

27 **JAVA 24: A STORY OF
CODE, CONFLICT & CONQUER**70 **EVALUATING
RAG PIPELINES WITH THE RAG TRIAD**

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

Edition 6 / 2025

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY #JAVAPRO

Modern Software Systems on the JVM

42 **NAVIGATING THE WAVES OF CONCURRENCY:
EXPLORING JAKARTA CONCURRENCY**47 **THE ROLE OF QUARKUS
IN THE MODERN JAVA ECOSYSTEM**58 **THE SIMPLEST WAY
TO BUILD RESILIENT APPLICATIONS**65 **BUILD LOCAL LLM:
OLLAMA, QUARKUS & TESTCONTAINERS**74 **BUILDING AI-DRIVEN APPLICATIONS
WITH SPRING AI**171 **FROM BREACHES TO BLACKOUTS: THE HUMAN
CONSEQUENCES OF SUPPLY CHAIN ATTACKS**176 **MASTERING API SECURITY IN JAVA:
OWASP BEST PRACTICES**189 **THE RISE OF JVM LANGUAGES:
KOTLIN, SCALA, GROOVY & MORE****JCON**
EUROPE
COLOGNE
2026April 20 - 23, 2026
www.jcon.one



SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

From Java to the JVM: Building Systems That Last

Modern software is no longer built as isolated applications. It lives within complex systems that must scale, evolve, remain secure, and deliver value over long periods of time. This issue of JAVAPRO focuses on exactly that reality: how to engineer modern software systems on the JVM across the full lifecycle: from design and development to operation, security, and organizational impact.

Java at 30: A Platform That Keeps Evolving

This year, Java celebrates its 30th anniversary. This milestone is not about nostalgia, but about relevance. Together with the JVM, Java has continuously adapted to new demands, evolving from a language for monolithic applications into the backbone of highly concurrent, distributed, and resilient systems. Advances such as Virtual Threads, Structured Concurrency, and modern Jakarta standards demonstrate how the platform simplifies scalability and concurrency while preserving the stability enterprises rely on.

The JVM as a Multi-Language Ecosystem

Java is no longer the JVM's only expression. Today's JVM is a mature, multi-language platform supporting a wide range of paradigms. Kotlin, Scala, Groovy, and other JVM languages expand expressiveness and productivity while remaining interoperable with existing Java systems. Real-world stories such as the adoption of Kotlin Multiplatform to consolidate business logic in large-scale logistics show how JVM technologies help organizations reduce duplication, manage complexity, and establish a single source of truth across platforms.

Architecture and Scalability as System Concerns

Modern systems are shaped by architectural choices that go far beyond individual components. Concurrency models, micro-services, and micro frontends influence build pipelines, deployment strategies, runtime behavior, and team collaboration. Several articles in this issue illustrate that scalability is not only a technical challenge, but also an organizational one, requiring thoughtful trade-offs and architectures that can evolve over time.

Productivity, Debugging, and Quality by Design

Sustainable systems depend on developer productivity and built-in quality. Standardized project setups, build automation, and advanced debugging techniques are not optional optimizations - they are foundations for long-term maintainability. A data-driven approach to testing and quality assurance reinforces a key insight: quality is not achieved by running more tests, but by running the right tests at the right time, evaluated by their reliability, usefulness, and feedback value.

Security and Responsibility in Modern Systems

Security is a defining characteristic of modern software systems. APIs expose critical functionality and must be protected accordingly. Beyond that, software supply chain attacks highlight a broader responsibility: most systems consist largely of third-party components, and vulnerabilities can have consequences far beyond financial loss. Disruptions to healthcare, infrastructure, or critical services show that software engineering increasingly has real-world, human impact.

Beyond Frameworks and Rituals

Technology alone does not create successful systems. Methodologies such as Agile, Scrum, Kanban, or linear approaches are great tools - not guarantees. Sustainable value emerges where technical discipline, organizational context, leadership, and a shared understanding of outcomes come together. Modern software systems are built not only with better tools, but with better judgment.

Building for the Long Term on the JVM

This issue presents the JVM as what it is today: a robust, adaptable, and future-ready platform for building modern software systems. By combining innovation with stability, the JVM enables teams to tackle complexity responsibly and with confidence.

We hope this issue provides valuable insights and inspiration for your work on modern software systems on the JVM.

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.

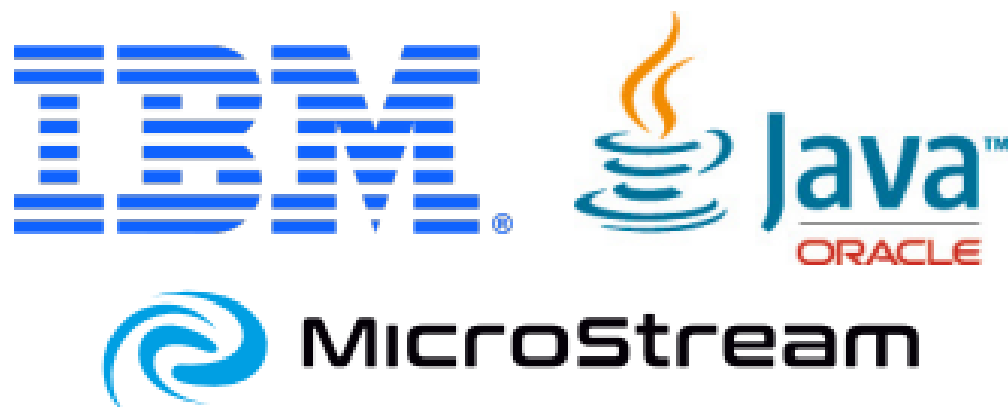


LinkedIn

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors



Inhaltsverzeichnis

Core Java

Marián Varga

Java Has Been Connecting Systems and Communities for 30 Years

11

Core Java

Richard Gross

Fixing the Billion-Dollar Mistake	15
Null References: A Historic Mistake with Dramatic Consequences	16
The Actual Cost	16
The Error Is Not Where You Found It	17
The Trend Towards Safety in Programming Languages	17
Everyone Tries to Deal with Null	18
Python Tries to Deal with Null	18
C# Tries to Deal with Null	18
Java Tries to Deal with Null	19
Java Has OptionalReturn	19
Add Compile-Time Null Checks with Nullaway and JSpecify	20
Fixing the Billion-Dollar Problem with the Compiler	22
Gradual Migration	22
6+1 Tipps to Get the Most Out of Null-Checking	23
The Future – Six Ph.D. Theses, Knotted Together	26
References	26

Core Java

Lutske de Leeuw

Java 24: A Story of Code, Conflict, and Conquer	27
What Are Object Headers?	28
How Does Class Loading Work?	30
The Rust Temptation	31
JEP 491: Synchronize Virtual Threads without Pinning	31
Empowering a Team of Java Explorers	33
The Power of Shared Knowledge	33

Core Java

Merlin Bögershausen

More Action, More Overview	34
Threads in General	35
Java Threads API	36
JEP 444 Virtual Threads	37
Structured Concurrency	38
Scoped Values	39
Final Thoughts	40

Serverside Java

Gautham Krishnan

Navigating the Waves of Concurrency: Exploring Jakarta Concurrency	42
Understanding Concurrency	43
Enter Jakarta Concurrency	43
Key Features and Components	43
Managed Executors	43
Context Propagation	44
Managed Threads	45
Embracing Best Practices	46
Conclusion	46

API and Frameworks

Martin Stefanko

The Role of Quarkus in the Modern Java Ecosystem	47
Quarkus to the Moon and Beyond	48
What is Quarkus?	49
Developer-Focused Framework	51
Innovation	54
Community Engagement	55
Summary	57

API and Frameworks

Giselle van Dongen

The Simplest Way to Build Resilient Applications	58
Why Resiliency Matters	59
Resilient Code Execution and Communication	59
Implementing Sagas and Distributed Transactions	61
Stateful Entities and State Machines	61
What Can You Build?	62
Detailed Observability	62
Try It Out!	63

AI and ML

Jonathan Vila López

Building Local LLM AI-Powered Applications with Quarkus, Ollama, and Testcontainers	65
Project Overview	66
1. Integrating Quarkus with Ollama	66
2. Using Testcontainers for Integration Testing	68
3. Leveraging Quarkus Dev Services for Ollama	69
Conclusion	69

AI and ML

Atamel Mete

Evaluating RAG Pipelines with the RAG Triad	70
Typical RAG Pipeline	71
Approach 1: Evaluating Ingestion+Retrieval and Generation Separately	71
Approach 2: RAG Triad	72
Evaluation Frameworks	73
Conclusion	73

AI and ML

Timo Salm

Building AI-Driven Applications With Spring AI	74
Getting Started	75
The ChatClient API	75
Structured Output	76
Advanced AI Patterns	77
Image and Audio AI Models	79
Summary	79

DevOps

Sergei Chernov

How to Speed up Maven Builds	80
Checking the JDK	81
Kotlin	82
Build Caches and Build Scans	82
Develocity (Formerly Gradle Enterprise)	83

Parallel Builds	84
Takari Timeline Extension	85
How Parallelism Works in Maven?	86
Removing Unnecessary Dependencies	87
Splitting Large Modules (Independent Parts)	88
Splitting Large Modules (Dependent Parts)	89
Moving Code Generation to Libraries	89
Stop Publishing Unnecessary Artifacts	90
Persistent CI/CD Build Agents	90
Use Maven Profiles	90
Why Not Just Switch to Gradle?	90
Conclusion	90

IDE and Tools Thomas Froment

A New Generation's Chance to Shape the Future: Building the Next Great Java IDE in Open Source	91
A New Generation's Chance to Shape the Future: Building the Next Great Java IDE in Open Source	92
The Eclipse IDE and the Rich Client Platform: Two Sides of the Same Coin	92
The Birth of Initiative 31: A Bold Vision for Modernisation	92
The 2025 Reality: Focus on Skia	93
Revamping the Core: SWT Modernisation Roadmap	93
The March 2025 Eclipse IDE Release: Performance Improvements and Community Contributions	94
A New Chapter for Eclipse IDE	95
A Call to Action: How Java Developers Can Contribute	96

IDE and Tools Manoj Nalledathu Palat

Breakpoints – The Lifeline of Debugging – An Eclipse Features Guide	98
--	-----------

IDE and Tools Vipin Menon

Observability Landscape in Java	108
First Generation: Standard Debugging	109
Second Generation: Focused on Production	110
Third Generation: Cloud Native Observability	112
The Future of Observability in Java	114
Conclusion	115

IDE and Tools Andres Sacco

Code Wizardry: Casting Spells to Fix Bugs	116
Impact of Bugs and Bad Practices	117
Common Consequences	117
Highlight the Causes	119
Static Code Analysis Tools	120
Automating Fixes	121
What's Next?	127
Conclusion	127

IDE and Tools Mohamed Ashraf Bayor

Japo: The Ultimate Search Engine for Java Annotations	128
Challenges of Java Annotations	129
Introducing Japo: Java Annotations Repo	130
Key Features of Japo	131
How Japo Helps Java Developers	133

Looking Ahead: Japo's Vision	134
--	-----

IDE and Tools

Giovanni van der Schelde

Unleashing the Power of Maven Archetypes	135
Introduction	136
How Maven Archetypes Work	136
Creating an Archetype	137
Property Interpolation with Velocity	139
Keeping Archetypes Up to Date	140
Final Thoughts and My Experience with Archetypes	141

IDE and Tools

Manoj Nalledathu Palat

Launching a Java Debugger in Eclipse - A Myriad of Options	142
Java Application	144
Eclipse Application	147
Junit	148
Junit Plug-in Test	148
Prototypes:	149
Debug Favourites	150
Launch Groups	151
External Launch Configuration:	153
Exporting and Importing a Launch Configuration	153

Project Management

Marin Niehues

Agile, Scrum, Kanban and the Lies We Tell Ourselves About Creating Value	154
The Underlying Tension: Why "Agile vs. Linear" Is a False Dichotomy	155
A Growth Mindset as the Baseline for Success	155
The Roots of "Agile" and Its Limitations	156
The Validity of Linear Methods	156
The Common Dysfunctions That Undermine Both Approaches	157
The Real Key: Value Creation Over Ritual	158
The Stacey Matrix: A Key to Choosing Your Method	158
Where Scrum, Kanban, XP, and Linear Methods Shine	159
Cultivating a Growth Mindset Across the Organization	160
Common Dysfunctional Patterns to Avoid	160
Busting the Myths	161
Strategies for Achieving Real, Sustainable Value	161
Measuring True Value in Any Approach	162
Conclusion: A Call for Contextual, Value-Driven Approaches	162

Web Development

Christian Siebmans

Integrating Micro Frontends into Existing Applications – Lessons Learned	163
Two Teams and Lots of Reusability	164
Problems Arise	164
How We Get Types for Backend Communication Automatically	164
A Quick and Very Dirty Fix	164
What Is Module Federation?	165
Building Plugins for Our Application	166
Implications for Our Code Base	166
Module Federation and Web Components – a Match Made in Heaven	166
Sharing Dependencies Across Federated Modules	167

Importing Federated Modules	167
Unforeseen Difficulties	168
How We Define and Load Remotes Dynamically	168
Our Application Is Not Ready for Federated Modules	169
Outlook	169

Security

Steve Poole

From Breaches to Blackouts: The Human Consequences of Software Supply Chain Attacks	171
Setting the Scene – 9 Trillion Dollars	172
Consequences and Risks	173
Moving Forward	174

Security

Muaath Bin Ali

Mastering API Security in Java: OWASP Best Practices	176
Why API Security Matters: The Real Cost of Breache	177
Broken Object Level Authorization	177
Broken Authentication	178
Broken Object Property Level Authorization	178
Unrestricted Resource Consumption	179
Broken Function Level Authorization	180
Unrestricted Access to Sensitive Business Flows	180
Server Side Request Forgery (SSRF)	181
Security Misconfiguration	182
Improper Inventory Management	183
Unsafe Consumption of APIs	183
Conclusion:	183

JVM Languages

Anshika Koul

Kotlin Multiplatform's Cross-Platform at Norway's Nearly 400-Year-Old National Postal Service	184
Problem Statement – Complex World of Logistics	185
What Is Kotlin Multiplatform?	186
How KMP Streamlined Our Business Logic Across Platforms	186
So, Have We Finally Demystified the Chaos?	187
Wins, Woes, and Wisdom	187

JVM Language

Mihaela Gheorghe-Roman

The Rise of JVM Languages: Kotlin, Scala, Groovy, and More	189
Java and the JVM Languages: A Brief History	190
Kotlin: A Pragmatic JVM Language Alternative to Java	190
Scala: Bridging Object-Oriented and Functional Programming in JVM Languages	190
Groovy: A Dynamic JVM Language Enhancing Java	190
Clojure: A Functional Lisp for the JVM	191
JRuby & Jython: Bringing Ruby and Python to the JVM	191
Other JVM Languages	191
Why the JVM Remains Relevant	191
Conclusion	192

Testing and Quality

Steve Poole

A 10X Approach to Quality Assurance	193
Bugs Cost	194

When's the Best Time to Find Bugs?	194
How Should We Really Test?	194
Changing the Status Quo	195
Three Challenges to Overcome	195
We Have Almost All the Pieces	195
Testing Is Multi-Dimensional	195
Tortoise and the Hare vs the Feather and the Boulder	196
Bringing It Together	196
Risk / Reward Rebalanced	197
Changing Behavior	197
More Thoughts	197
Summary	198



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.



Java Has Been Connecting Systems and Communities for 30 Years

Community # JavaEE # Spring

Author: Marián Varga

Marián Varga is a co-author of the upcoming book Mastering RESTful web services with Java. He is building a community focusing on integration at love2integrate.com



LinkedIn

From fulfilling the slogan The Network is the Computer, to the Java EE standard, to supporting both SOAP and RESTful Web Services, to creating a strong ecosystem and communities supporting microservices and cloud-native applications that can't do without an emphasis on APIs. Intertwined with the Internet since its inception, Java supports existing as well as brings in its own vendor-neutral standards and de facto standards reflecting current challenges, such as MicroProfile or supporting integration with AI.

Java turns 30 this year and no one can count how many applications have been created with Java over the three decades. But those applications rarely exist in isolation!

I always try to remind people working on development projects about the often overlooked importance of remote APIs and integration aspects that are vital to connect pieces of software together. I also often emphasise that effective communication and collaboration among project team members are more important than solving most technical challenges. In this article, I want to show the great role that Java has been playing in connecting software systems and integrating different groups of people in the software industry.

Java was created at Sun Microsystems whose slogan was “The Network is the Computer”. At the time Java was born, people started to realise that not just any network would do. The Internet was what would shape the future of computing.

Already the first release of Java offered developers easy built-in support for many **features crucial in the Internet era**: the HTTP protocol, texts in any language encoded in Unicode, efficient concurrency thanks to threads, rendering web graphics formats like JPEG, GIF or PNG etc.

At first, Java aimed at embedded computing. The platform-independent JVM and Java API were defined so that we can write Java code once and run it anywhere (on any device). However, what most early adopters of Java actually got to play with were applets: programs running inside a web browser, using the Internet as a distribution channel to download the bytecode that could run on any physical CPU architecture and any operating system. In the context of JavaScript of that time being incompatible across browsers or not available at all, Java applets pioneered the interactive Internet experience we take for granted today.

The heart of the Web is HTTP, a client-server protocol. It soon became apparent that a very good place for Java is the server side. The second major version of Java came with the **Java (2), Enterprise Edition** specification. Java EE allowed Java to excel on the server side by abstracting the complexity of a server handling many client requests to the Java EE container. “Plugging in” to the Java EE APIs, for example the Servlet API, the application developers could focus on implementing their business logic.

Java is not the only platform supporting the concept of an application server. What makes it unique is the number of diverse commercial vendors and open source projects that have implemented the Java EE specification. The **vendor neutrality** continues with the Jakarta EE standard managed by the Eclipse Foundation. Developers familiar with the standard can switch from one container implementation to another one. While enabling competition and innovation, the standard keeps the Java community connected.

However, vendor-neutral standards were not always in time to provide developers what they needed to build and integrate their applications effectively. Early versions of Java EE were infamous for not being nice to use and requiring a lot of boilerplate code. **Spring Framework** addressed the issue by providing its own abstractions on top of Java standard APIs, thereby winning the hearts of lots of enterprise Java developers. But even when you use Spring for your server-side application, you always also depend on Java EE. Many of the abstractions provided by Spring inspired similar features in newer Java/Jakarta EE specifications.

Besides integrating remote APIs, our applications need to be integrated internally, connecting multiple modules and layers. Spring Framework brought a dependency injection framework to avoid coupling between the modules. Now we have dependency injection in Java EE with the CDI standard.

In addition to synchronous communication using the ubiquitous HTTP and easily available thanks to the **Servlet API**, Java EE also recognised the importance of asynchronous communication via message brokers by defining the **JMS standard**. By removing the direct links between the communicating parties, the message-oriented integration decreases coupling and improves flexibility and scalability of systems. Thanks to the standard API, the same Java code can work with different messaging server implementations.

We are also experiencing an increase in popularity of non-JMS asynchronous communication platforms like Kafka or various cloud-based messaging services. With Java you can use their respective native clients directly, but there are also options that let you **abstract the details of the messaging platform** and use them as just “dumb pipes” in solutions based on Enterprise Integration Patterns. Apache Camel offers a mature Domain-Specific Language and Spring Cloud Stream a modern functional programming style.

The success of the Web and its support by different devices, operating systems and applications brought the idea that the infrastructure of the Web could be used for more than just HTML pages. The term **Web Services** was adopted for the use of the web protocols and infrastructure to connect applications in general. The first Web Services standard (also colloquially known as “SOAP Web Services”), created by the World Wide Web Consortium (W3C), enabled unprecedented interoperability of APIs implemented by diverse technology stacks. In particular, it made integrating Java-based applications with Microsoft-specific implementations much easier.

Due to the diversity of the Java community, multiple libraries and frameworks for implementing the W3C Web Services standard started to appear: Apache Axis, CXF or Metro. A declarative, annotation-based approach to Web services was standardised later by JAX-WS.

The W3C Web Services turned out to be more complex than necessary, duplicating features already available in HTTP itself. Many developers and architects started to prefer **RESTful Web Services** and exchanged XML for the simpler JSON as the format for their data.

REST is a very popular API style and there are different Java frameworks that you can implement it with: various implementations of JAX-RS (now Jakarta RESTful Web Services) or Spring Web. All of them use annotations to specify the API endpoint paths, methods, parameters etc. declaratively and automatically serialize/deserialize JSON payloads to/from Java models.

Java frameworks make providing and consuming web services simpler and faster, however if an API is to be supported for a longer time and/or for many clients, it pays off to maintain the API specification as a document that is clearly separated from the implementation, using a widely accepted specification language. The W3C (SOAP) Web Services standard includes the WSDL format to specify the APIs. For RESTful Web Services the most commonly used specification language evolved from Swagger to **OpenAPI**.

The specification-first approach to building APIs means starting with a high-quality OpenAPI document. The **OpenAPI Generator** is a mature open-source project written in Java that can generate Java interfaces and models from OpenAPI. Using Maven and Gradle plugins it can be integrated in the build, so that the Java code is always in sync with the API specification. The generator can also generate code in many other languages, so using OpenAPI is a great way to integrate with non-Java systems.

If you prefer the code-first approach, all common Java application frameworks support generating OpenAPI documents and a **Swagger web-based UI** from the annotated Java code.

No matter if your APIs are specification-first or code-first, for multiple reasons it is recommended to keep them in a layer separate from the core domain logic of your application.

This also means that each of the layers should use its own objects for holding application data. Annotation processing is here to the rescue again, we can map data transferred between the layers declaratively using **MapStruct**.

We use a special type of Java objects, the so-called Data Transfer Objects (DTOs) to model the API payloads as well as the data passed to the domain layers of the application. They are not typical objects as defined by the Object-Oriented Programming, they usually do not contain methods other than those needed to build the object (including validation) and read their attributes. Working with this type of objects aligns well with the increasing popularity of Data-Oriented Programming (a subset of Functional Programming). To make the data objects safe for streams and concurrent processing, it is recommended to make them immutable. Java reacted to this trend by providing the **record** type that is immutable and reduces the boilerplate code needed to define a DTO.

As the volume of traffic and scalability requirements for many APIs grew, the classical thread-per-request programming model supported by the Servlet API (and other server-side APIs) started to be perceived as a bottleneck, mostly because integration-related code spends a significant proportion of its processing time waiting for I/O. At first many Java projects decided to tackle the problem by switching to the reactive programming paradigm, building applications that process data using **reactive streams** implemented by RxJava or Project Reactor. The different reactive streams implementations share the same standard interfaces that became part of the Java language.

However, for most applications the additional complexity of reactive programming was not justified by the mere need to avoid blocking threads. **Virtual threads** coming from Project Loom can address the problem in a more elegant way. Applications can continue using the thread-per-request coding model (that is simple to read and debug), while Java standard libraries and the JVM take care of reusing the limited number of platform threads (threads available at the operating system level) so that using a million (virtual) threads, many of which are blocked, is no longer a problem.

Here we can see the pattern that a common problem occurring in the software industry first triggers solutions based on new libraries and frameworks that are viable and adopted quickly thanks to the flexibility and type-safety of the Java language.

Later the multiple solutions are standardised to make the implementations interoperable. Even later the implementations can be simplified by new features coming to the Java language and JVM. Ultimately the whole Java community can reuse the common problem solution in different contexts easily.

Scalability, frequent deployments and reducing dependencies among development teams are motivation factors behind the growing attractiveness of distributed applications based on the Service-Oriented Architecture, later renamed to Microservices. It is no surprise that good integration of the services using good API design and implementation are very important in such systems. Deployment and operation of the many separate services shifted from application servers (JEE containers) to Docker containers and **cloud deployments**.

Stand-alone services based on an **executable JAR** instead of being deployed to a separate JEE container were introduced by Spring Boot. Later, to achieve radical improvements in start-up time and computing resource efficiency it was necessary to change as much as possible of the process of finding application components, their configuration and interconnection, **from dynamic processing during the application start-up to evaluation at build time**. That idea led to the creation of frameworks like Micronaut and Quarkus.

The new frameworks tried to make it easy for developers to switch to them by reusing concepts from either the existing de facto Spring standard (Micronaut using controllers) or from Java/Jakarta EE. Not everything from Jakarta EE was practical to support in the microservices frameworks and on the other hand there were new aspects of the services that can benefit from standardisation to avoid fragmenting the Java community. Also maintained under the umbrella of the Eclipse Foundation we now have the **MicroProfile** specification, the standard that shares the Core profile with Jakarta EE and is implemented by Quarkus, Helidon and others.

MicroProfile contains many parts that are relevant to integration:

- Jakarta RESTful Web Services, JSON Binding and JSON Processing (already mentioned above)
- Jakarta OpenAPI to generate OpenAPI from Java code
- Jakarta REST Client to consume RESTful web services
- Jakarta Fault Tolerance with retries, circuit breakers etc.
- Jakarta JWT Authentication to handle security aspects by specialised OAuth servers
- Jakarta Telemetry to monitor our services and APIs well
- Jakarta Config and Health so that the services can be configured and orchestrated in real environments

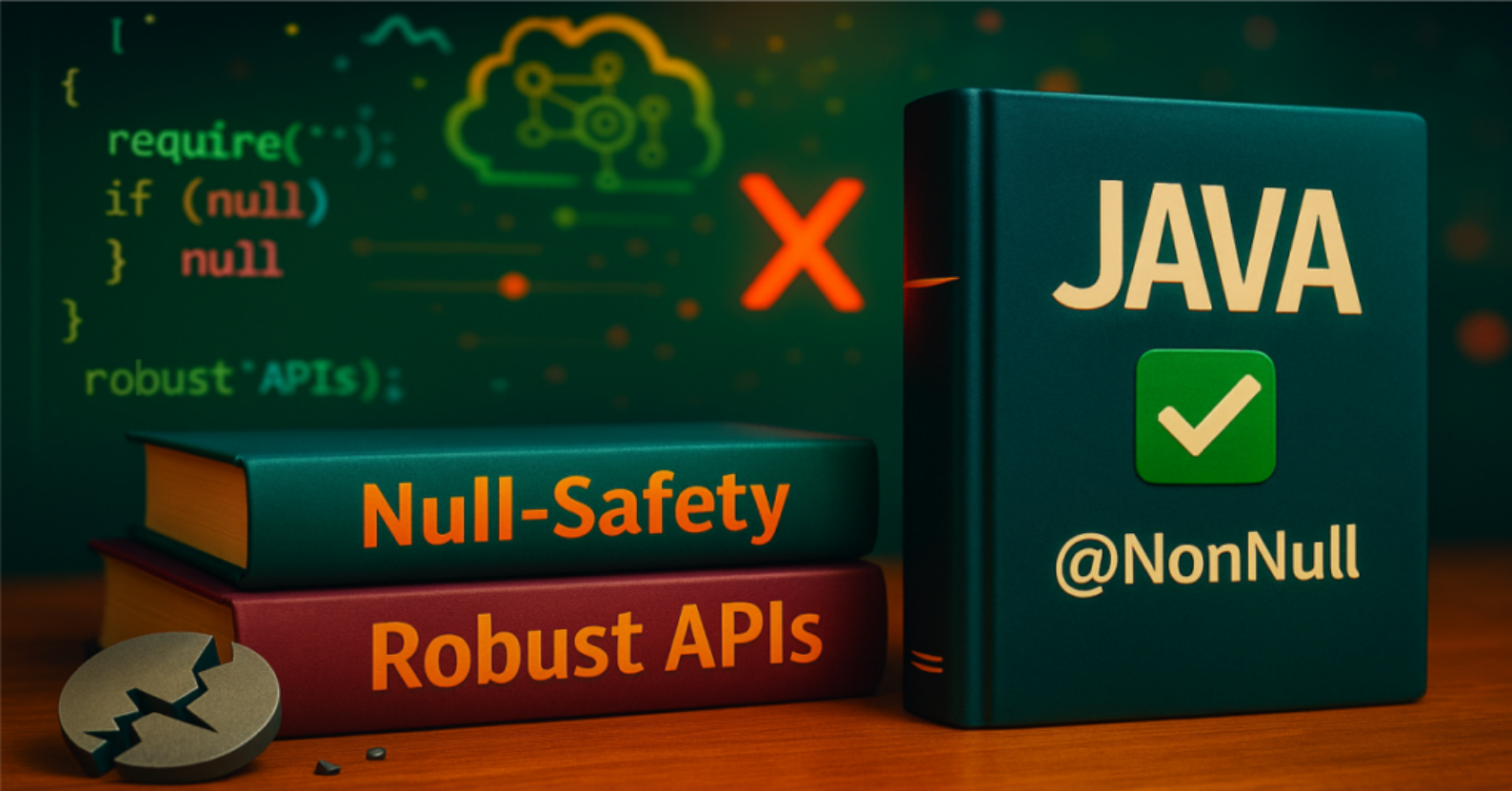
When talking about integration, we should not forget testing. Of course, with Java we have the great JUnit framework to automate our tests. But the layers of our applications handling integrations and remote APIs have their specifics. The critical parts often don't take the form of lines of our Java code whose test coverage is measurable as a percentage. Integration bugs are usually hidden in configurations, authentication, de-/serialisation etc. So it is not enough to focus on just a couple of classes, simulating the rest by mocks.

We need to involve the infrastructure provided by the application framework to run meaningful **integration tests**. Most frameworks include their own integration testing support, unfortunately this area is not unified across the frameworks by a strong standard like Java/Jakarta EE or MicroProfile. But you can run your application including any framework as a black box and let the integration tests communicate with it using the real APIs over the real network protocols. In Java you can make this kind of tests using the nice DSL provided by the **RestAssured** library. To simulate APIs out of the test there is another great Java-based tool, **WireMock**.

The hot topic of today is the business usage of AI and especially LLMs. The landscape in this area is changing very rapidly and clearly both the software structure and hardware requirements of AI systems are very different from traditional enterprise applications. For these reasons it is likely that for many businesses it makes sense to **integrate existing systems with LLM or RAG services** via remote APIs. Although there are many fast-evolving and competing AI products, Java already offers adapters to abstract away the differences in their APIs, let's just mention Langchain4j and Spring AI.

I believe that one of the factors that has allowed Java to be a successful language and platform for 30 years is its ability to connect both software systems and developer communities. Projects that relied on Java were able to benefit from efficiencies in both the development process and the execution of delivered applications. Developers can expect Java to stay relevant in the future.

However, good API designs and integration solutions are not automatic even when using Java. Moreover, mistakes are not easy to fix once APIs are in use. API development skills increase with focused training and experience. Applying these skills requires that the company and the project recognise the importance of integration and give it appropriate attention.



Fixing the Billion-Dollar Mistake

NullSafety

NullPointerException

TechnicalDebt

Author: Richard Gross

Richard Gross is the head of software archeology at MaibornWolff. He drives forward topics such as software modernisation and technical excellence. As a team lead and architect, he helps teams to achieve a return on investment as early and continuously as possible. To achieve this, he coaches teams in code quality, test automation and collective ownership. His technical focus is on hexagonal architectures, TestDSLs, hypermedia APIs and the expressive and unambiguous modelling of the domain as code. Richard shares his knowledge through active pair programming, as well as training, technical articles and presentations at international conferences. He has also long been involved in the open source project CodeCharta, which enables non-developers to gain an understanding of the quality of their software.



LinkedIn

There are mistakes that are just costly. Then there is the null reference. A language feature that was added in 1965 “simply because it was so easy to implement”. Since then, null references have caused countless bugs, system crashes and security problems – and cost billions of dollars in maintenance and troubleshooting. It has been eliminated in many new programming languages and this article will show the practical ways to eliminate this language mistake even in Java.

Null References: A Historic Mistake with Dramatic Consequences

Sir Tony Hoare is best known for developing quicksort, the Hoare logic (a set of logical rules for reasoning rigorously about the correctness of computer programs) and formulated the dining philosophers problem along with Edsger Dijkstra. I apologise if my summary of his immense impact on software engineering is too truncated. He is also a Turing Award winner. He received the award for his “fundamental contributions to the definition and design of programming languages”.

One of the programming languages he worked on in 1965 was ALGOL W and in particular its comprehensive type system for references in an object-oriented language [Wikipedia](#). This language did check every reference at compile-time to make sure every pointer points to something that exists and is of the appropriate type. This eliminated a whole set of errors that cannot occur.

Case closed, null pointer completely avoided. If not for the fact that Tony worked for a company that sold ALGOL computers and most potential customers did not have ALGOL but Fortran programs. To run these programs on the ALGOL machines, a transpiler was written from Fortran to ALGOL. After about a year of work the transpiler was done but no customer wanted to use it. The transpiled Fortran programs all produced subscript errors caused by pointers that pointed to null and/or had the wrong type. Great, you have found a whole set of errors that could occur.

The customers wanted their programs to run though. They did not want to fix all the problems first. And Tony’s company wanted to sell computers. So, he added the null reference. It was not the only solution to the problem. Even back then Sir Tony Hoare knew of another solution with disjoint unions. But null was easier to implement. Now you have to either check every reference at runtime or risk disaster. But at least you could run Fortran on the ALGOL machines.

The problem was that ALGOL inspired many other programming languages such as C or Simula. Simula was billed as ALGOL 60 with classes and considered the first object-oriented programming language. It also inspired Bjarne Stroustrup, creator of C++, and James Gosling, creator of Java [Wikipedia](#). This all leads to Sir Tony Hoare apologising in 2009 for inventing the billion-dollar mistake [infoq](#), humorously I might add.

The Actual Cost

The null reference is a problem prevalent in most modern programming languages. Perhaps the only problem more widespread is the *Year 2k Bug* which did cost an estimated 300 billion USD [Wikipedia](#). In relation the null pointer reference with its countless bugs, system crashes and security problems might actually have cost billions of dollars in maintenance and troubleshooting. It is after all a problem that has been around for 60 years.

That the null pointer is widespread is supported by data from Harness (formerly OverOps) which in 2020 crunched anonymized stats from over a thousand Java applications monitored by them [harness.io](#). They found that 97% of logged errors are caused by only 10 unique errors. The number one error being the `NullPointerException`. It took the top spot in 70% of their production environments.

Language-agnostic is the CWE Top 25 Most Dangerous Software Weaknesses List. The list demonstrates the currently most common and impactful software weaknesses. This list changes year by year as the list reflects the 30.000 reported Common Vulnerabilities and Exposures (CVE) of that year. The NULL Pointer Dereference has been in spot [21 \(2024\)](#), [12 \(2023\)](#), [11 \(2022\)](#), [15 \(2021\)](#) and [13 \(2020\)](#).

That list however “just” looks at security issues. How much time is spent on maintenance and troubleshooting for all the bugs and system crashes is not included.

Next, we will dig into why the maintenance and troubleshooting is so expensive. While reading the next chapters, keep in mind that developers have been fixing these problems for the last 60 years. In total, the mistake might actually have cost around a billion over the years.

The Error Is Not Where You Found It

Take a look at the following code. Can you guess where the error is?

```
1 public String findNotebookMaker(EmployeeId id) {
2     var employee = company.getById(id);
3     if (employee == null)
4         return "Employee does not exist";
5     /* ... */
6     return employee.notebook().maker();
7 }
```

The problem is the following:

```
1     return employee.notebook().maker();
2         ^^^^^^^^^^^^^^^
3     // java.lang.NullPointerException:
4     // Cannot invoke "Notebook.maker()" because the return value of "Employee.notebook()"
        is null
```

Which is impossible.

An employee always has a notebook! We designed it that way!

Bewildered developers

The thing is, under certain conditions (f.ex. notebook is being repaired), an employee does **not** have a notebook. The bewildered developers only knew the state before this feature was added. To them an employee always had a notebook. Or maybe they just **forgot** implementing it. In the end it does not matter. The exception flew anyway, and it is not the problem. The actual problem is here:

```
1 public void startNotebookRepair(EmployeeId id) {
2     var employee = company.getById(id);
3     if (employee == null)
4         return;
5
6     /* ... */
7     company.put(id, employee.withNotebook(null));
8     //           the actual problem ^^^^
9 }
```

The problem is not where the exception occurs but where the null value is set. In production code there can be a lot of conditions that lead to **null** being set. First, we need to find the condition that leads to our null pointer in `findNotebookMaker`.

Then we need to decide if setting null in `startNotebookRepair` was correct or a bug. If it was a bug, we can only hope that no other condition will also set the notebook to null, because that will cause the next exception. If it was correct, we need to decide how `findNotebookMaker` should react to a null notebook. If we just **return null**; then we have just delegated the decision making again.

Clearly stack traces are useless for null pointer exceptions. They do not actually show what caused the problem. We have to search for the actual point of origin and then change the design accordingly. Which is what makes null pointers so expensive.

The wrong way to address this problem is to always check every reference for null before access and always write mitigation strategies. The better way is to find a way to mark what can be null and for the compiler to remind us whenever that changes.

The Trend Towards Safety in Programming Languages

We have already seen that ALGOL W provided a comprehensive type system; that unfortunately did allow the null reference. A feature that most popular programming languages kept ever since. But starting around the year 2000 languages were created and subsequently have become popular that saw things differently.

2005 saw the release of [F#](#), a new functional programming language for .NET. It is billed as empowering “everyone to write succinct, **robust** and performant code”. F# has the **null** keyword for interoperability with C# or VB.NET but it is not permitted as a regular value [microsoft](#). Instead, F# has the **option** type.

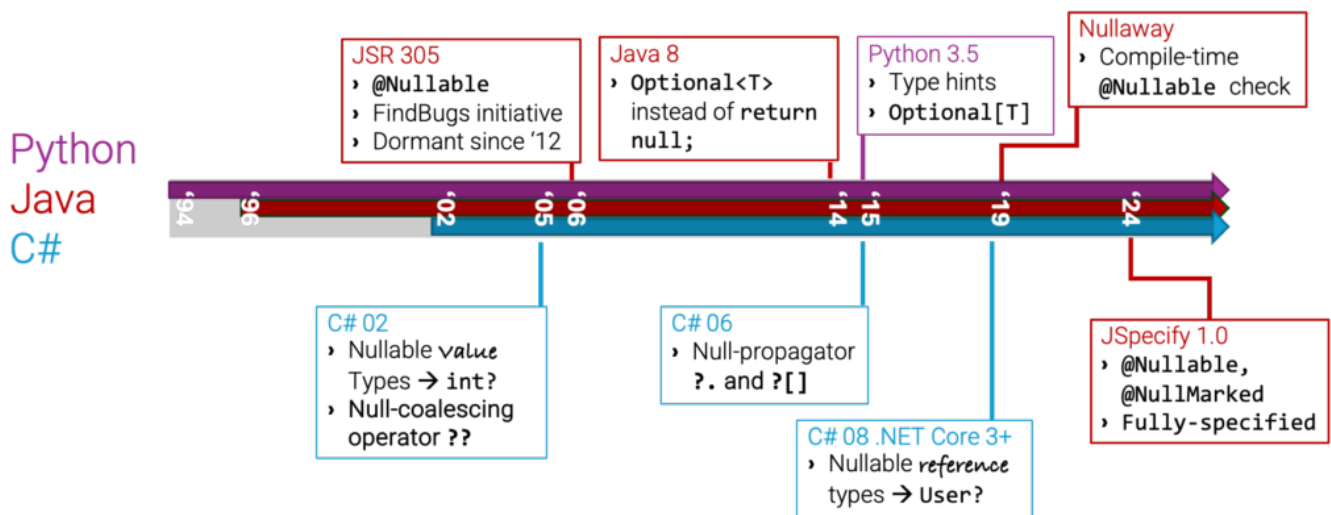
In 2010 we got [Rust](#), “a language empowering everyone to build **reliable** and efficient software.”. In Rust there are no **null** references [rust-lang](#). Instead, Rust has **optional** pointers.

[Kotlin](#) arrived in 2011, offering a “modern, concise and **safe** programming language”. It has **null** but you can only assign it to nullable types [kotlinlang](#).

Robust, reliable, safe. There is a certain trend here. Newer programming languages are designed with the lessons from the past. But what about older programming languages?

Everyone Tries to Deal with Null

According to the surveys by [Redmonk](#), [TIOBE](#) and PYPL, the top 5 programming languages year over year include Python and Java, most often C# as well. These languages are also 20 to 31 years old. Thankfully they have not rested on their laurels but continued to adopt to the changing needs, specifically when it comes to null handling.



1

Abb. 1: Timeline of null-check improvements

Python Tries to Deal with Null

Python, perhaps best known for making everything simple and fun (see [xkcd 353](#) for “proof”) uses dynamic type checking. So it came as a surprise when they added [type hints](#) in 2015. With Python 3.5 it was possible to describe what the expected types and return types were (`def greeting(name: str) -> str:`). It was also possible to declare that something was `Optional[T]` and could be `None`. These hints will inform you that something you try to access might be `None` (Python’s version of **null**). There is still *no type checking happening at runtime* though.

C# Tries to Deal with Null

C#, Java’s “cousin”, has not been shy to evolve. In 2005 the language already added nullable **value** types, i.e. the ability to declare even primitives as nullable (valid: `int? number = null;`). It wasn’t before 2019 that they added nullable **reference** types. These allow classes to be declared as non-nullable (not allowed: `User user = null;`) or nullable (allowed: `User? user = null;`). The compiler enforces that a non-nullable type has to be initialised to non-null and cannot

be set to **null** ever. The compiler also enforces that before accessing a nullable reference, the developer always has to check for **null**, which is where null-propagators `?.` or null-coalescing operators `??` come into play.

In C# finding the Notebook maker could look like the following:

```
1 public string? FindNotebookMaker(EmployeeId id) {&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
2     /* ... */
3     return employee.notebook?.maker();
4     //           ^
5     //           returns null, if notebook is null
6 }
```

or it could look like:

```
1 public string FindNotebookMaker(EmployeeId id) {&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
2     /* ... */  
3     return employee.notebook?.maker() ?? "No maker";  
4     //                                     ^^  
5     //      returns left side if non-null otherwise the "No maker"  
6 }
```

The first example delegates the null-handling to whatever is calling the method. The second example decides that a default value is the right choice. Neither is better and other choices are possible, but we have to choose.

Nullable types force us to make the choice ahead of time. We avoid the run-time bug because we get to make a decision while we design the feature and thus while we have the most knowledge about the problem.

Nullable types make C# designs better since 2019. Not all designs though, because enabling it for all projects would mean so many compile-time errors that no one would upgrade to C#8. The same problem that the Fortran to ALGOL transpiler had. Instead, the nullable types are only enabled in a *nullable aware context* which can gradually be enabled per module and enabled/disabled for individual files.

Java Tries to Deal with Null

Java, known for going the extra mile when it comes to backward compatibility, should have the hardest time fixing the billion-dollar mistake. **null** is so baked into the language that tackling the problem would require “six Ph.D theses, knotted together.” [openjdk](#).

As luck would have it [one JDK project](#) has been doing these Phd theses for the past 10 years and their work will reach preview in the coming years. It would be great though if we could use something **right now**.

Java Has OptionalReturn

Something that has been available in the JDK since 2014 is `Optional<T>`. This type is great to make APIs fluent but it does not actually eliminate null pointer exceptions. It is not suitable for optional fields or parameters either. To see why we have to go back to the original design intention [openjdk](#).

Optional should have actually been called `OptionalReturn<T>`, as in optional method return. It was designed together with the Java 8 Streams API. See the following code:

```
1 String employeeNameById(EmployeeId id){
2     return employees.stream()
3         .search(it -> it.id() == id)
4         .name();
5     //          ^^^^^^^
6     // non-obvious potential NullPointerException
7 }
```

`search()` returns the query result or nothing, when nothing matches. This example where “no result” is represented by `null` is fictional however. In the actual Stream Api they added a limited mechanism to represent “no result” and keep the Api fluent (see also Optional - the mother of all bikesheds [youtube](#)):

```

1 String employeeNameById(EmployeeId id){
2     return employees.stream()
3         .filter(it -> it.id() == id)
4         .findFirst()
5         // returns an Optional
6         .map(Employee::name)
7         .orElse("UNKNOWN");
8 }

```

Methods that might return “no result” can instead return `Optional<T>`. Then the caller has the option how to react to that. What it is not good for is the other direction, passing it as a parameter:

```

1 myMethod(a, Optional.ofNullable(b), c, Optional.ofNullable(d));

```

While not horrible this method call is now quite noisy and widespread use of `Optional.of` makes our code base very cluttered. The clutter becomes even worse when we have to interact with optional fields:

```

1 if(sth.a().isEmpty()
2    && sth.b().isPresent()
3    && sth.b().get().bb().isPresent()){
4
5     aMethod(sth.b().get(), sth.c().get());
6 }

```

Rather than being helpful, `Optional<T>` has become a crutch that clutters the codebase. Additionally it does not actually eliminate `NullPointerException`s because we can still write the following

```

1 record Card(String name, Optional<Integer> value){}
2 void play(Optional<Card> card){ }
3
4 void doSomething(){
5     var card = new Card("Ace", null);
6     //                ^^^^^
7     //                perfectly valid
8     play(null);
9     //    ^^^^^
10    // same for this
11 }

```

It would be much better if could keep the nullness information but without the clutter and with actual protection from null pointers. This is where Nullaway and JSpeccy come into play.

Add Compile-Time Null Checks with Nullaway and JSpeccy

NullAway is a compile-time nullness checker. It was written by Uber because they wanted a tool to eliminate `NullPointerException`s that can be deployed large-scale (see their [2019 paper](#)). In their experience the build-time overhead is less than 10%, so they can run it on every build. It is a plugin for [error prone](#), a static code analyser from Google. To use it you have to add it as a plugin to your gradle or maven file ([full example](#)):

```

1 // add processor to pom.xml (or .gradle):
2 // abbreviated for readability, full example in Github
3 <build><plugins>
4   <plugin>
5     <artifactId>maven-compiler-plugin</>
6     <configuration>
7       <compilerArgs>
8         <arg>
9           -Xplugin:ErrorProne
10          -Xep:NullAway:ERROR
11          -XepOpt:NullAway:AnnotatedPackages=de.richargh.module-a
12        </>
13      </>
14    </>
15    <annotationProcessorPaths>
16      <path>
17        <groupId>com.google.errorprone</>
18        <artifactId>error_prone_core</artifactId>
19        <version>${error-prone.version}</version>
20      </>
21      <path>
22        <groupId>com.uber.nullaway</>
23        <artifactId>nullaway</artifactId>
24        <version>${nullaway.version}</version>
25      </>
26    </>
27  </plugin>
28 </plugins>
29 </build>

```

NullAway will now assume that all variables in the packages `de.richargh.module-x` are non-null by default. Any line where we pass null to a non-nullable field will be greeted with a compile-time error like this: `[NullAway] passing @Nullable parameter 'null' where @NonNull is required`

Next, we can start marking fields that are actually nullable as such. Up until 2024 we would have used the quasi-standard [JSR-305](#) for that. It was a set of annotations proposed by the static analyser FindBugs. Although widely adopted ([maven central lists](#) packages from Google, Atlassian, Eclipse and many more) the JSR has been dormant since 2012 and was never finalised, the annotations never completely defined. Until [JSpecify](#) took up the work.

JSpecify is a set of annotations, specifications and documentation for static analysis. It is defined by Google, Oracle, Uber (NullAway), Broadcom (Spring), JetBrains, PMD, Sonar, and many more. Version 1.0.0 was released 2024 and includes four nullness annotations that are generally useful in the following pairings:

- To declare types in package, class, method, or constructor as non-null by default, use `@NotNullMarked`. Anything that can be null needs to be explicitly marked as `@Nullable`.
- To declare types in package, class, method, or constructor as null by default, use `@NullUnmarked`. Anything that can never be null needs to be explicitly marked as `@NonNull`.

To use these annotations with NullAway you simply have to depend on them:

```

1 // abbreviated for readability
2 <dependencies>
3   <dependency>
4     <groupId>org.jSpecify</groupId>
5     <artifactId>jSpecify</artifactId>
6     <version>1.0.0</version>
7   </dependency>
8 </dependencies>

```

No additional configuration is necessary. Uber, who developed NullAway, is after all a member of JSpecify. So is Broadcom, the developers of Spring. The next Spring Framework is currently shipping the first milestone releases where their own JSR305 annotations are completely replaced with JSpecify. They also use NullAway to enforce those [spring.io](#). Whether Spring Boot 4 will also have all the new annotations is still not decided as of time of writing.

Fixing the Billion-Dollar Problem with the Compiler

Once you have the annotations you simply have to use them to mark something as `@Nullable`. Our notebook repair example becomes:

```
1 /// Shop.java v2  
2 public void startNotebookRepair(EmployeeId id) {&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
3     var employee = company.getById(id);  
4     if (employee == null)  
5         return;  
6  
7     /* ... */  
8     company.put(id, employee.withoutNotebook());  
9 }  
  
10  
11 /// Employee.java v2  
12 public record Employee(  
13     EmployeeId id,  
14     String name,  
15     @Nullable Notebook notebook) {  
16     //      ^^^^^^^^  
17     // new JSpecify annotation  
  
18     /* withoutNotebook() ... */  
19 }  
20
```

Which means the following throws a compile-time exception:

[illegible]

And we have to design a solution:

[illegible]

The compiler helped us immensely. It forced us to fix the root cause, not a symptom. We had to specify whether setting `null` was allowed in the first place.

This compile-time check is the same approach that Rust, F#, as well as Kotlin have picked and the one that C# migrated toward. Granted the other programming languages picked a different syntax but the idea remains the same. In fact, the idea remains the same as the one that Sir Tony Hoare had in 1965.

Gradual Migration

Adding NullAway to a code base is amazing. You will catch so many bugs hiding in plain sight. Every green field project should do so. But adding it to the entirety of existing projects with thousands or millions of code lines would be a nightmare. This

is where the configurations options of NullAway [github](#) come into play. They're all prefixed with `-XepOpt:NullAway`.

The simplest way to start is with `:AnnotatedPackages=de.richargh.module-a,de.richargh.module-b`, which takes comma-separated list of packages to check. Start with one module or even just a small part of a module, then gradually add more. If you just want to **experience** NullAway, then start with a **simple** module. Generally it is better to start in a **business-critical** module. Even if the migration takes longer, the value of migration is just that much higher and easier to pitch. Once you have migrated enough modules you might decide to flip the inclusion. Include all and strip those you have not migrated yet via `:UnannotatedSubPackages=de.richargh.module-y,de.richargh.module-z`
`:AnnotatedPackages=de.richargh`.

The alternative to `AnnotatedPackages` is to mark packages that NullAway should check directly via JSpecify annotations. To do so replace `:AnnotatedPackages=` with `:OnlyNullMarked=true`. Then null-mark all desired modules in their respective `package-info.java`:

```
1 @NullMarked
2 package de.richargh.module-a;
3
4 import org.jspecify.annotations.NullMarked;
5
6 // place this package-info.java into the root of the module you want to mark
```

Eventually you will also want to flip to exclusions here, null-mark your whole project and exclude modules:

```
1 @NullUnmarked
2 package de.richargh.module-z;
3
4 import org.jspecify.annotations.NullUnmarked;
```

Whatever approach you pick, you need to be careful of objects that come from unmarked modules. NullAway has no information about the nullness of objects from beyond the edge of a module. It will let you access them without checks.

It is thus in your best interest to quickly migrate modules that are often exchanging data with each other. A prime example is a module that is used by multiple modules. Until this shared module is checked with NullAway, you cannot trust the output in any other module, even if that module is already checked with NullAway. Otherwise your null-safety will always be on shaky ground.

6+1 Tips to Get the Most Out of Null-Checking

Don't trust beyond the edge.

Anything not checked by NullAway is beyond the edge. As previously discussed, that includes your own unchecked code. It also includes most third-party code. Spring Framework being a rare, annotated exception. Particularly dangerous is deserialised json however. Mapping-Frameworks like Jackson see null-checks as validation. And Jackson does not do validation:

```
1 /// RenterDto.java
2 public record RenterDto(
3     String id,
4     String name){
5 }
6
7 /// somewhere else, perhaps in a Controller
8 var json = "{ \"name\": \"Alex\" }";
9 var result = objectMapper.readValue(json, RenterDto.class);
10 //
11 //
12 // result = RenterDto[id=null, name=Alex]
13 result.id() == null; // ...true
```

So be sure you know what happens at the edge. For Jackson in particular you might want to add validation checks. You could write those with Hibernate Validator and write your own [ConstraintValidator](#). Here is a custom [DefaultIsNonnullableValidator](#) to get you started.

Make non-null your default

NullAway assumes this by default. It is still worth repeating. Something being optional is the exception, not the rule.

Model Optional Fields with @Nullable

Do not use `Optional<T>` but stick to annotations:

```
1 public class <coolClass> {
2     private @Nullable Address address;
3 }
4
5 public record <coolRecord> (
6     @Nullable Address address){
7 }
```

Never pass null, consider explicit methods instead

```
1 // if you have this method:
2 var p1 = notebookPriceFor(
3     employeeId, notebookType);
4
5 // then do not do this:
6 var p2 = notebookPriceFor(
7     null, notebookType);
8 // ^^^^
9 // What does passing null mean semantically?
10
11 // create an explicit method that indicates that a query without an employee, is just an
12 // estimate
13 var p2 = estimateNotebookPrice(
14     notebookType);
```

Mark nullable method returns

```
1 @Nullable myMethod(A a, B b) {
2     /*...*/
3 }
```

Occasionally consider Optional for method returns, which gives choices to caller

Suppose you have the following repository method:

```
1 // Optional is quite useful for repositories
2 Optional<Address> getAddressById(employeeId id) {
3     return Optional.ofNullable(addresses.get(id));
4 }
```

Then the caller has all the choices:

```

1 // Choice A: caller says null is not allowed
2 var account = accounts.findAccountById(id).orElseThrow();
3
4 // Choice B: caller says default is possible
5 var account = accounts.findAccountById(id).orElse(Account.none());
6
7 // Choice C: caller uses fluent chain
8 var budget = accounts.findAccountById(id)
9                 .map(Account::budgetId)
10                .flatMap(budgets::getById)

```

Return empty collections or arrays, not nulls

```

1 // not like this
2 // just makes the API more difficult to use for all callers
3 @Nullable List<Address> recentAddresses(){
4     return employee.isTrackingAllowed()
5                     ? recentAddresses
6                     : null;
7 }
8
9 // like this
10 List<Address> recentAddresses(){
11     return employee.isTrackingAllowed()
12                 ? recentAddresses
13                 : Collections.emptyList();
14 }

```

Use JDK 16 records for domain types

```

1 /// Address.java
2 public record Address(
3     String city,
4     @Nullable String street
5 ) {
6 }
7
8 /// somewhere-else.java
9 var address1 = new Address("
10     Brühl, "Berggeiststraße 31-41");
11 var address2 = new Address("
12     Brühl);
13 //      ^^
14 // Compile-time error because parameter is missing
15 // null is allowed, we just have to be explicit about it

```

When working with null-checks, you want to gradually reduce ambiguity. A typical source of ambiguity are partially instantiated domain objects. Records do not allow such cases and require all parameters to be declared, even if they are null. Records are also very compact and reduce a lot of boilerplate. If your records have a lot of states in which they can start out, then consider using static factory methods to instantiate them.

The Future – Six Ph.D. Theses, Knotted Together

Last I talked about fixing the billion-dollar mistake (JavaLand 2023), I was hoping we would get JDK null-safety sometime around 2029.

Roughly one year later, Brian Goetz, steward of the Java language, held a talk about [Java’s Epic Refactor](#). In this talk he gave major updates on [Project Valhalla](#), aka “six Ph.D theses, knotted together.” [openjdk](#). Their goal is to bring value classes into Java that “code like a class, work like an int”, a goal which is very different from any null-marking support. Brian announced that as part of that work they would also bring *Null-Restricted and Nullable Types* [openjdk](#) in Java.

Once this enhancement lands in the JDK, we would be able to do the same thing that NullAway with JSpecify gives us today, but without NullAway or JSpecify and with a more compact syntax. Instead of `@Nullable` we would postfix types with `?`, instead of `@NonNull` we would postfix types with `!`. Instead of a compiler *plugin*, all would be checked by the *compiler*. Our employee would now look like this:

```
1 public record Employee(  
2     EmployeeId! id,  
3     String! name,  
4     Notebook? notebook) {  
5 }  
6 // very similar to how employee would be modelled in C# or Kotlin.
```

The `!` are currently a requirement to mark something as *null-restricted*.. Without those markers the code stays backward compatible and the nullness of a type is *unspecified*. No spec has been finalised though.

It would be great to have something like `@NullMarked` in Java as well. Something that we can set, so we only have to mark *nullable* types with `?` and that is indeed part of suggested future work [openjdk](#). Before this simplification can happen, the actual *Null-Restricted and Nullable Types* have to be implemented, and that is the crux of the matter. Restricted and nullable types are not the goal of Project Valhalla. They are something the project needs to make value types extra awesome. I think we will get them, but take that with a grain of salt. I am not a Valhalla developer. In the meantime, NullAway is a great choice to finally fix the billion-dollar mistake.

References

- [NullAway](#) has good documentation if you want to get started
- My full NullAway+maven example can be found [here](#)
- My previous presentation slides and recordings for the topic can be found [here](#)

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



SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES



www.javapro.io
www.javapro.io



Java 24: A Story of Code, Conflict, and Conquer

Java24 # VirtualThreats # ObjectHeaders

Author: Lutske de Leeuw

Lutske de Leeuw is a Software Engineer at Craftsmen, specializing in full-stack development with a focus on Java and Angular. Passionate about machine learning, board games, and cats. Committed to sharing knowledge so that everyone can benefit. Loves organizing events like Devox4Kids, ApeldoornJUG, JUG Noord, and other knowledge-sharing initiatives.



LinkedIn

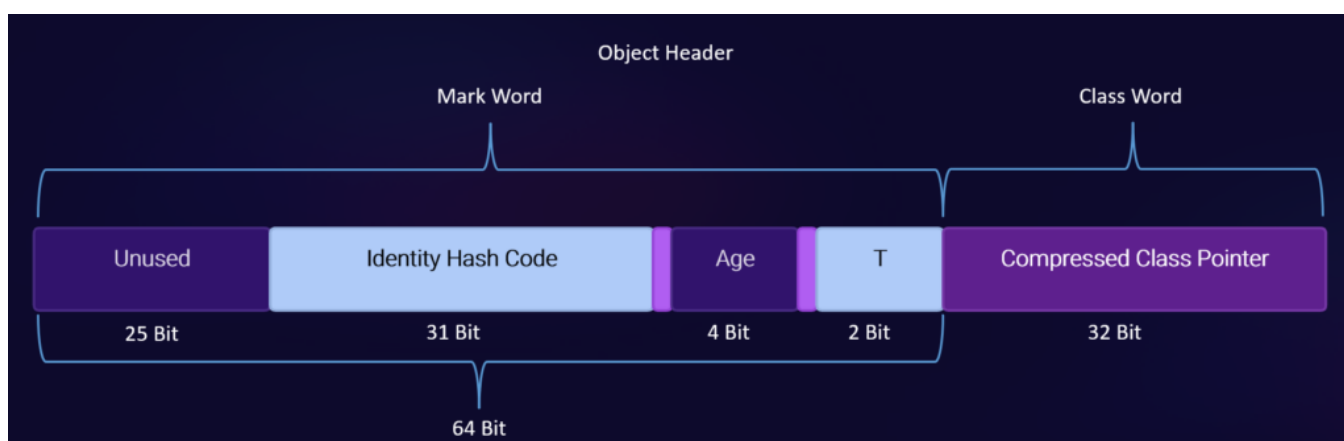
In a digital world far away from here, runs a Java application without problems. The logs are quiet, the servers are stable, and Emma and her team live in peace. But then, the first attack arrives! First innocent. A little spike in the memory use. But then another, and another. Within a couple of days, it turns into a disaster. Garbage collectors are going wild. The latency rises. The servers are shaking. The memory realm is flooded by invisible enemies: unnecessarily inflated object headers. They hide in every object, wasting valuable bytes and creating chaos in the heap.

This is the start of the memory wars!

Emma and her team are puzzled. Their code hasn't changed. Their traffic is stable. Where does this growth come from? They dig into the heap and discover the hidden enemy: the unnecessarily inflated object headers.

What Are Object Headers?

In Java, every object in memory carries a small amount of extra data known as an object header. This header stores important metadata that the JVM uses to manage the object. Typically, it includes things like a mark word (for garbage collection, synchronization and hash codes) and a class pointer (to identify the object's class and its methods). While this metadata is crucial for the JVM's functionality it adds overhead. For example, each object can carry 12 to 16 bytes of header data. Which adds up quickly in applications with millions of objects, leading to wasted memory and increased garbage collection pressure.



When Is an Object Header Created?

An object header is created the moment an object is instantiated with the new keyword. When the JVM allocates memory for an object on the heap, it not only reserves space for the object's fields but also for its header. This happens even before the object's constructor finishes running. So, from the very beginning of an object's life, the JVM is already tracking it with that metadata.

A silent war rages in their system. The enemy is invisible, but the damage is real. Emma's team is fighting with everything they got. They tweak the garbage collection. They analyse the heap dumps. They try manual optimisations. Even object pooling isn't working. But the object headers keep growing as an enemy army. Every object takes 16 bytes of unnecessary ballast with them. The servers are getting overloaded. The application is in danger of being defeat.

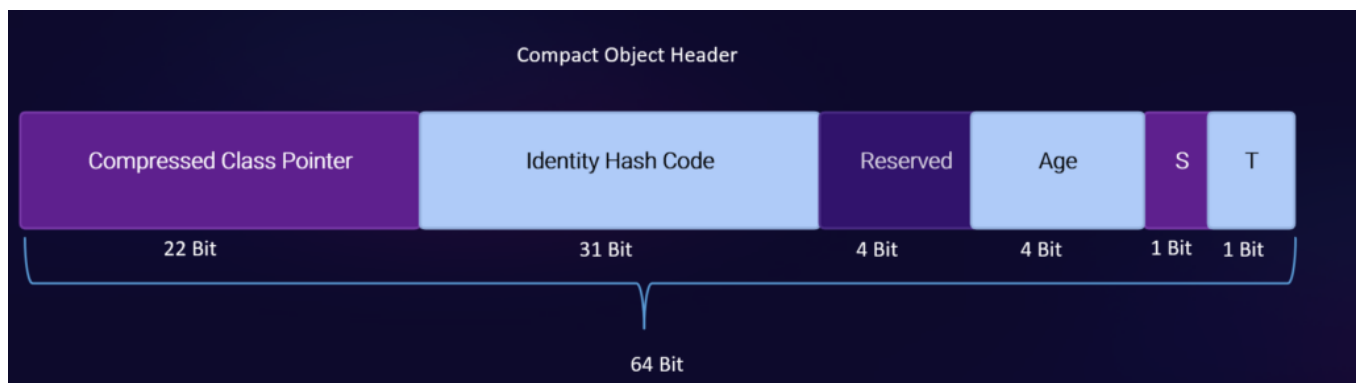
Emma stands before a hard choice. Does she keep fighting with old techniques and endless optimisations, or does she search for a new weapon? Something that can change this war? The team feels the pressure or the deadlines and the increasing performance issues. There is tension in the air. The enemy seems stronger than ever, the change to get grip on the situation, slims every minute.

But deep in the Java 24 release notes, Emma discovers the secret weapon: JEP 450, compact object headers! JEP stands for JDK Enhancement Proposal

Why Is the Compact Object Header Slimmer?

The compact object header in Java 24 is slimmer because it trims away unused bits and optimizes how essential metadata is stored. For example, the old object header reserved bits for features like biased locking, which sped up synchronization

but was disabled in Java 15. Those bits were just sitting there, useless. The compact header repurposes or discards these bits, packing necessary data, like class pointers and mark words, more efficiently. This tighter layout reduces the header size from 12 or 16 bytes to as little as 8 bytes in some cases, meaning less memory overhead per object and better cache locality for faster access.



This is no normal update. This is a strategic breakthrough in the war against memory waste. Emma takes her change. She starts the application with this new weapon:

```
1 java -XX:+UseCompactObjectHeaders -jar mijn-geweldige-craftsmen.jar
```

The -XX is needed because this is still an experimental feature in Java 24.

The clock ticks. Emma runs her application with the new activated weapon. She sees immediate change. The memory usage slims. The object headers shrink, freeing up more space in the heap. The system, which was just about to collapse under the pressure, is finally breathing a sigh of relief.

Because of the JEP 450 flag, the JVM optimizes object headers by reorganizing how metadata is stored. It's compressing this metadata, so headers can shrink by up to 50%. Reducing memory waste, improving cache efficiency, and lowering garbage collection activity. Emma watches her metrics. Memory usage drops, garbage collection pauses shorten, and response times improve. Without touching a single line of code, her application transformed into a lean, efficient machine. All thanks to the power of compact object headers!

The memory wars show us an important lesson. In every application is a silent war raging. It's not just about writing good code, but also on using the opportunities that the JVM gives us. And unused bytes and unnecessary overhead can lead to a big problem, when they aren't addressed.

Slow Startup Time

The dust had barely settled from the Memory Wars. The team felt victorious. But in software, peace is fleeting.

It was a regular Friday afternoon. The team was winding down, wrapping up their last commits, and debating what everybody was going to eat that night. Emma, always the curious one, lingered a little longer at her desk. When John, the backend engineer with an uncanny ability to break things, let out a dramatic sigh.

"Our app takes forever to start. I wish we could fix that!" he blurted, spinning slowly in his chair.

Emma chuckled. "It's not that bad, right?"

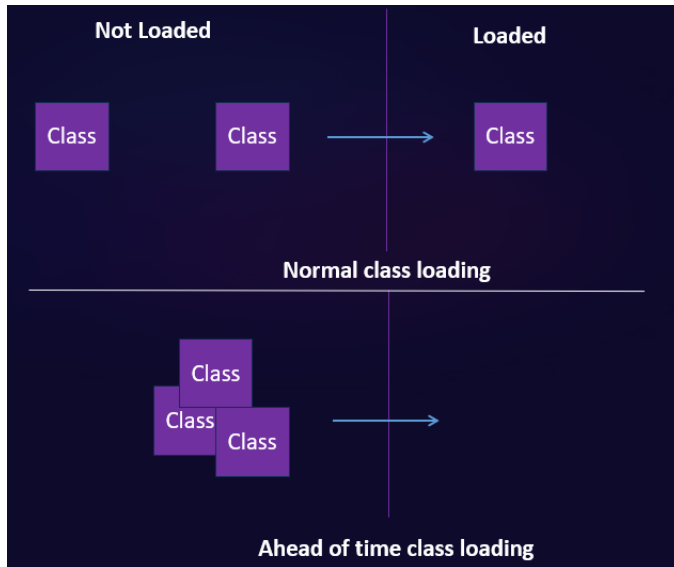
John stopped spinning and stared at her. "I went for a coffee, came back, and it was still booting. And I make slow coffee!"

Everyone laughed, but Emma couldn't shake the thought. The weekend arrived, but her mind kept replaying John's words. Could they really fix the startup time? And why did she care so much? Was it just professional pride or maybe something else?

Saturday morning Emma sat at her desk, with a fast coffee in her hand. Determined to crack the mystery. She dove into research. Digging through JVM documentation like an archaeologist searching for a lost treasure. What she found was fascinating.

How Does Class Loading Work?

The JVM, like an overcautious librarian, loads classes on demand. Whenever the application references a class. The JVM scrambles to find and load it, resolving dependencies as it goes. In small applications, this isn't a big deal. But in large, complex systems, especially microservices, this class-by-class loading adds serious startup lag. It's like inviting guests to a party and making each one wait at the door while you look up their name on a list. Emma leaned back, frowning. "What if, we could load the guests before the party starts?"



JEP 483: Ahead-of-Time Class Loading & Linking

By Sunday afternoon, Emma had discovered her golden ticket: JEP 483, Ahead-of-Time class loading & linking. JEP 483 introduced a way to optimize class loading by allowing the JVM to load and link classes ahead of time. Instead of waiting to load each class on demand. The JVM can preload and prepare them during the build process. Reducing the work needed at runtime. It's like the JVM setting up the entire party room in advance, so when the application starts, it can just swing the doors open and let everyone in without delay.

Emma's heart raced. She could see the solution taking shape. She didn't text John. Not yet. She wanted to test it herself.

Sunday evening. Emma's living room was dark except for the glow of her screen. She started by packaging her hobby project:

mvn clean package

Running an executable JAR in production isn't always ideal. So, she created an exploded Jar instead:

```
1 java -Djarmode=tools -jar target/this-amazing-application.jar extract
```

With her app ready, she measured the baseline startup time:

```
1 java -jar this-amazing-application.jar
```

The app started in 2.09 seconds. Not bad for a small side project. But Emma wanted more speed. Next, she recorded the application run to capture loaded classes:

```
1 java -XX:AOTMode=record -XX:AOTConfiguration=app.aotconf
2     -cp this-amazing-application.jar
3     com.emmasworld.common.ThisAmazingApplication
```

Note: By the time this article is published, Java 24 will be out, and you might not need the -XX flag anymore, as AOT features will no longer be experimental.

After using the app, click here and there, to load the necessary classes. Emma created the ahead-of-time (AOT) bundle:

```
1 java -XX:AOTMode=create -XX:AOTConfiguration=app.aotconf
2     -XX:AOTCache=app.aot -cp this-amazing-application.jar
```

Finally, she ran the app with the ahead of time cache:

```
1 java -XX:AOTCache=app.aot
2     -cp this-amazing-application.jar
3     com.emmasworld.common.ThisAmazingApplication
```

It worked! The app now started in 1.09 seconds. Almost twice as fast. Emma jumped up, startling her cat awake with her victory cheer.

The Rust Temptation

The team basked in the glow of their recent success, having optimized their Java application's startup time by 40%. Yet, John, ever the performance enthusiast, remained restless

One morning, he casually floated an idea during the daily stand-up:

"What if we rewrote the performance-critical parts of our application in Rust?"

The suggestion hung in the air, met with raised eyebrows and thoughtful nods. Rust, with its promise of memory safety and zero-cost abstractions, had been making waves in the systems programming world. John argued that its efficiency could push their application's performance to new heights.

Emma listened intently but felt a pang of hesitation. Transitioning to Rust would mean a steep learning curve and potential integration challenges. She wondered if there was a way to achieve similar performance gains within the Java ecosystem they were already familiar with.

Rust is fast and handles threads well. But managing millions of threads manually? That's like untangling a pile of yarn by hand. Meanwhile, Emma's cat, oblivious to the elegance of threads, got tangled in real-life threads from her knitting project. As Emma carefully freed him, she laughed. "Let's stick with Java," she said, watching her cat bat at the remaining yarn. "It's less of a headache for both of us."

JEP 491: Synchronize Virtual Threads without Pinning

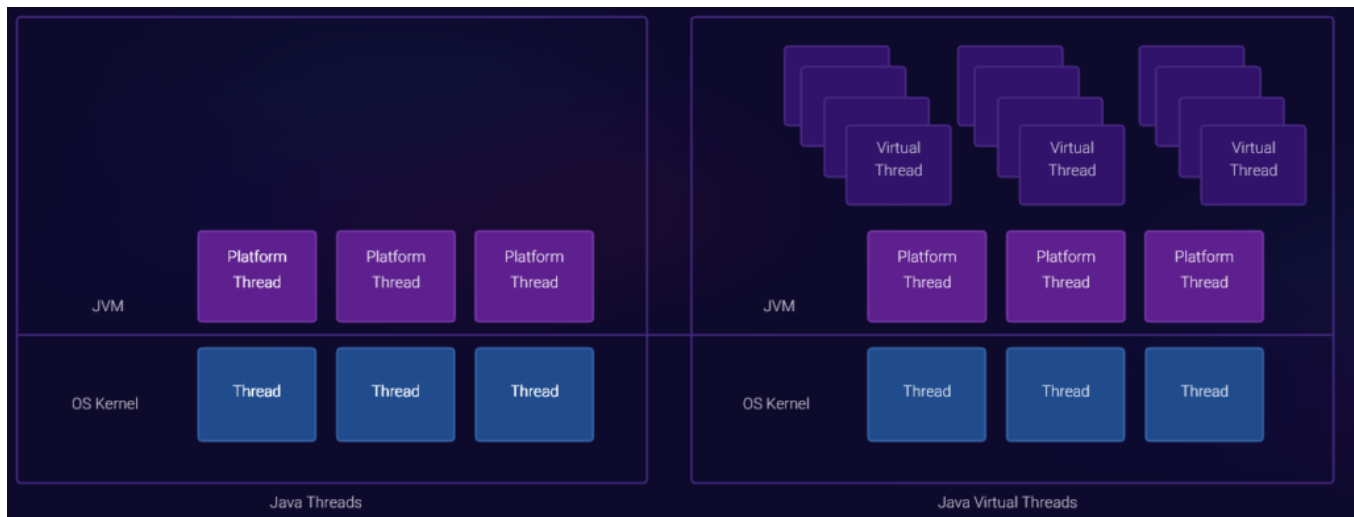
Determined to explore all avenues, Emma delved into recent advancements in Java. Her research led her to JEP 491: Synchronize Virtual Threads without Pinning. This enhancement aimed to improve the scalability of Java applications by allowing virtual threads to release their underlying platform threads when blocked in synchronized methods or statements. This change would enable applications to handle a massive number of threads more efficiently.

Prior to this enhancement, virtual threads were pinned to platform threads when entering synchronized blocks, limiting scalability. With JEP 491, this limitation was addressed, allowing virtual threads to unmount even within synchronized contexts, thus freeing up platform threads for other tasks.

Emma saw potential in this approach. By leveraging virtual threads, they could achieve high concurrency without the complexities of integrating a new programming language.

How Threads Works

In classic Java, when you create a thread, it maps directly to an operating system (OS) thread. The JVM (Java Virtual Machine) asks the OS to create a "platform thread," which is a wrapper around a real kernel-level thread. The OS manages this thread, switching between different threads when needed a process called "context switching." But these platform threads are heavy. They use a lot of memory, and the OS can only handle so many at a time. If a thread waits (like for a database call), the platform thread is stuck doing nothing, wasting resources.



How Virtual Threads Work

Virtual threads change the game! Instead of each Java thread using an OS thread, the JVM creates lightweight virtual threads. These virtual threads run on top of a small pool of platform threads, like passengers taking turns on a limited number of seats. When a virtual thread needs to wait, Java pauses it, freeing up the platform thread to handle another task. When the waiting is over, Java resumes the virtual thread right where it stops. This approach massively reduces memory use and lets Java apps run millions of threads without overwhelming the OS. It's a simpler, more efficient way to handle lots of concurrent tasks!

```
1 synchronized byte[] getData{
2     byte[] buf = ...
3     int nread = socket.getInputStream().read(buf);
4 }
```

If this method blocks during the read operation, the JVM will pin the virtual thread to its platform thread, essentially consuming the resources until data is available. Such blocking limits scalability, as pinned virtual threads occupy platform threads that could otherwise support additional virtual threads. JEP 491 proposes allowing virtual threads to unmount even within synchronized blocks, freeing their platform carriers to execute other virtual threads.

```
1 synchronized void waitForCondition() {
2     while(!condition){
3         wait();
4     }
5 }
```

For example, a virtual thread that calls `Object.wait()` in a synchronized block will unmount and remount when signaled to continue, as shown below:

With this approach, applications can scale better without requiring developers to abandon **synchronized** in favour of `java.util.concurrent` locks.

Experimenting with JEP 491

Eager to validate her findings, Emma proposed an experiment. They would refactor a performance-critical section of their application to utilize virtual threads and observe the impact. The team agreed, intrigued by the possibility.

Over the next week, they refactored the application's data processing module to use virtual threads. This involved replacing traditional thread pools with virtual threads, allowing each task to run independently without consuming significant system resources.

Upon deployment, the results were impressive. The application's throughput increased by 50%, and resource utilization dropped significantly. Tasks that previously waited for thread availability now executed promptly, leading to a smoother and more responsive system.

Empowering a Team of Java Explorers

Emma's relentless curiosity didn't just speed up their app, it sparked something bigger. Her team, energized by the success with virtual threads and compact headers, started hosting regular "JEP Fridays." Every week, someone would dive into a new JDK Enhancement Proposal, experimenting, sharing findings, and brainstorming ways to improve their system. Even failed experiments were valuable, turning into mini-knowledge sessions that helped everyone level up.

The Power of Shared Knowledge

What started as a quest to fix startup times became a culture shift. Emma, once just the developer who stayed late out of curiosity, became a mentor who encouraged others to explore, break things, and learn. Because that's the magic of knowledge-sharing. It turns individual discoveries into team superpowers. And while they never rewrote the app in Rust, John did knit Emma's cat a tiny sweater out of leftover thread.

Written by Lutske de Leeuw, software engineer at Craftsmen.

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



More Action, More Overview

Concurrency

ScopedValues

VirtualThreads

ScopedConcurrency

Author: Merlin Bögershausen

Merlin is a Software Engineer, Architect and Oracle ACE with over 10 years of experience in different domains and languages. His main focus is Java Enterprise Applications with modern and Next-Generation Java. As a Migration Engineer, he helps teams and individuals utilize new features and supports them in migrating. Besides developing, speaking at conferences and his parental role, he tries to participate in community events, teach folk to land gliders (yes, I am a Flight Instructor!) and play volleyball.



LinkedIn

Java applications have to contend with various concurrency problems. There is a solution here with new constructs.

Java concurrent programming is as old as Java itself. The first version already contained the basic constructs on which today's concurrent processes are based. Threads and Runables make it possible to decouple the sequential processing of operations and achieve higher throughput. Over the years, a number of changes have been made to Java's concurrency API to address various weaknesses. The ExecutorService API to save resources or the Future API to avoid having to wait for long-running operations. Now there has been renewed movement in the concurrency topic, virtual threads decouple the JVM threads from the operating system in order to save further resources and make the JVM more efficient. This opens up new possibilities for developing elegant APIs. In this article, I would first like to take a look back and consider which abstractions are available, what the difficulties were and why virtual threads are a solution here. I will then introduce the constructs Structured Concurrency and Scope Values that build on this and look at the whole thing using a few examples. These examples are available on GitHub as GIST and can be viewed in addition to this article.

Threads in General

Threads are a concurrent programming construct used to group operations together and execute these groups in parallel. The operations within a thread run in a certain order. A thread is therefore a small program within a program. A thread is created within a thread or process. This operation is called fork and is outlined in the figure below.

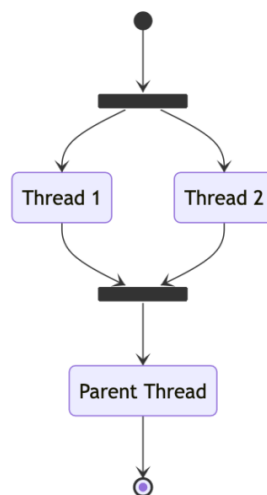


Abb. 2: The process of creating a thread in general

Process of creating a thread in general When forking, a new thread is created from a process or thread as its child, whereby a thread can have several children but only one parent thread. The parent thread creates two new threads that exist independently of each other and are executed. After they have completed their operations, the execution reunites in the parent thread. The parent thread can only be terminated once all child threads have been completed. The join operation can be used to explicitly wait for the completion of a child thread or the join can take place implicitly before the parent thread is cleaned up.

However, a thread consists not only of its program code, but also of metadata that is used so that the operating system can execute threads. This metadata includes its own ID, the ID of the parent thread, a specific area in the memory area and a status. This status is necessary to understand why the direct coupling between an operating system thread and a JVM thread can be problematic, the status diagram is given here.

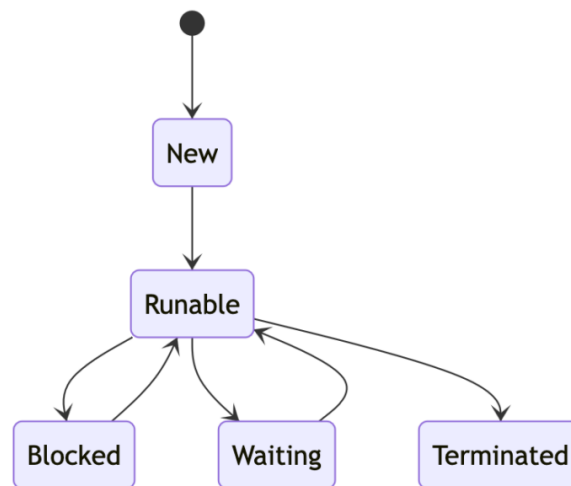


Abb. 3: Status diagram of a thread

A thread first starts in the New status. In this status, it is not yet executable and is not executed. This status is intended for initialization. Areas are allocated on the memory, the IDs are set and priorities are configured. Only when the configuration is complete does the thread switch to the Runnable status. In this status, the thread can be selected for execution by the operating system. If the thread is selected, it executes a certain number of operations from its program until it is interrupted. Normally, the thread then remains in the Runnable status. However, the thread can also switch to the Terminated status after it has completed its work. In this status, the thread is finished and is no longer selected for execution, but still occupies resources. Two further states are Blocked and Waiting, both of which signal a thread that is currently not executable. Blocked refers to a thread that is blocked by another thread. For example, by a synchronized block or access to IO. The Waiting status indicates that the thread is currently waiting for another thread. For example, it is waiting for a response. These two states, Blocked and Waiting, are states that affect the executability of a thread. All five states are mirrored in the JVM. This means that if a JVM thread is marked as Blocked, the corresponding thread is also marked as Blocked in the operating system, as JVM threads are a mirror image of operating system threads. In all operating systems, there is an upper limit for the number of threads. If this limit is reached, no new thread can be created and the operating system is unable to act. For this reason, concurrent programming tries to use threads sparingly and not execute every operation in a new thread.

Java Threads API

The Thread API in Java has existed since version 1.0 and has only experienced small changes. A thread in the JVM always reflects an operating system thread. To define a thread, the class `Java.lang` is expanded and the `RUN` method is implemented with the required operations. In the example below, a class of example thread is defined. Like any class, this can be transferred to an instance with the `New` Operator.

```

1 class ExampleThread extends Thread {
2     public void run() { /* Do it */ }
3 }
4
5 var t1 = new ExampleThread();
6
7 t1.start(); t1.join();

```

This instance of the thread is the smallest schedulable unit of the JVM. However, as already outlined in the last paragraph, it is not the JVM but the operating system that manages the selection of the running thread. After creation with `new`, the JVM and platform thread have the status `New` and have been initialized. In our Java program, further initializations can be carried out before the thread is transferred from the `New` status to the `Runnable` status with the `Thread#start` method call. If the thread is selected for execution by the platform's scheduler, processing of the logic starts. To wait for the scheduling of a child thread in the generating thread, there is the `Thread#join` method and a version with an explicit timeout.

Another way to execute code in parallel is to implement the Runnable interface as shown in the example.

```
1 class ExampleRunnable implements Runnable {
2     public void run() { /* Do it again */}
3 }
4
5 var runnable = new ExampleRunnable();
6 var t2 = new Thread(runnable);
```

A runnable is a component of a thread and the smallest unit of concurrent work. A runnable groups operations that belong together. The big difference between a runnable and a thread is that a runnable is not bound to a platform thread. Instead, it defines which operations are executed and thus decouples the tasks from the technical execution.

The two concurrency models in Java also aim to decouple technical thread management, execution and the flow of business logic. The `ExecutorService` is an interface and has been available since Java 5. Its aim is to use the operating system threads as efficiently as possible. Tasks can be provided to it in the form of a `Runnable`, which are executed later. An `ExecutorService` can use one or more threads to carry out the work. The `Future` construct has been around since Java 5 and has the purpose of executing long-running operations concurrently. This means that the business code does not have to be blocked while waiting for a response from a database. Both can be combined and lead to an efficient decoupling of long-running business code as well as resource-saving parallel execution. Both solutions share two problems

1. Exceptions that are thrown in the concurrent operations are encapsulated in other exceptions and thus make error handling not straightforward
2. A blocked execution also blocks the operating system thread used

In order to be able to use the operating system threads even more efficiently, the direct connection between the operating system and the JVM thread must be resolved.

JEP 444 Virtual Threads

JEP 444 Virtual Threads is intended to break the link between JVM and operating system threads. As a result, virtual threads are no longer bound to an operating system thread. As less information has to be stored, virtual threads are significantly lighter than platform threads. Of course, a virtual thread must also be executed by a platform thread and therefore an operating system thread. The efficient use of the available threads is realized at the JVM level.

The API finalized by JEP 444 in Java 21 is rather small, but the necessary changes within the JVM were all the more enormous. At the core of the API changes for software developers is the new factory method in the listing below.

```
1 Thread.ofVirtual()
2     .name("virtualThread-1")
3     .inheritInheritableThreadLocals(true)
4     .uncaughtExceptionHandler((thread, throwable) ->
5         System.err.println(thread.getName() + ": " + throwable))
6     .start(() -> IO.println("Hello from virtual thread"));
```

`Thread#ofVirtual` is used to create a type of builder for a virtual thread. The name is then set and it is configured that the `ThreadLocals` are to be inherited. In addition, an explicit fallback `ExceptionHandler` is created. The `start` method is used to configure the runnable to be executed and start the thread. No separate operating system thread is created here; when a platform thread is created with the `Thread#ofPlatform` method, an operating system thread is also created. After creation, the virtual and platform threads behave in the same way from an application perspective.

In addition to the changes in the thread class, a new `ExecutorService` has also been provided. This uses a new virtual thread for each task instead of a thread pool.

```
1 Executors
2     .newVirtualThreadPerTaskExecutor()
3     .submit(() -> IO.println("Hello, World!"));
```

The changes to the API introduced by JEP 444 are not major, but the features they enable are. The new language constructs Structured Concurrency of Scope Values are based on the virtual threads finalized in Java 21.

Structured Concurrency

The aim of the Structured Concurrency feature, which is currently in preview, is to understand related tasks that are executed in different threads as a single unit of work. This is intended to improve error handling, abort behavior, readability and maintainability. An example of a group of related tasks can be found in the illustration.

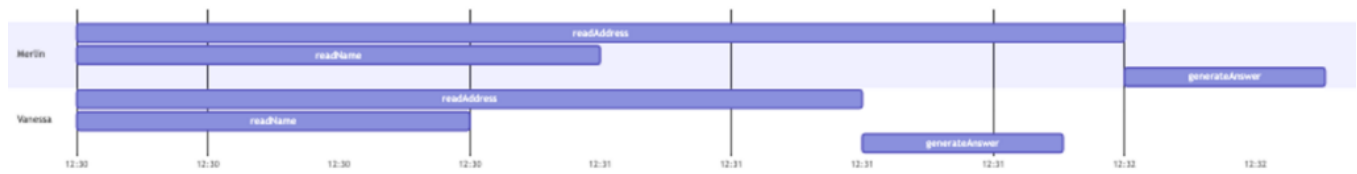


Abb. 4: Exemplary processing of a call in a service with communication to other services

This example shows two exemplary calls to a service. For each call, this service loads an address and names from databases or from other services. If both pieces of information are available, a response is generated. The address and name are loaded in parallel so that execution is not prolonged. Once they are available, the response is generated. Until the introduction of Structured Concurrency, this procedure would have been realized with some `CompletableFutures`. The problem with futures is the non-optimal error handling and the poor readability due to scattered interactions with the results. The `StructuredTaskScope` API (see listing) addresses all shortcomings.

```
1 public class StructuredTaskScope<T> implements AutoCloseable {
2     public <U extends T> Subtask<U> fork(Callable<? extends U> task);
3
4     public StructuredTaskScope<T> join() throws InterruptedException;
5     public StructuredTaskScope<T> joinUntil(Instant deadline)
6         throws InterruptedException, TimeoutException;
7
8     public void shutdown();
9     public void close();
10 }
```

The `StructuredTaskScope` implements `AutoCloseable`, indicating that it is intended for use with a Try-with-Resource block. This block makes it clear which tasks belong together and consolidates error handling. Within a `StructuredTaskScope`, a new task can be initialized using `fork`. Unlike a thread, this task is configured with a callable and returns a subtask that implements the `Supplier` interface. In addition to creating tasks, there are also two `join` methods designed to maintain the currently running processes. Each `join` call is blocking and waits until all tasks started in this scope have completed. The `joinUntil` method can also be used to set a deadline and throw a timeout. The two methods `shutdown` and `close` do not need to be called independently by developers; the Try-With-Resource block coordinates the shutdown. The listing below is an excerpt from a Concurrent RuleEngine that works with a StructuredConcurrency.

```
1 try (var scope = new StructuredTaskScope.ShutdownOnFailure()){
2     Supplier<String> adr = scope.fork(() -> readAddress());
3
4     scope.join().throwIfFailed();
5
6     var address = adr.get();
7 } catch (ExecutionException | InterruptedException e) {
8     throw new RuntimeException(e);
9 }
```

In the resource definition of the Try-With-Resource Block, the scope is initialized as a `ShutdownOnFailure` scope. At this point, `ShutdownOnFailure` means that the first failed subtask leads to a shutdown and thus termination of all tasks. Other common use cases are implemented in the Java API and custom implementations are possible. In the second line, the task of loading an address is transferred to the scope for execution and the supplier for the result is saved. In order to access the loaded value safely, the `join` method in line 3 is used to wait. With the `throwIfFailed` configuration, the scope is instructed to report a failed task as an exception. Once all tasks of the scope have been completed, the `get` method of the supplier interface can be used to access the produced value. The try-with-resource block also enables explicit exception handling in catch blocks. These are the exceptions that are produced by the scope. This enables central error handling, clearly structures the

sequence of tasks and improves overall maintainability. Nothing stands in the way of using virtual threads in our applications to coordinate concurrent work. Scoped Values have been introduced to make efficient use of shared data.

Scoped Values

In many concurrent applications, certain instances or data must be used consistently. One example of this is an SSLContext. Creating an SSLContext instance is time-consuming and resource-intensive. Using a new instance in each of millions of virtual threads is inefficient and pulverizes the lightness gained from virtual threads. To share immutable data between a thread and its child threads, scoped values were designed in Java. The differences to ThreadLocal instances are illustrated in the figure.

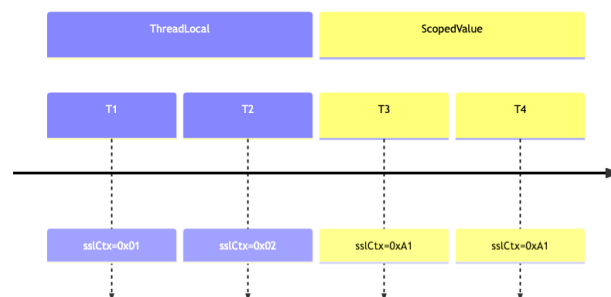


Abb. 5: Different treatment of instances when using ScopedValues and ThreadLocals

A ThreadLocal instance provides a new instance of the stored values for each thread, as can be seen from the different addresses in T1 and T2 for sslCtx. This construct allows behavior and configuration to be shared, but duplication increases memory consumption. With Scope Values, the approach is to provide the same instance in the child threads. The goals were defined as follows:

- **Ease of Use** – the flow of data should be easily traceable
- **Comprehensibility** – the lifetime of the shared data should be evident from the syntactic structure of the code
- **Robustness** – Data shared by a caller should only be retrievable by legitimate callers
- **Performance** – Data should be shared efficiently across numerous threads

The API required to manage and configure a scoped value is kept small, but is all the more powerful.

```
1 final static ScopedValue<SSLContext> VAL = ScopedValue.newInstance();
2
3 ScopedValue.where(VAL, value).run(() -> VAL.get());
```

The ScopedValue#newInstance factory method is used to create a new instance of a ScopedValue. This instance is not the value itself, but a data carrier that provides the actual value on request. The ScopedValue#where configuration method is used to configure which value the ScopedValue should have in this scope. A runnable is passed via the ScopedValue#run method, which is executed in this scope and gets access to the configured values via ScopedValue#get. In the following code example, two scoped values are used to manage an SSLContext and input data as a map.

```

1 ScopedValue<Map<String, String>> INPUTDATA = ScopedValue.newInstance();
2 ScopedValue<SSLContext> SSL_CTX = ScopedValue.newInstance();
3
4 ScopedValue.Carrier executionScope = ScopedValue
5     .where(INPUTDATA, Map.copyOf(data))
6     .where(SSL_CTX, SSLContext.getDefault());
7
8 executionScope.run(() -> {
9     String inputName = INPUTDATA
10         .orElseThrow(IllegalArgumentException::new)
11         .getOrElse("name", "JON");
12     if(SSL_CTX.isBound()) {
13         String lastName = getLastName(SSL_CTX.get());
14         inputName = "%s %s".formatted(inputName, lastName);
15     }
16     IO.println(inputName);
17 });

```

The two `ScopedValue` instances are created in the first two lines. In lines 3 to 5, the applicable values are set for the scope. An immutable copy of the input data and a default `SSLContext` are configured. At this point, a run is not called directly, but the `ScopedValue.Carrier` is saved for later use. This represents a scope in which a combination of `ScopedValues` has defined values. This `ScopedValue.Carrier` is always used if, for example, several tasks are to be executed in one scope. In the last block, such a task is passed for execution. Within the task, the system first checks whether a value has been configured for `INPUTDATA`. If this is not the case, an `IllegalArgumentException` is generated and thrown. A check that the values to be accessed have actually been set is necessary, as there is no guarantee of this within a `ScopedValue.Carrier`. Another way to check whether the value has been set is in the `SSLContext`. Here, the `ScopedValue#isBound` method is called before the access is carried out. The basic semantics are similar to those of an `Optional`. It is important to note that the run call executes the runnable in its own virtual thread.

The differences between `ScopeValue` and the similar `ThreadLocal` are shown in the table below.

SCOPEDVALUE	THREADLOCAL
Immutable Carrier	Supplier
Same instance	Instance per Thread
Rebindable	Mutable Value

Abb. 6: Comparison of `ThreadLocal` and `ScopedValue`

A `ScopeValue` is an unchangeable data transporter. In contrast to a `ThreadLocal`, which is a supplier of a specific value. A `ScopeValue` is always the same instance, whereby the provided value can be reassigned for child threads. Due to their similarity in the application, `ThreadLocals` should be replaced by `ScopedValues` wherever possible to enable the efficient use of virtual threads.

Final Thoughts

In this post, we've looked at Virtual Threads as a lightweight alternative to platform threads. We have seen `Scoped Values` and `Structured Concurrency`. These are optimized for use with Virtual Threads to share data and perform asynchronous operations. The examples shown are from an open source project in which a concurrent rule engine has been implemented.

I would like to give you a few tips on these new topics. Virtual threads should not be used sparingly, they are designed as throwaway instances. By reusing them, optimizations implemented in the JVM are undermined and the desired effect is not

achieved. I therefore recommend that you switch from platform threads to virtual threads wherever possible. In the simplest case, it is just a matter of replacing an `ExecutorService` instance.

During your migration, also pay direct attention to whether you could identify `ThreadLocal` instances that are never changed. These are candidates for a direct migration to `ScopeValues`. If you use IO operations in concurrent code, this is the place for Structured Concurrency. Do you need support in creating a migration plan or do you have further questions about understanding? Then I am happy to help at conferences and via common social media channels.

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

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Navigating the Waves of Concurrency: Exploring Jakarta Concurrency

JakartaConcurrency

Multithreading

OpenLiberty

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

“In a concurrent world, imperative is the wrong default!”

In the vast ocean of software development, concurrency stands as a formidable wave to be mastered. With the ever-increasing demand for high-performance, scalable applications, understanding and effectively implementing concurrency becomes crucial. Among the many tools and frameworks available, Jakarta Concurrency shines as a beacon, offering developers a robust set of tools to navigate the complexities of concurrent programming in Java. In this blog post, we'll explore Jakarta Concurrency through practical code examples, demonstrating its key features and best practices.

Understanding Concurrency

“I’m not saying writing concurrent code is hard, but even the ‘Hello, World!’ program deadlocks sometimes”

Concurrency, simply put, is the ability of a system to execute multiple tasks concurrently. In the realm of software development, this often means having multiple threads of execution running simultaneously. While this can lead to significant performance improvements and better resource utilisation, it also introduces challenges such as race conditions, deadlocks, and thread synchronisation issues.

Enter Jakarta Concurrency

Jakarta Concurrency, formerly known as Java EE Concurrency Utilities, is a powerful framework that provides high-level concurrency abstractions and utilities for Java applications. Part of the larger Jakarta EE ecosystem, Jakarta Concurrency offers developers a rich set of tools to tackle complex concurrency scenarios with ease.

Key Features and Components

Before diving into code examples, let's first ensure we have the necessary dependencies set up. Jakarta Concurrency is part of the Jakarta EE ecosystem, so make sure you have a compatible Jakarta EE runtime or application server installed. Additionally, ensure that you have the Jakarta Concurrency API included in your project dependencies.

```
1 <dependency>
2   <groupId>jakarta.enterprise.concurrent</groupId>
3   <artifactId>jakarta.enterprise.concurrent-api</artifactId>
4   <version>{version}</version>
5   <scope>provided</scope>
6 </dependency>
```

Managed Executors

One of the core components of Jakarta Concurrency is its managed executor service. This allows developers to offload tasks to a pool of worker threads, providing automatic management of thread lifecycle and resource allocation. Managed executors simplify the process of parallelising tasks and help optimize resource utilisation.

Let's see how we can use a managed executor to execute a task asynchronously.

```

1  @Resource
2  private ManagedExecutorService managedExecutorService;
3
4  @GET
5  @Path("managedExecutors")
6  @Produces(MediaType.TEXT_PLAIN)
7  public String getmanagedExecutors() throws InterruptedException,
   ExecutionException {
8      Future future1 = managedExecutorService.submit(() -> {
9          System.out.println("Job 1 running ...");
10         // This takes some while
11         System.out.println("Job 1 finished ...");
12     });
13     Future future2 = managedExecutorService.submit(() -> {
14         System.out.println("Job 2 running ...");
15         // This takes some while
16         System.out.println("Job 2 finished ...");
17     });
18     future1.get();
19     future2.get();
20     System.out.println("Jobs completed");
21     return "Jobs completed";
22 }

```

The output of the above program in console:

```

1  \[INFO] Job 1 running ...
2  \[INFO] Job 2 running ...
3  \[INFO] Job 1 finished ...
4  \[INFO] Job 2 finished ...
5  \[INFO] Jobs completed

```

Context Propagation

Another essential feature of Jakarta Concurrency is its support for context propagation. This allows contextual information, such as security context or transaction context, to be automatically propagated across asynchronous execution boundaries. This ensures consistency and integrity, especially in distributed systems where context management can be challenging.

```

1  @Resource
2  ContextService contextService;
3
4  @Resource
5  ManagedExecutorService managedExecutorService;
6
7  @GET
8  @Path("contextPropagation")
9  @Produces(MediaType.TEXT_PLAIN)
10 public String getContextPropagation()
11     throws InterruptedException, ExecutionException {
12
13     //Execution 1
14     managedExecutorService.execute(() -> {
15         try {
16             System.out.println(new InitialContext()
17                 .lookup("java:comp/env/replySuccess"));
18         } catch (NamingException namingException) {
19             namingException.printStackTrace();
20         }
21     });
22     ExecutorService executor = Executors.newFixedThreadPool(2);
23 }

```

```

24      // Executer without context
25      //Execution 2
26      executor.submit(() -> {
27          try {
28              System.out.println(new InitialContext()
29                  .lookup("java:comp/env/replySuccess"));
30          } catch (NamingException namingException) {
31              System.err.println("NamingException on ExecuterService as no context
32                  available.");
33          }
34      });
35      //Execution 3
36      executor.submit(contextService.contextualRunnable(() -> {
37          try {
38              System.out.println(new InitialContext()
39                  .lookup("java:comp/env/replySuccess"));
40          } catch (NamingException namingException) {
41              namingException.printStackTrace();
42          }
43      }));
44
45      return "Application context propagated with java.util.concurrent.ExecutorService
46          using jakarta.enterprise.concurrent.ContextService.";

```

In this example:

- ManagedExecuterService in Execution 1 is running with the Application Context and will be able to access JNDI lookup.
- ExecuterService in Execution 2 will not have Application Context and will not be able to access JNDI lookup.
- For getting the context, ExecuterService in Execution 3 is paired with the ContextService.

The output of the above program in console:

```

1  \[INFO] Hello from java:comp/env! The document was inserted successfully!
2  \[INFO]
3  \[err] NamingException on ExecuterService as no context available.
4  \[INFO] Hello from java:comp/env! The document was inserted successfully!

```

Managed Threads

With Jakarta Concurrency, developers can leverage managed threads to execute tasks in a controlled and managed environment. This helps mitigate common pitfalls associated with manual thread management, such as resource leaks and inefficient resource utilisation.

Let's dive into an example demonstrating the usage of managed threads:

```

1  @Resource
2  private ManagedThreadFactory managedThreadFactory;
3
4  @GET
5  @Path("managedThreads")
6  @Produces(MediaType.TEXT_PLAIN)
7  public String getManagedThreads()
8      throws InterruptedException, ExecutionException {
9      // Create and start a managed thread
10     Thread thread = managedThreadFactory.newThread(() -> {
11         System.out.println("Managed thread is executing");
12     });
13     thread.start();
14     return "Managed thread created";
15 }

```

Embracing Best Practices

While Jakarta Concurrency offers powerful tools for concurrent programming, it's essential to follow best practices to ensure robust and reliable applications:

1. **Use Managed Executors:** Whenever possible, leverage managed executors for task execution to benefit from automatic thread lifecycle management and resource optimization.
2. **Contextual Awareness:** Ensure proper propagation of contextual information across asynchronous boundaries to maintain consistency and integrity in your applications.
3. **Avoid Shared Mutable State:** Minimize the use of shared mutable state between concurrent tasks to reduce the risk of race conditions and synchronization issues.
4. **Error Handling:** Implement robust error handling and recovery mechanisms to gracefully handle exceptions and failures in concurrent execution paths.

Conclusion

Concurrency is a powerful tool in the arsenal of modern software development, but it comes with its own set of challenges. Jakarta Concurrency provides developers with a comprehensive set of tools and utilities to tackle these challenges effectively. By leveraging managed executors, context propagation, and asynchronous event handling, developers can design high-performance, scalable applications that harness the full potential of concurrency. However, it's crucial to follow best practices and guidelines to ensure the reliability and stability of concurrent systems. With Jakarta Concurrency, developers can navigate the waves of concurrency with confidence, unlocking new possibilities in Java application development.

"Hello, world? Hold on, I'll put you on hold, spawn a few more threads, and get back to you"

Want to try it out? [jakarta-concurrency-demo](#) using [Open Liberty](#).

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



The Role of Quarkus in the Modern Java Ecosystem

Frameworks

Innovation

Quarkus

Author: Martin Stefanko

Principal software engineer at IBM, BrnoJUG leader, author Quarkus in Action, MicroProfile committer, Quarkus, SmallRye, Wildfly, JBoss middleware (RESTEasy, Weld, ...), programming and microservices enthusiast.



LinkedIn

Since its inception in 2019, Quarkus has gradually grown both in user adoption and innovation. The number of its different research and development areas is just enormous. It's one of the most active Java projects you can find right now.

Last year, Quarkus celebrated its 5th birthday and one additional tremendous milestone — 1000 contributors! These are contributors to the main repository only, not considering the additional 170 repositories under the Quarkiverse organization (which we tackle later in this article).

Quarkus pushes boundaries everywhere you can imagine. So, what does this say about Quarkus's contribution to the Java ecosystem?

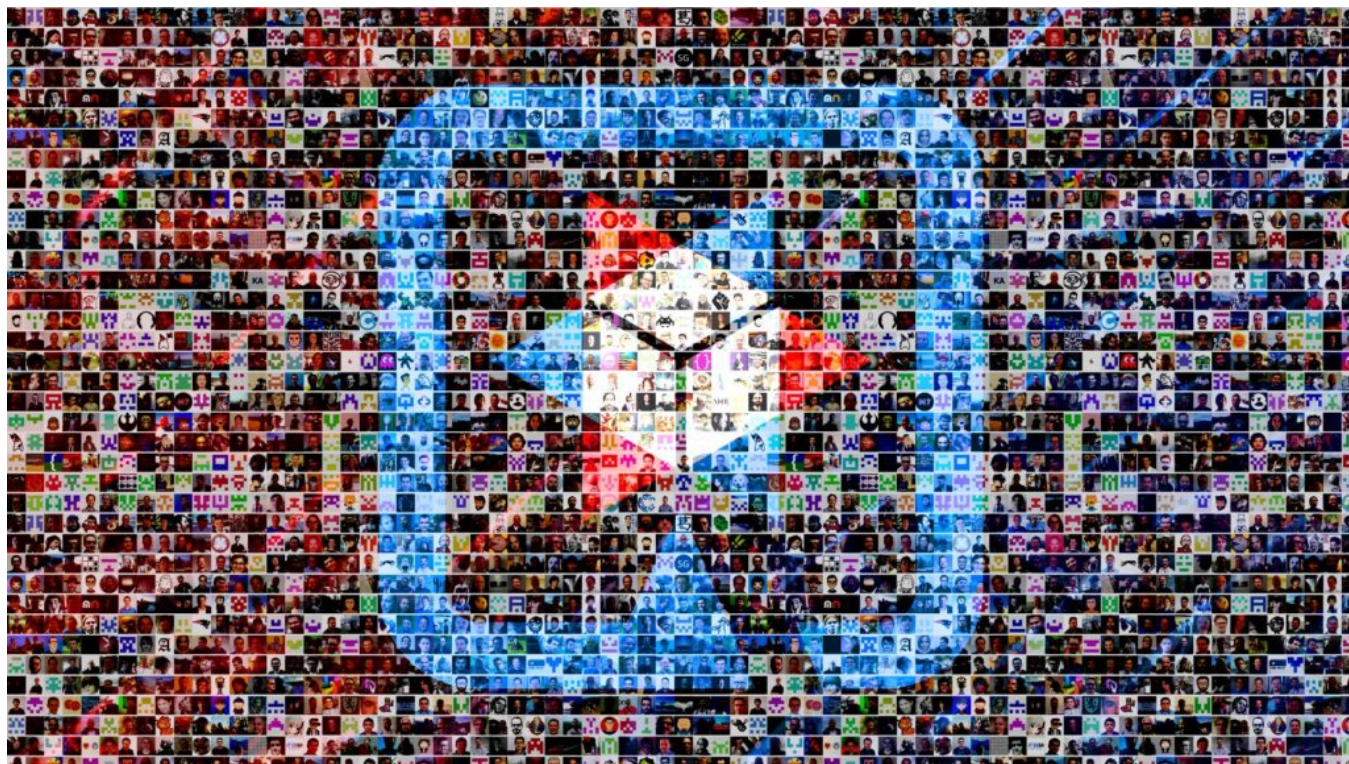


Abb. 7: [Quarkus 1000 contributors](#)

Quarkus to the Moon and Beyond

No one would have guessed 6 years ago what could become of Quarkus when a few very clever minds at Red Hat came together and decided to design a new Java framework. Undoubtedly, no one would imagine it would revolutionize traditional Java development and take off so fast. Quarkus brought a new way of thinking to the engineering workflows — a framework that not only cares about the performance but also puts impressive emphasis on the developer experience.

In my experience, there is no issue proving that Quarkus has something to offer to any audience. High throughput, lower cloud costs, and sustainability are easy to justify, especially when we have so many real users and their applications that validate what Quarkus offers (<https://quarkus.io/userstories/>).

However, I'm also an engineer. I love to code and understand the pains of everyday development tasks. This is why I was blown away when I first opened Quarkus's Dev mode. Before that, I was used to either application server workflow or rebuilding my fat JAR after each change. The Dev mode was so amazing that all applications I wrote from there (even the application server tooling) were powered by Quarkus from then on.

But there was one more thing that astonished me: how fast everything moved. The activity level in Quarkus's GitHub repository was unknown to me before. Pull requests were reviewed and merged so quickly that your PR would be out of sync in hours (even minutes from time to time). An automated bot automatically tagged each issue for the right people to look at. More importantly, the rate of releases was just impressive. There was a new release every two weeks and a new feature release every month. Quarkus is keeping this pace to this day, demonstrating its commitment to both innovation and the community.

For this article, we can split the main Quarkus features that have an influence on its position in the Java ecosystem into four categories:

- Performance characteristics and optimizations
- Dev mode and everything it encloses
- Innovation
- Community engagement and support

Feel free to jump to relevant parts if you are already familiar with Quarkus's basics. However, we need to start from the beginning by providing the complete picture of where Quarkus stands today.

What is Quarkus?

If this is the first time you are reading about Quarkus, I want to start with a brief overview of this framework and its aims. Java has come a long way since it was created 30 years ago. Initially designed for long-running applications that didn't require fast startup times or low memory utilization, Java struggled when we started moving our workloads to the cloud. These exact metrics became a prominent influence on our production costs.

Most of the Java ecosystem wasn't ready for this change, which was one of the main drivers behind creating Quarkus. Quarkus provides a very modern approach to application development while still building on the shoulders of giants — popular and production years-tested technologies like Hibernate, Netty, or Vert.x. This is why its popularity has skyrocketed since its announcement. And it is still continuously growing.

More than a JVM framework for enterprise application development, Quarkus provides a platform for building microservices, serverless, or traditional monolithic applications, as you'd expect. However, it is by no means limited to cloud-native applications, and you can also use Quarkus to build any Java application you can think of: command line (CLI) applications, edge, GitHub bots, GitHub actions, Kubernetes operators, you name it.

Technology Packed Runtime

Quarkus's website defines Quarkus as “a Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards” (<https://quarkus.io>). If we break it down:

Kubernetes native means Quarkus makes sure that user experience in Kubernetes is straightforward. It provides many functionalities that help with Kubernetes deployments and management. Sometimes, you don't even have to write any YAML because Quarkus can generate all descriptors for you.

By being *tailored for OpenJDK HotSpot and GraalVM*, the definition means that Quarkus aims to provide a suitable platform for both the traditional JVM deployments (runnable JAR) and GraalVM (<https://www.graalvm.org>) native binaries. The native compilation with GraalVM takes the performance mechanisms that Quarkus provides to another level. Natively built binaries start in mere milliseconds and rely on a very low memory. Quarkus guarantees that you can freely choose between the two as long as you use the integrations provided by Quarkus (called extensions). If you do so, then a simple flag switch triggers compilation to native binary with GraalVM. Quarkus integrates all relevant configurations for the GraalVM so your application compiles successfully.

Crafted from the best-of-breed Java libraries and standards only says that the technology stack behind Quarkus is not entirely created from scratch. Many of the included technologies have been around for some time (e.g., Vert.x, Jackson, Hibernate). Quarkus provides a unique way of integrating them, which we will look into next. But in the end, you use the same Hibernate as in any other framework, so the learning curve is short.

Build Time/Runtime Split

Quarkus is a very performant framework because it splits all application processing into two phases: build time and runtime. Every framework needs to execute various tasks when your application starts: loading and parsing of the configuration, classpath scanning (annotations), analyzing dependencies, and creating an application model. Traditional frameworks need to execute all these operations when the application starts. Every time it starts. So, with every restart, all of this processing needs to be repeated, even if the application hasn't changed.

Quarkus moves as much framework processing as possible to the build time (when you compile and package your application). This way, the processing results are recorded and reused when the application starts. In other words, they are computed during build time only once and reused at runtime.

Traditional Frameworks

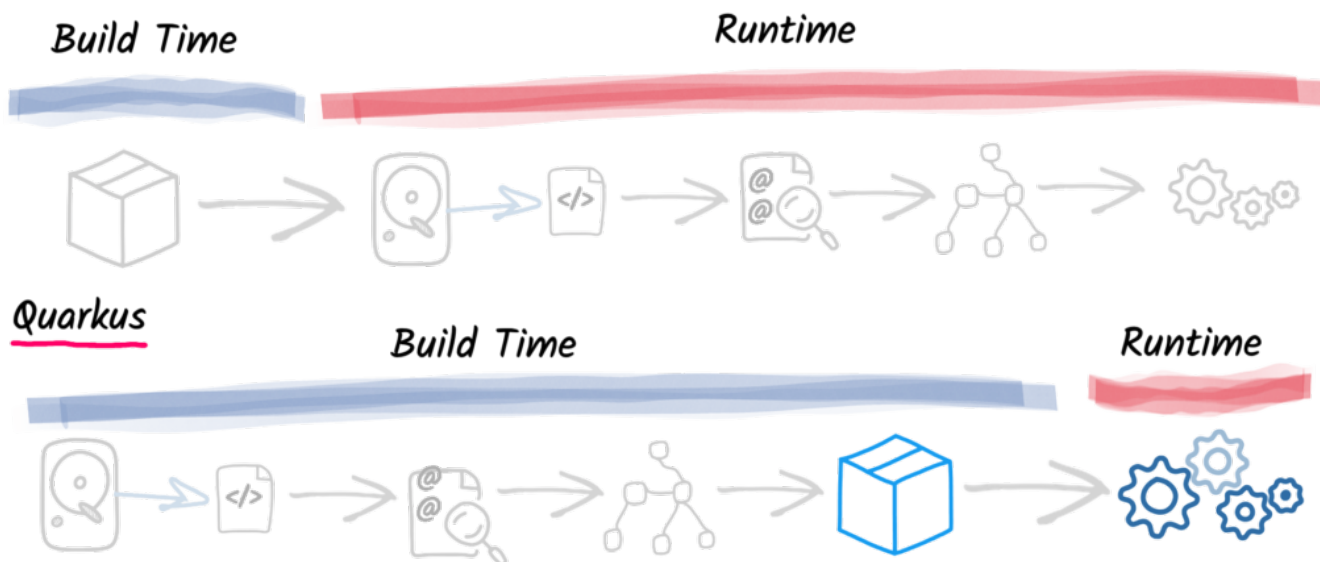


Abb. 8: Quarkus build time/runtime split (<https://quarkus.io/performance/>)

Extensions – Integrating with Quarkus

All the processing described above is further exposed for integrations that are called extensions. Each extension integrates some library or functionality so the extension developer can choose which parts of the library processing can execute during build time or what needs to run at runtime. For instance, imagine the path on which your OpenAPI document is exposed. It doesn't change that often, so recording it only once during build time is sufficient. On the other hand, something like the port on which our application runs would be unreasonable to fix at build time since it might run on different ports in different environments, so it needs to be overridable at runtime.

Additionally, extensions can provide a helpful configuration for seamless GraalVM native builds. The extension developers know the integrated library, which means they understand what kind of code their library user needs to write. Providing any of the usually required configurations for GraalVM build (e.g., reflectively accessed classes) is thus done in the extension itself, so the users don't need to provide it manually.

There are currently more than 750 extensions available at <https://quarkus.io/extensions/>.

Quarkiverse

Many library developers see the value that Quarkus can bring to their users and want to provide an extension for them. Because the number of extensions grew so quickly that Quarkus's main repository [github](#) started to get too big to work with, we created the Quarkiverse (Quarkus + Universe). Quarkiverse [github](#) is a GitHub organization that encapsulates community Quarkus extensions that we couldn't fit into the main repository. The community contributors maintain individual repositories but are fully integrated into the Quarkus ecosystem. The repository maintainers can release their extensions as they need, and Quarkus users can pick them for their application in precisely the same way as with the core extensions. Creating a Quarkiverse extension is easy; just follow the [onboarding process](#).

The features described in this section are not all that Quarkus and its extensions can do. One of the best features Quarkus provides is encapsulated in its development experience called the Dev mode. Each extension can provide its own "magic" that makes the work with the integrated library easier or just plainly more fun. Let's look into what we can do in Quarkus Dev mode next.

Developer-Focused Framework

Whether or not any tech is popular is always about the technology. Ask yourself what is more important to you when choosing your framework. Is it performance? Or the tooling support? Quarkus bet that it is the overall development experience. If you think about it, so many frameworks and even some programming languages have been created solely to improve the development experience.

Dev Mode

While performance characteristics are important, they are primarily relevant in production environments where we must restrict utilized resources. However, developers also want technology that helps them be productive and focused on developing their applications. Technology that understands their needs and provides a simple way to solve their problems. Quarkus invests heavily in the developer experience (which pays back).

The single most important productivity tool in Quarkus is its Dev mode. Dev mode is a continuous run of your Quarkus application in which Quarkus embraces the REPL (read-eval-print loop) concept. By simply changing our source code, Quarkus automatically recompiles and reruns our application, showing the result of our work in mere milliseconds. But it doesn't stop there.

The Dev mode can start with, for instance, Quarkus CLI [cli-tooling](#) like this:

```

1 # or ./mvnw quarkus:dev or ./gradlew quarkusDev
2 $ quarkus dev
3 ...
4
5 Listening for transport dt_socket at address: 5005
6
7 --/ _ _ \ / / / _ | / _ \ / / / / / _ /
8 - / / / / / / _ | / , _ / , < / / / \ \
9 --\ _ _ \ _ _ / / | / / | _ / / | _ \ _ _ /
10 2025-02-28 14:33:50,534 INFO [io.quarkus] (Quarkus Main Thread) quarkus-app 1.0.0-
    SNAPSHOT on JVM (powered by Quarkus 3.19.1) started in 1.356s. Listening on: http://
    localhost:8080
11
12 2025-02-28 14:33:50,538 INFO [io.quarkus] (Quarkus Main Thread) Profile dev activated.
    Live Coding activated.
13 2025-02-28 14:33:50,538 INFO [io.quarkus] (Quarkus Main Thread) Installed features: [cdi
    , rest, smallrye-context-propagation, vertx]
14
15 --
16 Tests paused
17 Press [e] to edit command line args (currently ''), [r] to resume testing, [o] Toggle
    test output, [:] for the terminal, [h] for more options>

```

Notice that it automatically opens a debug port for connection if needed. However, you will find that using the Dev mode's features can often be a substitute for using the debugger. This command starts an interactive terminal; the commands are listed at the bottom of the screen. Try pressing **h** while the Dev mode's terminal is in focus:

```

1 The following commands are available:
2
3 == Continuous Testing
4
5 [r] - Resume testing
6 [o] - Toggle test output (disabled)
7
8 == Exceptions
9
10 [x] - Open last exception (or project) in IDE (none)
11
12 == HTTP
13
14 [w] - Open the application in a browser

```

```

15 [d] - Open the Dev UI in a browser
16
17 == System
18
19 [s] - Force restart
20 [e] - Edits the command line parameters and restarts ()
21 [i] - Toggle instrumentation based reload (disabled)
22 [l] - Toggle live reload (enabled)
23 [j] - Toggle log levels (INFO)
24 [h] - Show this help
25 [.] - Enter terminal mode
26 [q] - Quit the application

```

As you can see, Dev mode provides many functionalities. For instance, the instrumentation-based reload (**i**) changes the quarkus reload in a way that if you only change the bodies of methods, Quarkus only dynamically replaces the bytecode in your running JVM without the need to restart the application:

```

1 2025-02-28 14:40:43,865 INFO [io.qua.dep.dev.RuntimeUpdatesProcessor] (vert.x-worker-
  thread-1) Application restart not required, replacing classes via instrumentation
2 2025-02-28 14:40:43,879 INFO [io.qua.dep.dev.RuntimeUpdatesProcessor] (vert.x-worker-
  thread-1) Live reload performed via instrumentation, no restart needed, total time:
  0.046s

```

Of course, if you do something that the bytecode replacement cannot handle, the restart still takes place. Take some time to experiment with these options, it will surely pay out.

Quarkus's Dev mode encompasses many other functionalities that we utilize in our everyday tasks. Features like continuous testing, Dev UI, or Dev Services make development such an enthusiastic experience that it's not surprising people like to use Quarkus. Let's now go over these features and explain what they are about.

Continuous Testing

If you press **r** in your Dev mode terminal, Quarkus starts the continuous testing, the automatic running of your unit tests in the background, when the Dev mode reloads with your changes. At the bottom of the screen, you can see:

```

1 --
2 All 1 test is passing (0 skipped), 1 test was run in 261ms. Tests completed at 14:46:02
  due to changes to GreetingResource.class.

```

And if you change your application (so it will break the test):

```

1 2025-02-28 14:50:09,554 ERROR [io.qua.test] (Test runner thread) Test
  GreetingResourceTest#testHelloEndpoint() failed
2 : java.lang.AssertionError: 1 expectation failed.
3 Response body doesn't match expectation.
4 Expected: is "Hello from Quarkus REST"
5 Actual: Hello from Quarkus article
6
7 ...
8
9 1 test failed (0 passing, 0 skipped), 1 test was run in 234ms. Tests completed at
  14:50:09 due to changes to GreetingResource.class.

```

Notice that the Dev mode detects the file saving in this case, and the test is automatically rerun in the background. Continuous testing is a potent tool since you can see the effects of your changes directly while you're typing them. I usually have my IDE on one screen and the Dev mode running on the one next to it. I type my changes in the IDE and only pay attention to the green color changing to red in the Dev mode terminal to notify me that my changes are breaking some tests. It is also a great tool to "force" you to do more Test-Driven Development.

If you think Quarkus always reruns your entire test suite, you don't need to worry about it. Quarkus tries to deduct which classes you are changing and only runs the tests that are relevant to your changes.

Dev UI

Dev UI provides a graphical interface to your Dev mode and displays valuable information about your application. It also allows you to execute some operations (e.g., starting continuous testing).

You can open Dev UI on <http://localhost:8080/q/dev-ui> (or a different port if you change it) or by pressing `d` in the Dev mode terminal:

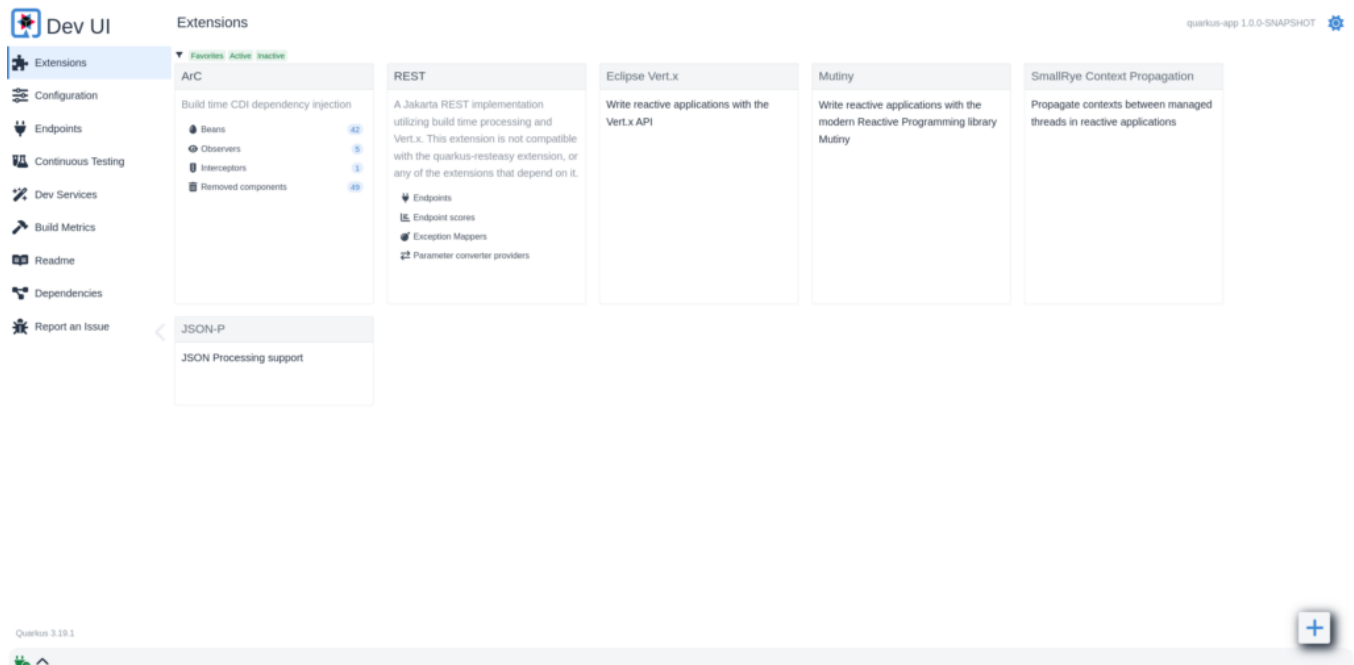


Abb. 9: Dev UI is available on <http://localhost:8080/q/dev-ui> when Quarkus runs in Dev mode

The Dev UI landing page called “Extensions” provides an overview of the extensions installed in the current application. Each extension gets a card that it can enhance with various functionality (e.g., list CDI beans, view OpenAPI document, or start GraphQL). Each extension can freely include whatever it needs.

The second tab takes you to “Configuration,” which lists all available configuration options in the current application. It changes if you add or remove extensions. The lock symbol at the beginning of the line means that the configuration property is fixed at build time (this doesn’t matter in Dev mode, but it will when you compile the application). You can also save the configuration directly in this menu, it will propagate to your configuration file.

We can’t go over everything here, but you can also find tabs for the continuous testing or Dev Services that we cover next. Again, take some time to look around.

Dev Services

Dev Services represent an abstraction of remote dependent services, whether they are databases (Postgres, MongoDB), messaging providers (Kafka, RabbitMQ), security (Keycloak), etc. The Dev services are automatically started by the Quarkus’s Dev and Test modes. So, you don’t have to mix the provider instances for development and testing, which is extremely useful in combination with continuous testing. It essentially boils down to starting a particular OCI (Docker or Podman) container for the specific provider (utilizing Testcontainers) and injecting the connection configuration, but it doesn’t have to. Some Dev services (e.g., Derby or H2) start in process. You can find the list of all extensions that support Dev services at <https://quarkus.io/guides/dev-services>.

You can turn off most of the Dev services with configuration. Either with the explicit disabling property (e.g., `quarkus.keycloak.devservices.enabled`) or with an implicit definition of the uniquely defining instance of the provider. For instance, if you define the connection URL to a database (`quarkus.datasource.jdbc.url`), Quarkus will use your configured database and won’t start a new container.

If your Dev mode starts a Dev Service that can be utilized by multiple Quarkus instances (e.g., Keycloak, Kafka, RabbitMQ), the Dev Service is, by default, shared among all applications that need it. Of course, that also means that you need to track which Dev mode “owns” a particular Dev Service because if you close it, all other Dev modes will likely break. But that’s a small price to pay. Quarkus understands that you want to use technology (because you added an extension for it), so it will ensure your experience is as seamless as possible. You don’t need to learn how to run everything on your machine. If you have Docker (or Podman) installed, Quarkus takes care of everything for you.

The Power of Quarkus’s Dev Mode

To give you an idea of the typical development workflow, I want to describe the car rental application we developed in our latest Quarkus book called “Quarkus in Action,” which was published in January 2025 (there is a link to a free e-book at the end of the article). The car rental application consists of five Quarkus services that communicate together via various protocols and messaging technologies, including Apache Kafka, RabbitMQ, GraphQL, gRPC, and REST:

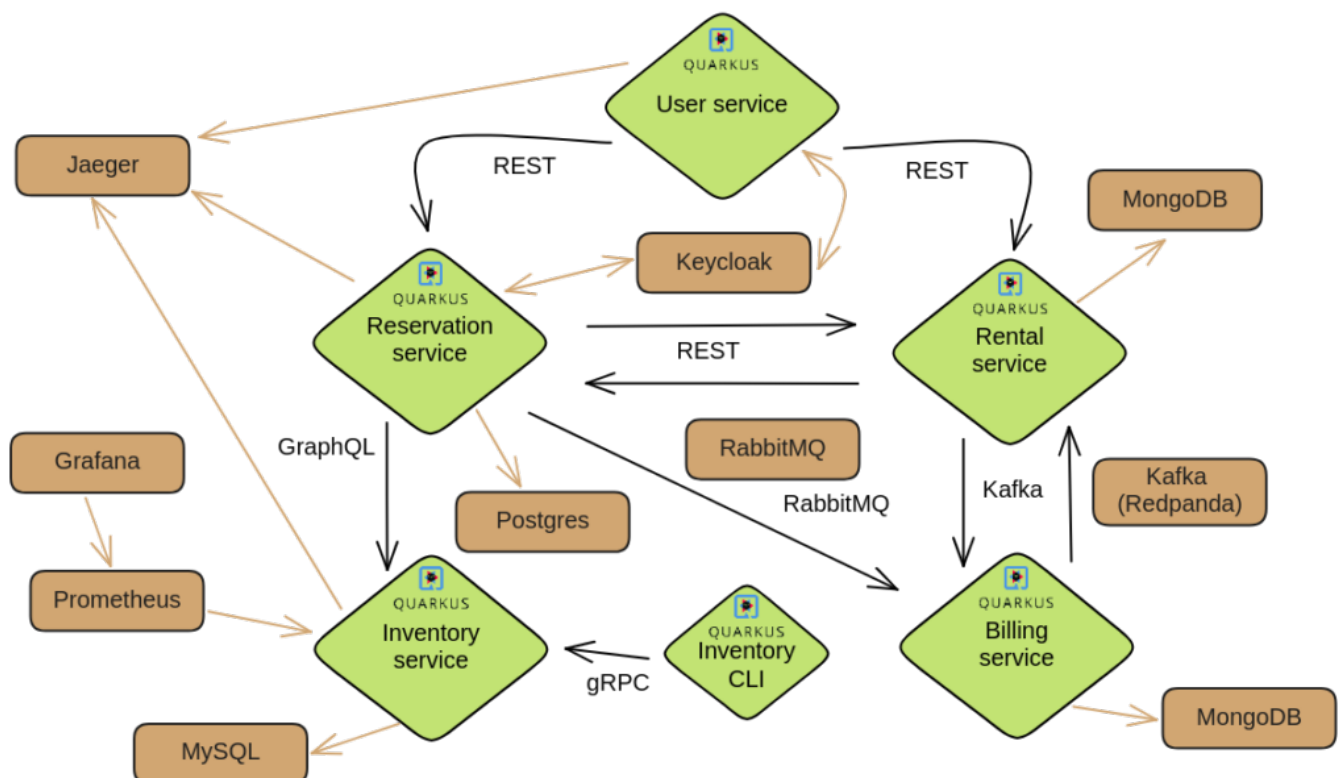


Abb. 10: The architecture of the car rental application developed in the Quarkus in Action book

In the architecture diagram, the Quarkus services represent user-developed services and are marked with the Quarkus logo. The Inventory CLI is an example command line application that manages the car fleet. You can find the code at <https://github.com/xstefank/quarkus-in-action>. Suppose we start the remaining business services in Dev mode (they all run on different ports). In that case, all remaining services start automatically as Dev Services (disclaimer: in the book, we use an older Quarkus version that doesn’t have the LGTM Dev service — <https://quarkus.io/guides/observability-devservices-lgtm>). Together, 10 containers start automatically because we run all Quarkus applications in Dev modes, and everything is correctly connected. We, as users, don’t need to supply any configuration for these services manually, but of course, it’s possible to override the defaults if needed.

Innovation

Quarkus is one of the leading platforms for new Java investigations and innovations. If you take any of the latest new developments (virtual threads, HTTP 3, AI), Quarkus is always there to define how Java evolves. We don’t have space here to go over all research areas, but let’s take an example in the latest popular technology: Large Language Models (LLMs).

AI and LangChain4j

The leading Java library for any AI-related work is called LangChain4j (<https://github.com/langchain4j/langchain4j>). Quarkus has top-level support for it in various extensions. (<https://quarkus.io/extensions/?search-regex=langchain4j>). Additionally, Quarkus core engineers are also the primary maintainers of the upstream library. Quarkus is thus the defining force behind the Java developers who use AI in their applications.

If you are interested, LangChain4j extensions are also great examples of what Quarkus extension provides regarding library integration. You can see how it transforms the programmatic LangChain4j APIs to the annotation-based model that is more common in enterprise application development. However, if you prefer to use LangChain4j directly, you are free to do so.

Model Context Protocol

One of the latest innovation areas in AI integration is the Model Context Protocol (MCP, <https://modelcontextprotocol.io>). MCP is a new protocol that aims to define how applications provide context to LLMs. The context represents tools (operations that LLM can execute), resources (file contents, URLs, etc.), and pre-defined prompts.

Quarkus LangChain4j was one of the first to define a Java API for the MCP servers and clients. The defined programming model aligns with the standard LangChain4j integration in Quarkus. Plugging in MCP thus seems extremely familiar, even if the technology just came out.

Recently, Quarkus also introduced the Quarkus MCP servers (<https://github.com/quarkiverse/quarkus-mcp-servers>), which is the first project that groups Java-based MCP servers. With all of this work, you can see how Quarkus demonstrates that Java is still a modern and powerful language, even for the latest technological trends.

Community Engagement

Quarkus quickly became one of the most popular JVM projects on GitHub. With over 1000 contributors, it already has more than 50,000 commits. And we are only talking about the main [quarkusio/quarkus](https://github.com/quarkusio/quarkus) repository, not including any of the 170 Quarkiverse repositories, which would add even more activity. As you can see in the following image, the commit frequency has been steady throughout the years:

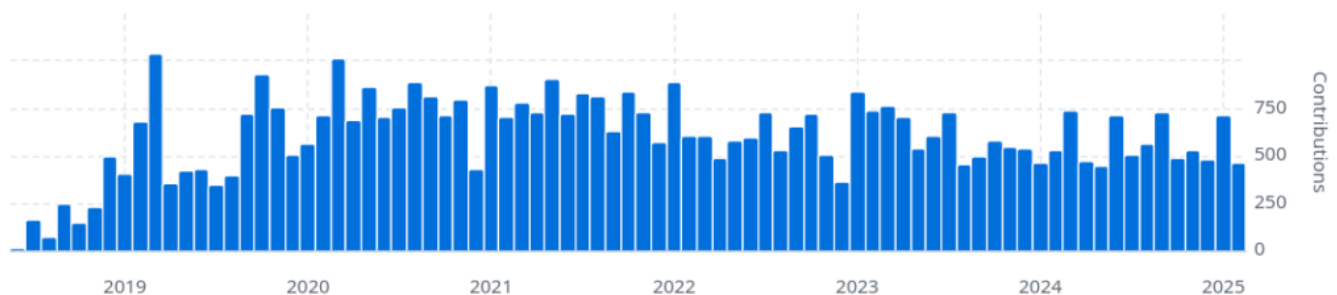


Abb. 11: quarkusio/quarkus repository commit frequency from GitHub

Similarly, the release frequency follows the same line. Quarkus releases a new feature (minor) release every month with a few bug fixes (micro) releases in between.

```
1 $ git for-each-ref --sort=-creatordate \
2   --format '%(refname) %(creatordate)' --count=10 refs/tags
3 refs/tags/3.15.3.1 Thu Feb 27 09:50:51 2025 +0000
4 refs/tags/3.8.6.1 Thu Feb 27 09:49:23 2025 +0000
5 refs/tags/3.19.1 Wed Feb 26 10:26:24 2025 +0000
6 refs/tags/3.18.4 Wed Feb 19 10:50:51 2025 +0000
7 refs/tags/3.19.0 Wed Feb 19 10:49:52 2025 +0000
8 refs/tags/3.19.0.CR1 Wed Feb 12 09:57:58 2025 +0000
9 refs/tags/3.18.3 Wed Feb 12 09:53:22 2025 +0000
10 refs/tags/3.18.2 Wed Feb 5 09:39:17 2025 +0000
```

With such a frequent release cycle, you might think it could be hard to keep up with upgrades; this is why Quarkus provides two ways in which you can consume releases: latest/greatest and Long-Term Support (LTS) releases. Both have their relevant user bases depending on preference and project use cases.

Users can easily upgrade their Quarkus applications with the built-in Quarkus update mechanism. You can run one of the following commands to update your Quarkus project to the latest available version:

```
1 # CLI
2 quarkus update
3
4 # Maven
5 ./mvnw quarkus:update
6
7 # Gradle
8 ./gradlew quarkusUpdate
```

If you want to stick to a particular stream (e.g., LTS), you can add the `--stream` (CLI and Gradle) or `-Dstream` flag (Maven).

For instance, if your Quarkus project is on 3.15.1, `quarkus update` will bump it to 3.19.1 (latest release as of the time of this writing), and `quarkus update --stream=3.15` will bump it to 3.15.3.1 (latest 3.15.x LTS release).

Quarkus LTS

Since not all users want to move as fast as Quarkus offers, Quarkus introduced the concept of long-term releases (LTS), which are streams that the Quarkus community keeps supported for more extended periods (usually one year). You can check which releases are LTS and the proposed release dates at <https://github.com/quarkusio/quarkus/wiki/Release-Planning>. LTS releases are carefully monitored, and only the selected fixes are being backported from the main release branch. So, if you prefer stability to features, Quarkus LTS is a perfect fit.

A new LTS version is released every 6 months, and there are two micro/bug-fix releases in between every 2 months. For more information, check [this blog post](#).

Quarkus LTS is also supported as an enterprise subscription from Red Hat in a product called Red Hat build of Quarkus (RHBQ).

Contributing to Quarkus

The real value of this community is in its approach to contributions. With such a frequency of changes, it's very valuable that every PR gets reviewed within hours. You don't need to wait for days to get things done. And if you have any issues, you just need to ask. Quarkus engineers are very willing to help you, no matter your problem. All of GitHub's issues, pull requests, discussions, mailing lists, stack overflow, social media, and Zulip (discussion forum) are actively monitored by Quarkus engineers, so you will have no problem finding help if you need it.

Each contribution matters. It's not only about the code. For instance, if you want to write the tests or the documentation (it's called guides in Quarkus) to ease yourself into contributing to Quarkus, that's awesome!!! It's also a fantastic way to get to know the framework in the fastest possible way.

Where Do I Start If I Want to Contribute to Quarkus?

There are several good ways to start contributing to Quarkus. One way would be the `good first issue` the main GitHub repository issues — [GitHub Issues](#). The Quarkus team is actively trying to add new issues to this label.

Another way to contribute if you don't know where to start is working groups — quarkus.io. Working groups provide a way to organize work around specific topics. They include research, coding, testing, documentation, and anything else that might come to mind. They aim to group people with similar interests who want to move a specific Quarkus area forward. There are no requirements. Everyone contributes what they want and when they want. Additionally, this can be an excellent way for newcomers to get in touch with the core Quarkus engineers right from the start. They will be more than happy to start you up on the topic, even if you don't have much experience. Working groups can be an excellent way for you to get involved in a topic that is interesting to you.

Summary

Quarkus is undoubtedly one of the most interesting projects in the Java ecosystem. It offers many features that are not only relevant to running Java in production but also to how enjoyable the experience of Java development can be. It's heavily interested in simplifying end application work, proving that Java continues to be a compelling language for modern application development.

Its approach to build time optimizations provides a substantial performance boost for the main application metrics, including the utilized memory and the startup times prominent in cloud environments. Additionally, the Dev mode brings so much enjoyment to day-to-day development that Java engineers find it hard to go back to traditional Java workflows after they try it for the first time. Features like continuous testing, Dev UI, and Dev Services provide such individual productivity boosts that it's not surprising.

Since everything Quarkus is open source, contributing is simple. The community is extremely helpful, so you will always be able to find help. This is proved by more than 1000 contributors in just the first five years of Quarkus.

As promised, if you want to learn more about Quarkus, you can download our latest book, "Quarkus in Action," for free from the Red Hat Developer — redhat.com.

So, what is Quarkus's role in the modern Java ecosystem? It's one of the leading research, development, and innovation drivers that enable Java to remain one of the most popular languages in the world. Quarkus's role is to push boundaries, and it's not stopping any time soon.

› **Back to the Table of Contents** ‹



The Simplest Way to Build Resilient Applications

CloudNative

JVM

Microservices

Resilient Applications

Serverless

Author: Giselle van Dongen

Giselle is a Developer Advocate at Restate, where she helps its users with understanding how Restate simplifies the development of resilient and scalable distributed apps. Before that, she worked in the field of big data analytics and distributed stream processing, and obtained a PhD on this topic at Ghent University.



LinkedIn

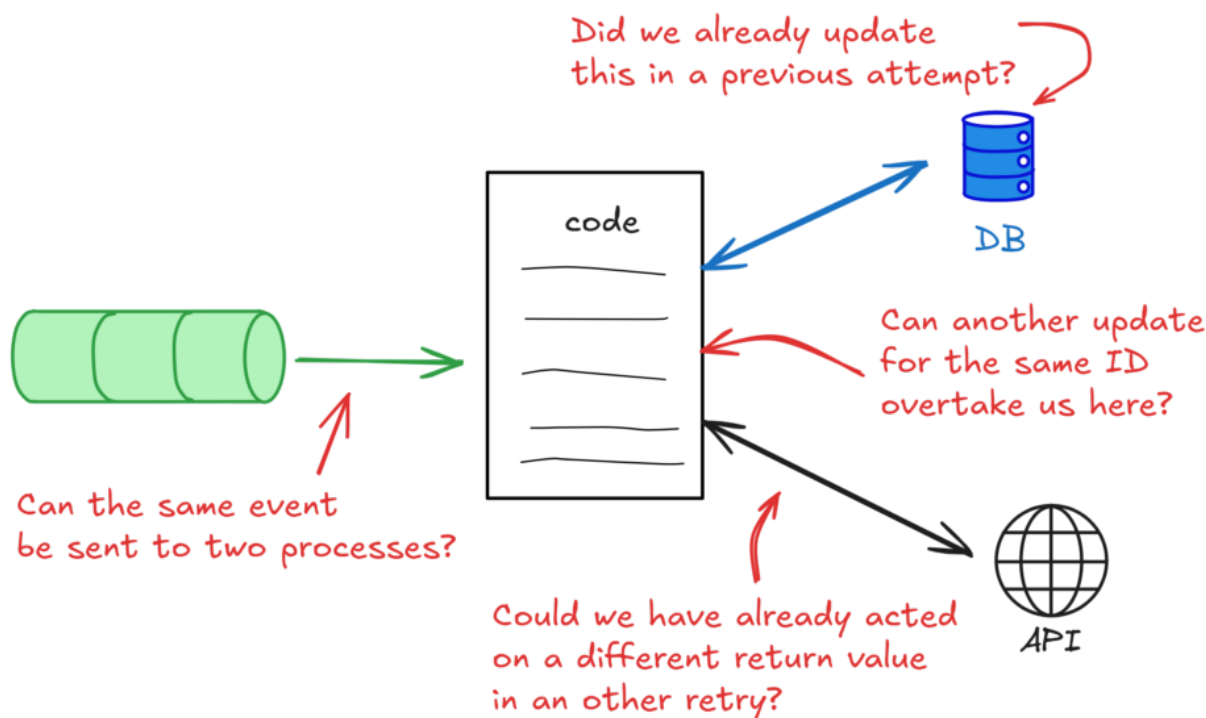
Restate is an open-source system that makes it easy to write resilient, consistent and scalable applications. Restate lets you write mission-critical applications, like payment workflows, user management, and AI agents, without worrying about resiliency.

[Restate](#) is an open-source system that makes it easy to write resilient, consistent and scalable applications. Restate lets you write mission-critical applications, like payment workflows, user management, and AI agents, without worrying about resiliency. All the hard stuff is abstracted away into a set of durable building blocks that you can use to write your applications.

You write regular functions and services and Restate makes them resilient, consistent and scalable

Why Resiliency Matters

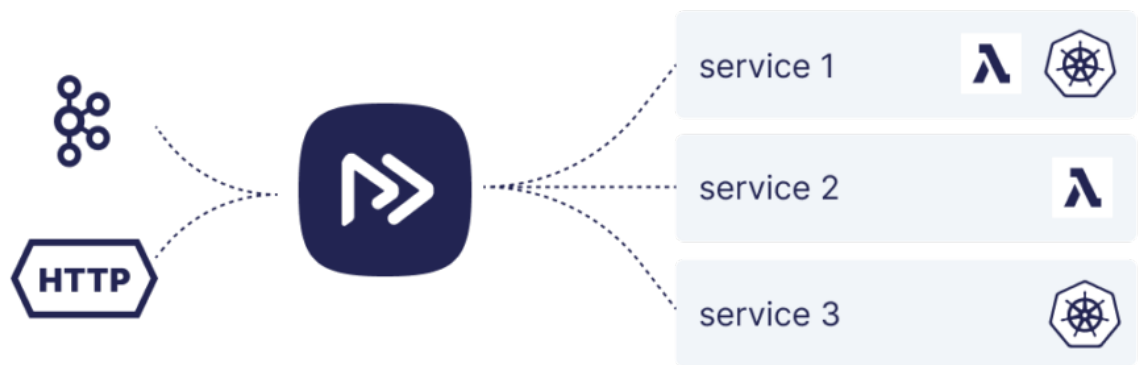
It's still too hard to write applications that are fully resilient. Wouldn't it be great if you could stop worrying about all possible failure scenarios and race conditions in your applications such as charging customers twice, users withdrawing more cash than they have, and selling a single ticket multiple times? Application code is flooded with retry, recovery, and coordination logic across message queues, key-value stores, workflows, and cron jobs. This turns our applications into distributed systems that are hard to reason about and hard to debug.



A computer lets you make more mistakes faster than any invention in human history, with the possible exceptions of handguns and tequila. – Mitch Ratcliffe

Resilient Code Execution and Communication

Restate solves this by capturing all application activity—RPC calls, timers, promises, API calls—into [a single, reliable log](#). In case of failures, Restate retries the execution and recovers the process and all previous state. Restate automatically mitigates against a large class of infrastructure failures, race conditions, timeouts, and other inconsistencies.



Your code runs as Java applications on your existing infrastructure – whether that’s FaaS, Kubernetes, servers, or containers. The Restate SDK is embedded as a dependency.

The Restate Server which stores the execution log, sits in front of your services like a reverse proxy or Kafka broker. It’s a single binary (no extra dependencies like databases) that’s easy to deploy and operate.

Persist Execution Progress

The Restate SDKs provide programming constructs with built-in retries and recovery. You can work with objects, functions, and futures as if failures don’t exist – Restate can recover them anytime, anywhere.

These building blocks are made available via the Restate Context (`ctx` in the example). Whenever a line of code gets executed that uses the context, an event gets sent to Restate to persist that this took place. When there is a failure halfway through, Restate triggers a retry, and recovers the previous results, instead of re-executing the actions.

Imagine we want to set up a list of subscriptions for a user, and create a recurring payment. With Restate this looks as follows:

```

1  @Service
2  public class SubscriptionService {
3
4      @Handler
5      public void add(Context ctx, SubscriptionRequest req) {
6          // create an ID that's stable across retries
7          var paymentId = ctx.random().nextUUID().toString();
8
9          // use the ID to create exactly one recurring payment
10         var payRef =
11             ctx.run(() -> createRecurringPayment(req.creditCard(), paymentId));
12
13         // Restate ensures all subscriptions will be created successfully
14         for (String subscription : req.subscriptions()) {
15             ctx.run(() -> createSubscription(req.userId(), subscription, payRef));
16         }
17     }
18 }

```

Resilient RPC and Messaging

Restate sits like a reverse-proxy or message broker in front of your services, proxying calls and ensuring they’re delivered and processed. It supports various communication patterns:

- [Request-response calls](#)
- [One-way messages](#)
- [Scheduled tasks](#)
- [Awakeables](#) that let you model slow async interactions like human-in-the-loop

Calls can be made over HTTP, Kafka, or with the SDK-generated clients. All calls are resilient out of the box without separate message queues, custom retry logic, or schedulers.

Idempotency for Free

Restate handles deduplication of retried requests, by letting you add an [idempotency](#) token to your requests. Each request runs to completion exactly once.

```
1 Client restateClient = Client.connect(RESTATE_URL);
2 SubscriptionServiceClient.fromClient(restateClient)
3   .send()
4   .add(
5       subscriptionRequest,
6       CallRequestOptions.DEFAULT.withIdempotency(requestId));
```

Implementing Sagas and Distributed Transactions

Restate guarantees code runs to completion, enabling you to implement resilient [sagas](#) in a try-catch block. Restate handles retries and recovery.

1. **Track compensations:** As functions execute, undo operations are tracked in a list.
2. **Guaranteed rollback:** When unrecoverable errors occur, Restate ensures all compensation actions run to completion.

```
1 @Service
2 public class SubscriptionSaga {
3
4     @Handler
5     public void add(Context ctx, SubscriptionRequest req) {
6
7         List<Runnable> compensations = new ArrayList<>();
8
9         try {
10             var paymentId = ctx.random().nextUUID().toString();
11
12             compensations.add(() -> removeRecurringPayment(paymentId));
13             ctx.run(STRING, () -> createRecurringPayment(req.creditCard(), paymentId));
14
15             for (String subscription : req.subscriptions()) {
16                 compensations.add(() -> removeSubscription(req.userId(), subscription));
17                 ctx.run(() -> createSubscription(req.userId(), subscription));
18             }
19         } catch (TerminalException e) {
20             for (Runnable compensation : compensations) {
21                 ctx.run(() -> compensation.run());
22             }
23             throw e;
24         }
25     }
26 }
```

This lets you implement distributed transactions across multiple services and systems in simple sequential code, without having to resort to resiliency patterns like event sourcing.

Stateful Entities and State Machines

Restate has an embedded K/V store, which lets you implement stateful services by storing state directly in Restate.

- **Consistent state:** State changes become just another type of execution progress. Restate makes sure that your state is always consistent with where your execution is at. You can implement [durable state machines](#), [stateful AI agents](#), or actors, without extra infrastructure.
- **Scalability, consistency, concurrency:** Restate guards [consistency](#) by ensuring only one handler writes to a single state value at a time. You can scale your services without worrying about multiple writers, lost updates, race conditions, or inconsistencies.

- **Stateful serverless:** Restate attaches the service's K/V state to each request, allowing your handlers to work with local state. Your stateful functions can be deployed to serverless platforms like AWS Lambda and scale up and down as needed.

```

1 @VirtualObject
2 public class SubscriptionObject {
3
4     StateKey<String> SUBSCRIPTION = StateKey.of("subscription", STRING);
5
6     @Handler
7     public void add(ObjectContext ctx, ObjectUtils.SubscriptionRequest req) {
8
9         ctx.set(SUBSCRIPTION, "awaiting_payment");
10        var paymentId = ctx.random().nextUUID().toString();
11        boolean success =
12            ctx.run(
13                BOOLEAN,
14                () -> createRecurringPayment(req.creditCard(), paymentId));
15
16        if (!success) {
17            ctx.set(SUBSCRIPTION, "payment_failed");
18            return;
19        }
20
21        ctx.set(SUBSCRIPTION, "creating_subscription");
22        ctx.run(() -> createSubscription(req.userId(), req.subscription()));
23
24        ctx.set(SUBSCRIPTION, "created");
25    }
26 }

```

What Can You Build?

Restate gives you everything you need to build resilient applications for a [large variety of use cases](#):

- Saga-based distributed transactions
- Human-in-the-loop workflows
- Durable state machines (for tracking factory equipment)
- Consistent database interactions
- Idempotent async payment processing
- Stateful AI agents

Detailed Observability

While traditional observability tools stop at the request boundary, Restate is aware of the code execution itself. It knows how services communicate and what the application state is. This gives it a unique position for observability.

Restate makes all the information it stores available to you via a [UI](#), CLI, and built-in tracing capabilities. You can answer questions like

- “Where in the call chain and at what line of code did my handler get stuck?”
- “What step is failing with what error?”
- “Which code path did my handler follow?”
- “What other handlers are experiencing the same issues?”
- “Are there still ongoing executions on this deployment or can I remove it safely?”

The screenshot displays the Restate server v1.2.2 interface. At the top, there are tabs for 'Overview', 'Invocations', and 'State'. The 'Invocations' tab is active, showing a table of invocations. The table has columns: Id, Created at, Target, Status, Invoked by, and Journal. The 'Status' column has a dropdown menu. The 'Journal' column has a link to view the journal for each invocation. A modal window is open, showing the 'Journal' for a specific invocation. The journal contains a list of steps, including 'chat', 'get current_agent_name', 'set current_agent_name', 'get customer_context', 'wait run', 'set customer_context', 'get input_items', 'wait run', 'set input_items', 'set current_agent_name', and 'wakes up from sleep'. At the bottom of the interface, there is a search bar with the text 'Status is not' and a 'Query' button.

Id	Created at	Target	Status	Invoked by	Journal
1.	2m 12s ago	CustomerServiceSession / chat	Suspended for 1m 6s	Ingress	14 entries
1.	2m 58s ago	CustomerServiceSession / chat	Succeeded		
1.	3m 40s ago	CustomerServiceSession / chat	Succeeded		
1.	3m 52s ago	InvoiceSender / send_invoice	Scheduled		
1.	4m 1s ago	InvoiceSender / send_invoice	Scheduled		
1.	6m 43s ago	InvoiceSender / send_invoice	Scheduled		

```

# chat (parameters, headers)
get current_agent_name → void
set current_agent_name = "Triage Agent"
get customer_context → void
await run("get customer context") → [{"passenger_name": "Alice", "confirmati...
set customer_context = [{"passenger_name": "Alice", "confirmation_number": "12...
get input_items → void
await run("LLM call - Triage Agent") → [{"output": [{"id": "fc.67d442d86e93...
await run("LLM call - FAQ Agent") → [{"output": [{"id": "fc.67d442d7eccc81...
await run("LLM call - FAQ Agent") → [{"output": [{"id": "fc.67d442d8a78c81...
await run("LLM call - Triage Agent") → [{"output": [{"id": "esp.67d442d801...
set input_items = [{"content": "When is my flight?", "role": "user"}, {"id...
set current_agent_name = "Triage Agent"
wakes up from sleep in 14m 33s.
  
```

Try It Out!

Restate is open source, so go to the [documentation](https://docs.restate.dev) (docs.restate.dev) to get your first demo app running. If you have any questions, join Restate's [Discord](#) or [Slack](#) community.

Restate is also available as a [fully managed cloud service](#), if all you want is to use it and let us operate it.

› [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



Building Local LLM AI-Powered Applications with Quarkus, Ollama, and Testcontainers

AI # LLM # Ollama # Quarkus # Testcontainers

Author: Jonathan Vila López

Java Champion, Organiser at BarcelonaJUG and cofounder of JBCNConf and DevBcn conferences in Barcelona. Currently working as a Developer Advocate in Java at Sonar (SonarLint, SonarQube), focused on Clean Code & Security. I have worked as a (paid) developer since the first release of The Secret of Monkey Island, about 30 years ago using Go on Kubernetes for a Service Mesh layer on top of Istio | Java on Kubernetes for K8s Operator, Rest API, using Quarkus, GraalVM, Apache Camel | PHP | VB | Python | Pascal | C I am very interested in simulated reality, psychology, philosophy, and Java.



LinkedIn

Traditionally, many AI-powered applications rely on cloud-based APIs or centralized services for model hosting and execution. While this approach has its advantages, such as scalability and ease of use, it also introduces challenges around latency, data privacy, and dependency on third-party providers.

This is where local AI models shine. By running models directly within your application's infrastructure, you gain greater control over performance, data security, and deployment flexibility. However, building such systems requires the right tools and frameworks to bridge the gap between traditional software development and AI model integration.

In this article, we explore how to combine Quarkus, a modern Java framework optimized for cloud-native applications, with Ollama, a platform for running AI models locally. We'll also demonstrate how tools like Testcontainers and Quarkus Dev Services simplify development and testing workflows. Using the PingPong-AI project as a practical example, you'll learn how to build and test AI-driven applications that harness the power of local models.

The PingPong-AI project demonstrates a simple implementation of AI-powered functionality using Quarkus as the backend framework and Ollama for handling AI models. Let's break down the architecture and walk through key components of the code.

Project Overview

In PingPong-AI, Ollama is used to simulate a simple conversation model where a curious service generates questions around a topic and a wise service responds with informed answers, that will generate more questions on the curious service.

The project integrates Quarkus with Ollama to create an AI model-driven application. It leverages Quarkus's lightweight and fast development model to serve as a backend for invoking and managing AI interactions. Here's an overview of what we'll cover:

1. Integrating Quarkus with Ollama
2. Using Testcontainers for Integration Testing
3. Leveraging Quarkus Dev Services for Simplified Development

1. Integrating Quarkus with Ollama

Why Ollama?

Ollama simplifies the deployment and use of AI models in applications. It provides a runtime for AI model execution and can be easily integrated into existing applications.

Quarkus and Ollama Integration

The integration with Quarkus is handled using REST endpoints to interact with Ollama. The Quarkus application serves as the middleware to process client requests and communicate with the Ollama runtime.

Here's a snippet from `PingPongResource.java`, which defines the REST endpoint for the PingPong interaction:

```
1  @Path("/chat")
2  public class CuriousChatResource {
3      @Inject
4      CuriousService curiousService;
5
6      @Inject
7      WiseService wiseService;
8
9      @POST
10     @Produces(MediaType.TEXT_PLAIN)
11     @Consumes(MediaType.TEXT_PLAIN)
12     @Path("/{numberOfQuestions}")
13     public String chat(@PathParam("numberOfQuestions") Integer numberOfQuestions, String
        topic) {
14         StringBuilder sb = new StringBuilder();
15         sb.append("> Topic: ").append(topic).append("\n");
16         while (numberOfQuestions-- > 0) {
17             String question = curiousService.chat(topic);
18             sb.append("> Question: ").append(question).append("\n");
19             topic = wiseService.chat(question);
20             sb.append("> Answer: ").append(topic).append("\n");
21         }
22
23         return sb.toString();
24     }
25 }
```

In this code:

- The `CuriousChatResource` handles client POST requests.
- The injected `CuriousService` and `WiseService` interface with the Ollama runtime to process the message.

These Ollama services are responsible for calling the Ollama runtime. Here's a snippet from `CuriousService.java`:

```
1  @RegisterAiService
2  @SystemMessage("You are a curious person that creates a short question for every message
    you receive.")
3  public interface CuriousService {
4      public String chat(@UserMessage String message);
5  }
```

We can even use different models for each service, specifying the configuration property that identifies the model. This example comes from `CuriousService.java`:

```
1  @RegisterAiService(modelName = "curiousModel")
```

And we identify the model in `application.properties`:

```
1  quarkus.langchain4j.ollama.wiseModel.chat-model.model-id=tinydolphin
2  quarkus.langchain4j.ollama.curiousModel.chat-model.model-id=tinylama
```

2. Using Testcontainers for Integration Testing

Why Testcontainers?

Testcontainers is a Java library for running lightweight, disposable containers during tests. In PingPong-AI, Testcontainers are used to set up an environment with an Ollama runtime for integration testing.

Example: Setting up a Testcontainer for Ollama

The test class demonstrates how to configure and use Testcontainers:

```
1  @QuarkusTest
2  class CuriousChatResourceTest {
3      @Inject
4      CuriousService curiousService;
5
6      @Inject
7      WiseService wiseService;
8
9      @Test
10     @ActivateRequestContext
11     void testFXMainControllerInteraction() {
12
13         // Perform interaction and assertions
14         var curiousAnswer = curiousService.chat("Barcelona");
15         var response = wiseService.chat(curiousAnswer);
16
17         Log.infof("Wise service response: %s", response);
18         // Using llama2 model we can check if the response contains 'Barcelona', but not
19         // with tinyllama
20         assertFalse(response.isEmpty(), "Response should not be empty");
21     }
22     @Test
23     @ActivateRequestContext
24     void testChatEndpoint() {
25         given()
26             .when()
27                 .body("Barcelona")
28                 .contentType(ContentType.TEXT)
29                 .post("/chat/3")
30             .then()
31                 .statusCode(200)
32                 .contentType(ContentType.TEXT)
33                 .body(not(empty()))
34                 .body(org.hamcrest.Matchers.stringContainsInOrder("Question:", "Answer:",
35                     "Question:", "Answer:", "Question:", "Answer:"));
36     }
37 }
```

Key Points:

- The `@QuarkusTest` annotation allows Quarkus to run the application in a test-friendly mode.
- Quarkus finds a service (Ollama) for which it needs an instance and it will spin up a container for that.

3. Leveraging Quarkus Dev Services for Ollama

What Are Quarkus Dev Services?

Quarkus Dev Services simplifies the setup of required services during development. For this project, Quarkus Dev Services can spin up an Ollama runtime container automatically.

Configuring Dev Services

The `application.properties` file includes configurations for enabling Dev Services:

```
1 quarkus.langchain4j.ollama.chat-model.model-id=tinyllama
2 quarkus.langchain4j.ollama.devservices.model=tinyllama
3 quarkus.langchain4j.log-requests=true
```

With these configurations, Quarkus Dev Services automatically starts an Ollama container when the application is run in development mode or in test, removing the need for manual setup.

Development Workflow

You can launch the application in development mode using the following command:

```
1 ./mvnw quarkus:dev
```

This command:

- Starts the Quarkus application.
- Automatically sets up an Ollama runtime container.
- Enables hot-reloading for rapid development.

Conclusion

The PingPong-AI project demonstrates a seamless integration of Quarkus with Ollama, making it easy to build AI-powered applications. By leveraging Testcontainers and Quarkus Dev Services, developers can efficiently test and develop their applications in a containerized and automated environment.

Key Takeaways:

- Quarkus provides a lightweight framework for building and deploying Java applications.
- Ollama simplifies AI model integration with backend systems.
- Testcontainers and Dev Services streamline the testing and development workflows.

To explore this project further, check out the [PingPong-AI GitHub repository](#).

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



Evaluating RAG Pipelines with the RAG Triad

GenAI

LLM

PromptEngineering

Author: Atamel Mete

He's a Software Engineer and a Developer Advocate at Google in London. He build tools, demos, tutorials, and give talks to educate and help developers to be successful on Google Cloud.



LinkedIn

Retrieval-Augmented Generation (RAG) emerged as a dominant framework for feeding Large Language Models (LLMs) the context beyond the scope of their training data and enabling LLMs to respond with more grounded answers and fewer hallucinations based on that context.

However, designing an effective RAG pipeline can be challenging. You need to answer questions such as:

1. How should you parse and chunk text documents for vector embedding? What chunk size and overlay size should you use?
2. What vector embedding model should you use?
3. What retrieval method should I use to fetch the relevant context? How many documents should you retrieve by default? Does the retriever actually manage to retrieve the applicable documents?
4. Does the generator actually generate content that is in line with the retrieved context? What parameters (model, prompt template, temperature) work better?

The only way to objectively answer these questions is to measure how well the RAG pipeline works, but what exactly do you measure, and how do you measure it? This is the topic I'll cover here.

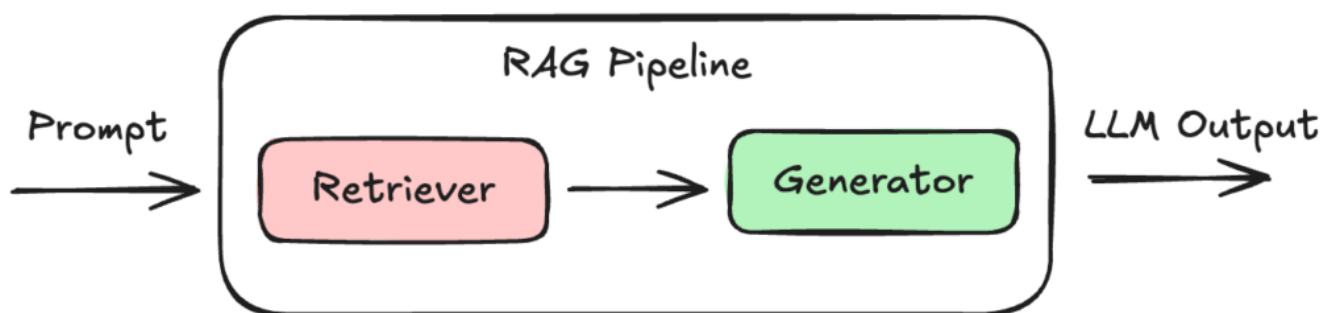
Typical RAG Pipeline

A typical RAG pipeline is made up of three separate phases: **ingestion**, **retrieval**, and **generation**.

In the **ingestion** phase, you split documents into smaller text chunks (with a certain chunk size and overlay size), use an embedding model to convert those text chunks into vectors (numerical representations of the text), and save them into a vector database. This is typically a one-time operation (unless you need to change one of the parameters, such as the embedding model or chunk size).

In the **retrieval** phase, the **retriever** part of the RAG pipeline takes the user's question, uses the same embedding model to convert it into a vector, and performs a similarity search against the text chunks in the vector database. This retrieves the top K similar text chunks that get aggregated into a **context** for the generation phase.

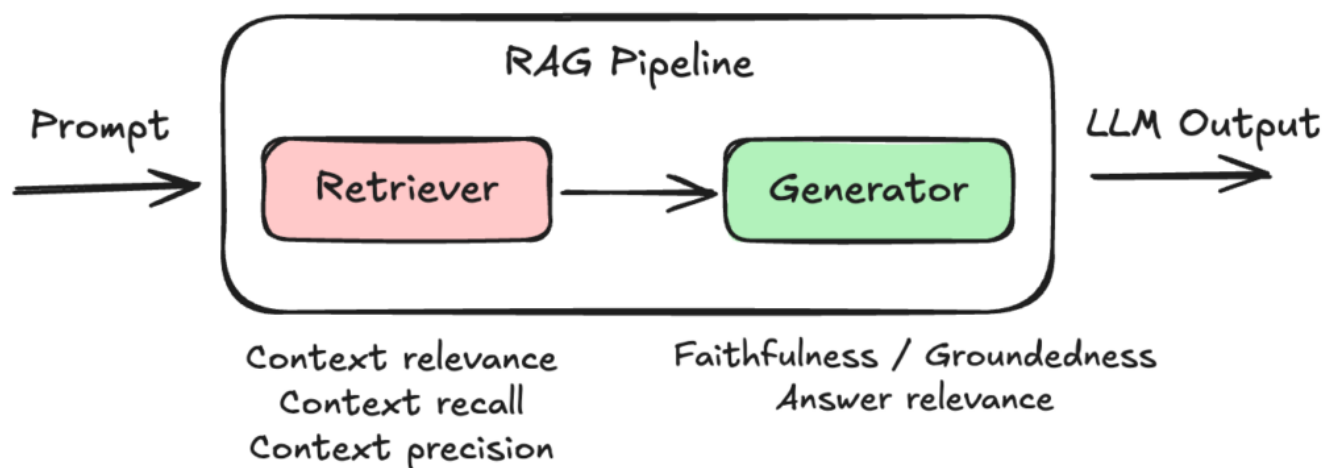
In the **generation** phase, **the generator** takes the **context** from the retrieval phase and generates a response to the user using the supplied parameters, such as the LLM, prompt template, and temperature.



I've seen two approaches in measuring the effectiveness of the RAG pipeline.

Approach 1: Evaluating Ingestion+Retrieval and Generation Separately

In this approach, you evaluate the **RAG pipeline's ingestion + retrieval** and **generation** phases separately using their separate metrics.



For the **ingestion+retrieval** phase, you need to measure how effectively you're storing and retrieving relevant context. These are the helpful metrics:

- Context Relevance evaluates the relevance of the retrieved context for the given input. A low score usually indicates a problem with how the text is chunked, embedded, and retrieved.
- [Context Recall](#) evaluates how many of the relevant documents were successfully retrieved. It focuses on ensuring important results are included. Higher recall means fewer relevant documents were left out.
- [Context Precision](#) evaluates whether nodes in the retrieved context that are relevant to the given input are ranked higher than irrelevant ones.

Note that both **context recall** and **context precision** require an expected output (the ideal answer to a given input) to compare against.

For the **generation** phase, you need to determine how effective the generator is in generating relevant and grounded output with the supplied context. These are the metrics:

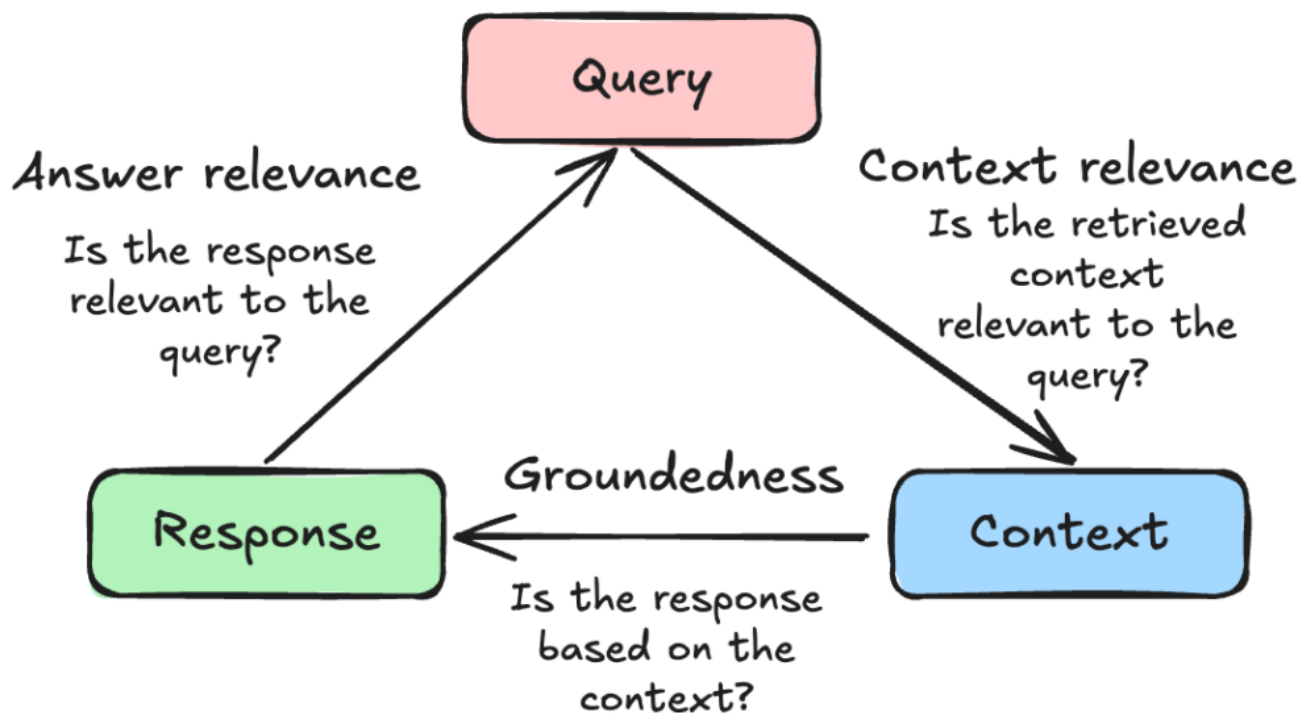
- [Faithfulness / Groundedness](#) evaluates whether the actual output factually aligns with the retrieved context. If this score is low, it usually points to a problem in your **model**. Maybe you need to try a better model or fine-tune your own model to get more grounded answers based on the retrieved context.
- [Answer relevance](#) evaluates how relevant the actual output is to the given input. If this score is low, it usually points to a problem in your **prompt**. Maybe you need better prompt templates or better examples in your prompts to get more relevant answers.

This approach allows pinpoint issues in the ingestion+retrieval or generation phases. You can determine whether the retriever is failing to retrieve the correct and relevant context or whether the generator is hallucinating despite being provided the right context.

As already mentioned, both **context recall** and **context precision** require an expected output. This might not be possible to determine upfront. That's why the **RAG triad** emerged as the alternative **referenceless RAG evaluation method**.

Approach 2: RAG Triad

In this approach, you evaluate the RAG pipeline as a whole using the subset of the RAG evaluation metrics: **answer relevance**, **faithfulness**, and **context relevance**.



Since **context precision** and **context recall** are not part of the RAG triad, this allows evaluations without expected outputs or a reference to compare against.

Evaluation Frameworks

At this point, you might be wondering: How do I implement the RAG triad? Ideally, you use an LLM evaluation framework. There are many out there. These are the ones I used:

- DeepEval has been my go-to LLM evaluation framework. It provides the metrics needed for the RAG triad and a guide to it.
- [TruLens](#) is another LLM evaluation framework with a [RAG triad guide](#).
- [Ragas](#) has an extensive list of [metrics](#), and it can be used for the RAG triad.

Do you know any other good evaluation frameworks? If so, let me know!

Conclusion

In this blog post, I explored a couple of different approaches to evaluating RAG pipelines and introduced the RAG triad metrics. If you're looking for sample implementations, I have some [DeepEval RAG evaluation samples](#) in my GitHub repo.

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



Building AI-Driven Applications With Spring AI

AI

Spring

SpringBoot

Author: Timo Salm

Principal Solutions Engineer at VMware Tanzu by Broadcom with over a decade of experience in modern applications, DevSecOps, and AI.



LinkedIn

Artificial intelligence (AI) is becoming increasingly essential to modern applications. While AI encompasses many different techniques, the current industry focuses on Generative AI (GenAI) due to the latest advancements in large language models (LLMs).

Traditionally, Python has been the dominant language for integrating AI capabilities into applications. However, for Java developers adopting Generative AI, the Spring AI project offers an attractive solution that enables the seamless development of enterprise-grade applications while keeping pace with the rapidly evolving AI landscape.

Spring AI abstracts complex interactions with various AI providers providing REST APIs, such as OpenAI, Anthropic, Microsoft, Google, Amazon, and even local LLMs. Its model-agnostic nature allows for easy switching between models, and, as usual in Spring, you still have access to functionalities and configurations unique to a particular model. The framework automatically converts AI model output into Java objects, ensuring type safety across your application and provides other fundamental features like multimodality, AI-related observability, and model response evaluation testing.

Additionally, Spring AI supports more advanced AI patterns, such as Tool Calling, Retrieval-Augmented Generation (RAG), and the Model Context Protocol (MCP), to provide context to LLMs.

Spring AI is built upon the core building blocks of the Spring Framework and other Spring projects like Spring Data for integrating vector databases. Spring Boot simplifies and speeds up the development of AI-powered features through auto-configuration.

Getting Started

Now that you understand what Spring AI offers let's dive into some hands-on coding!

To use Spring AI in your Spring Boot application, it's first necessary to add the appropriate dependency for a selected AI provider. In this example, Ollama is used, allowing you to run various LLMs locally. It is also recommended to add the Spring AI Bill of Materials (BOM) until Spring Boot begins managing the versions of Spring AI dependencies. This BOM specifies compatible versions of all the required dependencies.

```
1 implementation platform("org.springframework.ai:spring-ai-bom:${springAiVersion}")
2 implementation 'org.springframework.ai:spring-ai-ollama-spring-boot-starter'
```

Spring AI provides Spring Boot auto-configuration for all its features and supported AI providers, allowing you to achieve results quickly. The default configuration for Ollama assumes that the LLM model is Mistral and that the API for interacting with the model is hosted on the same machine listening on Ollama's default port, 11434.

```
1 ollama pull mistral
```

You can easily change Spring AI's auto-configuration defaults with configuration properties or code, such as swapping the Mistral for the Llama 3.2 model.

```
1 spring.ai.ollama.chat.model=llama3.2
```

The ChatClient API

The `ChatClient` API, which is idiomatically similar to the `WebClient` and `RestClient` APIs, makes integrating AI-powered functionality into a Spring Boot application seamless. Its abstraction layer enables developers to switch AI providers with minimal configuration and no code changes, aligning with Spring AI's key design principle of portability. Its abstraction layer enables developers to switch AI providers with minimal configuration and no code changes, aligning with Spring AI's key design principle of portability.

A `ChatClient` instance can be created utilizing the Builder pattern with a `ChatClient.Builder` object, available for injection into your Spring beans through auto-configuration.

```

1 @Component
2 class MyComponent {
3
4     private final ChatClient chatClient;
5     // Constructor-based dependency injection
6     public MyComponent(ChatClient.Builder chatClientBuilder) {
7         this.chatClient = chatClientBuilder.build();
8     }
9     ...
10 }

```

The `ChatClient` API supports synchronous and streaming communication. Sending a basic prompt in synchronous mode to an AI model requires just four lines of code.

```

1 String answer = this.chatClient.prompt()
2     .user("What is the capital of Germany?")
3     .call()
4     .content();

```

The `prompt()` method initializes the interaction, allowing you to construct the prompt by adding messages, like, in this case, a user message. The `call()` method sends the constructed prompt to the AI provider and retrieves the response. Finally, the `content()` method extracts and returns the model's output as a `String`.

Messages can include placeholders enclosed in curly braces to represent dynamic values. Instead of providing a plain text message, you need to use a function of type `Consumer<PromptUserSpec>` with the `user()` method to specify the values for these placeholders. A `Consumer` in Java is a functional interface that takes a single input, in this case of type `PromptUserSpec`, and performs an operation on it without returning a result.

```

1 String country = "Germany";
2 String answer = this.chatClient.prompt()
3     .user(promptUserSpec -> promptUserSpec
4         .text("What is the capital of {country}?")
5         .param("country", country))
6     .call()
7     .content();

```

For AI models that support multimodality, which means they can process different input types, such as text, images, and audio, the `PromptUserSpec` also provides a `media()` method to include pictures and audio alongside text prompts.

```

1 Resource imageResource = new ClassPathResource("berlin.png");
2 String answer = this.chatClient.prompt()
3     .user(promptUserSpec -> promptUserSpec
4         .text("What is the name of this city?")
5         .media(MimeTypeUtils.IMAGE_PNG, imageResource))
6     .call()
7     .content();

```

Structured Output

The `ChatClient` API provides multiple ways to handle AI-generated output. In the previous example, we used the `content()` method to retrieve the model's response as a `String`. However, the API also supports mapping the output directly into Java objects. This is achieved using system messages, which are special instructions in the prompt that guide the model's behavior. The `ChatClient` API's `system()` method allows you to incorporate system messages into a prompt similar to user messages.

```

1 String answer = this.chatClient.prompt()
2   .user("What is the capital of Germany?")
3   .system("You are an expert in history. Provide detailed explanations.")
4   .call()
5   .content();

```

Additionally, you can set a default system message for `ChatClient` instances using the `defaultSystem()` method in the `ChatClient.Builder`.

To enable the automatic mapping of AI-generated output into Java objects within the `ChatClient`, the **StructuredOutputConverter** appends a system message to the prompt behind the scenes. This message instructs the model to format its response in a specific format, such as JSON, which is then mapped to a Java object.

```

1 record City(String name, String zipcode) {}
2
3 City capitalOfGermany = this.chatClient.prompt()
4   .user("What is the capital of Germany?")
5   .call()
6   .entity(City.class);

```

Advanced AI Patterns

While prompt engineering helps guide an AI model's responses, it has limitations. No matter how well a prompt is crafted, an LLM can only generate answers based on its pre-trained knowledge and the context provided within the prompt itself. To overcome these constraints, AI systems increasingly adopt agentic patterns, where agents – autonomous intelligent systems capable of performing specific tasks without human intervention – enhance an LLM's capabilities by planning, executing actions, and dynamically retrieving or generating new information. Spring AI provides the essential building blocks for implementing and using AI agents.

Tool Calling

One such building block is Tool Calling, also known as Function Calling, which enables LLMs to leverage external APIs to perform specific tasks and incorporate up-to-date information. In this approach, when an LLM receives a user prompt, it evaluates whether invoking a defined external function is necessary to fulfill the request. If so, the model generates a structured request detailing the function to be called and the required parameters. The client system then processes this request, executes the specified function, and returns the result to the LLM.

Spring AI provides several ways to define callable tools. One of them is the declarative approach using the `@Tool` annotation, offering a description field to explain the purpose and functionality to the LLM. Similarly, the `@ToolParam` annotation is used to describe individual parameters of these tools.

```

1 @Service
2 class WeatherService {
3
4     @Tool(description = "Fetches the current temperature in a city")
5     Double fetchCurrentTemperature(
6         @ToolParam(description = "Name of the city") String city) {
7         ...
8     }
9 }

```

To use a defined tool in a specific chat request, pass an instance of the tool class to the `tools()` method in the `ChatClient` prompt. To make tools available across all chat requests, register them at `ChatClient.Builder`'s `defaultTools()` method.

```

1 String answer = this.chatClient.prompt()
2   .user("What is the current temperature in the capital of Germany?")
3   .tools(weatherService)
4   .call()
5   .content();

```


Under the hood, Spring AI sends tool definitions along with the prompt to the LLM, allowing it to decide whether a function call is needed. If so, the model responds with the tool's name and required parameters. Spring AI then executes the tool with the provided inputs, retrieves the result, and returns it to the LLM. The model incorporates this data to generate a more accurate and contextually relevant response.

The Tool Calling capability of LLMs is also fundamental to the **Model Context Protocol (MCP)**. This open protocol standardizes the connection between AI models and various data sources and tools. It allows developers to build agents on top of LLMs. MCP follows a client-server architecture, where an AI application (MCP host) communicates through embedded MCP clients to MCP servers to access specific resources or functionalities.

Spring AI has recently introduced support for implementing MCP clients and servers based on the official MCP Java SDK. While this is fantastic, it deserves a more detailed discussion on its own, which is beyond the scope of this article.

Retrieval-Augmented Generation

Another advanced technique for enhancing the output of large language models by integrating external data is called Retrieval-Augmented Generation (RAG). With RAG, external data is transformed into vector embeddings stored in a vector database. A conversion of a user query into a vector embedding is also necessary to retrieve semantically relevant data from the database, which is then added to the LLM's context to improve responses.

To implement a basic RAG flow, you first need to integrate your data into a vector store. Spring AI facilitates this process through an Extract, Transform, Load (ETL) pipeline. Initially, a `DocumentReader` extracts content from various sources, such as PDFs, and converts it into structured `Document` objects. Next, these documents are split into chunks using a `DocumentTransformer` to ensure they align with the context window limitations of AI models. Finally, the chunks are stored in a vector database using a `DocumentWriter`, which is extended by the `VectorStore` interface.

```
1 jDocumentReader documentReader = new PagePdfDocumentReader(pdfResource);
2 List<Document> documents = new TokenTextSplitter().apply(documentReader.get());
3 vectorStore.accept(documents);
```

With the `VectorStore` interface, Spring AI provides an abstraction layer that enables seamless adaptability with minimal code changes for integrating various vector databases. Each supported vector database has its own individual Spring Boot starter dependency. In this example, PGvector, an open-source vector extension for PostgreSQL, is used.

```
1 implementation 'org.springframework.ai:spring-ai-pgvector-store-spring-boot-starter'
```

A `VectorStore` instance in Spring AI requires an embedding model, which is available through the `EmbeddingModel` API. Since most AI providers offer embedding models, additional dependencies are usually unnecessary. Thanks to Spring Boot's auto-configuration, a `VectorStore` and an `EmbeddingModel` instance are available with minimal setup.

```
1 spring.ai.ollama.embedding.model=nomic-embed-text
```

Spring AI's support for Retrieval-Augmented Generation is built upon its `Advisor` API, which enables developers to intercept and transform data exchanges with LLMs. For common RAG workflows, Spring AI offers two out-of-the-box advisors. The `QuestionAnswerAdvisor` is designed for a basic RAG flow, while the `RetrievalAugmentationAdvisor` provides a modular architecture for custom retrieval and augmentation strategies to support more advanced RAG flows, such as routing between multiple vector databases.

Advisors can be configured for a specific chat request using the `advisors()` method within the `ChatClient` prompt or across all chat requests via the `ChatClient.Builder`.

```
1 String answer = this.chatClient.prompt()
2     .user("What are the best cities to visit based on my travel guides?")
3     .advisors(new QuestionAnswerAdvisor(vectorStore))
4     .call()
5     .content();
```

In this example, a `QuestionAnswerAdvisor` is configured with an auto-configured `VectorStore` instance. The retrieval behavior can be fine-tuned using an optional `SearchRequest` parameter, which defines how relevant data is searched within the vector database. Another parameter allows developers to provide additional guidance to the LLM on how to interpret and utilize the retrieved data in the prompt.

Image and Audio AI Models

Underlying the `ChatClient` is a `Model` API interface that supports all types of AI models, including embedding, image, and audio models. While the `ChatClient` is necessary to simplify complex prompt creation for chat models, image and audio models typically require less intricate prompts. In these cases, direct interaction with the `Model` API is sufficient. As an additional convenience, Spring Boot's auto-configuration pre-configures `Model` API instances for all types of models, allowing for easy injection into your Spring beans.

```
1 String prompt = new PromptTemplate("Generate a picture of {city}")
2     .render(Map.of("city", "Berlin"));
3 ImageGeneration imageGeneration = imageModel.call(prompt).getResult();
4 Image image = imageGeneration.getOutput();
5 String imageUrl = image.getUrl();
```

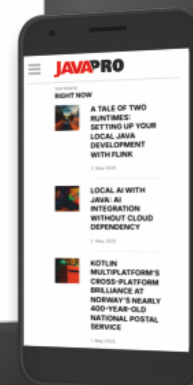
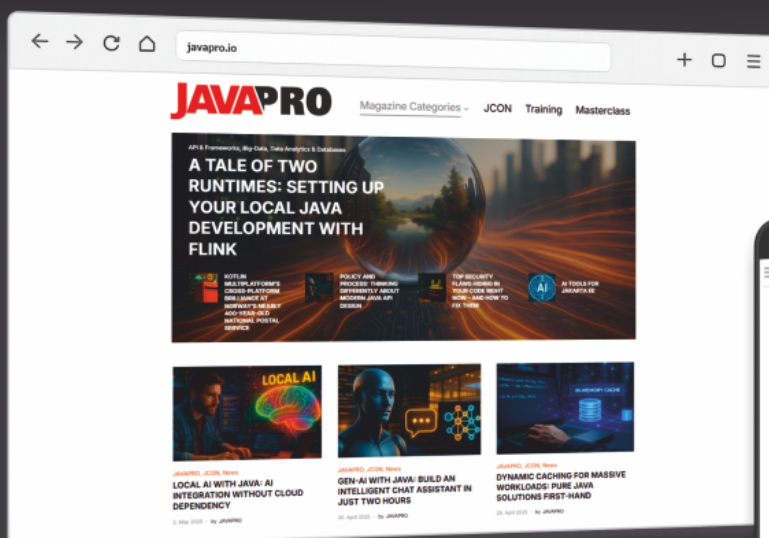
Summary

To sum up, Spring AI enables you to integrate GenAI capabilities into Spring Boot applications as seamlessly as databases. It supports various AI models with a focus on portability and provides fundamental features like type-safe responses and multimodality. Beyond that, Spring AI provides essential building blocks for agentic systems and supports the Model Context Protocol, enabling integration with third-party tools and resources. With these features, Spring AI helps you stay at the forefront of evolving AI technologies.

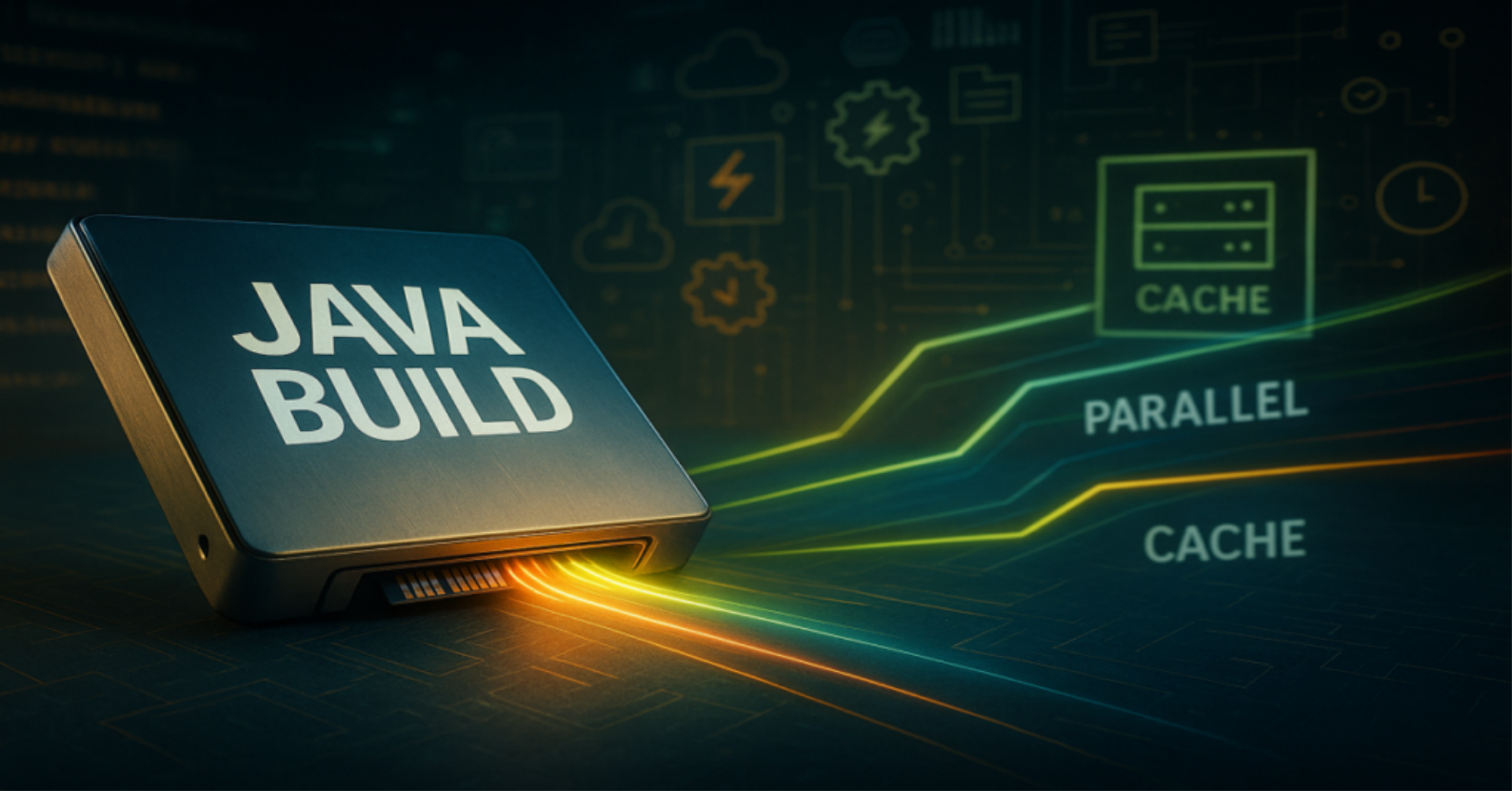
› [Back to the Table of Contents](#) ‹

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



How to Speed up Maven Builds

Build

Gradle

Maven

Optimize

Author: Sergei Chernov

Seasoned java developer, last years mostly focused on developer productivity engineering. Occasional open-source contributor, author and speaker.



LinkedIn

As a Maven project grows in size, the question of build speed and test execution time becomes more important. One of the most obvious ways to optimize is to switch to another build system (e.g., Gradle). However, before making the switch (which will inevitably bring new challenges—we'll discuss this at the end), let's first explore what we can do to speed things up while staying on Maven. Many of these tips will be useful regardless, even if you eventually decide to migrate.

We'll start with the basics, discuss caches and build scans, and then take a deeper look under Maven's hood to identify and fix bottlenecks. We'll also compare it in detail with Gradle. Each tip alone might not make a huge difference, but combined, they can significantly boost performance!

In this article, we'll focus mainly on optimizing compilation.

Checking the JDK

If your engineers are using MacBooks with Apple processors (or Windows on ARM), it's worth checking whether an x64 JDK is running in emulation mode. While technologies like Rosetta provide convenience, they also introduce noticeable additional CPU load. You can check this in the console with the command **mvn -v**:

```
1 mvn -v
2
3 Apache Maven 3.9.6 (bc0240f3c744dd6b6ec2920b3cd08dcc295161ae)
4 Maven home: /opt/homebrew/Cellar/maven/3.9.6/libexec
5 Java version: 17.0.10, vendor: Amazon.com Inc., runtime: /Users/sergei/Java/amazon-corretto-17.jdk/Contents/Home
6 Default locale: en_US, platform encoding: UTF-8
7 OS name: "mac os x", version: "14.5", arch: "aarch64", family: "mac" 
```

```
1 mvn -v
2
3 Apache Maven 3.9.6 (bc0240f3c744dd6b6ec2920b3cd08dcc295161ae)
4 Maven home: /opt/homebrew/Cellar/maven/3.9.6/libexec
5 Java version: 17.0.11, vendor: Oracle Corporation, runtime: /Users/morph/Java/oracle-x64-jdk-17.0.11.jdk
6 Default locale: en_US, platform encoding: UTF-8
7 OS name: "mac os x", version: "14.5", arch: "x86_64", family: "mac" 
```

Abb. 12: Check the value of “arch”

You can automate this check and show a warning message (or even fail the build) on local environments that you can't control. This can be useful if you want to solve this issue for your team of developers.

A ready-to-use solution for this is [jvm-arch-maven-extension](https://github.com/seregamorph/jvm-arch-maven-extension) – depending on configuration it will print warning message or fail the build:

```
1 <extensions>
2   <extension>
3     <!-- https://github.com/seregamorph/jvm-arch-maven-extension -->
4     <groupId>com.github.seregamorph</groupId>
5     <artifactId>jvm-arch-maven-extension</artifactId>
6     <version>0.1</version>
7   </extension>
8 </extensions>
```

Read the project documentation to find insights if you would like to implement your own build monitoring or validation.

Another important factor when choosing a JDK is that, on average, each new JDK release is faster than the previous one. This includes improvements in **javac** performance. The difference is especially noticeable when upgrading across multiple major versions. So, if you're still using an older JDK, this is another good reason to update.

The JDK vendor also matters—performance can vary significantly. For example, Oracle JDK is not the fastest option. There's a common recommendation to use the same JDK distribution as in production, but the choice is up to you.

Kotlin

If your project uses **Kotlin**, it's worth upgrading to version **2.x**. JetBrains has done a great work rewriting the compiler, and with **K2** compilation will be noticeably faster—even if your modules contain a mix of Java and Kotlin.

For a safer upgrade, it's best to follow these three steps:

1. First, update to **Kotlin 1.9.25** (the latest **1.x** version).
2. Then, switch to **Kotlin 2.0.21** (the latest **2.0.x** version) while still using the old compiler mode.
3. Finally, enable the **K2 compiler**.

The **compiler version mode** (which is different from the Kotlin plugin version) is set using the `language-version` parameter:



```

1 <kotlin.version>2.0.21</kotlin.version>
2 ...
3 <plugin>
4   <groupId>org.jetbrains.kotlin</groupId>
5   <artifactId>kotlin-maven-plugin</artifactId>
6   <version>${kotlin.version}</version>
7   <configuration>
8     <jvmTarget>17</jvmTarget>
9     <javaParameters>true</javaParameters>
10    <args>
11      ...
12      <arg>-language-version=1.9</arg>
13    </args>
14    <compilerPlugins>
15      <plugin>spring</plugin>
16    </compilerPlugins>
17  </configuration>
18 </plugin>
  
```

```

1 <kotlin.version>2.0.21</kotlin.version>
2 ...
3 <plugin>
4   <groupId>org.jetbrains.kotlin</groupId>
5   <artifactId>kotlin-maven-plugin</artifactId>
6   <version>${kotlin.version}</version>
7   <configuration>
8     <jvmTarget>17</jvmTarget>
9     <javaParameters>true</javaParameters>
10    <args>
11      ...
12      <arg>-language-version=2.0</arg>
13    </args>
14    <compilerPlugins>
15      <plugin>spring</plugin>
16    </compilerPlugins>
17  </configuration>
18 </plugin>
  
```

Build Caches and Build Scans

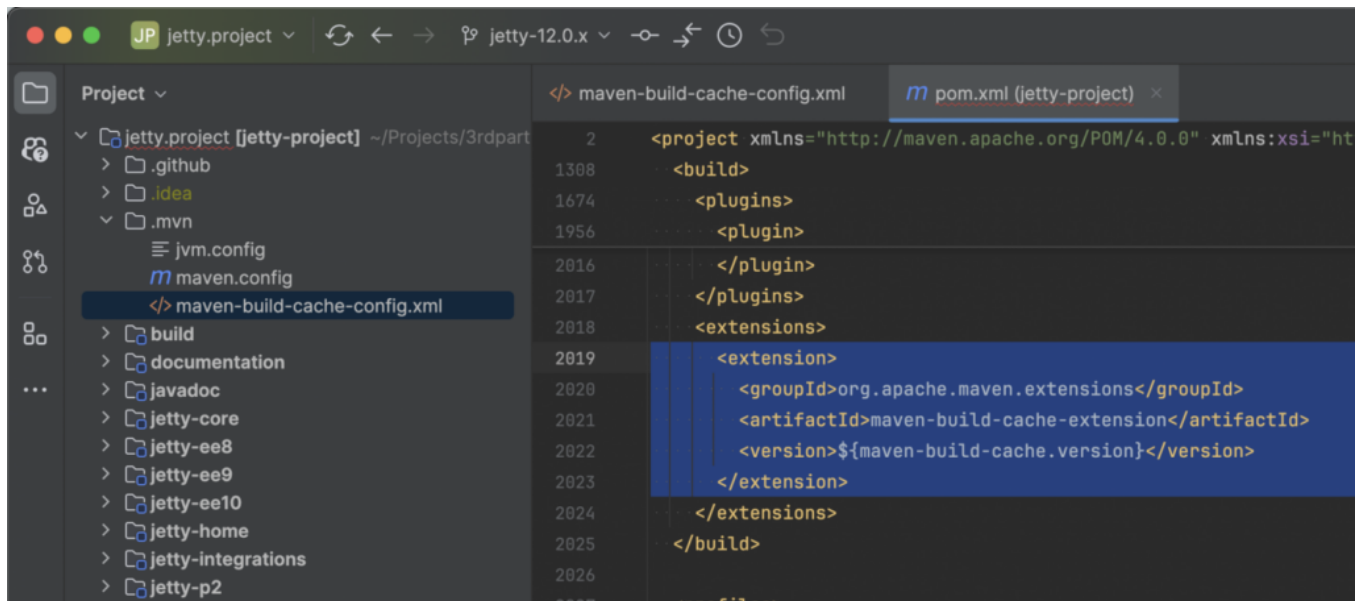
One of the biggest performance issues with **Maven** is the lack of a built-in caching mechanism. And it doesn't look like this will change in upcoming versions. However, there are external solutions available.

Apache provides the [Maven Build Cache Extension](#), which supports both **local** and **remote** caches (mainly for CI/CD). The main limitation of this solution is that it caches **Maven Lifecycle Phases**. This works well for medium-sized projects without heavy customization. If your project doesn't rely on custom plugins or advanced features like selective test execution, it's worth considering.

A typical Apache Maven Build Cache Extension setup consists of two files:

- `.mvn/extensions.xml`
- `maven-build-cache-config.xml`

Alternatively, instead of defining the extension in `.mvn/extensions.xml`, you can declare it in the **root pom.xml**, as done in the [Jetty](#) project (the config file is available [here](#)):



It turns out that **Maven** is quite flexible when it comes to customizing its behavior. You can even create your own extensions or plugins for build caching, distributed testing, and selective test execution.

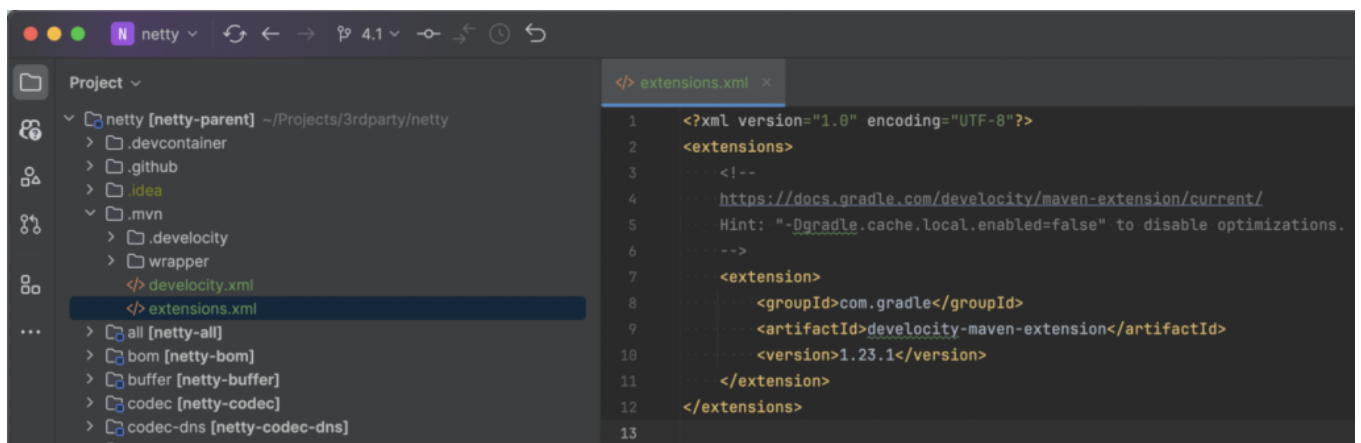
For example, here's my project for caching **maven-surefire-plugin** and **maven-failsafe-plugin**: [github](#)

Develocity (Formerly Gradle Enterprise)

Another option is **Develocity** from **Gradle**, a closed-source solution. It adds caching to Maven via the **Develocity Maven Extension**, which works differently from Apache's approach. Instead of caching **Maven Lifecycle Phases**, it caches each plugin goal, making it more granular. This caching mechanism is similar to what **Gradle** uses.

- **Remote caching** requires a paid license for Develocity.
- **Local caching and build scans** are available for free, with build scans stored on Gradle's cloud servers.
- If security policies prevent using build scans, they can be disabled while keeping local caching enabled.

The setup also requires two files:



By default Develocity will ask to submit build scan information to the cloud service. You should consult with your security team if that's acceptable. If not you can disable submitting build scan in the extension configuration.

Let's add the extension to the **Netty** project and compare the build speed. Compared to the default time (**2m11s**), the build is almost **twice as fast**. After the build finishes, the console will show a **build scan link**, something like this: <https://gradle.com/s/lh7h42es3riko> There, you can check detailed execution stats, including cache performance.



About build Scans **Develocity** provides detailed information about the project build, including resource usage, cache efficiency, and timelines. This makes it much easier to diagnose issues and identify bottlenecks.

Here's an example of build scans from Spring projects: <https://ge.spring.io/scans>

Parallel Builds

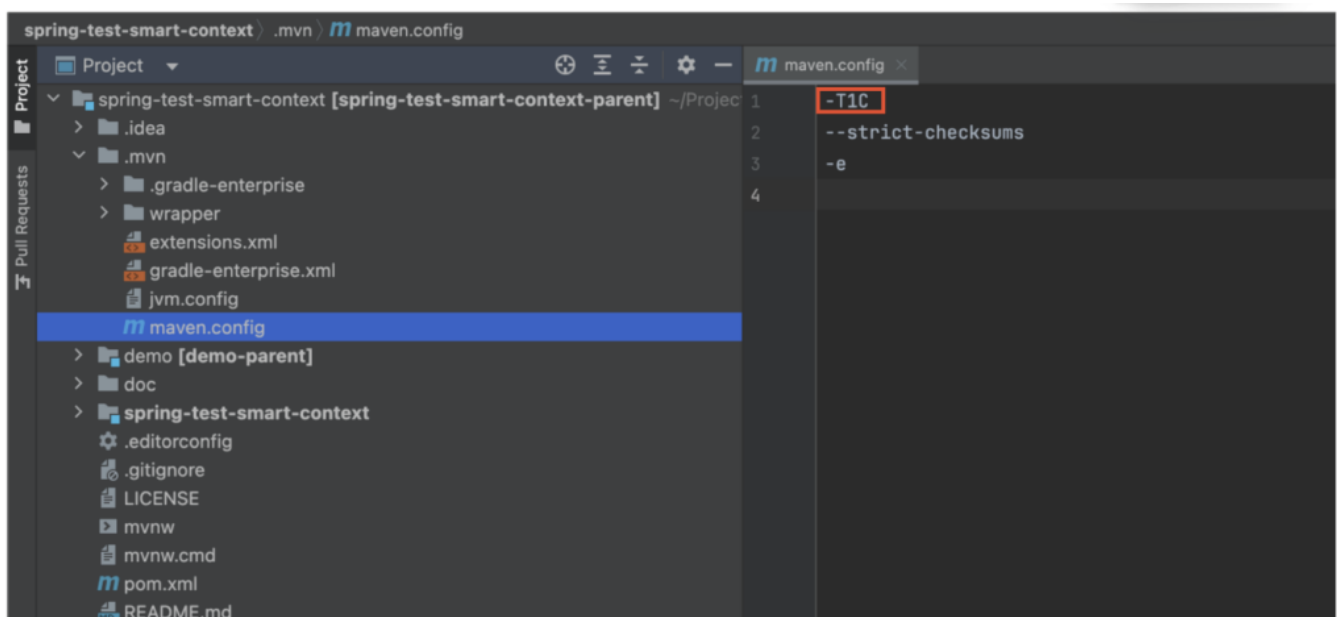
By default, Maven builds a project using a single thread, but it also supports multi-threaded builds using the `-T` parameter. You can specify a fixed number of threads, or use a core count multiplier `C` to determine the number of threads based on the available CPU cores:

```
1 # 6 threads
2 mvn clean package -T6
3
4 # same number of threads as CPU cores
5 mvn clean package -TC
```

It's important to note that not every project supports multi-threaded builds correctly. For example, if tests use fixed port numbers, conflicts will occur. In such cases, refactoring the tests to use dynamic or random ports can significantly improve build speed.

Another downside is that the build log will become **mixed** for all parallel tasks, which can make diagnosing issues more difficult.

You can set these parameters by default in the `.mvn/maven.config` file, so you don't need to specify them each time in the command line:



Let's run the build one more time with caches disabled. This will help us better identify the bottlenecks in parallelization.


```
1 mvn clean install -DskipTests=true -Dgradle.cache.local.enabled=false -T6
```



In the timeline diagram above, you can see the distribution of work across different threads—at some points, 5 threads are active simultaneously, while at other times, only one is active. This represents our **bottleneck** and the first candidates for optimization.

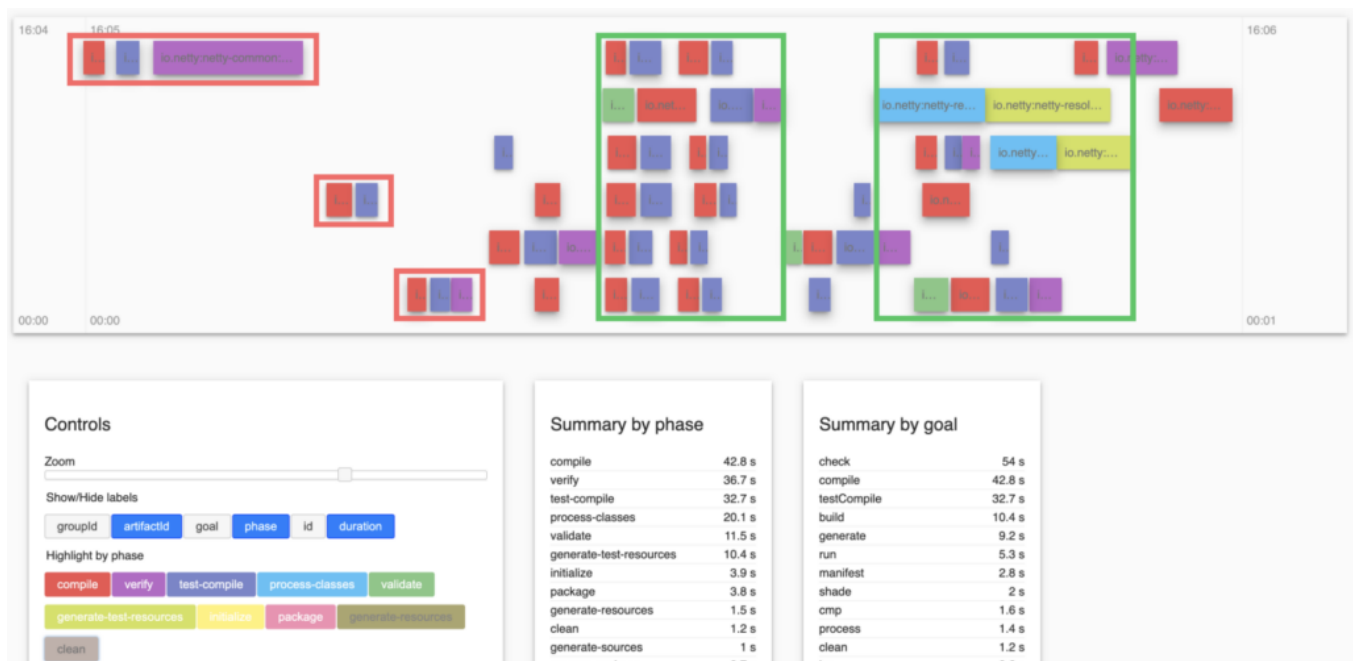
Takari Timeline Extension

Before we discuss how to improve parallelism, there's another way to visualize bottlenecks in parallel builds, especially if strict security policies prevent uploading build scans to an external cloud. The open-source Takari timeline extension can help there.

It's easy to integrate into the project:

```
1 <extensions>
2   <extension>
3     <!-- https://github.com/takari/maven-timeline -->
4     <groupId>io.takari.maven</groupId>
5     <artifactId>maven-timeline</artifactId>
6     <version>2.0.1</version>
7   </extension>
8 </extensions>
```

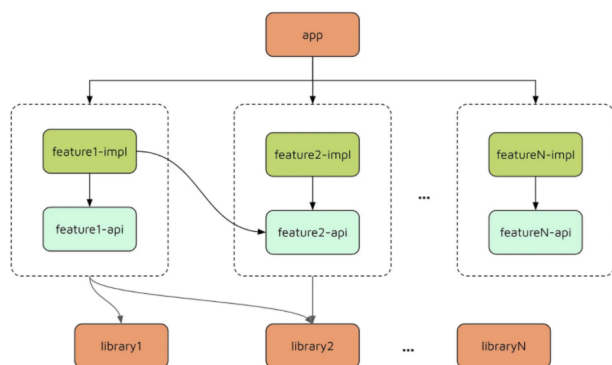
After any execution of maven command in the root `target` directory you can find HTML report with the timeline of task execution like this:



On the timeline you can distinguish tasks with low parallelism (build bottleneck) and good parallelism. Now let's try to discover Maven under the hood, how it works and how distributes tasks between cores.

How Parallelism Works in Maven?

The key here is multi-modularity. A single-module project offers almost no space for optimization in terms of parallelism. It's the presence of multiple modules that creates opportunities for improving build times through parallel execution.



For the modules on the diagram, parallelism is possible between those that do not depend on each other. For example, `feature1-impl` can be built in parallel with `feature2-impl`. Let's take a look under the hood of Maven to understand how it handles parallel builds.

We'll focus on a single module that declares dependencies on other modules and libraries from the repository (**upstream dependencies**), as well as modules that depend on the current one (**downstream dependencies**).

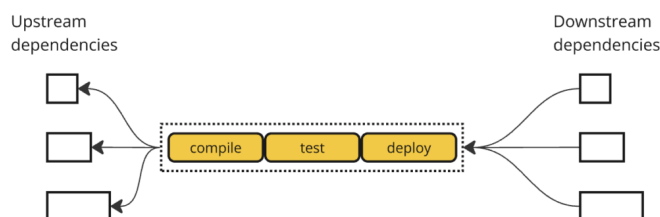


Abb. 13: Build lifecycles are simplified

The execution of this dependency tree is managed by **MultiThreadedBuilder**, which has a very simple contract:

Before starting the build of the current module, the scheduler waits for the complete build (all phases) of all modules it depends on (upstream dependencies), regardless of the dependency scope. The same rule applies in the opposite direction—any module that depends on the current one will wait for all build phases to complete before it starts its own build. All phases within a single module are executed sequentially by a single thread.

In other words, even if our hypothetical module `core` depends on `test-utils` in the `<scope>test</scope>`, Maven will not start building `core` until it has built and run the tests for `test-utils`. This greatly reduces parallelism, and the multi-core processor remains underutilized.

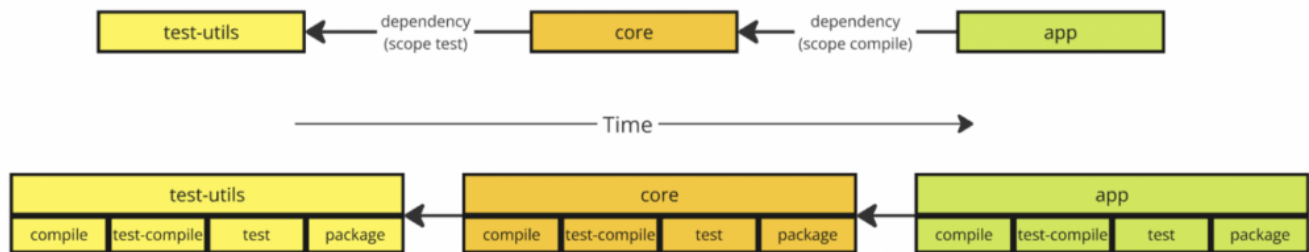
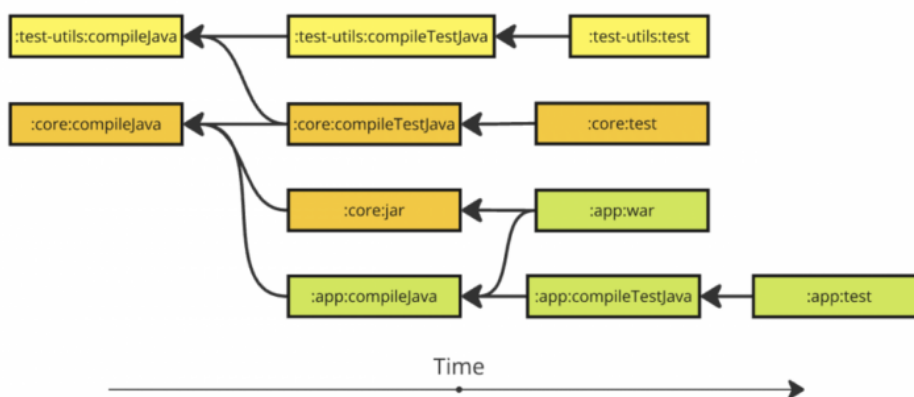


Abb. 14: Maven: no parallelism for modules with Dependencies on each other

Compare with Gradle



In this sense, Gradle performs much better. In Gradle, the dependency tree is built not between modules (Project) but between tasks (Task). This allows Gradle to parallelize tasks more effectively, resulting in faster builds (though at the cost of higher memory consumption).

Additionally, Gradle doesn't wait for the compilation and test execution to complete before it assembles artifacts (like **jar** or **war** files). This further optimizes the build process and reduces the overall build time compared to Maven, which waits for all phases of a module to complete before moving forward.

Removing Unnecessary Dependencies

The fewer dependencies a module has, the sooner Maven can build it (and the sooner it will be available for other modules that depend on it). To manage and remove unnecessary dependencies, you can use the built-in dependencies plugin in Maven.

For example, you can use the following command to list all the dependencies of a module:

```
1 mvn dependency:analyze
2
3 ...
4 [WARNING] Unused declared dependencies found:
5 [WARNING]   org.springframework.boot:spring-boot-starter-web:jar:2.4.1:compile
6 [WARNING]   org.springframework.boot:spring-boot-starter-data-jpa:jar:2.4.1:compile
7 [WARNING]   org.hibernate.validator:hibernate-validator:jar:6.1.6.Final:compile
8 [WARNING]   com.h2database:h2:jar:1.4.200:runtime
9 [WARNING]   com.fasterxml.jackson.core:jackson-databind:jar:2.11.3:compile
10 [WARNING]   com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar:2.11.3:compile
11 [WARNING]   org.springframework.boot:spring-boot-starter-test:jar:2.4.1:test
```

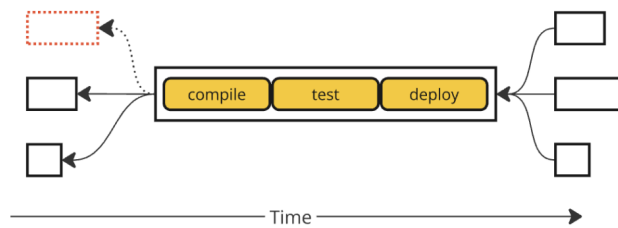


Abb. 15: When removing unnecessary dependencies, the module's build shifts to the left on the timeline.

It is important to be cautious when removing dependencies and not rely too heavily on the plugin's advice. It's quite possible that the removed dependency should remain in the final build. However, the lower it is in the dependency tree (most likely, your project has a module like **app** that depends on all others—make sure it's there), the better the parallelism.

Splitting Large Modules (Independent Parts)

It makes sense to look at the structure of the largest modules in the project—these are likely to have more dependencies, and there's a chance to find independent groups of classes. By splitting them into separate modules, we can likely reduce the dependencies of each one.

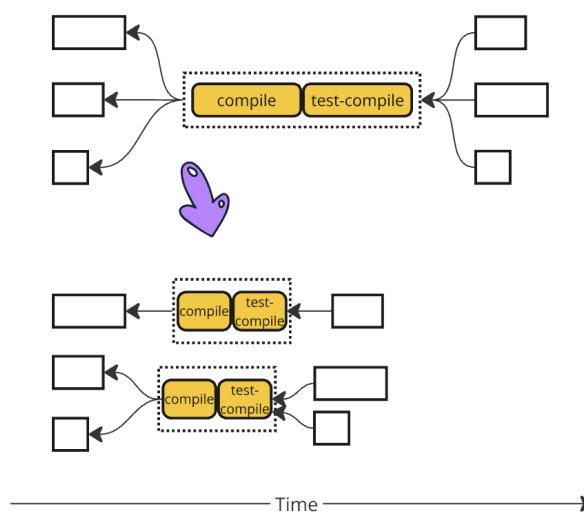
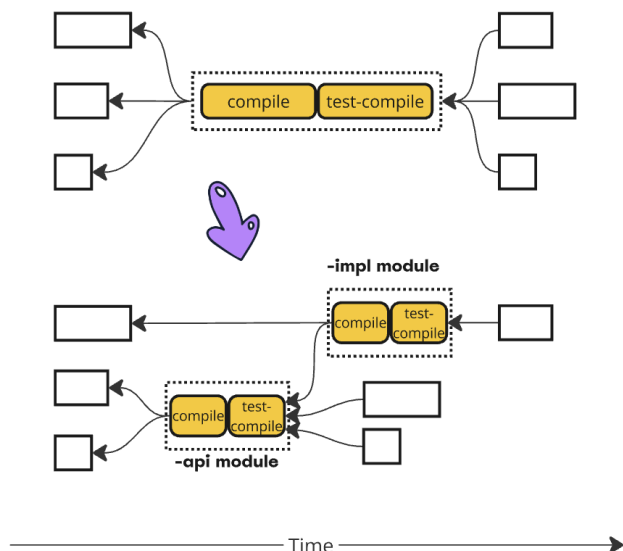


Abb. 16: After splitting, Maven will be able to build the modules independently and in parallel

You can achieve a significant speedup by breaking down several of the largest such modules. The text above describes how to find the first candidates for optimization (bottlenecks without parallelism).

Splitting Large Modules (Dependent Parts)

If you actively use interfaces and Dependency Injection (or SPI via ServiceLoader), large modules can be split into dependent parts: `-api` (with interfaces) and `-impl` (implementations). This approach generally promotes better module interface design by hiding implementation details. Additionally, it significantly reduces the number of required dependencies.



Most modules that depended on the large split module will now depend only on its `-api` part. The exception is the **app** modules and integration test modules, which still require the `-impl` modules.

Moving Code Generation to Libraries

There are two main types of code generation:

1. From static descriptors like **protobuf** or **avro** files, SQL (e.g., **jOOQ**), **OpenAPI** from YAML.
2. Generated from the source code itself – **Apache Feign**, **OpenAPI** and others.

We are primarily talking about the first type, as it is not tied to the current code and can be easily removed. Essentially, such code generation depends only on the tool itself and its “schema.”

In general, having code generation directly in the project is convenient, as everything is in one place—the schemas, source files, and generated code. But it creates a number of problems: firstly, you cannot simply open the project in IDEA and compile it directly through the IDE—it requires running Maven commands. After that, any application or test run from the IDE will trigger a full rebuild because the development environment doesn’t know how to properly handle an incremental build. Furthermore, once the code is generated, it will eventually become outdated—for example, when schema updates occur, you might encounter unclear bugs that can only be resolved by regenerating the code. Either way, the generated code also needs to be compiled, which consumes resources. If schemas or SQL don’t change very often, it makes sense to move everything into a library and include it as a dependency.

If You Really Want to Keep Code Generation

In the case of switching to Gradle, this can work more efficiently due to caches. Additionally, in IDEA, build operations are delegated to Gradle by default, so the build will automatically decide whether regeneration is necessary or not. If staying with Maven, you can reconfigure the code generation from `target/generated-sources/java` to `src/main/java`, meaning you will include this code in version control and update it every time the schema changes. Initially, this approach may seem strange, but it turns out to be a good compromise that works well for relatively large projects.

Once again, I want to emphasize why moving code generation is important: You will gain the ability to switch from a Maven build (locally) to a fully incremental build in IDEA—and it is definitely faster almost all the time.

Stop Publishing Unnecessary Artifacts

If you're developing a library, it's useful to publish `sources.jar`. However, if you're building an application, you shouldn't deploy every module separately, especially with `-javadoc`, `-sources`, `-test-sources`, and `-test-jar` (instead, publish only the final artifact, ideally as a multi-layered Docker image to save space). These extra operations not only consume a lot of storage in repositories but also significantly increase build time.

Even if you're not publishing them, check if such configurations exist in your project (again, unless you're building a library)—you might want to remove them:

- `maven-source-plugin` (`jar-no-fork` goal)
- `maven-javadoc-plugin` (`jar` goal)

Persistent CI/CD Build Agents

For CI/CD build agents, consider using persistent instances that are reused across multiple builds. This can significantly reduce build times by avoiding unnecessary dependency downloads from the repository. Additionally, previous build caches can remain stored in the local filesystem.

However, balance is key—keeping agents alive too long may increase operational costs and make it harder to diagnose issues that could be easily solved by restarting a fresh agent.

Use Maven Profiles

If certain build steps aren't always necessary, make them optional using Maven profiles. If only a small percentage of developers need specific steps (e.g., code generation or validation), avoid enabling them by default. You'll need to balance simplicity vs. optimization—decide what should run by default without extra parameters.

Use profiles for CI/CD steps or provide helper scripts so developers don't have to memorize how to enable/disable specific tasks.

Why Not Just Switch to Gradle?

Despite its advantages, Gradle has drawbacks:

- Higher memory and disk usage—cache directories can quickly grow in size on large projects.
- Imperative build scripts—while flexible, they can become hard to maintain and may break on major updates.
- Pipeline migration—switching to Gradle requires rebuilding all CI/CD pipelines and retraining engineers.
- Cache issues—even with Develocity, diagnosing build problems (like “poisoned” caches) can be frustrating.
- Gradle daemon quirks—can sometimes hang or even refuse to stop using `gradle --stop`.

Surprisingly, Maven can sometimes be faster, as Gradle's configuration phase can take a long time—Maven might finish recompiling in the same period (this statement is mostly related to compilation time).

Conclusion

In the long run, mature projects tend to migrate to Gradle (mainly for test optimization). However, many actively developed projects still work perfectly fine with Maven. It's far from obsolete—it continues to evolve and maintains a steady market share among Java projects.

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



A New Generation's Chance to Shape the Future: Building the Next Great Java IDE in Open Source

DeveloperTools

Eclipse

IDE

OpenSource

Author: Thomas Froment

Thomas is a passionate software engineer with extensive experience in developing Java application servers for industrial companies such as Alcatel-Lucent and Nokia. He co-founded and served as CTO of Komyu, a consulting firm focused on cross-functional management and community development. Previously, Thomas held several roles at Thales, including Head of DevOps & IT and Inner Source Transformation Lead. Since joining the Eclipse Foundation in February 2024, Thomas has been the Program Manager for Development Tools, overseeing the Cloud Dev Tools and Open VSX working groups as well as Eclipse IDE.



LinkedIn

A New Generation's Chance to Shape the Future: Building the Next Great Java IDE in Open Source

The Eclipse IDE has been a key platform for Java developers and a pillar of the open source community for over two decades. It has helped countless developers build applications, from enterprise software to experimental projects, supporting Java's ecosystem growth. But like all technology, it must evolve to stay relevant. Enter "Initiative 31": a bold step toward reimagining what an IDE can be in the 21st century.

In 2025, Eclipse IDE begins an ambitious modernisation journey. Java developers have a unique opportunity to help shape the future of this essential tool. Whether you are an experienced developer or new to the ecosystem, Initiative 31 aims to make Eclipse an OS-agnostic IDE. The primary goal is to reduce maintenance costs across operating systems while ensuring a future-proof solution for developers worldwide.

This article explores the initiative's technical details, the SWT modernisation process, and how you can contribute to this evolution.

The Eclipse IDE and the Rich Client Platform: Two Sides of the Same Coin

Before exploring Initiative 31, it is important to distinguish between the Eclipse Platform and the Eclipse IDE.

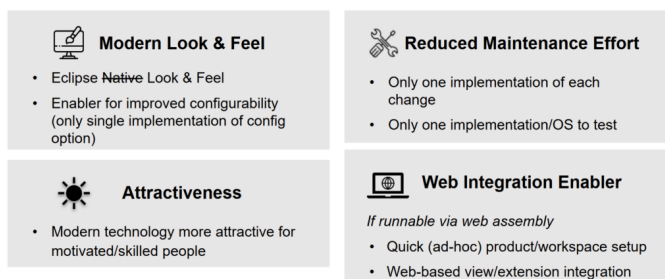
The Eclipse Platform is a set of frameworks and tools designed for both IDEs and rich client applications (RCP), developed mostly in Java. With core services like SWT (Standard Widget Toolkit), JFace, and Equinox, it supports the development of desktop applications beyond traditional IDEs. This versatility has made it valuable in many industries like automotive, space, healthcare systems, avionics, transportation or embedded systems, where reliable and customizable desktop applications are essential. Eclipse remains strong in sectors needing robust and adaptable tools.

Despite its industrial use, the Java development community has gradually moved away from Eclipse as an IDE. Many developers now prefer commercial IDEs or single-vendor open source solutions. These alternatives are often polished and user-friendly but come with trade-offs. Proprietary or single-vendor tools may offer convenience and streamlined workflows but are tied to specific ecosystems, limiting flexibility. In contrast, Eclipse and its platform provide transparency, community-driven development, and adaptability, ensuring long-term sustainability in a changing software landscape.

The Eclipse IDE is an application built with the Eclipse platform, and it is used primarily for Java development. It includes a code editor, debugger, and integration tools, but stands out for its extensibility. With LSP (Language Server Protocol) support, Eclipse IDE can extend beyond Java to other programming languages. This flexibility demonstrates how the Eclipse Platform remains a strong foundation for development environments and desktop tools.

The Birth of Initiative 31: A Bold Vision for Modernisation

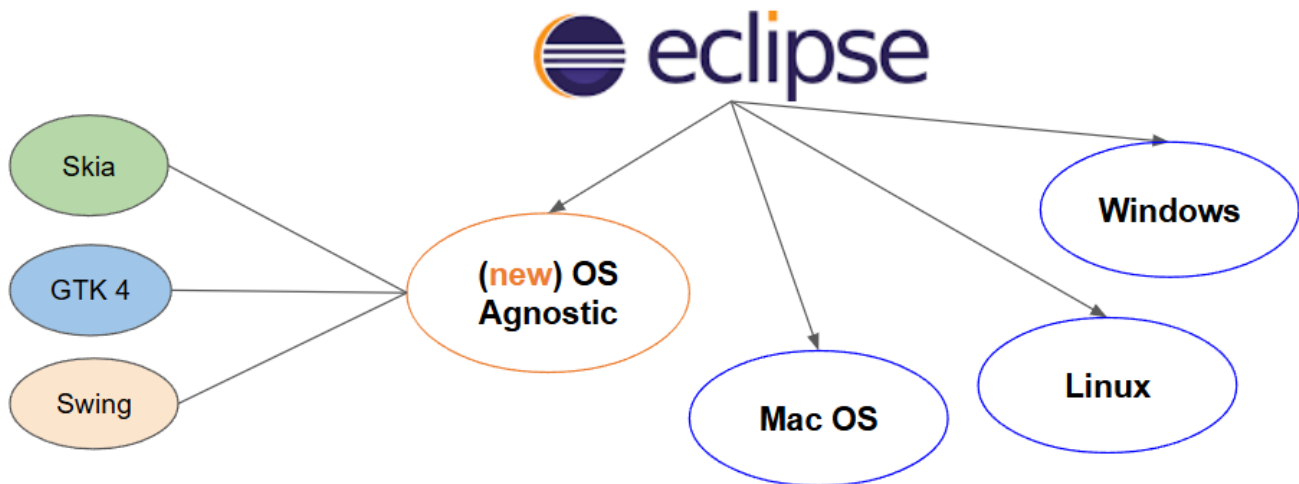
[Initiative 31](#) was launched in 2024 with a clear goal: modernising the Eclipse IDE by exploring three technologies to replace or enhance SWT (Standard Widget Toolkit). SWT is the graphical user interface (GUI) toolkit that underpins the Eclipse IDE. It provides a native look and feel by using the operating system's widgets. This allows Eclipse to integrate seamlessly across platforms while offering better performance than other GUI toolkits. However, maintaining SWT across multiple operating systems has become complex, increasing the need for modernisation. Initiative 31 aims to create an **OS-agnostic** version, reducing maintenance complexity and costs while improving adaptability and scalability, making sure it is compatible with the existing API



The initiative is led by Heiko Klare, Technical Lead of the project at Vector Informatik, a key member of the Eclipse IDE Working Group eclipseide.org. His leadership has been crucial in evaluating new technologies and making key decisions for Eclipse's future. Under his guidance, the team has focused on assessing various options to modernise SWT.

At the outset, the project focused on evaluating three technologies:

- **Skia** – A modern 2D graphics library known for its smooth rendering and performance, particularly favored for cross-platform solutions like **Flutter**.
- **Swing** – A classic toolkit that had served Eclipse for years but was in need of an update to stay relevant.
- **GTK4** – A cross-platform GUI toolkit, particularly suited for Linux environments, but offering potential for broader OS compatibility.



The name “Initiative 31” reflected the plan: test these three technologies throughout 2024 and narrow down the options. The vision was to choose one to lead the future of Eclipse’s graphical user interface (GUI). However, as the project progressed, it became clear that **Skia** and **GTK4** were the most viable options.

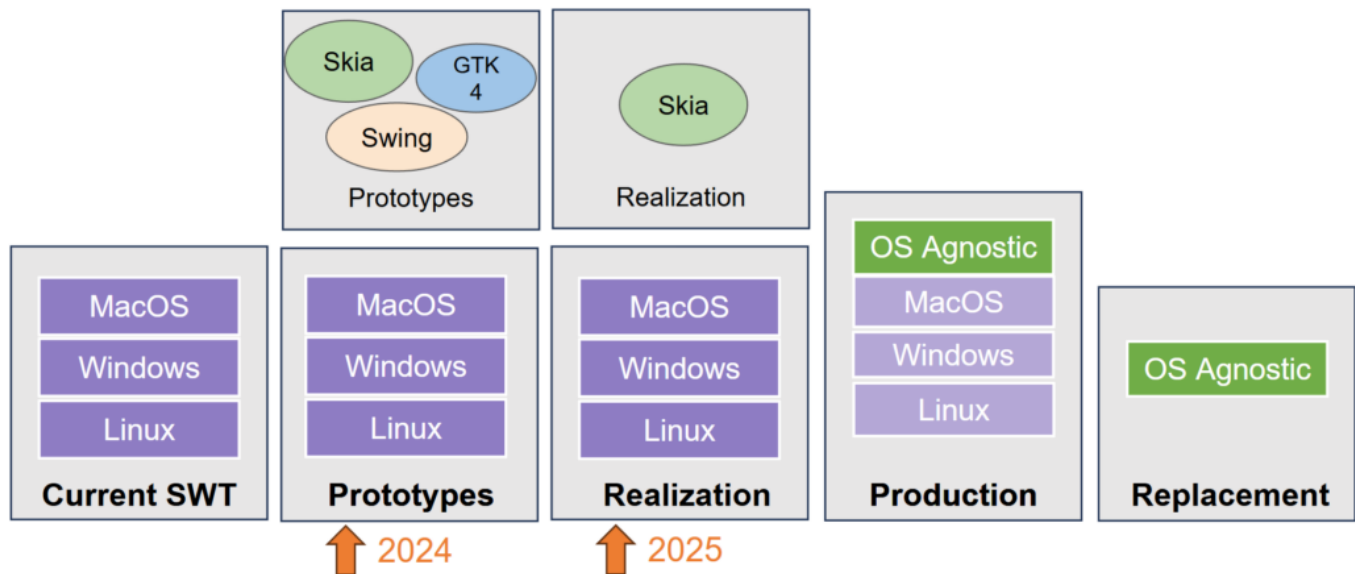
The 2025 Reality: Focus on Skia

By 2025, Initiative 31 has shifted its focus. **Skia** has emerged as the primary choice for modernising Eclipse IDE’s user interface. Skia has become the leading option due to its modern graphics capabilities, cross-platform scalability, and performance benefits. It offers a smoother, more responsive UI.

Given that **GTK3** will no longer be supported on Linux starting in 2025, the **Eclipse Platform** must support **GTK4** moving forward, regardless of which option ultimately becomes the primary solution in the Initiative 31 project. Anyway, as always, in the spirit of **open source**, the future direction of Eclipse IDE will depend on the contributions and decisions of the community.

Revamping the Core: SWT Modernisation Roadmap

The roadmap for Eclipse IDE’s modernisation started with three mature and stable implementations of **SWT**. In 2024, the team began prototyping these three technologies, and as we move into 2025, it’s clear that the focus has shifted primarily to **Skia**. The team is now in the realization phase for Skia, with the ambition to create an **OS-agnostic** implementation. For the time being, this means there will be four implementations: the existing native ones for **Linux**, **Windows**, and **Mac OS**, alongside the new OS-agnostic solution. The ultimate goal is to eventually consolidate all these implementations into a single one, rendering the native versions obsolete:



However, when this happens depends on the community right now... This is where the opportunity to contribute becomes most exciting: **you** could be part of making this vision a reality, starting now!

Opportunities for Incremental Process with Skia

One key advantage of the Skia approach in this context is its ability to be integrated incrementally into the Eclipse IDE. Unlike traditional “big bang” approaches, where everything changes at once, Skia’s architecture supports gradual adoption of new rendering techniques and widgets. This allows Eclipse IDE to modernise without disrupting workflows or causing compatibility issues for existing users.

Skia’s integration is designed to be non-disruptive, enabling a smooth transition. This incremental approach makes it a strong choice for modernising the Eclipse UI.

Integrating the Ability to Exchange Renderer for Rendering Tasks

One key feature of Skia is its ability to swap out the renderer used by Eclipse. This allows Skia to handle rendering while maintaining native rendering for other tasks. This flexibility lets developers gradually transition to Skia’s rendering capabilities without rewriting the entire UI. It is especially useful for performance-critical or custom-drawn content, such as GEF diagrams, where Skia’s rendering performance provides a clear advantage.

Incremental Replacement of Native Widgets with Custom-Drawn Widgets

Skia’s modular approach allows for the gradual replacement of native widgets with custom-drawn versions. While this may reduce performance when using native custom rendering (e.g., Graphics Context or GC), it enables a smooth transition away from legacy widgets. This avoids a “big bang” migration, allowing Eclipse IDE to evolve one widget at a time without disrupting users.

However, replacing native widgets with custom-drawn ones can create a lock-in effect. This makes it harder to maintain a mix of native and custom widgets in complex UI layouts. Integration can become challenging, especially when combining different rendering methods.

Despite these challenges, the incremental approach with Skia remains a viable path forward. It enables Eclipse to modernise gradually while minimizing disruption. This allows for steady improvements in user experience and performance without the risks of a sudden, large-scale transition.

The March 2025 Eclipse IDE Release: Performance Improvements and Community Contributions

The March 2025 [release](#) of Eclipse IDE is released, marking a significant milestone in the platform’s evolution. Alongside Initiative 31, Eclipse IDE has seen major improvements over the past two years, particularly in performance. Performance

issues have long discouraged some users, but recent optimization efforts have led to noticeable gains in speed and responsiveness.

The Eclipse community has continued modernising the platform, with contributors in 2023 and 2024 playing a key role in these improvements. The March 2025 release reflects these efforts, introducing optimizations and new features.



This version is the result of years of work, with the number of contributors reaching 271: a notable increase from the previous average of around 200. This growth in contributions highlights stronger community engagement, reflected in the range of enhancements in this release.

New Features and Enhancements in the March 2025 Release

Some of the key highlights of the March 2025 release include:

- **Enhanced Light Theme** – A new, lightweight tab design and a flat look improve the overall user experience while maintaining familiar functionality.
- **Edge as Default Browser** – Microsoft Edge replaces Internet Explorer, offering better integration and up-to-date web capabilities.
- **Improved Search Filter** – Updates to the Import/Export and Show View features improve how category names are handled in searches, allowing for more accurate results.
- **Enhanced Code Folding** – Now supports control statements, improving navigation in large codebases.
- **Custom Formatters for Java** – More flexibility in formatting Java primitives and arrays.
- **Stack Traces in Variables View** – Displays exception stack traces directly in the Variables view, simplifying debugging.
- **Quick Fixes and Cleanups** – Updated for Java 21 and beyond, providing automatic fixes for common issues.
- **Supports Java 23** – Full compatibility with the latest Java version, ensuring access to new features and APIs.

The March 2025 release reflects the ongoing progress of Eclipse IDE, incorporating contributions from the open source community. As the Eclipse platform continues to evolve, this update is part of the broader effort to modernise the IDE while maintaining its flexibility.

And AI? GitHub Copilot in Eclipse Marketplace

Concurrently with the March 2025 release, GitHub Copilot, the commercially licensed extension, is now available in the [Eclipse Marketplace](#). This tool has been requested by the community and is now offered as an extension for developers seeking AI-powered code suggestions.

A New Chapter for Eclipse IDE

Initiative 31 marks a turning point for Eclipse IDE, giving developers a chance to shape the future of Java tools. By modernising the platform, Eclipse aims to stay at the forefront of the open source ecosystem. This is an opportunity for Java developers to contribute to a tool that has influenced software development for two decades.

The path ahead presents challenges, but with community effort, Eclipse IDE will continue to evolve. It will remain a key tool for Java developers worldwide.

And beyond... Building a Fully Open Source Ecosystem: Jakarta EE, Adoptium, and MicroProfile

The future of Java development depends on an open source ecosystem that goes beyond simply hosting code on platforms like GitHub. It requires a thriving community that drives real change. Tools, frameworks, and runtimes like Jakarta EE, Adoptium, and MicroProfile form the foundation of this ecosystem. The goal is a fully open, collaborative stack—from development environments to cloud-native frameworks—ensuring transparency, flexibility, and inclusivity.

Jakarta EE brings enterprise-grade solutions to the cloud, while MicroProfile enhances it with microservices-optimized capabilities. Adoptium keeps Java runtimes high-performance and open, meeting the needs of developers worldwide. This vision extends beyond code; it fosters a global ecosystem shaped by collective efforts. Open source is more than a model—it is a movement essential for solving today's complex challenges. By adopting these tools and contributing to their growth, we help build a more sustainable, flexible, and impactful future for technology and society.

A Call to Action: How Java Developers Can Contribute

While the technical roadmap is promising, none of this will be possible without the support and contributions of the Java development community. Whether you're an experienced contributor or someone looking to get involved for the first time, Initiative 31 offers a multitude of ways to make an impact.

- **Code Contributions:** If you're familiar with Java development and have experience working with SWT, Skia, or GTK4, this is a perfect opportunity to contribute your expertise to the project.
- **Bug Fixes and Feature Requests:** Every developer can contribute by reporting bugs, submitting feature requests, and testing new features.
- **Documentation:** With such a large and complex project, clear documentation is critical. You can contribute by writing tutorials and guides or improving existing resources for new users.
- **Application Compatibility Testing:** Developers who maintain **RCP-based applications** are encouraged to test compatibility as the project evolves. This ensures a smooth transition and helps identify potential issues early. It's also an opportunity to explore a possible path toward a **new generation of tools without having to rewrite everything from scratch**.

The **source code** is available on [GitHub](#). The **technical team holds weekly meetings**, and the **chat room is open to everyone**. Feel free to join, share your ideas, and contribute your thoughts to the project. Your input can help shape the future of Eclipse IDE.

If you're **new to Eclipse projects**, you are invited to start with the **entry point for new contributors** at [GitHub](#). This space is dedicated to answering your questions and helping you get started. Even better, the **Eclipse IDE Working Group provides "Community Mentors"**: experienced contributors ready to guide you through your first steps in the project.

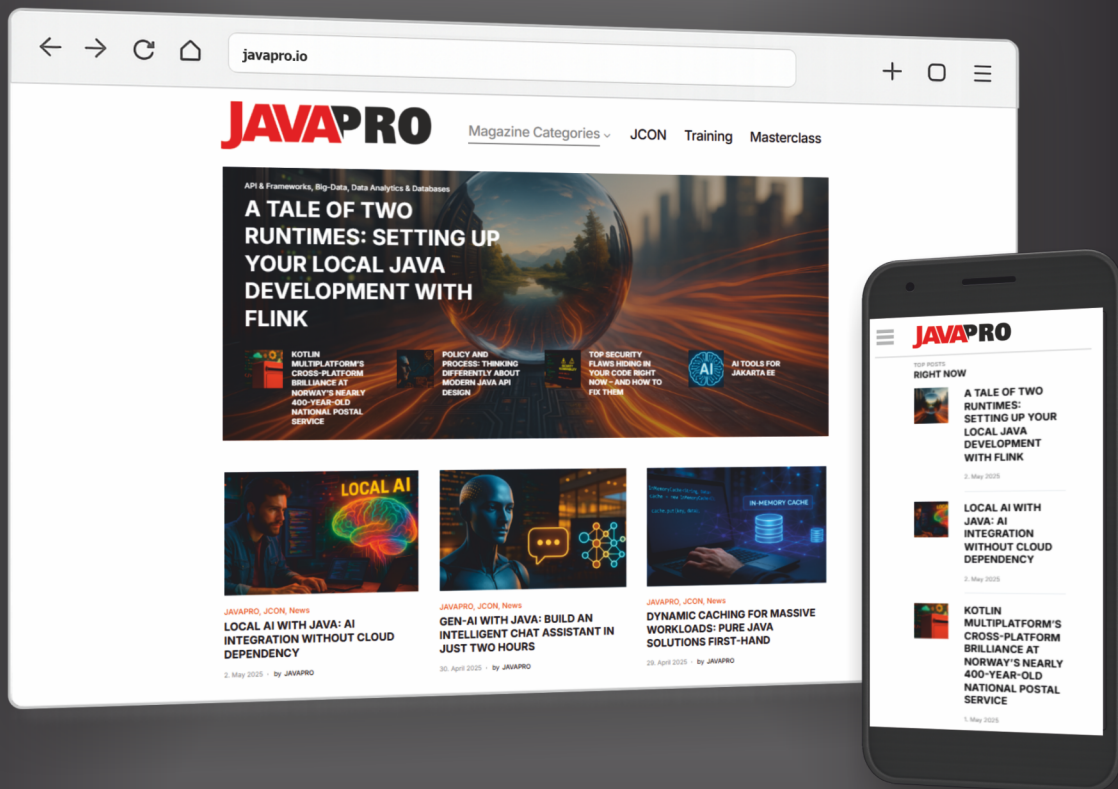
› [Back to the Table of Contents](#) ‹

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



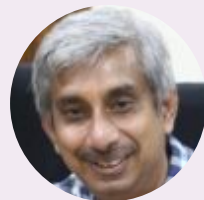
Breakpoints – The Lifeline of Debugging – An Eclipse Features Guide

Debugger

Eclipse

Author: Manoj Nalledathu Palat

Manoj is an Open Source Lead/Committer working at IBM leading the Java Compiler in Eclipse. In parallel, Represented Eclipse Foundation in the Experts Group for Platform JSRs for multiple Java SE and IBM in Eclipse IDE WG Steering Committee. Passionate about training (Worked as Professor of Practice from January 2024-Dec 2024 at NIT, Calicut), Provide training internally at IBM regularly, blogs sometimes on [medium](<https://medium.com/@manojnp>), mentors people. Holds a Masters degree in Computer Science from Indian Institute of Science(IISc), Bangalore and B-Tech in Computer Science from National Institute of Technology (Formerly REC) Calicut, India.

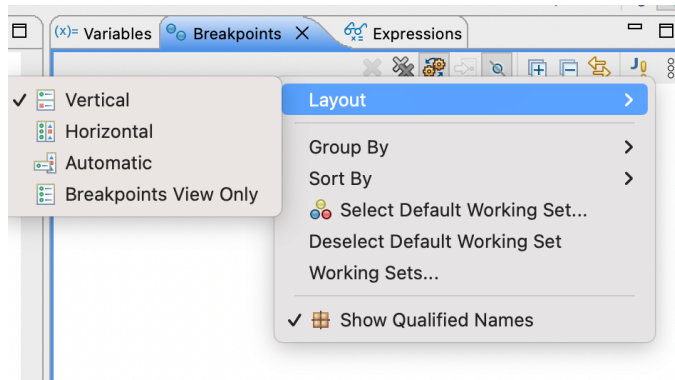


LinkedIn

Each and Every programmer would have got stuck with some debugging issue or the other so many times. And would have stopped the program at a breakpoint. And here is a detailed look at breakpoints.

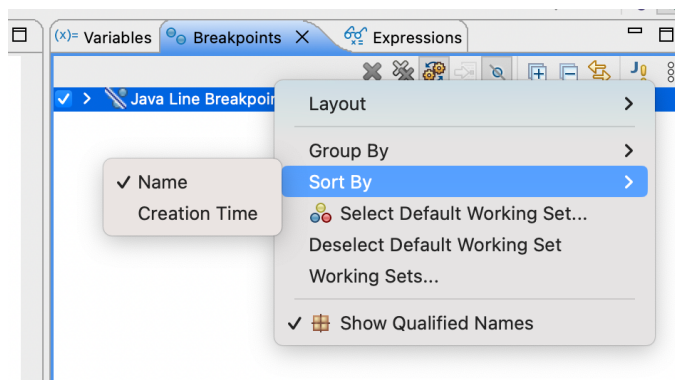
As an Eclipse Committer, I have shown the Eclipse IDE pictures and have explained from the Eclipse IDE point of view. Every time we want to inspect a variable, see the stack, and what not, we need to break the program at some point — and hence we require breakpoints. There are a whole lot of features with breakpoints, let us start with the basics:

Let us start with the layout — this is just how we view this tab.

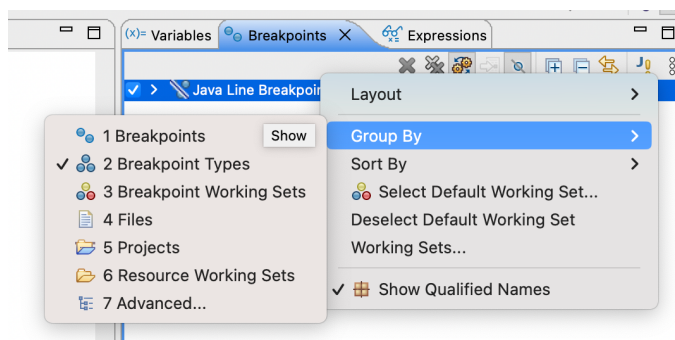


The division is between the breakpoints and the relevant details — if we just want the breakpoints, select “Breakpoints View Only”. Will come to what the details contain in a minute.

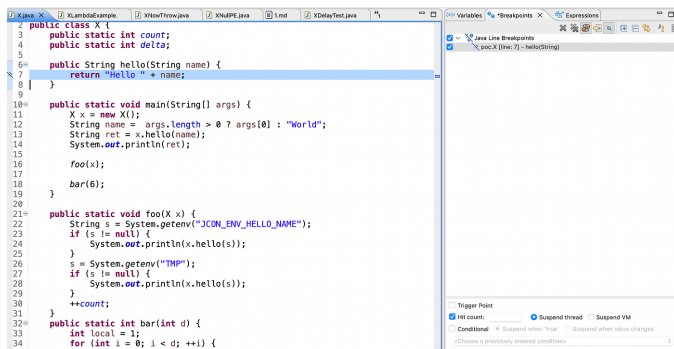
Before we start adding breakpoints, adding that we can sort them by Name or Creation time ->



Let us also Group the breakpoints by type –

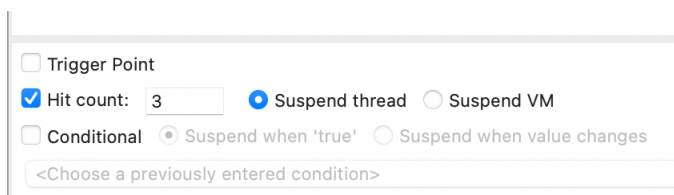


We can look at other options later.. We still haven't come to breakpoints, so let us put one breakpoint:



Multiple things to observe here — This breakpoint now comes under the type “Java Line Breakpoints” -> this is a breakpoint at a line.. aren’t all breakpoints a line break point — yes, all “explicit” breakpoints are line breakpoints — will become clear as we move one. We cannot anyway put a breakpoint explicitly beyond a line granularity. So one best practice is to “Not” to overcrowd your line.

Options:



Take a look at the “Hit Count” in the above — Stop when you see the elephant for the third time — this just says stop after so many hits –

Now lets look at “Conditional” — Let’s say don’t stop when you see a peacock, see when you see a dancing peacock. In our case, stop when i is even:

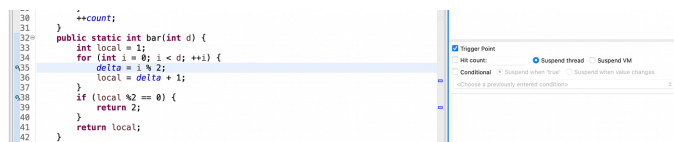


An interesting option is the sub-option of “stop when the value changes” ->



Keep that it will stop when the condition changes from true to false and vice versa. In this particular case, this degenerates to stopping at every hit since this condition alternates.. that’s how we beat the system :).

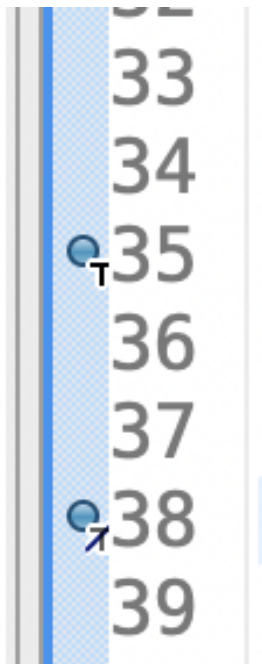
Trigger Points – The First Line of Defence:



We can enable a break point as a trigger point — Only if that break point is hit, other break points will be enabled.

What happens if there are group of trigger points? If any one of them is hit, the rest of the breakpoints are enabled and all the trigger points are disabled. Its like trigger points being the first line of defence.

How do you recognize a trigger point ? One with a T

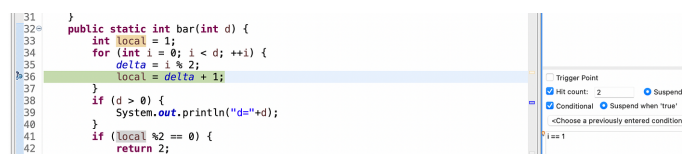


One with a T being cut is a breakpoint enabled once the trigger is hit.

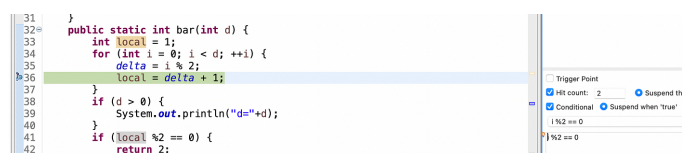
Now let's play a game: How about the combination of these — namely — Trigger Point, Condition, and Hit Count. Let's Look at these:

Trigger point + condition -> The condition has to be true

Hit Count + Condition -> Count is incremented only for condition? Take a guess..



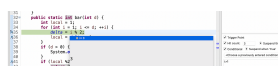
At line 36, we want the program to stop when both the conditions are true — ie Hit count should be 2 and the i should be equal to 1. it stops, since there is no conflict here.. Now let us change the condition to `i%2 == 0`, ie i is even.



When will this stop?

In fact, it will not stop. Hit count will be incremented anyway. So if it has to stop, the condition has to be true at Hitcount == 2. So this is an && relationship.

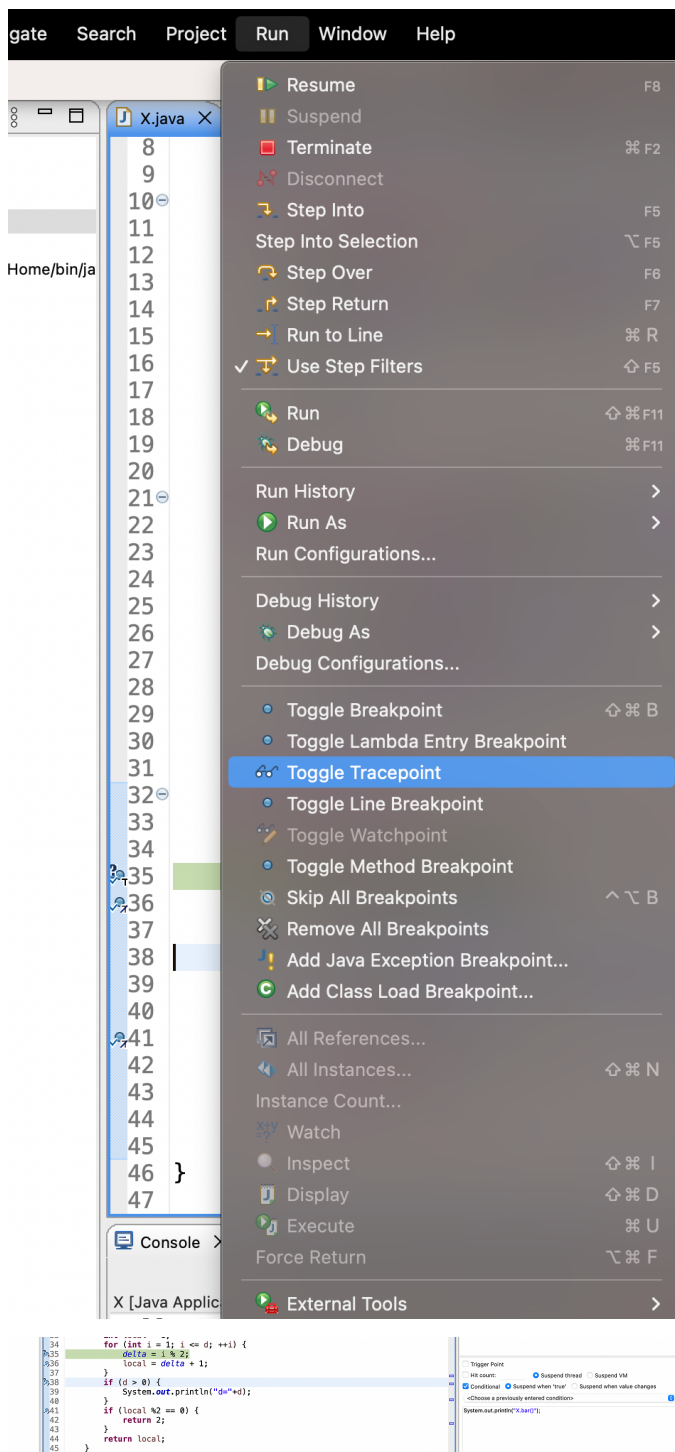
Now if you put all the three on, then again its an && condition.



Note: I did find some inconsistent behaviour, trying to reproduce. If you hit upon an erroneous behavior please file a bug at <https://github.com/eclipse-jdt/eclipse.jdt.debug/issues>.

Tracepoints:

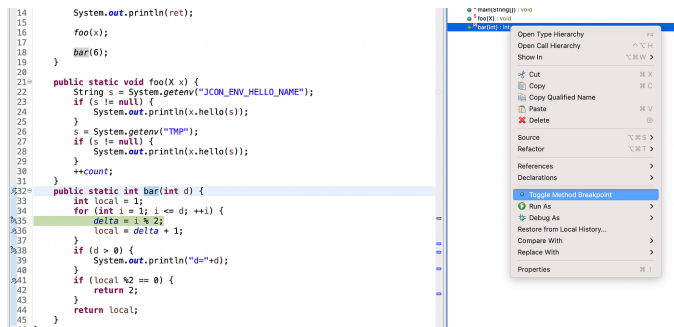
Put a breakpoint without really breaking there! — Tracepoints



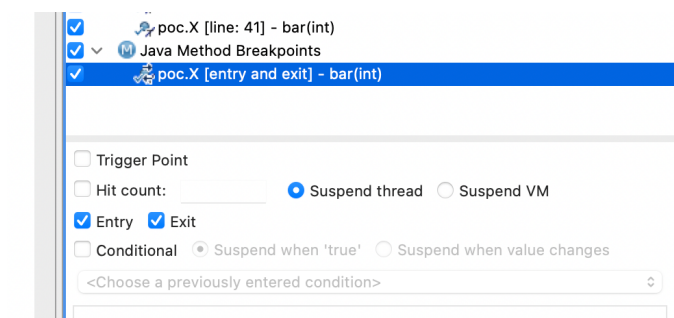
Essentially they are conditional breakpoints which does not have a proper condition — Here it just prints out the value — we just want some print statements without disturbing the code.

Method Breakpoints – Entry/Exit

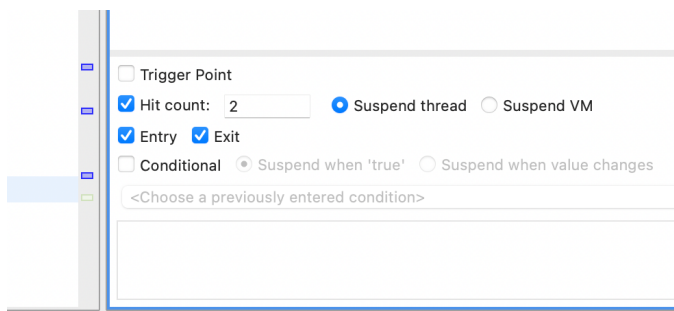
To put a method entry/exit breakpoint, just go to the outline view and say “toggle Method Breakpoint”



This puts a breakpoint in the entry of method. We can also go to the line number of the method declaration and add a breakpoint. How can we put exit breakpoints?



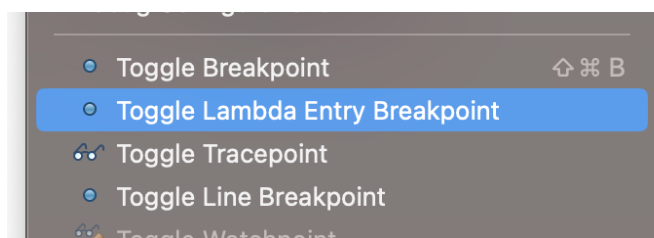
Click on Exit. This will put at all the return statements — in the case above, at line numbers 42 and 44 for method bar().



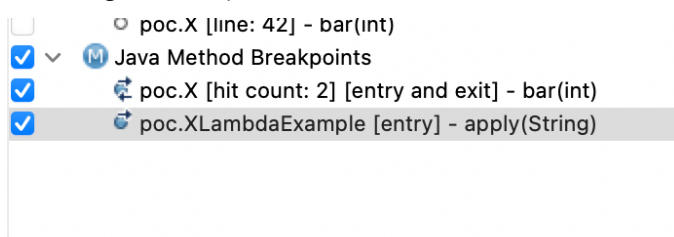
We can also put hit count here — in this case 2, only the second entry will be stopped.

Lambda Breakpoint

From the Run menu, click on “Toggle Lambda Entry Breakpoint”



This will get a breakpoint similar to Method one



Watchpoints:

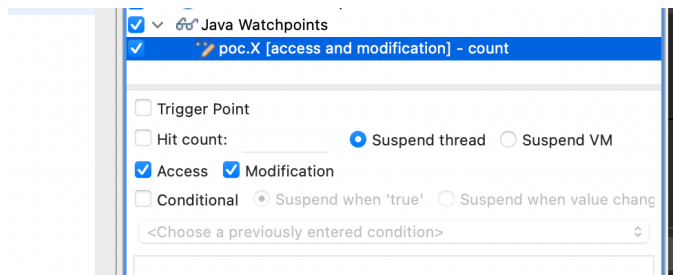
Click on fields and put a breakpoint —

```

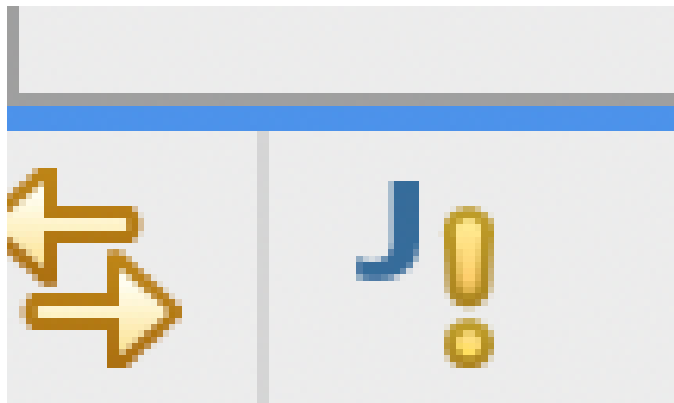
2 public class X {
3     public static int count;
4     public static int delta;
5 }

```

A watchpoint gets enabled



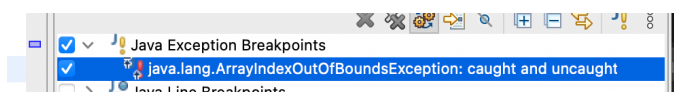
This has two fields — Access and Modification. Access is for reading and Modification for writing — Enabling them respectively will stop at read and write resp.

Exception Breakpoints

Click on the above icon to set breakpoints for Exceptions —



Here we add for AIOOBE.



And it has two options — caught and uncaught. Let's look at them

```

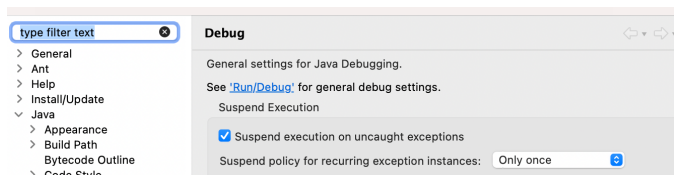
1 package poc;
2
3 public class XNowThrow {
4
5     public static int catchAI00BE(int[] a) {
6         try {
7             a[a.length] = 10;
8         } catch (ArrayIndexOutOfBoundsException e){
9             return -1;
10        }
11        return 0;
12    }
13
14    public static int throwAI00BE(int[] a) {
15        if (a != null) {
16            a[a.length] = 10;
17        }
18        return 0;
19    }
20
21    public static void main(String[] args) {
22        catchAI00BE(new int[5]);
23        throwAI00BE(new int[5]);
24    }
25 }
26

```

line 7 — caught

line 16 — uncaught.

If there is an issue, check the master preferences



Make sure that this setting is as desired.

Another interesting option is the policy for recurring exceptions — we can set this to stop only for the first instance.

Helpful NullPointerException

Though not about breakpoints, this is a recent feature which tells which variable was null — for eg, in the following code:

```

1 package poc;
2
3 public class XNullPE {
4
5     private static int foo(Object o, Object a) {
6         return o.hashCode() + a.hashCode();
7     }
8     public static void main(String[] args) {
9         foo(new XNullPE(), null);
10    }
11 }
12

```

the console output gives:

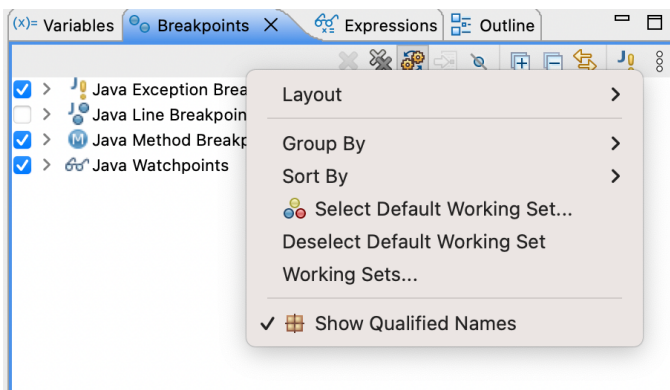
```

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Object.hashCode()" because "a" is null
at poc.XNullPE.foo(XNullPE.java:6)
at poc.XNullPE.main(XNullPE.java:9)

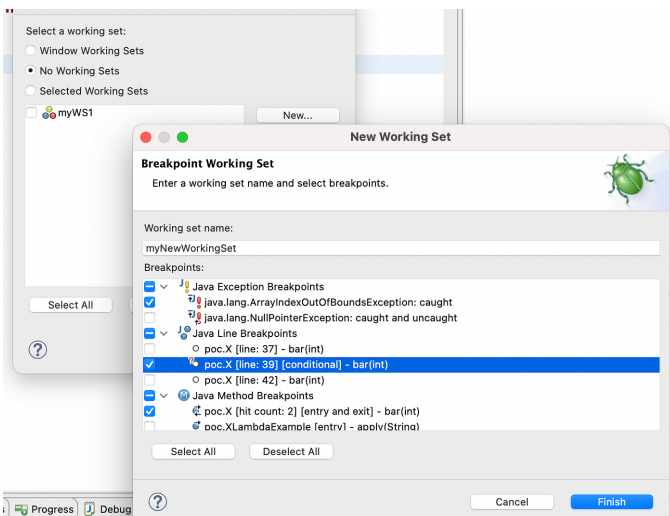
```

mentioning that “a” is null, not leaving the programmer to guess which variable was null.

Working Sets for Breakpoints:

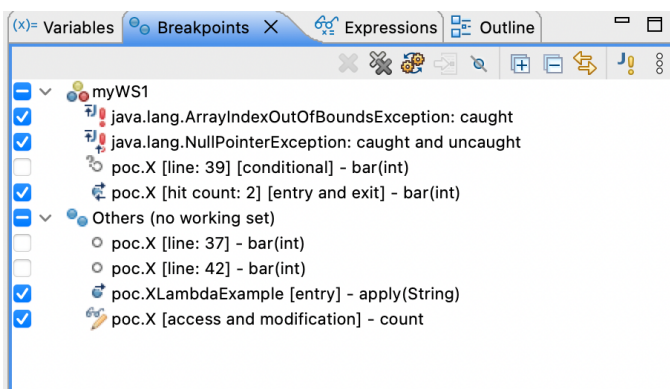


click on the three vertical dots to get the menu below



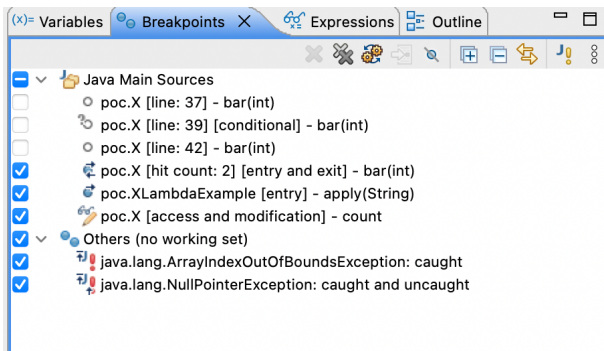
Now we can create and add the breakpoints to a working set. And then what? Where should we use this?

Now we can group by working sets:

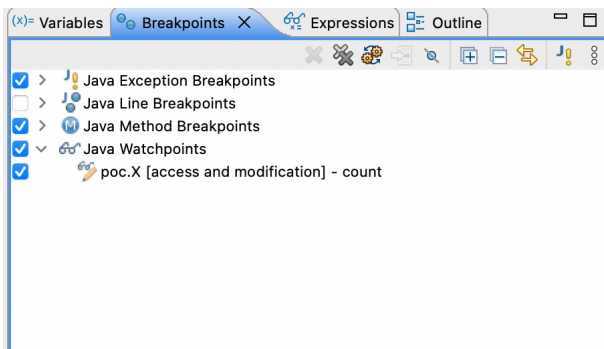


Others goto a default working set.. Enable/Disable can be done at a working set level

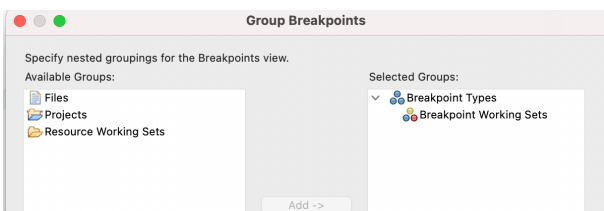
We can group by resource working sets as well



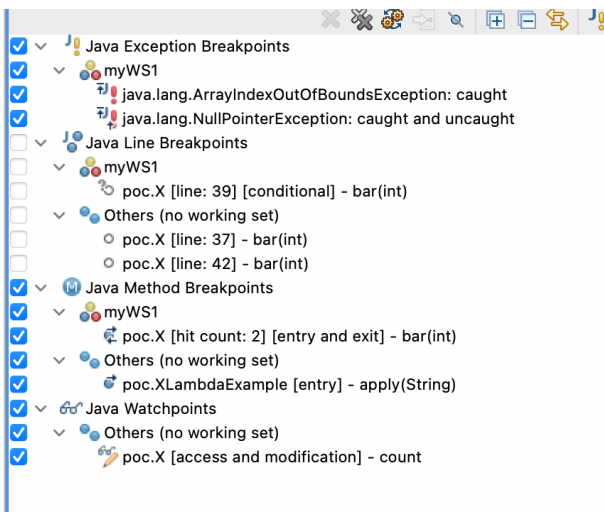
We can choose as per convenience. A common option is by type:



More “complicated” configurations also can be specified:



And we can view it as:



I guess its time to stop about the organization/grouping of breakpoints

We can of course export and import breakpoints — helps in sending this to colleague, for example for working together. So much for the breakpoints – Happy Debugging!

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



Observability Landscape in Java

Debugger # DeveloperTools # Observability # Tracing

Author: Vipin Menon

He's a software professional with over 13 years of experience in the industry, specialising in a diverse range of technologies, including Java, React, Node.js, JavaScript, and various web and cloud technologies. Currently, Vipin is contributing to the design and development of Application Performance Monitoring for Instana at IBM. A passionate advocate for open-source software, he has made significant contributions to projects such as OpenJDK, Node.js, and Swift. He played a pivotal role in implementing key security algorithms in IBM Java 11 as part of the integration of TLS 1.3 within the security module. With a strong focus on innovation, he holds six patents, one publication, and shares his insights through numerous blogs on technology topics, featured on platforms like Medium and other blogging platforms.



LinkedIn

Java has become a well-established and widely adopted language. As software systems became more complex with the shift to microservices, Java observability tools evolved to meet new demands. It has evolved to address these new challenges, offering improved capabilities for monitoring, tracing, and debugging. This article explores the evolution of Java observability and its adaptation to modern cloud-native solutions. Lets discuss how they have shaped into the landscape of observability in Java applications today.

First Generation: Standard Debugging

When Java was first released in 1995 the tools for observing and debugging applications were minimal. The developers could only rely on standard system calls, debugging tools, and performance monitoring frameworks.

Standard Output

The most straightforward way to check what was happening inside a Java application was using `System.out.println`. The concept was simple to implement and understand.

```
1 class Demo {
2     public static void main(String[] args)
3     {
4         System.out.println("Welcome to JavaPro");
5     }
6 }
```

```
1 $java Demo
2 Welcome to JavaPro
```

It quickly became impractical for larger applications or production systems. It made the code messy and hard to maintain.

Java's **exception-handling mechanism** which automatically generates stack traces when exceptions occur, was a key feature of the language from the very beginning. When a Java application throws an exception, the Java Virtual Machine (JVM) generates a stack trace that provides a detailed report of the method calls leading up to the exception, including the exact line numbers where the error occurred. This was a significant advantage over many other programming languages at the time, as it allowed developers to easily trace the root cause of errors without needing to manually track down the source of the issue.

```
1 Exception in thread "main" java.lang.ArithmeticException: / by zero
2     at Demo.main(Demo.java:5)
```

This don't provide real-time insights and was not sufficient for monitoring complex systems.

Logging

Logging in Java was first introduced with the inclusion of the `java.util.logging` package. The primary goal of `java.util.logging` was to allow developers to log messages at different levels (`INFO`, `WARNING`, `ERROR`) and to control the output such as to the console or to log files.

```
1 Mar 09, 2025 8:26:13 PM Demo main
2 INFO: This is an INFO message
3 Mar 09, 2025 8:26:13 PM Demo main
4 SEVERE: This is a SEVERE message
```

At this stage, developers primarily write text-based logs to trace the application state and errors manually. Hence, the external libraries like Log4j, SLF4J, and Logback became more popular due to their enhanced features and flexibility.

Profilers

jstat (JVM Statistics Monitoring Tool)

This tool provides users real-time JVM statistics. It collects various performance metrics like GC, memory usage and class loading.

```
1 # Usage:
2 # JIT compiler performance
3 # jstat -compiler <process id> <interval>
4 #
5 % jstat -compiler 95150 10000
6 Compiled Failed Invalid Time FailedType FailedMethod
7      3449      1      0    1.44          1 jdk/nio/zipfs/ZipFileSystem iteratorOf
```

jstack (Java Stack Trace Tool)

jstack extracts thread dumps from a running JVM. It helped in diagnosing issues like deadlocks, blocked and long-running methods in a multi-threaded application.

```
1 # Usage:
2 # thread dump of a running process compiler performance
3 # jstack <process id>
4
5 % jstack 95150
6 Full thread dump OpenJDK 64-Bit Server VM (21.0.3+9-LTS mixed mode, sharing):
7 ..
8 }
9
10 "main" #1 [8451] prio=5 os_prio=31 cpu=206.64ms elapsed=12808.27s
11 ..
12   at java.util.concurrent.locks.LockSupport.park(java.base@21.0.3/LockSupport.java:221)
```

jinfo

jinfo provides information about the JVM configuration and runtime environment.

```
1 # Usage:
2 # Display JVM Properties
3 # jinfo <process id>
4
5 % jinfo 95150
6 Java System Properties:
7 file.separator=/
8 ...
9 java.vm.specification.name=Java Virtual Machine Specification
10 ..
11 path.separator=:
12 stdout.encoding=UTF-8
```

These builtin tools were very handy and useful in basic monitoring and troubleshooting. However, they had limitations when it comes to complex applications.

The second generation built-in observability tools in Java typically refers to tools that offer more advanced features and improved monitoring capabilities compared to the first. These tools are more focused on performance analysis, real-time monitoring, diagnostics and profiling with enhanced capabilities for production environments.

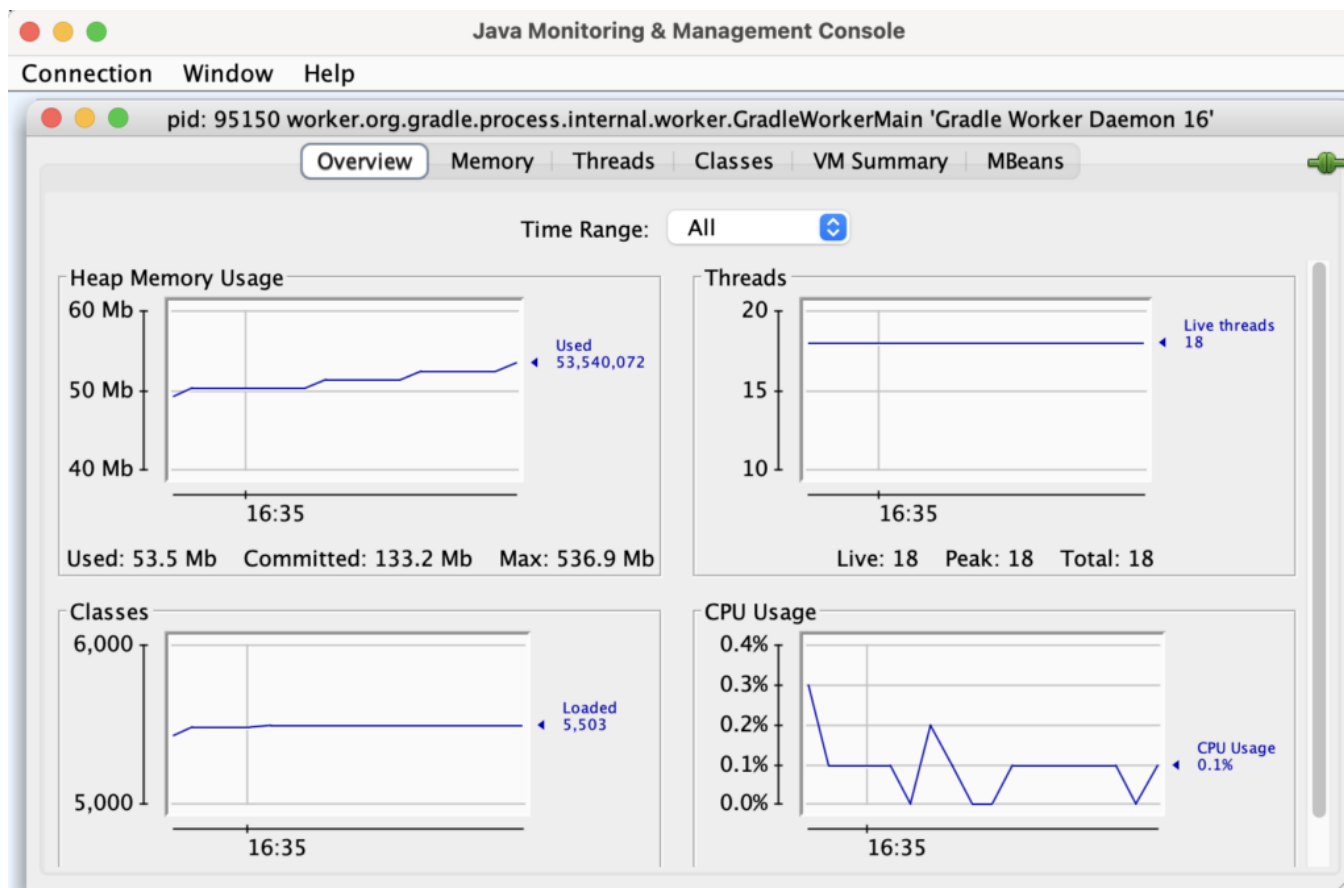
Second Generation: Focused on Production

The second generation built-in observability tools in Java typically refers to tools that offer more advanced features and improved monitoring capabilities compared to the first. These tools are more focused on performance analysis, real-time monitoring, diagnostics and profiling with enhanced capabilities for production environments.

jconsole:

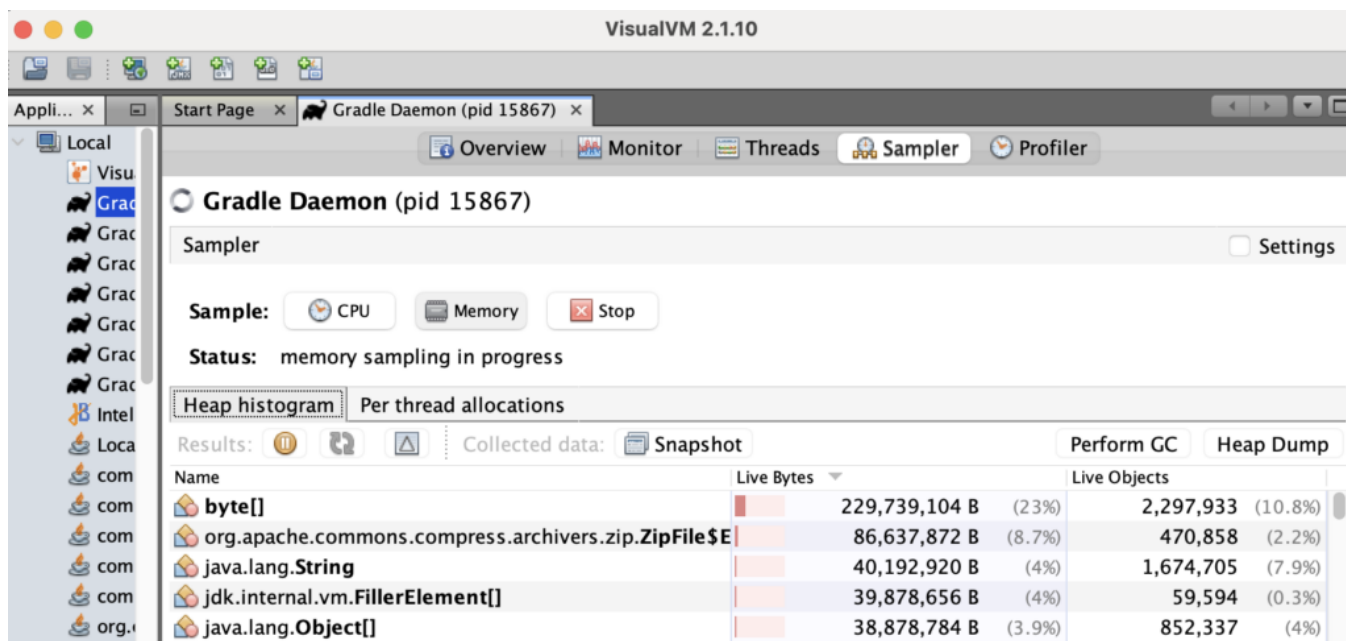
jconsole provides information about performance and resource consumption of applications running on a JVM. It complies to the JMX specification. You can connect to either a local or remote process. It helps capture memory monitoring, thread monitoring and information about registered mbeans.

```
1 # jconsole usage
2 # jconsole <process id>
3 # jconsole 95150
```



jvisualVM

jvisualVM provides a more detailed analytics on JVMs internal behavior and can even detect memory leaks by providing an extensive analysis on heap. JVisualVM is no longer comes with the standard JVM installation and comes separate.



Java Flight Recorder

Java Flight Recorder is a tool that is capable of collecting both diagnostic and profiling data from a running java application. It has low performance overhead. It can collect data on events, and is even capable of providing black box analysis.

```
1 % java -XX:+FlightRecorder -XX:StartFlightRecording=duration=60s,filename=myrecording.jfr
   Demo
2 OpenJDK 64-Bit Server VM warning: Option FlightRecorder was deprecated in version 13.0
   and will likely be removed in a future release.
3 [0.190s][info][jfr,startup] Started recording 1. The result will be written to:
4 [0.190s][info][jfr,startup]
5 [0.190s][info][jfr,startup] /Users/vipinmenon/Downloads/myrecording.jfr
6 The application is running. Press Ctrl+C to stop.
```

Third Generation: Cloud Native Observability

The rise of cloud-native architectures, containerization, and orchestration has marked the last decade. These technologies introduced new complexities into Java application observability. Developers required a more dynamic and decentralized approach to keep up, even though the inbuilt tools remained useful. This led to the rise of modern APM tools like Instana, DynaTrace and New Relic that could automatically instrument java applications to track their performance.

Distributed tracing was another revolutionary approach, enabling the correlation of logs, metrics, and traces across distributed systems. Consequently, this provided deeper insights into application health, performance and improved troubleshooting capabilities.

Modern tracing also introduced an open source framework for collecting, processing and exporting telemetry data including logs, metrics and traces. [OpenTelemetry's Java SDK](#) allows developers to easily instrument their applications for distributed tracing and metrics collection. This has made observability more standardized, simplifying the process of collecting telemetry data from Java applications running in cloud-native environments.

A Java agent attaches itself to any Java application and automatically collects the data without requiring any code changes to the application. Below steps demonstrates the configuration of [open telemetry java agent](#).

1. Download the open telemetry java agent.

```

vipinmenon@192 java-simple % curl -L -O https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/latest/download/opentelemetry-javaagent.jar

% Total    % Received % Xferd Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
  0     0    0     0    0     0      0      0      0      0      0      0      0
  0     0    0     0    0     0      0      0      0      0      0      0      0
100 21.2M 100 21.2M    0     0 8245k    0  0:00:02  0:00:02 --:--:-- 15.1M

```

2. Set the agent and environment variables.

```

vipinmenon@192 java-simple % export JAVA_TOOL_OPTIONS="-javaagent:/Users/vipinmenon/Public/workspace/TestCases/opentelemetry/java-simple/build/libs/java-simple.jar" \
    OTEL_TRACES_EXPORTER=logging \
    OTEL_METRICS_EXPORTER=logging \
    OTEL_LOGS_EXPORTER=logging \
    OTEL_METRIC_EXPORT_INTERVAL=15000

```

3. Start the application

```

vipinmenon@192 java-simple % java -jar ./build/libs/java-simple.jar
Picked up JAVA_TOOL_OPTIONS: -javaagent:/Users/vipinmenon/Public/workspace/TestCases/opentelemetry/java-simple/build/libs/java-simple.jar
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
[otel.javaagent 2025-03-31 09:10:29:210 +0530] [main] INFO io.opentelemetry.javaagent.tooling.VersionFile: /Users/vipinmenon/Public/workspace/TestCases/opentelemetry/java-simple/build/libs/java-simple.jar
2025-03-31T03:40:30.418Z INFO 'Starting DiceApplication using Java 21.0.3 with PID 60010 (/Users/vipinmenon/Public/workspace/TestCases/opentelemetry/java-simple/build/libs/java-simple.jar started by vipinmenon in /Users/vipinmenon/Public/workspace/TestCases/opentelemetry/java-simple)'
000000000000 [scopeInfo: otel.DiceApplication:] {}
2025-03-31T09:10:30.418+05:30 INFO 60010 --- [main] otel.DiceApplication
10 (/Users/vipinmenon/Public/workspace/TestCases/opentelemetry/java-simple/build/libs/java-simple.jar)
TestCases/opentelemetry/java-simple)
2025-03-31T03:40:30.426Z INFO 'No active profile set, falling back to 1 default profile: "default"' :
: otel.DiceApplication:] {}
2025-03-31T09:10:30.426+05:30 INFO 60010 --- [main] otel.DiceApplication
e: "default"

```

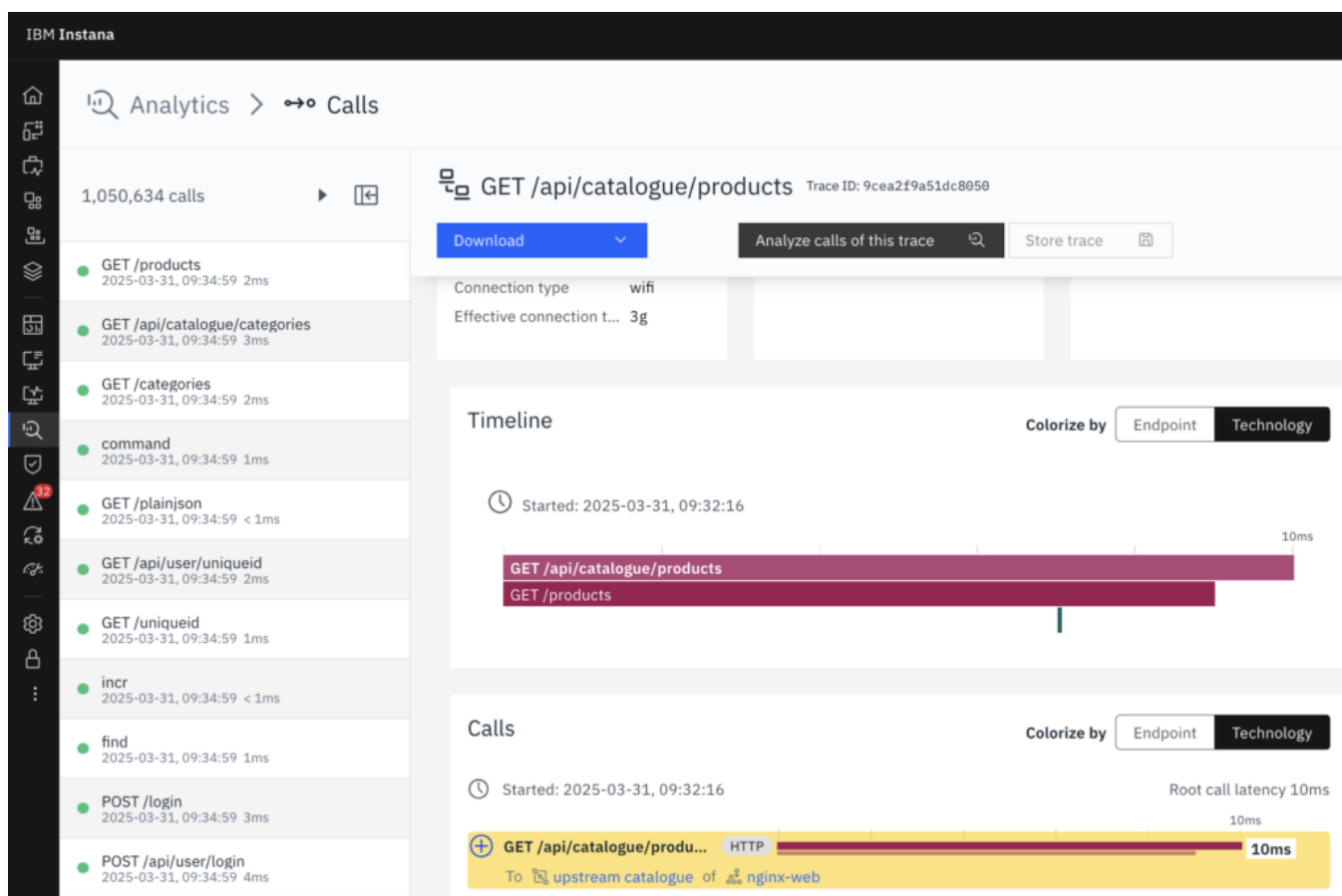
4. Visualize the open telemetry traces from your application

```

2025-03-31T09:11:16.142+05:30 INFO 60010 --- [nio-8080-exec-2] otel.RollController : Anonymous player is rolling the dice: 5
[otel.javaagent 2025-03-31 09:11:16:156 +0530] [http-nio-8080-exec-2] INFO io.opentelemetry.exporter.logging.LoggingSpanExporter - 'GET /rolldice' : c351acd74f1ae5a52cd7f1f9ff4b5bbe7d699df91a1c74d8 SERVER [tracer: io.opentelemetry.tomcat-10.0.2.14.0-alpha] AttributesMap{data={network.peer.port=63554, http.request.method=GET, http.route=/rolldice, server.address=localhost, client.address=0:0:0:0:0:0:1, network.peer.address=0:0:0:0:0:0:1, url.path=/rolldice, url.scheme=http, thread.name=http-nio-8080-exec-2, server.port=8080, http.response.status_code=200, thread.id=43, network.protocol.version=1.1, user_agent.original=curl/8.6.0}, capacity=128, totalAddedValues=14}

```

Open telemetry is introduced to standardize the traces logs. This means it can be fed into any monitoring tools like Instana, dynatrace which is capable of providing a visualization experience with it along with the performance metrics on a single pane of glass.



Since the data format will remain consistent, any tool that supports OpenTelemetry will be able to read and visualize the data. As a result, this interoperability ensures seamless integration across various observability platforms and prevents vendor locking.

The Future of Observability in Java

As Java applications continue to evolve in the cloud-native era several emerging trends are expected to shape the next phase of observability

AIOps (Artificial Intelligence for IT Operations) is making its way into observability. It can analyze telemetry data to detect anomalies, predict failures, and automatically trigger responses to potential issues.

Observability tools integrated with AI could provide insights directly to developers through chat interfaces allowing developers to fix problems in real time.

Machine learning models could always bring concerns with potential risk to data privacy and security. While these trends are expected to grow and bring more convenience, it is crucial to ensure that we collect only the relevant information. Moreover, we must focus on extracting the maximum throughput from the data gathered.

Conclusion

Java observability has come a long way since its inception, and its evolution truly reflects the spirit of enhancing system reliability, performance, and continuous improvement. As cloud-native technologies evolve, Java's observability tools are likely to continue innovating. This ongoing progress will further empower developers to maintain robust, performant, and reliable applications in an increasingly complex world. The future of Java observability is exciting, with AI, machine learning, and predictive analytics shaping the next generation of monitoring tools. These innovations will empower developers to maintain robust, performant, and reliable applications, ultimately ensuring that Java remains a vital part of the software ecosystem for years to come.

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

JCON2026
www.jcon.one

JAVAPRO

APR
20-23



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



Code Wizardry: Casting Spells to Fix Bugs

Analysis

Bugs

Author: Andres Sacco

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



LinkedIn

Multiple developers may introduce changes to develop new features or fix bugs during an application's life. At the same time, these changes could introduce problems in the code, such as bad practices regarding the declaration of attributes, duplicate code, or pieces of code with a high risk of producing errors.

In many cases, this situation could be detected on a peer review of the code for another developer but implies that someone needs to spend too much time focusing on small details instead of focusing on the problem that the code tries to fix. The appearance of tools like PMD, Checkstyle, SonarLint, and many others that detect problems analyzing the code mitigates most of the problem, but let's see in detail the impact on the day-to-day of the company or developer.

Impact of Bugs and Bad Practices

Harmful practices could result in many problems in the application, but not all of them appear together. Each developer may introduce a slight code smell or problem, such as the way to declare variables or not consider the possibility of null values in some attributes, where no one will detect it because it is a part of the application that is not frequently used. Still, when many developers do the same for an extended period, the problem will be visible and have a huge impact, causing all the developers to spend many hours trying to understand why this happened.

Common Consequences

Introduce a series of poor practices, and the impact could trigger a cascade of unintended consequences, including memory leaks that gradually affect the application's performance by consuming more and more resources until the application crashes, experiences unpredictable behavior, or even becomes vulnerable to security threats. Additionally, performance bottlenecks could turn features into nightmares, while logic errors could lead to incorrect data processing or cause the application to break.

Reduced Maintainability

When the code is infested with bugs, it causes problems for the application and makes it significantly more challenging to maintain. Some issues that impact maintainability include spaghetti code, cryptic error handling, and poorly documented fixes, which diminish the developer's understanding of why the code is structured that way.

Imagine an application where a bug related to resource synchronization was fixed with a random *Thread.sleep()* to prevent race conditions. The problem looks to be solved initially, but when the application grows and processes more information, a new issue appears about the time the thread sleeps.

```
1  public class Counter {
2      private int count = 0;
3
4      public void increment() {
5          try {
6              Thread.sleep(100); // Temporary "fix"
7          } catch (InterruptedException e) {
8              Thread.currentThread().interrupt();
9          }
10         count++;
11     }
12
13     public int getCount() {
14         return count;
15     }
16 }
```

Another problem could be not correctly tackling the `NullPointerException`, such as encapsulating in a try/catch block instead of using another structure like `Optional`.

```
1 public class UserService {
2     public String getUserDisplayName(User user) {
3         try {
4             return user.getName().toUpperCase(); // It could throw NullPointerException
5         } catch (NullPointerException e) {
6             return "Unknown User"; // Hides real problem
7         }
8     }
9 }
```

One last example could be a bad comparison in the enumeration to do something depending on the value; in some cases, instead of using the enumeration methods or comparing with the value, some developers invoke a `toString()` method and compare two different strings.

```
1 enum LogLevel {
2     INFO, DEBUG, ERROR;
3 }
4
5 public class Logger {
6     public void log(LogLevel level, String message) {
7         // BAD: Using toString() for comparison
8         if (level.toString().equals("DEBUG")) {
9             System.out.println("DEBUG: " + message);
10        }
11        if (level.toString().equals("ERROR")) {
12            System.err.println("ERROR: " + message);
13        }
14        if (level.toString().equals("INFO")) {
15            System.out.println("INFO: " + message);
16        }
17    }
18 }
```

Many problems are not mentioned, but they all have the same consequences for the application: they reduce the code's readability and maintainability.

Performance Issues

Performance issues are often connected with maintainability when someone fixes a problem inefficiently, and the code remains in production for a long time. For example, instead of using **StringBuilder's** `append` method to concatenate a list of strings, someone could consider using the `add` operation, which produces a new object each time the application uses its resources.

```
1 import java.util.List;
2
3 public class StringExample {
4     public String buildString(List<String> items) {
5         String result = "";
6         for (String item : items) {
7             result += item; // Inefficient because it creates a new String each time
8         }
9         return result;
10    }
11 }
```

In another case, someone tries to obtain more information about a particular situation on a block of code, which creates random results, so decides to introduce a lot of logs or use **System.out.println()**, incurring at some point in performance overhead.

```
1 public class PaymentProcessor {
2
3     // This debug flag should be removed in production
4     private static final boolean DEBUG_MODE = true;
5
6     public void processPayment(double amount) {
7         // Debug logging that incurs performance overhead
8         if (DEBUG_MODE) {
9             System.out.println("DEBUG: Starting payment process for amount: " + amount);
10        }
11        // More code to do something
12    }
13 }
```

Some of these problems could be obvious to the experienced developer, but not in all cases. Let's consider the scenario where someone creates an MR with 50 files with changes, which in most cases are not just two lines on each file, so the complexity of paying attention to all of them and detecting each possible problem is low.

Highlight the Causes

When bugs appear, it's not enough to address the symptoms and understand how to fix them—it's necessary to find the root causes of those problems, not just patch the code and continue because, in that case, the problem will continue happening.

Sometimes, developers lacking experience are unaware of subtle issues, or unrealistic deadlines force the team to implement quick fixes instead of addressing the real problem. It could be just one reason or a combination, but whatever the root cause, the solution is ignored or delayed, producing recurrent issues.

Lack of Experience

In most cases, the experience and level of the developers are factors that could increase or decrease the number of bugs and the possibility of detecting those problems in a review process. Another thing to consider is how many of your company's developers know about the feature in the latest version of Java that your company uses; this is especially useful because many things change across different versions, like the appearance of pattern matching or new ways to declare variables. It does not make sense for your company to use a static analysis tool to detect problems if the developers do not understand the behind-the-scenes issues.

The following is a possible example of a harmful use of a new feature that was included in many versions ago.

```
1 // Definition of a record, a feature introduced in recent versions of Java.
2 public record Person(String name, int age) { }
3
4 public class RecordExample {
5     public static void main(String[] args) {
6         Person person1 = new Person("Alice", 30);
7         Person person2 = new Person("Alice", 30);
8
9         // BAD USAGE: Using '==' to compare two records.
10        if (person1 == person2) {
11            System.out.println("The persons are equal (using '==').");
12        } else {
13            System.out.println("The persons are different (using '==').");
14        }
15    }
16 }
```

The error is quite simple, but not in all cases. It is not trivial because the application does not have just a few lines of code.

Time Constraints

The deadlines for launching a new application or feature or solving a bug decrease code quality because there is no time to check everything. Most developers focus on the happy path or just a few cases to check if nothing terrible occurs. This approach of not paying attention to the quality of the code is like a bomb that could explode shortly because not all lousy situations arise at the beginning. A flow may depend on a series of conditions that are not frequent.

The worst scenario is that your company does not have at least one static analysis tool to detect and track the problems. If the deadline reduces the chance to check and fix everything, at least you have a tool to help you when the issues appear instead of finding yourself across all the code problems.

The Complexity of Modern Applications

The evolution of application design, such as the use of hexagonal instead of layer architecture, the use of too many libraries or frameworks to simplify some repetitive tasks, and changes in the mechanism for interacting with databases, such as Spring Data or another library, or communicating with other applications, such as synchronous communication or using events, increases the complexity and the number of things that developers need to know.

This situation exacerbates the problem because the developers must pay attention to many small details to take action. This is not a problem, but without a correct peer review process or the use of a tool to detect it, most of them pass through the developers' noise, and no one sees them.

Code Reviews or Automated Checks

Another big problem affecting the quality of the code is the lack of a proper validation process. Some companies delegate everything on each Merge Request to a developer, such as checking whether the code senses the feature description and has the correct number of tests, units, or integration.

This type of approach, in most cases, produces big problems because anyone can check everything all the time without making a mistake, and the risk when someone does something is zero. Still, there are ways to mitigate, like a combination of developers who check whether the code makes sense and delegate the analysis to a tool. This situation is the best scenario if your company's tool can detect and fix problems without intervention. Still, as a suggestion, everyone needs to understand why the tool changes the code because it will reappear if no one understands the situation.

Static Code Analysis Tools

These tools play a crucial role in software development. They scan the code to detect fragments that do not adhere to coding standards, identify potential bugs, and suggest solutions for these problems. Developers can use these tools to help identify issues early in development. The tools can be run locally before pushing the code into the repository. This practice reduces the time and effort needed to fix problems after they are detected, especially when they appear on a website as reports of the issues.

Tools

There are too many tools related to code analysis, but this subsection focuses on three popular ones: PMD, Checkstyle, and SonarLint. Each tool offers unique benefits: PMD scans for bugs and inefficiencies, Checkstyle enforces style and coding conventions, and SonarLint provides real-time feedback directly within your development environment.

Let's look at each tool in more detail to understand its scope and benefits to developers.

Tool	Features
SonarLint	- Provide real-time feedback directly in your IDE - Have integration with SonarCloud/SonarQube so it's possible to use the same set of rules locally rather than remotely. - Quick detection of vulnerabilities and bugs in the code, offering a description of how to fix them and the reasons <u>why</u> is a good practice in most cases.
Checkstyle	- Enforce code standards and style conventions, check the code, and show the existing problems. - It's one of the tools that offers more <u>customization</u> of the rules and messages about issues. - It can be used inside your project with Maven/Gradle or as a plugin in your IDE.
PMD	- Detect a great range of problems like bugs and code smells. - Support the creation of custom rules to detect or check specific things <u>on</u> the code.

Abb. 17: Table - Key features of the most relevant tools in the static code analysis

Other tools exist, such as Spotbugs, Huntbugs, and Spoon, but most do the same thing as the three previous tools mentioned in the first paragraph. They all have pros and cons; no silver bullet will solve all your problems.

Limitations of These Tools

All the static code analysis tools are incredible and invaluable for detecting a broad range of issues in the early development steps, but they all have limitations at some point. Primarily, they all identify problems and show why some of the code is incorrect but cannot automatically fix them, requiring one or more developers to manually review and address the flagged matters.

Resolving these problems implies considerable time and effort, especially if you start using these tools with large or legacy applications with many issues. Not all developers have to check an extensive list of problems, understand them, and try to fix each.

Another problem is recurrence, the number of times that the same problem appears in the code, which implies applying many times to introduce the same fix; in some cases, it is just changing one line in the code, or, in another case, it is more complex but in the simple cases could be tedious do the same corrections many times.

Automating Fixes

The next level of the static analysis code is not just detecting the problems; in some cases, the issue is simple, just fixed without human intervention. Of course, in the beginning, someone needs to configure the tools as the others do, but the interventions tend to be zero afterward. One of the most prominent exponents of these types of tools is error-prone.

Introduction to the Error Prone

Error Prone is an open-source static analysis tool developed by Google. It integrates directly with the Java Compiler using Maven or Gradle or as a plugin in the IDE. The tool hooks into the compile-time process, inspecting the code being compiled simultaneously to detect potential bugs or pitfalls before proceeding. This proactive approach catches issues overlooked during traditional testing or code reviews.

In addition to identifying problems, the tool sometimes suggests fixes or can automatically apply corrections to your code without your intervention. To achieve this, one core library contains the most relevant rules related to Java and how to fix them, while other companies, like Picnic, create their own set of rules, which can be combined with those defined by Google.

Unlike traditional tools, which operate as a separate step in the development workflow during the code review process or as part of continuous integration execution, Error Prone reduces context switching and provides quick feedback to developers regarding issues. Furthermore, it identifies complex problems that most conventional tools may miss and offers a detailed analysis level that enhances your code's reliability.

Benefits of Using Error Prone

Using `error_prone` brings multiple benefits, including early detection of complex issues, automated fixes, and immediate feedback that helps enforce best practices and boost productivity. Let's see more in detail the most relevant:

- **Detecting complex issues:** The tool automatically detects problems related to harmful practices or performance issues without writing extra rules. However, it's possible to include an external set of rules or create relevant new rules. It could be a good idea to analyze which companies make their own rules instead of building one from scratch.
- **Auto-fixing specific problems:** This benefit is one of the most significant differences from other tools because it minimizes the manual effort needed to resolve common issues. You can configure the tool to display the problems and use additional commands to indicate whether to fix them or to perform both actions simultaneously. It's advisable to separate the process into two distinct steps or commands until you fully understand the rules and which ones are appropriate to enable. After that, you can use a single command to conduct the entire process.
- **Enhanced Developer Productivity Through Immediate Feedback:** Receiving real-time feedback on code enables developers to avoid spending hours identifying issues in a code block and minimizes the time other developers must dedicate to reviewing each line of code.
- **Enforcing Best Practices and Avoiding Common Pitfalls:** This tool highlights issues in the code and identifies the parts that stray from established best practices. When developers receive prompt feedback on code quality, they are less likely to make the same mistakes in the future, motivating them to write more robust and maintainable code.

Example of Use

Let's begin with an example of how this tool can be used in a project that may encounter quality-related issues. The project's source code, which will be used to practice everything, is in this [repository](#) on GitHub.

The first step is to add the dependency related to **error_prone**, found in the [Maven Repository](#), to the application's configuration. The application uses Maven, but it could be the same on Gradle.

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-compiler-plugin</artifactId>
6       <version>3.11.0</version>
7       <configuration>
8         <source>21</source>
9         <target>21</target>
10        <encoding>UTF-8</encoding>
11        <compilerArgs>
12          <arg>-XDcompilePolicy=simple</arg>
13
14
```

```

15         <!-- Enable error_prone as a plugin -->
16         <arg>-Xplugin:ErrorProne</arg>
17     </compilerArgs>
18     <annotationProcessorPaths>
19         <path>
20             <groupId>com.google.errorprone</groupId>
21             <artifactId>error_prone_core</artifactId>
22             <version>2.30.0</version>
23         </path>
24     </annotationProcessorPaths>
25 </configuration>
26 </plugin>
27 </plugins>
28 </build>

```

By adding this simple configuration to the maven-compiler-plugin, your project can use error_prone. If you want to include external libraries with extra rules, you should include them on the section path after or before error_prone_core.

There is a problem if your project uses Java 16 or up because there is a strong encapsulation, so running any command on Maven with those changes will fail. A way to fix this problem is to add a file on **.mvn/** with the name **jvm.config** and add flags like the following:

```

1  --add-exports jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED
2  --add-exports jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED
3  --add-exports jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED
4  --add-exports jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED
5  --add-exports jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED
6  --add-exports jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED
7  --add-exports jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED
8  --add-exports jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED
9  --add-opens jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
10 --add-opens jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED

```

Alternatively, it's possible to introduce the same configuration into the pom file. By making these changes to the application and executing the simple command **mvn clean verify**, all possible bugs or problems in the code will be checked by default. The entire list of checks for this tool is available at this [link](#).

Let's examine some common errors that might occur in the source code of any project. If it's necessary to throw an exception when a record does not exist in the database, instead of doing it correctly, your code looks like this:

```

1  public ReservationDTO getReservationById(Long id) {
2      Optional<Reservation> result = repository.getReservationById(id);
3      if (result.isEmpty()) {
4          LOGGER.debug("Not exist reservation with the id {}", id);
5          new TWAException(APIError.RESERVATION_NOT_FOUND);
6      }
7      return conversionService.convert(result.get(), ReservationDTO.class);
8  }

```

The following output will appear on the console, indicating that something is wrong and which could be the possible solution.

```

1  [ERROR] COMPILATION ERROR :
2  [INFO] -----
3  [ERROR] /home/asacco/code-wizardry/api-reservations/src/main/java/com/twa/reservations/
4  service/ReservationService.java:[48,13] [DeadException] Exception created but not
5  thrown
6  (see https://errorprone.info/bugpattern/DeadException)
7  Did you mean 'throw new TWAException(APIError.RESERVATION_NOT_FOUND);'?

```

Not all problems can have the same severity; for example, imagine that you need to create an **equals** method on the entity class like the following block.

```

1  public class Price {
2
3      private Long id;
4      private BigDecimal totalPrice;
5
6      private BigDecimal totalTax;
7
8      private BigDecimal basePrice;
9
10     // Setters and Getters
11
12
13     @Override
14     public boolean equals(Object o) {
15         if (this == o)
16             return true;
17         if (o == null || getClass() != o.getClass())
18             return false;
19         Price price = (Price) o;
20         return Objects.equals(id, price.id) && Objects.equals(totalPrice, price.
21             totalPrice)
22             && Objects.equals(totalTax, price.totalTax) && Objects.equals(basePrice,
23                 price.basePrice);
24     }
25 }

```

If you execute the command again to verify the code, the console does something like the following block:

```

1  [WARNING] /home/asacco/code-wizardry/api-reservations/src/main/java/com/twa/reservations/
    model/Price.java:[48,20] [EqualsGetClass] Prefer instanceof to getClass when
    implementing Object#equals.
2    (see https://errorprone.info/bugpattern/EqualsGetClass)
3    Did you mean 'if (!(o instanceof Price))'?

```

The basic rules can detect many other things, but this article does not explain all of them.

Running error_prone auto-fixes

Detecting bugs and providing information on how to fix them is necessary. However, in some cases where the problem is trivial, the plugin should resolve this issue without any intervention. To achieve this, it's essential to modify the plugin's configuration by specifying what tasks it should handle to fix the issue. Let's make some adjustments to address all occurrences of the problems mentioned in the previous section; replace the last line that enables `error_prone` with the following:

```

1  <arg>-Xplugin:ErrorProne -XepPatchChecks:DeadException,EqualsGetClass -XepPatchLocation:
    IN_PLACE</arg>

```

After that, when any command is executed, all the problems will be fixed automatically without any intervention. Of course, it's possible to indicate that solve all of the issues, not just a few of them, but as a suggestion, always try to identify which problems are good candidates to solve automatically and which of them it's okay to have a report or notification about the issue.

Advanced Configuration

Not all issues require the tool's default configuration; in some cases, changing the severity of a type of issue or just disabling it because it's irrelevant to your application is essential. `Error_prone` offers the possibility of having a granular configuration considering all these possible scenarios.

The following table refers to some of the most common parameters:

Parameter	Description
-XepAllErrorsAsWarnings	Converts all <u>issues that the library could detect as errors</u> into warnings. This could be useful for compiling everything, but <u>at the same time, we need to know which problems are in the code.</u>
-XepAllSuggestionsAsWarnings	Downgrades all suggestion messages to warnings.
-XepAllDisabledChecksAsWarnings	Instead of completely ignoring <u>checks that have been disabled</u> , this flag treats them as warnings.
-XepDisableAllChecks	Disable all <u>the checks of the library</u> . This could be useful <u>to bypass all the validations of the tool temporarily.</u>
-XepDisableAllWarnings	Suppresses all warning messages from the tool so no warnings are shown during compilation.
-XepDisableWarningsInGeneratedCode	Disable all the library checks. This could be useful temporarily to bypass all the tool's validations.

Abb. 18: Table - Default parameters to use on the configuration

In addition to the previous parameters, it's possible to change the severity or disable/enable some specific rules. Let's see in the following table how to do it with DeadException.

Parameter	Description
<code>-Xep:DeadException</code>	This parameter turns on the validation with <u>a severity</u> that the rule has by default.
<code>-Xep:DeadException:OFF</code>	This parameter turns off the validation.
<code>-Xep:DeadException:WARN</code>	With this parameter, it's possible to change the severity of the rule to WARN.
<code>-Xep:DeadException:ERROR</code>	With this parameter, it's possible to change the severity of the rule to ERROR.

Abb. 19: Table - Custom parameters to modify the behavior of the different rules

With all of these options in the tool's configuration, you can achieve an excellent level of validation and recommend what is relevant to your project.

Using External Libraries of Rules

Sometimes, the rules offered by default `error_prone` are insufficient or do not cover all the scenarios in the typical applications. To solve that, it's likely to add some extensions that add new rules; one of the most famous is [Picnic's library](#), which offers a wide range of regulations related to the use of loggers or the ways to use some variables.

To use this extension on the rules, it's necessary to add two new dependencies in the same place where it is located `error_prone_core`, like appears on the following fragment:

```

1  <annotationProcessorPaths>
2      <!-- Error Prone itself. -->
3      <path>
4          <groupId>com.google.errorprone</groupId>
5          <artifactId>error_prone_core</artifactId>
6      <version>${error-prone.version}</version>
7      </path>
8      <!-- Error Prone Support's additional bug checkers. -->
9      <path>
10         <groupId>tech.picnic.error-prone-support</groupId>
11         <artifactId>error-prone-contrib</artifactId>
12     <version>${error-prone-support.version}</version>
13     </path>
14     <!-- Error Prone Support's Refaster rules. -->
15     <path>
16         <groupId>tech.picnic.error-prone-support</groupId>
17         <artifactId>refaster-runner</artifactId>
18     <version>${error-prone-support.version}</version>
19     </path>
20 </annotationProcessorPaths>

```

It's essential to check the compatibility of the versions on Picnic's `error_prone` with the core listed in the [official documentation](#).

After that, if the compilation process occurs again, it's possible to see new errors that appear in the following output:

```

1  [INFO] /home/asacco/Code/code-wizardry/api-reservations/src/test/java/com/twa/
    reservations/util/ReservationUtil.java:[43,37] [Refaster Rule] ImmutableListRules.
    ImmutableListOf1: Refactoring opportunity
2    (see https://error-prone.picnic.tech/refasterrules/ImmutableListRules#
        ImmutableListOf1)
3    Did you mean 'itinerary.setSegment(ImmutableList.of(segment));'?
4
5
6  [WARNING] /home/asacco/Code/code-wizardry/api-reservations/src/test/java/com/twa/
    reservations/service/ReservationServiceTest.java:[26,33] [Slf4jLoggerDeclaration]
    SLF4J logger declarations should follow established best-practices
7    (see https://error-prone.picnic.tech/bugpatterns/Slf4jLoggerDeclaration)
8    Did you mean 'private static final Logger LOG = LoggerFactory.getLogger(
        ReservationServiceTest.class);'?

```

Reducing the scope or the number of enabled rules is relevant in the same way that it's necessary to do with the basic rules.

What's Next?

There are many resources on different topics connected with the quality of the applications and how bugs affect an application. Still, just a few tackle solving or mitigating it using static code analysis. The following is just a short list of resources:

- [Re-Engineering Legacy Software by C. Birchall](#)
- [SonarQube in Action by G. Dubois, A. Delmas, and M. Rodiere](#)
- [Clean as You Code: Elevating Software Quality through Continuous Code Improvement By Peter McKee](#)
- [Lincheck: Testing concurrency on the JVM](#)
- [Streamlining Large-Scale Java Development Using Error-Prone by Sander Mak](#)

Other resources that could be great for understanding some concepts related to good practices and the relevance of the quality of an application are:

- [Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin](#)
- [Good Code, Bad Code by Tom Long](#)

This is just a small list of all the available resources on quality and static analysis. If something is unclear, find another video, book, or resource.

Conclusion

Problems related to lousy code quality are ineligible; they could appear at the beginning of an application or take time, depending on many aspects, such as the developer's experience, the time they have to review, and their focus on each little detail.

One of the most effective ways to mitigate or reduce the problem is to use static analysis tools. Of course, tools like `error_prone` will help the developer detect the issue because, often, this tool could fix it. This tool, like others, allows developers to understand where a problem is and improve their ability to identify incorrect things, so consider using one in your project or company.

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

```
@Entity
class RepoController {
    @Entity customerRepository
    W@customerRepository();
}

@Override getCustomer() {
    return customerRepository;
}
}
```

Entity

@ Table

javax.persistence

Table

Maven artifact

@ MappedSuperclass

AI code sample

Japo: The Ultimate Search Engine for Java Annotations

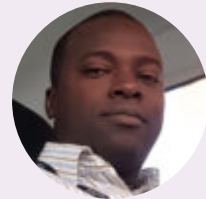
AI

DeveloperTools

Japo

Author: Mohamed Ashraf Bayor

Mohamed is a Software Engineer with over 15 years of experience in Java development. Throughout his career, he has successfully navigated diverse high-technology industries, including FinTech, Aerospace Manufacturing, Telecommunications and Healthcare. Passionate about open source contributions, Mohamed launched two impactful projects in November 2021: Proporec (<https://froporec.org/>) and Jisel (<https://jisel.org/>). In October 2023, Mohamed also introduced JAPO (Java Annotations Repo - <https://japo.dev/>), which marks the world's first search engine dedicated to searching and exploring Java annotations.



LinkedIn

In the evolving landscape of Java development, annotations play a critical role in simplifying code, enhancing readability, and enabling powerful framework capabilities. Yet, finding detailed, accurate, and consolidated information about Java annotations has long been a challenge for developers. **JAPO** (Java Annotations Repo) is the game-changing solution addressing this gap. Offering a feature-rich search engine, JAPO enables Java developers to explore annotations, Maven artifacts, and much more with unparalleled ease.

This article delves into JAPO's capabilities and its transformative impact on Java development.

Challenges of Java Annotations

Annotations have been a key part of Java since their introduction in **September 2004** with [JSR 175](#), released in [Java 5](#). By using annotations, developers can embed metadata into code, allowing libraries and frameworks to automate tasks, validate data, and support declarative programming.

However, with thousands of annotations spread across different frameworks and libraries, finding the right one can be overwhelming. Developers often need to dig through documentation, browse multiple websites, or inspect source code just to understand how an annotation works or whether it fits their needs. The lack of a centralized resource makes this process time-consuming.

Library / Framework	Approximate Annotations Count
Jakarta EE	400
Hibernate	200
Spring Framework	200
Spring Boot	150
JUnit 5	80
Project Lombok	60
Swagger	50

Abb. 20: Approximate Total Count of Annotations Across Maven Artifacts (Manually Counted in 2022)

In addition, annotations often have similar names, making it easy to confuse one with another. Some are framework-specific, while others are standard Java annotations, adding another layer of complexity. Developers may also struggle to determine which annotations are still actively maintained and which are deprecated or replaced by newer alternatives.

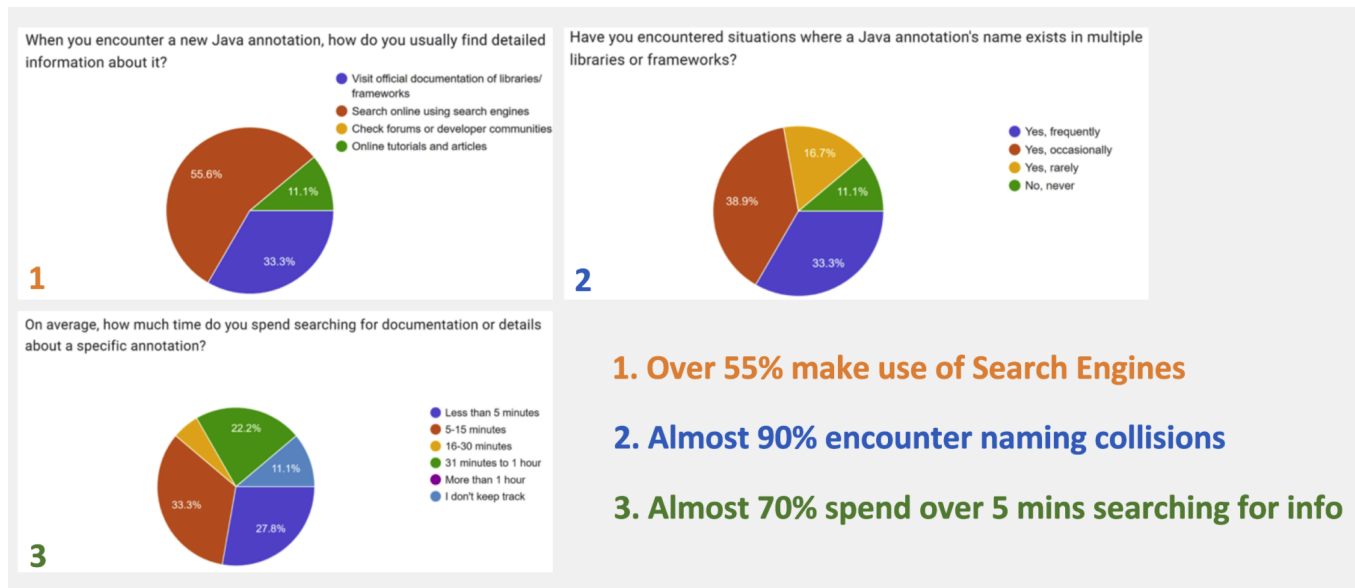


Abb. 21: Based on a 2023 survey of Java developers, compiled from 68 responses

JAPO addresses these challenges by offering an intelligent and user-friendly search platform. By consolidating annotation details from various sources, developers can quickly access accurate information. With **powerful search features** and **AI-generated code samples**, JAPO streamlines the discovery process, saving developers valuable time and effort.

Introducing Japo: Java Annotations Repo

JAPO is a search engine designed for Java developers, providing detailed information about Java annotations across various libraries and frameworks. To suit different development needs, JAPO provides multiple access points:

- **REST API:** A robust backend API for seamless integration, available at <https://japo.dev/api>.
- **Web Interface:** A user-friendly, interactive web interface accessible at <https://japo.dev>
- **IntelliJ Plugin:** A fully integrated solution for IntelliJ IDEA, available for download on the [JetBrains Marketplace](#)

Key Features of Japo

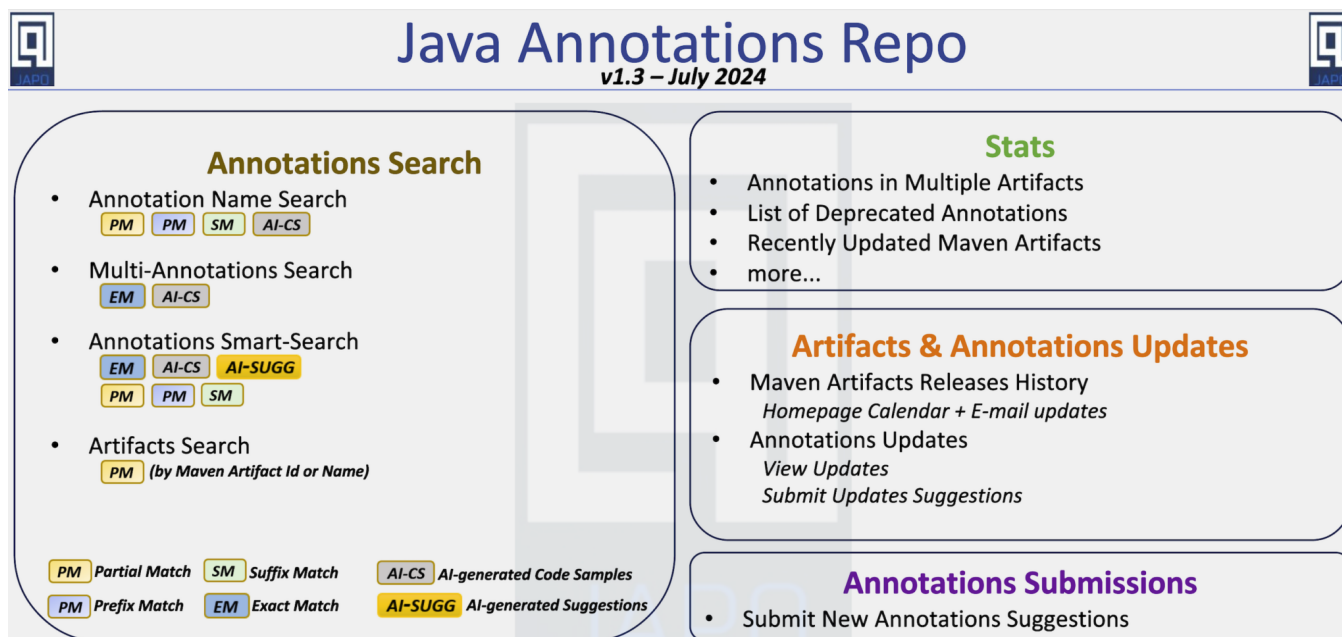


Abb. 22: Summary of features offered by JAPO API v1.3 (July 2024)

Annotation Search

JAPO offers a **powerful and flexible annotation search system**, allowing Java developers to efficiently find and analyze annotations across various libraries and frameworks. With multiple search modes, JAPO ensures precise and relevant results, making annotation discovery easier than ever.

- **Annotation Name Search** – Finds annotations based on different matching criteria. By default, it searches for the **simple name**, but if the query contains a dot (.), it switches to the **fully qualified name**. It supports **partial match (PM)** for annotations containing the query string, **prefix match (PM)** for annotations starting with the query, and **suffix match (SM)** for annotations ending with the query.
- **Multi-Annotations Search** – Allows developers to search for **multiple annotations in a single query**, making it easier to analyze related annotations. This feature supports multiple annotation names separated by spaces, performs **exact matches (EM)** on each provided name, and works with both **simple and fully qualified names**.
- **Annotations Smart-Search** – An advanced search mode that enhances multi-annotation queries with **AI-powered intelligence**. It supports **all match types** (exact, partial, prefix, and suffix) while providing **AI-generated suggestions (AI-SUGG)** to recommend relevant annotation names based on the query.
- **Artifacts Search** – Searches for **Java artifacts** by **Maven Artifact ID** or **Artifact Name**. It returns results that match any of the words provided in the search criteria. All Maven artifacts returned in the search results belong to **Java libraries or frameworks that contain annotations**.

JAPO's annotation search feature provides a comprehensive and efficient way to explore Java annotations and artifacts. Whether using **direct name searches**, **multi-annotation queries**, **AI-enhanced smart search**, or **artifact lookup**, JAPO streamlines the process and helps developers quickly find the annotations they need.

Additionally, whenever a search returns annotation data, **JAPO utilizes AI to generate code samples (AI-CS)** for each annotation in the search results, giving developers practical usage examples instantly.

IntelliJ Plugin Exclusive Features

The plugin enhances developer productivity by embedding JAPO search features into IntelliJ IDEA.

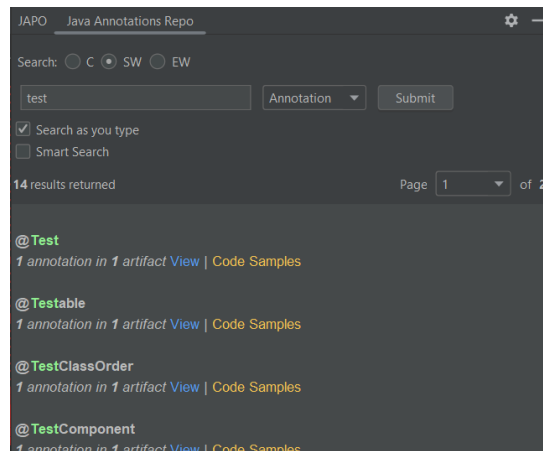


Abb. 23: Search

[Get the JAPO Plugin for IntelliJ – Available on JetBrains Marketplace](#)

In addition to its simple user interface, which allows users to manually enter search criteria for various search features, JAPO offers two additional capabilities: **Scan with JAPO** and **Search with JAPO**. Both features can be accessed directly from the **IntelliJ Project Explorer** or the **Code Editor**, making annotation discovery even more seamless.

Scan with JAPO

- **From Project Explorer** – Right-click on a file (typically a Java class or interface, but it can also be any text file containing annotations) and select “**Scan with JAPO**” from the pop-up menu.
- **From Code Editor** – When the file is open, right-click anywhere within the editor and select “**Scan with JAPO**” from the pop-up menu.

Either action will trigger a **Multi-Annotations Search**, and the results will be displayed in the **JAPO plugin window**.

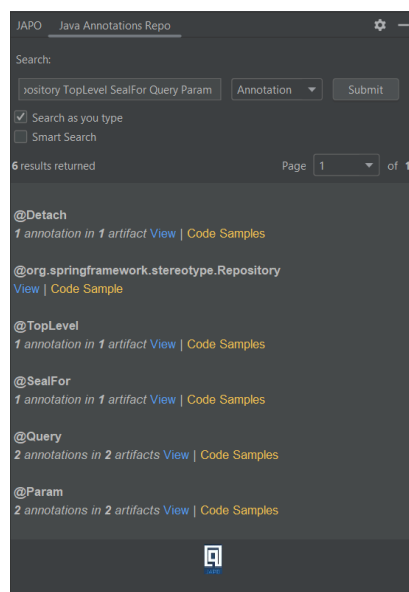


Abb. 24: “Scan with JAPO” performed on a file containing the listed annotations

Search with JAPO

This action will trigger an **Annotation Name Search**, and the results will be displayed in the **JAPO plugin window**.

- **From Code Editor** – Select an annotation name (either simple or fully qualified), right-click on the selected text, and choose “**Search with JAPO**” from the pop-up menu.

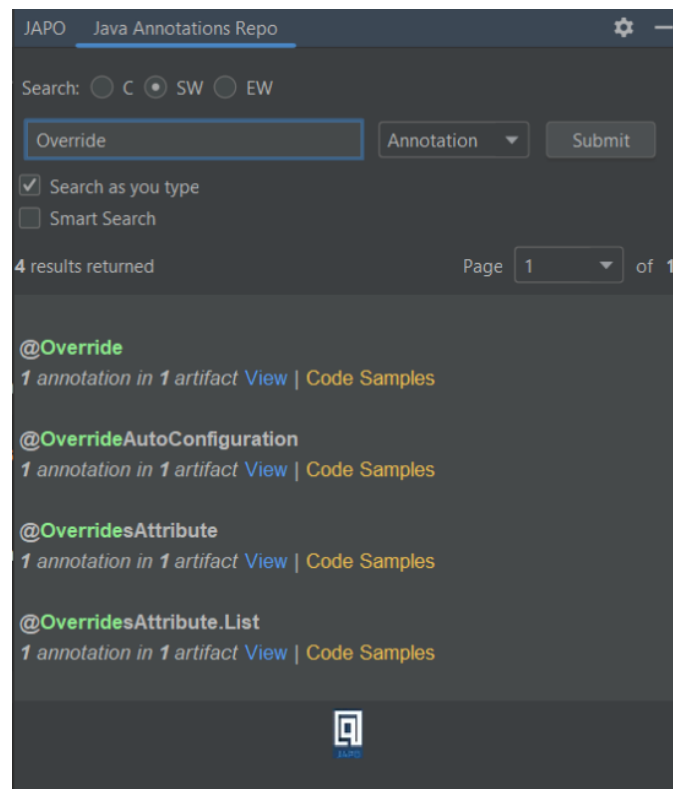


Abb. 25: “Search with JAPO” performed *after selecting “Override” text in the Code Editor*

Other Features

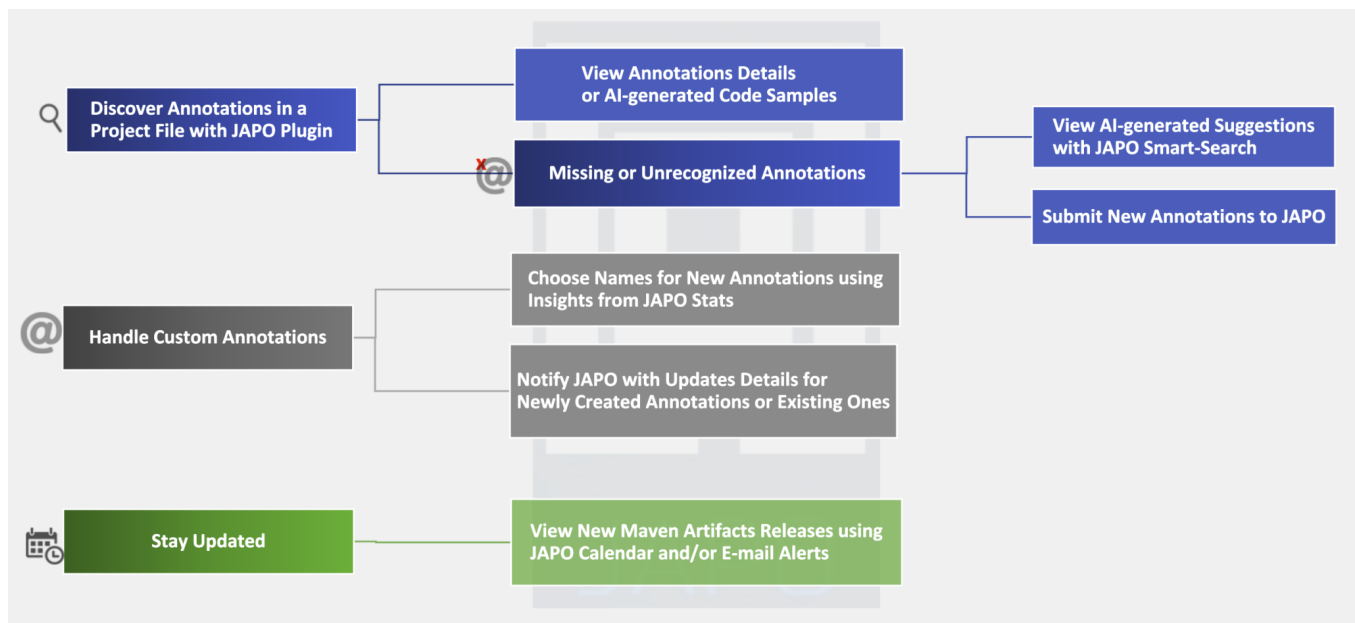
In addition to its core functionalities, **JAPO** offers several other features to enhance the user experience and provide valuable insights into Java annotations and Maven artifacts.

- **Stats** – View annotation statistics, including deprecated annotations and recently updated Maven artifacts.
- **Artifacts & Annotations Updates** – Stay informed about new releases and changes:
 - **Maven Artifacts Release History** – Track updates via the homepage calendar and e-mail notifications.
 - **Annotations Updates** – View recent changes or submit update suggestions.

These additional features make JAPO not just a search engine but a **collaborative platform** for tracking, managing, and improving Java annotations.

How Japo Helps Java Developers

Understanding and managing Java annotations efficiently is crucial for developers working with modern Java applications. JAPO simplifies this process by offering powerful search capabilities, annotation insights, and real-time updates, all seamlessly integrated into IntelliJ IDEA. The workflow below illustrates how JAPO enhances annotation discovery, management, and tracking.



With JAPO, developers can quickly scan project files to analyze annotations, access detailed insights, and leverage AI-generated code samples. The plugin also detects missing or unrecognized annotations, providing AI-powered suggestions through Smart-Search. Users can contribute by submitting new annotations, helping expand JAPO's database.

For custom annotations, JAPO offers naming suggestions based on usage statistics and allows developers to notify the platform about newly created or updated annotations.

To keep developers informed, JAPO tracks Maven artifact and annotation updates, delivering release history through a calendar and email alerts. This ensures teams stay current with the latest changes and best practices.

By integrating these features, JAPO enhances productivity, simplifies annotation management, and helps Java developers stay ahead of evolving trends.

Looking Ahead: Japo's Vision

As Java development evolves, so does the landscape of annotations. The increasing complexity of frameworks and libraries calls for a tool that consolidates annotation data while adapting to developers' changing needs. JAPO aims to be that essential tool, continuously improving to meet future challenges.

In the coming months, we plan to expand JAPO with new platforms and integrations, strengthening its role as the ultimate Java annotation repository. We will enhance our AI-driven search, making it easier for developers to find, understand, and apply the right annotations. As more contributors share their knowledge, JAPO's database will grow, covering a broader range of annotations and artifacts. This will empower teams to build robust, scalable, and efficient Java applications.

JAPO is transforming how developers work with annotations, turning a tedious process into one that is fast, efficient, and intuitive. By simplifying annotation discovery and management, JAPO makes Java development easier, more effective, and more enjoyable.

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



Unleashing the Power of Maven Archetypes

Archetypes

Maven

Plugin

Author: Giovanni van der Schelde

A software engineer specialising in Java. I'm passionate about writing clean, maintainable code and solving complex technical challenges. I not only develop software but also work as a trainer, speaker, and open-source contributor, where I love sharing knowledge and helping others grow.



LinkedIn

Creating new projects from scratch is a common task in today's world. Often, this involves manually setting up folders and files or using tools like Spring Initializr, Micronaut Launch, Eclipse Starter for Jakarta EE, Quarkus Starter, or IntelliJ to streamline the initial setup. But what if there was a way to automate and standardise these tasks, taking your project setup to the next level?

While these tools are excellent for generating a basic “Hello World” application, they often fall short when it comes to adding project-specific configurations and components that are standard within your organisation. As a result, you might find yourself spending time in a tedious and error-prone process such as copy-pasting parts of similar applications, renaming files, and replacing variables.

Introduction

For those unfamiliar with Maven, it is one of the most widely used project management tools in the Java ecosystem. Maven manages a project's build, reporting, and documentation from a central configuration. It operates through plugins that hook into its lifecycle phases, although some plugin goals can be used standalone.

One of those plugins is the Maven Archetype Plugin, which you might have encountered when following the official Getting Started Guide. Officially supported by the Maven Project [1], this plugin allows developers to create and use archetypes. Archetypes are original patterns or models from which other projects of the same kind are made. It is an abstract representation of a kind of project that can be instantiated into a concrete customised project.

Using the Maven Archetype Plugin, developers can quickly and consistently generate new projects. This not only boosts productivity but also ensures that the generated components adhere to best practices, guidelines, and standards within your organisation, while eliminating repetitive and tedious tasks.

How Maven Archetypes Work

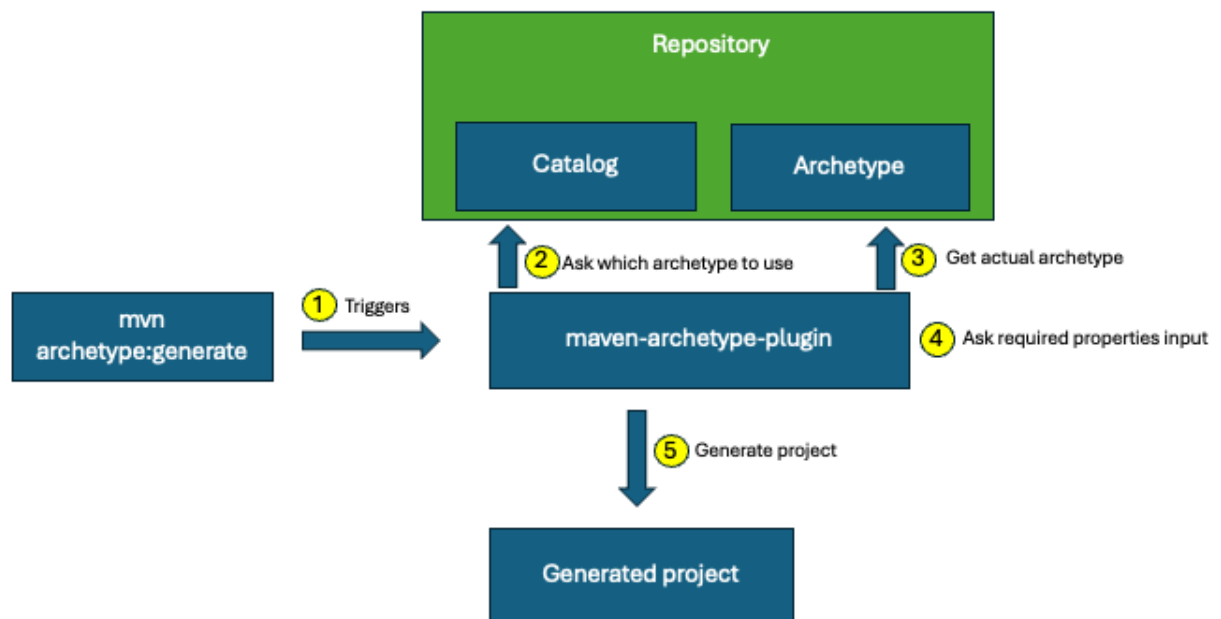
Before diving into creating an archetype, it's important to understand how they work. Like any other dependency, archetypes are typically hosted in remote or local repositories, using the same configuration as dependencies for resolution. If an archetype is not managed by a repository, you must add an additional repository entry with the ID `archetype` in the `settings.xml` file [2].

The Maven Archetype Plugin discovers available archetypes through a catalog file. This catalog stores information about archetypes, including their GAV (GroupId, ArtifactId, and Version), the repository they are hosted in, and their description. If the repository is omitted, the same repository as the catalog is used to locate the archetype.

The plugin comes bundled with several predefined archetypes, which can be displayed using the command: `mvn archetype:generate -DarchetypeCatalog=internal`.

This command uses the internal catalog. Besides the internal catalog, there is also a local catalog found in the home directory at `~/.m2/archetype-catalog.xml`, and a remote catalog accessible at `https://repo.maven.apache.org/maven2/archetype-catalog.xml`. They can be specifically targeted using `remote` or `local` as the value of the `archetypeCatalog` parameter.

Archetypes are packaged in a JAR file that contains a descriptor. This descriptor includes metadata about the archetype, such as required properties for project generation, possible default values, and the files to include and generate. Listing 10 gives a brief overview of this process. [10]

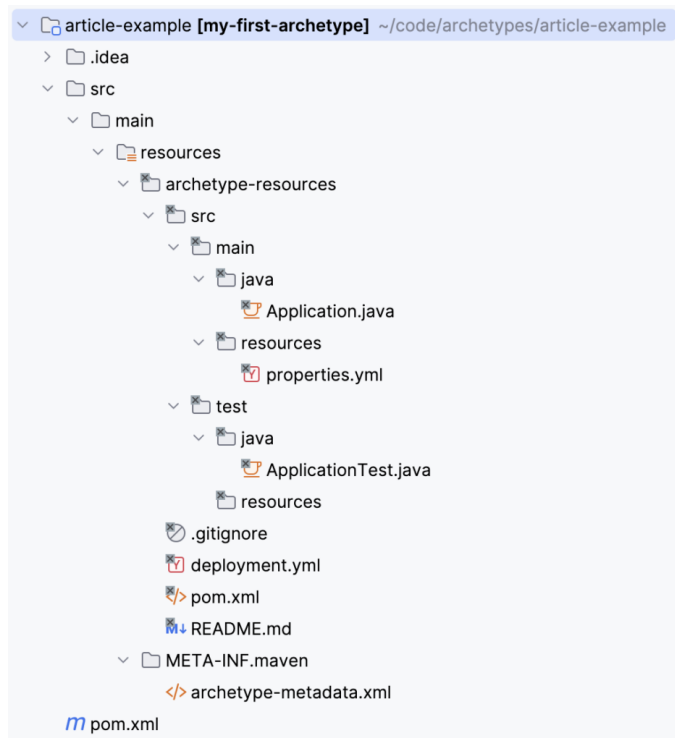


Generating a project from an archetype can be done interactively or non-interactively using additional command-line flags. The process involves selecting an archetype, entering additional configuration parameters or user input, and generating the project. The templating engine used, Velocity, expands the properties in the files and generates the dynamic content on the fly.

Creating an Archetype

Maven Archetypes can be created from scratch or by using the `archetype:create-from-project` goal, which generates an archetype from an existing project but includes all files. This requires manual filtering of unwanted files afterwards. Alternatively, the `maven-archetype-archetype`, which is an archetype to generate a simple archetype, generates most of the setup but is outdated. In this example, we will build an archetype from scratch, though the same concepts apply to both methods.

Creating an archetype project involves a different setup compared to most Maven projects. Let's assume the current working directory is `~/archetype`. The process begins with creating a new `pom.xml` file and the `src/main/` folders. Next, place the archetype descriptor in the `src/main/resources/META-INF/archetype-metadata.xml` file. The files that are copied and dynamically filled are placed in the `~/archetype/src/main/resources/archetype-resources` folder. These archetype resources could be build scripts, YAML pipelines, Git ignores, Java classes, and any other standardised files for your organisation. Listing 9 shows this archetype setup. [9]



Start by creating a new Maven project with a POM similar to the one in listing [3].

```

1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance"
2  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven
    -4.0.0.xsd">
3  <modelVersion>4.0.0</modelVersion>
4
5
6  <groupId>com.gioovds</groupId>
7  <artifactId>my-first-archetype</artifactId>
8  <version>1.0-SNAPSHOT</version>
9  <packaging>maven-archetype</packaging>
10
11
12  <build>
13    <pluginManagement>
14      <plugins>
15        <plugin>
16          <groupId>org.apache.maven.plugins</groupId>
17          <artifactId>maven-archetype-plugin</artifactId>
18          <version>3.2.1</version>
19        </plugin>
20      </plugins>
21    </pluginManagement>
22
23    <extensions>
24      <extension>
25        <groupId>org.apache.maven.archetype</groupId>
26        <artifactId>archetype-packaging</artifactId>
27        <version>3.2.1</version>
28      </extension>
29    </extensions>
30  </build>
31 </project>

```

Notice the use of a different packaging type. The archetype-packaging extension defines custom lifecycle bindings [4], allowing Maven to execute the appropriate plugin goals for this new packaging type.

The `archetype-metadata.xml` also known as the descriptor, has two attributes. The name describes the display name when the user selects the archetype. Partial indicates whether the archetype represents an entire project or only a part, such as a specific module or selection of classes. It also contains properties required to generate the archetype. The properties will be passed through the plugin and replaced by Velocity during project generation. The example in listing [5] shows some configuration options.

```

1 <archetype-descriptor
2   xmlns="http://maven.apache.org/plugins/maven-archetype-plugin/archetype-descriptor
  /1.1.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/plugins/maven-archetype-plugin/
  archetype-descriptor/1.1.0 https://maven.apache.org/xsd/archetype-descriptor
  -1.1.0.xsd"
5   name="first-archetype"
6   partial="false">
7
8   <requiredProperties>
9     <requiredProperty key="service-name">
10      <defaultValue>my-first-service</defaultValue>
11    </requiredProperty>
12    <requiredProperty key="team-name">
13      <validationRegex>^[A-Z].*$</validationRegex>
14    </requiredProperty>
15  </requiredProperties>
16
17  <fileSets>
18    <fileSet encoding="UTF-8" filtered="true" packaged="true">
19      <directory>src/main/java</directory>
20    </fileSet>
21    <fileSet encoding="UTF-8" filtered="true" packaged="false">
22      <directory/>
23      <includes>
24        <include>**/*.yaml</include>
25      </includes>
26      <excludes>
27        <exclude>secret.config</exclude>
28      </excludes>
29    </fileSet>
30    <fileSet encoding="UTF-8" filtered="false" packaged="false">
31      <directory/>
32      <includes>
33        <include>README.md</include>
34      </includes>
35    </fileSet>
36  </fileSets>
37 </archetype-descriptor>

```

The descriptor also configures which filesets to include and how to include them. The filtered attribute controls whether the files within this set contain properties that need to be replaced by Velocity. If set to **false**, the files are copied as-is. The packaged attribute places the files in a packaged structure, meaning files within this fileset will be placed in folders resembling a package-like structure. For example, if a project was generated with a groupId of `com.gioovds` and the file-set directory to include was `configuration`, the files within would be placed in `/my-service/configuration/com.gioovds`. This is mainly used for class files. The non-packaged files will be placed in the root folder `/my-service`.

To install and use the archetype, the archetype's JAR must be deployed to your local repository using `mvn install`. Then, generate a new project by running `mvn archetype:generate -DarchetypeCatalog=local`. The command line flag tells the plugin we are only interested in locally installed archetypes, to make it easier to find the newly installed one.

Property Interpolation with Velocity

Velocity templates in Maven Archetypes allow for dynamic content generation by utilising properties and custom logic.

For basic property replacement, Velocity properties use the same format as Maven properties, denoted as `${property}`. Properties defined in the `requiredProperties` element of the archetype descriptor can be accessed this way. Additionally, GAV values can also be used. For example, in a template POM file within the `archetype-resources` folder, these properties can automatically fill the `groupId`, `artifactId`, and `version`.

However, when replacing properties in filenames, the property name is surrounded by two underscores on each side. For example, if there is a property `service-name` and a file needs to include this property in its name, it could be named `__service-name__Application.java`. Within this file, other properties would use the standard `${property}` format, as shown in Listing [6].

```

1 package ${groupId};
2
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.springframework.boot.autoconfigure.domain.EntityScan;
5 import org.springframework.boot.SpringApplication;
6
7 @SpringBootApplication
8 @EntityScan(basePackageClasses = ${service-name}Application.class)
9 public class ${service-name}Application {
10
11     public static void main(final String[] args) {
12         SpringApplication.run(${service-name}Application.class, args);
13     }
14
15 }
```

In some cases, it's useful to derive values from other properties. Velocity Template Language (VTL) statements allow for defining new variables that can be processed by the engine.

For example, consider a project that uses two different usernames for database access and migrations, following a naming convention like `name_db_user` and `name_migration_user`. If the archetype has a property named `database-username` filled by the user with a value like `contract_db_user`, a VTL statement can be used to derive the migration username instead of defining a separate property. The statement could split the value by an underscore and select the first and last parts to determine the migration username [7].

```

1 #set ($split = $database-username.split("_"))
2 #set ($splitLastIndex = $usernameSplit.size() - 1)
3     username: ${split[0]}_migration_${split[$splitLastIndex]}
```

This is one of the many examples of how VTL statements can be used to manipulate these properties. More information and advanced usages can be found in the Velocity User Guide [8].

Keeping Archetypes Up to Date

Software evolves over time, and so do the decisions and requirements that shape it. Consequently, archetypes need to be updated to reflect these changes. Manually checking if an archetype still works can be tedious. Fortunately, the plugin provides tools to automate this process.

The `integration-test` goal of the plugin can be utilised to automate the testing of archetypes. This goal is bound to the Maven integration test phase by default. It generates projects from the current archetype and optionally compares these generated projects to a reference copy.

Each test is represented as a sub-directory in `src/test/resources/projects`. This directory should contain an `archetype.properties` file. This file contains the properties used to generate the project, just like a user would via the command line.

Optionally, it can contain a `goals.txt` to specify one or more goals to run against the generated project. And an optional `reference/` directory containing a copy of the expected project structure and content for comparison.

Since an archetype project is a Maven project itself, and possibly contains other projects that define dependencies, it can benefit from automated dependency management tools to stay up to date. Tools like Renovate and Dependabot can automate the process of updating the archetype project and its dependencies. In combination with the test suite the archetypes require minimal maintenance and will stay updated over time without unknowingly breaking.

Final Thoughts and My Experience with Archetypes

I have successfully introduced archetypes in two organisations, mostly receiving great feedback from the teams. They are actively used by the developers, enabling them to start development quickly without the hassle of copy-pasting projects, making unnoticed renaming errors, and recreating all the necessary application and deployment configurations.

However, before enthusiastically introducing archetypes to your team, consider the following questions:

- How often do you need to create new services or partial components?
- Do you have the necessary automation in place to test and update the archetypes?
- How often do your project requirements change?

Creating and maintaining archetypes involves a learning curve, especially when working with Velocity templates. If the frequency of creating new components is low, the initial effort and continuous maintenance might not be justified. Without automated pipelines for updates and tests, the maintenance effort might outweigh the benefits. Additionally, frequent changes can impact the usefulness of archetypes since they are not backward compatible and cannot be re-applied to existing projects.

Ready to enhance your development workflow with Maven Archetypes? Start by assessing your team's needs, set up automated pipelines if you haven't already, and dive into the creation of your first archetype. The benefits of a more efficient and error-free setup process are just a few steps away. Don't wait—begin your journey with Maven Archetypes today and empower your teams to focus on what they do best: writing great code.

[1] maven.apache.org/plugins [2] maven.apache.org/archetype [4] github.com [8] velocity.apache.org

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

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



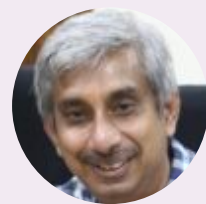
Launching a Java Debugger in Eclipse - A Myriad of Options

Debugger

Eclipse

Author: Manoj Nalledathu Palat

Manoj is an Open Source Lead/Committer working at IBM leading the Java Compiler in Eclipse. In parallel, Represented Eclipse Foundation in the Experts Group for Platform JSRs for multiple Java SE and IBM in Eclipse IDE WG Steering Committee. Passionate about training (Worked as Professor of Practice from January 2024-Dec 2024 at NIT, Calicut), Provide training internally at IBM regularly, blogs sometimes on [medium](<https://medium.com/@manojnp>), mentors people. Holds a Masters degree in Computer Science from Indian Institute of Science(IISc), Bangalore and B-Tech in Computer Science from National Institute of Technology (Formerly REC) Calicut, India.



LinkedIn

How to start a debugger? Seriously? Is it really worth a story? Let us find out about the different features of launching..
We need to of course launch in Debug mode :

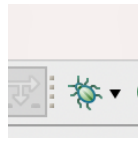


Abb. 26: Debug Button

And then pick up the following option:

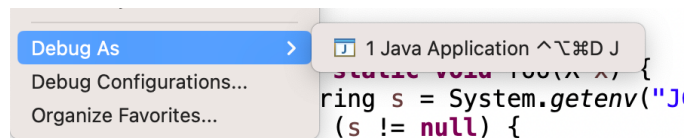


Abb. 27: Pull down from the debug button

For the detail-oriented, there is this Show Command line option:

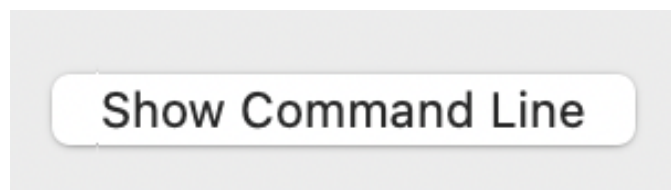


Abb. 28: click to see command line

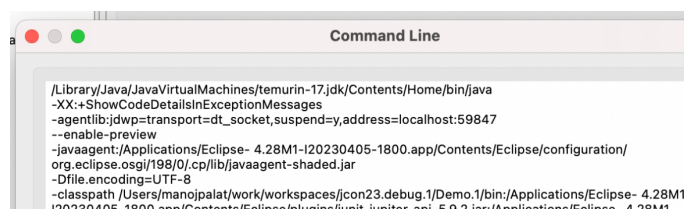


Abb. 29:

Ok, let's escape back to the world of UI fast:

There are different launch options, as you can see:

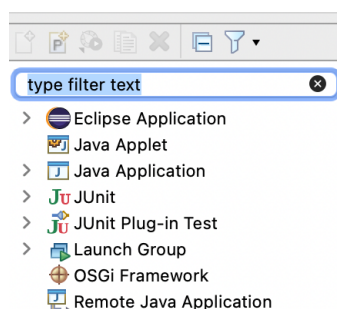


Abb. 30: Launch Options

Java Application, Eclipse Application, JUnit, etc. Let's go through some of them. Please remember the following button to start a new launch configuration:

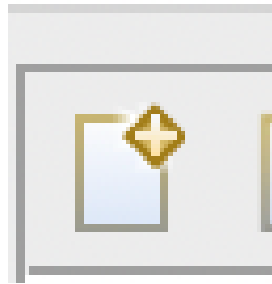


Abb. 31: Button to Start a new launch

Java Application

Clicking on a new Java launch configuration gives us the following:

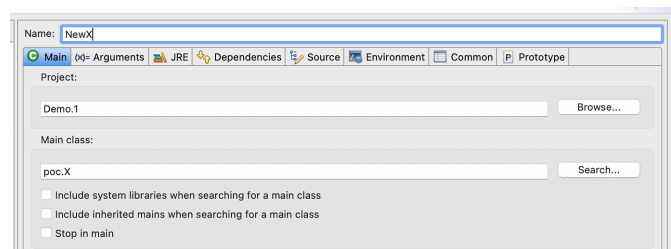


Abb. 32: Create a New Java Application Launch Configuration

Main

Let's look at the "Stop in main"

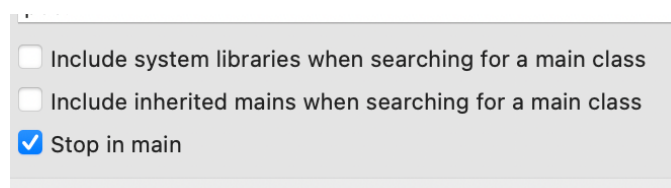


Abb. 33:

This is equivalent of putting a breakpoint in the main method albeit implicitly.

Moving on to the Arguments tab, we can put variables out there both for program and vm as shown below.

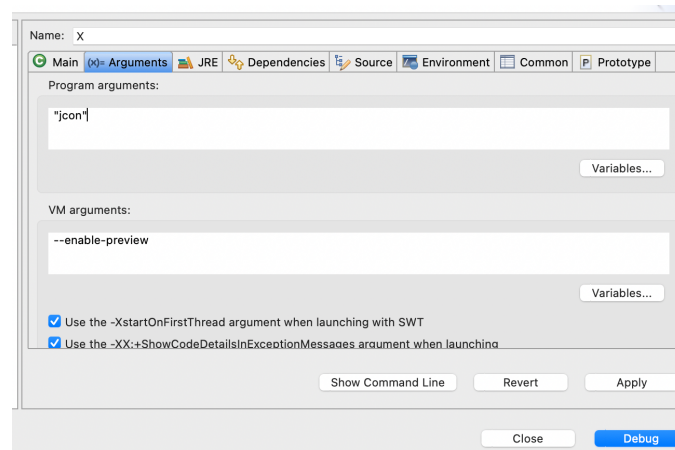


Abb. 34:

Additionally, if you want to add some pre-defined variables, click on variables for the list

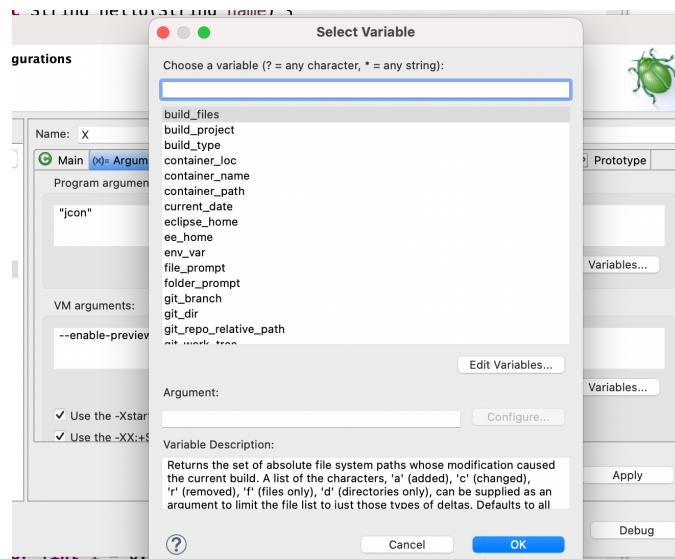


Abb. 35:

The JRE Tab:

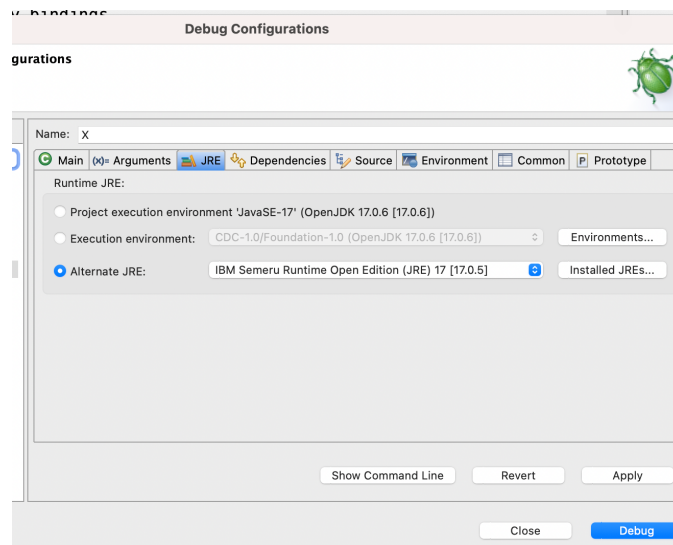


Abb. 36:

Click on the Installed JREs to installed your favourite JRE here.. How is this helpful? Especially if you are using one of the later JREs for testing out preview features

Environment Variables:

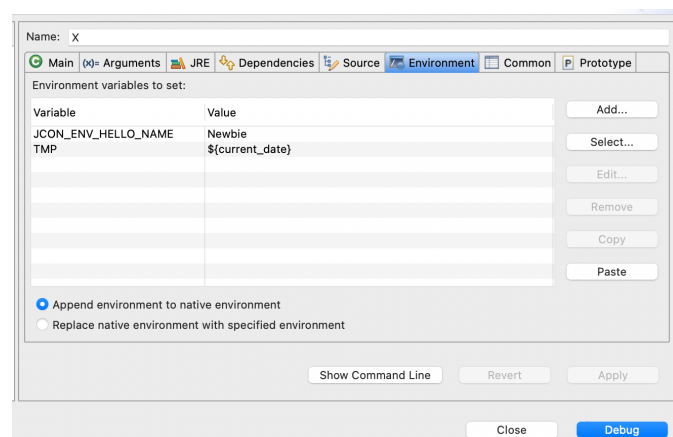
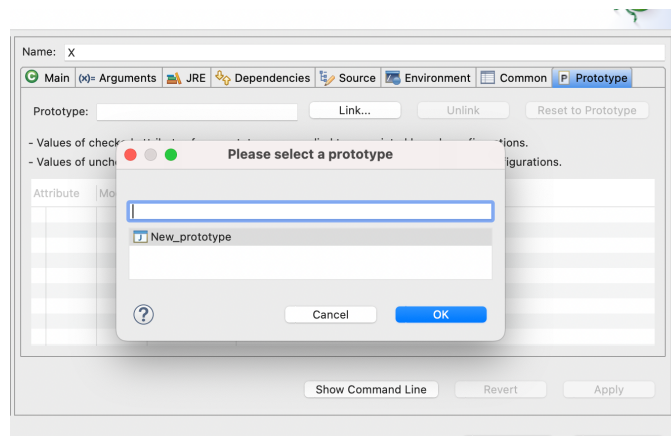


Abb. 37:

Click Add to create new env, Click select to add existing ones.

You have other options as well — appending — works well for paths /replacing — good for single element vars

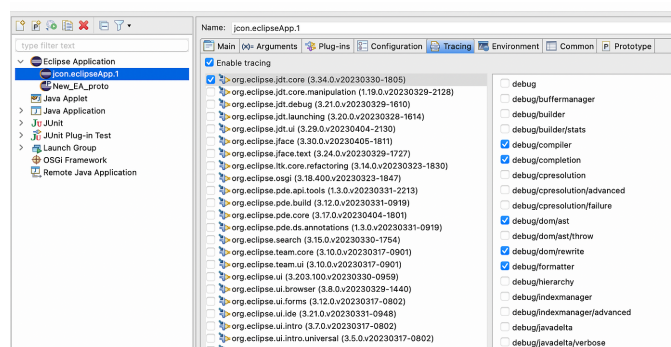
Prototypes:

**Abb. 38:**

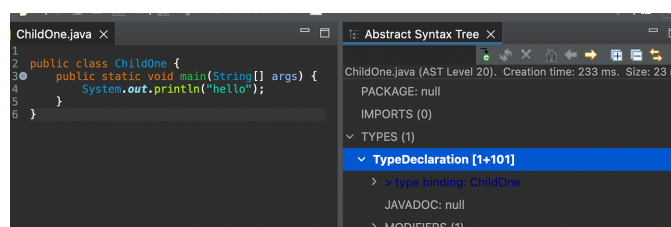
Click on link to a prototype — let us say for now its just a template. will come back to this.

Eclipse Application

You can start a new launch as Eclipse Application. In that case, you have an additional option called Tracing as shown below:

**Abb. 39:**

Click on one of the plugins and a list of options gets thrown up — these are debug messages that would help which are existing in the plug-ins. Where is this useful? Well one practical case or cases we found this one quite popular is for people using DOM — sometimes to validate that these plugin indeed give the correct data to the upper layer — for example, the source length of a construct as shown below in the child eclipse.

**Abb. 40:**

Junit

The keep alive option:

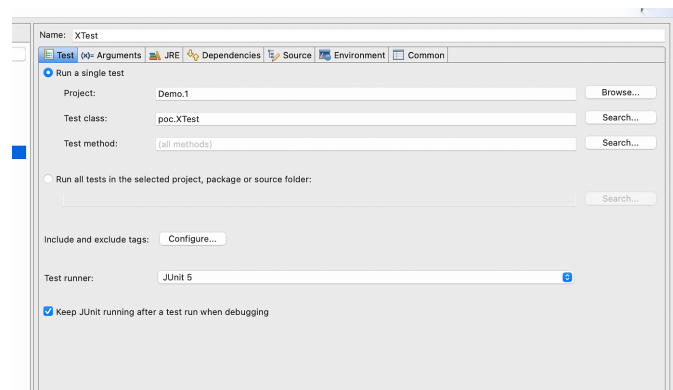


Abb. 41:

This is useful when the start-up times are more — no example here

And one point to note here is that the tab is not “main” but “Test”

Junit Plug-in Test

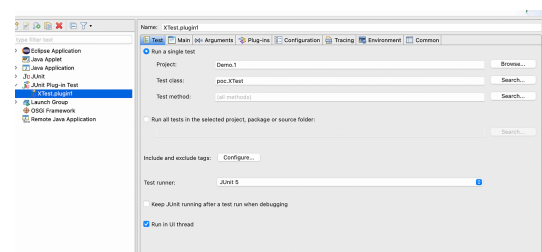


Abb. 42:

Here as expected you will have a “Main” tab, a “Test” tab and a “Tracing” tab.

Prototypes:

More like a template -> Create one prototype using

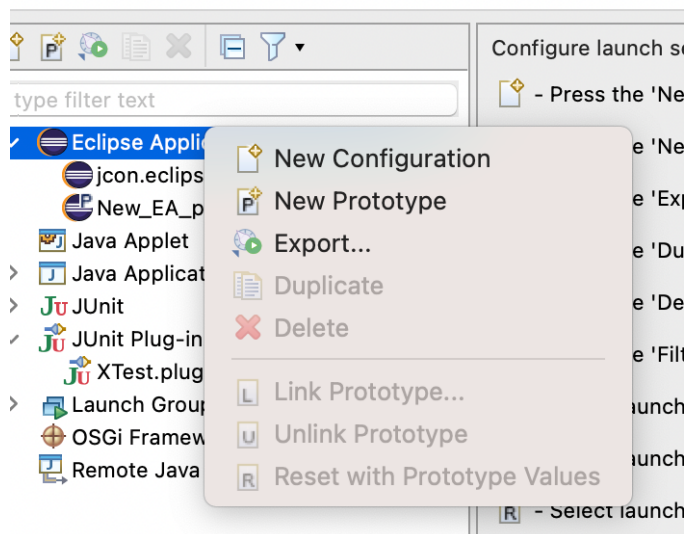


Abb. 43:

And now, click on “Prototype” tab for the list of options:

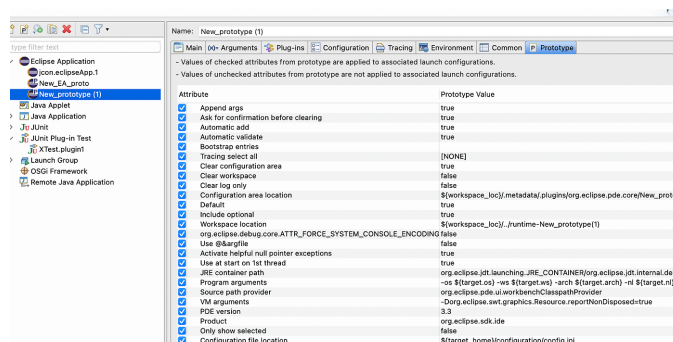


Abb. 44:

What's the use? One place I found useful is again while testing out preview features — say create one prototype — lets call it EA, standing for early access and add a few more VM arguments as shown below:

☒	Program arguments	-os \${target.os} -ws \${target.ws} -arch \${target.arch} -nl \${target.nl} -consoleLog
☒	Source path provider	org.eclipse.pde.ui.workbenchClasspathProvider
☒	VM arguments	-Dorg.eclipse.swt.graphics.Resource.reportNonDisposed=true --release 21 --enable-preview
☒	PDE version	3.3

Abb. 45: New options appended: — enable-preview — release 21

But how do we use this?

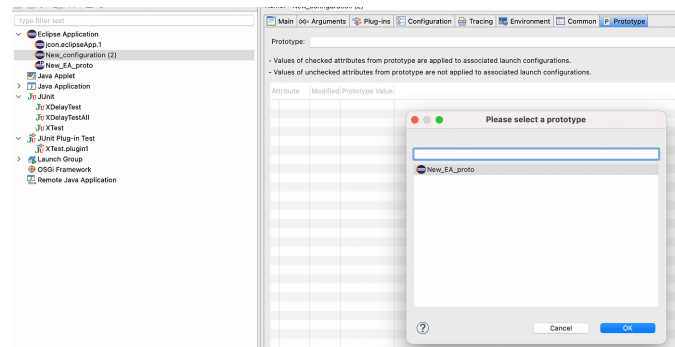


Abb. 46:

Create a new configuration and link to the prototype you want. An easier method is to click P

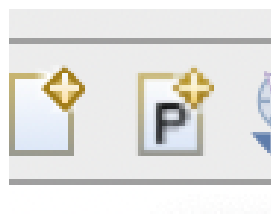


Abb. 47:

Debug Favourites

Click to add a configuration that's faster to access

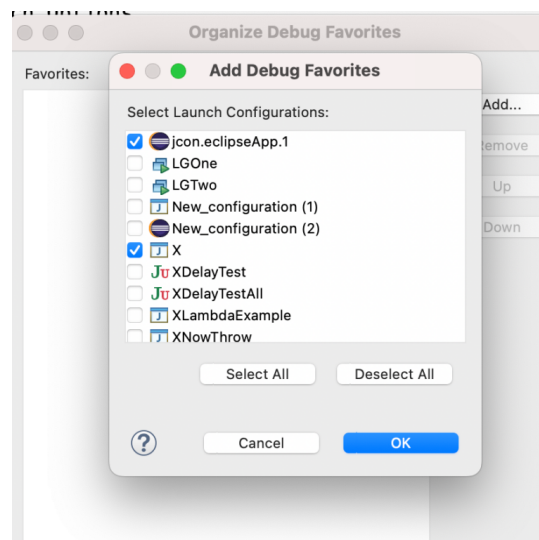
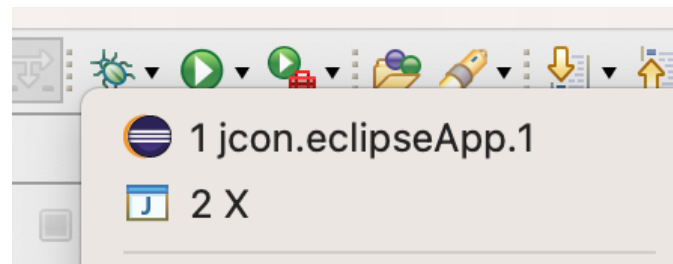


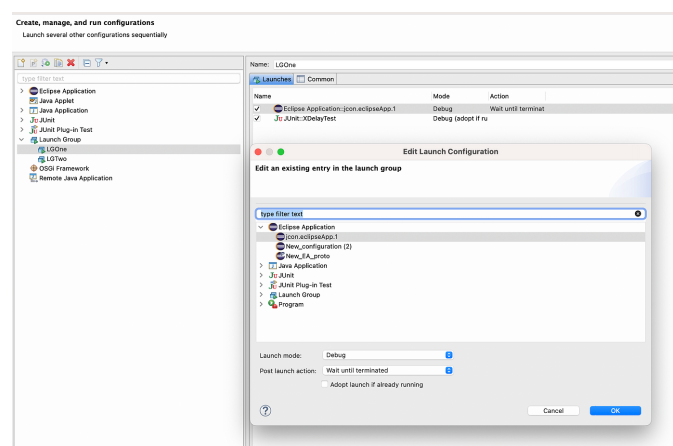
Abb. 48:

Use up and down as well. And these will be added as favourites as shown below:

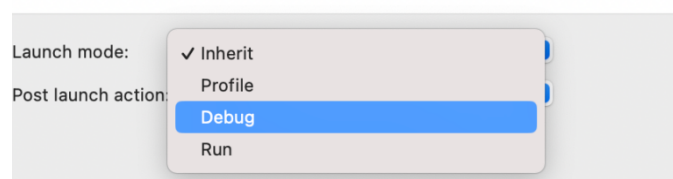
**Abb. 49:**

Launch Groups

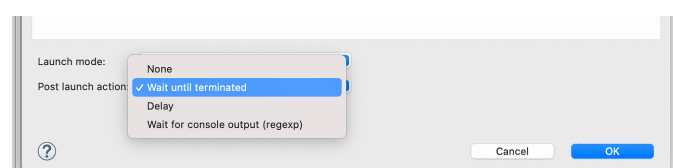
This is an interesting thing — sometimes you want run a few launches together — A typical use which I find is when I test my Java compiler — I want to run the junit test case, as well as to figure out how it behaves when a child eclipse is spawned. Though fundamentally, the bottom-most layer is the same, the upper layers are different in these cases and hence if the fix is in bottom most layer, it helps to launch these two together. For that we have something called launch groups

**Abb. 50:**

And these do have interesting options as well — eg: Launch mode — remember, launch is not just “Debug”, you can “run” as well.

**Abb. 51:**

More interesting option is the “post launch option” — these have some intriguing options again:

**Abb. 52:**

The above “Wait until terminated” will cause the Junits to run one after the other..More interesting option of the below:

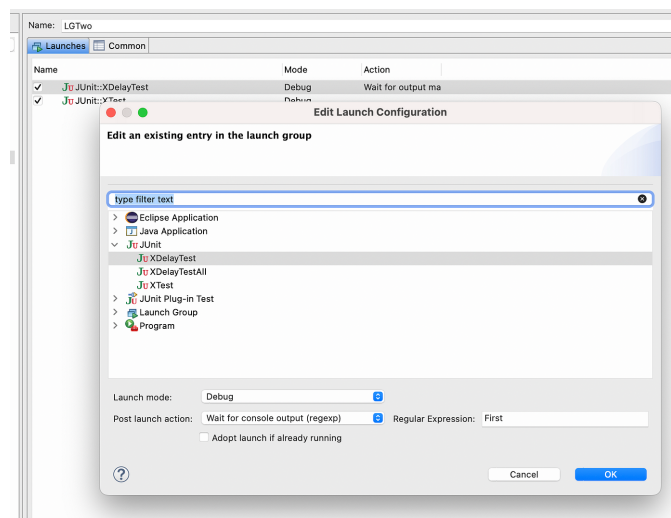


Abb. 53:

The second test will happen after “First” gets prints on to the console — In the XDelayTest, the first test prints that message — so essentially, the first test of XDelayTest finishes and then these two run in parallel:

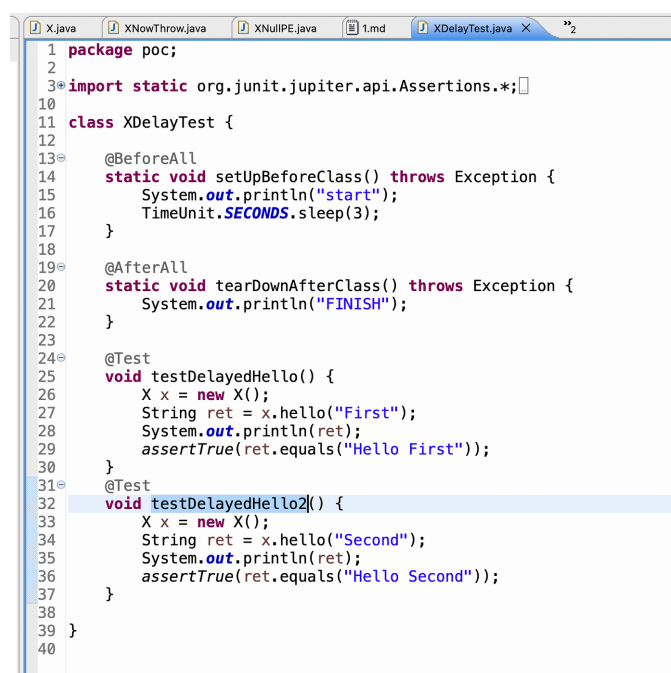
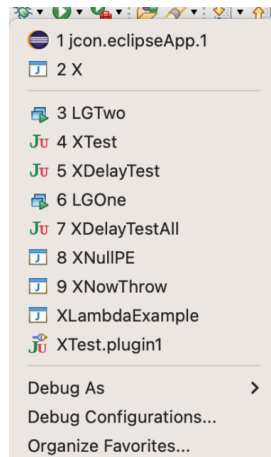


Abb. 54:

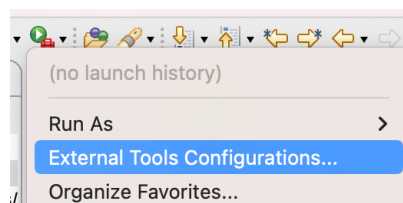
Here, “First” gets printed and then the XTest of LGTwo starts.

And trivia: click on ctrl-<n> to invoke the launch no — n

**Abb. 55:**

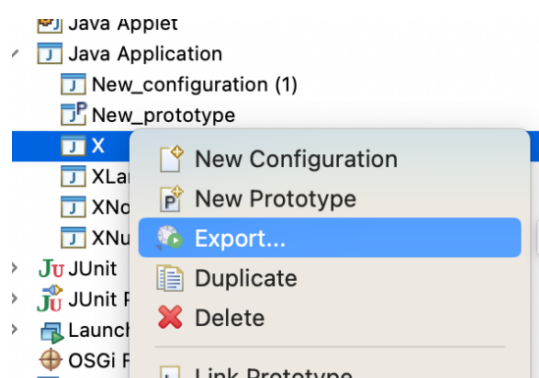
External Launch Configuration:

You can add your own launcher and create a new external launch configuration as shown below:

**Abb. 56:**

Exporting and Importing a Launch Configuration

You just want to export a config to a file:

**Abb. 57:**

You can import using the Eclipse preferences..

Go, Launch your Eclipse!!

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



Agile, Scrum, Kanban and the Lies We Tell Ourselves About Creating Value

Agile

Scrum

Kanban

Author: Marin Niehues

Marin Niehues is Chief Strategy Officer at PLOIN GmbH and advises companies on how to combine strategy and working methods in a meaningful way. He shows how to achieve real success with a clear focus and solid structures.



LinkedIn

But often, the excitement dies down. Companies feel like Agile isn't living up to its promises. Teams feel trapped in rituals like standups, retrospectives, and Kanban boards, with little real improvement. Meanwhile, organizations often keep using the same leadership styles, the same lack of psychological safety, and the same outdated metrics. People do the actions of sprints or continuous flow, but see only a little benefit from the big changes.

For more than 20 years, Agile frameworks like Scrum, Kanban, and Extreme Programming (XP) have been praised as the solution to ongoing problems in product delivery, project management, and value creation. Consultants have said that with daily stand-up meetings, sprints, user stories, and visually appealing boards, organizations will release better products faster, align teams seamlessly, and impress customers on an ongoing basis.

Why does this happen? One major oversight is our collective assumption that "Agile," as defined by frameworks like Scrum or Kanban, must be a universal solution for every context and every product challenge. Second, we assume that "linear" project management methods are no longer useful. The truth is more complicated. Agile and linear approaches can be very powerful when used in the right way and supported by the right mindset.

In this article, we will explore the deeper, often unspoken challenges that get in the way of both Agile and traditional linear methods. We will discuss how the most important factor in creating value is not a particular framework but rather the organizational and cultural context. This context includes your leadership approach, your technical discipline, and, most importantly, your commitment to continuous learning and adapting. If we see Agile frameworks (like Scrum, Kanban, and XP) and linear methods (like waterfall and stage-gate) as just a few tools in a bigger toolbox, we can finally deliver value to our customers.

The Underlying Tension: Why "Agile vs. Linear" Is a False Dichotomy

For years, project managers have been debating whether Agile and linear (or predictive) methods are at odds with each other. Traditional linear methods, also called "Waterfall," follow a step-by-step sequence:

1. **Requirements analysis**
2. **Design**
3. **Implementation**
4. **Verification**
5. **Maintenance**

Each step is done before moving on to the next one. This is usually done with formal approvals. This approach works well when the requirements are clear from the start and the environment is stable. Some examples of this are construction projects, regulated industries, or highly standardized manufacturing. When used in these situations, linear frameworks can be very good. They provide clarity, (ideally) predictable steps, and a detailed roadmap that aligns well with regulatory or compliance demands.

Agile frameworks were created to deal with situations where things are always changing and not many things are known for sure. One example of this is the development of software in markets where things are changing quickly. In these situations, it is very important to be able to make quick changes, test ideas, and change course based on feedback. Scrum, Kanban, XP, and other Agile methods focus on frequent collaboration, where teams can organize themselves and deliver small improvements over time.

There's a lot of debate about this, especially when companies try to apply the same approach to everything. The truth is, the best approach depends on the situation. Some jobs can be handled well by a more organized approach, while other jobs need to be more flexible and can't be done with a set plan.

In today's world, we understand that things can be complicated. Sometimes, simple tasks can be done with simple methods. But when things get complicated and change a lot, it's better to use different methods. Agile methods are often better for complicated problems that change often. Companies often mix these two types of methods to deal with the complexity of their products, projects, and plans.

A Growth Mindset as the Baseline for Success

No matter if you use Agile frameworks or more traditional, linear approaches, the key to delivering real value is having a growth mindset. Psychologist Carol Dweck introduced the term "Growth Mindset," which means believing that abilities and intelligence can be developed through dedication and hard work. This mindset helps people overcome challenges and stay eager to learn.

1. **Willingness to Experiment**; Agile is all about trying new things, and the same is true of effective project management. In project management, managing risk can be a kind of testing. Whether you're testing a hypothesis in a Scrum sprint or running a pilot study in a phase-gate process, having a growth mindset means learning from both successes and failures.
2. **Openness to Feedback**; Frequent feedback loops are a key part of Agile. But even in a more traditional approach, effective teams still do stage reviews that include feedback from stakeholders and customers. If the project team and leadership are open to new ideas, they'll embrace feedback that might require changes to the plan.
3. **Continuous Learning and Adaptation**; When a growth mindset is common in an organization, people there are always looking for ways to improve. They don't just do things the way they've always done them. They always want to get better at planning (whether they plan in a step-by-step way or in a way that involves trying out different ideas) and what they deliver (the things that really matter). They see frameworks as tools that can be improved over time, not as rules that should be followed without question.

An environment lacking a growth mindset will fail to maximize either Agile or linear frameworks. Instead, it will stagnate, resisting change and punishing failure, which ultimately cripples innovation and adaptation regardless of the method in use.

The Roots of “Agile” and Its Limitations

The **Agile Manifesto** (2001) called for new ways of working, emphasizing:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Since then, frameworks like Scrum, Kanban, and Extreme Programming (XP) have been created to put these values into action. The idea was (and still is) appealing: if teams focus on delivering small amounts of value regularly, constantly check and adjust their work, and work together, they can respond quickly to changing customer needs.

But many organizations have adopted Agile methodologies without fully understanding them.

1. **Ritualized Ceremonies without Real Change**; Standups, sprints, and retrospectives become meaningless. People “go through the motions” but rarely use these ceremonies to deal with big problems (like micromanagement, incentives that are not aligned, and poor technical quality).
2. **Partial or Shallow Technical Practices**; Many people do not pay enough attention to technical issues and working off tech debt often leading to the underlying code quality or systems architecture issues remaining unresolved.
3. **Leadership Misalignment**; Sometimes, executives say they want “faster output” using Agile methods, but they still use the old top-down way of managing teams, which destroys autonomy and empowerment.
4. **Cultural Mismatch**; Many organizations don't have the right environment for people to work together. They need to have psychological safety and trust. Without these, no framework will work.

Agile frameworks can enhance strong foundations, allowing for quick feedback loops, experimentation, and close collaboration. However, they don't replace the need for clarity of purpose, clear goals, a strategy, a healthy organizational culture, and supportive leadership..

The Validity of Linear Methods

Today, many people are excited about Agile transformations. Because of this, it's easy to think that linear methods are old-fashioned. But linear or predictive project management can actually be very effective in certain situations.

- Requirements are relatively stable and well-understood at the outset and can not be negotiated – especially in a highly regulated environment
- The regulatory or compliance environment demands formal documentation of each phase
- Significant portion of the work is repetitive or standardized, such as manufacturing a known product in large quantities

For example, think about making medicines. In this case, it's very important to follow the rules set by the FDA's Current Good Manufacturing Practices (cGMP). A step-by-step process makes sure that each important step (like creating the medicine, testing it in animals, testing it in humans, and increasing the amount of medicine made) is finished and documented with few mistakes.

Another example is when a government agency is working on a civil engineering or construction project. In these cases, there are often very specific design details, budgets, and deadlines set in contracts. While agile methods can be useful in the early stages of designing a project, once construction begins, a more traditional, step-by-step approach is necessary to make sure that building codes, safety standards, and engineering tolerances are followed.

It's important to understand that even in these areas, having the ability to adapt and think in a growth mindset can still lead to positive results. For example, many large construction companies are starting to use Lean Construction methods. These methods involve teams using visual task boards (similar to Kanban) and daily meetings to identify and address issues. They still follow the usual project phases and approval procedures.

The Common Dysfunctions That Undermine Both Approaches

Whether an organization chooses Agile frameworks or more linear methods, they often stumble for the same underlying reasons:

1. Lack of Clear Vision and Goals

- Teams don't understand why they're building a product or completing a project.
- In a linear context, this leads to missed specifications or misaligned deliverables.
- In Agile, it manifests as constant pivoting without any coherent strategic direction.

2. Siloed Departments and Poor Collaboration

- Traditional organizations often separate Development, QA, Operations, and Marketing, each with its own objectives.
- No matter the method, these silos create bottlenecks and communication gaps.
- The transition from one phase to the next (in linear projects) or sprint to sprint (in Agile) becomes difficult.
- Actually delivering something and getting it into a productive state is a huge pain with a lot dependencies involved that hinder the market-readiness

3. Command-and-Control Leadership

- Executives who micromanage every step along the way don't let the team make decisions for themselves.
- In a linear approach, this fosters mindless compliance rather than engagement.
- In Agile settings, it destroys the self-organizing dynamic and growth mindset essential for iterative improvement.

4. Misaligned Incentives and Metrics

- Organizations that reward meeting deadlines instead of achieving real results end up with surface-level success indicators.
- Agile teams that are measured purely on "story points" rather than value delivered to the customer suffer the same fate.

5. Lack of Psychological Safety

- Individuals are afraid to share concerns, admit mistakes, or propose innovative ideas.
- Agile ceremonies such as retrospectives lose their meaning because no one feels safe enough to speak openly.
- In linear projects, issues remain hidden sometimes even in a formal "phase review" which leads to them being uncovered far too late.

6. Insufficient Technical Practices

- Whether writing code or executing a manufacturing process, subpar technical discipline leads to rework, delays, and poor quality.
- Agile demands robust engineering practices to release iteratively. Linear demands rigorous documentation and planning. Either can fail if technical excellence isn't pursued.

These problems don't care what framework you're using. The big mistake is thinking that a method like Agile will suddenly fix deeper problems in how your organization and culture work.

The Real Key: Value Creation Over Ritual

Value is the ultimate currency of any work system. Value isn't about how many features you build or tasks you complete; it's about how well you solve real customer problems, generate revenue, and deliver results that matter. Both agile and linear methods can help you get there, but only if you always keep value in mind.

1. Value Definition

- In an Agile context, teams often define value in user stories, focusing on what the end-user or customer actually needs.
- In a linear context, the scope statement or requirements document sets the baseline for what value looks like.
- In both cases, clarity on why a feature (or project requirement) is valuable to the customer and user of our work is most important thing to define.

2. Measurement and Feedback Loops

- Agile teams embed short feedback cycles – demos, reviews, and iterative releases – to evaluate customer value quickly.
- Linear projects rely on phase gates and milestones to evaluate progress.
- The difference is frequency and adaptability. However, if you focus on the result and are willing to change your approach if feedback indicates that you are not on the right track, a linear project can also include elements of adaptation.

3. Outcome vs. Output

- “Output” is the volume of work done: tasks completed, lines of code written, story points burned.
- “Outcome” is the impact of that work: customer satisfaction, cost savings, revenue growth, time-to-market reduction.
- Both Agile and linear processes can fall into the trap of measuring output (e.g., sprint velocity or Gantt chart progress) instead of real outcomes.

4. Inspect and Adapt

- The Agile principle of “inspect and adapt” isn't just for Scrum or Kanban. Other methods like linear approaches also have reviews, but they're less often.
- The question is: do you have the courage to act on the findings? Too often, organizations only go through the ceremony (retrospective or phase review) and ignore its implications.

When you anchor your work in value creation and maintain the agility to respond to real feedback, you're practicing the essence of agile value creation – no matter how formally “Agile” or “linear” your framework appears on paper.

The Stacey Matrix: A Key to Choosing Your Method

One of the most useful models for deciding how to approach project management – whether agile, linear, or a hybrid – is the Stacey Matrix. Developed by Ralph Douglas Stacey, this framework helps categorize work along two dimensions:

1. **Clarity of Requirements** – How well do they know where to go?
2. **Clarity of Methodology and Technology** – How much predictability or clarity exists in the technology, requirements, and context?

By plotting these on a matrix, organizations can distinguish between different levels of complexity:

1. Simple (Clear)

- Here, both requirements and methods are well understood. Processes are repeatable, and few unknowns exist.
- Traditional, linear management excels in this space because the path from start to finish is obvious and uncontroversial.
- Example: routine manufacturing tasks with standardized operating procedures.

2. Complicated

- The objectives are understood, but the methods require expert analysis and may not be obvious to non-specialists.
- While a linear approach can still work – often guided by experienced professionals – some elements of Agile or iterative processes (like prototyping or repeated verification) can help manage risks and improve solutions.
- Example: designing complex but well-defined systems, such as an engine, where specialists must carefully plan each step but the end goal is relatively stable.

3. Complex

- Requirements and solutions are unclear, evolving, or uncertain. Multiple variables interact in unpredictable ways.
- Agile approaches often thrive here because short feedback loops allow teams to adapt as they learn.
- Example: software product development in a competitive, fast-changing market, where you discover ideal features only through user feedback.

4. Chaotic

- Extreme instability and ambiguity dominate. Objectives or methods might be unclear, and urgent intervention is necessary.
- The priority is to stabilize the situation quickly – often through short experiments or triage.
- As soon as possible, teams should move the work out of the chaotic domain into a more manageable (complex or complicated) state.
- Example: crisis management (e.g., responding to a major cybersecurity breach).

The Stacey Matrix helps to avoid the pitfall of using the same management style for all projects. If you're dealing with Complex issues, an Agile approach might work well. For Simple or Complicated projects, a more traditional or even a hybrid approach might be better.

Where Scrum, Kanban, XP, and Linear Methods Shine

Scrum

- Best for teams developing a product in an environment where requirements evolve quickly.
- Time-boxed sprints provide focus and structure, while roles like Product Owner and Scrum Master create clarity.
- Suited to complex work with uncertainty, where rapid feedback cycles are critical to course-correct.

Kanban

- Ideal for steady flows of incoming requests (e.g., support tickets, minor enhancements) or teams needing flexible scheduling.
- Visualizing work-in-progress (WIP) and setting WIP limits helps expose bottlenecks and optimize flow.
- Continuous adaptation rather than fixed sprints fosters incremental improvement.

Extreme Programming (XP)

- Strengthens teams with rigorous technical practices: test-driven development, pair programming, continuous integration.
- Highly effective when high code quality and frequent releases are non-negotiable.
- Can be combined with Scrum or Kanban to add engineering discipline.

Linear / Predictive Approaches

- Staged approach ensures each phase meets strict quality or regulatory criteria.
- Useful when the scope is well-defined upfront, and the cost of rework is prohibitive.

It's perfectly legitimate – and often wise – to mix these methods. For example, many organizations use a phase-gate process at the portfolio level to greenlight projects based on strategic alignment, and then delegate how teams work (Scrum, Kanban, or a hybrid) based on the complexity of the project. In the end the goal should be to create customer value at the lowest possible cost while maintaining a high level of quality and adherence to all applicable regulations.

Cultivating a Growth Mindset Across the Organization

No matter if you are leading teams that work in an agile way or running a program that follows a more traditional path, it is important to make sure that a growth mindset is present at all levels. Here is how you can do that:

1. Leadership Modeling

- Leaders set the tone by encouraging experimentation, talking about their own learnings and be open about new ideas.
- This signals that it's safe to fail small and learn fast – essential for both rapid iteration and stage-gate innovation.

2. Rewarding Learning, Not Just Success

- Reframe “failure” as a learning opportunity, not a personal shortcoming.
- Recognize team members who propose improvements, run experiments, or take the initiative to solve tough problems – even if the outcomes are imperfect.

3. Inclusive Decision-Making

- Collaboration should not be confined to Agile ceremonies or linear reviews.
- When teams see their input valued, they develop a shared sense of ownership and accountability for the successful value delivery.

4. Ongoing Coaching and Support

- Provide resources for professional development and skill-building, whether technical (e.g., TDD in software) or conceptual (e.g., risk analysis in linear planning).
- Offer coaching at all levels, from executives to frontline teams, to reinforce continuous improvement.

A growth mindset changes retrospectives and phase reviews from simple checks of boxes into opportunities for a meaningful change. It brings people together to solve problems and deliver a finished product to the customer instead of blaming each other. This basic idea is the foundation of any successful method, whether it is Agile or linear.

Common Dysfunctional Patterns to Avoid

1. “We’re Doing Scrum, but...”

- ...our teams have no real autonomy.
- ...our increments never actually reach end-users for feedback.
- This superficial implementation leads to cynicism and stagnation.

2. “We Put Up a Kanban Board, but...”

- ...we ignore WIP limits and never visualize the actual bottlenecks.
- ...our managers still use the board solely for micromanagement.
- ...we do zero system-level improvements, making Kanban a fancy to-do list.

3. “We’re Lean, but...”

- ...we focus solely on cost-cutting rather than building quality in.
- ...our leaders still operate in a top-down manner, stifling the continuous improvement spirit.
- ...employees fear retribution for surfacing problems.

4. “We’re Linear, but...”

- ...we keep forcing stage gates even when the requirements are unclear and keep shifting.
- ...we skip necessary reviews or cut corners on compliance to meet arbitrary deadlines.
- ...we miss the chance to incorporate incremental learnings in each phase.

In all these situations, the problem isn't the framework itself, but the way people act and think in the organization that makes it ineffective.

Busting the Myths

- **Myth #1: “Agile will fix our broken culture.”** Reality: Culture is shaped by shared values, behaviors, and leadership actions. Agile can highlight issues, but it cannot fix them on its own.
- **Myth #2: “Linear methods are outdated and irrelevant.”** Reality: In regulated or standardized environments linear methods offer efficiency and clarity unmatched by iterative approaches. They remain the go-to for many standardized or heavily regulated industries.
- **Myth #3: “We’ll succeed if we follow the [Scrum / PMBOK / Kanban] Guide to the letter.”** Reality: Framework guides provide structures and best practices, but real success depends on how well you address deeper issues—vision, trust, technical discipline, leadership style.
- **Myth #4: “Our teams can be Agile even if the rest of the organization isn’t.”** Reality: You can pilot change in small pockets, but true transformation requires cross-functional alignment, executive support, and often a change in organizational design.
- **Myth #5: “Agile is only for software; linear is only for manufacturing.”** Reality: Agile principles such as iterative learning and collaboration can benefit marketing campaigns, sales workflows, and even parts of large-scale engineering. Conversely, linear methods have a place in certain software scenarios where scope and risk are highly constrained.

Strategies for Achieving Real, Sustainable Value

1. Contextual Assessment

- Before adopting or overhauling any methodology, assess the complexity, volatility, and regulatory constraints of your projects.
- Use frameworks like the Cynefin framework or a simple risk/complexity matrix to determine if Agile, linear, or hybrid methods make sense.

2. Strong, Aligned Leadership

- Leadership must articulate a clear vision and be willing to change their own behaviors.
- The shift to an outcome-focused culture often requires new incentives, metrics, and management practices.

3. Invest in Coaching and Facilitation

- Skilled Agile coaches can help teams adopt Scrum or Kanban effectively, but you may also need leadership coaches to foster cultural change at the executive level.
- Similarly, project management professionals (PMPs) or Lean Six Sigma experts can refine linear processes and optimize organizational workflows.

4. Start with a High-Impact Pilot

- Target a strategic project that can benefit from either Agile or improved linear methods.
- Dedicate resources, secure executive buy-in, and ensure cross-functional collaboration.
- Publicize successes (and lessons) to build momentum for broader organizational shifts.

5. Address Technical or Process Debt

- In software teams, tackle code quality and automate testing. In manufacturing or construction, standardize procedures and ensure safety and compliance.
- Remove the friction that impedes rapid iteration or disciplined linear checkpoints.

6. Measure What Matters

- Shift from output-based metrics (story points, tasks completed) to outcome-based metrics (customer satisfaction, time-to-market, ROI, defect rates, risk reduction).
- Align rewards and recognition with these outcome metrics to drive the right behaviors.

7. Continually Reflect and Adapt

- Retrospectives or phase reviews should lead to concrete action items, not just discussion.
- Encourage transparency: celebrate the success of improvements, and learn openly from failures.
- Scale effective practices to new teams or departments once they are proven in pilots.

Measuring True Value in Any Approach

Ultimately, whether you're sprinting every two weeks or marching through a linear plan, value must be your goal. A growth mindset and a supportive culture enable teams to constantly asking questions:

- **What customer or business problem are we solving?**
- **How do we measure success in solving that problem?**
- **What feedback loops confirm whether we are on the right track?**
- **Are we prepared to pivot if the data suggests a change in direction?**

By shifting the focus from “How many features did we ship?” to “What measurable impact did we deliver?” and “What problem did the customer solve with our product or service?” you are creating real value for your customers. Agile's frequent feedback cycles make this easier in dynamic environments, while linear's structured checkpoints make it easier in more predictable environments.

Conclusion: A Call for Contextual, Value-Driven Approaches

Agile – in the form of Scrum, Kanban, XP, or other frameworks – can be immensely powerful. Linear methods can be equally effective in the right environment. But both approaches can fail spectacularly if organizations ignore the core building blocks: leadership support, psychological safety, technical discipline, cultural alignment, and a **relentless focus on value**.

Instead of framing your delivery strategy as a either “Agile or Waterfall”, recognize that both are merely means to an end. The real question is: What is your organizational context and how can you most effectively create value for your customers?

Companies must move beyond the superficial adoption of ceremonies or checkpoints. They must aim for a growth mindset that empowers teams to continuously learn, adapt to new information, and refine their practices-whether iterative or linear-based on what truly drives results. When the entire organization aligns behind a clear vision and purpose, trust replaces micromanagement, and frameworks become enablers.

So the next time someone says, “We need more agility,” or “Agile doesn't work,” take a step back. Look at your culture, your structures, your goals, and your mindset. Assess whether your situation calls for an iterative loop of inspect-and-adapt or the clarity and rigor of a well-defined project plan-or perhaps a mix of the two. Above all, keep asking: **Where is the customer value, and how do we know we're delivering it?**

That's the only metric that really matters.

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

APR
20-23



www.jcon.one

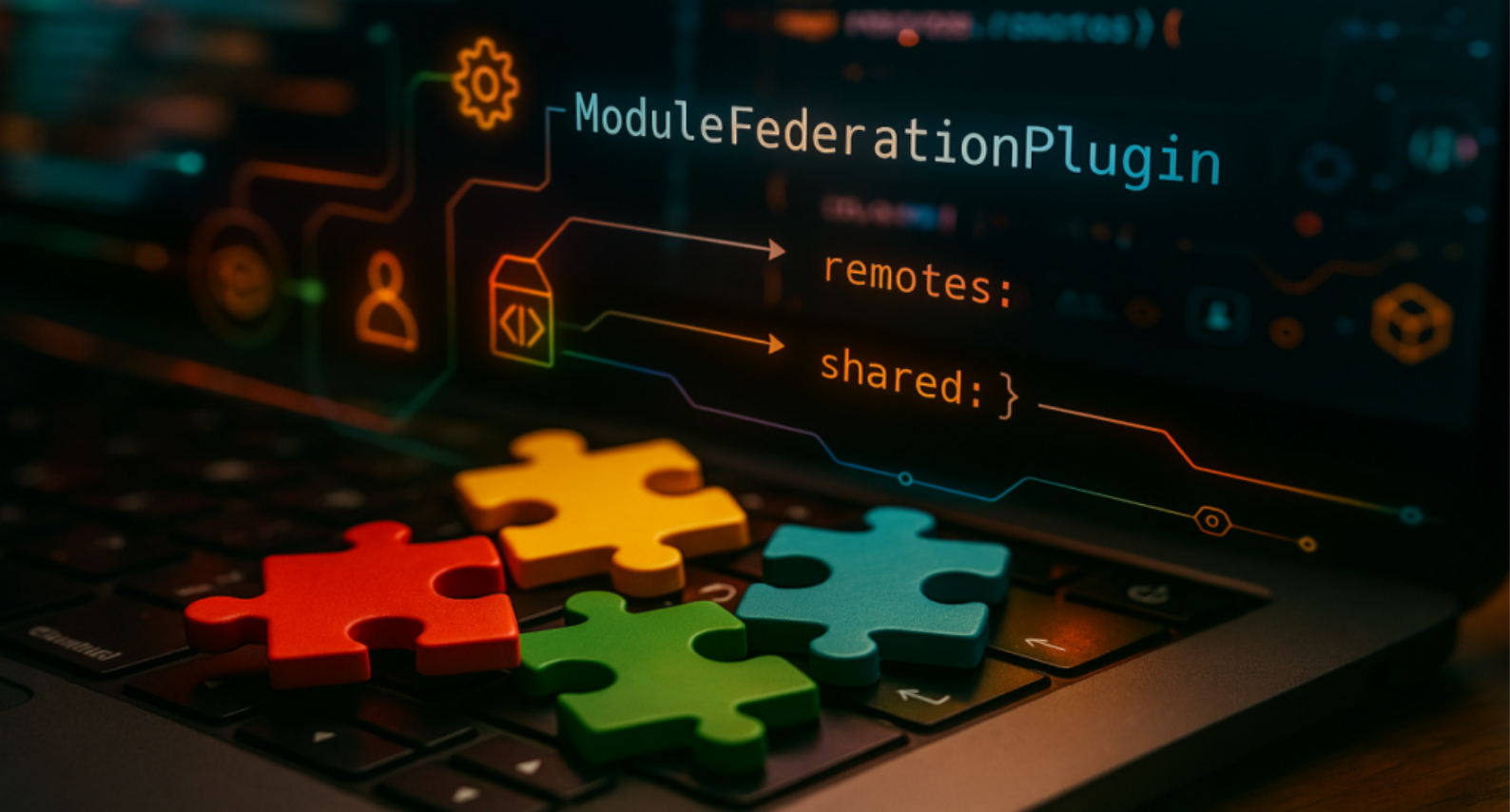
JCON2026

JCON EUROPE 2026

International Java Community Conference

📍 CINEDOM COLOGNE

JAVAPRO



Integrating Micro Frontends into Existing Applications – Lessons Learned

Reusability # TypeScript # WebComponents

Author: Christian Siebmans

Christian Siebmans is an expert in the development of enterprise web applications. He is passionate about user interfaces, because he is a strong believer that frontends should make people's work easier, not harder.



LinkedIn

The application has been in production for several years – can you jump on a hype like micro frontends in such a scenario? We took the plunge and redesigned our existing frontend to incorporate the micro frontends pattern. What motivated us to do this and what we learned along the way, you will discover in this article.

Two Teams and Lots of Reusability

We had the following situation: Two project teams within the same company develop web portals independently from each other. However, management had mandated that as many components as possible should be shared and reused among the two projects. Reusability was a core goal set for both projects.

Due to the use of a microservice architecture, it was easy for the backend. For instance, the authentication microservice was shared between the projects. However, each web portal would host its own instance of shared microservices.

In the frontend, reusability was also possible in many areas: The design system as well as the login workflow were shared across projects, for example.

Another example of frontend reusability was the central administration application. This application allows specialist users to setup new principals, lock individual users, create system-wide announcements, and so on. These functionalities on the backend live within the authentication microservice. Since this microservice is shared between projects, it made sense to share the frontend as well.

Problems Arise

The web portals worked without a hitch in production like this for many years. However, then we were handed a user story saying that we should add a cleanup function for specific data. The problem here is that the cleanup happens in a specific microservice that is not shared between projects. This sounds trivial, but sadly it was not. Our initial approach was to integrate the functionality and only show it to the user based on the user's authorities.

How We Get Types for Backend Communication Automatically

We use TypeScript in our frontends for static type checking. To get types for our microservices' APIs we generate an OpenAPI specification using a Maven plugin during the build process of the microservice. We then zip the specifications and use Maven to store them in our Nexus repository. In our frontend projects we then use a Maven POM to download the specifications for all of the necessary microservices. A tool then generates TypeScript types out of these specifications. Those types are for backend requests and responses. In theory this sounds awesome, but in practice it means that the frontend cannot be build before the necessary backends have been built, because we would be missing the API specifications for our types.

A Quick and Very Dirty Fix

To implement our user story we have to reference our specific microservice in our shared central administration application. Ouch! You can see a graphical representation of our dependency debacle in figure 1.

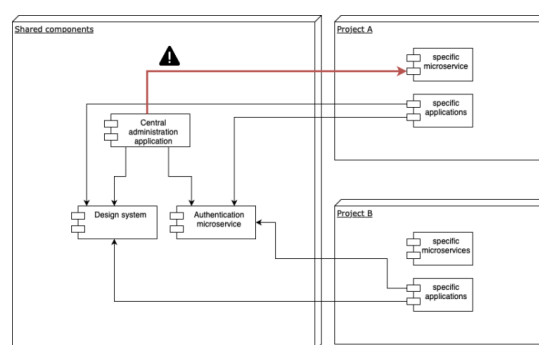


Abb. 58: Our dependency hell that we created for ourselves.

Most of the times both projects coordinated a release date, when all release builds would be built in order. This worked really well for the first couple of releases, while we had added the dependency to the specialist microservice. However, suddenly one of the projects had to postpone its release date. To accommodate the shared components it was agreed upon that those would be built on the earliest release date to not block the other project.

This worked fine, until it came to the release build of the central administration application, since it was referencing the specialist microservice, which had not been built yet. The TypeScript types could not be generated and therefore the application could not be built. Temporarily, we could solve the situation quickly by changing the version of the API specification to use. The specification hadn't changed since it was first introduced, so we just used the specification from the previous release.

However, it was clear that this could only be a temporary, dirty fix. So, we began looking for a more permanent solution. We thought about multiple different solutions, however extracting specialist project functionality into micro frontends certainly was the most appealing: we would have separate (build) projects for our micro frontends, which also meant we could build and release them independently from the main application.

To bundle our source code into the JavaScript code that is shipped to clients in production we use webpack. Its purpose is to orchestrate transpiling TypeScript into JavaScript and to optimize both the code and assets such as fonts and images. Thankfully, webpack 5 introduced a new feature called module federation especially built for micro frontends.

What Is Module Federation?

Zack Jackson, the creator of module federation, describes it as follows:

„Module federation allows a JavaScript application to dynamically load code from another application — in the process, sharing dependencies, if an application consuming a federated module does not have a dependency needed by the federated code — Webpack will download the missing dependency from that federated build origin.”

Jackson, Zack. Webpack 5 Module Federation: A game-changer in JavaScript architecture. [medium](#)

Webpack provides the *ModuleFederationPlugin* out of the box to define and load federated modules in a webpack application. A federated module is made up out of the following parts:

- *name* is the name of the federated module and names its scope
- *remoteEntry* is the entry point which is used to initialize the federated module
- *remotes* are a list of federated modules that the given federated module loads and uses itself
- *exposes* defines a list of elements that can be used by consuming federated modules
- *shared* is an object that describes how dependencies are handled and if certain dependencies should be shared between federated modules (for example as a singleton)

Every webpack project can be a federated module. Listing 1 shows an example federated module that only exposes a file *helpers.js* at runtime.

```
1  const { ModuleFederationPlugin } = require("webpack").container;
2  const pluginConfig = {
3    plugins: [
4      new ModuleFederationPlugin({
5        name: "example",
6        exposes: {
7          "./helpers": "./src/helpers",
8        },
9      },
10     ],
11   },
12 }
```

Listing 1: An example of a simple federated module that only exposes a single file.

Building Plugins for Our Application

At the very beginning we asked ourselves what the API for our application should be. Since we could no longer bundle the code at build time, we had to create a suitable interface to extend our application through federated modules. We took inspiration from our JetBrains development environments: They are massively extendable by plugins. This sounded like the perfect analogy for our application. Of course our plugin interface would be much simpler, since the plugins to load would be determined at runtime and users did not need to configure or manage plugins themselves. For users plugins should be invisible.

For our plugin architecture we built a TypeScript interface. This interface defines all the things necessary to to define a plugin:

- a list of all the routes a plugin defines
- a scope for the plugin, which is used to separate route, store, etc. entries from the rest of the application. It must be unique.
- a render method which returns the elements a plugin has rendered for a given route.
- an init method used to initialize the plugin and load its dependencies
- several hooks to communicate with the host application's infrastructure such as router, store etc.

This interface is stored in one of the common, shared frontend libraries. It allows to define application plugins wherever needed and thus to extend applications at runtime.

Implications for Our Code Base

After we had defined the interface for plugins we needed to implement our plugins by implementing it. We already had a frontend library for every backend microservice – the perfect place to implement our plugins. However, this has an interesting implication: Since federated modules are a feature implemented in webpack, this also meant that we needed a webpack project for federated modules. This also meant we needed a webpack configuration per frontend library project and that it would need to be built and bundled with webpack. This in turn meant we needed to alter our build scripts for frontend libraries to allow building webpack projects, too. Last but not least, the resulting bundle needed to be versioned, archived, and deployed to a server. These were massive changes we had to make to our CI/CD pipeline and they were quite complicated given all the scripts were already used in production.

Module Federation and Web Components – a Match Made in Heaven

We develop our applications using web components. Since the web standards are very low-level we use a library called Lit to make the development of components simpler. To register new, lazy loaded components with the browser, we call `window.customElements.define`. Element names must contain a hyphen (so they can be separated from built-in components) and their names must be unique. Otherwise, the browser will throw a critical error. Our components are self-defining, so the call to `customElements.define` is part of the same component source code file. This has the benefit that the component is automatically registered once the component source code is loaded. Listing 2 shows a very simple HelloWorld component that is self-defining.

```
1 import { LitElement, html } from "lit";
2 import { customElement } from "lit/decorators.js";
3
4 class HelloWorld extends LitElement {
5   public render() {
6     return html`<p>Hello world!</p>`;
7   }
8 }
9 window.customElements.define("hello-world", HelloWorld);
```

Listing 2: A simple component with Lit that is automatically defined once the source file is loaded.

We share our components as npm packages. Our frontend libraries as well as our applications use these packages as dependencies. In a standard webpack setup, webpack makes sure that all source files are only part of the bundle once. This also makes sure that we cannot run into the critical browser error mentioned above with a component being registered twice. Sadly, this is not how things work with module federation.

Sharing Dependencies Across Federated Modules

The configuration option `shared` of the `ModuleFederationPlugin` allows us to define how dependencies are shared between federated modules. This setting is quite powerful: dependencies can be marked as singletons or the required version number for a package can be specified. Module federation follows [Semantic Versioning similar to npm](#) here. So a version number like `^1.0.0` is valid.

Listing 3 shows a sample configuration for sharing dependencies.

```
1  const { ModuleFederationPlugin } = require("webpack").container;
2  const pluginConfig = {
3    plugins: [
4      new ModuleFederationPlugin({
5        name: "admin",
6        remotes: {
7          lib: "lib@http://localhost/lib/remoteEntry.js",
8        },
9        shared: {lit: {singleton: true}},
10      ]
11    }
12  }
```

Listing 3: Configuration of a federated module which defines the dependency `lit` as a singleton.

Beware though: The name of the package as a `shared` key means that only elements in the package root will be shared. For example the import is shared between federated modules if the shared key is `lit: import {LitElement} from "lit"`. However, the following import is not shared: `import {ref} from "lit/decorators/ref"`. `Ref` is not exported a root package level, so it is not shared. If you want to share every export of a dependency, you have to append the `shared` key with `/`. For example `lit` becomes `lit/`. There are lots of more configuration options for sharing dependencies that are explained in the [webpack documentation](#).

Importing Federated Modules

At this point we have learned to define federated modules and to share dependencies between them, but how do you consume and use them? Every federated module may define as many federated modules it references in its configuration. This is done through the `remotes` object. Here the key is the name of the federated module. The name will be used to load exposed elements of the module. The value of the object is a string referencing the absolute path to the federated module and the name given to the federated module by its author. (The name used as object key and the name in the object value should match.) Listing 4 shows a very simple example of a federated module `admin` referencing another federated module called `lib`.

```
1  const { ModuleFederationPlugin } = require("webpack").container;
2
3  // Plugins part of the webpack configuration
4
5  plugins: [
6    new ModuleFederationPlugin({
7      name: "admin",
8      remotes: {
9        lib: "lib@https://someserverontheinternet.com/lib/remoteEntry.js",
10      },
11      shared: {"lit/": {singleton: true}},
12    }),
13  ]
```

Listing 4: Plugins part of the webpack configuration showing a federated module referencing another federated module.

To load an exposed module out of `lib` inside of `admin` we can simply import it like this: `import { someExport } from "lib/exposedModule"`. This is very easy and very clean, because it uses standard JavaScript syntax, we are used to.

Unforeseen Dificulties

I have to admit that referencing the absolute path to the `remoteEntry` in the webpack configuration is less than ideal. And it did not work for us! We use staging to deploy our artifacts. This means we need to have the same artifact deployed in different stages. So, we cannot use the absolute URL. Luckily, there is a plugin for that. The *external-remotes-plugin* allows us to place placeholders in the URL specified in the webpack configuration. We can then at runtime set those placeholders using `window.location.origin`. This works for us, because we are deploying all JavaScript to the same machine.

Another problem surfaced in our failing CI build. We were using a Content Security Policy that mandated that all assets loaded must be verified by a SHA-384-Hash. For this task we use the plugin *webpack-subresource-integrity*. It generates the SHA-384-Hashes automatically at build time. However, since we are loading federated modules at runtime, we can no longer determine SHA-384-Hashes for them at build time. After a quick consultation with IT security it was decided to change the Content Security Policy for the central administration application to *self*. This enforces that scripts can only be loaded from our server.

How We Define and Load Remotes Dynamically

The local test was successful, the CI build is passing. For the final test, I gave my changes to my colleague on the other project to verify my changes. Everything works, which is great. After all, they shouldn't notice anything, but he does notice something in his developer tools: The application is trying to load the `remoteEntry` of our federated module, although we never actually import anything from it. It turns out that module federation is not so dynamic after all. This is a problem, because the federated module will never be deployed for the other project.

The solution is to remove all *remotes* from the webpack configuration. To still load remotes dynamically, we can use the `@module-federation/utilities` package. It provides a function *importRemote* which we can use to load an exposed element out of a dynamically loaded remote. This concept is called [Dynamic Remote Container](#). Listing 5 shows how we use *importRemote* to load a plugin.

```

1  import { importRemote } from "@module-federation/utilities";
2  import { LitElement } from "lit";
3  import type { Plugin } from "projects/common";
4
5
6  class MyApplication extends LitElement {
7    // ...
8    protected async setupRemotes() {
9      if (!this.specialAdminFunctionalityNeeded) {
10         return;
11      }
12      const remote = `${window.location.protocol}//${window.location.host}`;
13      // load the plugin class
14      const pluginClass = (await importRemote({
15
16         url: `${remote}/modules/super-special-admin-module`,
17         scope: "super__special_admin_module",
18         module: "../features/admin-feature",
19      })) as () => Plugin;
20      // create an instance of the plugin
21      const plugin = new pluginClass();
22      // call lifecycle hooks on plugin instance
23      plugin.load();
24      plugin.init();
25
26    }
27
28    // ...
29  }

```

Listing 5: Dynamically loading a remote and plugin if it is necessary.

First, in *remote* we initialize the remote URL. URLs to federated modules still need to be fully qualified. The parameters of *importRemote* are the following:

- URL: the absolute URL to the remoteEntry of the federated module
- scope: the scope under which we access our federated module. Must be identical to its name.
- module: the module that should be loaded from the federated module.

The result of our *importReport* call in Listing 5 is our plugin implementation that contains the project specific administrative functionalities. We create an instance of this class and then we call our various lifecycle methods such as *load()* on it, to make it usable within our application. Listing 5 is just for illustrative purposes, in the real world application there is more setup needed to make a plugin work within the application. However, this is due to our plugin architecture and not federated modules.

After those changes, I asked my colleague to test again. Fortunately, he did not notice anything out of the ordinary anymore. He merged the changes and since that day we have been using federated modules in our central administration application. It was quite a bit of effort to get here, but in production this has been running without issues ever since.

Our Application Is Not Ready for Federated Modules

I have mentioned at the beginning of the article that we tried to implement federated modules into an existing application without changing much of the application itself. There are multiple issues which showcase that our application was not necessarily made for federated modules.

My personal favorite is something I discovered during testing: When I navigated to a page rendered by a federated module and then instructed the browser to do a page refresh (for example by pressing F5), the application would route me to the 404 page. The reason for this was timing. The router was started before the federated module was loaded and hence the router did not yet know that the federated module provides such a route. The simple fix was to load all plugins before the router was started.

This is not a perfect fix by any means, but it is also something our application architecture currently does not really account for. Most frameworks such as Angular seem to solve this problem by storing (root) route definitions in the application's initial bundle. However, this trades performance against flexibility. In our case a little delay was feasible (which also only happens on the initial load of the application, since afterwards the browser caches the assets).

Another interesting observation is that our application is now made up out of thousands of small JavaScript files. It is a testament to the optimizations webpack usually would run on a bundle: making sure that most code is bundled together into small chunks. Due to our dependency sharing hints, webpack can no longer anticipate how a dependency might be used. A dependency might not only be used in the initial application bundle, but also in federated modules down the line. However, this is not a problem. Especially since HTTP/2 is optimized for transferring small, individual files. In return, small files have positive impact on caching. Changes no longer mean entire chunks need to be replaced.

Outlook

Our first project with module federation uses our plugin architecture as typed interface. This works well in our case, but of course it would be really nice to just have type definitions for the federated modules.

This is one of the features provided by Module Federation 2.0. Types are generated and served in realtime at development time. This works recursively, so even if a federated module exposes types of a module it consumes itself. In addition necessary types for dependencies like Lit or React are exported automatically. It is a game changer for module federation.

But that is not all: Module Federation 2.0 brings even more to the table. The module federation runtime code has been extracted from webpack and merged into a new runtime library. This makes module federation bundler independent. It allows support for new bundlers like rspack or Vite. In addition, federated modules can now be consumed with any bundler, no matter with which bundler they were built. Moreover, the new runtime library provides more flexibility when loading and initializing federated modules. We used *importRemote* to dynamically load a federated module at run time, but the new runtime library greatly simplifies this.

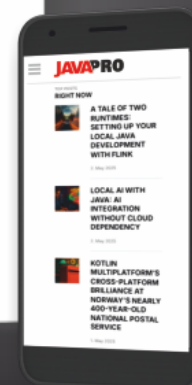
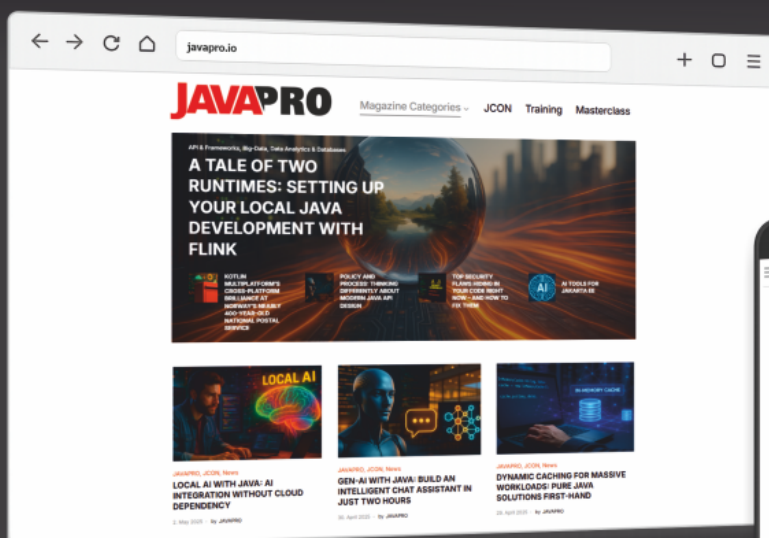
In addition the new release introduced developer tools for Chrome based browsers. This plugin provides information on federated modules at development time such as dependencies and exposed modules. Lastly, it provides hot reloading for federated modules.

Module Federation 2.0 boosts significant upgrades compared to the initial webpack 5 implementation. This is especially true for the developer experience. Our own experiences using it in production as well as the team's continued commitment to evolve module federation show that it has become a fantastic option to implement micro frontends.

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

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



From Breaches to Blackouts: The Human Consequences of Software Supply Chain Attacks

Cybersecurity

Hacking

Vulnerabilities

Author: Steve Poole

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



LinkedIn

Have you ever considered the impact of Cybercrime? How do your choices in terms of applying good security practices matter? Given that the current estimate for the annual bill worldwide is 9 Trillion US Dollars, there is obviously a significant financial reward for the bad guys, but have you ever wondered how that breaks down or if there’s more to their motivations and actions than just money?

This article will explore the financial element and broader human implications.

Setting the Scene – 9 Trillion Dollars

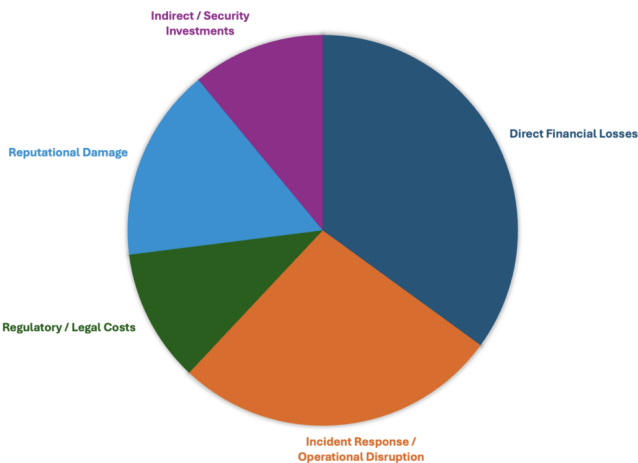


Abb. 59: current 2024/2025 estimate

That’s the current 2024/2025 estimate. The number is growing almost exponentially. Various groups calculate this number, from government agencies to involved commercial companies and not-for-profit organisations.

When you drill into the headline number, the impact becomes more understandable. There are five groups in this breakdown. Other versions calculate the effects differently, but it’s still mostly the same. The percentages of each segment are probably +/-10%, so this picture is broad strokes. Use it for illustrative purposes only.

Segment	Detail	Rough estimate
Direct Financial Losses	Theft, fraud, and ransomware payments	~ \$3.0 trillion
Incident Response and Operational Disruption	Downtime, recovery operations, lost productivity.	~ \$2.7 trillion
Regulatory and Legal Costs	Penalties, lawsuits, fines	~ \$1.0 trillion
Reputational Damage	Customer loss, diminished brand value	~ \$1.5 trillion
Indirect Impacts and Security Investments	Cyber insurance premiums, security system upgrades, preventive measures	~\$ 0.8 trillion

It Starts with Software Supply Chain Attacks

It’s possible to slice these figures in many other ways. We can estimate that ransomware attacks alone account for 20-25% of the total, about the same as data breaches and privacy violations. However you measure it, the attack categories are all individually worth trillions or significant fractions of one.

Software supply chain attacks, though, are different. They are often the foundational element of the others. Software weaknesses allow attackers to access systems. They enable ransomware, data theft, espionage, sabotage, and disruption. Software supply chain attacks are also at the root of most cyber-warfare activities.

No wonder software supply chain attacks are escalating rapidly. Estimates are that close to 1 million software packages are compromised, and the number is growing fast.

What Is a Software Supply Chain?

A software supply chain refers to the entire ecosystem of developing, delivering, and maintaining software, from code creation to deployment. It includes source code, dependencies, CI/CD pipelines, build tools, package repositories, cloud services, and infrastructure.

For engineering managers, these are the elements that you consider when ensuring security, compliance, and efficiency in how software is built and deployed. For executives, it's about risk management, regulatory compliance, and business continuity.

Here's the rub. When we talk about software supply chains, we're not just talking about the elements you control. Your software supply chains include all the software supply chains for *all* the software supply chains for your dependencies, tools, and platforms.

The 10% Factor

In a modern application, only about 10% of the code involved is written by the producer. The other 90% comes from the (primarily) open source that makes up the tools, frameworks, libraries, and platforms needed to realise the application. That means 90% of the code is shared with others. Software vulnerabilities are rarely in your code. There are in the code you consume—and the more popular this third-party code is, the more appealing it is to the bad guys.

Supported or Not?

Another significantly unsettling statistic is that of all the open-source out there, the OpenSSF statistics tell us that only 11% is maintained. That means, on average, your application has support for less than 20% of the code. 10% your team has written and 9.9% from open-source teams.

The software industry is certainly not paying attention to a growing tide of bad actors – whether motivated by greed, politics or military objectives.

Consequences and Risks

Using vulnerable software is prevalent. Governments, standards bodies, and industry groups are making significant efforts to address this situation. Unfortunately, the software industry is still in denial. Software creators everywhere are struggling to find a balance between reducing risks and maintaining a competitive advantage. That's understandable if it was just about money.

Although software supply chain attacks often start as technical breaches, they can quickly escalate into scenarios with severe human consequences.

We always use the term 'risk' concerning software security, imbuing the conversation with a financial flavour. We think in terms of insurance. We ask, "*How much risk should we take?*" as if we're placing a bet—which insurance sort of is. It's the wrong way of framing the question.

Russian roulette is a *kind-of* bet too, but not one you're likely to want to take. No kidding aside – we're past financial risk at this point. The ease of exploitation by bad guys and military cyber-groups means that we have to start asking ourselves "*how many people will die if we don't fix this vulnerability?*"

There have been indications before that people have been harmed by cybercrime, but as an indirect consequence. Ransomware attacks on hospital systems are obviously likely to cause delays to patients seeking treatment, and, no doubt, there have been people harmed or who may have died due to this.

Now we're seeing attacks that are deliberately aimed at hurting people.

The Human Element

Since Stuxnet in 2010, we've been aware that there are types of attacks aimed at hardware, where software is the delivery vehicle. These attacks are escalating. Even ransomware is evolving and direct harm to humans can be part of the equation. We're seeing direct examples where ransomware is effectively blackmail – *pay up or people will get hurt*. The implied threat we've seen with attacks on medical centres has become implicit.

The ransomware attack on Change Healthcare in 2024, one of the largest US healthcare technology companies, disrupted medical billing and prescription services nationwide. Several reports suggest delays in medication access, postponed surgeries, and financial strain on hospitals, directly impacting patient health.

A 2023 ransomware attack on a University of Alabama at Birmingham Hospital led to system outages affecting patient records and delaying treatments. A pending lawsuit alleges that a baby died due to these disruptions.

Another 2023 cyberattack on a third-party NHS IT provider caused significant disruptions in ambulance dispatch systems, leading to delays in emergency response times.

More Examples

A few more examples to illustrate this alarming reality clearly:

- **Colonial Pipeline Attack (2021):** Hackers disrupted fuel supplies across the United States, causing panic buying, shortages, and widespread economic disruption.
- **Saudi Aramco Petrochemical Plant (2017):** Cybercriminals attempted to trigger an explosion, directly threatening human lives and environmental safety.
- **Water Utility Hacks in Florida (2021) and Pennsylvania (2023):** Attackers targeted utilities supplying essential services, narrowly avoiding large-scale health crises.
- **Tesla Remote Unlock and Start Exploit (2023)** Researchers demonstrated a vulnerability in Tesla's Bluetooth Low Energy (BLE) authentication that allowed remote unlocking and starting of vehicles.
- **Medtronic Insulin Pump Vulnerability (2023):** vulnerabilities in Medtronic's insulin pumps, which could allow an attacker to modify insulin delivery.
- **Philips Medical Imaging Equipment Hackability (2024):** Vulnerabilities in Philips medical imaging systems were flagged as potentially allowing attackers to manipulate diagnostic results or disrupt scans

Each of these incidents underscores the evolution of cyberattacks, from financial nuisances in the beginning to existential threats today. Organizations must rethink how they define and approach "risk". It's time to shift the focus to the potential human consequences rather than purely financial implications.

Thinking Differently About Vulnerabilities

Next time a security scan flags a vulnerability, consider the impact more clearly. Consider the consequences for the consumer and for your software creation process. Look beyond the public impact score and examine the vulnerability within your unique circumstances. Don't disregard vulnerabilities in the build tools, compilers, etc. They are part of the software supply chain, too, and if compromised, can be used to infect your software. If the component implicated is challenging to upgrade, consider if it's still needed. Often components with vulnerabilities are likely to gain other ones later on. Poor software engineering practices can lead to repeated failure to detect problems, so it can be in your best interests to move to an alternative, saving time and effort and reducing exposure.

Moving Forward

Changing the status quo is challenging. Many governments worldwide are involved and are working with various industry groups to define counter measures. Unfortunately for the software industry, this is more stick than carrot.

Changes are coming that will bring rigour and regulations, not to mention financial and personal consequences, to those involved in creating software. New laws are being enacted, and existing ones are being extended or amended. For instance, the EU is expanding its product liability directives to cover harm caused by software

For executives, this new reality demands urgent action. Organizations must assess internal impacts and how disruptions affect users, suppliers, and the broader public. What happens when essential services become unavailable due to cyberattacks? Could your company's software inadvertently become a weapon in attackers' hands?

Businesses have a responsibility to protect the public proactively. Transparency about potential vulnerabilities and clear communication of preventive measures are essential. Incorporating robust practices such as continuous software monitoring, stringent security assessments, and transparent vulnerability disclosures can mitigate human consequences.

Developers, too, have a crucial role. Choosing secure, up-to-date software components and rigorously vetting third-party dependencies reduces vulnerabilities. Improving build and delivery processes in alignment with evolving legislation enhances security and safeguards human lives.

As cyber threats become increasingly aggressive and directly harmful, protecting human lives must drive the response to software supply chain attacks. Both software producers and consumers hold shared responsibility. Proactive steps today could prevent catastrophic human outcomes tomorrow.

› **Back to the Table of Contents** ‹



Mastering API Security in Java: OWASP Best Practices

#API

#OWASP

Author: Muaath Bin Ali

Muaath Bin Ali is a Principal Software Engineer with over 15 years of experience. As the leader of RiyadhJUG, he regularly hosts speakers and Java champions worldwide. Muaath shares his insights through blogs on MezoCode.com and fooja.io, assisting Java developers in building real-world software architectures and excelling in their careers.



LinkedIn

Building secure applications is a top priority for any developer, and when it comes to APIs, security becomes even more crucial. APIs expose sensitive data and functionality, making them attractive targets for attackers. The [OWASP API Security Top 10](#) provides a list of the most critical security risks to be aware of.

You will be mastering the API Security in Java by mitigating these risks, **offering practical Java code examples to illustrate both bad practices and best practices for each vulnerability.**

Why API Security Matters: [The Real Cost of Breaches](#)

In today's interconnected world, APIs are the backbone of countless applications and services. But this connectivity comes at a price: a single vulnerability can have catastrophic consequences. API breaches can lead to:

Data Breaches: Loss of sensitive customer data, including personal information, financial details, and proprietary business information.

Financial Loss: Direct costs from stolen funds, recovery efforts, regulatory fines, and lawsuits.

Reputational Damage: Loss of customer trust, negative media coverage, and long-term brand impact.

Operational Disruption: Downtime, service interruptions, and loss of productivity.

The financial impact of data breaches is staggering. According to Hacker's news "[Cost of a Data Breach Report 2023](#)," the average cost of a data breach reached an all-time high of \$4.45 million in 2023. Some notable examples of security breaches and their costs include:

Equifax (2017): Exposed the personal data of 147 million people, costing the company over \$1.4 billion.

Uber (2016): Exposed the data of 57 million riders and drivers, leading to a \$148 million settlement.

These examples highlight the importance of prioritizing API security. By understanding and mitigating the OWASP Top 10 vulnerabilities, you can significantly reduce the risk of a breach and protect your organization and users from significant financial and reputational damage.

Broken Object Level Authorization

Description: Without proper object-level authorization, attackers could access other customers' orders or personal information. This happens when APIs don't enforce access control based on the object (e.g., order, customer) being accessed.

Tips:

Implement robust authorization checks for every API endpoint that accesses data. Use a consistent and centralized authorization mechanism. Consider using role-based access control (RBAC) to manage permissions.

Avoid Practice:

```
1 @GetMapping(/orders/{orderId})
2 public Order getOrder(@PathVariable Long orderId) {
3
4     // No authorization check! Anyone can access any order.
5     return orderRepository.findById(orderId).orElseThrow();
6 }
```

Good Practice:

```

1  Long currentCustomerId = userService.getCustomerIdFromPrincipal(principal);
2  // Retrieve the order
3  Order order = orderRepository.findById(orderId).orElseThrow();
4
5  // Check if the order belongs to the current customer
6  if (!order.getCustomerId().equals(currentCustomerId)) {
7      throw new ForbiddenException(Unauthorized access to order);
8  }
9
10 return order;
11 }

```

Broken Authentication

Description: Weak authentication could allow attackers to take over customer accounts, leading to fraudulent purchases or data theft. This occurs when APIs have flaws in their authentication mechanisms, such as weak passwords, missing multi-factor authentication, or vulnerable session management.

How to Prevent:

- Enforce strong password policies and use robust hashing algorithms (e.g., bcrypt, Argon2).
- Implement multi-factor authentication (MFA) to add an extra layer of security.
- Use secure session management techniques, such as short session timeouts and secure cookie attributes.

Avoid Practice:

```

1  // Storing passwords in plain text
2  @PostMapping(/customers)
3  public Customer createCustomer(@RequestBody Customer customer) {
4
5  // Store the password without hashing!
6  return customerRepository.save(customer);
7  }

```

Good Practice:

```

1  // Using bcrypt for password hashing
2  @PostMapping(/customers)
3  public Customer createCustomer(@RequestBody Customer customer) {
4
5  // Hash the password before storing
6  String hashedPassword = BCrypt.hashpw(customer.getPassword(), BCrypt.gensalt());
7  customer.setPassword(hashedPassword);
8
9  return customerRepository.save(customer);
10 }

```

Broken Object Property Level Authorization

Description: Attackers could modify sensitive order details, like the shipping address or order total, without authorization. This happens when APIs don't properly control which properties (fields) of an object a user can update.

Tips:

- Implement fine-grained authorization checks for each property of an object.
- Validate user input to ensure they are only attempting to modify allowed properties.
- Use a whitelist approach to specify which properties are modifiable by different user roles.

Avoid Practice:

```

1 // Allowing any customer to modify any field of an order
2 @PutMapping(/orders/{orderId})
3 public Order updateOrder(@PathVariable Long orderId, @RequestBody Order updatedOrder) {
4
5     Order existingOrder = orderRepository.findById(orderId).orElseThrow();
6     existingOrder.setShippingAddress(updatedOrder.getShippingAddress());
7     existingOrder.setTotalAmount(updatedOrder.getTotalAmount());
8
9     // ... update other fields ...
10    return orderRepository.save(existingOrder);
11 }

```

Good Practice:

```

1 // Allowing only specific fields to be updated based on user roles
2 @PutMapping(/orders/{orderId})
3 public Order updateOrder(@PathVariable Long orderId, @RequestBody Order updatedOrder,
4     Principal principal) {
5
6     Order existingOrder = orderRepository.findById(orderId).orElseThrow();
7     User currentUser = userService.getUserFromPrincipal(principal);
8
9     // Only allow customers to update their own shipping address
10    if (currentUser.getRole().equals(CUSTOMER) && existingOrder.getCustomerId().equals(
11        currentUser.getId())) {
12        existingOrder.setShippingAddress(updatedOrder.getShippingAddress());
13    }
14
15    // Only allow admins to update the total amount
16    if (currentUser.getRole().equals(ADMIN)) {
17        existingOrder.setTotalAmount(updatedOrder.getTotalAmount());
18    }
19
20    return orderRepository.save(existingOrder);
21 }

```

Unrestricted Resource Consumption

Description: An attacker could overload the system with requests for product information, impacting performance for legitimate customers. This occurs when APIs lack rate limiting, allowing excessive requests that consume resources like bandwidth, CPU, or memory.

How to Prevent:

- Implement rate limiting to restrict the number of requests a client can make within a specific time.
- Use throttling mechanisms to control the flow of requests and prevent overload.
- Consider using caching to reduce the load on the server for frequently accessed data.

Avoid Practice:

```

1 // No rate limiting on product searches
2 @GetMapping(/products)
3 public List searchProducts(@RequestParam(query) String query) {
4     return productRepository.findByQuery(query);
5 }

```

Good Practice:

```
1 // Using Spring Security's @RateLimiter annotation
2 @GetMapping(/products)
3 @RateLimiter(key = #request.remoteAddress;, time = 1m, count = 10) // 10 requests per
  minute per IP address
4 public List searchProducts(@RequestParam(query) String query) {
5     return productRepository.findByQuery(query);
6 }
```

Broken Function Level Authorization

Description: An attacker could gain access to administrative functions, like managing product inventory or viewing sales reports. This happens when APIs don't properly restrict access to sensitive functions based on user roles or permissions.

Tips:

- Use role-based access control (RBAC) to define permissions for different user roles.
- Enforce authorization checks before allowing access to sensitive functions.
- Use a centralized authorization mechanism to manage permissions consistently.

Avoid Practice:

```
1 // Allowing any authenticated user to access admin functionality
2 @GetMapping(/admin/reports)
3 public String getSalesReport() {
4     // Generate and return sales report - accessible to all authenticated users!
5     return reportService.generateSalesReport();
6 }
```

Good Practice:

```
1 // Using Spring Security's @PreAuthorize annotation
2 @GetMapping(/admin/reports)
3 @PreAuthorize(hasRole('ADMIN'))
4 public String getSalesReport() {
5     return reportService.generateSalesReport();
6 }
```

Unrestricted Access to Sensitive Business Flows

Description: Attackers could exploit a checkout process without proper controls to place fraudulent orders. This occurs when APIs expose sensitive business logic without adequate protection, allowing attackers to manipulate workflows or bypass security checks.

How to Prevent:

- Implement strict validation and input sanitization for all inputs related to sensitive business flows.
- Use appropriate security controls, such as CAPTCHAs or fraud detection mechanisms, to mitigate automated attacks.
- Carefully design and review business logic to prevent unauthorized access or manipulation.

Avoid Practice:

```
1 // No validation or fraud checks during checkout
2 @PostMapping(/checkout)
3 public Order checkout(@RequestBody Order order) {
4     // Process the order without any validation or fraud checks
5     return orderService.processOrder(order);
6 }
```


Good Practice:

```
1 // Implementing validation, fraud checks, and inventory management
2 @PostMapping(/checkout)
3 public Order checkout(@RequestBody Order order) {
4
5     // Validate order details
6     if (!orderValidator.isValid(order)) {
7         throw new BadRequestException(Invalid order details);
8     }
9
10    // Perform fraud checks
11    if (fraudDetectionService.isFraudulent(order)) {
12        throw new ForbiddenException(Potential fraudulent order detected);
13    }
14
15    // Check inventory and reserve items
16    if (!inventoryService.reserveItems(order.getItems())) {
17        throw new BadRequestException(Not enough items in stock);
18    }
19
20    return orderService.processOrder(order);
21 }
```

Server Side Request Forgery (SSRF)

Description: An attacker could exploit this to access internal systems or files on the server, potentially stealing customer data or disrupting services. SSRF vulnerabilities occur when an API fetches a URL provided by the user without proper validation, allowing attackers to potentially access restricted resources.

How to Prevent:

- Validate and sanitize all user-supplied URLs before using them in server-side requests.
- Avoid using raw user input in URL construction.
- Use a whitelist of allowed domains and protocols for external requests.

Avoid Practice:

```
1 // Fetching product images from a user-supplied URL
2 @GetMapping(/products/{productId}/image)
3 public byte[] getProductImage(@PathVariable Long productId, @RequestParam(imageUrl)
4     String imageUrl) throws IOException {
5
6     URL url = new URL(imageUrl);
7
8     // Fetch and return the image from the provided URL without validation
9     return IOUtils.toByteArray(url.openStream());
10 }
```

Good Practice:

```

1 // Validating the URL against a whitelist of allowed domains
2 @GetMapping(/products/{productId}/image)
3 public byte[] getProductImage(@PathVariable Long productId, @RequestParam(imageUrl)
4     String imageUrl) throws IOException {
5
6     if (!isValidImageUrl(imageUrl)) {
7         throw new BadRequestException(Invalid image URL);
8     }
9
10    URL url = new URL(imageUrl);
11    return IOUtils.toByteArray(url.openStream());
12 }
13
14 private boolean isValidImageUrl(String imageUrl) {
15     // Check if the URL is in a whitelist of allowed domains
16     for (String allowedDomain : allowedImageDomains) {
17         if (imageUrl.startsWith(allowedDomain)) {
18             return true;
19         }
20     }
21     return false;
22 }

```

Security Misconfiguration

Description: An insecure configuration could expose sensitive data, such as customer payment details or internal server information. This broad category covers various misconfigurations, including insecure default settings, unnecessary features enabled, and missing security updates.

Tips:

- Use secure default configurations and harden systems according to security best practices.
- Regularly review and update configurations to address potential vulnerabilities.
- Disable unnecessary features and services to minimize the attack surface.

Avoid Practice:

```

1 // Exposing sensitive server information in error responses
2 @ExceptionHandler(Exception.class)
3 public ResponseEntity handleException(Exception ex) {
4
5     // Return the full stack trace in the error response
6     ErrorResponse error = new ErrorResponse(ex.getMessage(), ex.getStackTrace().toString());
7
8     return ResponseEntity.badRequest().body(error);
9 }

```

Good Practice:

```

1 // Returning generic error messages to clients
2 @ExceptionHandler(Exception.class)
3 public ResponseEntity handleException(Exception ex) {
4
5     // Log the exception with full details for debugging
6     log.error("An error occurred: ", ex);
7
8     // Return a generic error message to the client
9     ErrorResponse error = new ErrorResponse("An internal server error occurred.");
10    return ResponseEntity.badRequest().body(error);
11 }

```

Improper Inventory Management

Description: Undocumented or outdated APIs could be vulnerable to attacks, especially if they handle sensitive customer data. This risk highlights the importance of maintaining an accurate and up-to-date inventory of all APIs, their versions, and their security posture.

Good Practice:

- Use API documentation tools like Swagger or OpenAPI to keep the documentation current.
- Regularly review and update the API inventory, deprecating old versions and ensuring proper security for all active endpoints.
- Implement versioning for APIs to manage changes and deprecations effectively.

Unsafe Consumption of APIs

Description: Failing to validate data from third-party APIs, like a payment gateway, could lead to financial fraud or data breaches. This risk emphasizes the need to treat data from external APIs with the same level of scrutiny as user input.

How to Prevent:

- Validate and sanitize data from third-party APIs before using it in your application.
- Implement input validation checks to ensure data conforms to expected formats and types.
- Use a whitelist approach to accept only known and trusted data values.

Good Practice:

```
1 // Validating payment data received from a payment gateway
2 @PostMapping(/payments)
3 public PaymentResponse processPayment(@RequestBody PaymentRequest request) {
4
5     // Validate the payment data received from the payment gateway
6     if (!paymentValidator.isValid(request)) {
7         throw new BadRequestException(Invalid payment data);
8     }
9
10    // ... process the validated payment ...
11 }
```

Conclusion:

By understanding the OWASP API Security Top 10 and applying these best practices, Java developers building e-commerce systems can create more secure and trustworthy platforms. Remember to prioritize security throughout the development lifecycle, use secure libraries, and stay informed about emerging threats to ensure the ongoing protection of your e-commerce applications.

Reference:

- <https://javapro.io/2025/06/04/best-practices-for-api-design-in-java/>

› **Back to the Table of Contents** ‹



Kotlin Multiplatform's Cross-Platform at Norway's Nearly 400-Year-Old National Postal Service

CrossPlatform

Kotlin

Multiplatform

Author: Anshika Koul

Anshika has 10 years experience developing software at Posten Bring - Norwegian National Postal Service, building logistics systems and APIs that enable millions of people in the Nordics to order and receive packages. She values studying users, and fostering cross-departmental collaboration to demystify unruly & complex business rules. A typical day at work involves climbing stairs to the 17th floor of Posten Bring's towering headquarters in the heart of Oslo, where she will transform whiteboard scribbles into deployed features. Off-duty you'll find her chasing sunsets and culinary adventures, sampling flavours from around the globe.



LinkedIn

When we think of Norway's National Postal Service, we think of legacy, tradition, and history – a service that has been delivering letters, packages, parcels, and sometimes even joy, for nearly 400 years.

But what happens when such an ancient institution faces the modern challenges of software development and cross-platform integration? How does a company like Posten Bring navigate the world of fragmented business logic across multiple platforms and systems?

But before I answer these questions, I want to ask **YOU** – Have you ever had a complex business rule-set problem? You know the kind where you have business rules (logic) spread across multiple platforms, systems, and applications? Some of them might even be alter egos of the same rule hiding in different apps. You never know which one might turn up where; there is no single source of truth, solving errors takes hours of debugging, and making changes requires (oftentimes) painful coordination across teams and applications (ugh).

We faced this exact challenge at Posten Bring As – Norwegian National Postal Service, founded way back in 1647 – a 378-year-old institution with software spanning more generations than most families! Our business logic was scattered across platforms, some of which were as ancient as the company itself (okay, maybe not that old, but close enough)!

In this article, I'm spilling the tea. We will discover how the team at a century-old Norwegian postal services giant tackled modern challenges with cutting-edge technology.

I'm going to walk you through our journey with Kotlin Multiplatform, including the hurdles we encountered, the lessons we learned, and why KMP is a game-changer for anyone grappling with similar challenges.

This article is for you if troublesome & unruly business logic keeps you up at night & you want that to end NOW. It is also for you if you like to hear a good story.

Problem Statement – Complex World of Logistics

Let's start with a bit of context — From the moment you place an online order, to the time it shows up at your door, there's a lot happening in the background. Behind every package being ordered and delivered, there are **millions of lines of code** and **hundreds of thousands of business rules** governing the flow of that package. But let's briefly peel the onion here and take a closer look at how Posten Bring dealt with this chaos. Because—spoiler alert—**logistics is inherently complex**. And it's not just a Posten Bring issue, it's an industry-wide challenge.

In the past, we worked with a **huge monolithic system**, where all the business rules lived in one big beast of a codebase. Over the past decade, that beast has been broken down into smaller, more manageable apps. However, even though the monolith is no longer the centre of everything, we faced a new problem: **duplicated business rules** everywhere, spread across multiple codebases. And we had no central source of truth for those rules.

The shift from a monolithic system to microservices had its downsides. While we escaped the monolith, we were left with multiple codebases, each containing its own version of business logic. And that's when the **real challenges** began.

Can anyone here relate? Has anyone else spent hours, or maybe days, sifting through **millions of lines of code** trying to figure out where a business rule was implemented and why it was different in every app?

Debugging in this environment wasn't exactly "fun".

It was like playing whack-a-mole – fix one issue, and another pops up!



Abb. 60:

And it wasn't just production issues; even a simple question from our product owner could send us on a **wild goose chase**.

Inconsistent rules, slow feature development, and painful deployments led to a fragmented codebase, creating a high cost in business logic verification. This resulted in frustrated developers, **stifled creativity**, and an **increased likelihood of bugs**, all while hindering overall progress and clarity on feature success.

What did we want? Simple: **One codebase to rule them all**. We wanted a **centralised source of truth** for our business rules, one that didn't feel like defusing a bomb every time we touched it.



Abb. 61:

With the problem clearly identified, the next question was: **How could we solve this fragmented codebase chaos?**

The solution turned out to be **Kotlin Multiplatform**.

What Is Kotlin Multiplatform?

Kotlin Multiplatform (KMP) is an open-source framework developed by JetBrains that allows you to share code across multiple platforms (iOS, Android, web, and more) while retaining the benefits of native programming.

By using KMP, you can write common logic once and seamlessly run it on different platforms, making cross-platform development more efficient and scalable.

How KMP Streamlined Our Business Logic Across Platforms

At first, it wasn't about KMP. We just wanted a centralised solution for our code. In 2019, we started pulling out these duplicated business rules and dumping them into a Kotlin library, which was then packaged as a **JVM artifact**, which we used as a dependency in our backend applications. The goal? A centralised place for our backend APIs to share code, and for us to **finally have a semblance of control**.

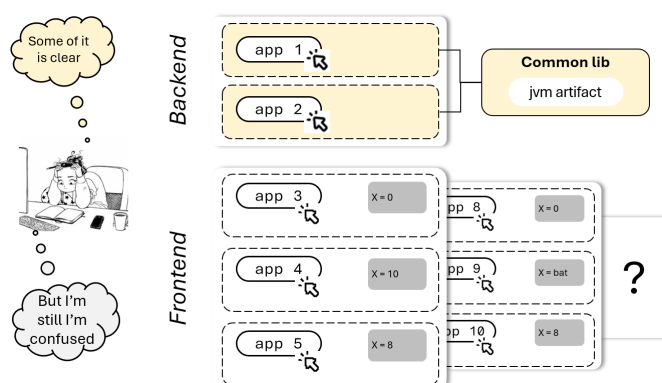


Abb. 62:

For a while, this worked wonderfully well. But in 2021, we realised something important: **We needed to extend this solution to our frontends** as well. We'd already built this great shared library and wanted to somehow use it for frontend validation too. We figured that if we could integrate the library we'd created two years ago into our frontend apps, we could maintain consistency and streamline development across the board.

That's when we turned to **Kotlin Multiplatform**.

KMP not only addressed our immediate needs but also offered several key advantages that made it the ideal choice for us.

- **Kotlin's interoperability with Java**
- **Strong community support**
- **Support for diverse build targets**
- **Proven success with industry leaders**
- **Rich ecosystem of libraries**
- **Comprehensive documentation**

Curious to see how we pulled it off with KMP? Watch the full talk [here](#), where we dive into the code behind our success.

So, Have We Finally Demystified the Chaos?

Well, if you ask me, I'd give a resounding **YES!** Our team is walking around with a spring in their step, joy in their heart, and maybe even a bit more colour in their life.

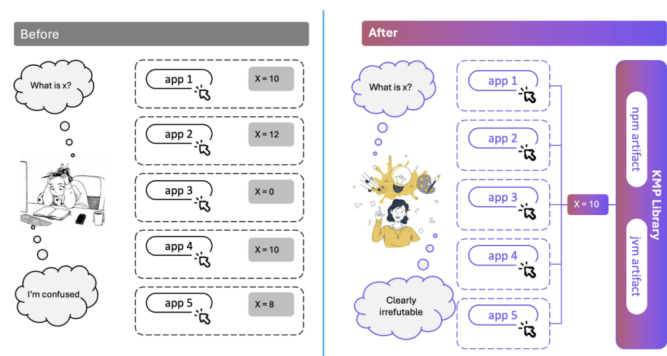


Abb. 63:

Gone are the days of wild goose chases and endless debugging. We've cleared up a huge chunk of the mess thanks to Kotlin Multiplatform, but let's not kid ourselves—it's not all sunshine, rainbows, and unicorns.

While KMP has done wonders for organising our business logic, there are still a few *X*'s lurking around like uninvited guests at a party. It's like trying to clean out a garage—you manage to get rid of most of the clutter, but there's always that one mysterious box of forgotten stuff. We'll get to those lingering issues, don't worry!

So, yes, we're making progress.

Here's to fewer headaches, fewer wild goose chases, and a lot more high-fives!

Wins, Woes, and Wisdom

As with any transformation, there have been some clear wins and a few challenges along the way.

Positives: KMP Made Our Work Life Better

- **Centralized Services & Business Rules**
- **Write-Once, Use-Everywhere**
- **Lower Maintenance & Testing Costs**
- **Faster Feature Releases & Optimised Reliability**
- **Developer Bliss: Mondays are fun again.**
- **Centralised Documentation:** Automatic updates for code-driven docs.
- **Product Owner Magic:** Product owners reading code and raising PRs with ease.

Challenges: We're Not Perfect Yet

- **Experimental Features & Libraries:** Some parts of KMP are still in beta.
- **Business Rule Categorization:** It's a work in progress.
- **JS Package Size:** Generated JS can be bulky, but Webpack 5 helps.
- **Developer Adoption & Skill Siloing:** Learning curves and skill gaps are still being addressed.
- **Tech Transformation Trials:** Convincing non-technical stakeholders required extra effort.

The Road Ahead: Cooking Up Your Own KMP Masterpiece

Now that we've unraveled the spaghetti, I hope you're tempted to cook up your own KMP masterpiece!

Start small—no need to dive headfirst into the deep end. Begin with a simple use case and build from there. Iterate and improve through repeated cycles, making adjustments as you go. And remember, KMP allows you to share code or write natively based on your project's specific needs, so feel free to tailor your approach.

At the end of the day, KMP is about simplifying the complex, improving efficiency, and, of course, having fun along the way. So, grab your apron, get coding, and start cooking!

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

JCON2026
www.jcon.one

JAVAPRO



📍 CINEDOM COLOGNE

JCON EUROPE 2026

International Java Community Conference



The Rise of JVM Languages: Kotlin, Scala, Groovy, and More

Groovy # JVM # Kotlin # Scala

Author: Mihaela Gheorghe-Roman

Mihaela is a Software Architect at Systematic, Romania, and an Oracle ACE Associate for Java. 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. 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

As Java marks its 30th anniversary in 2025, it remains a foundational force in software development. However, one of its most enduring contributions is not just the language itself but the Java Virtual Machine (JVM). The JVM has paved the way for numerous modern programming languages by providing a stable, efficient, and interoperable runtime environment.

Over the years, developers have embraced JVM-based languages. These languages offer alternative syntax styles, functional programming paradigms, and dynamic scripting capabilities. At the same time, they maintain seamless compatibility with Java. This article explores the growth of JVM languages. It highlights Kotlin, Scala, and Groovy, along with other influential options such as Clojure, JRuby, and Jython. These languages have extended the reach of the JVM, empowering developers with increased flexibility and productivity.

Java and the JVM Languages: A Brief History

James Gosling and his team at Sun Microsystems developed Java in the early 1990s and officially released it in 1995. Its “write once, run anywhere” (WORA) principle revolutionized software development by enabling platform-independent applications through the JVM.

The JVM executes Java bytecode—an intermediate representation of Java source code—allowing programs to run on any system with a compatible JVM implementation. As developers recognized its potential beyond Java, new languages emerged, leveraging the JVM’s powerful runtime environment to introduce enhanced programming paradigms and features.

Over time, the JVM has evolved to support multiple enhancements such as Just-In-Time (JIT) compilation, improved garbage collection algorithms, and ahead-of-time compilation for performance optimizations. These improvements have solidified its position as a critical runtime environment for a variety of software applications, from mobile apps to large-scale enterprise systems.

Kotlin: A Pragmatic JVM Language Alternative to Java

Kotlin, created by JetBrains and officially launched in 2016, quickly gained traction due to its concise syntax, null safety, and modern language features. Designed for full interoperability with Java, Kotlin eliminates boilerplate code, making it a preferred choice for Android development and enterprise applications.

Google’s endorsement of Kotlin as an official Android language in 2017 significantly accelerated its adoption. Moreover, features like coroutines for asynchronous programming, extension functions, and improved type inference have further solidified Kotlin as a robust Java alternative.

Beyond Android, Kotlin is increasingly used in backend development with frameworks like Ktor and Spring Boot. Companies such as Pinterest, Trello, and Coursera have successfully integrated Kotlin into their technology stacks, highlighting its scalability and developer-friendly syntax.

Scala: Bridging Object-Oriented and Functional Programming in JVM Languages

Introduced in 2004 by Martin Odersky, Scala seamlessly integrates object-oriented and functional programming principles. Its expressive syntax and strong type system make it a powerful choice for building scalable applications.

One of Scala’s most notable applications is in Apache Spark, a leading framework for big data processing. By combining functional and imperative programming styles, Scala provides significant advantages for parallel and distributed computing, often surpassing traditional Java-based approaches in efficiency.

Scala is widely adopted by organizations working in data engineering, finance, and machine learning. Its compatibility with Java allows companies to gradually transition their existing codebases to a more functional paradigm while still leveraging the vast Java ecosystem.

Groovy: A Dynamic JVM Language Enhancing Java

First released in 2003, Groovy is a dynamically typed language designed to simplify Java development. With its support for scripting, closures, and metaprogramming, Groovy offers a flexible and developer-friendly syntax that closely resembles Java, easing adoption for Java programmers.

Groovy powers Gradle, a widely used build automation tool in Java and Android development. Additionally, it plays a crucial role in testing frameworks like Spock, which enhances Java application testing with expressive specifications.

Developers frequently use Groovy for scripting and rapid prototyping beyond build automation, which makes it a popular choice in DevOps and continuous integration workflows. Organizations often utilize Groovy for writing domain-specific languages (DSLs) tailored to specific business needs.

Clojure: A Functional Lisp for the JVM

Developed by Rich Hickey and officially released in 2007, Clojure is a modern Lisp dialect that runs on the JVM. It emphasizes immutability, functional programming, and concurrency, making it ideal for multi-threaded applications.

Clojure encourages a functional approach with pure functions and immutable data structures, unlike Java, which is primarily object-oriented. Its adoption has grown in fields like artificial intelligence, data processing, and financial computing due to its robustness in handling complex computations.

Clojure is particularly favored in startups and research-oriented projects that require high levels of concurrency and real-time data processing. Many developers appreciate its REPL (Read-Eval-Print Loop) environment, which allows for interactive programming and quick experimentation.

JRuby & Jython: Bringing Ruby and Python to the JVM

JRuby enables seamless integration with Java libraries and frameworks, making it an attractive choice for Ruby developers needing Java interoperability. Similarly, Jython provides Python developers access to Java's scalability and enterprise features, making it valuable in various enterprise applications.

Web developers widely use JRuby, especially in applications that require both Ruby's expressiveness and Java's robustness. Scientific computing and financial services have adopted Jython, where developers benefit from Python's simplicity alongside the power of Java-based tools.

Other JVM Languages

Several other JVM languages exist beyond the mainstream ones, catering to niche requirements:

- **Frege:** A functional programming language inspired by Haskell, introduced in 2011, designed to leverage JVM capabilities.
- **JPHP:** A PHP implementation on the JVM, introduced in 2013, allowing PHP applications to run in Java environments.
- **Xtend:** A Java-compatible language with advanced lambda expressions and template support, first released in 2012.

These languages highlight the JVM's adaptability, ensuring developers have diverse options while benefiting from Java's mature ecosystem.

Why the JVM Remains Relevant

Even after three decades, the JVM continues to be a dominant platform in software development. Its reliability, extensive library support, and strong developer community make it a compelling choice for modern programming languages. Performance optimizations, such as Just-In-Time (JIT) compilation and advanced garbage collection, further enhance its efficiency.

Additionally, the JVM remains central to cloud computing, microservices architectures, and big data processing. Its efficiency in handling complex workloads ensures the relevance of JVM-based languages in the evolving tech landscape.

Major cloud providers, including AWS, Google Cloud, and Microsoft Azure, offer strong JVM support, reinforcing its role in cloud-native development. Additionally, with frameworks like Spring Boot and Quarkus optimizing Java-based microservices, the JVM continues to evolve to meet modern software engineering demands.

Conclusion

The JVM has significantly evolved, extending beyond Java to support a diverse range of languages. Kotlin, Scala, Groovy, Clojure, JRuby, and Jython have enriched the ecosystem. They offer developers tailored solutions for various paradigms and application needs.

As Java celebrates its 30th anniversary, its influence extends far beyond its own syntax. The JVM is a cornerstone of modern software development. It fosters innovation and ensures a strong foundation for future advancements.

› **Back to the Table of Contents** ‹



A 10X Approach to Quality Assurance

CodeQuality

Testing

Author: Steve Poole

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



LinkedIn

It's often true that what you measure will define you. Measurements become critical indicators, which eventually become goals. Measure the speed of something, and soon, almost inevitably, the goal will be to increase it.

In time, those goals and measurements will become the driving force behind behavior. The original objectives will be lost to history, and the new focus will be considered the norm.

This transformation occurs frequently and is often the downfall of organisations. They forget what they are in business to achieve, get lost in the process, and have these fabricated objectives.

This happens in the process of creating software, too. Let's look at one example of how thinking differently about what you do and how you measure it can make a profound difference.

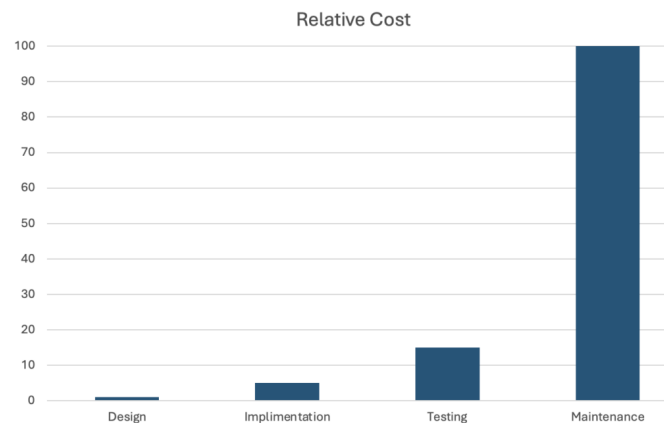


Abb. 64: Relative Costs

Bugs Cost

I expect most of you have seen charts like this one.

The 'cost' of finding a bug is described at each process stage. For example, if it costs \$1 to find the bug at design time, it takes \$ 100 to deal with it in production.

Don't worry about the stages' names; they will have different names depending on the development process, but the theory holds. This chart, or rather charts like this, are typical across the internet. The relative cost predictions change, but the sentiment is the same: delaying finding a bug costs money.

The temptation is to use this as ammunition to push testing to the left and make developers run more tests. It can also become a management excuse to do less testing. As I said at the beginning, what you measure defines you. If you only measure when a bug is found, you'll invent reasons to drive everything left.

When's the Best Time to Find Bugs?

Obviously, as soon as possible. Taken to the extreme, that would be when we press save on a design doc or source file. It would be great(!) if all possible tests could be exercised instantly and results fed back immediately.

Assuming you agree with that, we can also agree that since that isn't currently practical, our efforts in testing are, in effect, a pragmatic exercise of risk vs reward. It becomes a discussion about what not to test yet. What can we delay? It's not about not finding bugs, it's about doing it as efficiently and effectively as possible. It's about understanding that the chart above isn't telling you to move all testing left, it's telling you that a) you need to find as many of the hard-to-diagnose-remotely bugs you can before you ship and that your testing should include your ability to diagnose bugs remotely.

How Should We Really Test?

Any software production process has testing as one or more steps, from unit tests that developers run to full-grown integration testing carried out at the end of a release. There are probably other stages of testing; each step in the process, from conception to packaging to deployment, probably has some quality assurance effort.

Organising our testing strategy by stages makes sense, but it can quickly become sub-optimal if we take a compartmental rather than a holistic view.

Our risk/reward model says, “Run integration tests at the integration stage” and “Let developers run unit tests.” This model complies with our developer productivity pressures, as we strive to lessen the burden on the developer by eliminating “unnecessary chores.”

We’re saying that developer time is expensive, and we’re willing to risk bugs being found later because we perceive improved delivery.

We’ve implicitly labelled tests by who runs them, not by any material aspect of the test. If it’s called a unit test, it’s created/run by developers. If it’s called an integration test, it’s created/run by QA teams, even if the unit test is complex or the integration test is simple.

Changing the Status Quo

Compartmentalizing teams and tests is sub-optimal. There is a different approach that will work at any scale – whether a one-person project or beyond, but it takes time and evidence to make happen. So getting started requires some prep work and thinking through the objections you might encounter.

Three Challenges to Overcome

1: As discussed earlier, giving the developer extra work is considered non-optimal. Any plan to do this has to be well-considered and thoughtful. As we’ll see, it’s entirely possible, but we’ll need a few things to succeed.

2: There’s the question of testing ‘too’ much. A position many will take when this sort of discussion is aired: For example, whether a tree makes a sound in the forest when it falls or finding bugs that end-users are unlikely to discover is a worthwhile investment. Is all technical debt terrible? Running an extra 1000 hours of tests every build doesn’t sound like a good investment if it only produces one bug report.

3: The cost of shift left. It’s easy to say run the test earlier, but the tests that run later are usually part of some complex integration test harness or such like. The cost of developer time to learn even how to run these tests is often excessive. Hence, you frequently see conversations around producing a simplified example test case for developers. A test that quite usually sits in a special test bucket of developers’ favourite ‘known to find bugs’ test suite. This is often a bad idea in the long run because this ‘special’ test bucket becomes sacrosanct. A must-run-regardless test suite even though, over time, the value of the individual test lessens.

We Have Almost All the Pieces

For this cultural transformation to work, we need data because the most significant challenge will be changing mindsets. Some you may already have, others you’ll need to start collecting. None of this is rocket science.

Testing Is Multi-Dimensional

Our one-dimensional approach to organising tests and testing gets in the way. Thinking about when to run tests in simple chronological order: unit-test time, integration time, deployment time, etc, is inflexible, and over time leads to a loss of quality and increased costs.

Loss of quality because when quality problems occur, we tend to create new tests in the later, more complex testing stages rather than increasing the developers’ burden. That increases costs by raising the testing bill or the inevitable extra maintenance required to fix bugs that slipped through.

The answer is to take a step back and think about that original idealised objective. We can’t run every test every time code is saved, but we can organise our testing with that as a guiding principle.

There are a few essential dimensions: how quickly a test runs, how much domain knowledge is needed, and how often a test finds a bug vs. how ‘flaky’ it is. There are other connections and dimensions you might also consider, such as that illusive ‘how hard to debug’ element, but these are the primary ones.

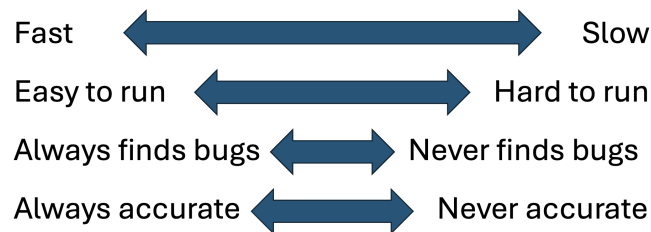


Abb. 65: Dimensions

With those dimensions, we can organise our tests into a few groups.

Tortoise and the Hare vs the Feather and the Boulder

Axis One, is about how fast a test takes to complete. It's that simple. From pressing the button to getting a programmatic answer (ideally excluding any framework tax)

Axis Two concerns the combination of domain knowledge required to run the test and any environmental considerations. One end of the axis is running the test in an IDE at a click without installing a database, app server, API gateway, etc.; the other end is needing to understand how to install Kubernetes, etc., and then knowing how to trigger an Istio failure.

Axis Three, how good the test case is at finding bugs is essential. If you're not recording this sort of metric, then optimising test runs will be challenging. We've agreed that we're in a risk/reward situation, so one risk you can take is to reduce the number of times you run tests that hardly fail while continuously running the ones that behave as your canaries.

Axis Four, is about noise. How often do you get false positives or false negatives from a test? Any test that isn't rock solid in execution needs scrutiny because otherwise, it's costing you time and effort that you don't need to spend.

Bringing It Together

Axis Three and Four: Usefulness and Accuracy can be considered filters over test execution. If a test has never found a bug but is quick to run, you might keep it, while one that takes an hour to run on AWS could easily be a candidate to remove – or at the very least rewrite. You can apply this filtering to your existing test organisations, and it will undoubtedly help. For many, this is already an everyday activity.

What moves us into more 10x territory is to consider the other axes.

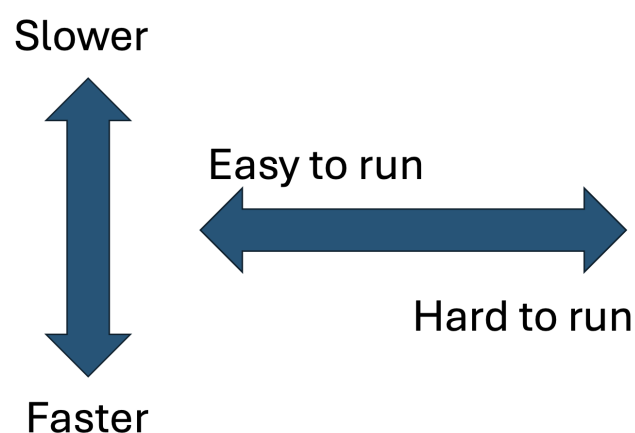


Abb. 66:

These two dimensions allow us to think about how we organise testing differently. Forget the typical linear process and organisation groupings and look at tests through this lens.

Risk / Reward Rebalanced

Tests that are fast and easy to run are perfect candidates to give to developers. With no or low domain knowledge required to execute, they should be easily incorporated into unit tests for running in the IDE or on the build server.

Tests at the other end, which are slow to run and require significant setup and domain knowledge, belong more to the QA teams.

Tests that have other combinations can be part of other organisations or may suggest using different strategies. Slow-running but easy-to-run tests might still be given to developers, but they get to run on the build server, which is not part of a developer's personal unit test activity. Slow tests might be run less often.

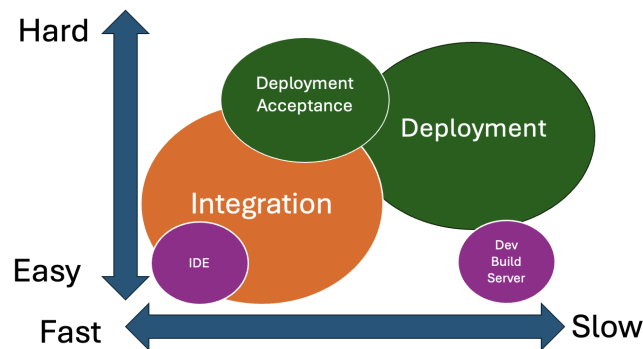


Abb. 67:

Changing Behavior

Once you have this thought process about measuring and organising tests, it becomes easier to encourage more flexible behaviour.

It helps to focus on consistently creating test cases that are as simple, fast, and easy to run as *feasible*. This also helps to create an understanding that tests don't 'belong' to any particular group. Tests need to be portable, so even if you must build complex test harnesses, try to keep the actual tests in a form that a developer can run without.

Developers will undoubtedly support broader unit tests during the build process if they are easy to run when things go wrong and don't take excessive time. Likewise, a cross-organisational focus on quality that uses these axes as guides will grease the wheels of the whole process.

More Thoughts

Once you've started down this road, you'll naturally gain a broader understanding of the objectives and value of testing.

You'll first realise that although the product code is what you ship, the test cases define the product. Your test suite is both a safety net and a jelly mould. The better your testing, the better it describes the expected behaviour of the product, the faster you can develop and the more radical you can be when changing the code. Knowing that the QA process has a developer's back and that every effort goes into continually balancing the risk/reward equation is liberating.

I mentioned other things to measure. Soon, you'll want to connect test results with previous code changes. Start to plan test execution based on the changes in the source files. Then perhaps you'll look at lines of code changed or cyclic complexity. Some might even run different tests depending on the developer making the change or the day of the week!

Eventually, you'll realise that this data lets you build specific test execution plans. Is a significant new feature in the plan? Then use this data to select the appropriate test cases for a focused unit-test effort for the dev team. Have some new developers onboard? – Build a list of 10 developer tests associated with 'first timer' open issues and get them fixed.

Summary

The takeaways of moving to a data-driven and test portability approach are more than a quality improvement. This change brings teams together and makes the whole team more productive. Instead of adding a burden to the developer, it reduces the workload because of the focus on running the appropriate tests in the proper time. It also makes everyone understand the relationship between the test suite and the product.

Measuring the ‘process’ of software development brings a scientific mindset to the risk/reward equation we started with. Now, decisions about development costs and investment can be more accurately determined. It aids quality assurance improvements by helping to identify areas of weak testing or over-testing. Now, you can construct test plans with the best mix of coverage and execution time and optimise the return from each test suite at each process stage.

Change is always a challenge – but this switch is one of the best ways to modernise how you ‘do’ development. It requires a little faith initially, but the data-driven approach and acceptance of portable tests will pay dividends over time.

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

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

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