAppsEnvironment 'Microsoft.App/managedEnvironments@2024-03-01'
6 resource containerApp 'Microsoft.App/containerApps@2024-03-01'
```

Key Infrastructure Components created:

1. Azure OpenAI Service with the GPT-Audio model
2. Azure Container Registry for secure image storage
3. Azure Container Apps environment for serverless container hosting
4. Managed Identity for keyless authentication
5. RBAC Role Assignments for secure resource access

Region & security

- Deploys to East US 2 for model availability.
- Enforces role-based access control boundaries across all resources.

3) Zero-Secret Security Architecture

With the infrastructure in place, the application authenticates without API keys using Azure AD and Managed Identity. The service layer acquires an access token at runtime and forwards it to Azure OpenAI:

```

1  @Service
2  public class OpenAIService {
3      private final TokenCredential credential;
4
5      public OpenAIService() {
6          this.credential = new DefaultAzureCredentialBuilder().build();
7      }
8
9      public byte[] generateSpeech(String text, String voice, String style, String format)
10     {
11         // Get access token for Azure Cognitive Services
12         TokenRequestContext tokenRequestContext = new TokenRequestContext()
13             .addScopes("https://cognitiveservices.azure.com/.default");
14         AccessToken token = credential.getToken(tokenRequestContext).block();
15
16         // Use token in Authorization header
17         HttpRequest request = HttpRequest.newBuilder()
18             .header("Authorization", "Bearer " + token.getToken())
19             .build();
20     }

```

4) Vibe System Architecture

On top of raw TTS, the demo layers expressive voice “vibes.” VibeService.java loads 12 predefined styles from vibes.json—including “Sarcastic.”

```

1  {
2      "name": "Sarcastic",
3      "description": "Voice Affect: Dry, pointed, subtly mocking; embody sarcasm and irony.\n
4      \nTone: Sardonic, witty, slightly condescending; convey clever mockery.\n\nPacing:
5      Normal with strategic emphasis on ironic points and subtle pauses.\n\nEmotion:
6      Cleverly sarcastic; express wit and subtle mockery.",
7      "script": "Oh sure, because that's exactly what everyone was thinking..."
8  }

```

5) Audio Generation Pipeline

From request to playback, the flow runs through a single controller - TtsController.java.

5.1 Input Validation and Rate Limiting

```

1  @PostMapping("/tts")
2  public String generateTts(@RequestParam String text, @RequestParam String voice,
3      @RequestParam String style, @RequestParam String format) {
4      // Validate input length (max 4,000 characters)
5      if (text.length() > 4000) {
6          text = text.substring(0, 4000);
7      }
8
9      // Rate limiting check (10 requests/minute, 100/hour, 50k characters/hour)
10     if (!rateLimitService.isAllowed(clientId, text.length())) {
11         // Return rate limit error
12     }
13 }

```


5.2 Azure OpenAI synthesis

The OpenAIService.java handles the actual speech synthesis:

```

1  public byte[] generateSpeech(String text, String voice, String style, String format) {
2      // Prepare request body
3      Map<String, Object> requestBody = new HashMap<>();
4      requestBody.put("model", "gpt-audio");
5      requestBody.put("input", processTextForSpeech(text));
6      requestBody.put("voice", voice); // 11 available voices
7      requestBody.put("response_format", format); // mp3, wav, opus
8
9      if (style != null && !style.trim().isEmpty()) {
10         requestBody.put("instructions", style.trim()); // Vibe instructions
11     }
12
13     // Make API call to Azure OpenAI
14     String url = String.format("%s/openai/v1/chat/completions", endpoint);
15
16     HttpRequest request = HttpRequest.newBuilder()
17         .uri(URI.create(url))
18         .header("Authorization", "Bearer " + token.getToken())
19         .header("Content-Type", "application/json")
20         .POST(HttpRequest.BodyPublishers.ofString(requestBodyJson))
21         .build();
22
23     return httpClient.send(request, HttpResponse.BodyHandlers.ofByteArray()).body();
24 }

```

5.3. Audio Storage and Retrieval

Generated audio is stored in AudioStore.java

```

1  @GetMapping("/audio/{id}")
2  public ResponseEntity<byte[]> streamAudio(@PathVariable String id,
3                                             @RequestParam boolean download,
4                                             @RequestParam String format) {
5      byte[] audioData = audioStore.retrieve(id);
6
7      HttpHeaders headers = new HttpHeaders();
8      headers.setContentType(MediaType.valueOf("audio/mpeg")); // MP3 format
9      headers.set("Accept-Ranges", "bytes");
10     headers.set("Cache-Control", "no-cache, no-store, must-revalidate");
11
12     if (download) {
13         headers.setContentDispositionFormData("attachment", "audio_" + id + ".mp3");
14     }
15
16     return ResponseEntity.ok().headers(headers).body(audioData);
17 }

```

6) End-to-end: creating the sarcastic MP3

With the pipeline wired, generating a sarcastic clip is a straight path from UI to audio.

Azure OpenAI TTS Audio Soundboard

Transform your text into natural-sounding speech using Azure OpenAI

Choose Voice

Voice: Select a voice

Alloy

Ash

Ballad

Coral

Echo

Fable

Nova

Onyx

Sage

Shimmer

Verse

Random

Choose Vibe

Shuffle Vibes

Description

Select a vibe to see the description

Sample Script

Select a vibe to see the sample script

Input Text:

My wife says I should do lunges to stay in shape. I guess that's a big step forward

83/4000 characters

Audio Format:

MP3 (Recommended)

Generate Voice

Use Vibe Script

Your Text

My wife says I should do lunges to stay in shape. I guess that's a big step forward

Voices: Ash
Style: Voice Affect: Dry, pointed, subtly mocking; embody sarcasm and irony. Tone: Sardonic, witty, slightly condescending; convey clever mockery. Pacing: Normal with strategic emphasis on ironic points and subtle pauses. Emotion: Cleverly sarcastic; express wit and subtle mockery.
Format: MP3

Generated Audio

0:06 / 0:06

Download MP3

File size: 106.5 KB

Abb. 8: Screenshot of the demo

6.1 Frontend interaction (index.html)

- **Choose a voice:** Alloy, Ash, Ballad, Coral, Echo, Fable, Nova, Onyx, Sage, Shimmer, Verse.
- **Pick a vibe:** click **Sarcastic** to auto-populate style and sample text.
- **Edit text** ($\leq 4,000$ characters).
- **Select format** (MP3 by default).
- **Generate:** click **Generate Voice**.

6.2 Backend Processing

In the backend, the `TtsController.java` now generates the sarcastic speech audio using the chosen style, stores it, and returns a unique audio ID to the frontend.

```

1 // TtsController processes the request
2 @PostMapping("/tts")
3 public String generateTts(@RequestParam String text, @RequestParam String voice,
4                           @RequestParam String style, @RequestParam String format) {
5
6     // Generate speech with sarcastic style
7     byte[] audioData = openAIService.generateSpeech(text, voice,
8             "Voice Affect: Dry, pointed, subtly mocking...", "mp3");
9
10    // Store audio with unique ID
11    String audioId = audioStore.store(audioData);
12
13    // Return success with audio ID for frontend
14    model.addAttribute("audioId", audioId);
15    model.addAttribute("success", "Voice generated successfully!");
16 }
```

6.3 Playback & download

The `index.html` now renders audio controls with download capability:

```

1 <audio controls autoplay>
2   <source th:src="@{'/audio/' + ${audioId} + '?format=mp3'}" type="audio/mpeg">
3 </audio>
4 <a th:href="@{'/audio/' + ${audioId} + '?download=true&format=mp3'}"
5   download="sarcastic-audio.mp3" class="download-btn">
6   Download MP3
7 </a>
```

7) Rate Limiting and Performance

One of the concerns with AI models is abuse so I added rate limiting through `RateLimitService.java`:

- 10 requests per minute per IP address
- 100 requests per hour per IP address
- 50,000 characters per hour per IP address
- Configurable limits via environment variables
- Graceful degradation with clear error messages

8) Container Apps Deployment Architecture

To handle scaling I edited the main.bicep config to deploy the app to Azure Container Apps with health probes and autoscale:

```

1 resource containerApp 'Microsoft.App/containerApps@2024-03-01' = {
2   properties: {
3     template: {
4       containers: [{
5         name: 'ttsazure'
6         resources: {
7           cpu: json('2.0')
8           memory: '4Gi'
9         }
10        probes: [
11          {
12            type: 'Liveness'
13            httpGet: { path: '/health', port: 8080 }
14          },
15          {
16            type: 'Readiness'
17            httpGet: { path: '/health', port: 8080 }
18          }
19        ]
20      }]
21      scale: {
22        minReplicas: 1 // Always-on for instant response
23        maxReplicas: 3 // Auto-scaling under load
24        rules: [{
25          name: 'http-scaling'
26          http: { metadata: { concurrentRequests: '10' }}
27        }]
28      }
29    }
30  }
31 }

```

9) Monitoring and Observability

A lightweight health endpoint surfaces key runtime signals:

```

1 @GetMapping("/health")
2 @ResponseBody
3 public Map<String, Object> health() {
4   return Map.of(
5     "status", "UP",
6     "audioStoreSize", audioStore.size(),
7     "rateLimitConfig", rateLimitProperties,
8     "timestamp", Instant.now()
9   );
10 }

```

10) The Final Product: Sarcastic MP3

The result is a **downloadable, expressive MP3**—in this case, a perfectly dry **sarcastic** read—generated securely (no secrets), reliably (autoscaled), and repeatably (azd up). The included [sarcastic.mp3](#) in the repository demonstrates the end-to-end system in action. Deploy it or give it a try at <https://aka.ms/ttsazure>

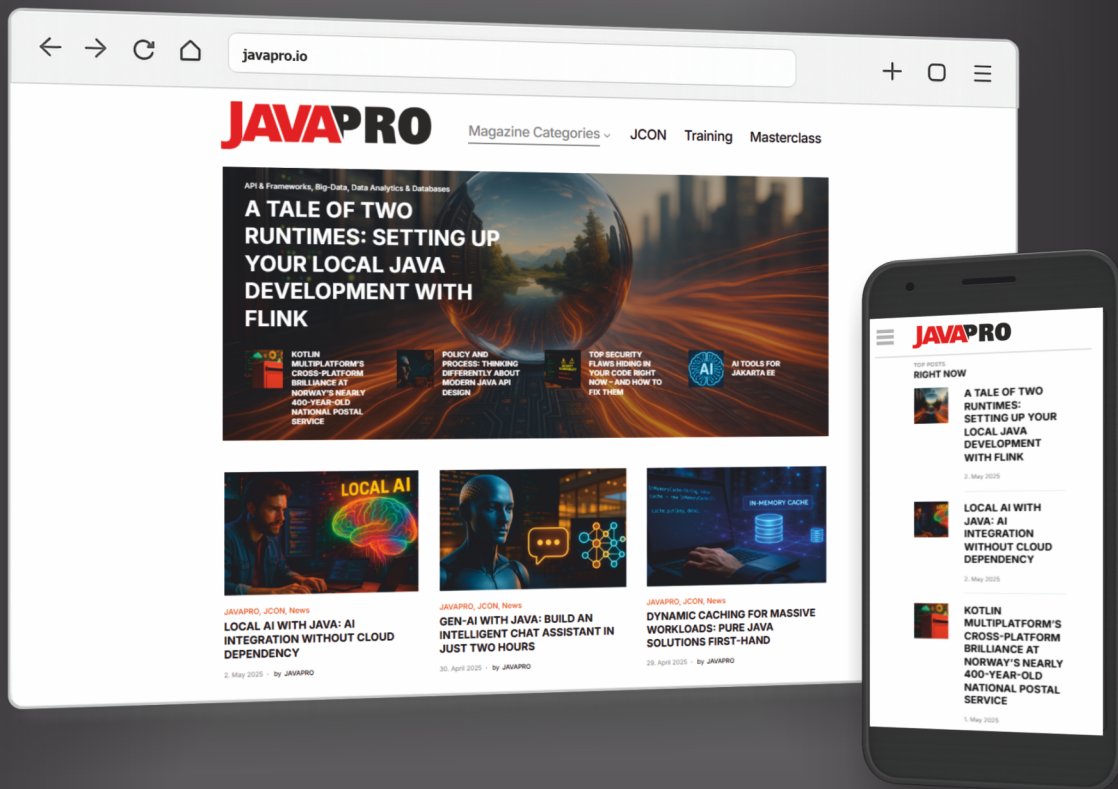
› [Back to the Table of Contents](#) ‹

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



Microservices: Lessons from the Trenches

Microservices

Architecture

Author: Rafał Maciak

Passionate about solving business problems with technology, especially using Kotlin. Rafał Maciak has extensive experience designing large-scale distributed applications. Believes that understanding the fundamentals of business and its model is key to building stable and maintainable systems. A pragmatist - likes to use the right tools for the right problems.



LinkedIn

During my 15-year journey in the world of software development, I spent the vast majority of my time designing and implementing systems using microservices architecture. I've been involved in multiple migrations from monolithic systems to microservices, as well as building new solutions from scratch. Along the way, my teams and I made plenty of mistakes. Some were quite trivial and easy to correct, others involved decisions that were difficult to reverse, or changes that required significant effort to fix. More than once, I found myself in situations where it seemed that working with this architecture was pointless and too challenging.

Fortunately, many failures gave me the right lessons, and realization that it's not just about the failures, as we also succeeded and got many things right. In this article, I share key takeaways from my experience, hopefully giving you the confidence to implement microservices architecture successfully. Let's get started!

Improper Service Boundary Designs

The first lesson addresses a common problem that I've encountered repeatedly in my projects: inappropriate service boundaries. In theory, these boundaries should align with business domains to minimize coupling and maximize cohesion. As is often the case, the theory is correct. I've seen firsthand how incorrect service boundaries have led to significant issues: developers facing constant headaches, errors become frequent, time-to-market increases, and ultimately, client or user satisfaction decreases. This is the complete opposite of what a microservices architecture should deliver.

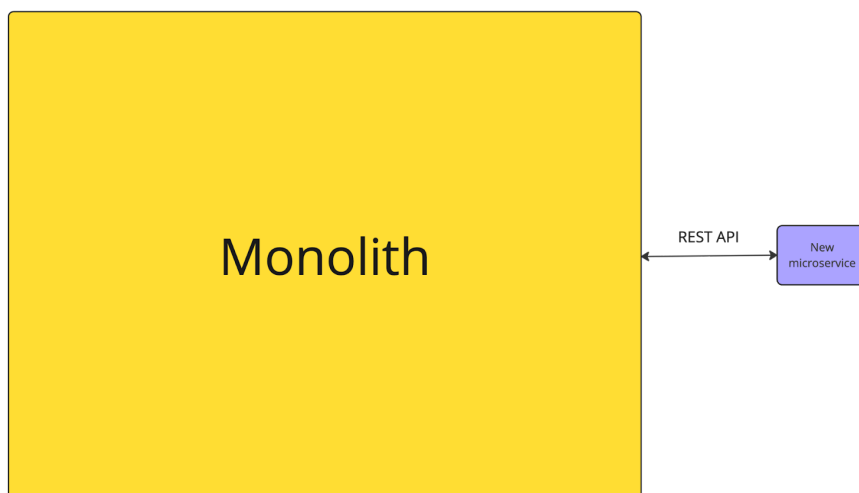
But why does this happen? In my opinion, the problem lies in the fact that while theory clearly defines what we want to achieve - namely, breaking services along business boundaries - it fails to provide a clear methodology for "how" to do it effectively. There is a surprising lack of guidance on how to properly partition a business domain into microservices to meet these goals. I've heard countless stories from colleagues about microservices projects that either failed or came dangerously close to failure due to this challenge. And I've personally been in the trenches, making mistakes in this area, more than once. Let me share two key lessons that cover this problem.

When Buzzwords Backfire – Lessons from "Project-Alpha"

I was working as a software developer on a project involving a large monolithic system. For the purposes of this article, I will refer to it as "project-alpha", since it was my first time with microservices, and I'll reference it throughout later chapters. At the time, I had no experience with microservice architectures, especially when it came to migrating a monolith. To make it even more thrilling, no one on our project team had any experience with microservices at that time.

At some point, a decision was made to upscale our architecture to microservices. The rationale was that the system was rapidly growing both in terms of customers and the number of features expected to be developed, which naturally meant a larger team working on the project. Looking back, I believe that the main reason for adopting a microservices architecture was the "hype" surrounding the technology at that time. All the conferences, meetups, podcasts, and so on were about microservices, much like what we see with AI today. It was an exciting topic (buzzword?), and everyone was eager to jump aboard. But in reality, we had no idea what we were getting ourselves into.

The first attempt at microservices involved implementing new functionality as a separate deployment unit, with the monolith communicating via a REST API. On the surface, it didn't seem like a bad approach. However, no one had properly analyzed how this new service would interact with the many existing functionalities within the monolith. At that point, all we knew was that the service was meant to handle a completely new concept within the system, and isolating it as its own microservice seemed logical. So, we did as planned:



We ended up with a big ball of mud with a microservice glued onto the monolith that was responsible for managing the new concept. So far, not so bad, right? Well, that was just the beginning. It quickly became apparent that there were very few business scenarios in which this new concept could operate in isolation. As requirements unfolded, we discovered that many functionalities in the monolith depended on the data stored in the microservice for this new concept. This tight coupling meant that critical business operations couldn't function without the service, undermining the very goal of a microservices architecture.

Instead of simplifying our system, we introduced significant complexity. We had to deal with challenges such as HTTP communication and its associated issues, additional infrastructure requirements, and the need to adopt specialized tools for microservices, to name just a few. Yet, we didn't gain any real benefits. Should we implement the new concept within the monolith as a separate module? The solution would have been simpler: with fewer HTTP calls, reduced potential for errors, and faster delivery of the feature.

So, what went wrong? In my opinion, the root issue was a lack of proper analysis regarding the functionality. We failed to map the business boundaries and design the solution accordingly. As a result, we blindly pursued microservices without fully understanding the consequences. And we ended up with a distributed big ball of mud, instead of properly distributed microservices. We paid the price: longer development times, more errors, and overall headaches from overengineering.

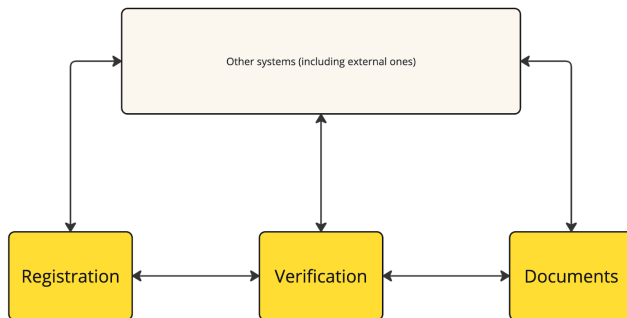
Before adopting microservices, I encourage you to spend time performing a deep analysis of how the projected business functionalities rely on each other, the type and degree of coupling they represent, and how it all translates into service boundaries.

Naive Boundaries Based on Identified Concepts

The second story involves similar issues, but this time, it didn't occur during the transition from a monolith to microservices. It happened while designing an entirely new part of the system. This component was responsible for registering and verifying business accounts on an e-commerce platform. During initial discussions with product owner, the following process stages were identified:

- Account registration - the user provides an email address, username, and password. Once submitted, the account is registered for the given email.
- Account verification - the user supplies company details, banking information, and completes a verification payment. At this stage, the company has limited access to the platform.
- Document submission - the user sends the required company documents to unlock all platform features.

For context, this solution was developed at a company that already had microservices and had well-established processes for quickly creating and deploying new microservices with all the necessary features (observability, service discovery, auditing, etc.). This environment created a strong temptation to implement each domain concept as a separate microservice, and that was the case here. It was decided to split the solution into three microservices: account registration, verification, and document handling.



Initially, apart from the overhead of HTTP communication and resource overuse, there weren't many drawbacks. However, problems soon began to emerge. The business process proved to be very unstable, requiring frequent changes, from minor adjustments to major variations depending on the registration country or company type. It quickly became apparent that the initially designed boundaries were far from ideal. Process changes required updates across multiple services, even when only one part of the process was affected (it was due to regional customizations). This interdependency made truly independent deployments impossible. Instead of enabling flexibility, changes in one service introduced new dependencies, leading to cascading failures.

So, what's the key lesson from this story? The simplest and more effective approach would have been to start with a modular monolith. That is, a single deployment unit composed of three modules. Over time, we could have observed its behavior to validate whether the proposed boundaries were correct. In our case, it became clear that the boundaries were far from perfect and didn't align with the business domains. Adjusting them within a monolith would have been much easier than doing so in a microservices architecture. Later, if necessary, those modules could be split into separate microservices, depending on other factors. In our situation, the solution would have worked perfectly as a single deployment unit.

Another issue I observed was a lack of deep understanding of the domain and its potential changes. We built a basic understanding based on a few discussions with the product owner, but that wasn't sufficient. The product owner often doesn't have complete knowledge of all aspects of the business and its domain. Instead, we should have organized analytical sessions with the business stakeholders (such as an Event Storming workshop) to gain a deeper understanding of the domain and its processes. This approach would likely have revealed that our initial implementation was merely a basic version of a process that could vary based on multiple factors, prompting a need for different boundaries and a revised design.

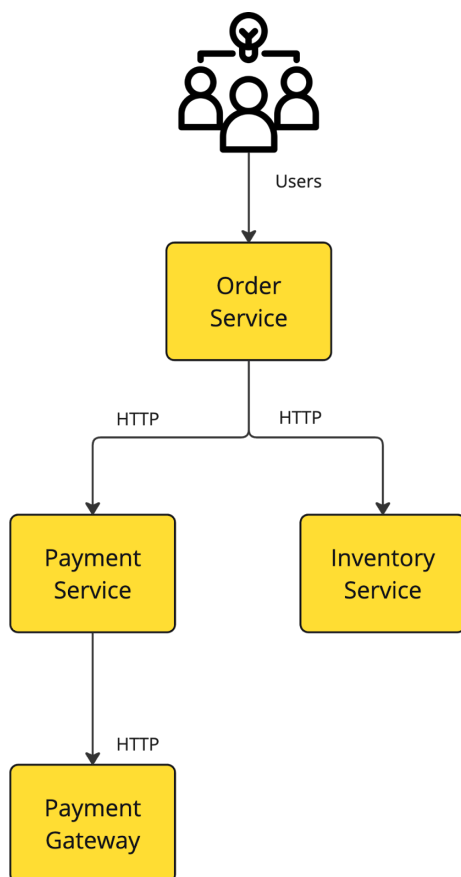
As I mentioned at the beginning of this chapter, I've encountered many real-life examples of similar issues where service boundaries were designed incorrectly, whether while creating a brand-new microservices solution or migrating a monolith to microservices. They all share a common root cause: a lack of proper domain understanding. This is why I always emphasize the importance of gaining a deep understanding of the business before writing any piece of code. In the microservices world, this is even more critical, as modifying service boundaries later is both difficult and costly.

Avoid Overuse of Synchronous Communication

When it comes to inter-service communication, we typically encounter two types: synchronous and asynchronous. Synchronous communication is deceptively simple, well-known, and often the first choice for integrating microservices. In contrast, asynchronous communication is more complex to implement and requires additional efforts in areas such as: observability, error handling, and managing eventual consistency, which is usually inherent in this approach. So, it's not for free, but in my experience, it should be the default choice for inter-service communication of push type (commands and events). Let's see why.

Synchronous Approach

What are the challenges associated with synchronous communication patterns in a microservice architecture? Consider, for example, a basic order creation process on an e-commerce website. In the happy path, an order is created only if the item is available in inventory and the payment is successful. Once the order is confirmed, the product is deducted from the inventory. Let's assume that this process is distributed across several microservices.



Synchronous communication between these services introduces several challenges:

- **Temporal coupling** - this occurs when one service requires another to be up and running simultaneously in order to correctly process business operations. In our scenario, if any microservice is unavailable or unable to process the request, the entire order creation process fails. This is a highly undesirable outcome for business operations. That was also the case in Project-Alpha, where the monolith required the microservice to process business operations.
- **Increased latency** - creating an order involves multiple inter-service connections, each adding overhead. As a result, the process is slower compared to a monolithic implementation. Website users don't like waiting.
- **Cascading failures** - if, for example, the Payment Gateway service encounters an infrastructure error (e.g. an inability to connect to the database), the error propagates from the Payment Service to the Order Service. Each service in the chain must handle the error. It results in unnecessary overhead.
- **Scaling challenges** - during peak periods (like Black Week), the system must scale to handle increased requests. With synchronous communication, scaling a single service is not always sufficient. Usually, all services involved in the synchronous communication chain must be scaled together, which is far less efficient.

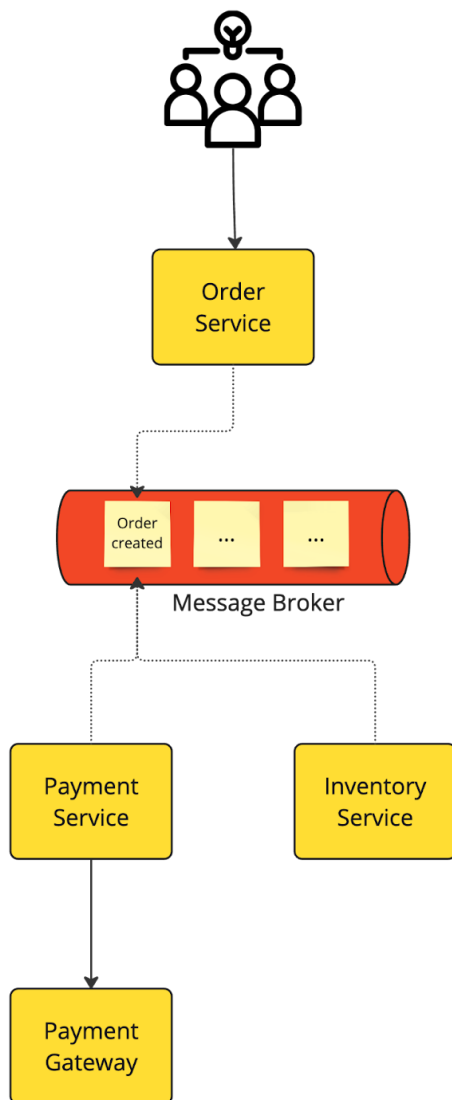
Additionally, due to the nature of distributed systems, transaction management becomes complicated in such cases. Imagine that we first deduct an item from the inventory and then attempt to process the payment. If the payment fails, whether due to technical issues or transactional problems (e.g., incorrect credit card information), we cannot easily roll back the changes in the database. However, this issue is not unique to synchronous communication, as it also occurs with asynchronous methods. Thus, this topic is not the focus of these lessons.

However, synchronous communication is not inherently bad, and in some cases, it's the right choice. First, it's simple to implement, and I dare say everyone knows how. It also works perfectly for queries when you need to retrieve data from another service.

Asynchronous Approach

Now, let's examine how asynchronous communication can help in this scenario. There are several ways to implement it, but for this example, I have chosen to use a message broker to transmit events from the Order Service to other services. In a happy path scenario, the order creation process might follow this flow:

1. When a user places an order, the Order Service creates the order and emits an OrderCreated event to the message broker. The user immediately receives confirmation that the order was created successfully.
2. The Payment Service listens for the OrderCreated event and processes the payment (using the Payment Gateway).
3. The Inventory Service also listens for this event and deducts the item from the inventory.



How does this solution address the aforementioned problems?

- **Decoupling services improves resilience:** eliminating the temporal coupling between the Order Service and both Payment Service and Inventory Service improves the system's resilience. In the event of a temporary outage of e.g., Payment Service, users can still create orders. The payment will simply be processed later once the Payment Service is restored.
- **Reduced latency for order creation process:** from the user's perspective, the process is much faster. There is minimal communication overhead, as the order only needs to be persisted in the Order Service's database and an event emitted.
- **Failures in downstream services are not propagated upstream:** for example, if the Payment Service encounters technical issues, the event processing can be retried without affecting the Order Service.
- **Each service can be scaled independently:** for instance, we can scale up the Order Service without having to scale the Payment Service. With a message broker serving as a buffer, the system can handle a higher volume of orders.

While this solution sounds promising, as always, there is no free lunch. Using this approach entails some costs and draw-

backs:

- **It requires a messaging infrastructure:** i.e., a message broker, which involves setup and maintenance costs. However, once established, it can be reused across multiple scenarios.
- **Message broker becomes a single point of failure:** if it goes down, all business operations halt. Special measures must be taken to reduce the likelihood of this event.
- **Error handling may be more complex compared to synchronous communication:** for example, if a payment fails because the user's credit card is locked, the Payment Service must communicate with the Order Service to cancel the order (for example, using another event).
- **Communication with users becomes more complicated:** if a payment fails, we must inform the user that the order has been canceled or that different credit card details are required, even though the order was already created and the user received confirmation. This typically necessitates asynchronous communication methods such as push notifications or emails.

Based on my experience with microservices, asynchronous communication may initially seem complex to implement and might require additional infrastructure, but in the long run, it delivers better resilience, system stability, and scalability. When combined with event-driven architecture, it becomes a powerful tool. In fact, I consider asynchronous communication the default choice for all push-type interactions (commands and events), resorting to synchronous methods only when there is a compelling reason to do so.

Again, you should analyze the feature you're about to develop to determine whether using asynchronous communication will pay off. In some cases, such as for non-critical paths or during prototyping, using async may result in overengineering. Last but not least, please avoid the trap of overusing async methods for queries. That's usually the wrong choice.

Missing or Inappropriate Tools

Do you remember "project-alpha" from the earlier lessons? That project taught me a lot. Another key lesson I want to share is that working with microservice architecture is extremely challenging without the right tools. Especially when migrating from a monolith to microservices, failing to adjust your toolbox for this architecture can turn your project into a nightmare. Let's explore this in more detail.

In the project discussed in this lesson, the decision was made to adopt microservices. You might recall that a new service was created alongside the monolith. You probably assumed it was deployed in Kubernetes or another orchestrator, as best practices show. However, that wasn't the case. At that time, a decision was made to develop a deployment solution involving Kubernetes, but for reasons unknown to me, it was continually delayed. This was also before cloud services like EKS became widely used. Nevertheless, the new service still needed to be deployed. Since the original monolith was deployed to a Tomcat server, we decided to deploy the new service as a separate .war file. The drawbacks of this approach: the lack of failure isolation and deployment independence, are self-evident.

But that was just the tip of the iceberg. Not only was the deployment platform missing, but we also lacked any log aggregation tools, not to mention tracing. Troubleshooting was done by directly accessing logs on the Tomcat server and jumping between different .war files and their instances - a truly terrible experience.

Moreover, the monitoring and alerting tools were not equipped to support microservices. In reality, we were completely blind and deaf when it came to understanding what was happening within the system, the communication between the monolith and the new microservice, its performance, and so on.

Additionally, there were no service discovery mechanisms in place. The monolith's instances had the hostname and port of the new microservice defined in their configuration separately for each instance.

I know this all sounds wild. However, I'm sharing these experiences to raise awareness about the problems caused by missing or inappropriate tools when adopting microservices. This architecture is far more complex than a monolith and requires more sophisticated tools for deployment, observability, maintenance, and more. If you decide to migrate to microservices, make sure to include adjusting or creating your toolbox in the roadmap.

Organizational and Process Changes Are Also Crucial

All of the lessons so far focus on a technical dimension. They are, to varying degrees, issues that software engineers can address or approach differently to ensure the successful implementation of microservices. Except in cases where employing this architecture causes indefensible overengineering.

However, there is another set of challenges that programmers cannot easily fix - let's call them organizational issues. These might include:

- flawed organizational processes
- an incorrect mindset among team members
- problems with team organization

Essentially, any factor unrelated to technology that prevents us from fully realizing the benefits of a microservice architecture. Throughout my journey with microservices, I have encountered several such challenges. In this chapter, I will describe two common ones.

Deployment Process from a Bygone Era

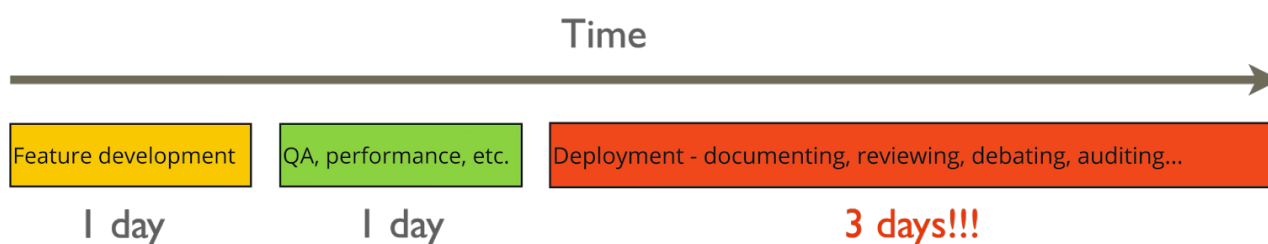
If you've been in the industry for say, 12+ years, you probably remember those "old days" of massive deployments. You and your team worked for several weeks or even months, and your code was deployed to production only once a quarter, semi-annually, or, in extreme cases, even once a year. But what a celebration it was!

Anyway, that era of producing software has faded into oblivion. Nowadays, we're agile. We create software iteratively, gather feedback from users and the business, and adjust the software in short cycles. The benefits of microservice architecture, such as the ability to execute independent deployments, break code into smaller projects, and isolate failures, truly support the modern approach to software development.

However, my experience shows that this isn't always the case. I once worked on a project for a large corporation, serving customers in nearly every country. Our goal was to build a data integration platform for various departments and systems within the company, as well as for integration with third parties. The data flew from one system - be it an internal department or an external partner - to another, there were thousands of such integrations.

From a technical perspective, using microservices for this project was the right approach. The platform handled enormous traffic - hundreds of thousands of users worldwide - with a critical need for scalability and failure isolation (an error in a single integration shouldn't affect others). There was also a demand for independent deployments of the integrations. The codebase was enormous, with 60 developers contributing to the platform's development.

In my opinion, the decision to use microservices to build this platform was a good one. What went wrong? The company implemented microservices without adapting the mindset and processes to fully leverage this architecture and its benefits. The biggest problem was the deployment process. You would expect that at such a scale with microservices, you could deploy new integrations or changes several times a day (or at least a week). Nothing could be further from the truth, as the deployment process was literally borrowed from systems that were deployed, let's say, only once a quarter. Even if the team could develop and test an integration in a day or two, the deployment process took a minimum of three days - yes, three days.



What caused deployment delays? To deploy a microservice (or a change to it), developers were required to prepare extensive documentation detailing the microservice and the integration it handled. This documentation was then submitted to a "Configuration Manager," who was responsible for reviewing it, conducting audits, and generating additional documentation. Finally, the changes were reviewed by a group with the final "go/no-go" authority over the integration.

One might argue that it's beneficial for changes to be audited, well documented, and confirmed by a dedicated review team, and I agree. However, this could be done far more efficiently without hindering the team's speed, causing frustration, and increasing time to market. We made multiple attempts to improve this process, but ultimately, it came down to the mindset of those in charge. They were unwilling to accept that things could be done more efficiently in a modern microservices architecture.

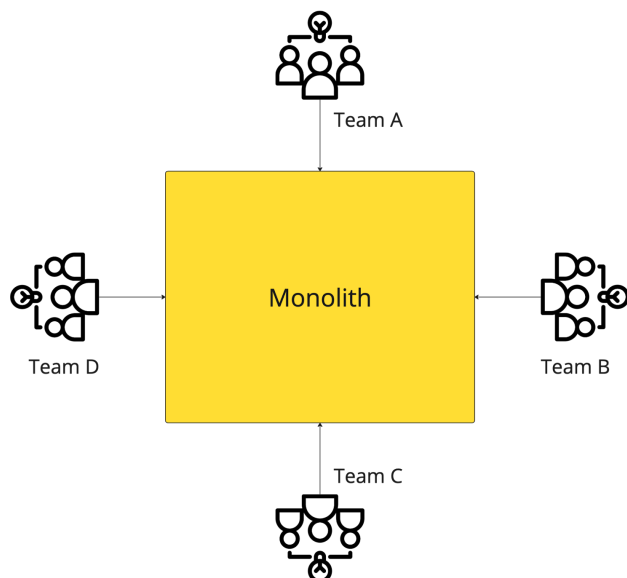
The lesson from this story is that unlocking the full potential of microservices requires not only technical changes to the architecture but also adjustments to the organization's processes and the mindset of its people. I also recommend measuring the process using, for example, the “deployment frequency” and “lead time for changes” DORA metrics to monitor its health and identify areas for improvement.

Inefficient Team Organization

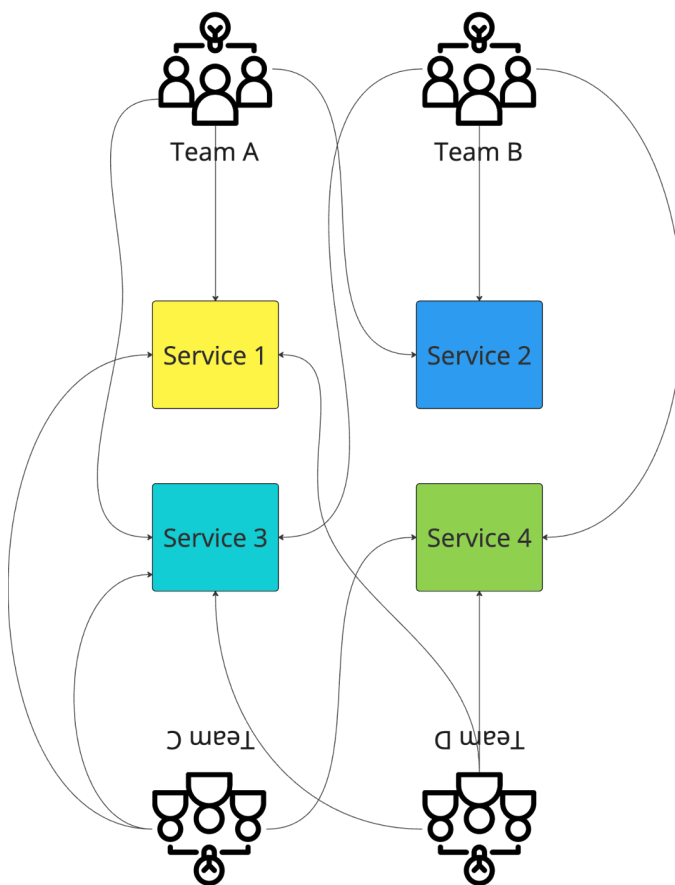
One of the greatest benefits of using microservices, in my opinion, is reducing programmers' cognitive load by modularizing the code into separate codebases and assigning specific teams to each microservice. As a programmer, you no longer need to understand and work with the entire codebase, as was necessary with a monolith (at least without well-suited modules). Instead, you focus on a specific piece of the solution. This approach truly improves team efficiency and enhances developer productivity.

However, when migrating from a monolith to microservices, it is crucial to carefully design not only the microservice boundaries but also the team organization and the responsibilities each team will have. It's very easy to simply transfer the mindset and workflow from a monolithic environment to a microservices architecture, which ultimately can make our lives more complicated rather than easier. In this lesson, I will share a personal story about how my team and I made this mistake and how it complicated our work.

Initially, the project was supported by a monolith with four teams contributing to it:



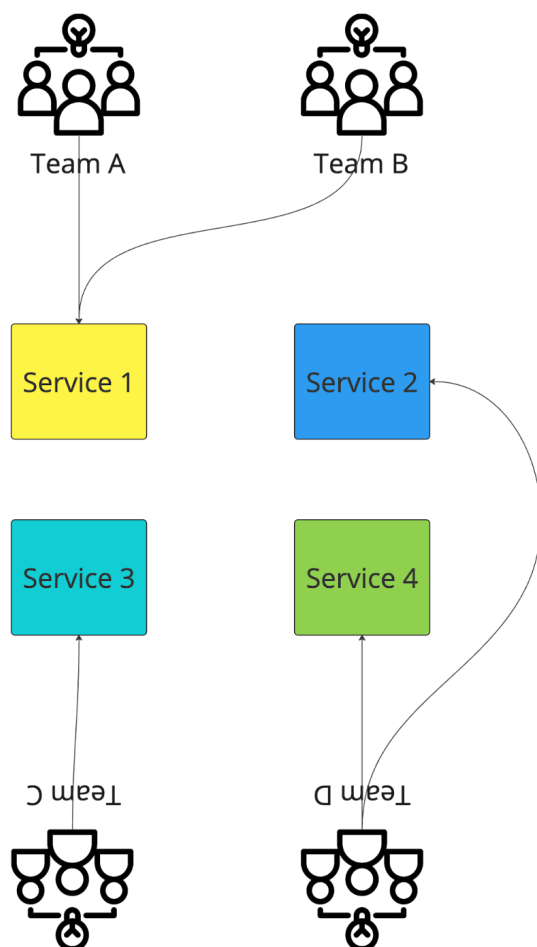
At some point, a decision was made to divide the system into microservices. This time, we approached it correctly by conducting a careful analysis of how to partition it into business modules and by using the Strangler Fig Pattern to iteratively extract these modules from the monolith. What we failed to do was redesign the team organization and reassign responsibilities after the migration. Instead of clearly defining each team's responsibilities by assigning one (or two) microservices per team, we simply replicated the monolithic solution in the new architecture. We ended up with something similar to this:



What problems did this cause?

- **Cognitive overload:** every developer still needed to understand all parts of the code, which became even more challenging as the code was split across multiple codebases.
- **Increased coordination overhead:** issues such as merge conflicts and deployment conflicts became more frequent.
- **Unclear separation of responsibilities:**
 - Which team should handle incidents occurring in a specific service?
 - Which team ensures quality and coding standards in a specific service?
- **Scattered business logic:** knowing all the microservices led to a tendency to spread the business logic across multiple services.

Working in such an environment was far from ideal, so we quickly realized that we needed to fix the issue. We reallocated responsibilities so that each team had a clear assignment for a specific module. Please note that the assignment is not necessarily “one-to-one,” which is perfectly acceptable when it is a conscious decision driven by technical requirements.



What's the lesson? It's not enough to design the microservice boundaries after migration, we must also plan how the teams will work with them. This planning should be done upfront to ensure success.

Takeaways

In this article, I shared the most important lessons from my professional experience with microservice architecture. Here are the key takeaways that should improve your experience with microservices:

- **Gain a deep understanding of the business** - thoroughly understand your business and its processes. This will help you design proper microservice boundaries and reduce the risk of being caught off guard by changes that could drastically alter your solution.
- **Start with a modular monolith** - before fully committing to microservices, consider starting with a modular monolith. It's much easier to adjust boundaries when the modules are not distributed.
- **Justify the use of microservices** - ensure that microservices are the right choice for your project. Collect and confirm the driving factors behind their adoption, and don't blindly chase buzzwords.
- **Equip yourself with the right tools** - invest in the right solutions early on. Working with microservices without the proper tools and an efficient infrastructure can be extremely challenging.
- **Avoid overusing synchronous communication** - while it is easy to implement, excessive reliance on it can introduce unnecessary complexity.
- **Rethink challenging processes** - identify and refine any processes that might complicate working with microservices.
- **Restructure teams if necessary** - consider reorganizing teams and their responsibilities to enhance efficiency in a microservices environment.

Remember, microservice architecture is just a tool. It should assist you, not hinder you.

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

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



How Quarkus Seamlessly Combines Architecture, Performance, and Cloud-Native

Architecture

DevOps

Efficiency

Quarkus

Author: Holger Tiemeyer

Responsibility, pragmatism, domain knowledge, and professionalism. These are the core values and attributes that characterize successful software architects. Based on many years of project experience, Holger Tiemeyer imparts these skills and knowledge in his training courses, speaks about them at conferences and at Universities and publishes articles in specialist magazines. As a computer scientist with a minor in psychology, he volunteers at iSAQB e.V.



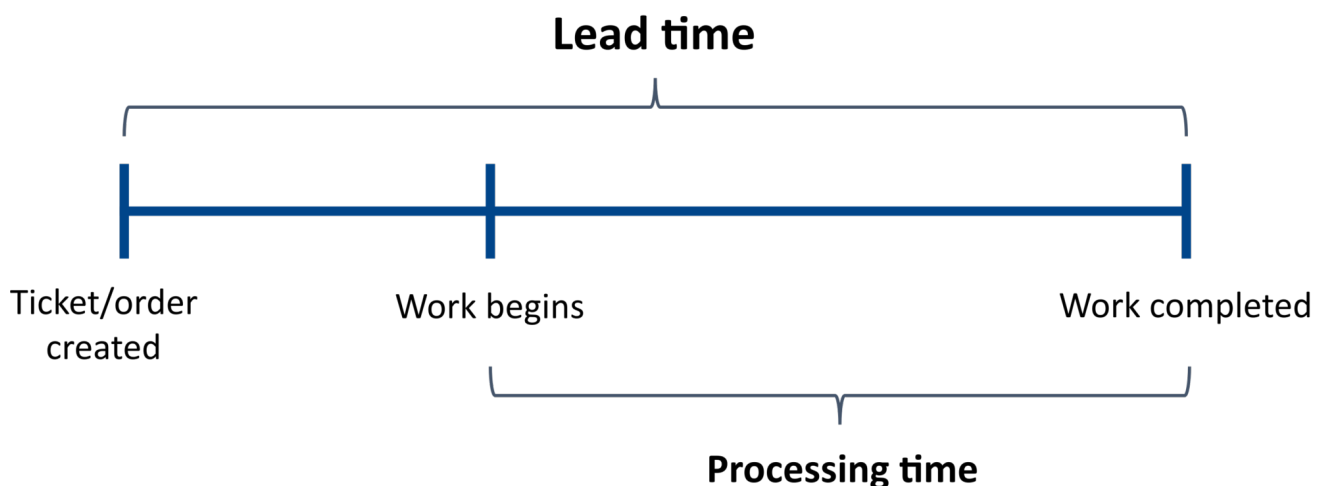
LinkedIn

Given the increasing ubiquity of digital technologies, which are becoming increasingly and naturally embedded in our environment, the essential question arises as to what role Java will play in this future development of IT systems. Mark Weiser, pioneer of ubiquitous computing, described this technological paradigm shift as follows: “The best technology is the one we don’t even notice because it naturally enriches our everyday lives.” (paraphrased from [LMU19]).

Ubiquitous computing is already demonstrating how technology is becoming omnipresent and seamlessly integrated into our environment as an invisible companion. A key factor here is the almost universal access to networks (such as Wi-Fi), which is increasingly becoming standard infrastructure internationally. This access enables the real-time networking of smart devices, from smart homes to wearables, and forms the basis for technologies that work intuitively in the background and communicate with each other seamlessly without disruptive interference. In addition, ubiquitous computing promotes an ever-closer **sympiosis between humans and machines**, in which new technologies are emerging that are increasingly replacing classic interfaces to the digital world such as smartphones and tablets. Devices such as mixed reality glasses (e.g. *Meta Quest* or *Apple Vision Pro*) open up a new form of interaction in which digital information is projected directly and contextually into the field of vision. The goal is no longer just general interaction, but rather the precise adaptation of digital content to the individual needs of the user. This development leads to the concept of **hyper-personalization**: systems respond in real time to location, preferences, and goals, providing tailored information, entertainment, or support in everyday life. This points to a future in which digital experiences are seamlessly integrated into personal life.

In order to survive in an increasingly dynamic market environment, companies and organizations must no longer simply react but actively shape the future. To this end, **short innovation cycles** and **maximum speed and efficiency** in the provision of new services are becoming key success factors. However, it is precisely this requirement that presents modern software development with a familiar dilemma: developers need flexibility and speed to respond agilely to new requirements, while operations must ensure security and stability. This tension is becoming increasingly apparent, especially in an era dominated by digital platforms and cloud ecosystems. **DevOps** attempts to resolve this divergence by connecting development and operations through shared processes, close collaboration, and automation. Nevertheless, the central challenge remains: optimizing the so-called **lead time** of a requirement from idea to productive deployment.

Figure 1 shows a horizontal timeline representing the course of a development process. It is divided into two main sections: **lead time and processing time**. Lead time covers the entire period from the creation of a requirement to its productive deployment. Within this span, processing time marks the part in which the actual implementation, i.e., the active processing of the requirement, takes place. The remaining time consists of waiting and coordination phases that prolong the process.



As market dynamics increase, the ability to minimize lead time continues to gain importance. Modern cloud infrastructures form the foundation for this: they ensure stability, security, and flexibility and enable innovations to be implemented more quickly and reliably. The key here is to strike a balance between high development speed and consistent quality. The **Reactive Manifesto** (see [REA13]) provides guidance in this regard. It serves as a model for modern, networked software architectures and defines four central, fundamental principles for cloud-native applications and, in particular, for microservice-based systems:

- **Message orientation:** Systems communicate asynchronously and use messages to scale independently and respond flexibly to events.

- **Resilience:** Applications remain stable even under load and in the event of errors, as they rely on self-healing mechanisms and isolation.
- **Elasticity:** Systems scale dynamically with the load, both horizontally by adding components and vertically by intelligently allocating resources (e.g., replicas, etc.).
- **Responsiveness:** Applications respond to user requests and events in real time, enabling immediate and consistent user experiences.

Building on these principles, **reactive programming** has established itself as a programming paradigm that was developed specifically for processing asynchronous events and building high-performance, scalable systems. In contrast to traditional imperative programming, where commands are executed sequentially and blocking, reactive programming relies on a non-blocking, event-driven model. Data flows and changes are treated as streams that are processed as soon as new events arrive. This approach enables applications to use resources more efficiently and respond more dynamically to peak loads, as tasks no longer have to wait for each other. Reactive programming offers significant advantages, especially in the context of modern microservices architectures, by supporting technologies such as elastic scaling and resilient transactions. It thus forms a central bridge between the requirements of the Reactive Manifesto and the practical implementation of modern cloud-based systems.

The interplay between cloud infrastructures, the principles of the Reactive Manifesto, and reactive programming has a clear objective: to create efficient, resilient, and scalable systems that meet the demands of modern markets. To guarantee both speed and stability, the focus is on the key components of the cloud-native paradigm: DevOps, microservices, containerization, and continuous integration and continuous delivery (CI/CD). These pillars form the basis for agile and highly available IT landscapes that can not only be deployed more quickly but also respond dynamically to changing loads and requirements while enabling future paradigms (see above).

But it was precisely at this point that Java, long the dominant language for (large) companies and organizations, showed weaknesses. While it impressed for decades with its stability and broad support, its use in modern, **container-based environments** such as **Kubernetes** increasingly proved to be a challenge. Long start-up times, high resource consumption, and deployment complexity made life difficult for Java-based applications in a world of microservices and dynamic autoscaling. These disadvantages led developers and architects to frequently rely on non-hierarchical programming paradigms or lean frameworks to better meet the requirements of the cloud-native era.

From Weakness to Strength: How Quarkus Is Redefining Java

Quarkus, a framework developed by Red Hat specifically with containerized cloud-native applications in mind, addresses precisely this issue. The aim of this framework is to retain the advantages of Java while overcoming its weaknesses in modern infrastructures, thereby optimizing it for the needs of the cloud-native era. It thus provides an answer to the central conflict of objectives in DevOps: speed in development without compromising the stability or security of the system. Applications are prepared as far as possible at compile time, for example by Quarkus shifting decision-making processes such as dependency injection or annotation scans from runtime to build time, enabling faster, more resource-efficient applications. The result: fast start times, minimal resource consumption, and a drastic reduction in overhead costs. This is a decisive advantage, especially for containers that are frequently started and stopped to meet the requirements of an elastic infrastructure.

With support for **GraalVM**, Quarkus also enables applications to be compiled as **native images** (not to be confused with container images). These images are highly optimized binary files that do not require a JVM and are perfectly tailored for containerized environments: The startup time of such applications is often in the millisecond range. Additionally memory consumption is significantly reduced.

Traditionally, Java code is compiled into bytecode, which is then interpreted by the JVM at runtime or optimized through just-in-time compilation (JIT). This means that the final machine code generation (i.e., the executable code for the operating system and hardware) takes place while the application is running. This approach is flexible, but it costs startup time and resources because the JVM must first be loaded and initialized.

When creating a native image with GraalVM, on the other hand, the Java code is not only compiled to bytecode, but is completely converted to executable machine code at build time. The result is a standalone, native binary file that can be executed directly by the operating system without the need for a JVM. This is possible because native image compilation statically analyzes and integrates all classes, methods, and frameworks required at runtime.

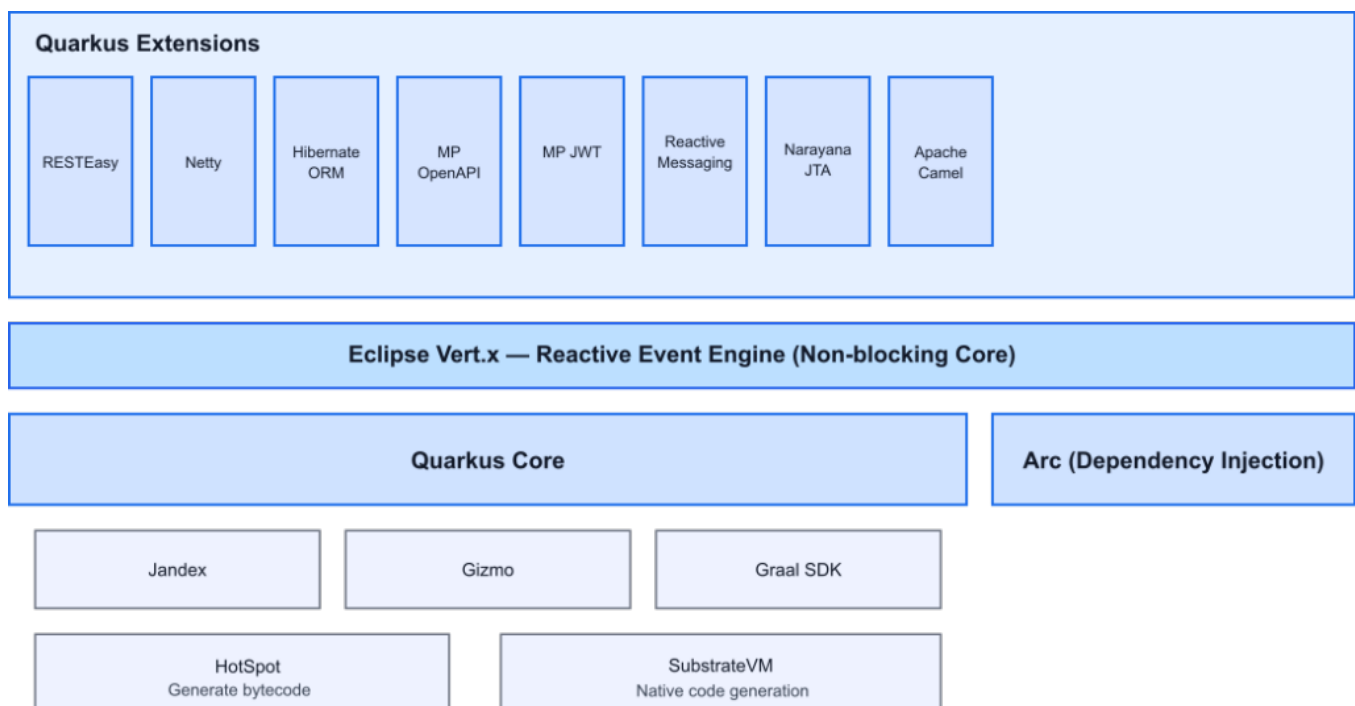
A native image can be integrated into a container image as an executable, native binary file. The advantage of this is that the container image has far fewer dependencies. For example, the image does not need a JVM because the native image

can already execute the Java code without it. This results in smaller container images, faster container starts, and lower resource consumption, which is essential in highly scalable and flexible environments such as Kubernetes.

A Look at the Basic Principles and Structure of Quarkus

To fully leverage the potential of Quarkus, it is worth taking a look at the inner workings of this modern framework. A solid understanding of how it works enables targeted, efficient use and tailored adaptation to individual requirements. Quarkus is based on two central principles: **first-class Kubernetes-native Java** and **developer joy**. The first principle emphasizes high performance, smooth integration into modern cloud environments, and optimized containerization. The second focuses on the developer's experience by promoting a more productive and enjoyable way of working.

At the heart of the Quarkus architecture are two closely linked elements: **Eclipse Vert.x** and the **Quarkus Extensions**. Together, they implement the principles of the Reactive Manifesto and create a platform that offers both high flexibility and exceptional performance.



Quarkus Core forms the technical foundation. It includes key tools such as **Jandex**, a framework for rapid analysis and processing of annotations in bytecode, and **Gizmo**, a library for bytecode generation during the build. This is complemented by the **Graal SDK**, which prepares applications for the creation of native images. These components enable Quarkus to analyze, adapt, and optimize code as early as the build time.

Building on this, **Eclipse Vert.x**, as a non-blocking, event-driven runtime, ensures the reactive performance of Quarkus. It enables asynchronous processing of data streams and events, allowing resources to be used optimally and processes to be executed in parallel. Requests, data flows, and messages are treated as continuous streams that are processed in real time. Quarkus abstracts the technical complexity and provides developers with a uniform, reactive programming base that combines high performance with ease of use.

The architecture is rounded off by the **Quarkus Extensions** system, which simplifies the integration of external technologies and services (such as databases, security solutions, or monitoring tools). They standardize the handling of these components, automate configurations, and shift many tasks to the build phase. This results in lightweight, optimized applications that offer maximum flexibility in development mode and are designed for stability and efficiency in production mode. These extensions form the crucial link between development and operation and play a key role in seamlessly uniting both worlds. Their behavior adapts dynamically to the respective context of use. This clear separation between development mode and production mode is one of the central principles of Quarkus and ensures a balanced interplay of agility and stability.

In development mode, the focus is on speed, flexibility, and immediate feedback to promote the creative process and short iteration cycles. Production mode, on the other hand, is designed for efficiency, reliability, and optimal runtime conditions.

This dual focus forms the basis for consistent deployments and a smooth transition from development to operation. It is therefore in line with modern DevOps practices.

Below, we take a detailed look at both modes to show how Quarkus specifically supports different requirements at each stage of the software lifecycle.

Extensions in Dev Mode: Efficiency Through Automatic Container Provisioning

One of Quarkus' most outstanding features in development mode is its ability to use Dev-Services to automatically containerize and provision external services such as databases (e.g. PostgreSQL), identity providers (e.g. Keycloak), or message brokers (e.g. Kafka) and more if they are not already available. This principle was developed to provide developers with a ready-to-use environment and relieve them of the time-consuming initialization and configuration of such services.

As soon as a Quarkus extension is integrated, for example the database extension `quarkus-jdbc-postgresql`, Quarkus analyzes whether the required resource is already available. If, for example, an active PostgreSQL instance is missing, Quarkus automatically starts a suitable Docker container that provides the resource. The runtime environment is orchestrated with **Docker** (or **Podman**), provided these are installed on the development machine. The connection to the started container instance is also configured automatically. Developers only need to focus on their application, as Quarkus takes care of the entire provisioning and configuration.

Communication with the launched container takes place via the usual Quarkus configuration mechanisms, such as the `application.properties` file located in the `src/main/resources` directory of the Quarkus project. This file allows parameters such as database URLs or access data to be defined dynamically. Quarkus automatically updates the configuration, eliminating the need for manual adjustments or restarts. This ensures a smooth development process and promotes productivity.

Another advantage of this integration in dev mode is the consistent environment provided by Quarkus. Developers can be sure that the automatically started services accurately reflect later production conditions such as database versions or API standards. This minimizes conflicts that may arise later on from the outset.

On a technical level, the behavior of Dev-Services can be controlled via specific configuration parameters. For example, it is possible to disable Dev-Services if necessary and to use an existing infrastructure instead:

```
quarkus.datasource.devservices.enabled=true
```

This configuration value is specified in the `application.properties` file. The default value is `true`.

In Quarkus' Dev mode, not only is development supported by the automatic provisioning of services via Dev-Services, but application testing is also seamlessly covered by the integration of Testcontainers (see [TCS23]). Testcontainers is a popular Java library that allows tests to be run in realistic, container-based environments without the need for additional manual steps. Quarkus automatically starts the necessary containers for external services such as databases or message brokers, creating a production-like, reproducible test environment. This greatly simplifies development and testing processes, as there is no need to maintain a separate test infrastructure.

In Quarkus, configurations can be specified for specific runtime profiles, such as `dev`, `test`, or `prod`. This is done using **profile prefixes** such as `%test` in `application.properties`. For example, if a PostgreSQL database is required in test mode, the following setting is sufficient to automatically provide it via test containers:

```
1 %test.quarkus.datasource.devservices.image-name=postgres:16
2 %test.quarkus.datasource.username=user
3 %test.quarkus.datasource.password=password
4 %test.quarkus.datasource.db-name=testdb
```

With this configuration, Quarkus only automatically starts a PostgreSQL test container during test execution. In development or production mode, the regular database configuration remains active. This ensures a clean separation of environments to produce reproducible and stable tests.

Extensions in Production Mode: Focus on External Service Provision

While dev mode enables the automatic provisioning of containers and services, Quarkus behaves differently in production mode. In this mode, the focus is on ensuring that external services, such as databases, identity providers, or message bro-

kers, are already available in the target environment. In production mode, Quarkus does not manage container instances itself, but ensures that the application can interact seamlessly with the deployed services.

Configuration is also done via the `application.properties` file or environment variables, allowing applications to adapt flexibly to different environments. For example, cloud-native mechanisms such as ConfigMaps or Secrets can be used in Kubernetes to provide access data or connection endpoints. Once the container infrastructure is ready, Quarkus automatically accesses these resources. This ensures the stability of productive environments and, at the same time, prevents Quarkus applications from relying on potentially unsecured containers.

Another advantage of Quarkus is its ability to automatically generate IaC (**Infrastructure as Code**) scripts for production environments. Using specific extensions, such as the Kubernetes or OpenShift extension, Quarkus creates YAML files that describe the infrastructure required for production.

Developers can insert these generated YAML files directly into their continuous integration and deployment (CI/CD) pipelines, which significantly simplifies and accelerates the transition from development to production mode.

The Quarkus extensions are closely integrated with the build-time optimization process, analyzing source code, incorporating necessary dependencies, and automatically performing configuration tasks. Unnecessary components are removed during the build, making applications more compact and resource-efficient. This means developers have less technical complexity to deal with, while Quarkus optimizes performance and trims the code for efficiency in the background.

The integrated development tools, which consistently implement the concept of developer joy, make a significant contribution to high productivity. Live coding is particularly noteworthy: changes to the code or configurations are applied almost in real time without the need for a restart. This significantly shortens feedback cycles and keeps the focus on the actual logic of the application rather than the build process.

Another element of flexibility is the `application.properties` file, which is used to define central settings such as ports, database connections, and messaging parameters. It unleashes its full potential in containerized environments: configurations can be dynamically adjusted via environment variables, enabling smooth transitions between development, test, and production environments. This virtually eliminates the need for time-consuming manual adjustments during deployment.

This automation continues with tight integration with Docker and Kubernetes. Quarkus automatically generates appropriate container images and deployment manifest files during the build. This means that applications are delivered ready for containers. This is a significant advantage for continuous integration and deployment (CI/CD) processes, where speed and reliability are crucial.

Despite its modern, reactive, and cloud-native orientation, Quarkus remains fully compatible with established Java standards such as **Jakarta EE** and **MicroProfile**. Companies can continue to use existing architectures and code bases and modernize them step by step without having to completely change their technology.

With this combination of modular architecture, build-time optimization, reactive runtime environment, and developer-friendly tools, Quarkus sets a new standard for cloud-native Java applications. It combines speed with stability, reduces complexity, and provides a well-designed foundation for modern software development. Quarkus makes Java efficient, scalable, and ready for the demands of next-generation distributed systems.

Reactive Development with Quarkus in Dev Mode

The following application example shows how to create a simple task management system (to-do app) based on the principles of reactive development from the outset. The example describes the implementation of a small REST API that allows users to create, view, and delete tasks (e.g., to-do entries). Instead of tolerating blockages caused by classic, synchronous database and API calls, Quarkus' reactive capabilities are used to process all requests efficiently and non-blockingly. Hibernate Reactive serves as a tool for communicating with the database, and Quarkus' reactive APIs (e.g., RESTEasy Reactive) ensure that HTTP methods such as `GET` or `POST` are executed completely asynchronously.

The following explanations assume that Java is available, Quarkus is already installed, and Docker is active on the development system, as Quarkus' Dev-Services will automatically provide the PostgreSQL database as a container.

The objective is as follows: A reactive REST API that manages tasks and interacts asynchronously with a PostgreSQL database is to be created. All responses from the API and database operations are reactive, thus conserving the application's resources.

In the following, the Quarkus CLI is used to create a new project with the necessary reactive extensions. These extensions enable us to use the desired reactive functionality for REST, Hibernate, and the database:

```
1 quarkus create app com.example.todo-reactive --extensions="resteasy-reactive, hibernate-reactive-panache, reactive-pg-client"
```

- **resteasy-reactive**: Provides a reactive implementation of JAX-RS for RESTful APIs.
- **hibernate-reactive-panache**: Enables non-blocking ORM functionality based on Hibernate.
- **reactive-pg-client**: Provides a reactive PostgreSQL client to execute queries asynchronously.

In the next step, a database entity `src/main/java/com/example/todo/Task.java` is created, which can be processed with Hibernate Reactive. Thanks to Panache extensions, working with database models is clearly structured and offers reactive support. The source code for this entity is as follows:

```
1 package com.example.todo;
2 //SNIPP
3 @Entity
4 public class Task extends PanacheEntityBase {
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     public Long id;
8     public String description;
9     public boolean completed;
10 }
```

- **PanacheEntityBase** offers a range of predefined methods such as `persist()` or `findAll()`, which automatically return reactive streams and operate in a non-blocking manner.
- Hibernate Reactive ensures that all database calls are executed asynchronously and in a resource-efficient manner.

The REST controller `src/main/java/com/example/todo/TaskResource.java` handles requests such as retrieving all tasks, adding new tasks, and deleting individual entries in a fully reactive manner.

```
1 package com.example.todo;
2 import io.smallrye.mutiny.Uni; // Reactive API for asynchronous data processing
3 //SNIPP
4 @Path("/tasks")
5 @Produces(MediaType.APPLICATION_JSON)
6 @Consumes(MediaType.APPLICATION_JSON)
7 public class TaskResource {
8
9     @GET
10     public Uni<List<Task>> listAll() {
11         return Task.listAll(); // Returns a reactive Uni instance
12     }
13
14     @POST
15     @Transactional
16     public Uni<Task> create(Task task) {
17         return task.persist().replaceWith(task); // Persists asynchronously and
18             returns the task
19     }
20
21     @DELETE
22     @Path("/{id}")
23     @Transactional
24     public Uni<Void> delete(@PathParam("id") Long id) {
25         return Task.findById(id).onItem().ifNotNull().transformToUni(task ->
26             task.delete()).replaceWithVoid(); // Deletes the task asynchronously
27     }
28 }
```

As you can see, the logic of this controller is based on **Uni**, a class of the Mutiny API used for asynchronous and reactive programming in Quarkus. All method returns are non-blocking and are processed asynchronously by Hibernate Reactive.

The `application.properties` file controls the automatic configuration. For development, a PostgreSQL container is automatically started using Dev Services:

```
1 quarkus.datasource.db-kind=postgresql
2 quarkus.datasource.devservices.enabled=true
3 quarkus.hibernate-orm.database.generation=drop-and-create
```

- Quarkus recognizes from the configured data source that a PostgreSQL database is required and automatically starts a suitable container.
- The `drop-and-create` setting creates the database fresh for each new session, which is helpful during development.

The application can now be started in dev mode: `./mvnw quarkus:dev` Quarkus automatically starts the application and PostgreSQL container through Dev Services. Changes to the code are applied immediately without the need to restart.

This example shows how easy it is to design a fully reactive application with Quarkus. Thanks to Hibernate Reactive, Panache, and RESTEasy Reactive, all requests are processed asynchronously, which not only increases efficiency but also ensures a scalable architecture. Combined with Dev Mode and automated Dev Services, Quarkus offers a user-friendly platform for modern, reactive software development.

The Strategic Advantages of Quarkus

The advantages of Quarkus go far beyond technical details. They have a direct impact on economic efficiency and competitiveness. The fast start-up time and low resource consumption significantly reduce operating costs. This is a crucial factor in cloud environments, where companies often pay according to a pay-as-you-go model.

In addition, Quarkus integrates seamlessly into common DevOps processes. The build time optimizations are a natural advantage in the context of CI/CD pipelines, as they eliminate long start times for tests and deployments. The ability to use GraalVM Native Images is also a perfect fit for serverless architectures, where short start times are a prerequisite for cost efficiency.

Last but not least, Quarkus offers developers a platform for turning innovative ideas into reality more quickly. Minimizing overhead in the development phase and seamless integration into cloud-native infrastructures such as Kubernetes give teams the flexibility they need to implement rapid iterations and experimental projects. Quarkus not only overcomes the barriers of the traditional Java world, but also proves to be a strategic tool that helps companies remain competitive in the age of microservices, containerization, and cloud computing.

Quarkus is therefore much more than a framework. It is a powerful tool for those who want to lead Java into the future. And it forces companies and developers alike to rethink their assumptions about Java. Quarkus impressively demonstrates that even an aging ecosystem like Java can always be reinvented to meet the demands of the next technological era.

Literature & Links

[LMU19](#)

[REA13](#)

[TCS23](#)

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



Improving Platform Observability with Distributed Tracing and OpenTelemetry

Observability

Distributed Tracing

Author: Alexandre Touret

With over two decades of experience as a seasoned software architect, Alexandre Touret is fueled by a passion for technology, cloud computing, and software development. Throughout his career, he has designed, built, and managed systems of varying scales across a diverse range of industries, including insurance, financial services, and mobility.

He takes great pleasure in sharing his insights and experiences with others, whether by teaching, speaking at conferences, organizing hands-on workshops, or writing articles. By engaging with the community, he aims to inspire and provide practical insights drawn from his hands-on experience.



LinkedIn

Observability has become crucial for modern applications. Distributed tracing, in particular, is essential for diagnosing issues within complex transactions. OpenTelemetry has emerged as the indispensable standard in this field. This article explores the fundamentals of distributed tracing, its implementation, and its impacts.

Distributed Traces: An Essential Pillar

In a context where applications are increasingly complex, it is crucial to be able to observe each transaction in detail to quickly identify problems and ensure continuous evolution of the platform. Among the three main pillars of observability—logs, metrics, and traces—traces are often less known but equally essential.

Unlike logs, traces allow tracking the performance of transactions in production and identifying contention points. They also provide the ability to follow synchronous transactions (such as HTTP or SQL requests) or asynchronous ones (such as JMS or Kafka messages).

History and Evolution

In 2010, Google published a case study on their use of Dapper¹: their distributed tracing infrastructure. This platform addressed a pressing need to correlate different transactions with each other and facilitate their end-to-end visualization in increasingly distributed and complex platforms.

Thanks to this work, open-source specifications and projects came into being. Initially, one of Dapper's creators, Ben Sigelman, created the open-source project OpenTracing, which was later integrated into the OpenTelemetry project.

The ecosystem continues to grow. Beyond the major cloud providers offering their solutions, there are many open-source or proprietary products that cover all or part of the scope: Jaeger, Grafana, or Elastic APM.

A summary can be found at the end of this article.

Some Basic Principles

The various mechanisms related to traces revolve around the following concepts:

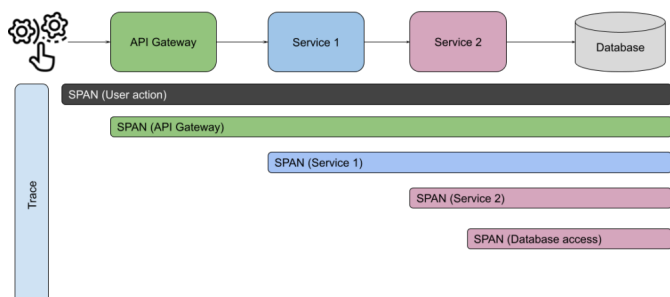
- SPANS
- TRACES
- Context Propagation

SPANS

SPANS represent an operation such as a method call, a REST call, or a database query. They are identified by a header (`SPAN_ID`) and can have a parent-child relationship (e.g., a method called by another method) through the declaration of a header (`PARENT_SPAN_ID`).

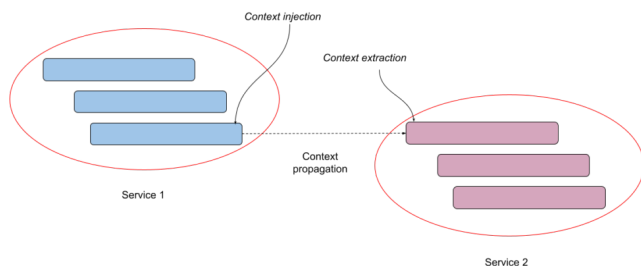
TRACES

Traces are a collection of SPANS that represent an end-to-end transaction.



Context Propagation

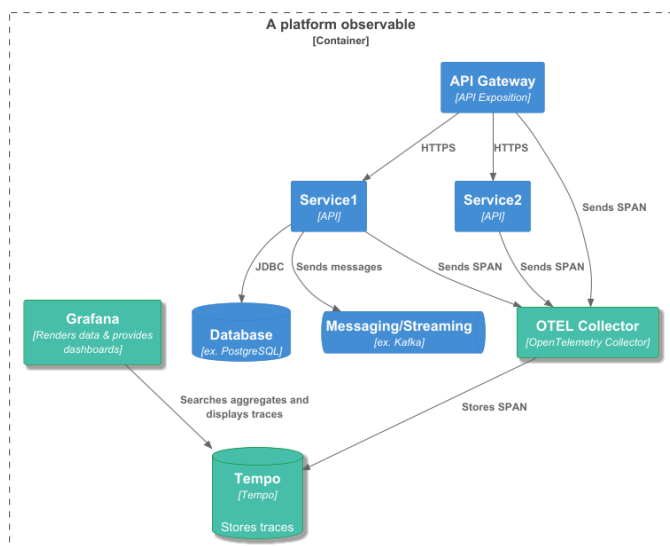
How can operations across different applications and processes be correlated? To address this question, all the concepts mentioned below are automatically propagated.



In this way, the called services automatically have the information needed to retrospectively link them to the calling services. To ensure the portability of this information, various tools and infrastructures rely on the W3C Trace-Context standard².

The Architecture

The typical architecture to implement looks like this:



In green, you will find the modules necessary for processing, storing, and displaying traces. All application modules (shown here in blue) send TRACES and SPANS to the OpenTelemetry collector. This collector then stores the data in the database (shown here as Grafana Tempo).

Implementation and Data Broadcast

TRACES and SPANS are created, distributed, and sent by OpenTelemetry, which is compatible with many platforms³: Java, NodeJS, Python, Go, or Rust.

Two transmission protocols are available: HTTP and GRPC.

When integrating into your programs, three implementations are possible: using the API (and coding these features), using the library for a “zero code” approach, or instrumentation.

Depending on the programming languages, you have several options available.

Each option has its advantages and disadvantages, which need to be evaluated based on your technical and organizational context.

The use of a library offers a quicker start and is essential for native compilation (Java with GraalVM, Go, or Rust).

Instrumentation, on the other hand, involves installing an agent that will be passed as an argument to the software at startup.

An example in Java:

```
1 $ export JAVA_OPTS=-javaagent:/opt/agent/javaagent.jar $ java $JAVA_OPTS -jar myapp.jar
```

In my opinion, this option provides more flexibility in the software lifecycle.

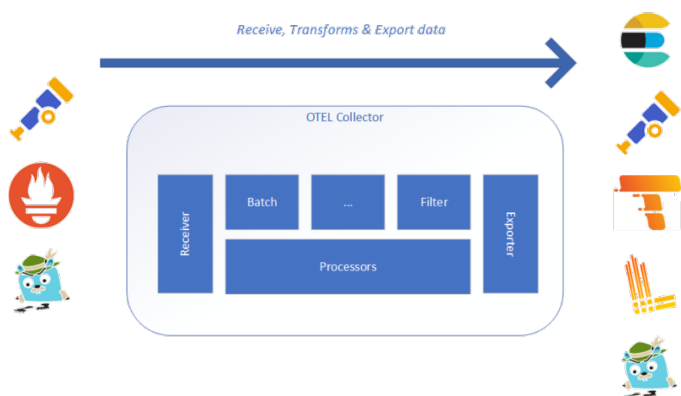
Indeed, beyond the fact that it works “out of the box,” it allows you to decouple the lifecycle of the functional deliverable from that of the agent. For example, if the agent needs to be updated, there is no requirement to re-deliver the business deliverable.

In both cases, it will still be necessary to configure the properties needed for the dissemination of SPANS and TRACES through environment variables:

```
1 OTEL_SERVICE_NAME: api-gateway OTEL_EXPORTER_OTLP_ENDPOINT: http://collector:4317
  OTEL_EXPORTER_OTLP_PROTOCOL: grpc OTEL_RESOURCE_ATTRIBUTES: "source=agent"
```

Aggregation

You might have noticed the URL mentioned in the above configuration. Indeed, all observability-related streams, particularly traces, pass through a collector. OpenTelemetry provides a default one. There are other implementations as well, such as Grafana Alloy.



This tool can be compared to an ETL (Extract, Transform, Load)⁴. It provides an interface for all observability-related sources, transforms and aggregates them, and finally sends them to a backend (e.g., a database).

Here is a deliberately simplistic configuration to collect traces and store them in the Grafana Tempo database:

```

1 // (1)
2 receivers:
3   otlp:
4     protocols:
5       grpc:
6         endpoint: 0.0.0.0:4317
7       http:
8         endpoint: 0.0.0.0:4318
9 // (2)
10 processors:
11   batch:
12 // (3)
13 exporters:
14   otlp:
15     endpoint: tempo:4317
16
17 // (4)
18 extensions:
19   health_check:
20   pprof:
21   zpages:
22
23 // (5)
24 service:
25   extensions: [health_check, pprof, zpages]
26   pipelines:
27     traces:
28       receivers: [otlp]
29       processors: [batch]
30       exporters: [otlp]

```

Explanation:

1. Declaration of interfaces where data is transmitted by all applications
2. Declaration of actions to be performed
3. Declaration of output interfaces (here, the database accessible via the URL tempo)
4. Declaration of various extensions
5. Declaration of the action pipeline to be executed

Now, let's imagine we want to deepen this process. For example, if we wish to systematically exclude certain URLs (e.g., /health or /metrics), it is possible to do so within the collector through tail-sampling in the collector:

```

1 processors:
2   tail_sampling:
3     decision_wait: 10s
4     num_traces: 100
5     expected_new_traces_per_sec: 10
6     decision_cache:
7       sampled_cache_size: 500
8   policies:
9     [{
10       name: test-policy-9,
11       type: string_attribute,
12       string_attribute: {key: http.url, values: [\\health, \\metrics],
13         enabled_regex_matching: true, invert_match: true}
14     }]

```


Once the processor is declared, you just need to add it in the pipeline declaration to activate it:

```
1 service:
2   extensions: [health_check, pprof, zpages]
3   pipelines:
4     traces:
5       receivers: [otlp]
6       processors: [batch,tail_sampling]
7       exporters: [otlp]
```

For more details on the available options, you can consult the official documentation and the GitHub repository.

Storage

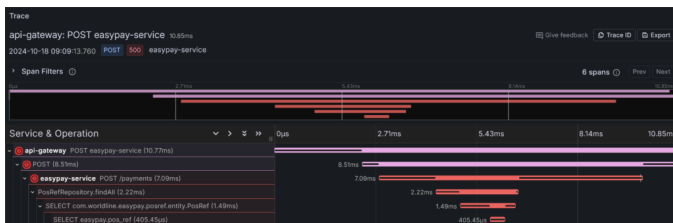
Traces and SPANs are automatically broadcasted and aggregated on a platform (we will delve deeper into implementation later). This platform allows for storing, visualizing, and correlating all the information. Among the solutions available in the market, there's the open-source Grafana platform, which covers the entire spectrum.

For instance, the Tempo database allows for the storage of traces, and Grafana enables querying and visualizing those traces.

Display

The added value of OpenTelemetry and observability, in general, becomes evident when all the data is presented in a report. Below is an example with Grafana. Through what is known as a “Flame Graph,” you can break down all method calls as well as the various transactions between different stakeholders, whether through REST API calls, database queries, or asynchronous messaging.

Unlike APM (Application Performance Monitoring) solutions, which are used only in dedicated environments (e.g., environments dedicated to performance testing), this display provides a snapshot of transactions executed in production.



To optimize the display and provide more ease and flexibility in error analysis, it's also possible to correlate logs and traces. To do this, it's necessary to add the `SPAN_ID` and `TRACE_ID` identifiers to the logs.

For example:

```
1 { "sequenceNumber":0, "timestamp":1719910676210, "nanoseconds":210150314, "level":"INFO", "
   threadName":"kafka-binder-health-1", "loggerName":"org.apache.kafka.clients.
   NetworkClient", "context":{"name":"default", "birthdate":1719907122576, "properties":{"
   applicationName":"easypay-service", "instance":"easypay-service:9a2ac3f0-c41e-4fcd
   -8688-123993f1d5db"}}}, "mdc": { "trace_id":"8b277041692baa8167de5c67977d6571", "trace\
   _flags":"01", "span_id":"13ff9e44be450b8e"}, "message":"\[Consumer clientId=consumer-
   null-1, groupId=null] Node -1 disconnected.", "throwable":null}
```

Thus, in the visualization tool, we can link traces and logs to have more contextual information.

Compatibility

OpenTelemetry is the de facto standard for transmitting observability-related data, particularly traces. The main solutions on the market are compatible with it. If you wish to further enhance observability in your platforms, here is a non-exhaustive list of market solutions offering aggregation and visualization of distributed traces. Here is an overview of various market solutions that include trace management in their portfolios. They are all compatible with OpenTelemetry.

Note: Solutions that do not offer an agent or collector rely on OpenTelemetry.

Performance

A common question is the impact on performance. It is undeniable. Depending on the environment, technology, or the number of SPAN per transaction, it can affect the expected quality of service to varying degrees.

To meet execution time requirements, it is necessary to conduct performance tests as early as possible to measure the impact. Based on the results, it may be necessary to perform trace sampling.

Through this, we may process only a percentage of requests (e.g., 10%). There are two possible sampling mechanisms: “head sampling,” applied at the agent or application level, and “tail sampling,” applied in the collector. The first is mandatory if your application is subjected to high volumes. This will prevent the creation of SPAN, network calls, and storage.

At Worldline, we conducted several tests on performance impacts. On Java applications, on average, we encounter an impact of up to 15% on response times. Like with logs, this must be weighed against the benefits that traces can bring in error analysis.

Conclusion

You will note that OpenTelemetry offers many features. Among these, traces provide a complementary view to logs and a more complete vision of platforms. However, they should not be confused: they will not replace logs. They highlight distributed transactions and provide more contextual information about an error.

Moreover, one of the main advantages is that it is easily implementable from development. For example, in Java, adding an agent at the JVM startup captures traces without modifying the code. This mechanism is very useful for observing applications more finely, whether they are “cloud native” or “LEGACY.”

Finally, if you want to explore all aspects related to observability in more detail, I conducted a workshop on this topic with my colleague David PEQUEGNOT. It is available at the following address: [Workshop](#) . You will find the source code of the examples presented in this article.

- <https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/>
- <https://www.w3.org/TR/trace-context/>
- <https://opentelemetry.io/docs/languages/>
- https://en.wikipedia.org/wiki/Extract,_transform,_load

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



JAVAPRO
THE FREE MAGAZINE FOR THE JAVA COMMUNITY

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



Debugging In The Dark

Debugging

Observability

EngineeringMindset

Authors: Gireesh Punathil and Aditi Srinivas

Gireesh Punathil is an IBM master inventor, Senior Technical Staff Member, committer or steering committee member for several leading open source projects. He is part of IBM Application Runtimes team that develops and supports websphere, java, eclipse and many other tools and frameworks. He spent a good share of his career in developing, porting and debugging platform tooling software: web servers, virtual machines, compilers, ide, monitoring agents and cloud components.



LinkedIn

Aditi Srinivas is a Software Developer at IBM ISL. Passionate about researching and developing new technologies.



LinkedIn

Some bugs scream, while others whisper. but the worst ones hide in the shadows. they slip past logs, contradict facts or exist only in our own assumptions. this is not about breakpoints or stack traces. it is about what happens when we have few or no clue. debugging in the dark is when data fails you, and what remains is your reasoning, curiosity and the gut.



At the lowest levels of computing where raw bits meet the metal, failures are loud and exact: For example a `SIGSEGV` means memory was not allocated. Similarly a `SIGILL` means a bad instruction. The path to resolution is straight, trivial and methodical.

Enter large scale enterprise deployments! As we ascend into virtual machines and managed runtimes that run complex workload, failures grow abstract too, and the space between cause and effect begins to blur. In systems like the JVM, the distance between the developer's intent and the machine's execution is huge. It is layered across interpreters, compilers, runtime optimizers, and countless abstractions in the pathways that connect those.

When things go wrong in such environments, debugging requires higher order reasoning, including reverse abstraction. You peel back each layer, trying to de-abstract the system one level at a time, until you are face to face with the real problem, grounded in the design or the code, at a level you can reason about.

In this article, we walk through the darkest terrains of software problem solving where traditional tools fall short, and you are left staring at the dead end of problem determination. we dissect those scenarios scientifically and illustrate the tools and methodologies that help to make progress.

We will cover five darkness types:

- Missing or too few data: no logs, no metrics and no dumps. Just behaviour.
- Too much data: lots of noise in the diagnostic docs.
- Contradicting data: different sources hint at different causes.
- Non-reproducibility: it happened once and won't come back.
- Bug zombies: non-existent bugs, assumptions, bias or blind spots.

For each type, we define the mental state we need to adopt before diving in. Then we explore the tools and methods that help us make progress step by step, until we reach convergence.

Data Is Too Less

In a world flooded with logs and traces, there are few systems that remain silent, giving very little insight when something goes wrong. When there are no logs, no traces, no dumps and no telemetry, debugging turns into guesswork. Your strongest asset is not code or data, but your reasoning.

Example: legacy systems with minimal or no observability / serviceability capability.

Mindset

- Comfort with uncertainty: with very less data available for debugging, systems are unclear. This necessitates your mind to be in comfort with uncertainty and patient with ambiguity.
- Iterative failures: with minimal available data, you may not have answers for everything, and your theory may be wrong. Be ready to walk through iterative failures.
- Patience: be patient and observe the system's behaviour carefully to catch any deviating behaviours or bugs.



Tools

- Logical reasoning: simulate code paths mentally: how the system reached this state?
- Hypotheses: theories that can potentially explain the issue. Multi-thread re-entrance could be a possibility?
- Intuition & gut instincts: Does this feel like a race condition?

Methods

Be a sharp observer through carefully crafted and modulated interactions:

- Iterative experimentation: change one thing at a time (code, config, input) and watch the change. Which change made the bug to be suppressed or delayed? Example: stopping usage of a suspected feature avoids a reported leak.
- Mental modelling: reconstruct system state in your head or in notes. What does the change imply? Example: the suspected feature may be causing the leak.
- Code forensics: Examine version differences, code paths, test artefacts for hints of what changed. Run targeted stress or edge cases to force the bug to manifest wide. Example: invoke the suspected feature a hundred times, which would amplify the leak too a hundred times.

Convergence

- Test theories decisively: prove or disprove. One at a time. Segregate proven theories.
- Connect the proven tests: correlate the valid theories. Map design with observed symptoms. Chain them together to form one or more coherent theories.
- Build the narrative: converge to a plausible root cause by iteratively ruling things out and refining the story.

Data Is Too Much

Sometimes we get overwhelming data that confuses us. logs in gigabytes, thousands of metrics per second, stack traces from dozens of threads, and dashboards flashing red everywhere. Too much floodlight causes us headache and misleads from focussing.

Example: micro services based deployments with too many observability artefacts.



Mindset

- Selective attention: filter out noise from signals. Zoom in and out as required.
- Correlation and causation: just because two things exist together doesn't mean they relate. Also doesn't mean one caused the other.
- Comfort in complexity: learn to live in systems that are too big to fully comprehend.

Tools

When everything is logged, metrics become meaningless unless correlated and segregated: * Dimensional filter: metrics filtered out by tags to narrow the scope. example: by user, region, endpoint or service. * Timeline maps: correlate events by time to reconstruct the sequence of actions and narrow down the scope of the issue within life span of its occurrence and the cause. * Anchor logs: a subset of log files collected through potentially multiple iterations to pivot analysis around it.

Methods

- Define the blast radius: scope in the affected systems and get their logs. Example: a `java.io.SocketException` would imply communication problem between two network endpoints. Filter in data pertinent to only those two end points.
- Apply known patterns: look for classic symptoms. Example: identify what data is generated when end points were communicating without issues. Differentiate good and bad cases with the trace data.
- Breadcrumb collection: zoom in on the window when the issue happened, reducing to few solid, coherent data points. Example: identify the file descriptor pertinent to the failing socket, and follow the activities on the fd, right from its inception.

Convergence

- Trace metric to code: use logs, traces and metrics to walk through the abstraction layers and reach the code. Many things might look suspicious, but focus on what changed recently or exclusively.
- Compare the normals: what's different this time? Version, traffic pattern, data skew?
- Build your narrative: once the noise dies down, what is left becomes your story lines.

Data Is Contradictory

Logs tell one story, metrics tell another and your intuition tells something else. It is like investigating a crime where all witnesses lie. Truth feels splintered, and logic smells of bias. and debugging becomes contradiction management to start with.

Example: a hybrid cloud environment with heterogenous components, each in a different technology stack, timezone and versions.



Mindset

- trust nothing fully while holding multiple truths: allow conflicting theories to co-exist until evidence matures.
- Revisit the premises: theory, data, tools, methods – all can be flawed or biased.
- Be patient with ambiguity: systems can behave inconsistently under stress or scale.

Tools

When data contradicts, triangulate the pseudo-truths to locate the real truth

- Multi-source telemetry: logs, metrics or traces from different layers of the stack – client, proxy, server or database. These sources should measure similar events from independent perspectives.

- Runtime configuration: wide range of configuration from the operating system to the platform and from the middle-ware to the application to help perform targeted debugging with clinical precision.
- Trace visualisers: visual trace help you see which span caused delay, how retries or timeouts happened or where exceptions were thrown. This helps reconcile anomalies with relative ease.

Methods

When different views of the system disagree, investigate not only the system components and entities, but the lens through which we see those.

- Independent audits: trace how the data was collected. List what can be trusted. Compare independent sources measuring the same entity. Example: client side and server side logs should both match the latency, as they collect the same thing, but independently.
- Reproduce with control: try to recreate the scenario in a sandbox. Example: live debugging to examine minute details and explain and eliminate contradictions. Understand how each data is gathered, what triggers a log line, what averages a metric. Example: latency metric is average of latencies over time, that includes both fastest and slowest responses.
- Identify and reconcile biases: some tools skew or drop data under pressure. For example, JVM JIT profiling. Use distributed tracing to piece together end-to-end behaviour across services. Example: a frontend timeout may appear as a client bug, but tracing reveals that a downstream service failed, causing a delay in the invoker.

Convergence

- Find the common denominator and reconcile: what evidence appears in all viewpoints? What event could logically produce the current outcome? What could explain it?
- Use eliminative reasoning: discard impossible interpretations until the likely path emerges.
- Patch the mental model: continuously update your understanding to fit the full behaviour.

Problem Not Reproducible

This is a classic nightmare. We tackle the situation that is not about missing, overflowing, or conflicting data. But a problem that refuses to repeat itself. The hardest bugs are those that appear once and vanish. Typically, these issues surface at the intersection of environment, timing, workload, and many other randomness seeded from scale, concurrency, hardware quirks, traffic variations, or timing. You are unable to reproduce the issue because your setup is unable to reach this entropy level. You are debugging the memory of a failure, not the failure itself!

Example: production environment with peculiar execution characteristics not present in staging or dev systems.



Mindset

- Think probabilistically: accept that this bug may only appear under extremely narrow conditions. Your mental model may not match how the system behaves under load.
- Don't rush: every probe or change must be minimally invasive, because we don't know what we are cutting through is the right place or not.
- Detective mindset: rare bugs may be the most revealing. think of possible ways to catch it next time it shows up. Just because you can't trigger it again doesn't mean it doesn't exist or that it is resolved. You may not get a full picture. Learn to operate on fragments.

Tools

- Session snapshots: logs, traces, thread dumps, core files, or heap states together provide a lot of information about the execution environment to match.
- Temporal evidence: that record system state timelines around the event. This can be used to mimic a local recreate.
- Crash signatures: that record stack fingerprints for pattern matching.

Methods

If we cannot reproduce the problem, at least let's explore the problem terrain.

- Event reconstruction: use data from logs and metrics to reassemble what the system was doing before failure. Example: gc patterns, load conditions, memory allocation patterns, installed memory, processor count etc.
- Compare with known paths: contrast this execution path with known healthy scenarios. Gather fingerprints (stack traces, error codes, memory signatures) from production to drive theory. Example: allocation and deallocation patterns.
- Spot weakness zones: look for paths that depend on timing, concurrency or unbounded input. What are the difference between pass and fail? Version, config, infra, data, load patterns, network characteristics etc.

Convergence

- Form theories backed by weak evidence: a single crash dump can seed a chain of reasoning.
- Design detectors: add custom guards or assertions where you suspect the break. Systematically vary input data to find triggers and thresholds.
- Treat rare failures as first class problems and promote them to backlog, track them over time, rather than ignore them as flakes.

The Problem Is in Our Mind!

This final part goes deep into ourselves, not into logs and dumps. The most elusive bugs often don't live in the code, system, or environment, but in our assumptions. We see a bug because we believed something false. We missed the cause because we trusted something untrue. Sometimes, everything is functioning as intended, except our understanding. Maybe we assumed a variable was always set, misunderstood what a function actually returns, or trusted logs to tell the whole truth. The system behaved exactly as designed, it was our mental model that failed. These bugs are the most frustrating and humbling, because the code didn't lie, we just misunderstood it!

Example: the most obvious example is CPU and memory monitoring tools showing data that causes panic and suspicion of all sort to an onlooker who watches it close.



Mindset

When debugging with your own assumptions, the best mindset is to play down your preassumptions and lean into introspection.

- Challenge your certainty. The stronger your belief, the harder you must test it. Confidence is not proof. Evidence and the awareness thereon is.
- Debug your own reasoning – step back and inspect how you are thinking, not just what you are thinking.
- Be kind to yourself – mistakes born from intuition or expertise are still mistakes.

Tools

- Assumption log: a dump of all relevant assumptions in the bug context. To be used in rubber ducking.
- Design doc and user guide: design doc reveals what the system was intended to do. To compare your expectations with architectural reality. User guide clarifies how the system is supposed to behave.

- Debug diary: that logs your observations, hypotheses and changes. Reviewing this over time helps you spot tunnel vision or repeated assumptions.

Methods

To debug your brain, you must reverse-engineer your beliefs:

- Rubber ducking: revalidate each belief. Explain your thought out loud or to a peer and ask them to validate. Let them be mindful that false assumptions often seep through words. Treat every assumption as suspect. Confirm it in code or test. If a library call confuses you, wrap it in logs and tests until it's demystified.
- Revisit first principles: re-read docs and code slowly and literally. Skip no words. Then map it with the behaviour and outcome. Step back to core definitions and start again. Does this system take this much load? Is it abnormal?
- Watch out for biases: confirmation bias (only seeing what supports your theory) and anchoring (fixating on early clues) are all mental traps. Step away deliberately into another engaging task and return later. Clarity is rebuilt in the brain each time after it was flushed.

Convergence

The goal here is not just to solve this bug. Instead to upgrade your model of reality:

- Admit what you didn't know and learned new. Don't bury or hide it. Write it down or share it.
- Reframe mistakes as training: misunderstandings are not failures; those are practice runs for insight. Every course correction moment is a learning.
- Get curious, not defensive: the best debuggers are not always the smartest. They are the most curious.

Summary

We have now learned, like seasoned engineers, how to stay calm in the storm and how to fare through uncertainties with the help of powerful tools and methods such as:

- Reasoning when data is missing
- Prioritise when data overflows
- Cross verify when data contradicts
- Instrument and surgical debug for rare issues
- Debug ourselves when the bug is a zombie

In summary, debugging in the dark is not a skill; it is a culture – integration of mindset, tools and methods. This culture turns confusion into curiosity and uncertainty into insight!

So the next time you hit that moment where nothing makes sense, don't give up. Just pause. You are not lost. You are just entering a darker room with a number of coloured lights to show you the way!

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



The Art of Static Code Analysis

JavaParser

ParsingTechniques

CodeAnalysis

Author: Martin Toshev

Martin Toshev is a solution architect and IT consultant conducting professional trainings for novice and experienced developers. He is a Java enthusiast and one of the leads of the Bulgarian Java User group (BG JUG) where he is one of the organizers of jPrime conference.



LinkedIn

The Necessity for Static Analysis of Source Code ...

Most Java (and not only) developers have used at minimum some sort of static analysis tools to perform a tasks such as (to name a few):

- deriving source code metrics such as lines of code or [cyclomatic complexity](#);
- discovering bugs, vulnerabilities, or code smells such as unused variables (what popular IDEs typically do);
- performing automated refactoring or code completion;
- enforcing code and quality standards.

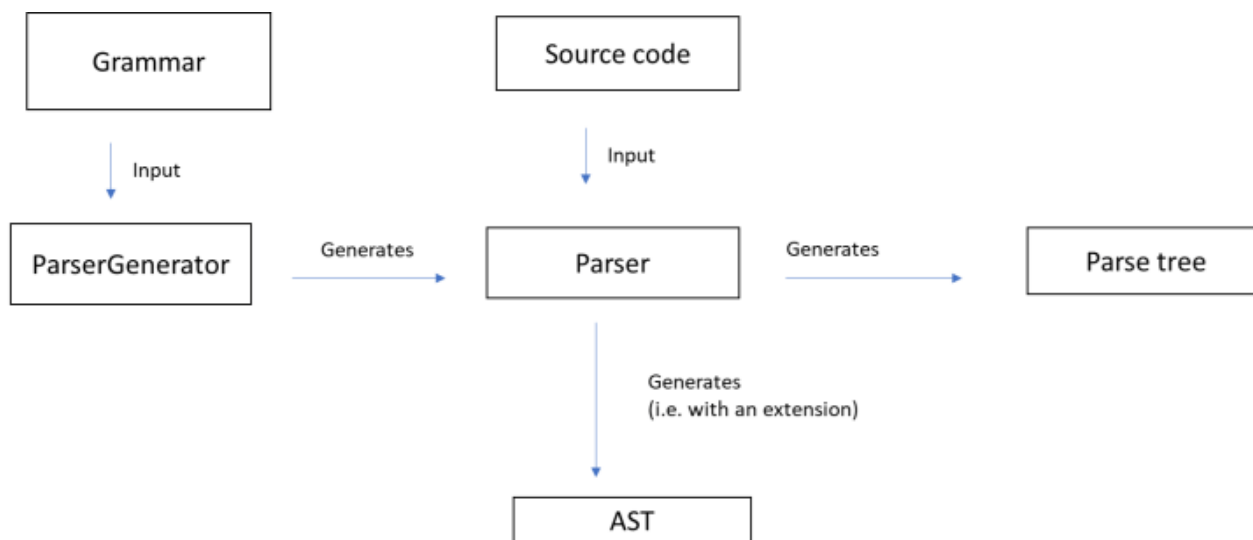
To perform static code analysis, we typically need a proper representation of source code, suitable for analysis. A programming language can be described by a formal grammar. Furthermore, a parser can be created or generated following the rules of a formal grammar to create a proper representation (typically a parse tree) from source code. Based on the type of language we want to represent, we can use different types of formal grammars:

- regular grammars (i.e. regular expressions): they are available in most programming languages but are typically used for more basic parsing tasks. In many cases, they are not suitable for parsing a modern programming language as they are slow and hard to maintain;
- context-free grammars (i.e. BNF or eBNF): one of the most well-known formats is BNF (and its variants), a parser can also be generated by the grammar rules;
- other formal grammars (i.e. PEG).

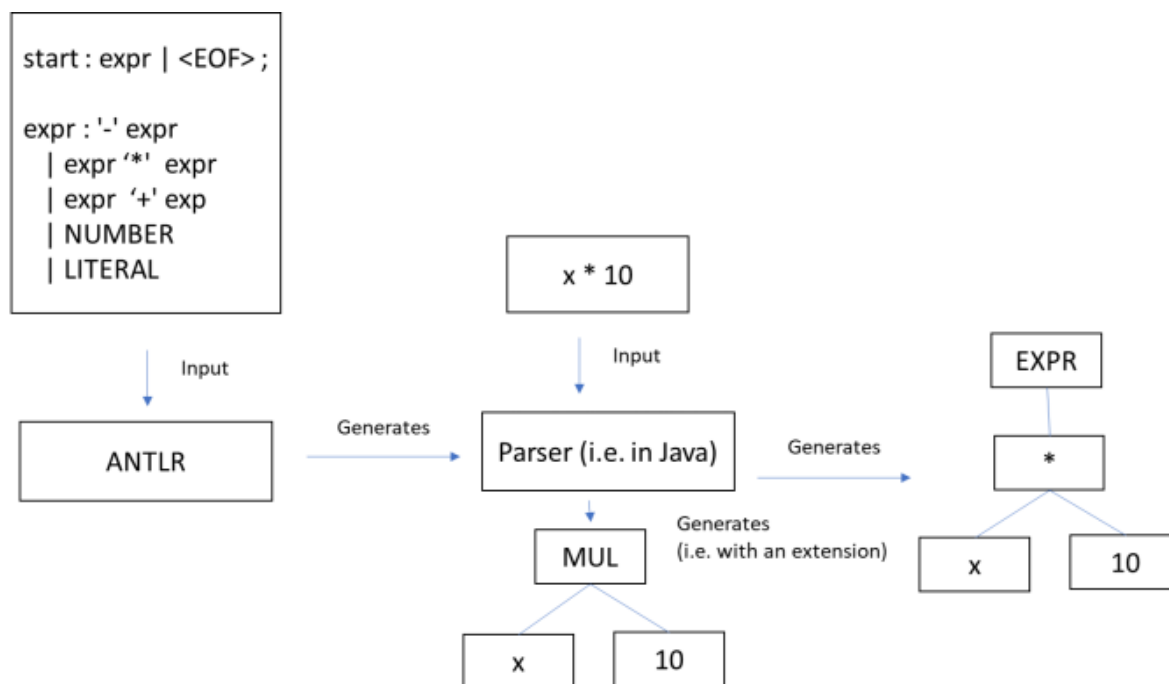
It is not uncommon that in the early days, different tools for static code analysis required writing a parser manually, which is not a trivial task

Parser Generators to the Rescue ...

Tools can be created to generate parsers based on a target context-free grammar rules. This is, for example, the case with tools like LEX and YACC written in C and generating code in C. At a high level parser generation is illustrated by the following diagram:



In the early days of Java Sun Microsystems has developed a parser generator called Jack which was then later renamed to JavaCC (which stands for Java compiler-compiler). Another popular alternative for generating a parser for a Java grammar is ANTLR (ANother Tool for Language Recognition). Both of these parser generators are well supported and written in Java. JavaCC (similar to YACC for C) can combine grammar rules with Java code that is included in the generated parser. However JavaCC provides code generation only for Java while ANTLR is general purpose, has a large number of grammars for a number of programming languages and provides the possibility to generate parsers in different languages. Both of these tools work with formal grammars in eBNF form. Considering, for example, the above general diagram here's how the process of parser generation looks like in Antlr with a simple example of an expression parser generator:



The parse tree that is generated by the parser itself provides more effort in terms of code analysis so that is why typically parser generators provide the possibility to generate a more concise representation which eliminates extra symbols and provides additional symbol resolution capabilities: the AST (abstract syntax tree). The process of using ANTLR or JavaCC in a standard Maven/Gradle project is very similar. For example for ANTLR:

- add Antlr4 dependency and plugin in Maven/Gradle build file;

```

1 <dependency>
2   <groupId>org.antlr</groupId>
3   <artifactId>antlr4-runtime</artifactId>
4   <version>4.7.1</version>
5 </dependency>
6 ...
7 <plugin>
8   <groupId>org.antlr</groupId>
9   <artifactId>antlr4-maven-plugin</artifactId>
10  <version>4.7.1</version>
11  <executions>
12    <execution>
13      <goals>
14        <goal>antlr4</goal>
15      </goals>
16    </execution>
17  </executions>
18 </plugin>...
  
```

- create grammar files under `src/main/antlr4` (in g4 format, Java 20 grammar files available under [GitHub](#));
- generate lexer and parser using Maven/Gradle build.

Once this is in place the generated parser can be used to generate a parse tree that can have i.e. a listener attached for specific executions during the parsing process. Example using ANTLR-generated parser:

```
1 String content = "public class Example { public void func(int x){ return x + 10; } }";
2
3 Java20Lexer lexer = new Java20Lexer(CharStreams.fromString(content));
4 CommonTokenStream tokens = new CommonTokenStream(lexer);
5 Java20Parser parser = new Java20Parser(tokens);
6 ParseTree tree = parser.compilationUnit();
7 ParseTreeWalker walker = new ParseTreeWalker();
8 ExprListener listener = new ExprListener();
9 walker.walk(listener, tree);
```

An alternative way to create a parser is by a parsing expression grammar (PEG). A library that implements this approach (the grammar rules are written in Java code directly as part of the application) is Parboiled.

Java Libraries to the Rescue ...

Parser generators and PEG parsers are quite generic. They also may not be up to date with the desired Java version. As an alternative a specializing parsing library can be used such as JavaParser or Eclipse JDT.

JavaParser

It is based on JavaCC, it is well maintained and provides support for JDK 21. It provides enhanced symbol resolution and generates an AST from the source code. In addition it provides capabilities to query the AST via a DSL provided by the library, generate code from the AST or modify it. It is really simple to get started using the library, the following example counts the number of methods in a class:

```
1 public static int countMethods(File file) throws FileNotFoundException {
2     CompilationUnit cu = StaticJavaParser.parse(file);
3     int count = 0;
4     for (Node node : cu.findAll(MethodDeclaration.class)) {
5         count++;
6     }
7     return count;
8 }
```

To get started using JavaParser it is sufficient to include the following dependency:

```
1 <dependency>
2     <groupId>com.github.javaparser</groupId>
3     <artifactId>javaparser-symbol-solver-core</artifactId>
4     <version>3.26.3</version>
5 </dependency>
```

Eclipse JDT

Eclipse JDT (Java Developer Tools) is the main fuel behind the Java editor in Eclipse IDE that provides advanced capabilities like partial compilation, code completion etc. In earlier days of Eclipse it was not straightforward to use JDT outside of the Eclipse IDE primarily because these required a number of extra dependencies to be dragged as well. Now Eclipse JDT is available as standalone library via the following dependency:

```
1 <dependency>
2     <groupId>org.eclipse.jdt</groupId>
3     <artifactId>org.eclipse.jdt.core</artifactId>
4     <version>3.36.0</version>
5 </dependency>
```

The following example implements a method to count the number of methods in a Java class:

```

1  public static int countMethods(File file)
2      throws IOException, MalformedTreeException, BadLocationException {
3
4      String source = FileUtils.readFileToString(file, Charset.defaultCharset());
5      Document document = new Document(source);
6      ASTParser parser = ASTParser.newParser(AST.JLS21);
7      parser.setSource(document.get().toCharArray());
8      CompilationUnit unit = (CompilationUnit) parser.createAST(null);
9
10     int count = 0;
11     List<AbstractTypeDeclaration> types = unit.types();
12     for (AbstractTypeDeclaration type : types) {
13         if (type.getNodeType() == ASTNode.TYPE_DECLARATION) {
14             List<BodyDeclaration> bodies = type.bodyDeclarations();
15             for (BodyDeclaration body : bodies) {
16                 if (body.getNodeType() == ASTNode.METHOD_DECLARATION) {
17                     count++;
18                 }
19             }
20         }
21     }
22     return count;
23 }

```

As you can see, there are multiple options you can choose from in order to start writing a tool on your own for static analysis of Java code.

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



JAVAPRO

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



Beyond Version Control: How Git Can Power Smarter Technical Decisions

GitAnalytics

GIT

CodeInsights

Author: Francisco Contreras

Francisco Contreras is an Experienced Chief Technology Officer with a strong background in managing and leading software development teams. Proven track record in the information technology and services industry. Expert in Agile methodologies, team management, project leadership, IT consulting, and driving technological innovation. Actively involved in community work as a co-organizer of the Java User Group Nicaragua and former organizer of Flutter Managua. Strong information technology professional with a Master's degree in Computer/Information Technology Administration and Management from Universidad Nacional de Ingeniería.



LinkedIn

When most developers think about Git, they think about branches, commits, merges, and the occasional conflict that makes them question their life choices. But fortunately Git is more than just a version control system, it's also a powerful source of data—data that can drive strategic decisions, improve workflows, reduce risk, and help your team write better software.

In this article, we'll explore how you can use Git not just to manage code, but to **gain insights that lead to smarter, faster, and more reliable decision-making**. Whether you're a developer, a team lead, or a CTO, Git can offer you a hidden layer of intelligence you may not be using to its full potential.

The Untapped Power of Git History

Every day, a software developer team sends a bunch of commits to a Git repository, to enhance code, rewrite functions, create new features, solve bugs (or even introduce them). Git doesn't just store code—it stores a **timeline of decisions**: what changed, who changed it, and why. Each commit message, merge, branch, and tag captures intent and context. And every timeline can tell a story that can be analyzed to extract patterns, uncover bottlenecks, evaluate team dynamics, and even identify technical debt.

This data, overseen some times, can provide very useful insights for better decision making, so I want to go by a couple of interesting questions we normally ask about projects that can be answered by Git data.

What Parts of the Codebase Are the Most Fragile or Risky?

There are many factors to consider when evaluating whether a particular file, class, or module in a codebase is fragile or potentially risky to maintain. While some indicators require runtime behavior, production logs, or developer intuition, Git itself offers a rich and often underused source of historical data that can reveal where complexity, instability, or maintenance effort is concentrated.

Not all parts of a codebase carry equal risk. In practice, software systems often follow a Pareto distribution: a small percentage of files — sometimes as little as 4% to 6% — account for the majority of changes, bugs, or ongoing developer attention. These files are known as hotspots. They evolve frequently, are tightly coupled to business logic, and often serve as integration points between different areas of the system. Because of this, they require a large amount of effort to review, test, and maintain.

Highly changing files can be a sign of flexibility and adaptability, but they can also signal design issues, unclear responsibilities, or repeated patches for persistent bugs. A module that sees frequent commits, large diffs, or involvement in many issues may be fragile not just because of its code, but because of the invisible pressures acting on it — unclear ownership, under-testing, or architectural bottlenecks.

By analyzing Git metadata — such as commit frequency, number of contributors, lines added or removed, and how often a file is touched by feature or bug-related changes — teams can begin to quantify which parts of their system are most likely to cause trouble in the future. These Git-derived metrics provide a foundation for making informed decisions about where to focus refactoring, add safeguards, or initiate knowledge sharing before the risks turn into incidents.

Commits per Period of Time

What it tells you:

A file or module that receives frequent commits over a specific time window (e.g., weekly, monthly) is likely a **hotspot** in your system. This typically means:

- It's central to business logic or system operation.
- It's changing too often, possibly due to bugs, poor design, or high coupling.

How to use it:

Identify areas of high churn to prioritize for **unit testing** or **refactoring**.

Correlate spikes in commit frequency with sprint cycles or production incidents.

Detect files that are unexpectedly active—this can reveal design violations or “God classes.”

Tip: Use `git log --since="3 months ago"--name-only` and aggregate by file to detect recent hotspots.

Lines Changed per Period of Time

What it tells you:

Some files may not receive many commits, but when they do, they involve **large changes** (e.g., 100s of lines added or removed). This signals:

- High **complexity or volatility** per change.
- Risk of **regressions** or **incomplete reviews**.
- Possibly **too much logic crammed** into a single module.

How to use it:

- Prioritize files with large diffs for **pair programming**, **peer review**, or **modularization**.
- Combine line change data with test coverage tools to ensure these critical changes are well-verified.
- If a single developer is doing large, solo changes—raise a flag.

Tip: Use `git log --numstat` to track added/removed lines per file.

Issues Solved In Files per Period of Time

Since Git doesn't track issues directly, **your team should get into the habit of including issue or task IDs in commit messages**. This small discipline transforms your Git history into a rich data source, enabling you to trace which files were involved in specific tasks, bugs, or features. It becomes much easier to analyze development patterns, evaluate the scope of changes, and make informed decisions grounded in real activity. If you're already following this practice, you're in a strong position to extract meaningful insights and build metrics that reflect both technical and functional evolution.

What it tells you:

Tracking the number of **issues (features, bugs, tasks)** associated with a specific file or module over a given time window (e.g., per sprint, month, or quarter) provides valuable insights into:

- How often the **module is involved in active development**.
- The **evolutionary role** of the file: is it central to product changes, or peripheral?
- Whether a module is showing signs of **high functional volatility**, which may increase maintenance complexity.
- Indirect indicators of development attention or instability, depending on context.

This metric doesn't measure effort directly, but it helps identify files that:

- Are core to ongoing business logic, or
- May require extra testing, documentation, or stability planning.

How to use it:

- Extract issue codes (e.g., `FEAT-101`, `BUG-55`) from commit messages for a given file or directory using Git.
- Filter by a time range (`--since="3 months ago"` or by sprint tag).
- Count how many unique issues have touched the file during that period.
- Correlate with other metrics like churn, number of contributors, or lines changed to identify high-risk or high-ROI areas.

Tip: Use a command like:

```
git log --since="3 months ago"--pretty=format:'%s'-- | grep -oE '[A-Z]+-[0-9]+' |
sort | uniq
```

to get the number of distinct issues associated with a file in that timeframe.

Who Are the Key Contributors to Critical Modules?

In any software project, understanding **who is working on what** is as important as understanding **how the code works**. Developers aren't just writing lines of code — they're building domain knowledge, untangling complex logic, and making architecture decisions that are often invisible in documentation. Over time, this knowledge accumulates in the minds of individuals, not just in the repository.

This becomes especially important when it comes to **critical modules** — the parts of your system that are core to product functionality, carry high user impact, or are deeply integrated into your architecture.

Imagine a scenario where one developer has been the sole contributor to a core payment gateway or authentication service. That developer might be extremely productive, but they also represent a **single point of failure**. If they leave the company, go on extended leave (planned or not), or are reassigned, the team could face delays, bugs, or costly rewrites just to recover lost context.

Moreover, developers are a **significant investment** — salaries, onboarding time, and internal knowledge all have a real cost. If institutional knowledge is siloed in one person, the **ROI on that cost is fragile**.

Identifying the key contributors to critical modules allows teams to:

- Plan effective **onboarding and cross-training**,
- Distribute knowledge more evenly,
- Avoid key-person dependency,
- And make the system more resilient to team changes.

Git data makes this analysis possible, providing a clear record of who touched what, when, and how often.

Total Contributors per File

What it tells you:

The number of people who have modified a file can reveal **ownership patterns**:

- **Low contributor count** may suggest a **key-person risk** (e.g., only one person understands the file).
- **High contributor count** might reflect a **coordination challenge** (many people touching the same code could cause conflicts or instability).

How to use it:

- Encourage shared ownership for core files, but avoid “too many cooks” on unstable components.
- Match contributor count with actual **PR review history** to understand collaboration vs. chaos.

Tip: Use `git log --pretty="%an" -- | sort | uniq -c` to see who has changed a file and how often.

Former Contributors (No Longer on the Team)

What it tells you:

Files last touched by people who have left the team are **knowledge silos**. These areas may: Contain undocumented logic.

- Be harder to debug or extend due to lack of context.
- Represent a long-term risk to project continuity.

How to use it:

- Flag these files as **candidates for review, documentation, or pair refactoring**.
- Assign ownership or bring them into the attention of active developers.
- If bugs arise in these areas, treat them as **higher-risk** due to lack of original authorship.

Tip: Cross-reference `git log --author` output with your current team roster to identify abandoned ownership.

Limitations and Cautions

While Git provides incredibly valuable metadata, it's essential to understand what it **can** and **cannot** tell you — and avoid drawing misleading conclusions from it.

Overvaluing Commit Count

Not all commits are equal. Some developers commit frequently in small, atomic chunks. Others may push fewer but larger updates. Judging contribution or impact by commit count alone can lead to skewed perceptions of productivity. A single, well-crafted architectural change might be more valuable than dozens of minor formatting updates.

Misinterpreting Inactivity

Just because a developer doesn't appear active in Git doesn't mean they aren't contributing. Activities like architectural planning, mentoring, testing, or debugging often leave no direct trace in Git. Similarly, stable, well-designed code may simply not need frequent changes — and that's a good thing. Don't go just firing people just because their activity is lower than the others.

Blind Spots in Runtime Behavior and Code Quality

Git tells you what changed, but not how the code behaves in production or how maintainable it truly is. It doesn't reveal performance regressions, flaky tests, memory leaks, or runtime errors caused by external dependencies. Nor can it assess internal quality metrics like code complexity, duplication, or test coverage. A file may change frequently without Git ever indicating whether it's becoming harder to read, more fragile, or less secure. To gain a full understanding of code quality and stability, Git data needs to be complemented with tools for static analysis, runtime monitoring, and testing coverage—only then can teams see both how the system evolves and how well it holds up.

Missing Context of Non-Code Work

Important discussions, product decisions, or code reviews often happen outside Git: in Jira, Slack, pull request comments, or meetings. Git doesn't capture that nuance. You may see that a file was changed 12 times, but not understand that those changes were driven by a constantly shifting product spec.

Limitations in Attribution

Git can tell you who made a change, but not necessarily **why** — or whether that person actually authored the code or just merged it. Commits from bots, pair programming sessions, or mob coding might misrepresent who holds knowledge about a part of the system. **Be careful!**

Lack of Business Impact Insight

Git doesn't know which features succeeded in the market, what generated revenue, or which code drove customer satisfaction. That means Git cannot answer **value-based questions** — only **activity-based ones**. High activity does not automatically mean high impact.

Use Git as a decision-support tool, not as a replacement for human insight, team retrospectives, or broader business context.

Conclusion: Your Git Repository is a Goldmine

Git holds more than just your code — it holds a detailed, timestamped narrative of **how your software evolved**. Each commit, merge, and file touched offers insight into how your team thinks, adapts, and solves problems over time.

By treating Git as a data source — not just a file store — you unlock a new dimension of **technical awareness**:

- You can **spot bottlenecks** before they become crises.
- You can **highlight high-risk areas** before they trigger regressions.
- You can understand how team members contribute, collaborate, and take ownership of critical parts of the system.
- And you can **inform your decisions** about testing, refactoring, hiring, onboarding, and even roadmap planning — with evidence, not guesswork.

But like any powerful tool, Git analysis must be handled with care. Use it to support your understanding, not to replace context, conversations, and common sense.

Next time you open your Git terminal, don't just look at what changed — ask **why** it changed, **how often**, and **by whom**. In a world where engineering time is precious and complexity grows quickly, **your Git history might be the most valuable source of truth you're already sitting on** — all you have to do is start listening to it.

Java

The Open Source, Deterministic Engine Maintaining Java's Next 30 Years

JavaModernization

CodeAutomation

LegacyCode

Author: Bryan Friedman

Bryan Friedman is passionate about building and shipping software, educating users, and enabling customers for success. With a love for technology and problem-solving, he has spent the past several years as a product leader, focusing on application platforms, cloud computing, and infrastructure automation, working across Product Management, Technical Product Marketing, and Developer Relations. Now a Technical Marketing Lead at Moderne, he previously led the Customer Learning and Developer Engagement teams at VMware. Before that, he was a Product Manager and Director at CenturyLink Cloud, which was the culmination of over ten years of working as a software developer and in a number of other various information technology areas.



LinkedIn

Java is entering its fourth decade as one of the world's most important programming languages. From banks to telecom to retail, Java still powers systems that billions of people depend on daily. The language's stability and portability are legendary, but so is the technical debt that has accumulated over thirty years of upgrades, frameworks, and shifting best practices.

Large enterprises often face sprawling portfolios of hundreds or even thousands of Java applications, many built atop older frameworks and outdated APIs. Banks still rely on Java 8 systems to process trades. Healthcare systems depend on decade-old Spring applications. Telecom providers run on Java servers written before cloud-native architectures became the norm.

Modernizing these mission-critical apps is not a matter of preference—it's a necessity for security, compliance, maintainability, and developer productivity. Coding with old versions of Java means missing opportunities to adopt modern features like records, switch expressions, or virtual threads that improve code clarity and performance. Yet the traditional methods of upgrading Java codebases don't scale, whether it's manual refactoring, big-bang rewrites, or one-off consulting projects.

That's where OpenRewrite comes in. OpenRewrite is an open-source automated refactoring framework designed to modernize Java safely, deterministically, and at scale.

At its core is the [Lossless Semantic Tree](#) (LST), a data model that captures code structure, semantics, type attribution, and even formatting. Unlike a traditional abstract syntax tree (AST), the LST preserves all the details required for accurate search and transformation.

On top of the LST, OpenRewrite provides recipes: modular, rules-based programs that search for patterns in code and apply transformations. Recipes can:

- Upgrade Spring Boot applications across major versions
- Migrate from JUnit 4 to JUnit 5
- Replace insecure cryptographic functions with NIST-approved alternatives
- Update deprecated APIs to their supported replacements
- Adopt new Java language features and standards

By combining a compiler-accurate data model of source code with deterministic recipes, OpenRewrite provides a foundation for the next era of Java modernization—one that is precise, testable, and even compatible with AI.

Why AI Assistants Fall Short

It's tempting to imagine that large language models (LLMs) like ChatGPT, Amazon Q, or GitHub Copilot could solve the modernization problem outright. After all, these tools can generate code, suggest refactorings, and even scaffold migrations. But when applied to enterprise-scale Java estates, AI alone runs into serious limitations:

1. **Lack of context.** LLMs don't "see" a codebase's full dependency graph, build system, or architectural patterns. Without that context, suggestions may compile in isolation but fail across a fleet of applications.
2. **Inconsistent precision.** AI-generated changes are probabilistic. They may look correct in isolation but aren't guaranteed to preserve semantics, dependencies, or even basic formatting. At enterprise scale, even a 1% error rate can translate into thousands of broken builds. The only fallback is manual review of every change, which becomes intractable when dealing with millions of lines of code across hundreds of repositories.
3. **One-off, repo-by-repo change.** AI-generated refactorings tend to be one-to-one—each suggestion a unique butterfly. That may work for a single file or repo, but it doesn't translate into a repeatable process for an entire code estate.

A common response is **retrieval-augmented generation (RAG)**, which fetches relevant code snippets or documents and feeds them into the model's context window as additional input. But at enterprise scale, this quickly breaks down:

- **Too much code.** A single organization may have tens of thousands of repositories and billions of lines of code. You can't "retrieve" enough context to cover all relevant dependencies or architectural conventions.
- **Context window paradox.** LLM performance *degrades* when fed extremely large contexts. The more you shove into the window, the less coherent the results.
- **Still probabilistic.** Even with RAG, models may suggest plausible changes that fail when scaled to hundreds of services.

For large-scale modernization, using AI to refactor code, even with RAG, is a non-starter.

This isn't just theory. It's visible in how both Amazon Q Developer and GitHub Copilot approach large-scale Java modernization: they rely on OpenRewrite under the hood. Why? Because LLMs alone don't have the data they need. They can't

guarantee semantic preservation, reproducibility, or scalability. OpenRewrite recipes provide the deterministic substrate these platforms depend on to execute safe, large-scale upgrades, while the AI layers focus on user interaction and orchestration.

The foundation that makes this possible is the Lossless Semantic Tree—a compiler-accurate model of source code that captures structure, semantics, types, dependencies, and even formatting. The LST is the unique data backbone that enables OpenRewrite to deliver modernization at true enterprise scale.

The Lossless Semantic Tree (Deep Dive)

When developers hear about program analysis, they typically think of Abstract Syntax Trees (ASTs). ASTs are the backbone of most compilers: they parse source code into a structured, hierarchical representation. That's enough for compilation, but not for automating refactoring. The AST model has several limitations that make it insufficient to perform accurate transformations and searches across a repository:

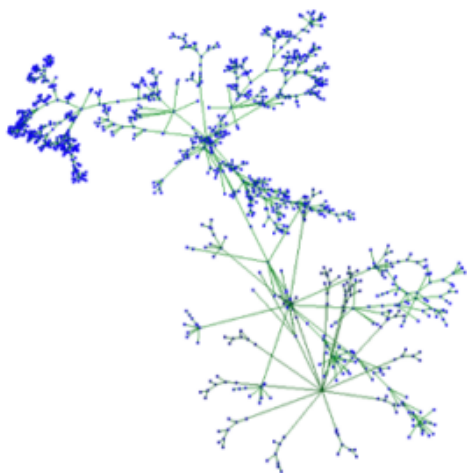
- Discarding “irrelevant” details like formatting, whitespace, and comments.
- Lacking full type attribution (e.g., resolving method overloads or generics across classpaths).
- Not accounting for transitive dependencies, leaving potential gaps in type and symbol resolution.
- Optimized for the one-time task of compilation, not the ongoing maintenance and evolution of large, production codebases.

Enter the **LST**, OpenRewrite's next-generation source code model. The LST has everything an AST does, plus the additional information needed for accurate code search and transformation that ASTs don't have:

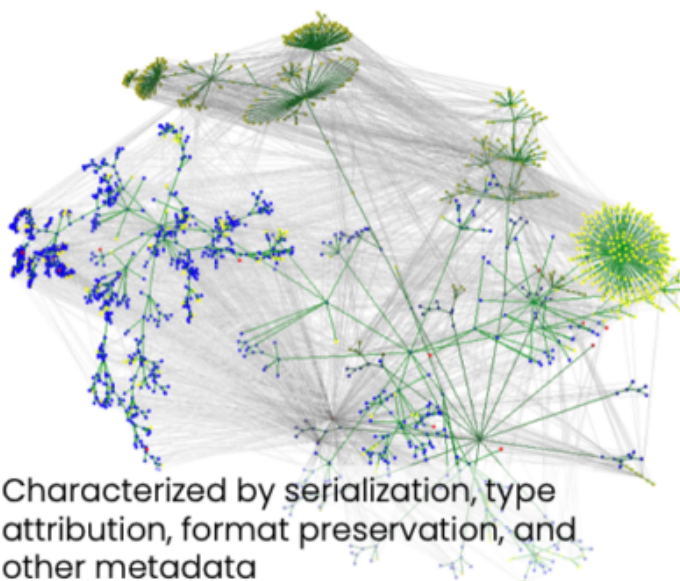
- **Formatting and comments** are preserved, so automated refactoring doesn't wipe out developer intent. This is critical to ensure that refactoring operations insert code that matches the local style of the development team that owns it.
- **Type attribution** is included, meaning every expression is resolved to its type, even across dependency graphs. This is absolutely required to accurately match patterns in the code.
- **Semantic relationships** like imports, annotations, modifiers, and generics are fully modeled so transformations know what needs to remain and what can be removed.
- **Idempotence**: repeated application of a recipe on the same code will not change it again, ensuring predictability.

Since it is produced using both syntactic and semantic analysis, with full type attribution and formatting and comments included, an LST artifact represents a full-fidelity model of the source code. You can see the density of an LST compared to related AST below:

Abstract Syntax Tree (AST)



Lossless Semantic Tree (LST)



+ another 600,000 nodes that didn't fit in this graphic

This rich data lets the modified LST be printed back to source code in the original style, enabling accurate automated refactoring. This enables accurate, automated code refactoring and remediation.

Take this simple Java class:

```
1  import org.slf4j.Logger;
2  import org.slf4j.LoggerFactory;
3
4  public class Example {
5      // the users name
6      private String name = "Java";
7      private static final Logger logger = LoggerFactory.getLogger(Example.class);
8      public void greet() {
9          System.out.println("Hello, " + name);
10         logger.info("Hello, {}", name);
11     }
12 }
```

An **AST** might represent this as a **ClassDeclaration** with two **VariableDeclarations** and a **MethodDeclaration**. But the AST won't have some important information because it:

- Will strip the comment `// the users name`
- Won't encode the exact formatting of the string concatenation in the `System.out.println` method call
- Doesn't have any whitespace information to know whether it uses tabs or spaces
- Has no type information to know what kind of logger the `logger` variable is

The **LST**, by contrast, keeps:

- The comment attached to the `name` variable declaration
- All whitespace information including empty lines, newlines, tabs, and spaces
- The exact formatting of `"Hello, " + name` (including operator spacing)
- Type attribution that tells us:
 - `logger` is specifically of type `org.slf4j.Logger`
 - `"Hello, "` is a `String` literal
 - `name` is a `java.lang.String` field

To get an idea of the level of information an LST has about the code, you can use the [TreeVisitingPrinter](#) provided by Open-Rewrite to print out an LST, as is done below for the example code above. (This is only part of the LST; the full model captures even more detail!)

```

1  ----J.CompilationUnit
2      |-----J.Import | "import org.slf4j.Logger"
3      |         \---J.FieldAccess | "org.slf4j.Logger"
4      |             |---J.FieldAccess | "org.slf4j"
5      |                 |---J.Identifier | "org"
6      |                 |         \-----J.Identifier | "slf4j"
7      |                 |         \-----J.Identifier | "Logger"
8      |-----J.Import | "import org.slf4j.LoggerFactory"
9      |         \---J.FieldAccess | "org.slf4j.LoggerFactory"
10     |             |---J.FieldAccess | "org.slf4j"
11     |                 |---J.Identifier | "org"
12     |                 |         \-----J.Identifier | "slf4j"
13     |                 |         \-----J.Identifier | "LoggerFactory"
14     \---J.ClassDeclaration
15         |---J.Modifier | "public"
16         |---J.Identifier | "Example"
17         \---J.Block
18             |-----J.VariableDeclarations | "// the user's nameprivate String name = "
19             |         |---J.Modifier | "private"
20             |         |---J.Identifier | "String"
21             |         \-----J.VariableDeclarations.NamedVariable | "name = "Java""
22             |             |---J.Identifier | "name"
23             |             \-----J.Literal | ""Java""
24             |-----J.VariableDeclarations | "private static final Logger logger =
25             |         LoggerFactory.getLogger(Example.class)"
26             |         |---J.Modifier | "private"
27             |         |---J.Modifier | "static"
28             |         |---J.Modifier | "final"
29             |         |---J.Identifier | "Logger"
30             |         \-----J.VariableDeclarations.NamedVariable | "logger =
31             |             LoggerFactory.getLogger(Example
32             |                 |-----J.Identifier | "LoggerFactory"
33             |                 |---J.Identifier | "getLogger"
34             |                 \-----J.FieldAccess | "Example.class"
35             |                     |---J.Identifier | "Example"
36             |                     \-----J.Identifier | "class"
37             \-----J.MethodDeclaration | "MethodDeclaration{Example{name=greet,return=
38             |         void,parameters=[]}"
39             |         |---J.Modifier | "public"
40             |         |---J.Primitive | "void"
41             |         |---J.Identifier | "greet"
42             |         |-----J.Empty
43             |         \---J.Block
44             |             |-----J.MethodInvocation | "System.out.println("Hello, " + name
45             |                 |         )"
46             |                 |         |-----J.FieldAccess | "System.out"
47             |                 |                 |---J.Identifier | "System"
48             |                 |                 \-----J.Identifier | "out"
49             |                 |         |---J.Identifier | "println"
50             |                 |         \-----J.Binary | ""Hello, " + name"
51             |                 |             |---J.Literal | ""Hello, ""
52             |                 |             \---J.Identifier | "name"
53             |                 \-----J.MethodInvocation | "logger.info("Hello, {}", name)"
54             |                     |-----J.Identifier | "logger"
55             |                     |---J.Identifier | "info"
56             |                     \-----J.Literal | ""Hello, {}""
57             |                         \-----J.Identifier | "name"

```

This fidelity makes a huge difference. For example, if we are looking for SLF4J log statements to migrate to a different logging framework, we know for sure that `logger` is the right type as opposed to being something else, like Logback or Log4j.

When you're upgrading 500 Java services from Spring Boot 2.x to 3.x, these small details add up. You don't want to replace a logging statement if it isn't of the type that you're looking to replace. If comments vanish, developers lose trust in automation, and if formatting changes unexpectedly, pull requests become noisy and unreviewable.

The LST ensures that automated transformations are semantically correct, syntactically identical where unchanged, and visually indistinguishable from developer-written code. That's why it's the foundation for safe mass-scale Java modernization.

Recipes: Deterministic Automation for Java

Where LSTs provide an accurate map of the code, the next step is automating transformation. In OpenRewrite, this takes the form of [recipes](#). A recipe is a modular program that applies search and refactoring operations to the LST.

Unlike some ad hoc scripts or AI-generated changes, recipes are deterministic:

- **Repeatable:** the same recipe produces the same output on any codebase.
- **Auditable:** recipes are versioned and stored alongside source code.
- **Idempotent:** rerunning them doesn't introduce new changes.
- **Composable:** small recipes can be combined into larger transformations.

OpenRewrite recipes use a [visitor pattern](#) to handle LST traversal and manipulation. A visitor is the part of the recipe that actually walks through the source code tree (the LST) and decides what to do when it encounters different elements. As the visitor moves through the code, it can look for patterns such as a method call, an annotation, or a class declaration. When it finds something relevant, it can apply a transformation like updating an import, changing a type, or rewriting an expression.

The traversal is systematic: the visitor ensures every relevant node in the tree is examined, so no cases are missed. This pattern makes recipes predictable, reusable, and easy to reason about. For practitioners, this means modernization can be automated across thousands of repositories, producing pull requests developers can trust. Recipes can be expressed in different ways, depending on how much customization you need.

Types of Recipes: Building Blocks of Refactoring

Recipes may be declared in YAML, or written as a Java program that extends the Recipe class and implements visitor methods to perform the desired search and changes. OpenRewrite also offers a very convenient [templating mechanism](#) to make it more convenient for Java programmers to manipulate the LST.

Declarative recipes are the simplest form, used to compose and configure existing transformations via YAML. Here's a simple declarative recipe that might be used to migrate from JUnit 4's `@Test` to the JUnit 5 version:

```
1 type: specs.openrewrite.org/v1beta/recipe
2 name: com.example.MigrateToJUnit5TestAnnotations
3 displayName: Migrate @Test annotations from JUnit 4 to JUnit 5
4 description: Replace `@org.junit.Test` annotations with `org.junit.jupiter.api.Test`.
5 recipeList:
6   - org.openrewrite.java.ChangeType:
7       oldFullyQualifiedTypeName: org.junit.Test
8       newFullyQualifiedTypeName: org.junit.jupiter.api.Test
```

Running the recipe should update the code as follows:

```
1 // Before
2 import org.junit.Test;
3
4 public class MyTest {
5     @Test
6     public void testSomething() {
7         System.out.println("Some test");
8     }
9 }
10 // After
11 import org.junit.jupiter.api.Test;
12
13 public class MyTest {
14     @Test
15     public void testSomething() {
16         System.out.println("Some test");
17     }
18 }
```

Composition and Scale

Since recipes are composable, the above example could be incorporated into a larger recipe to perform a full [JUnit 4 to 5 migration](#). Or in the case of a [Spring Boot 2 to 3 recipe](#), it includes thousands of cascading sub-recipes that includes things like:

- Updating `javax.*` imports to `jakarta.*`
- Migrating off of deprecated methods
- Updating application property keys in Properties or YAML files
- Changing necessary dependency versions (in `pom.xml` or `build.gradle`)

Each is deterministic and testable in isolation. Together they form a modernization playbook.

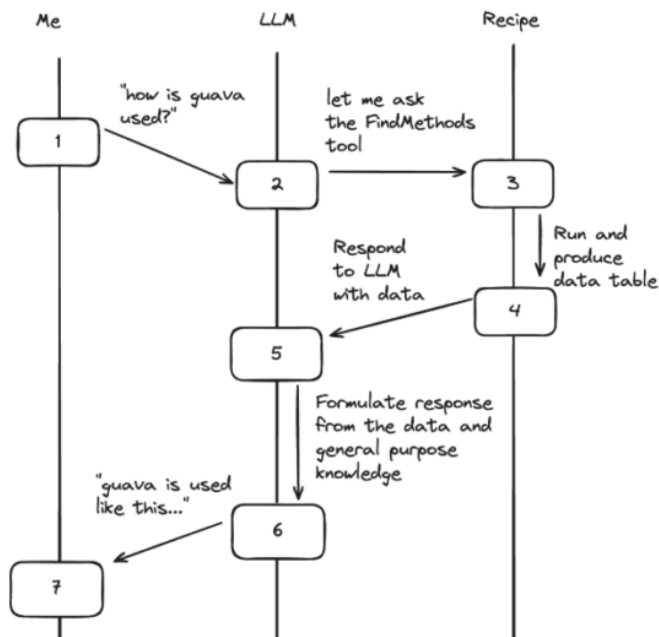
AI + Recipes: A Hybrid Model that Scales

As we've seen, RAG-based approaches don't scale, but there is an alternative—the tool-calling paradigm. Instead of overloading an LLM with raw code, you connect it to real-time data through external services and APIs. OpenRewrite recipes are such tools. They don't just refactor code, they act as a query and transformation language for massive code estates.

When an LLM can call recipes, the workflow changes:

- The model doesn't need to understand everything in context.
- It simply chooses the right recipe(s) based on developer intent.
- The recipe then queries and transforms billions of lines of code with full compiler accuracy.

In this paradigm, the LLM orchestrates, but OpenRewrite does the heavy lifting. See the flow from user prompt to LLM to recipes interacting with LSTs:



Tool calling is a valuable method in these ways:

- **Scalability:** Recipes operate across thousands of repos, something no context window can handle.
- **Precision:** Changes are deterministic, testable, and repeatable.
- **Auditability:** Every recipe run can be version-controlled, logged, and reviewed.
- **Developer trust:** The AI isn't a black box—it's a conversational layer that calls deterministic tools.

The net effect: AI becomes a gateway to automation, not a fragile generator of one-off “butterfly” refactorings. Recipes give AI the leverage to modernize at enterprise scale. Each solution has its best use:

- **LLMs are best at:** understanding intent, summarizing patterns, prioritizing changes, explaining results.
- **Recipes are best at:** querying, transforming, and enforcing deterministic changes across massive code estates.

Together, they create a workflow that scales: AI makes it conversational, recipes make it safe.

How AI Is Accelerating the Growth of OpenRewrite

One of the most practical ways AI applies to large-scale Java modernization is by helping author OpenRewrite recipes, which then serve as repeatable modernization stamps across a codebase. OpenRewrite already has an extensive [recipe catalog](#) with thousands of recipes for common migrations, upgrades, and other transformations. But the pace of change in languages and frameworks means there will always be gaps.

Every codebase has quirks, and every enterprise faces unique challenges. That's where OpenRewrite's design shines: recipes aren't a fixed set of rules, they're programs. That flexibility means teams can go beyond off-the-shelf transformations and encode their own logic, whether it's upgrading frameworks, enforcing compliance, or capturing company-specific modernization playbooks.

Lowering the Barrier for Custom Recipes

Authoring recipes often requires deep knowledge of OpenRewrite's visitor APIs and program analysis concepts. Many enterprises can run existing recipes but creating their own can take time and investment. AI is shifting that balance.

Large language models, including tools like Claude Code, now make it possible to go from nothing to a working recipe in minutes. Developers can generate a first draft quickly, then use OpenRewrite's deterministic execution to ensure the transformation results are safe, testable, and repeatable. AI reduces startup friction, while the platform guarantees rigor.

In fact, writing OpenRewrite recipes aligns extremely well with AI assistance:

- **Clear before/after contract:** The test framework is straightforward. Given code before and after, it's obvious whether a recipe works as intended. That makes it hard for AI to “cheat” its way into looking correct without being correct.
- **High volume of examples:** With thousands of existing recipes, AI can learn from established patterns and produce results that align with best practices.
- **Guidance from documentation:** Pointing an agent at the OpenRewrite docs provides even more context, helping it generate the recipes accordingly.
- **Organizational playbooks:** Teams can supply custom prompts or specific directives (like CLAUDE.md files) with their own conventions, ensuring AI-authored recipes reflect internal standards.

The result: AI doesn't replace understanding recipe authoring concepts, but it changes the learning curve. Developers still need to grasp how recipes traverse and transform code so they may guide the AI assistants in the right direction and provide those important directives, yet they can start contributing much faster and with less specialized knowledge than before.

Use Cases in Automated Java Modernization

Now let's see how OpenRewrite is being put to use in the industry. It isn't just about one-off migrations. It's a platform for applying deterministic, scalable change to every dimension of a Java estate. The framework enables practitioners to tackle common modernization challenges systematically, with proven results.

Framework and Language Migrations

Keeping pace with framework and JVM evolution is one of the biggest sources of technical debt. OpenRewrite recipes provide structured migrations for:

- **Spring Boot 2 → 3** (Jakarta EE namespace changes, actuator endpoints, configuration updates)
- **JUnit 4 → JUnit 5** (imports, annotations, assertions)
- **Java 8 → 21** (language feature adoption, deprecations, build updates)

Example: Choice Hotels used OpenRewrite to migrate 526 applications from Java 8 to Java 21 in one pass. Without automation, they estimated years of effort. With recipes, they executed coordinated changes across hundreds of repos, validating them in CI/CD and freeing developers to focus on new feature delivery.

Security and Compliance Remediation

Modernization isn't just about keeping up—it's about staying safe. With OpenRewrite, organizations can:

- Replace insecure cryptographic functions
- Patch vulnerable dependencies
- Enforce secure coding standards across every repository

Example: A global bank leveraged OpenRewrite recipes to audit cryptographic anti-patterns across thousands of Java applications. One critical finding involved insecure usage of `Cipher.getInstance("AES/ECB/PKCS5Padding")`. This work accelerated their readiness for post-quantum cryptography.

Technology Shifts and Cloud Readiness

Enterprises often need to adapt their code to align with new infrastructure or cloud-native patterns. Recipes can:

- Update APIs and SDKs when moving to new cloud providers
- Adjust configuration for Kubernetes and containerized environments
- Remove deprecated libraries that block modernization

Example: Healthcare provider MEDHOST applied OpenRewrite recipes across more than 5 millions lines of Java to replace obsolete APIs and removed unsupported libraries, eliminating technical debt that had slowed their delivery cycles.

Standardization and Code Quality

At enterprise scale, consistency is key. OpenRewrite makes it possible to enforce organization-wide rules for:

- Logging and telemetry frameworks
- Naming conventions and annotations
- Dependency version alignment

Example: Interactions, a provider of conversational AI, improved maintainability, reduced friction in reviews, and increased overall developer efficiency by ensuring that every pull request passed automated recipe checks in the CI/CD pipeline for formatting and common issues.

The real power of OpenRewrite emerges when these use cases are no longer treated as isolated projects, moving from reactive firefighting to continuous modernization. Framework upgrades happen in sync across every repo. Vulnerabilities are remediated before they spread. Internal standards are enforced consistently.

The pattern is clear: where previously modernization was too expensive or disruptive to attempt, automation through OpenRewrite recipes makes it achievable at enterprise scale. Organizations gain what some practitioners call tech stack liquidity—the ability to evolve their entire Java portfolio as easily as refactoring a single class.

Continuously Maintaining Java's Next 30 Years

Java has endured for three decades not because it stood still, but because it continuously evolved with new language features, new frameworks, and new deployment models. The challenge for the next thirty years isn't whether Java will evolve again, but how organizations can keep pace without drowning in technical debt. This is even more urgent as AI accelerates code generation.

The answer is determinism. Without deterministic automation, modernization becomes a series of one-off projects, each falling behind as soon as the next release lands. With deterministic recipes, modernization becomes continuous:

- Dependency upgrades can be applied across every repository.
- Deprecations can be remediated before they cause breakage.
- New JVM features can be adopted systematically, not piecemeal.
- Security fixes can be applied consistently and everywhere.

OpenRewrite provides the foundation for this model. It is already enabling enterprises to modernize millions of lines of Java safely and repeatably. AI can enhance this work by making recipe discovery more conversational and speeding up the authorship experience, but the heavy lifting of actual code transformation at scale requires the rules of OpenRewrite. For practitioners, the message is simple: the tools now exist to tackle modernization across your entire estate, no matter how large or complex. By embracing [deterministic automation](#) you're not just maintaining code, you're securing Java's future for the next thirty years.

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

APR
20-23



www.jcon.one

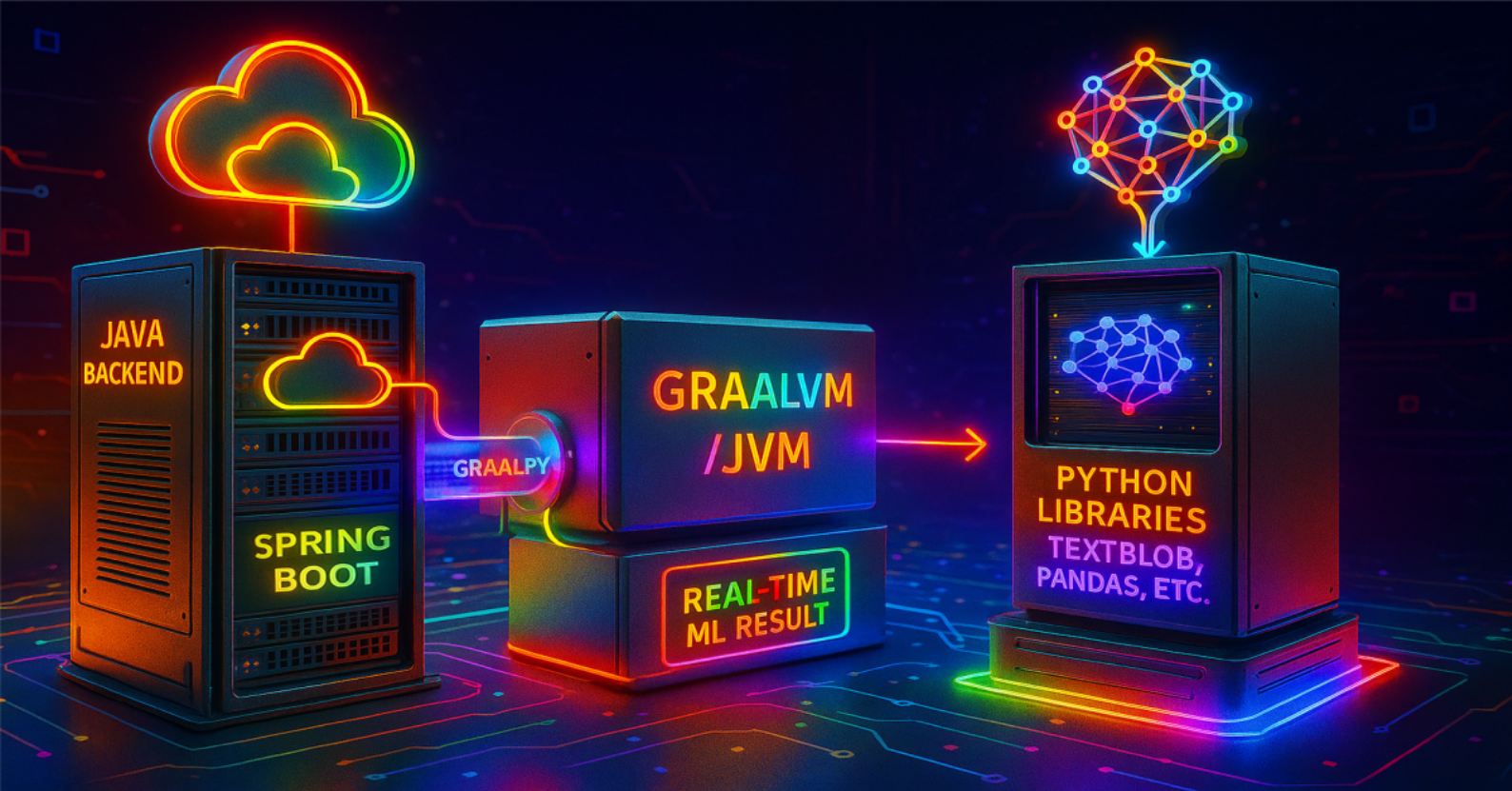
JCON2026

JCON EUROPE 2026

International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



Bridging Java and Python for AI/ML in Production: The Case for GraalPy on GraalVM

AI # API # GraalPy # GraalVM # Microservices # Python

Author: Vishal Shanbhag

Entrepreneur | Startup Mentor | Blogger | Technology Consultant | Tech Lead and Programmer | Trekker. With 20+ years of experience, Vishal Shanbhag has a proven track record as a Senior Technical Consultant, Software Architect, Lead Developer, and Pre-Sales Consultant.



LinkedIn



Abb. 9: Java with a dash of Python

Abstract

In the Java stack, tapping into Python’s powerhouse of NLP and AI/ML libraries often means messy inter-process plumbing—until **GraalPy** changed the game. With GraalVM’s Python runtime embedded directly in your JVM, you can import and run Python libraries like **TextBlob** for sentiment analysis straight from Java. This approach offers a viable alternative for the more complex solutions, such as HTTP, gRPC, or subprocesses.

In this article, we’ll explore a proof-of-concept project built with **Spring Boot** that bundles Python sentiment scoring into a Java API. We’ll show you how the setup is quick, the code stays clean, and the performance is surprisingly good, all while keeping the entire process in-memory.

_Source project: <https://github.com/vshanbha/graalpy-sentiment>

Introduction

I am a career Java programmer, and until AI/ML entered everyday conversations, I was happy using Java for almost everything development. I have even used Java (through GWT) for creating web-based user interfaces that ultimately get deployed as HTML + CSS + JavaScript. That was before the arrival of Angular and React. We could, of course, use HTML + JavaScript, but in those days, web development with compiled code wasn’t actually available any other way. Java + GWT offered an elegant (albeit slightly slower) solution for a Java-first approach.

I digress, though, just to make a point. The point being that I like to create software so that full-stack developers do not have to learn too many different languages just to maintain Production code. That said, at present, frontend technologies have changed a lot, and GWT is now legacy. Full-stack programmers have to learn at least HTML + CSS + Typescript (or JavaScript). That’s just bread and butter now, and at least up until recently, it was sufficient.

The AI ERa

For much of my career, I wasn’t doing any projects that involved AI integration. That changed in the last 5-6 years, and more and more projects were using AI/ML for something or other. Back then, I worked with AI/ML specialists who only spoke one language. Words like Python, Anaconda, and Pandas suddenly meant that I wasn’t watching Animal Planet or National Geographic, but that I was talking to my AI/ML colleagues.

At one point, I remember doing a Google search involving the words Machine Learning and Java, but Google decided that I should learn Machine Learning using Python and gave me results about Getting Started accordingly. It was like Java didn't exist. While that statement isn't true anymore, now and then, one comes across a problem involving AI / ML / NLP where the solution lies in using one or another Python library.

Suddenly, I came to the realisation that as a full-stack or Backend Engineer, being a Java Pro wasn't enough. You also need to delve a little bit into Python because of your AI/ML specialist colleagues, who, by the way, don't speak Java.

That leads me to this article. The friction between enterprise-grade **JVM-based production environments** and Python's vast **ML/NLP ecosystem** is a common challenge for many developers. Historically, integrating the two meant complex and often slow inter-process communication (IPC).

In this article, our thesis is to demonstrate an alternative way: by seamlessly embedding Python code directly within a Java + Spring Boot API using **GraalPy**. We'll also set up the Project as a Spring Boot microservice so that it can be easily run within any [containerised deployments](#).

Why Combine Java and Python?

Java is an undisputed champion for building large-scale, resilient, and high-performance enterprise applications, thanks to its robust type system, mature tooling, and strong community. Python, on the other hand, excels in scientific computing, data analysis, and machine learning due to its simple syntax and a rich ecosystem of libraries.

Combining these two languages allows you to leverage the best of both worlds. Imagine a Java microservice that needs to perform quick, on-the-fly sentiment analysis on user feedback in real-time. Rather than setting up a separate Python service and dealing with HTTP, gRPC or another form of interprocess connection, you can now use a library like [TextBlob](#) directly from your Java code. Common use cases for this **polyglot** approach include sentiment scoring, rapid experimentation with machine learning models, and building efficient, self-contained microservices. This is where GraalPy, Oracle's GraalVM Python runtime, comes in.

Overview of the PoC Project

The proof-of-concept (PoC) project is a simple **Java Spring Boot REST API** that calls Python sentiment analysis logic. The key is that, thanks to **GraalPy**, there's no need for subprocesses or HTTP bridging. The Python code is pure, in-memory embedding within the same Java Virtual Machine.

Here's a high-level look at the architecture: a REST controller receives a string of text and passes it to a Java service. This service then uses the GraalPy runtime to invoke a Python function, which uses TextBlob to calculate the sentiment polarity and subjectivity. The results are then returned to the Java service and sent back as a JSON response.

The magic happens with the **Polyglot API**, which allows Java and Python to communicate as if they were one language. A simple snippet of this "glue code" looks like this:

```
1 // Java code to invoke the Python function
2 try (Context context = GraalPyResources.createContext()) {
3     context.eval(Source.newBuilder("python", new File("sentiment_analyzer.py")).build());
4     Value sentimentFunction = context.getBindings("python").getMember("analyze_sentiment");
5     Value result = sentimentFunction.execute(text);
6     // Process the result
7 }
```

Developer Setup and Project Structure

Setting up the project is quick.

1. **Clone the repository:** `git clone https://github.com/vshanbha/graalpy-sentiment`
2. **Java + Maven config:** The `pom.xml` file is configured to include the necessary **GraalPy dependencies**. It pulls in the `org.graalvm.polyglot` and `org.graalvm.python` libraries, which are essential for the interop classes.

3. **Python virtual environment setup:** The Python libraries (like TextBlob) are installed within a virtual environment, which keeps the project's dependencies isolated and manageable.

The project structure is a standard Spring Boot layout, with a key addition: the Python source files reside within the project's resource directory. The heart of the application is a Spring Boot REST controller that calls a service layer. This service layer is where the **Java-to-Python bridging** occurs, encapsulating the GraalPy logic and keeping the controller clean.

Core Code Snippets

Let's look at the two key parts of the application.

First, the Python side, which is simple and clean:

```
1 # sentiment_analyzer.py
2 import json
3 import time
4 from textblob import TextBlob
5
6 def analyze_sentiment(text):
7     start = time.time()
8
9     analysis = TextBlob(text)
10    sentiment_score = analysis.sentiment.polarity
11    sentiment_classification = (
12        "positive" if sentiment_score > 0.1 else
13        "negative" if sentiment_score < -0.1 else
14        "neutral"
15    )
16
17    end = time.time()
18    print(f"[Sentiment] Took {(end - start) * 1000:.2f} ms")
19
20    return json.dumps({
21        "score": sentiment_score,
22        "classification": sentiment_classification
23    })
```

All we do here is use the TextBlob library, pass it a text and get back the sentiment. We have also added a time taken print statement to help us get a sense of the time taken in the actual Python processing.

Next, the Java side, which uses the **GraalPy runtime** to load and invoke this Python code:

```

1 // Java SentimentService.java
2 @SpringBootApplication
3 @RestController
4 @RequestMapping("/analyze")
5 public class SentimentAnalysisController {
6
7     private final Context context;
8
9     private Value sentimentFunction;
10
11     public SentimentAnalysisController() {
12         this.context = GraalPyResources.createContext();
13     }
14
15     @PostConstruct
16     public void initialize() {
17         try {
18             // Load and execute the Python script
19             File scriptFile = new File("src/main/resources/sentiment_analyzer.py");
20             Source scriptSource = Source.newBuilder("python", scriptFile).build();
21             context.eval(scriptSource);
22
23             Value sentimentFunction = context.getBindings("python").getMember("
24                 analyze_sentiment");
25         } catch (IOException e) {
26             // Handle error
27             e.printStackTrace();
28         }
29     }
30
31
32     public ResponseEntity<String> analyzeText(String text) {
33         try {
34             if (analyzeSentimentFunction == null) {
35                 return ResponseEntity
36                     .status(HttpStatus.INTERNAL_SERVER_ERROR)
37                     .body("Python function not initialized.");
38             }
39
40             try {
41                 Value result = analyzeSentimentFunction.execute(text);
42                 return ResponseEntity.ok(result.asString());
43             } catch (Exception e) {
44                 System.err.println("Error analyzing sentiment: " + e.getMessage());
45                 e.printStackTrace();
46                 return ResponseEntity
47                     .status(HttpStatus.INTERNAL_SERVER_ERROR)
48                     .body("An error occurred while analyzing sentiment.");
49             }
50
51         }
52     }

```

GraalPy's magic is in its **type marshalling**, which handles the conversion of data types between Java and Python seamlessly. However, in this code, we have skipped that by using JSON strings due to the aim of sending an HTTP response as JSON.

Broadly, the above snippet has two important pieces

- An *initialize* function that looks up the relevant Python code file and loads it using GraalPy. This allows us to interact with the Python function as if it were a Java function.
- An *analyzeText* function that delegates the actual analyses to the Python function that was initialised at boot time.

GraalPy also leverages the JVM's **JIT (Just-In-Time) compiler**, which can optimise the combined code for surprisingly good

performance. More on that later.

Running the App & API Demo

After building the Spring Boot application, you can run it with `mvn spring-boot:run`. The API will be available at `http://localhost:8080/api/sentiment`.

You can test it with a simple `curl` command:

```
1 curl -X POST \  
2   http://localhost:8080/api/sentiment \  
3   -H 'Content-Type: text/plain' \  
4   -d 'This is an amazing and fantastic project.'
```

The response would be a JSON object, showing the calculated score and subjectivity. And with that, we are ready to use it in Production. Are we?

Proof Is in the Performance (or Was It Pudding)

The key takeaway here is performance. Because the Python code runs **in-process**, there is minimal latency, unlike a microservices approach that would involve network overhead. That's just not it, per GraalPy documentation, [the code can even run faster than CPython](#).

The GraalPy documentation claims that this is for the Python benchmark tests, and when using GraalPy with JIT compilation. However, I believe that the proof of the pudding is in eating it, or the proof of the performance is in testing it.

For the test Machine, I am using the following configuration

- **Hardware Model:** Apple Inc. Macmini6,2
- **Memory:** 16.0 GiB
- **Processor:** Intel® Core™ i7-3615QM × 8
- **Graphics:** Intel® HD Graphics 4000 (IVB GT2)
- **Firmware Version:** 429.0.0.0.0
- **OS Name:** Kali GNU/Linux Rolling
- **OS Type:** 64-bit
- **Kernel Version:** Linux 6.12.38+kali-amd64

Yeah, it's a really old Mac Mini with a Linux install. The hardware still works after 10+ years, so I decided that it could serve as a Linux test machine for such side projects.

Python Performance

Therefore, let's put this to the test and compare it with an equivalent pure Python API. We can use a simple Python Flask API router to wrap the Sentiment Analysis function and expose the output as an API endpoint. This is similar to what we have done already in Java:

```
1 @app.route('/analyze', methods=['GET', 'POST'])  
2 def analyze():  
3     if request.method == 'GET':  
4         text = request.args.get('text', '')  
5     elif request.method == 'POST':  
6         text = request.get_data(as_text=True)  
7     sentiment_json = analyze_sentiment(text)  
8     response = json.loads(sentiment_json)  
9     return jsonify(response)
```

Now that we have an equivalent API in both Java and Python, it's time to do a comparison of pure Python vs Java + GraalPy + Python. I fully expect that the complexity in the GraalPy approach will make things significantly slower.

We have designed a simple [JMeter](#) configuration that does the following

- Loads a CSV containing a bunch of different texts.
- Hits the Sentiment API multiple times based on a configured number of users (Threads)
- Each user runs calls to the same API a set number of times (loop count)

Since Python is where we would consider designing an API to consume Python libraries, we'll use Python readings as a baseline and as the numbers to beat.

Given that we are using Flask API, we can simply execute `flask run`, but Flask advertises that the Flask server is only for development. We'll instead use `gunicorn` to spin up a production equivalent:

```
gunicorn -b 0.0.0.0:8080 sentiment_analysis_api:app
```

Then we start the JMeter test for 10 users with 10 loops, and we observe the logs:

```
1 [2025-09-12 20:46:28 +0200] [10915] [INFO] Starting gunicorn 23.0.0
2 [2025-09-12 20:46:28 +0200] [10915] [INFO] Listening at: http://0.0.0.0:8080 (10915)
3 [2025-09-12 20:46:28 +0200] [10915] [INFO] Using worker: sync
4 [2025-09-12 20:46:28 +0200] [10916] [INFO] Booting worker with pid: 10916
5 [Sentiment] Took 35.19 ms
6 [Sentiment] Took 0.45 ms
7 [Sentiment] Took 0.39 ms
8 [Sentiment] Took 0.29 ms
9 [Sentiment] Took 0.39 ms
```

Note that the first hit takes a lot longer, but after that, the function runs really fast. However, when we run the same test with 1000 users and 100 loops, the situation is slightly different.

The sentiment analysis function still takes under 1ms, but the overall JMeter test yields a different result.

Across a load of 1000 users and 100 samples per user, **we get an average response time of 793 ms** and a 99% Line at 1012 ms.

So our target performance metric to beat under load is 1 second.

Java + GraalPy + Python Performance

We can get an executable Jar file by running `mvn clean install`. We start the server using:

```
java -jar target/graalpy-sentiment-v0.3.3.jar
```

And after a few seconds, we get this to indicate that the server is started.

```
1 2025-09-12T21:09:11.864+02:00 INFO 14014 --- [ main] c.example.
   SentimentAnalysisController
2 : Started SentimentAnalysisController in 23.72 seconds (process running for 24.408)
```

Almost 24 seconds for startup is likely not good if we want to run this in serverless mode, but for most Java use cases, I would argue that the warmup time is acceptable.

Then we run a test of 10 users to check this. The test presents some serious issues with performance. Every thread is taking a while to warm up before the performance improves.

```
1 [Sentiment] Took 12181.00 ms
2 [Sentiment] Took 13175.00 ms
3 [Sentiment] Took 14254.00 ms
4 [Sentiment] Took 13542.00 ms
5 [Sentiment] Took 13545.00 ms
6 [Sentiment] Took 15283.00 ms
7 [Sentiment] Took 16405.00 ms
8 [Sentiment] Took 16673.00 ms
9 [Sentiment] Took 16442.00 ms
```



```

10 [Sentiment] Took 16794.00 ms
11 [Sentiment] Took 1680.00 ms
12 [Sentiment] Took 1177.00 ms
13 [Sentiment] Took 806.00 ms
14 [Sentiment] Took 62.00 ms
15 [Sentiment] Took 394.00 ms

```

This is clearly an issue because at scale (1000+ users), the Warm-up time for each thread will make up for a very bad experience. However, that's not the only problem. If we repeat the test, we see Warm-up time issues showing up again. The overall JMeter results are just too bad.

While trying a run with 1000 users and 100 samples per user, the average time taken kept hovering around the 8 – 10 second mark. That just seems wrong, so contradictory to Oracle's claims of a GraalPy run being potentially faster.

Further investigation was certainly needed. The startup log actually had a warning:

```

1 [engine] WARNING: The polyglot engine uses a fallback runtime that does not support
  runtime compilation to native code.
2 Execution without runtime compilation will negatively impact the guest application
  performance.
3 The following cause was found: JVMCI is not enabled for this JVM. Enable JVMCI using -XX
  :+EnableJVMCI.
4 For more information see: https://www.graalvm.org/latest/reference-manual/embed-languages/#runtime-optimization-support.

```

Who looks at warnings anyway !! We only care about Errors, don't we? Turns out this warning was important and in fact the hint to the performance issues. Just like Java, Python also has a JIT compilation and the JVMCI piece isn't enabled in my JVM. For this test, I am still using an Older JVM (JDK 21) more out of backwards compatibility interest than anything else.

```

1 openjdk 21.0.8 2025-07-15
2 OpenJDK Runtime Environment (build 21.0.8+9-Debian-1)
3 OpenJDK 64-Bit Server VM (build 21.0.8+9-Debian-1, mixed mode, sharing)

```

Adding the -XX:+EnableJVMCI flag didn't really make a difference, and the warning continued. Further reading about [JVMCI](#) suggested that this could indeed be the culprit. My somewhat trusty AI friend suggested downloading GraalVM and using it instead.

So I now installed GraalVM and used that to run the same code, and then the warmup (1 user 10 samples) immediately showed a difference.

```

1 [Sentiment] Took 4173.00 ms
2 [Sentiment] Took 27.00 ms
3 [Sentiment] Took 33.00 ms
4 [Sentiment] Took 19.00 ms

```

The initial Warmup is slow, but the runs after warmup are significantly faster. So clearly, the change of JVM helps. However, the warm-up time is still significant, so we still need a Warm-up for a multi-user run. We can then run a larger test of 1000 users and 100 samples.

Post warmup, a load of 1000 users and 100 samples per user, **we get an average response time of 1411 ms**. That is certainly way faster than before. It is still slower than pure Python, but I would say that this is still certainly acceptable.

There may be some further tuning possible by using JVM flags. I have not focused on many other things, such as CPU and Memory usage or generally observability and logging. For use in Production at scale, I would say there is much more to do but I would leave that to the curious reader

In my test, the Java + Python polyglot service was slower than a pure Python solution. So if we already have services built fully in Python, I wouldn't rush to replace them with Java. However, if building new services or enhancing existing services with the use of Python libraries, this polyglot option is certainly worth consideration.

For now, it is sufficient to say that Java + GraalPy does offer a viable alternative to Python code integration into Java-based services. This concept is especially interesting for teams that have a large amount of Java code and need to embed specific libraries from Python into their code for some speciality use cases while avoiding the overhead of learning Python or maintaining significant portions of code that are built only in Python.

Lessons Learned & Challenges

This approach offers significant advantages: it's **lightweight, eliminates IPC** (Inter-Process Communication), and allows you to leverage powerful **JVM tooling** for monitoring, debugging, and profiling the entire application.

However, GraalPy is still a maturing technology. One of the main challenges is dealing with Python packages that rely on **native C extensions**, as they may require specific configurations or may not be fully supported yet.

Another important point to note is the use of the GraalVM as the JVM. The usage of GraalVM ensures that GraalPy is effective and performant. Without GraalVM, the performance is drastically slower.

As of the latest release, v0.3.3 (July 28, 2025), the project is stable for usage as a starter project for any Java + Python use cases. It also includes a few more features relevant to Production such as authentication, logging support, port changes etc.

Production Considerations

When moving this concept to a production environment, you must consider several factors.

- **Packaging:** Creating a **GraalVM native image** can drastically reduce startup time and memory footprint, making it ideal for containerised deployments. You'll also need to consider how to package the necessary Python dependencies. In my tests startup times were nearly 1/4th of startup times with a Jar packaging.
- **Performance Monitoring:** While in-process execution is fast, it's crucial to monitor memory and CPU usage. The shared memory space means that a memory leak in the Python side can affect the entire JVM.

The decision to choose embedding vs. a separate microservice (using Docker, REST, or gRPC) depends on your specific needs. Embedding is best for low-latency, tightly coupled functionality, while a microservice approach offers more scalability and fault isolation. In this article, we have made a case for those attempts.

Conclusion

We've demonstrated how **GraalPy** provides a simple and effective solution for integrating Python's powerful NLP libraries directly into Java applications. By seamlessly embedding Python code, we can bypass the complexities of traditional IPC methods, resulting in a cleaner, faster, and more unified architecture.

This proof-of-concept shows that combining the enterprise strength of Java with the analytical power of Python is not only possible but can also be effortless. I invite you to explore the source code, try it out for yourself, and contribute to the project. Your feedback and contributions can help mature this powerful polyglot technology.

Try the project and contribute on GitHub: <https://github.com/vshanbha/graalpy-sentiment>

References

- TextBlob: <https://textblob.readthedocs.io/en/dev/>
- GraalPy: <https://www.graalvm.org/python/docs/>
- GraalVM Python Runtime: <https://github.com/oracle/graalpython/releases>
- Spring Boot Native Images: <https://docs.spring.io/spring-boot/reference/packaging/native-image/index.html>

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



The Michael Jordan Approach to Software Engineering

Leadership

Self-Improvement

Teamwork

Author: Erik Pronk

Erik Pronk is Managing Director of JDriven and a passionate software engineer. Primary focus on sharing knowledge and facilitating the personal and professional growth of colleagues and community members.



LinkedIn

When we think of greatness in the world of sports, one name that immediately pops in my mind is Michael Jordan. Widely regarded as the greatest basketball player of all time, Jordan's approach to his craft is legendary. While the world of software engineering might seem miles apart from the basketball court, there are valuable lessons we as software engineers can learn from the Michael Jordan approach and apply them to our work as software engineers.



Focus on Fundamentals

Jordan's greatness on the basketball court came from his understanding of the basics. His understanding of the game's fundamentals was impeccable, excelling in areas like shooting, ball-handling, and defense. His grasp of these essentials allowed him to make the right decisions under pressure, whether it was finding a teammate for an assist or taking the game-winning shot.

In software engineering, mastering the fundamentals is equally critical. A deep understanding of clean coding practices, data structures, algorithms, and coding principles forms the foundation of building reliable and scalable software.

It's tempting to dive headfirst into the latest frameworks, libraries, or technologies. However, just as Jordan built his career on a solid foundation of basketball fundamentals, software engineers who focus on mastering the basics can build systems that are efficient, maintainable, and scalable. Without this solid foundation, even the most cutting-edge technology stack will collapse under the weight of inefficiency, technical debt, or instability.

Work Ethic

Michael Jordan's work ethic was legendary. While his talent was obvious, it was his constant dedication to improvement that set him apart. Jordan wasn't just gifted, he worked harder than anyone else. Whether it was showing up to practice earlier than his teammates or staying late to perfect his jump shot, he was focussed on pushing his limits. He didn't fear failure; instead, he embraced it as a learning opportunity.

Jordan once said, *"I've missed more than 9,000 shots in my career. I've lost almost 300 games. Twenty-six times, I've been trusted to take the game-winning shot and missed. I've failed over and over and over again in my life. And that is why I succeed."*

This mindset of learning from mistakes and continually striving for greatness is what made him a champion on and off the court.

In software engineering, this work ethic translates to a commitment to constant learning and improvement.

Building high quality software isn't just about talent; it's about putting in the effort, making mistakes, and learning from them. Just like Jordan would practice free throws for hours, software engineers need to code, test, and debug to improve their skills.

Embracing Short-Term Pressure

Michael Jordan grew under high-pressure situations on the basketball court. Whether it was a game-winning shot or a crucial playoff match, he welcomed the challenge, seeing it as an opportunity to showcase his skills and rise to the occasion.

Similarly, software engineers should learn to embrace high-pressure scenarios. These moments characterized by tight deadlines, critical bugs, or demanding stakeholders, are opportunities to develop problem-solving abilities, adaptability, and resilience. By embracing short-term pressure you grow your software engineering skills and your mindset, allowing you to tackle complex projects with grace and effectiveness.

Continuous Self-Improvement

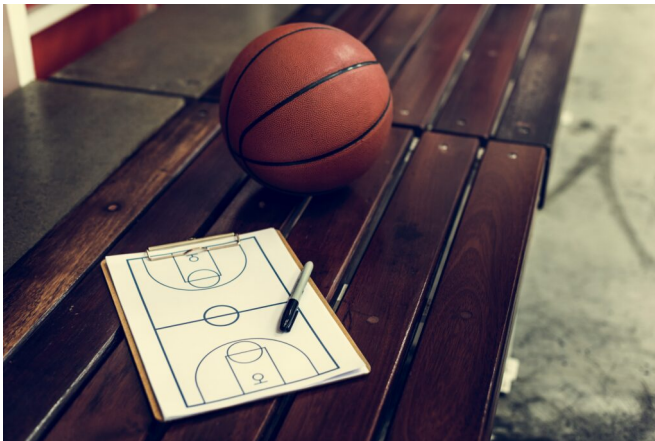
Even at the peak of his career, Michael Jordan constantly sought ways to improve his skills. He didn't rest on his laurels, he adapted to changes in the game, worked tirelessly to refine his skills, and pushed his limits to stay ahead of the competition. Whether it was mastering his fadeaway jump shot, improving his defense, or becoming a better leader for his team, Jordan embodied the principle of continuous self-improvement.

In software engineering, this same mindset is crucial. The tech landscape evolves rapidly, with new languages, frameworks, and tools emerging regularly. Adopting the "Michael Jordan approach" means embracing change and committing to lifelong learning to remain relevant, effective, and innovative in your craft. This means investing in yourself and staying up to date with trends, attending workshops and conferences, and adapting to new technologies. Create your own or your companies Tech Radar. What are the areas you want to explore and what are the technologies you want to say goodbye to.

Teamwork and Leadership

While Michael Jordan was a dominant individual player but he recognized that basketball is a team sport. His leadership and ability to inspire his teammates were powerful ingredients in the success of the Chicago Bulls, leading to six NBA championships. Jordan knew how to bring out the best in those around him, motivating his team to perform at their highest level at the right time and creating an environment where everyone could grow.

In software engineering, collaboration is equally critical. The most successful projects are built on effective communication, teamwork, and strong leadership. Just as Jordan elevated his team, you can help lead your software development team to success by fostering a positive work environment, encouraging collaboration by stimulating code reviews and pair programming, and helping every team member excel.



Staying Healthy and Looking After Yourself

As software engineers, we often spend long hours sitting at our desks, staring at screens, and diving deep into complex problems. It's easy to fall into the trap of prioritizing work over health. However, just as athletes like Michael Jordan prioritize their physical and mental well-being to perform at their peak, software engineers must also take steps to look after themselves. Neglecting your health can lead to reduced productivity, burnout and long-term consequences for both your work and personal life.

It's essential to find a healthy work-life balance, take breaks, and get regular exercise.

Also, don't underestimate the importance of mental health. Practice stress-reduction techniques, seek help when needed, and make time for activities you enjoy outside of work.

Remember, staying healthy and looking after yourself is a strategy for long-term success in your software engineering career.

To Sum It All Up

The Michael Jordan approach to software engineering is not about dunking a basketball but rather about achieving greatness in your field.

It's about work ethic, focusing on fundamentals, not being afraid of short-term pressure situations, continuously improving, and being a team player.

Here are five practical tips to help you adopt a similar mindset in your software engineering career:

1. Practice, Practice, Practice

- Just as Michael Jordan spent countless hours on the basketball court, you should dedicate significant time to practice your software development skills.
- Work on personal projects, contribute to open-source projects, and continuously seek opportunities to code and solve real-world problems.

2. Set Clear Goals

- Define clear and achievable goals for your growth, both personally and professionally, as well as project-specific objectives.
- Use these goals to motivate yourself and maintain a sense of purpose in your work.

Regularly review and adjust them as needed.

3. Learn Continuously

- Be a continuous learner. Technology evolves rapidly, so stay updated with the latest developments in your field.
- Take online courses, attend conferences or workshops, and read books or articles to enhance your knowledge and skills.

4. Embrace Failure and Criticism

- Michael Jordan faced numerous setbacks and criticisms throughout his career.

Similarly, in software engineering, you will encounter challenges and possibly criticism.

- Learn from failures and view constructive criticism as an opportunity for growth.

Don't be discouraged by setbacks; instead, use them as stepping stones to improvement.

5. Mentorship and Collaboration

- Surround yourself with colleagues and mentors who can guide and inspire you.
- Collaborate with experienced engineers, participate in code reviews, and seek feedback to refine your skills and develop a winning team dynamic.

By embracing these principles and finding the right balance that prioritizes our well-being, we, as software engineers, can strive to become the best versions of ourselves. This journey is not just about personal growth but also about making a meaningful impact on those around us, our colleagues, teams, and the broader tech community.

Beyond that, our contributions can leave a lasting imprint on the world of technology, shaping the world of tomorrow.

To close off with a personal note, I have always believed that there are many aspects of competing at the highest level in sports that overlap with our daily lives and work. By observing and adopting key elements from top athletes, their discipline, resilience, and continuous drive for improvement, we can enhance our own knowledge and expertise.

But perhaps the most valuable lesson is the understanding that success doesn't come easily. It requires consistent effort, perseverance, and a willingness to actively step out of our comfort zone and embrace discomfort as part of the growth process.



Conferences. How and Why

Conferences

DeveloperLife

CareerDevelopment

Author: Ina Bumstein

Ina Bumstein is an information technology specialist developing and architecting Java based solutions for more than 18 years. Almost in each new project dealing with something new or try finding better way to do familiar tasks with new possibilities available. Always interested in what is going on in the field, exploring different approaches for the problem at hand. Passionate about knowledge sharing and seeing it as important way of learning. Leading Java topic related information exchange initiative at workplace and part of Vilnius JUG organisers team.



LinkedIn

Ever thought of attending a conference, but decided it would be too much effort and overwhelming with so much great content already available online to give it a try? It is even more difficult to decide on attending if you have had a negative experience in the past.

My first attempt many years ago wasn't bad. A full day of going from one talk to another. Plus, feeling sleepy due to an early start. However, after attending the conference I didn't feel I'd want to do it again. Something was missing. There is no guarantee that your visit will be successful as there are too many aspects to take in account, but there are things you can do to increase the chances. I'll share with you my approach on how to do that.

First Step: Choosing

My view on the conferences changed significantly in 2019. At that time I was working on a solo development project and wanted to clear my mind. I heard a lot of good things about the Spring I/O conference in Barcelona. Nice location and a suitable place to learn more about the framework I use daily. It sounded like a perfect option. This leads me to the first tip:

Choose the event you like based on location and topics you want to learn about. Do your own research. Ask colleagues or friends who have been there if they liked it.

My rule of thumb: never join the conference just because it is free, somebody is pushing you to or paying for it on your behalf. Remember, you are investing your time, so it should be something that you personally want to do or see benefit in.

When making a decision you can also think of a conference as a holiday extension. If you want to visit a specific place and there is a conference in or nearby, you can check it out.

And yes, I prefer in person events. When joining remotely, you can't fully appreciate the energy. You are just constantly looking at your monitor screen. Having said that, online conferences are a good alternative, especially if you have found one matching your interests or prefer not to travel for whatever reason.

Second Step: Before the Event

To get the most of it, some preparation is useful. What I would recommend doing:

- Check the schedule beforehand and mark sessions you would like to attend. It can be several sessions at the same time.

The goal is to have favourite things highlighted to save time during the event. When there are 5 sessions in parallel and you have narrowed your choice to 2/3 most appealing, it helps making the final decision quicker as you don't need to go through all 5 again.

Important: The session list created is not a strict plan. Use it as a guide. At the same time, allow yourself to change your mind.

- Pay attention to what is recorded and what's not. Workshops almost never are. Don't miss the opportunity if you want to attend any of those.

Step Three: During the Event

So you are at the conference. You have registered and got your badge. What's next? People are buzzing around.

Back to my Spring I/O 2019 story. I do vividly remember myself standing there watching people around me and trying to figure out my plan for the day. At that moment a guy came by and said:

"Hi, first time?"

I replied, "Yes".

He said that I shouldn't be just standing like that, it is not how conferences are done.

Then he explained and showed me his way. Just use this simple question to break the ice like:

- Where are you from?

- Is it your first conference “A”?

Don't forget that you definitely have some common topics with other participants like sessions' content and technology you are using. If you are at a Java related conference, you can always talk about Java.

For introverts like many developers are, this could look scary, but trust me, it is worth it. I had the most interesting conversations and made the most valuable connections by using this technique and being open to talk. The so-called halfway track is memorable and will contribute to your unique experience.

Going to conferences nowadays I try to be that person to someone else by sharing tips like I'm sharing with you.

Besides the networking way I have mentioned earlier, there are a couple more tips regarding the sessions.

I always wonder when the majority is pushing quickly to the door of the auditorium at the end of the session. Finding courage to ask questions on camera might be challenging, but going and asking afterwards is good both for you and a speaker. Maybe you will inspire their next talk or you find yourself hacking with them trying to solve your problem. Also, listening to others' questions could be interesting. If you rush to the next thing you will miss this chance. Slow down a bit. That fear of missing out is your enemy. It also could cause your mind to block due to too much information.

Tip: Skipping a session is perfectly fine. You might be simply tired. Why push yourself if you are no longer paying attention? Or you are engaged in active discussion. Why stop it?

Step Four: After the Event

When the last session ends many think that the conference is over. In reality it is not like that. The final step is still to be done.

One of the things I always try to do is structuring my notes into summary. What I have seen, what my takeaways are etc. It helps me better to remember things and reach out for exact pieces of data when I need it to solve a task at hand. The final result can be:

- Just a summary for yourself.
- Post in your internal company format to share with colleagues.
- Public blog post.
- Source for the presentation for your colleagues or local community.

If sessions are recorded, watching ones that you couldn't attend or were highly rated by others, could be a nice touch.

Final Thoughts: The Why

Now you know all these things, let's try answering the question: why treat yourself to a conference?

For me it is:

- Networking. Most efficient at in person events.
- Possibility to ask questions – you can't do that when watching recorded content.
- Learning new things in a focused and fun way.
- Changing scenery and being out of your comfort zone from time to time is healthy for you.

Joining a conference will broaden your horizons and teach you new skills in the areas you haven't imagined it could. Participating in a knowledge sharing event can be exhausting and at the same time exciting and rewarding. Give it a try.

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



Design-first APIs for UI Components

WebComponents

APIDesign

FrontendArchitecture

Author: Lucien Immink

A Java developer that saw the light and embraced Clippy as his digital overlord. Over two decades in the field has taught Lucien how we often reinvent the wheel to accommodate new shiny frameworks. Now, as a Principal Consultant at Team Rockstars IT in the Netherlands, his mission is to discover elegant solutions for his clients. While we've moved beyond Flash and jQuery, the web platform technologies: HTML, CSS, and JavaScript remain essential. Rather than battling the platform, let's embrace it. Lucien Immink is a rewarded Google Developer Expert for Web Technologies and capabilities since 2022.



LinkedIn

Oh how the internet has changed. Where it began with HTML, static pages with minimal to no styling to a platform that can do almost everything we do with computers. Maps, games, productivity tooling and of course websites. They are all still presented using HTML, CSS and JavaScript.

One major thing that did change is what kind of people are needed/working on these applications. Back- and front-end developers, designers, product owners, scrum masters, users, library owners, library consumers, framework authors. The list goes on and on. The reason is because the work became too big, too complex, for a single person to handle; at least in most enterprise areas.

People that work together rely on common knowledge and documentation in order to understand what needs to be done (Definition of ready) and when it is done (definition done). People *communicate*.

Communication

Human communication is based on a set of rules. The more you use the rules the better you are in understanding the complex task at hand. The more you *use* and *repeat* the better you get at it. This process is of course called *learning*. Learning a new language is all about understanding the rules of that language. You also need to be able to properly translate the nuances of that new language to your inner language in order to understand the true meaning of what is communicated, the delicate process of nuance.

Listening to a person (input) and talking to someone (output), but also reading (input) and writing (output) are based on the language rules and both involved parties will greatly benefit from understanding the rules the language and the rules of communication (it's not polite to interrupt a person).

APIs

These same rules apply to computer communications. If a website want to know the inventory of a store back-end it needs to know what it should request and what to expect (response). These communication rules are called APIs, Application Programmer Interfaces. The more the consuming application knows about the rules of the providing service, the better. As developer you can define these rules yourself, but that makes it harder to have other systems integrate with your system since your rules might differ from the standards. A custom API is your own secret language.

An example of a defined standard for Restful APIs is called OpenAPI which defines a document that lays down the rules of the application like the version (soft of dialect) the security measures, the URLs, the paths and components to use.

Everything the consuming application needs in order to properly communicate to and from the given application.

UI Components

The same could be said for UI components. UI components used to be a part of the same system as the UI but that has changes a lot in recent times. Large companies use a design system to lay down the rules for how UI components need to be used, others provide a component library for others to use; material UI from Google for example. In both cases the team responsible for the component is not responsible for the UI of any or all applications that are built using those components.

Design first

Applications evolve and so do their APIs. A team could start with building the basic application and offer that to other teams but they can also sit down with all involved teams and team members to think about the API first. Think about what the best way is to provide and consume data from both sides of the API. The latter approach is called design-first and offers a way where stakeholders don't have to wait on an implementation before they can start work: all involved parties know what they can expect and what they need to deliver. Using tooling that is available, since the API is based on a standard, stakeholders can verify their work without any implementation!

Is a design-first approach possible when it comes to UI components? Let's find out!

Framework agnostic

One important thing that needs to be addressed first is that if the UI components can be consumed by other teams is that these components need to be framework agnostic. The component team is not responsible for making the choice which framework suits best for a given application. Even if both teams use the same framework it can still be problematic if the component is written in an older, or newer, incompatible version of the same framework.

This is where web components come in. Web components is a relative new addition to the browser platform and allow authors to create their own HTML like tags, called custom elements. Custom elements can be recognized because they have a dash(-) in their name like *my-element*, *hello-world* or *trsit-date-picker*. A powerful (but also sometimes problematic) part of web components is the shadow-dom which is a separated document object model that “hides” the dom from the outside (now called light-dom). Most styles and logic can leak out, but also can’t leak in. Perfect for the cases where your design system dictates a padding or some specific colour.

Web-Components API

Since web components are part of the browser platform they follow the *exact* same rules as all HTML elements do. Input is handled by properties and *slots* (a way to control the placement of light-dom elements within a web components), output is managed by firing and listening to events. For styling CSS is used and part of the CSS spec are CSS variables that can be set and overridden from the light-dom.

This all means that web components have a very predictable API, which we can use to our advantage: OpenAPI for web components; or Custom Elements Manifest.

Custom Elements Manifest (CEM)

Many tools need some machine-readable descriptions of custom elements: IDEs, documentation viewers, linters, graphical design tools, etc.

Custom elements manifest is an effort to bring together tool owners to standardize on a common specification for a description format.

CEM analyser is one of tools that use CEM and acts as an analyser for web components. Standard JavaScript and vanilla web components are supported by default. Dedicated web component libraries can be supported through the use of plugins. Support for Lit, Stencil, Fast and catalyst is currently available.

Although not pure design first, but by creating a skeleton component we can use CEM analyser to generate the CEM which can then be used for further tooling. Of course it is also possible to write the CEM first; true design-first and use it to generate the skeleton component.

Skeleton components

A skeleton component holds just enough information to properly define the API but nothing more. Don’t implement styling beyond the CSS variables, don’t add visuals that are not part of the API. This skeleton is your base UI component. What needs to be done though is: Define slots, events, properties and CSS variables, in other words: define the API of the UI component.

Create the skeleton component

In this example Lit is used, but any of the web component libraries supported by CEM analyser can be used.

```

1  import { html, css, LitElement } from 'lit';
2  import { customElement, property } from 'lit/decorators.js';
3
4  @customElement('hello-world')
5  export class HelloWorld extends LitElement {
6    static styles = css`
7      :host {
8        display: block;
9        background-color: var(--background-color, white);
10     }
11     p {
12       color: var(--text-color, black);
13     }
14   `;
15   @property()
16   type = 'wonderful';
17
18   private fire() {
19     this.dispatchEvent(new Event('[Bonus]'));
20   }
21   private fire2() {
22     this.dispatchEvent(new Event('[Risky]'));
23   }
24
25   render() {
26     return html`<p>Hello ${this.type} world</p>
27       <p>
28         <button @click=${this.fire}></button>
29         <button @click=${this.fire2}></button>
30       </p>`;
31   }

```

This will result in

Hello wonderful world



With 1 property *type*, 2 events: *[Bonus]* and *[Risky]* and 2 css variables: *--background-color* and *--text-color*.

This is actually enough for CEM analyser to generate the CEM but without any human readable comments. To improve the CEM JavaScriptDoc can be used, let's add some!

```

1  import { html, css, LitElement } from 'lit';
2  import { customElement, property } from 'lit/decorators.js';
3
4  /**
5   * @cssprop [--text-color=black] - Controls the text colour
6   * @cssprop [--background-color=white] - Controls the background colour
7   */
8
9  @customElement('hello-world')
10 export class HelloWorld extends LitElement {
11   static styles = css`
12     :host {
13       display: block;
14       font-size: 2em;
15       padding: 1em;
16       background-color: var(--background-color, white);
17     }
18     p {
19       color: var(--text-color, black);
20     }
21   `;
22   /**
23    * What type is the world today?
24    */
25   @property()
26   type = 'wonderful';
27
28   private fire() {
29     /** @type {Event} gift - I'm bearing a gift */
30     this.dispatchEvent(new Event('[Bonus]'));
31   }
32
33   private fire2() {
34     /** @type {Event} bomb - This was a mistake... */
35     this.dispatchEvent(new Event('[Risky]'));
36   }
37
38   render() {
39     return html`<p>Hello ${this.type} world</p>
40       <p>
41         <button @click=${this.fire}></button>
42         <button @click=${this.fire2}></button>
43       </p>`;
44   }
45 }

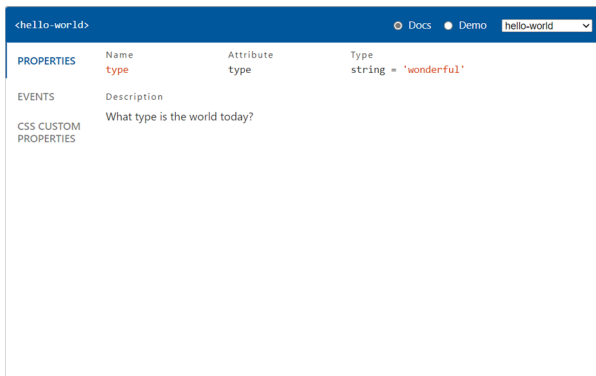
```

All lines that start with `/**` are treated as comments and by using the JavaScriptDoc annotations helpful text is added, directly in code, which is read by CEM analyser.

After running CEM analyser the CEM is generated from this skeleton component; now comes the fun part: using this API doc!

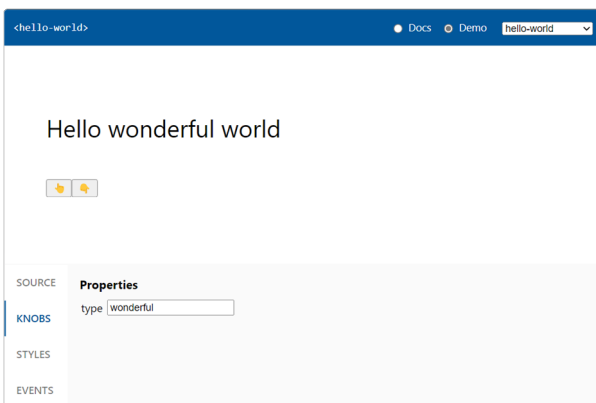
API Viewer/Explorer

`<api-viewer>` is a web component that read the CEM and uses it to visualize the API



On the left hand side the input/output is grouped by types: properties, events, slots, CSS variables, etc. Each tab shows the instances of that tab; in the example above the properties are shown, including the human readable comments that were added.

But why stop there? The component is fully operational and just like for example storybook it's possible to change the knobs of the component:



Change some settings and the updated state of the component is reflected in the source tab:



Tip: let the analyser run after each commit to keep the CEM 100% up-to-date. Up-to-date documentation and interactive playground within seconds!

AAAG

...but why stop there? The beauty of using a standard is collaboration, not only within a team or company but in a whole ecosystem. Developers and teams from all over the world can use CEM to add more tooling which can be used, regardless of what framework is used for the rest of the application.

- Interactive demonstrations
- Linting

- Testing
- Cataloguing
- Integrations
- Documentation
- ...

Automation as a Gift.

AAAG Examples

- **CEM-to-markdown** used CEM to create a markdown file holding the API of the actual state of the UI components; ideal to be added to the GIT repo or documentation site.
- **cem-plugin-reactify** will create react < 19 wrapper components for the custom elements.
- **custom-elements-vs-code-integration** will generate the integration json files needed to fully support the custom elements in VS code.
- **Storybook** has native support for CEM

Getting Started

To get started here are some great resources:

- [lit.dev](#) (web component library)
- [custom-elements-manifest.open-wc.org](#) (CEM)
- [custom-elements-manifest.open-wc.org/analyzer/plugins/intro/](#) (AaaG)
- [skeleton-component-generator](#) (CEM > component generator)

Wrapping Up

Human communication is based on a set of rules. The more you use the rules the better you are in understanding the complex task at hand. The more you use and repeat the better you get at it. This process is of course called learning. Learning a new language is all about understanding the rules of that language.

Applications evolve and so do their APIs. A team could start with building the basic application and offer that to other teams but they can also sit down with all involved teams and team members to think about the API first. Think about what the best way is to provide and consume data from both sides of the API. The latter approach is called design-first and offers a way where stakeholders don't have to wait on an implementation before they can start work.

Web components is a relative new addition to the browser platform and allow authors to create their own HTML like tags, called custom elements.

Many tools need some machine-readable descriptions of custom elements: IDEs, documentation viewers, linters, graphical design tools, etc.

Custom elements manifest is an effort to bring together tool owners to standardize on a common specification for a description format.

Although not pure design first, but by creating a skeleton component we can use CEM analyser to generate the CEM which can then be used for further tooling. Of course it is also possible to write the CEM first; true design-first and use it to generate the skeleton component. The beauty of using a standard is collaboration, not only within a team or company but in a whole ecosystem. Developers and teams from all over the world can use CEM to add more tooling which can be used, regardless of what framework is used for the rest of the application. True Automation as a Gift.

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

JAVA 25



Security in the Age of Java 25: New Language Tools for Safer Code

Cryptography

Immutability

Pattern Matching

Sealed Classes

Security

Author: Mihaela Gheorghe-Roman

Mihaela Gheorghe-Roman is a Software Architect at Systematic, Romania. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge at work, at the university, at different national and international IT conferences. In her professional career, spanning over 14+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular. Apart from her passion for programming, she loves parrots, she is enthusiastic about traveling the world and doing a lot of sports, from running at marathons, martial arts and to different aerobic trainings.



LinkedIn

Security is more than just encryption – it’s also about how you write code. There are several Java security features that can help prevent logic bugs, misuse of types, and unexpected state mutations. Learn how sealed classes, pattern matching, and records can reinforce domain modeling, reduce null checks, and prevent injection-prone scenarios.

Introduction: Why Java security features Matters More Than Ever

Application security has never been a static field. Over the past two decades, the Java platform has evolved not just as a runtime for enterprise applications but as a trusted foundation for industries ranging from finance and healthcare to government systems. Security breaches, once attributed primarily to poor encryption or missing authentication, now more often stem from subtler flaws: unchecked input, leaky abstractions, and poorly modeled domain logic.

Java 25 arrives at a time when developers must treat security not just as a matter of external libraries or firewalls, but as a language-level concern. While Java’s traditional strengths – the JVM sandbox, bytecode verification, and a strong type system – remain critical, the new features in Java 25 push the security frontier even deeper into how we write our code.

In this article, we’ll explore Java security features like sealed classes, pattern matching, and records as first-class security enablers. We’ll see how these features, when used wisely, not only reduce common logic errors but also close doors to injection attacks, privilege escalation, and state corruption.

Java Security Features Beyond Cryptography

When developers hear “Java security features”, they often think of:

- Transport Layer Security (TLS)
- Public key infrastructure (PKI)
- Password hashing libraries

All of these remain crucial. However, just cryptography cannot save a system whose domain model is broken. If a web service accepts an inconsistent request, stores it in a mutable object, and later passes it unchecked to a SQL query, no amount of SSL will prevent exploitation.

Modern security is all about writing code that’s safe by design. The new language features introduced in recent Java releases – and refined in Java 25 – put that philosophy into practice.

Sealed Classes: Preventing Inconsistent State

The Problem: Open Inheritance as a Security Hole

Traditionally, Java’s open inheritance model allowed anyone to extend a class. While this flexibility was convenient, it introduced subtle risks:

- Attackers (or careless maintainers) could create subclasses that violate invariants.
- Switch statements over enums or class hierarchies could miss unexpected implementations, leading to logic gaps.
- Security-critical code relying on exhaustive checks could be bypassed.

The Java Security Features Solution: Stronger Sealed Classes

Sealed classes were introduced in earlier versions, but Java 25 enhances their ergonomics and guarantees. By explicitly declaring which classes may extend a base type, developers lock down their hierarchies.

```
1 public sealed interface AuthenticationResult
2     permits Success, Failure, LockedOut {
3 }
4
5 public record Success(String username) implements AuthenticationResult {}
6 public record Failure(String reason) implements AuthenticationResult {}
7 public record LockedOut(String username, int remainingTime) implements
    AuthenticationResult {}
```


Now, only these three types can represent authentication outcomes.

Security Benefits

By narrowing inheritance to only the types you explicitly allow, sealed classes change the way we think about designing secure systems. Instead of relying on documentation or runtime checks, the compiler itself enforces the boundaries of your model. This shift turns what used to be a matter of discipline into a matter of guarantee, making it much harder for inconsistent or malicious code to sneak in.

Here are some of the key security advantages you gain by using sealed classes in Java 25:

Exhaustiveness When handling results, the compiler ensures you cover every possibility:

```
1 static void handle(AuthenticationResult result) {
2     switch (result) {
3         case Success s -> log("Welcome back, " + s.username());
4         case Failure f -> log("Login failed: " + f.reason());
5         case LockedOut l -> alert("Locked out: " + l.username());
6     }
7 }
```

If you forget one, the compiler reminds you. This prevents unhandled states from slipping through.

Tamper resistance No external code can forge a subclass with malicious payloads. This ensures that only known, vetted types exist in your hierarchy, reducing the risk of bypassing security checks. It also makes auditing and reasoning about the code much easier, because developers and security teams can be confident that all possible subclasses are visible and controlled at compile time.

Clear domain modeling By making all possible subclasses explicit, sealed classes give you a complete view of your domain. This transparency makes security-sensitive logic easier to audit and reason about, since you can see every valid case at a glance and ensure that no unexpected types can slip through.

Pattern Matching: Safer Type Inspections

The Problem: Unsafe Casting and Verbose Null Checks

Older Java code relied heavily on `instanceof` followed by manual casting. This practice was both verbose and error-prone:

```
1 if (obj instanceof User) {
2     User u = (User) obj; // cast required
3     if (u != null && u.isActive()) {
4         // ...
5     }
6 }
```

Developers often forgot null checks, or incorrectly assumed types, leading to runtime exceptions or even injection opportunities when unchecked data flowed downstream.

The Java 25 Solution: Pattern Matching Everywhere

Pattern matching goes beyond just reducing boilerplate. It allows developers to express intent clearly, letting the compiler enforce correct handling of each case. By combining pattern matching with sealed classes and records, you get a type-safe, declarative way to process inputs, reducing the likelihood of runtime errors and security vulnerabilities.

Java 25 expands pattern matching for `switch`, `instanceof`, and even record deconstruction. This creates safer, more declarative ways to handle data.

The following snippet shows how it allows you to handle different input types safely and concisely. Each case is checked at compile time, reducing the risk of runtime errors or unhandled cases:

```

1 static void process(Object input) {
2     switch (input) {
3         case User(var username, var email) ->
4             log("Valid user: " + username + " / " + email);
5         case Admin(var level) ->
6             log("Admin access level: " + level);
7         case null ->
8             log("Null input blocked");
9         default ->
10            log("Unknown type");
11     }
12 }

```

Security Benefits

Using pattern matching consistently not only makes your code cleaner, it also enforces security at compile time. Some of the most important safety advantages include:

- **Null safety:** Patterns can match `null` explicitly, ensuring developers account for it.
- **Exhaustiveness:** Just like sealed classes, pattern switches force explicit handling of all cases.
- **Cleaner code, fewer mistakes:** Less boilerplate means fewer opportunities to forget a check.

Records: Immutable by Default

The Problem: Mutable State and Security Bugs

Mutable objects are a common source of security flaws; for example, consider a web request object, which could be altered unexpectedly during processing.

```

1 class UserRequest {
2     String username;
3     String role;
4     // setters, getters
5 }

```

If an attacker manages to alter `role` between validation and authorization, privilege escalation may occur.

Java Security Features Solution: Domain Models as Records

Records, introduced earlier, become a cornerstone of secure modeling in Java 25. They are immutable, transparent carriers of data.

```

1 public record UserRequest(String username, String role) {}

```

Once created, a `UserRequest` cannot be mutated. Security checks performed at construction remain valid for the object's lifetime.

Security Benefits

Immutability is one of the most effective tools for preventing subtle runtime attacks. By ensuring that objects cannot be altered after creation, developers can reduce the chances of unexpected behavior or state manipulation. Some key benefits of Java security features in the form of Records include:

- **Immutability:** Prevents time-of-check-to-time-of-use (TOCTOU) attacks.
- **Conciseness:** Less code means fewer places where bugs can hide.
- **Serialization safety:** Records integrate well with frameworks, reducing custom parsing errors.

Case Study 1: Preventing SQL Injection with Safer Modeling

One of the most common attacks – SQL injection – often arises from treating user input as plain strings. Traditionally, developers would write code like this:

```
1 String query = "SELECT * FROM users WHERE username = '" + input + "'";
```

Even with prepared statements, careless concatenation creeps back in. Now, we can model safe query parameters as sealed classes and records.

```
1 public sealed interface QueryParam permits SafeString, SafeInt {}
2 public record SafeString(String value) implements QueryParam {
3     public SafeString {
4         if (value.contains(";"))
5             throw new IllegalArgumentException("Invalid input");
6     }
7 }
8 public record SafeInt(int value) implements QueryParam {}
9 public record UserLookup(SafeString username) {}
```

By wrapping inputs in validated records, injection risks are eliminated at the boundary. No unsafe string concatenation is possible without deliberately breaking the model.

Case Study 2: Java Security Features – Microservices Boundaries

In a microservices architecture, APIs often exchange JSON payloads. A poorly designed DTO (data transfer object) may allow attackers to slip in fields that backend services don't expect.

With sealed classes and records, the domain becomes closed:

```
1 public sealed interface PaymentCommand permits Initiate, Cancel {}
2 public record Initiate(String fromAccount, String toAccount, double amount) implements
    PaymentCommand {}
3 public record Cancel(String transactionId) implements PaymentCommand {}
```

When deserializing, developers can configure frameworks like Jackson to reject unknown subtypes. As a result, clients cannot forge new command types.

Pattern matching then ensures services handle every valid command:

```
1 switch (command) {
2     case Initiate(var from, var to, var amt) -> processInitiation(from, to, amt);
3     case Cancel(var tx) -> cancelTransaction(tx);
4 }
```

Case Study 3: Enforcing Business Rules at Compile Time

Developers often check business rules at runtime, but Java security features enable them to enforce these rules at compile time through sealed hierarchies.

Consider role-based access control:

```
1 public sealed interface Role permits Admin, User, Guest {}
2 public record Admin() implements Role {}
3 public record User(String username) implements Role {}
4 public record Guest() implements Role {}
```

When checking access:

```
1 void authorize(Role role) {  
2     switch (role) {  
3         case Admin a -> allowAll();  
4         case User u -> allowLimited(u.username());  
5         case Guest g -> deny();  
6     }  
7 }
```

No other role can sneak in, closing a common source of privilege escalation.

Best Practices for Java Security Features

Writing secure Java code is not just about following rules – it’s about embedding security into the design from the start. By adopting best practices that leverage the new language features in Java 25, developers can prevent common vulnerabilities, make their code easier to audit, and ensure that business rules are enforced consistently.

Key practices include:

- **Model domains explicitly with sealed classes:** Don’t let critical states be represented by loosely typed strings or integers.
- **Prefer using records for data transfer:** Immutability helps close many potential attack windows.
- **Leverage pattern matching for exhaustive handling:** Never rely on fragile chains of **if** and **cast**.
- **Use `null` explicitly in patterns:** Guard against unchecked nulls at compile time.
- **Audit serialization boundaries:** Ensure external input maps safely into records.
- **Design for minimal mutability:** Even outside records, prefer final fields and unmodifiable collections.
- **Integrate validation at construction time:** Use record constructors to enforce input rules.
- **Close your hierarchies:** Treat sealed classes as security fences.

Conclusions

Java 25 marks a turning point where security is no longer treated as an afterthought layered on top of applications, but as an integral part of the language itself. By adopting sealed classes to enforce strict domain boundaries, records to guarantee immutability, and pattern matching to ensure exhaustive handling of inputs, developers can eliminate entire classes of vulnerabilities before they ever reach production. Security in 2025 is about prevention through design, and Java 25 provides the tools to build software that is not only functional and performant but also resilient, auditable, and inherently safer by construction.

REFERENCES

- <https://openjdk.org/projects/jdk/25/>
- <https://www.baeldung.com/java-25-features>
- <https://openjdk.org/jeps/427>
- <https://docs.oracle.com/javase/specs/jvms/se15/preview/specs/sealed-classes-jvms.html>
- <https://blogs.oracle.com/javamagazine/post/java-immutable-objects-strings-date-time-records>
- <https://openjdk.org/jeps/468>

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



Why I Abandoned my 30-year Open Source Project

OpenSource

SoftwareMaintenance

Python

Author: Peter Verhás

Peter Verhás is a senior software engineer and architect with an electrical engineering and economics background (MSC and MBA). He created his first programs in 1979 and has since authored several open-source programs. He has worked in several telecommunications and finance industry positions and was the CIO of the start-up index.hu during its early days. Peter is a developer and architect for EPAM Systems in Switzerland, supporting clients, developer training programs, and internal mentoring. Peter is the author of several Java books published by Packt and has a blog <https://javax0.github.io> You can follow Peter at <https://twitter.com/verhas>, <https://www.linkedin.com/in/peterverhas/>, <https://github.com/verhas>.



LinkedIn

Introduction

I started an open-source project in 1996, I am abandoning now. It was not my first OSS project and certainly not the last one. It definitely was the one that lasted the longest and that I had the most faith in having an impact on the industry.

The project aimed to create a new approach to document maintenance, and over the years, I developed a few versions of software solutions that supported the idea.

I am not abandoning the idea, though. I still believe that the philosophy behind how we maintain documents is valid. It proved itself when I wrote my technical books and several documents. The last version of the software tool I created supporting this document management approach is the one I partially abandon. Not the approach to document maintenance.

In this article I will tell you a bit of the

- history of these software projects,
- how it worked technically,
- how it did not work in the industry, and
- why it did not work, and why and how do I abandon this project.

Program Enhanced Document a.k.a. PET

What is PET

The idea of Program Enhanced Document was to do only one straightforward thing originally: include a code snippet in the documentation.

When I wrote the documentation of the ScriptBasic interpreter, I created sample programs. I wanted to have these sample codes in the documentation along with the output of the programs. I developed the interpreter, the documentation, and the sample code snippets in parallel. Sometimes, I changed the language, the implementation, and the interpreter, and it also required changes in some of the sample code. The change in the sample code also changed the output many times.

After making the code change, I copied the samples and output into the documentation. It was tedious and error-prone, and after a while, as a developer, I decided that I should not manually perform a task that could be automated with a program. (Usually, this is the point when the original project is abandoned to start a new one. Not in this case, but ScriptBasic, which is still in use in some industrial products, was developed further till 2006 approximately.)

I developed a small macro language in Perl, which I named Jamal as an acronym for Just Another Macro Language. It is approximately 300 lines of Perl code that could convert one text file to another, processing the embedded macros in the meantime. There was a macro that could include a file in the output, so whenever a sample code file or the output changed, the document automatically updated to reflect the changes.

It worked.

Over the years, I have developed numerous technical documents, including two books on Java 9 and Java 11, as well as several chapters in other books as a co-author. Through these works, I realized, step by step, that PET, which I did not call by that name at the time, is a general approach that most technical writers should follow. It is a general approach that enhances the documentation source, typically in a textual markup format with additional information that can be processed automatically. The documentation written for humans is redundant. It is not only formatted but also contains information, which is already documented in the system. Sometimes information is repeated because you do not want the reader to turn pages or navigate links to re-read a single sentence that may apply to several parts of the document.

The conventional method of managing information in documentation is to copy and paste it from the documented system or other parts of the document. This increases the redundancy in the document, which is the source of inconsistencies.

PET, as a document maintenance philosophy, says that anything that can be maintained automatically in a document must be automated.

If you document a specific version of a Maven project, for example, do not copy the version number into the documentation. If you do so, it will remain the old, outdated version long after you have modified and released the documentation of the new version, confusing the readers. Instead, use some means to reference the version in the `pom.xml`, which is fortunately a very machine-readable format. It will automatically update the documentation with the actual version using the proper tooling.

If you have a configuration string you can use to tune your application, reference the code where it is defined instead of copying it. Create unit tests that demonstrate the use of the API and include them automatically in the documentation as code snippets. Do not copy a text that applies to many parts of the document. Instead, use some macro facility and reference the macro by name wherever you need that text.

PET ensures that the document is consistent, with a *single source of truth*, and that all information that can be updated automatically remains up-to-date.

Jamal, Perl version

The very first version of a program aiming PET was Jamal, written in Perl. The original summary page from 1996 is still available at

[GitHub](#)

It was used to maintain the documentation of the ScriptBasic interpreter, but also to unify the `Makefile` for the build. ScriptBasic is written in C, and the build used `make`. Makefiles are known for their system dependence, and thus, compilation requires one for Linux and a separate one for Windows. Additionally, multiple C compilers were available for the Windows platform, some of which were free, while others required a fee, each with its own specific `Makefile`. I wanted to provide ScriptBasic in source, compilable for multiple platforms. Jamal helped create a single `Makefile.jam` that could be converted into multiple different makefiles for various operating systems and compilers.

I used this implementation over the years, and it also had a few other users. They used it to maintain web pages.

Perl still exists, although it is a legacy language, so maintaining Jamal in Perl didn't seem feasible. The latest version of the code is available from 2010:

[GitHub](#)

It has built-in macros, user-defined macros, conditionals, and loops. Essentially, it implemented a complex and powerful templating system.

Pyama

When I wrote the first version of my book, I was sure to use something like Jamal. By this time, however, the Perl version was outdated, and I had not touched the code for several years. Although I could still run it and use it to maintain the source code of the book, it had a specific shortcoming. The book primarily focused on source code samples and explaining their behavior. When editing the file using Jamal, all I saw in the editor was the reference to the file and the specific snippet name, not the code. I had to keep the code open in a different window in a different viewer. When reviewing the modifications as the samples developed, I had to navigate a lot to the different snippets and source files.

To overcome this, I developed Pyama using a reverse approach. The basic operation of Jamal is maintaining a source documentation text and converting it to another text. You edit one file using the editor, and there is a generated file you never touch. This separation is a good and safe practical approach, also followed by some code-generating tools.

However, it is not like the laws of physics, which even politicians can't break. It is a practical rule and not an absolute. After all, you cannot say you never modify the source code maintained "manually" by any program. The editor itself is also a program. Even `vi`.

Pyama sees itself as an offline "editor". You write the commands controlling the editor into the edited file, and Pyama edits the file for you. For example, it updates the code snippets from their actual location. Some editors, like IntelliJ, also recognize when an edited file changes on the disk and update the loaded one if there is no conflict.

This way, there is only one file that you edit in two different ways. The only question is how often you save your file to version control and how much you trust your offline editor not to corrupt the text.

Pyama is available at <https://github.com/verhas/pyama>

It was updated seven years ago.

Unfortunately, the publisher insisted that I edit the book in Microsoft Word, and that rendered the Pyama tool unusable for the first book. For the second edition, I could use their online WYSIWYG editor using HTML behind the scenes, and I could use some specific scripts and the Chrome debug menu to apply PET. That is, however, another story.

Jamal in Java

Learning from these Python scripts that converted Markdown to a special version of HTML, the WordPress-based online editor used, and from the project Pyama, I forked off two projects. One was Java::GECI, a code generation tool that updates your Java source files by inserting generated code into them, and the other was the Java version of Jamal.

It started small by implementing the exact macros that had been there for 15 years in the Perl version. The Java version was cleaner than the Perl, a language that is known for being a hacker's tool. I split up the project into several modules and started to implement special macros into their own module to keep them separate.

Currently, there are more than 200 macros. They handle code features, like **if**, **for**, defining user-defined macros, or setting and resetting the macro start and end separator strings. There is a separate module to fetch and format text from source files, as well as from JAR files, JSON, YAML, XML, SQL, and so on.

I also integrated it into Maven as a plugin as well as an extension, AsciiDoc preprocessor, and JavaDoc as a preprocessor, and into Microsoft Word. It also integrates with the Java version of my ScriptBasic interpreter, as well as Groovy, JRuby, and Python, allowing you to include code from those languages within your document. It even includes a small BASIC interpreter that natively integrates with the macro language, allowing macros defined for text processing to be called programmatically.

(BTW: do not use the macro extension and do not create your `pom.xml` files with Jamal macros. It is not a good idea. I have learned over the years. These days, I try a different approach to avoid the XML configuration hell.)

It can generate extensive trace output in XML, showing the conversion process from input text to output text in each step. If that is not enough, you can start it in debug mode and use the React.js debugger front-end to execute the macro processing step-by-step.

Jamal Problems

Although Jamal is an enterprise-grade tool, I am not aware of anyone using it. On GitHub, it has 63 stars, which is one-tenth of the License3j project's 621 stars (as of 2025-07-21). I cannot know the reason; I can only guess.

I guess that the reason is that it is an enterprise-grade tool. It has numerous features, and thus it has a significant threshold to overcome before it can be used. Nobody wants to use all 200 macros, and finding the one or a few you need may be a significant effort. I have to look up the documentation for some, though I implemented them.

If I look at License3j, the most popular project in my repository (nothing to brag about because 621 stars are still close to nothing), it is a simple library that does one straightforward thing: read and write a proprietary license file format and use public-key cryptography to verify it has not been tampered with. There were numerous requests and bug reports over the decades.

Thus, I guess, Jamal is too complex to start with, even if you want to do program-enhanced text. But it is not only Jamal. There is a problem with the whole PET approach, which I will not solve in this article.

PET Problems

Program Enhanced Text is more complex than just writing Markdown or AsciiDoc. A text document may be incorrect, outdated, even unreadable, or contain inappropriate language, but it never causes a build failure. Developers already have many things that can break a build; they do not need another one. They are measured by working code, the number of test failures, and production incidents, rather than documentation accuracy.

License3j offers a feature that some people have needed for decades. ([SourceBuddy](#), [pyama](#)) provides a simple API for compiling Java code at runtime. Java::GECI creates code for programmatically definable code generators.

These are projects that are more commonly used (more stars) than Jamal. They all address some features that some developers need.

Developers do not need Program Enhanced Text. Enterprises do, but enterprises do not select technology. Developers working for the enterprises do.

In addition to the fact that writing PET documentation is more complex, there is another factor. Nobody reads the documentation. And this may be a good thing for PET, I discuss in the next section.

PET and LLM

A few months ago, I started to create an object-oriented functional programming language, which I named Turicum. ([turicum](#)) You can see I have a soft spot for scripting languages since I could get my hands on a copy of the “Dragon Book” in 1987 at TU Delft. I created a BASIC compiler in z80 on the ZX Sinclair Spectrum (does not exist anymore, source code was lost in the noise of some audio cassette). I created a programmable assembler (PCMAC). I created ScriptBasic (twice, in C and in Java), Jamal (twice, in Perl and Java).

I never expected Turicum to be widely used, and it may never be. I created it to experiment with some features that are not available in any other languages, to see how usable they are. (I promise, I will not derail there. I may write a separate article about closure reclosing, and uncurrying curried functions.)

Being at the age that can be inferred from the date I mentioned above, I tend to forget things. That is where documentation comes into the picture. I created the documentation of the language, and from time to time, I had to look up what I wrote about some of the language features I created just a month before. Sometimes it was a lot of time. I could not say I did not read the documentation, because I wrote it. Still, it sometimes took a few minutes to find the right chapter.

This is the kind of work that LLMs can do for us these days. I exported the Users’ Guide in PDF and added it to the context for ChatGPT. Then I started to ask questions and tried to experiment: what questions could it answer, and what answers it could not.

The factual text and explanations were generally satisfactory. Code fragments were not so much. It hallucinated Python code into the language, though Turicum is far from being a Python clone. So I asked ChatGPT itself what I should change in the documentation so that it does not hallucinate the syntax that much.

Interestingly, his/her advice worked. It was suggested to add a BNF to the documentation, so I did. Next, I asked ChatGPT to solve the eight-queen problem in Turicum, and it did.

With the advent of LLM solutions, it is no longer true that nobody reads the documentation. It is the LLM that reads the documentation, and the users can ask questions. There may still be a need for some tutorials and an introduction, but that is more like a video format.

However, if your reader is an LLM, then you have to structure and format the document for the LLM. It means that there is less need for explanatory and glue text. All you need is a precise reference style description, like an EBNF with some comments in the case of a programming language. You can maintain it in a separate document, but it will be kept more up-to-date if it is part of your source code.

You can place the reference documentation into program comments and automatically extract it from the code.

This is an existing practice that I have used many times, utilizing Jamal’s snippet functionality. When a function or functionality has several parameters, I create documentation for the parameters in Asciidoctor format within the source code comments. Jamal reads the source code, extracts these snippets, formats them with pattern matching and regular expression search and replace operations, and inserts them into the text. It is more likely to update the EBNF description of a command whenever the syntax of the command changes during development if the EBNF is part of the source code. If it is in a separate document, then it is easy to forget to update. Out of sight, out of mind.

Because there is less need for explanatory text and more need for reference text to be fed into the LLM, the importance of such documentation fragments increases.

Recommended Approach

What should you do? What do I recommend if you want to enjoy the advantages of PET but do not want to start with a heavy weapon like Jamal?

I was playing around with the idea of what I would do if I were not allowed to use Jamal, and still want one feature implemented in PET. This feature is to hierarchically number chapters, sections, and subsections in a Markdown document. It is a standard feature in Jamal, but I needed a document to pass to someone who I wanted to go on with the document maintenance, and was not likely to use Jamal.

I came up with the idea to maintain the structure of the document to resemble a JSP or ASP, separating program logic and text, while dropping macros. Instead of a newly implemented macro language, I decided to use Python. Well-known, good enough for the purpose. I vibe-coded with Claude the template handling with a simple prompt, and the result is the 100-line Python code in the repository:

[verhas/pet](https://github.com/verhas/pet)

It mixes text with Python code between `{%` and `%}` start and end strings. The Python code can load classes and functions defined in the directory `.pet` that you can supply with your document. Hierarchical counters are very simple 80 lines. If you need another feature from the 200 different macros that Jamal provides, you can write some scripts in a few minutes or hours, tops. You do not need to install any software, other than Python, that you probably already have on your machine.

Technical details on how to use this library are described in

[README2.md](#)

However, it is essential that I do not explicitly advocate the use of this library or any other alternative to Jamal. Use Jamal, if that fits you. It is there; if you report a bug, I probably will fix it. Use this library, copying it, modifying it to your needs. Use something else utilizing Python, Groovy, Ruby, or Rust to your taste.

What I advocate is using Program Enhanced Text (PET) documentation for any technology.

Summary

In this article, I described Program Enhanced Text documentation and how I learned the principles over the last 30 years. There is nothing revolutionary in it. All parts of it have already been used by someone somewhere.

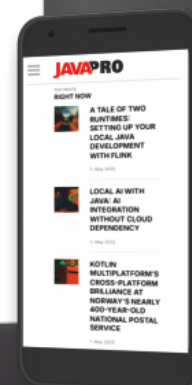
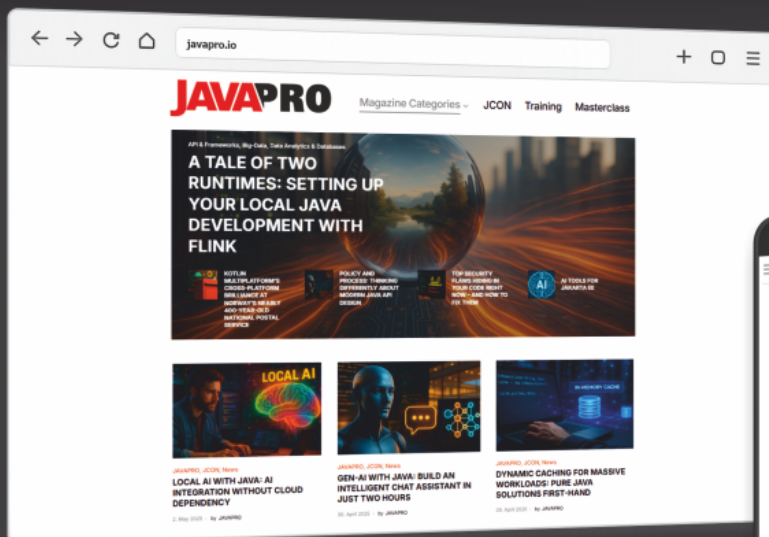
I described a tool I maintained in the last ten years to support PET, and also what I think you should follow, and why PET is essential with the dawn of LLM.

Go out and write PET documentation.

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io

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