

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY #JAVAPRO

Autonomous Java

17 **OPENJ9 GC POLICIES:
HOW TO CHOOSE THE RIGHT COLLECTOR**

27 **ELASTICSEARCH JAVA SDK:
NO MAGIC, JUST SOLID DESIGN CHOICES**

40 **CONNECTING JAVA REINFORCEMENT
LEARNING TO PYTHON GYMNASIUM**

76 **AGENTIC AI PATTERNS
FOR ENTERPRISE SOFTWARE**

85 **SEMANTIC VERSIONING
DONE AUTOMATICALLY**





SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

#Editorial #JAVAPRO

From Sustainable Engineering to Autonomous Java

Software development is undergoing a quiet but significant transformation. For years, much of our attention was focused on individual technologies, frameworks, and the language itself. New APIs arrived, architectures evolved, cloud-native became mainstream, and more recently, Artificial Intelligence entered the software development landscape at unprecedented speed.

Yet as systems continue to grow in complexity, a different challenge is emerging. The question is no longer simply how to build a feature. The question is how to build software that remains understandable, maintainable, scalable, and adaptable long after the first version has been deployed and how we prepare our applications for a future where systems operate with increasing autonomy.

This issue explores that journey toward **Autonomous Java** from multiple perspectives. We begin with the absolute bedrock of any self-steering system: software quality and maintainability. From applying SOLID principles in test automation and Page Object Models to understanding the design decisions behind the Elasticsearch Java SDK, we revisit core engineering practices that continue to matter regardless of technological trends. True autonomy cannot exist without a rock-solid, predictable architecture.

At the same time, modern software delivery increasingly depends on automation - the vital precursor to true autonomy. Versioning, releases, testing, deployment pipelines, and operational processes must work reliably at scale without manual intervention. Our coverage of automated semantic versioning demonstrates how routine development tasks can become predictable, repeatable, and far less error-prone when properly integrated into the engineering workflow.

Artificial Intelligence remains a major force shaping software development, but the conversation is rapidly evolving. Rather than focusing solely on models and prompts, this issue examines the next frontier: how agentic AI patterns can be integrated into enterprise systems and how autonomous software components are reshaping future application architectures.

Another recurring theme throughout this edition is the growing importance of understanding information at scale. Whether we search through source code, business data, logs, or documents, traditional approaches increasingly struggle to keep pace with the size and complexity of modern systems. Our exploration of semantic search technologies demonstrates how richer models and deeper contextual understanding are enabling developers - and autonomous agents - to interact with large codebases and information ecosystems.

We also look beyond established technology domains. Blockchain remains one of the most misunderstood topics in enterprise software, often viewed as a specialized field disconnected from mainstream application development. In reality, many of its underlying concepts will feel surprisingly familiar to experienced Java developers, and understanding them becomes increasingly valuable as decentralized, autonomous technologies mature.

Finally, operational excellence remains a defining characteristic of successful software systems. Choosing the right garbage collection strategy can have a significant impact on performance, scalability, and resource efficiency. Our examination of OpenJ9 garbage collection policies highlights how thoughtful runtime decisions continue to play a crucial role in modern, highly automated Java deployments.

Taken together, the articles in this issue share a common message: Technology continues to evolve rapidly, but long-term success rarely comes from chasing individual trends. It comes from building systems that remain reliable, understandable, and adaptable as technologies, requirements, and organizations change. Modern Java is uniquely positioned to lead this shift. Its ecosystem combines decades of engineering maturity with continuous innovation, allowing teams to embrace the future of autonomous systems while maintaining the stability and discipline required by production environments.

The future of software development will undoubtedly include AI, autonomous agents, and entirely new platforms. But regardless of what comes next, sustainable software engineering remains the foundation upon which all meaningful innovation is built.

A very special thanks goes to all the authors of this issue. By generously sharing their deep expertise with the Java community, they not only help drive the ecosystem forward but also make a truly invaluable contribution to the future of our industry.

Enjoy this issue.

Author: Markus Kett

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



LinkedIn



JAVAPRO

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io

Contents

Core Java

Savi Grover

Learning Java Automation the SOLID Way: A Guide to Learn Page Object Model With Java SOLID Principles	8
Java SOLID Concepts	9
Essential Features for Java Automation:	9
SOLID-ification of Object-Oriented Programming in Test Automation:	10

Core Java

Babneet Singh & Aleksandar Micic & Dmitri Pivkine & Tobi Ajila

OpenJ9 GC Policies: How to Choose the Right Collector	17
How to Choose an OpenJ9 GC Policy	18
OpenJ9 GC Policy Comparison Table	19
gencon	19
balanced	20
optavgpause	21
optthruput	22
metronome	22
nogc	23
Concurrent Scavenge (gencon mode)	24
What to Look For in OpenJ9 Verbose GC Logs	24
A Rough HotSpot Mental Model	25
Workload-Based Policy Guidance	25
FAQ	25
Final Thoughts	26
References	26

API & Frameworks

Stefano Fago

Elasticsearch Java SDK: No Magic, Just Solid Design Choices	27
Specification-Driven Code Generation	28
The Builder Pattern with Safety Guarantees	28
Type-Safe DSL Through Lambda Composition	29
Modeling Variants With Tagged Unions	29
Namespace Organization and API Modularity	30
Pragmatic Nullability Conventions	30
Transport Abstraction and Separation of Concerns	30
Exception Design and Error Handling	31
Trade-Offs and Considerations	31
Conclusion	31

API & Frameworks

Satya Ranjan

Java Developers, You're Already Ready for Blockchain — You Just Don't Know It Yet	32
Blockchain: What Java Developers Need to Know	33
Why Cardano?	33
What Is BloxBean?	34
Building Transactions with Cardano Client Lib	35
Your Local Blockchain in Seconds: Yaci DevKit	36
Under the Hood: Yaci Mini-Protocols	36
Indexing Blockchain Data: Yaci Store	37
Quick Start: Your First ada Transfer	38
The Broader Java Ecosystem on Cardano	38
Java in Production on Cardano	38

API & Frameworks

Guilherme Alves Silveira

Connecting Java Reinforcement Learning to Python Gymnasium	40
What's Reinforcement Learning? A Simple Introduction	41
Gymnasium in the Java Reinforcement Learning Ecosystem	42
Evaluating Java Reinforcement Learning Integration Options	43
JavaCPP	43
Setting Up the Python Environment	44
Java Reinforcement Learning Python Gymnasium: Integration Roadmap	45
Embed Python Lifecycle	45
PythonRuntime: Java Reinforcement Learning Python Bridge	46
Gym Class: Building Gymnasium Environments in Java	50

API & Frameworks

Bryan Friedman

Beyond Text Search: A Smarter Way to Understand Your Code	59
Why Text Search Keeps Showing Up	60
The Model Underneath: Lossless Semantic Trees	60
Structural and Semantic Aren't the Same	61
A Spectrum of Search, Not a Single Tool	61
Trigrep: Trigram Search, Made Smarter	62
Recipes: the Deep End of the Spectrum	63
Bridging Broad and Precise: Scoping a Recipe Run	63
Code Search for Agents	64
A Practical Search Workflow	64
Search Isn't Just a Feature Anymore	64

API & Frameworks

Yasin Baytürk

The 3 A.M. Deadlock	66
Why This Article Exists	67
What a Deadlock Actually Is, Without the Hand-Waving	67
The First 15 Minutes: A Triage Playbook	68
Five Deadlock Patterns That Cover Most Real Incidents	68
Building a Regression Test You Actually Trust	72
A Word About Retries	73
The Architectural Takeaway: Discipline Beats Cleverness	74
Want to Dive Deeper?	74

AI & ML

Kevin Dubois & Laura Cowen

Agentic AI Patterns for Enterprise Software	76
It All Starts with a Single AI Service	77
The Path from a Single AI Service to Agentic Systems	77
Tools, Function Calling, and MCP	77
Agentic AI Systems and Workflows	78
Additional Agentic Features	82
LangChain4j and Quarkus	84
Conclusion	84

DevOps

Jago de Vreede

Semantic Versioning Done Automatically	85
Semantic Versioning	86
History	86
Problem	86

Solution	86
Rules	86
Dependencies	88
Incorporate in Your Project	88
Conclusion	89
References	89

Security

Sven Ruppert

Sharing SBOMs Securely Without Giving Too Much Away	90
1. SBOMs Create Transparency — but Not Without Risk	91
2. Why Complete SBOMs Contain Sensitive Information	91
3. Least Disclosure: Sharing as Controlled Disclosure	91
4. Different Recipients, Different Information Needs	91
5. Public Transparency versus Private Exchange	92
6. Redaction: Several Secure Views from One Bill of Materials	92
7. Access Control Beyond Simple Roles	92
8. Demonstrability: Auditability and Release Binding	93
9. CI/CD Integration and Conclusion	93

Security

Sven Ruppert

The Security Innovations in Vaadin 25.2	95
The Foundation: Why Vaadin's Server-Side Architecture Is Considered Inherently Secure	96
Protection Against Unsafe URL Schemes (New in 25.2)	98
Safe Rendering of Untrusted HTML with SafeList (New in 25.2)	99
Clickjacking Protection Through X-Frame-Options: SAMEORIGIN (New in 25.2)	100
Supply-Chain Hardening Against Compromised npm Packages (New in 25.2)	102
The Existing Shield at a Glance, Into Which the Innovations Fit	102
Secure Configuration and Operation	105
Migration Notes: Security-Driven Behavioural Changes When Upgrading to 25.2	107
Conclusion: Security as a Default and Its Significance for Compliance Requirements	107

Testing & Quality

Alexei Bratuhin

How to Write Your Own DbUnit	109
The First Step to Dealing With a Problem Is Admitting That You Have a Problem. — Jase Robertson	110
Is There a Better Way?	110
How Come I Have Never Heard of It Before?	111
Refresher: Test Pyramid vs. Test Trophy, Testcontainers	112
What About X?	112
Problem Formulation and Initial Design	112
Quick Introduction to etcd and Its Java Client jetcd	112
Implementation	114
Down the Rabbit Hole	119
Conclusion	122

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors



JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92681 Erbenndorf
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 Erbenndorf
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.



Learning Java Automation the SOLID Way: A Guide to Learn Page Object Model With Java SOLID Principles

CoreJava # JavaAutomation # SoftwareTesting # SOLID

Author: Savi Grover

Savi Grover is a Senior QA Automation Engineer with extensive expertise in software testing, automation frameworks and quality assurance strategies.

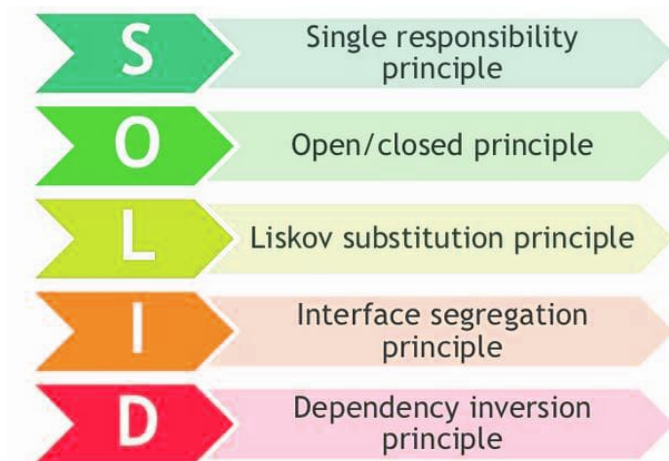


LinkedIn

Java is one of the most popular programming languages in the test automation field. A solid grasp of core Java concepts provides a foundation for building robust and maintainable automated tests. This article will explore how key Java programming concepts — including Object-Oriented Programming (OOP) principles (inheritance, polymorphism, abstraction, encapsulation) directly support tasks in software automation. We will see how Java SOLID principles are beginner's footprints for learning test automation through Page Object Model (POM).

Java SOLID Concepts

Here are the five SOLID principles: * **Single Responsibility Principle (SRP)**: A class should have only one reason to change, meaning it should have only one responsibility. This promotes high cohesion within a class. In short, One Class = One Reason! * **Open-Closed Principle (OCP)**: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means we can add new functionality without altering existing, tested code. * **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types without altering the correctness of the program. This ensures that inherited classes behave as expected when used in place of their parent classes. * **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use. This advocates for creating smaller, more specific interfaces rather than large, monolithic ones. * **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions. This promotes loose coupling and makes the system more flexible and testable.



Essential Features for Java Automation:

A strong command of core Java features is **essential for automation engineers**, as it enables them to design better test code and frameworks. Automated testing is, at its heart, software development; thus, the same principles that make production robust and reusable also apply to test scripts. Let's examine several fundamental Java concepts and see how they support software automation tasks:

- **Object-Oriented Programming (OOP)** – Encourages modular, reusable, and maintainable code through classes and objects. OOP principles (inheritance, polymorphism, abstraction, encapsulation) help design test frameworks (e.g. page object models, utility classes) that are scalable and easy to maintain.
- **Exception Handling** – Allows test scripts to handle errors gracefully. Automated tests must anticipate issues like missing elements or timeouts and either recover or log failures clearly without crashing the entire test run.
- **Collections Framework** – Provides dynamic data structures (like List, Set, Map) to store and manipulate groups of objects. In testing, collections are used for handling multiple elements (e.g. lists of web page elements) or datasets (e.g. lists of test inputs) efficiently.
- **File I/O (Input/Output)** – Enables reading from and writing to files. Automation often involves reading test data or configurations from external files (CSV, JSON, XML, etc.) and writing out logs or reports. Knowing Java file, I/O helps create data-driven tests and manage test artifacts.
- **Multithreading** – Supports concurrent execution, which is useful for **parallel testing**. Running multiple tests simultaneously (on multiple threads) can significantly reduce execution time and increase coverage in different environments.

- **Java Libraries and Frameworks** – Java’s rich ecosystem of libraries allows automation engineers to leverage pre-built functionality. For example, Selenium and Appium are libraries for browser and mobile automation respectively, and JUnit/TestNG are frameworks for structuring and running tests. Familiarity with using external libraries (via Maven dependencies or JARs) is crucial for setting up an automation project.

Each of these concepts plays a role in creating reliable and efficient test automation solutions. In the following sections, we’ll dive deeper into each concept, explaining the fundamentals and illustrating how they apply to real-world automation tasks.

SOLID-ification of Object-Oriented Programming in Test Automation:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of “objects” that combine data and methods. Java is a strongly object-oriented language, and its OOP features are central to writing well-structured automation code. Embracing OOP allows testers-turned-developers to design test scripts and frameworks that are easier to maintain, extend, and debug. The four pillars of OOP — **encapsulation, inheritance, polymorphism, and abstraction** — are particularly useful in automation framework design. Let’s discuss each principle that is analogous to SOLID guidelines.

A. Encapsulation – (Self-Contained Page Objects) and Utilities

Encapsulation is the mechanism of bundling data (attributes) and code (methods) together into a single unit (a class), while restricting direct access to some of the object’s components. In practice, this often means making class fields private and providing public getters/setters or methods to interact with them. Encapsulation helps protect the internal state of an object and exposes a clean interface.

In automation, encapsulation is used to create **Page Object Models (POM)** and utility classes that hide internal implementation details from the test cases. For example, a page object class might encapsulate web element locators and interaction methods for a login page. The test script calls high-level methods like login (“user”, “pass”) without needing to know how those methods locate and interact with UI elements – the locator details are hidden inside the page object class. This leads to cleaner tests and easier maintenance (changes in UI only require updates in the page class, not every test).

All classes in a well-designed test framework exhibit encapsulation by keeping their data members private and exposing only what is necessary.

In POM terms

- Each Page Object class should model one page/screen or a clear part of it.
- Test classes should contain test logic, not WebDriver plumbing.
- Utility classes handle cross-cutting stuff (waits, logging, configuration).

```

1  public class LoginTest {
2      WebDriver driver;
3      @Test
4      public void login() {
5          driver = new ChromeDriver();
6          driver.get("https://app.com/login");
7          driver.findElement(By.id("user")).sendKeys("username");
8          driver.findElement(By.id("pass")).sendKeys("secret");
9          driver.findElement(By.id("login")).click();
10         // + assertions, + waits, + screenshots, + reporting...
11     }
12 }
```

The above code –

- Starts the browser
- Navigates
- Locates elements
- Performs actions
- Asserts results
- Maybe logs, takes screenshots...

SRP-inspired POM refactor:

```

1  public class LoginPage {
2  private WebDriver driver;
3  @FindBy(id = "user") private WebElement username;
4  @FindBy(id = "pass") private WebElement password;
5  @FindBy(id = "login") private WebElement loginButton;
6  public LoginPage(WebDriver driver) {
7      this.driver = driver;
8      PageFactory.initElements(driver, this);
9  }
10 public void open() {
11     driver.get("https://app.com/login");
12 }
13 public void loginAs(String user, String pass) {
14     username.sendKeys(user);
15     password.sendKeys(pass);
16     loginButton.click();
17 }
18 }
19 // Test class: only describes behavior 'were testing
20 public class LoginTest {
21     private WebDriver driver;
22     private LoginPage loginPage;
23     @BeforeEach
24     void setUp() {
25         driver = new ChromeDriver();
26         loginPage = new LoginPage(driver);
27     }
28     @Test
29     void userCanLogin() {
30         loginPage.open();
31         loginPage.loginAs("username", "secret");
32         // assertions only
33     }
34 }

```

B. Second SOLID principle- O – Open/Closed Principle (OCP)**Inheritance – Reusing Common Test Code**

Inheritance allows one class to acquire the properties and methods of another. In Java, we use the extended keyword for a class to inherit from a base (parent) class. In test automation, inheritance is commonly used to create a **base test class or base page class** that contains shared setup, teardown, or utility logic, which is then extended by individual test classes or specific page classes. This avoids duplication and centralizes common functionality.

For example, you might have a BaseTest class that initializes the WebDriver, sets up timeouts, loads configuration, etc., in a @Before method, and quits the browser in an @After method. All your test classes then extend BaseTest to inherit this boilerplate setup/teardown, so each test doesn't need to write it anew. Similarly, a BasePage class might hold a reference to the WebDriver and common methods (like a generic click() or type() method), which specific page classes extend. This design makes use of inheritance to promote code reuse.

In POM terms

- Base POM framework should let us add new pages or behaviors without constantly editing old classes.
- Common logic goes in BasePage, reusable components, or helper layers.
- New flows are created by extending or composing, not rewriting.

How OCP helps a beginner:

Base page that rarely changes:

```

1  public abstract class BasePage {
2      protected WebDriver driver;
3      protected BasePage(WebDriver driver) {
4          this.driver = driver;
5          PageFactory.initElements(driver, this);
6      }
7      protected void click(WebElement element) {
8          element.click();
9      }
10     protected void type(WebElement element, String text) {
11         element.clear();
12         element.sendKeys(text);
13     }
14     protected String getTitle() {
15         return driver.getTitle();
16     }
17 }
18 LoginPage extends BasePage:
19 public class LoginPage extends BasePage {
20     @FindBy(id = "user") private WebElement username;
21     @FindBy(id = "pass") private WebElement password;
22     @FindBy(id = "login") private WebElement loginButton;
23     public LoginPage(WebDriver driver) {
24         super(driver);
25     }
26     public void loginAs(String user, String pass) {
27         type(username, user);
28         type(password, pass);
29         click(loginButton);
30     }
31 }

```

When a beginner wants to:

Add new pages → just create a new class extending BasePage.

Add a generic action (e.g., highlight before click) → change click() once in BasePage.

They extend behavior without rewriting all page classes. That's OCP in action.

C. Polymorphism – Flexibility in Test Code (WebDriver Example)

Polymorphism means “many forms” and allows treating objects of different classes through a uniform interface. In Java, polymorphism typically manifests as method overloading (same method name, different parameters) or method overriding (subclass provides its own implementation of a parent class method). For test automation, polymorphism is extremely useful when dealing with different types of pages or elements in a generic way, and when using interfaces.

A classic example of polymorphism in Selenium is how you instantiate browser drivers using the WebDriver interface. Code like

```
WebDriver driver = new FirefoxDriver();
```

or

```
WebDriver driver = new ChromeDriver();
```

is leveraging polymorphism. Here, WebDriver is an interface and FirefoxDriver/ChromeDriver are concrete classes implementing it. By programming to the interface (WebDriver), your code can work with any browser driver. This means your test logic (navigating pages, clicking elements) doesn't change if you swap out the browser – you could set driver to a SafariDriver or EdgeDriver and the test steps remain the same, thanks to polymorphism. This is an example of *upcasting* (a subclass instance treated as a superclass/interface type), which provides flexibility in automation scripts to support multiple browsers easily.

Polymorphism via method overriding is also prevalent. For instance, Selenium's browser drivers override the WebDriver interface methods like get() or findElement() with their browser-specific implementations under the hood. As a tester, you

call `driver.get(url)` regardless of the browser, and the correct browser-specific action occurs – a polymorphic behavior. Another example in test code might be having a generic Page interface or superclass and multiple page classes (LoginPage, HomePage, etc.) that override certain behaviors, allowing methods to accept or return a Page type and still operate on any concrete page.

Additionally, **overloading methods** can be used in test utility classes. For example, you might overload an `assertVisible()` method to accept different parameters (by locator, by WebElement, by timeout, etc.), giving flexibility in how you assert UI elements in tests.

Example (Polymorphism with WebDriver interface):

```
1 // Using WebDriver interface for flexibility (polymorphism)
2 WebDriver driver = new ChromeDriver();// could be FirefoxDriver, etc.
3 driver.get("https://www.example.com");
4 WebElement link = driver.findElement(By.id("myLink"));
5 link.click(); driver.quit();
```

In this code, driver is declared as the interface type WebDriver but at runtime can hold any browser driver. The calls to `get`, `findElement`, etc., work no matter which browser is used, illustrating polymorphism. If tomorrow we change to `WebDriver driver = new FirefoxDriver();`, the rest of the test code doesn't need modification – a powerful benefit for cross-browser testing.

D. Abstraction – Hiding Complexity in Framework Design

Abstraction involves hiding the complex implementation details and exposing only the necessary functionality. In Java, abstraction is achieved via abstract classes and interfaces. The goal is to reduce complexity for the user of a class by providing a simplified interface.

In test automation, abstraction is evident in framework design patterns. A prime example is again the **Page Object Model**: tests interact with high-level methods (like `loginPage.login()`) without needing to know *how* those actions are performed underneath. The internal mechanics (locators finding elements, WebDriver calls) are abstracted away from the test. This means the test writer focuses on *what* the test should do, not *how* it's done at the code level.

Another area of abstraction is having interfaces or abstract classes define a contract for certain behaviors. For instance, we can define an interface `ITestDataSource` with a method `getData(String key)`, and have multiple implementations (e.g., `CsvDataSource`, `DatabaseDataSource`) that provide test data from different sources. The test code can remain abstracted from the data source details by programming to the `ITestDataSource` interface.

In Java, using an interface such as Selenium's WebDriver is itself a form of abstraction – the WebDriver interface declares methods like `findElement()` but the actual implementation in `ChromeDriver` or `FirefoxDriver` is hidden. We saw earlier that this is polymorphism; it's also abstraction because the test code doesn't need to understand the inner workings of the `ChromeDriver`, only the abstracted WebDriver API.

Another common abstract concept in automation frameworks is an **abstract base page** or **abstract test class**. You may declare an abstract class `BasePage` with abstract methods like `isAt()` (which derived pages implement to verify the page is loaded). Tests then can use these abstract methods without caring about each page's specifics.

Example (Abstraction with an Interface for Pages):

```
1 public interface Page {
2     String getPageTitle();
3     boolean isAt(); // verify we are on the correct page
4 }
5 public class HomePage implements Page {
6     // ... locators and methods internal to HomePage
7     @Override public String getPageTitle() { return driver.getTitle(); }
8     @Override public boolean isAt() { return driver.getCurrentUrl().contains("/home"); }
9 }
```

E. Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base type

In POM terms

- Tests should work even if the underlying page implementation changes (e.g., web vs. mobile web, A/B variants).
- You can swap one Page implementation for another without breaking tests that depend on the *interface/abstraction*.

In this example, Page is an interface defining what any page should do.

HomePage implements it with details. A test can use

Page page = new HomePage(); and call page.isAt() without knowing the internal logic – the interface abstracts the concept of “being on a page”.

F. Interface Segregation Principle (ISP)

Many small, specific interfaces > one huge interface

In POM terms

- Instead of giant “god” interfaces or classes (e.g. IPage with 20 methods), create **small, meaningful interfaces** that express capabilities.
- This keeps Page Objects **focused** and reduces confusion for beginners.

```

1 public interface Searchable { void searchFor(String text); }
2 public interface Pageable {
3     void goToNextPage();
4     void goToPreviousPage();
5 }
6 public interface Filterable {
7     void applyFilter(String filterName, String value);
8 }
9 //A page that supports search and filters:
10 public class ProductsPage extends BasePage implements Searchable, Filterable {
11     @Override
12     public void searchFor(String text) {
13         // type in search box, click search
14     }
15     @Override
16     public void applyFilter(String filterName, String value) {
17         // select filter, click apply
18     }
19 }

```

For a beginner, ISP helps them:

- Think: “**What can this page do?**” instead of “What methods should I randomly add?”
- Recognize that **not every page must have every method**—only what’s relevant.

G. Dependency Inversion Principle (DIP)

High-level modules depend on abstractions, not on concrete details

In POM terms

- Tests (high-level) should depend on **interfaces / abstractions**, not hardcoded new ChromeDriver() or concrete pages.
- WebDriver, config, and Page Objects should be **injected**, not statically accessed.

```

1 public class LoginTest {
2     @Test
3     public void login() {
4         WebDriver driver = new ChromeDriver(); // hard dependency
5         LoginPage loginPage = new LoginPage(driver);
6         // ...
7     }
8 }

```

If you want to switch to Firefox, run locally vs. grid, or mock for unit tests, you must edit tests everywhere.

DIP-inspired design

Abstraction for driver creation:

```

1  public interface WebDriverFactory {
2      WebDriver create();
3  }
4  //Concrete factories:
5  public class ChromeDriverFactory implements WebDriverFactory {
6      public WebDriver create() {
7          // set up options, capabilities, grid URL, etc.
8          return new ChromeDriver();
9      }
10 }
11 //Test depends on abstraction:
12 public class LoginTest {
13     private final WebDriverFactory driverFactory = new ChromeDriverFactory();
14     @Test
15     void userCanLogin() {
16         WebDriver driver = driverFactory.create();
17         ILoginPage loginPage = new WebLoginPage(driver);
18         // ...
19     }
20 }

```

Later, you can swap ChromeDriverFactory for RemoteWebDriverFactory or FirefoxDriverFactory with minimal changes.

For beginners, DIP teaches:

- Don't hardwire WebDriver or page creation into tests.
- Build **pluggable** automation architectures.

How SOLID becomes a learning roadmap for POM beginners

You can think of SOLID as a **step-by-step learning path** for building good Page Object frameworks:

1. Start with S (SRP)

- Separate *tests* from *pages* from *utilities*.
- Aim for "1 class, 1 responsibility".

2. Then apply O (OCP)

- Introduce BasePage and Inheritance
- Add reusable wrappers (click, type, waits) so new pages are just extensions.

3. Next, L (LSP)

- Introduce **interfaces for pages** (ILoginPage, IDashboardPage).
- Design tests against these contracts so you can swap implementations (web/mobile).

4. Then, I (ISP)

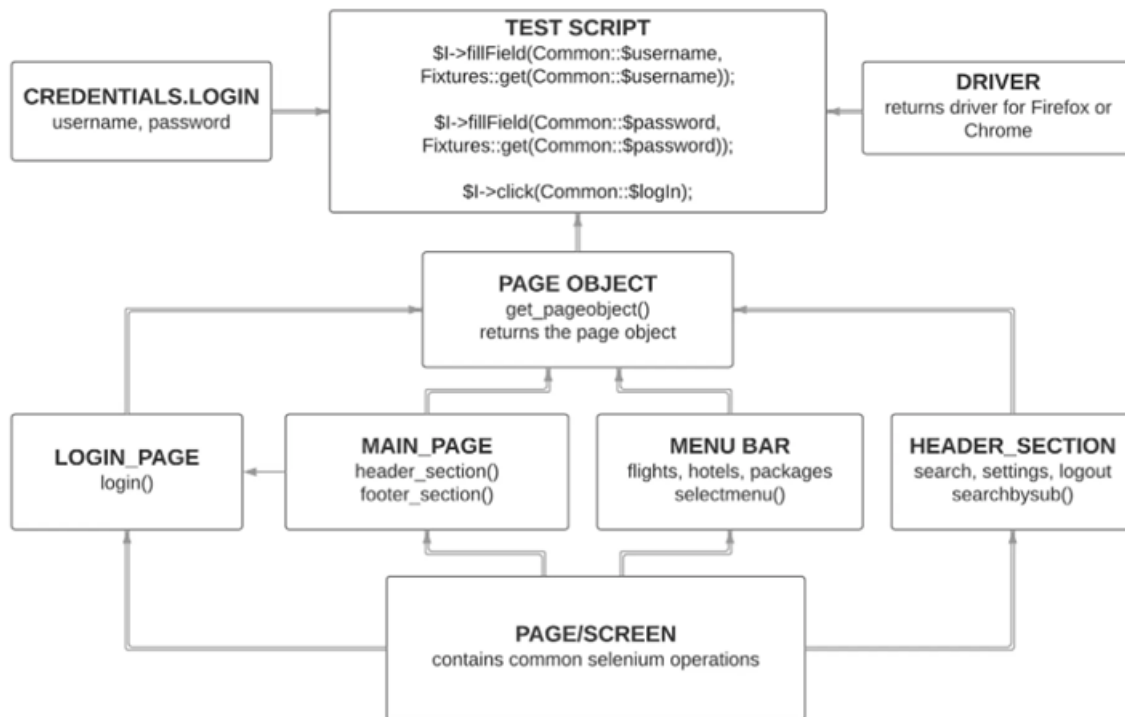
- Break down giant page interfaces/classes into **capabilities** (Searchable, Filterable, etc.).
- Avoid "one class that does everything."

5. Finally, D (DIP)

- Move WebDriver and Page object creation behind **factories** or a DI mechanism.
- Let tests depend on **abstractions**, not concrete driver/page types.

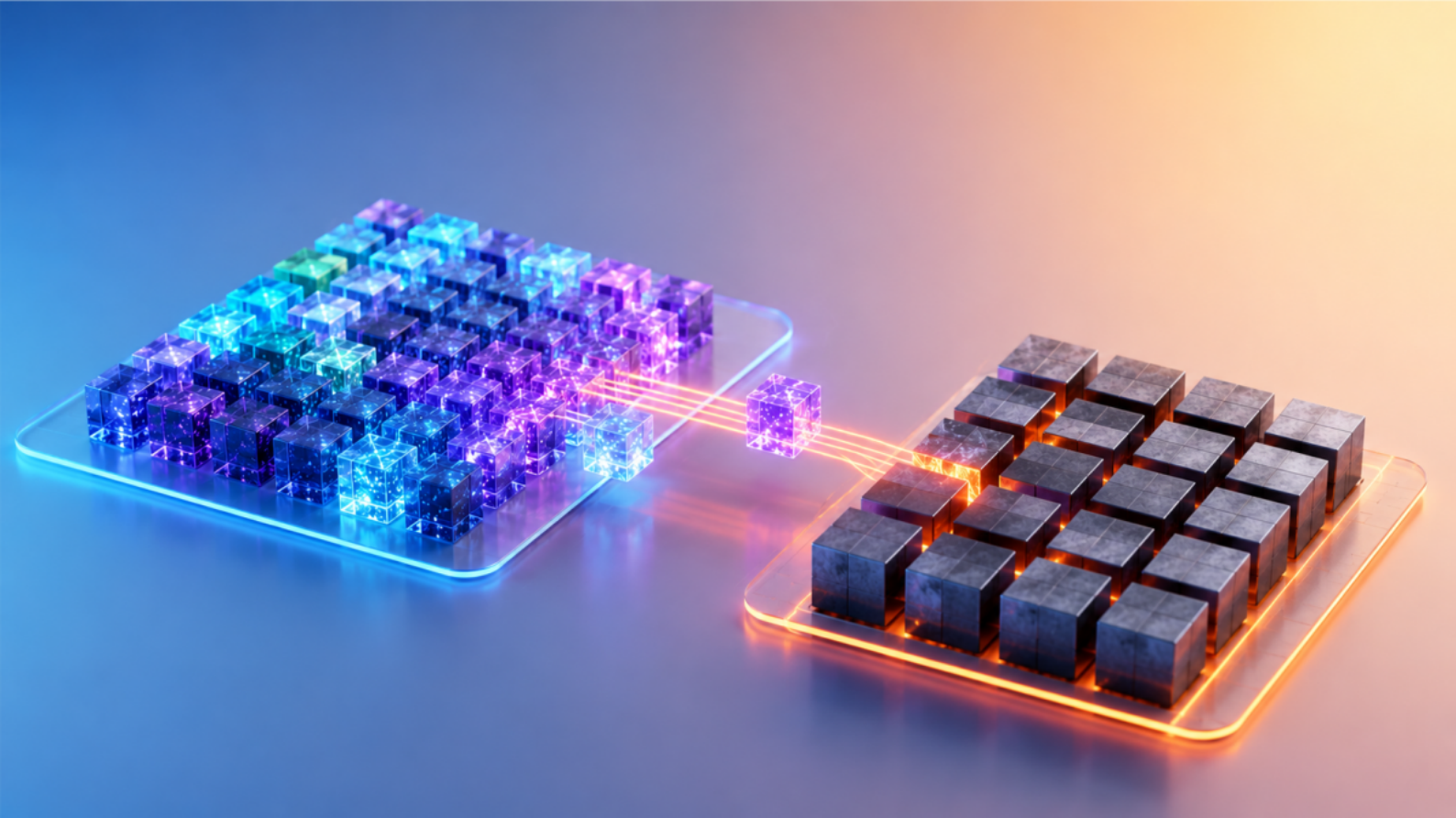
If you'd like, I can help you **take one of your existing POM test classes** and refactor it step-by-step using SOLID, so you can see the before/after clearly.

PAGE OBJECT MODEL



Page Object Model (POM)

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



OpenJ9 GC Policies: How to Choose the Right Collector

CoreJava

OpenJ9

Authors: Babneet Singh & Aleksandar Micic & Dmitri Pivkine & Tobi Ajila

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

Aleksandar Micic specializes in low-level optimizations, data structures, algorithms, parallel programming and building scalable, low-pause and low-overhead Garbage Collectors.



LinkedIn



Google Scholar

Authors: Babneet Singh & Aleksandar Micic & Dmitri Pivkine & Tobi Ajila

Dmitri Pivkine is Software Developer in Open J9 Java Garbage Collection Team since 2007.



LinkedIn

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



LinkedIn

For many OpenJ9 users, garbage collection tuning starts with the wrong question.

They ask, “Which collector is the fastest?” or “Which one is best for cloud?” In practice, the better question is: **what does my application actually need from the heap?**

That is the right starting point because OpenJ9 gives you several distinct GC policies, each designed around a different balance of throughput, pause time, heap geometry, and operational behavior. The default is **gencon**, and for a broad spectrum of Java applications that is still the right place to start. But once heaps grow, pause spikes become operationally expensive, or throughput becomes the dominant goal, a different policy may be the better fit.

Modern Java deployments make this more relevant, not less. Kubernetes microservices, virtual-thread-heavy services, streaming systems, API gateways, JVM-based AI orchestration layers, RAG services, batch analytics, and memory-dense backend platforms do not all behave in the same way. They should not all inherit the same GC assumptions by default.

This guide is not meant to turn every reader into a GC specialist. Its goal is simpler: help OpenJ9 users choose a sensible starting policy, understand the tradeoffs behind each option, and know when a different choice is justified. For the official baseline, see the [OpenJ9 GC policies](#) page and the [OpenJ9 -Xgcpolicy reference](#).

How to Choose an OpenJ9 GC Policy

The most useful way to think about GC policy selection is to begin with workload shape.

Are you allocating many short-lived objects? Is the heap large enough that pause consistency matters more than raw throughput? Is the workload interactive and latency-sensitive, or throughput-first and non-interactive? Do you need deterministic pauses, or just a better pause profile than you have today? Is the process so short-lived that reclaiming memory is mostly wasted work?

Those are the real decision points.

OpenJ9’s own documentation aligns well with that view. The default **gencon** policy is described as a strong general-purpose choice, especially for transactional applications with many short-lived objects. The same documentation also

emphasizes that policy choice should be informed by real GC logs rather than assumptions. In other words, selection is part of a measurement loop, not just a startup flag. See the [OpenJ9 GC policies](#) page and the [verbose GC logging guide](#).

Before diving into the policies, one note on comparisons: the HotSpot mappings below are **mental models, not equivalence claims**. They are there to help experienced HotSpot users quickly orient themselves, not to imply identical ergonomics or pause behavior.

OpenJ9 GC Policy Comparison Table

If you want the short answer first, start here:

OpenJ9 GC Policies – Workload Fit Overview

Policy	Best starting fit	Main strengths	Main tradeoffs	Modern workload examples
gencon	General-purpose server workloads with many short-lived objects	Strong default balance of throughput and pause time; broad-spectrum fit for many transactional workloads	Can run into fragmentation, global compaction pressure, or variability issues on some large or uneven heaps	Spring Boot APIs, REST services, event-driven services, Kafka consumers/producers, virtual-thread services, LLM/RAG gateways, agent orchestration backends
balanced	Large heaps, often in the tens of GB, where gencon shows compaction-driven pause spikes or rising pause variability	Better pause profile on large heaps; addresses several large-heap pain points more gracefully than gencon	More specialized than gencon; no hard pause bound; 64-bit only; can give up some throughput	Large monoliths, cache-heavy backends, JVM analytics coordinators, search/routing platforms, memory-dense stateful services
optavgpause	Large flat heaps where lower average pause matters more than throughput	Lower average pause than optthruput; can avoid some generational-specific overheads	Reduced throughput; still not a real-time policy; usually needs more heap than generational policies	Some session-heavy services, selected large-heap backends, specialized long-running services
optthruput	Throughput-first, non-interactive workloads where completion rate matters more than pause time	Good fit when total work matters more than pause consistency; useful flat-heap baseline	Long stop-the-world pauses can occur; typically wants more heap than generational policies	Batch ETL, offline analytics, reconciliation jobs, blockchain indexing pipelines, data preprocessing
metronome	Deterministic pause-sensitive systems with real pause bounds in the requirements	Short, bounded pauses with explicit utilization goals	Specialized tuning, fully expanded heap, limited platform support, noticeable throughput cost	Real-time control, telecom, industrial or embedded Java services
nogc	Very short-lived or tightly controlled runs where you want to remove GC effects for experimentation or baselining	Minimal GC runtime impact; useful for controlled experiments, performance runs, and GC-free baselines	No reclamation at all; heap exhaustion stops the VM; very few real-world long-running applications are a good fit	Short-lived tools, controlled benchmarks, tightly bounded helper processes
Concurrent Scavenge (gencon mode)	Large-heap, response-time-sensitive gencon workloads where lower average pause time matters	Can reduce average nursery pause time without leaving the generational model	Not a separate policy; worst-case pauses can still be significant; tuning may still be needed	Large-heap transactional services, response-time-sensitive server workloads already using gencon

The official descriptions for the six primary policies are in the [OpenJ9 GC policies](#) page and the [OpenJ9-Xgcpolicy reference](#). Concurrent Scavenge is documented separately under [OpenJ9-Xgc options](#), because it is a **gencon** mode rather than a separate policy.

gencon

In one line: **gencon** is the right starting point for most transactional, server-side Java workloads.

Most closely resembles: Hotspot **CMS** as a mental model on Java 8, and in some usage patterns the young-generation behavior people often associate with default server collectors. This is only an orientation aid, not a one-to-one mapping.

How It Works?

gencon is OpenJ9's default **generational concurrent** policy. The heap is split into a **nursery** and **tenure** space. Young objects are handled with **Scavenge** collections in the nursery, while old-generation work is handled through **concurrent global mark** with optional compaction. In practice, this design is optimized for workloads with high short-lived object churn. It also uses heuristics around **tenuring thresholds**, **Tenure headroom**, and **nursery tilting** to balance allocation pressure and liveness. Those heuristics are powerful, but under highly variable workloads they can also mispredict and contribute to **aborted Scavenges**, which are expensive to recover from.

Strengths

For mainstream server-side workloads, **gencon** often delivers the strongest overall throughput profile while still keeping average pause times and, in many cases, worst-case pauses within acceptable ranges. It has excellent footprint characteristics, aligns well with short-lived allocation churn, and requires the least justification to adopt because it is already the OpenJ9 default. It is broad-spectrum and remains the best baseline for many cloud-native services, messaging backends, and AI orchestration layers.

Weaknesses & Mechanical Limits

This is where the deep mechanics matter. If object sizes and lifetimes vary widely, **fragmentation** can build up and lead to **global heap compaction**. If class unloading becomes expensive, **global GC** can become more disruptive than average pause metrics suggest, especially in workloads that churn many class loaders. The simple nursery/tenure geometry can also become harder to balance when the heap reaches full expansion. In that state, the VM may struggle to decide whether the nursery should continue to grow at the cost of tenure, or vice versa.

Workloads using **JNI Critical** with large arrays can trigger extra global activity because pinned objects cannot be moved by Scavenge. Highly variable workloads can also expose weaknesses in **pre-emptive concurrent global kickoff**, **Tenure headroom**, and **nursery tilting** heuristics. In extreme cases that can lead to **aborted Scavenges**, which produce exactly the kind of long pauses teams are trying to avoid. Tuning can reduce the problem, but only up to a point.

Practical Recommendation

Start with **gencon** for most general-purpose Java services: Spring Boot and Jakarta EE applications, REST and GraphQL APIs, event-driven microservices, Kafka-connected services, virtual-thread-based request handlers, and most JVM-based AI orchestration layers. If the workload remains request-driven and dominated by short-lived allocation churn, the default is usually right. Move away from **gencon** only when the workload gives you a concrete reason, such as compaction-driven pause spikes, class-unloading costs, or large-heap variability that tuning does not adequately address. See the [OpenJ9 GC policies](#) page and IBM's [garbage collection tradeoffs and tuning with OpenJ9](#) article.

balanced

In one line: **balanced** is the first collector to test when larger heaps make pause behavior harder to control under **gencon**.

Most closely resembles: HotSpot **G1GC** as a mental model, especially for readers who think in terms of region-based heap management. The OpenJ9 new-user material also positions it as the closest OpenJ9 policy to the default HotSpot policy on Java 11.

How It Works?

balanced is a **64-bit-only** policy that divides the heap into **regions** managed by an **incremental generational collector**. Its purpose is to reduce maximum pause time on large heaps by changing the geometry of the problem. Instead of relying on a simple nursery/tenure split, it can perform more incremental work and react more flexibly to large-heap behavior. It can also transition **Partial GC** behavior dynamically, for example from evacuating style work to **mark-and-compact** when empty space is tight. That is one reason it often handles variability more gracefully than **gencon** under memory-dense conditions.

Strengths

The strength of **balanced** is not simply that it is “better for large heaps”. Its real value is that it reduces pressure from some of the most expensive whole-heap activities. **Defragmentation** becomes more incremental. **Class unloading** can happen with finer granularity. **JNI Critical pinning** tends to be less disruptive. **Eden size tuning** pressure is reduced because region-based geometry and balancing heuristics can adapt more smoothly than a strict nursery/tenure split. In some large, memory-dense systems, **balanced** can feel more plug-and-play than **gencon**, even if it is less commonly used.

Weaknesses & Mechanical Limits

The main tradeoff is that **balanced** buys flexibility at some cost in application throughput and average GC cost. It is not a real-time collector and does not provide a hard pause bound. Actual pause behavior still depends on allocation rates, survival rates, and fragmentation. In other words, it improves the pause profile for the kinds of large-heap problems it is designed for, but it does not turn a large heap into a deterministic one. It is also 64-bit only.

What counts as a **large heap**? The best answer is behavioral rather than purely numeric. In practice, many teams start looking harder at **balanced** once heaps move into the **tens of gigabytes**, often around **16 GB to 32 GB and above**, but there is no universal cutoff. The more meaningful trigger is a combination of heap size, live-set shape, allocation behavior, and symptoms in GC logs, especially **global collections**, **compaction-driven pause spikes**, or rising pause variability under **gencon**.

Practical Recommendation

Evaluate **balanced** when **gencon** is fundamentally workable but pause behavior becomes unacceptable as the heap grows. It is a strong candidate for large memory-heavy services, cache-rich monoliths, search or analytics coordinators, and AI-serving platforms that hold large caches, retrieval metadata, embeddings-related state, or other long-lived coordination structures. The right message is not “use **balanced** for low latency”. The right message is “use **balanced** when large heaps are making pause behavior less predictable under **gencon**, and you need a smoother pause profile without moving into deterministic GC territory”. See the [OpenJ9 GC policies](#) page and IBM’s [OpenJ9 tuning article](#).

optavgpause

In one line: **optavgpause** is a specialized flat-heap option when lower average pause matters more than throughput.

Most closely resembles: There is no perfect HotSpot equivalent. Conceptually it sits in the family of pause-oriented non-generational collectors, but it is better understood on its own terms as a flat-heap tradeoff policy.

How It Works?

optavgpause is a **flat-heap** policy that starts **global GC preemptively** so the cycle can complete before the heap is exhausted. It does not use a nursery/tenure generational split. That means some of the generational-specific mechanisms and pathologies simply do not exist in the same form. There is no nursery tilting, no classic aborted Scavenge, and no remembered-set maintenance cost in the same sense as a generational collector. The tradeoff is that more work must be done across the flat heap, which shifts cost from worst-case pause behavior toward average throughput.

Strengths

Its core strength is exactly what the name suggests: lower average pause time than **optthruput** on workloads where that tradeoff matters. OpenJ9 also notes that its non-generational write-barrier strategy can help some workloads that frequently modify large, old reference arrays. It can therefore be attractive when you intentionally want a flat-heap policy and you are trying to avoid some of the generational-specific overheads that show up in **gencon**.

Weaknesses & Mechanical Limits

The main tradeoff is reduced throughput. Like any non-generational policy, **optavgpause** usually wants more heap than a generational collector to keep overhead in the same general range. And OpenJ9 is explicit that in many situations the default **gencon** policy still performs better overall. That is an important limitation to state clearly because it prevents **optavgpause** from sounding like a universally safer or more modern answer when it is really a specialized one.

Practical Recommendation

Consider **optavgpause** when you have a large heap, you are intentionally evaluating flat-heap policies, average pause improvement matters, and **gencon** and **balanced** are not giving the behavior you need. It is more plausible for selected long-running services or session-heavy workloads than for mainstream request-driven cloud applications.

optthruput

In one line: **optthruput** is the right collector to evaluate when completion rate matters more than pause consistency.

Most closely resembles: Conceptually closest to a **throughput-first flat-heap** strategy, and as a rough HotSpot mental model it plays a role similar to how engineers think about **ParallelGC** style tradeoffs: throughput first, pause time second. That is a directional analogy, not an equivalence claim.

How It Works?

optthruput is a **flat-heap** policy that runs **global mark-sweep**, optionally followed by compaction, when allocation failure occurs. Because the collector requires **exclusive access to the heap**, application threads halt while GC runs. This is the most direct, throughput-first shape in the OpenJ9 policy family. It also provides a useful baseline when you want to understand what more advanced policies are costing you in mechanisms such as write barriers, remembered sets, or extra concurrent work.

Strengths

In the right workload, **optthruput** can deliver strong overall completion efficiency. It also avoids some issues with very large objects that can appear with **gencon**'s nursery/tenure structure. For teams doing experimentation, **optthruput** and **optavgpause** can also be useful comparison points for understanding the cost of more advanced policy machinery.

Weaknesses & Mechanical Limits

The downside is exactly as serious as it sounds: **long stop-the-world pauses**. That alone rules **optthruput** out for many user-facing services. Like other non-generational policies, it typically wants more heap than a generational collector to maintain comparable overhead. It is also worth stating plainly that “throughput-first” does not automatically mean “higher throughput than **gencon**“. In many real server workloads, **gencon** still wins overall. The value of **optthruput** is that it can be the right answer for the right class of workloads, not that it supersedes the default.

Practical Recommendation

Use **optthruput** when throughput is clearly the primary objective and latency spikes are acceptable. It fits naturally with non-interactive, throughput-first jobs: batch ETL, offline analytics, reconciliation work, scheduled reporting, blockchain indexing, ledger analysis, and large data preprocessing pipelines for AI systems. See the [OpenJ9 GC policies](#) page.

metronome

In one line: **metronome** is for systems that genuinely need bounded GC pauses.

Most closely resembles: Conceptually closer to the family of **deterministic low-pause** collectors, and as a rough mental model readers may compare its goals to what they expect from **ZGC/Shenandoah**-style pause-focused designs. But **metronome** is really its own thing: an incremental deterministic collector with explicit utilization goals.

How It Works?

OpenJ9 documents **metronome** as being designed for applications that require a precise upper bound on collection pause times together with a specified application utilization target. It runs in short interruptible bursts to avoid long stop-the-world pauses. It is supported only on **Linux x86-64 and AIX**, and it uses a **fully expanded heap** even when `-Xms` is lower than `-Xmx`. One mechanical consequence is that certain JNI array access paths can become slower because of **arraylet reconstitution**.

Strengths

Its strength is not convenience. Its strength is **determinism**. If your system genuinely requires bounded GC pauses, **metronome** is the OpenJ9 policy built for that purpose. That is the key reason it exists.

Weaknesses & Mechanical Limits

The costs are real: limited platform support, iterative heap and utilization tuning, a fully expanded heap, slower JNI array access in some scenarios because of arraylet mechanics, and noticeably compromised throughput compared with more general-purpose policies. This is not a collector to adopt casually, and it should not be described as a generic low-latency upgrade for mainstream cloud services.

Practical Recommendation

Use **metronome** for systems where deterministic pause behavior is genuinely part of the requirements: selected telecom, industrial, control, or embedded-style Java systems. See the [OpenJ9 GC policies](#) page.

nogc

In one line: **nogc** is a useful specialist tool for baselining and controlled runs, but very few real long-running applications should use it.

Most closely resembles: There is no real HotSpot equivalent because the defining characteristic here is the absence of reclamation, not a different reclamation strategy.

How It Works?

OpenJ9 says **nogc** handles allocation and heap expansion, but **does not reclaim memory**. If the heap is exhausted, an **OutOfMemoryError** is triggered and the VM stops. Mechanically, that makes it the cleanest way to remove GC reclamation effects from a run, but it also makes it unsuitable for most real systems.

Strengths

Its value is in **controlled situations**: benchmarking baselines, validating memory sizing, very short-lived programs, tightly bounded helper processes, and experiments where you want to remove the effect of GC from the measurement as much as possible.

Weaknesses & Mechanical Limits

The weakness is absolute: **memory is never reclaimed**. That makes **nogc** risky for almost all long-running services. OpenJ9 also explicitly warns to be careful with **finalization, direct memory access**, and **weak, soft, and phantom references** under this policy. And to make the point explicit: **very few real-world long-running applications are a good fit for nogc**.

Practical Recommendation

Treat **nogc** as a specialist tool, not a production default. Use it for controlled experiments, bounded helper processes, and GC-free baselines. See the [OpenJ9 GC policies](#) page.

Concurrent Scavenge (gencon mode)

In one line: **Concurrent Scavenge** is an advanced **gencon** mode that can improve average pause behavior for some large-heap, response-time-sensitive workloads.

Most closely resembles: Conceptually overlaps with how readers think about **ZGC** or **Shenandoah** in the sense that it tries to improve average pause behavior with more concurrent work, but it remains a **gencon** mode and does not provide the same style of worst-case guarantees.

How It Works?

It is **not a separate GC policy**. It is a **gencon** mode enabled through `-Xgc:concurrentScavenge`. OpenJ9 documents it as a pause-less garbage collection mode for the Generational Concurrent policy, intended to reduce GC pause times for response-time-sensitive, large-heap applications. It is **64-bit only** and cannot be used with the other policies.

Mechanically, it is useful to think of Concurrent Scavenge as replacing a traditional stop-the-world Scavenge with shorter **STW opening and closing phases** and a **Concurrent Copy** phase in between. That means average pause time can improve significantly, while preserving the benefits of a generational model.

Strengths

It sits in a very useful middle ground between “stay on plain **gencon**” and “switch policies entirely”. For some large-heap, response-time-sensitive services, it can reduce average nursery pause time without abandoning the generational model. **Concurrent Scavenge** preserves relatively strong throughput and footprint characteristics compared with non-generational alternatives.

Weaknesses & Mechanical Limits

It is still a tuning option, not a universal answer. **Worst-case pauses can remain significant**, especially under highly variable workloads or CPU over-utilization. Aborted Scavenges remain possible, which is why this is not a magical low-pause switch. It improves average pause behavior, but it does not erase every worst-case scenario.

Practical Recommendation

Evaluate **Concurrent Scavenge** before switching away from **gencon** if your workload is already fundamentally a good fit for a generational collector but average pause reduction matters, especially on large heaps in response-time-sensitive services. See [OpenJ9 -Xgc options](#).

What to Look For in OpenJ9 Verbose GC Logs

When checking the logs, it is important to know where to start.

OpenJ9’s verbose GC logs are one of the most useful official references because they support both tuning and troubleshooting. The logs capture GC operations, help analyze long pauses, show how free memory is divided in the heap before and after a cycle, and can be visualized with tools like GCMV. Logs are enabled with `-verbose:gc`, printed to STDERR by default, and can be written to files with `-Xverbosegclog`. See the [verbose GC logs guide](#).

The [annotated verbose GC log examples](#) page is especially useful because it moves from raw XML to interpretation. The examples show how allocation-failure triggers appear, how scavenge cycles are marked, and how memory sections describe nursery and tenure state before and after collection.

In practical terms, GC logs help confirm policy mismatch:

- if you are on **gencon** and keep seeing long global-compaction-related pauses, **balanced** may be worth testing
- if allocation-failure-driven cycles dominate a non-interactive batch workload, **optthruput** may be more natural
- if average pause behavior is the real problem on a large-heap **gencon** service, **Concurrent Scavenge** may be worth evaluating before switching policies entirely

For a tuning-oriented companion overview, IBM's [garbage collection tradeoffs and tuning with OpenJ9](#) article is also useful.

A Rough HotSpot Mental Model

For readers coming from HotSpot, a small amount of context helps.

OpenJ9's [new-user guide](#) gives two valuable orientation points. On Java 8, **gencon** is the OpenJ9 policy most similar to HotSpot **CMS**. On Java 11, **balanced** is the OpenJ9 policy most similar to the default HotSpot policy. These are **guideposts**, **not equivalence claims**. OpenJ9 has its own ergonomics, heap geometry, and tradeoffs. The value of the comparison is simply to help readers anchor expectations more quickly.

A similarly useful mental model applies to **Concurrent Scavenge**: it occupies a conceptual space that overlaps with collectors like **ZGC** or **Shenandoah** in the sense that it tries to improve average pause behavior with more concurrent work. But again, this is an orientation aid, not an equivalence claim.

Workload-Based Policy Guidance

If you want the shortest practical summary:

For cloud-native APIs, microservices, event-driven services, and most virtual-thread-heavy server applications, start with **gencon**.

For large memory-dense services where pause consistency has become the dominant pain point, evaluate **balanced** next.

For batch analytics, ETL, reconciliation, indexing, and offline preprocessing, consider **optthruput**, and in selected cases **optavgpause**.

For deterministic low-latency systems with real pause bounds in the requirements, evaluate **metronome**.

For very short-lived tools, memory baselining, or tightly controlled experiments, **nogc** can be useful, but only with clear heap boundaries and the understanding that memory will never be reclaimed. And for response-time-sensitive, large-heap **gencon** workloads where average pause reduction matters, **Concurrent Scavenge** is worth testing before switching policies completely.

FAQ

Which OpenJ9 GC policy should I use by default?

For most general-purpose Java server applications, start with **gencon**.

When should I switch from **gencon** to **balanced**?

When larger heaps start producing compaction-driven pause spikes, frequent global collections, or rising pause variability under **gencon**.

What counts as a large heap in OpenJ9?

There is no universal cutoff, but many teams start paying closer attention once heaps move into the tens of gigabytes, often around **16 GB to 32 GB and above**. The better trigger is behavior, not a number.

Is **Concurrent Scavenge** a separate GC policy?

No. It is a **gencon** mode enabled through `-Xgc:concurrentScavenge`.

Is **nogc** suitable for production?

Usually no. It is mainly useful for controlled experiments, baselines, and very short-lived processes. Very few real-world long-running applications are a good fit.

Final Thoughts

Most users do not need to overcomplicate OpenJ9 GC selection.

The best advice for the majority of modern Java applications is still straightforward: start with **gencon**, enable GC logging, establish a baseline from evidence, and switch policies only when your application behavior gives you a concrete reason. OpenJ9 offers specialized alternatives, but each one exists for a specific workload shape and tradeoff profile, not as a universal upgrade. See the [OpenJ9 GC policies](#) page for the authoritative policy descriptions.

References

- [OpenJ9 GC policies](#)
- [OpenJ9 -Xgcpolicy reference](#)
- [OpenJ9 verbose GC logs](#)
- [OpenJ9 verbose GC log examples](#)
- [New to OpenJ9?](#)
- [OpenJ9 -Xgc options, including Concurrent Scavenge](#)
- [IBM: Garbage collection tradeoffs and tuning with OpenJ9](#)

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

JCON2027
www.jcon.one

APR
26-29



JAVAPRO

📍 CINEDOM COLOGNE

JCON EUROPE 2027
International Java Community Conference



Elasticsearch Java SDK: No Magic, Just Solid Design Choices

API & Frameworks

FunctionalProgramming

SoftwareDesign

Author: Stefano Fago

Stefano Fago is an informatic passionate about distributed systems, software design, and software architectures with experiences as a consultant (in different contexts) and trainer (Java/J2EE), software designer, team leader, and speaker in some Italian meetups; interested in too many topics! Aficionado of the Commodore 64.



LinkedIn

Java's API ecosystem is massive, but quality is not granted for free, and not necessarily widespread. Building a good API is not trivial, and focus is required. You need balanced abstractions, keep the threading model sane, and provide developers a satisfying experience.

The Elasticsearch Java SDK caught my attention because it actually gets this stuff right. I dug into the design decisions that make it work, and the overall approach is worth understanding, even if there are trade-offs to consider.

Specification-Driven Code Generation

The Elasticsearch Java SDK employs a hybrid approach: generated code combined with hand-crafted components. The bulk of the client—model classes, builders, serializers, and namespace methods—generates from a canonical [API specification](#) written in TypeScript.

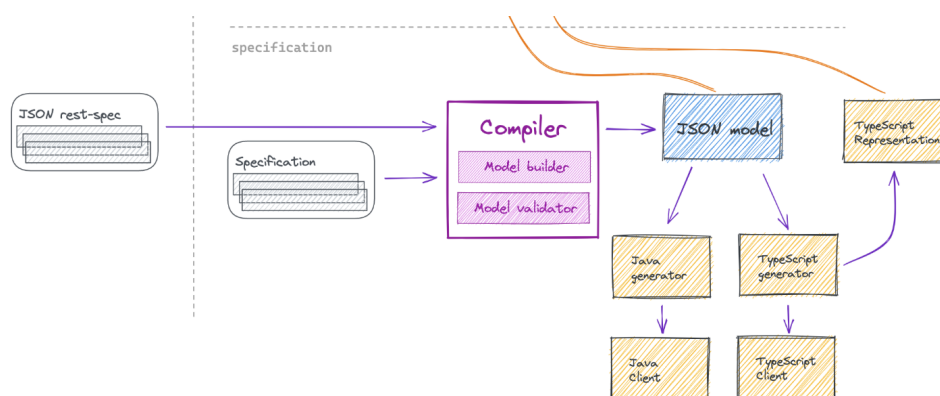


Abb. 1: Focus on java code generation by spec in Elasticsearch SDK

This specification-first methodology ensures consistency across hundreds of endpoints and multiple language implementations.

The hand-written components focus on critical concerns:

- Transport integration with the Low-Level REST Client
- Core infrastructure: authentication, TLS, retries, and JSON mapping
- API ergonomics: builder patterns, nullability conventions, naming standards
- [ADR-driven decisions](#) documenting design rationale

This division exploits the strengths of both approaches. Generated code maintains consistency and coverage, while human-authored code addresses concerns requiring nuanced judgment.

The Builder Pattern with Safety Guarantees

The SDK relies extensively on the Builder Pattern through a base `ObjectBuilder<T>` interface. Constructing search queries, index mappings, or bulk operations without builders would introduce significant complexity and error potential.

The implementation provides builder-lambda overloads `(() -> ObjectBuilder<T>)`, enabling inline construction of nested objects with type-safe closures. These appear in the public Javadocs as function-typed builder setters. Generated builders inherit from `ObjectBuilderBase`, which enforces single-use semantics via `checkSingleUse()` when `build()` is invoked.

Once `build()` executes, the builder becomes unusable. This restriction exists because internal structures, particularly collections, may be shared between the builder and the constructed object. Allowing reuse could introduce subtle corruption. The design establishes a clear contract: configure once, build once, then discard.

```

1  SearchRequest request = SearchRequest.of(s -> s
2      .index("products")
3      .query(q -> q
4          .match(m -> m
5              .field("name")
6              .query("laptop")
7          )
8      )
9  );

```

While this may initially appear verbose, it constructs a typed, nested DSL that corresponds directly to Elasticsearch's query structure. The `of()` static factory method instantiates the builder, applies the configuration function, and finalizes construction. The result is declarative Java code that mirrors the Elasticsearch query [DSL](#).

Type-Safe DSL Through Lambda Composition

The lambda-based API provides an effective mechanism for authoring deeply nested structures while maintaining strong typing. The API also provides an effective and IDE-friendly implementation. Consider this query composition:

```

1  client.search(s -> s
2      .index("products")
3      .query(q -> q
4          .bool(b -> b
5              .must(m -> m
6                  .match(t -> t
7                      .field("description")
8                      .query("wireless")
9                  )
10             )
11         )
12     )
13 );

```

Each lambda represents a typed configuration step. The approach avoids stringly-typed APIs and magic maps, instead constructing a statically verified tree. This eliminates common pitfalls associated with mutable builder chains where nulls and state management complicate reasoning about correctness.

Modeling Variants With Tagged Unions

Elasticsearch queries represent variant types—a `Query` may be a `MatchQuery`, `BoolQuery`, `RangeQuery`, or numerous other forms. The SDK models these through a [Tagged Union](#) pattern implementing the `TaggedUnion<Tag, BaseType>` interface:

```

1  public interface TaggedUnion<Tag extends Enum<?>, BaseType> {
2      Tag _kind();
3      BaseType _get();
4  }

```

A tagged union is a data structure that can hold values of different data types, but only one at a time. It's similar to a regular union, but includes a tag (or discriminator) that shows which data type is currently stored. This tag allows type-safe access to the stored value, preventing accidental data misuse. The `_kind()` method exposes the discriminator, while `_get()` returns the strongly-typed value:

```

1  Query query = Query.of(q -> q
2      .match(m -> m.field("title").query("elasticsearch"))
3  );
4  if (query._kind() == Query.Kind.Match) {
5      MatchQuery match = (MatchQuery) query._get();
6      // Process match query
7  }

```

This design solution allows reasoning with the union construct even without the syntactic support of the most recent versions of Java, while maintaining backward compatibility with those who already used Elastic products in previous installations. Starting with Java 16+, we have structural pattern matching that could be used in the future for an evolution of the SDK; for example, we can use the existing design and part of the new syntax:

```
1 switch (query._kind()) {
2     case Match -> {
3         MatchQuery match = (MatchQuery) query._get();
4     }
5     case Term -> {
6         TermQuery term = (TermQuery) query._get();
7     }
8 }
```

Namespace Organization and API Modularity

Elasticsearch exposes hundreds of endpoints across distinct functional domains. The SDK organizes these through namespace clients, each residing in a dedicated subpackage of `co.elastic.clients.elasticsearch`:

```
1 ElasticsearchClient client = new ElasticsearchClient(transport);
2
3 client.indices().create(c -> c.index("catalog"));
4
5 client.search(s -> s
6     .index("products")
7     .query(q -> q.match(m -> m.field("name").query("laptop"))))
8 );
```

Each namespace client exposes operations relevant to its domain. This organization provides several advantages: IDE autocomplete shows only applicable methods, the structure maps to the REST API organization, and maintenance becomes tractable. Namespace clients are lightweight objects created on demand without significant overhead.

Pragmatic Nullability Conventions

The SDK represents optional values as `null` with `@Nullable` annotations, rather than `Optional<T>`. This decision acknowledges the Java ecosystem's pervasive use of null. While `Optional`, it offers theoretical advantages; practical integration with null-based libraries would require constant wrapping and unwrapping. The pragmatic choice prioritizes ecosystem compatibility.

Transport Abstraction and Separation of Concerns

The SDK separates protocol handling from typed request/response modeling. By default, `RestClientTransport` delegates to a low-level HTTP client managing connections, pooling, retries, and node discovery:

```
1 ElasticsearchTransport transport = new RestClientTransport(
2     restClient,
3     new JacksonJsonMapper());
4 ElasticsearchClient client = new ElasticsearchClient(transport);
```

This separation implements the Adapter pattern, allowing HTTP stack substitution without modifying client code. The pluggable `JsonMapper` similarly implements the Strategy pattern for serialization, supporting alternatives like Jackson or JSON-B implementations.

Exception Design and Error Handling

The SDK employs a two-tier exception model separating application-level from infrastructure-level failures:

ElasticsearchException (runtime exception) captures server-side rejections, including validation errors, query syntax issues, and timeouts. It includes detailed error information from Elasticsearch.

TransportException (checked exception) captures transport-layer failures, including network errors, connection timeouts, and unavailable servers. Its cause contains the lower-level exception, typically **ResponseException** for **RestClientTransport**.

```

1  try {
2      SearchResponse<Product> response = client.search(s -> s
3          .index("products")
4          .query(q -> q.match(m -> m.field("name").query("laptop"))),
5          Product.class
6      );
7  } catch (ElasticsearchException e) {
8      logger.error("Query error: {}", e.error().type());    // Server rejected request
9  } catch (TransportException e) {
10     logger.error("Connection failed: {}", e.getCause());  // Transport failure
11 }

```

This design provides clear diagnostic information while avoiding complex exception hierarchies.

Not directly related to exceptions, but worth noting is the usage of [Closeable](#) and [Autocloseable](#), adopted for different elements, that report the possible use of the [try-with-resource](#) idiomatic pattern.

Trade-Offs and Considerations

Immutability and single-use builders produce short-lived allocations. In most applications, this overhead remains negligible. High-throughput scenarios may benefit from benchmarking and optimizing batching strategies. The SDK could leverage modern Java features—records, sealed classes, enhanced pattern matching. However, this would require continuous refactoring to track language evolution and potentially compromise backward compatibility with existing deployments.

Conclusion

The Elasticsearch Java SDK demonstrates that effective API design emerges from deliberate choices addressing real constraints. Generated code ensures consistency, builders enforce immutability, tagged unions provide type-safe polymorphism, and namespace organization maintains clarity at scale.

PATTERN	PURPOSE
Single-Use Builder	Prevents shared mutable state
Lambda-Based Fluent Interface	Type-safe nested DSL construction
Tagged Union	Explicit variant type modeling
Namespace Client	Domain-aligned API organization
Transport Adapter	Pluggable HTTP implementation
Strategy (JsonMapper)	Configurable serialization
Two-Tier Exceptions	Clear error categorization
Forward Compatibility	Non-breaking version evolution

These patterns address concrete problems: thread safety, type safety, maintainability, and developer experience. The resulting SDK achieves consistency without rigidity, supporting both current deployments and future evolution.



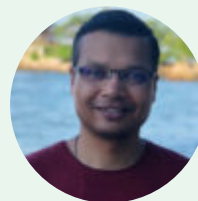
Java Developers, You're Already Ready for Blockchain — You Just Don't Know It Yet

API & Frameworks

Blockchain

Author: Satya Ranjan

Satya is an experienced Java developer and architect, as well as an open-source contributor focused on blockchain infrastructure and developer tooling for Java developers. He works extensively on Cardano-related projects, building libraries and frameworks for blockchain indexing, transaction processing, and developer experience. He currently works as a Lead Blockchain Architect at the Cardano Foundation and is one of the core maintainers of open-source projects under the BloxBean GitHub organization.



LinkedIn

The Java ecosystem has a mature way of doing things. Pick your framework. Use Maven or Gradle. Write tests with JUnit. Run everything locally with Docker. These patterns have been stable for years exactly because they work.

But most blockchain ecosystems were not built with Java developers in mind. Instead, Rust, JavaScript, and Go dominate. Cardano – my blockchain of choice – historically leaned on Haskell.

At the end of the day, if you wanted to build on blockchain as a Java developer, you were largely on your own.

[BloxBean](#) was built to fix this.

No matter if you prefer Spring Boot, any other Java framework, or actually default to Maven, Gradle, and writing clean Java services, you already have everything you need to start building on Cardano.

Blockchain: What Java Developers Need to Know

If you have spent your career building web applications, REST APIs, and microservices, blockchain might feel like a concept that belongs to a different world. It does not. At its core, a blockchain is a distributed, append-only log where entries are cryptographically linked and verified by a network of participants rather than a central authority. Think of it as a shared system of record where no single entity controls the data, and every write is permanently auditable.

For Java developers, the relevant mental model is straightforward: instead of writing to a PostgreSQL database controlled by your organization, you submit transactions to a decentralized network that validates, orders, and permanently records them. The data becomes immutable: once written, it cannot be altered or deleted.

This property makes blockchain particularly valuable for use cases where trust, transparency, and auditability matter. Top applications include financial records, supply chain provenance, identity verification, and governance.

Key Concepts

The key concepts that differ from traditional development are few but important:

- **Transactions** are not database operations. They are cryptographically signed instructions that transfer value or record data on the network.
- **Wallets** are not accounts. They are key pairs. A private key signs transactions, and a public key derives an address. You do not register anything; you generate it.
- **UTXOs** (Unspent Transaction Outputs) track value differently than traditional account balances. Rather than keeping a single number in a table, some blockchains manage individual “chunks” of value like cash in a physical wallet. When you “send a transaction”, you consume an entire UTXO and receive any leftover amount back as change, similar to buying a \$3 coffee with a \$10 bill.
- **Transaction finality** is the point at which a transaction becomes irreversible on the network, unlike a database commit that an admin could roll back.

BloxBean lets you learn these concepts and see them in action using familiar Java tools instead of learning a new language while also trying to understand blockchain.

Why Cardano?

Among the many blockchain networks available, Cardano stands out for several reasons that matter to developers.

Cardano offers deterministic transaction fees based on transaction size and script execution cost, not network congestion. This means that, unlike most other blockchains, you can predict the fee before submitting a transaction.

Cardano uses the EUTXO (Extended UTXO) model, which builds on the basic UTXO concept by allowing each output to carry additional data and logic. This lets applications design for parallel processing off-chain, even though transactions are still processed sequentially on-chain. Unlike simple UTXOs, which only track value, EUTXOs can store smart contract data, enabling more complex interactions while preserving predictability and security.

The platform is MiCA compliant and CCRI-certified for energy efficiency. These are the kind of engineering guarantees that enterprise teams look for when evaluating infrastructure.

The ecosystem is steered by multiple independent organizations, including the Cardano Foundation, Input Output, EMURGO, and Intersect, with a strong enterprise focus spanning through finance, supply chain, digital identity, real-world asset tokenization, and much more.

That enterprise ambition requires enterprise-grade developer tooling. And unlike many blockchain ecosystems that lock developers into a language, [Cardano is designed to be language-agnostic](#) — the Cardano community builds tools in Haskell, Python, JavaScript, Rust, and more. For Java developers, the gateway into this ecosystem is **BloxBean**.

What Is BloxBean?

BloxBean is a GitHub organization hosting a collection of open-source Java tools and libraries, each solving a specific piece of the Cardano development puzzle. Rather than one monolithic SDK, it's a set of focused projects that work well independently and even better together; all available as standard Maven or Gradle dependencies.

The projects are community-driven, actively maintained by the Cardano Foundation and the community, and built with one clear philosophy: Java developers should not have to abandon their existing skills to work with blockchain technology.

If you know how to add a dependency to your pom.xml, write a @Test method, or call a REST endpoint, you already have what it takes.

However, even though BloxBean removes the tooling barrier, blockchain development does introduce new concepts. The UTXO transaction model, transaction finality, and on-chain versus off-chain logic are ideas that don't have direct equivalents in traditional web development. The difference is that with BloxBean, you learn these concepts while building with tools you already know, not while simultaneously fighting an unfamiliar language, build system, and test framework. The learning curve becomes one thing at a time instead of everything at once.

Key Tools / Libraries

These are some of the key tools or libraries:

TOOL	WHAT IT DOES
Cardano Client Lib	Build and submit transactions, create wallets, <u>mint</u> native tokens.
Yaci DevKit	Spin up a local blockchain for development with your own explorer.
Yaci	Connect directly to Cardano nodes via mini-protocols and receive blockchain data in callbacks.
Yaci Store	Index and query blockchain data via REST APIs; provides Spring Boot starters so you can build your own custom indexer in any Spring project and listen to blockchain events using Spring's @EventListener — just like any other Spring event.

Abb. 2: Each tool solves a specific part of the development workflow, and together they form a cohesive stack where **your existing Java knowledge is immediately transferable**.

Building Transactions with Cardano Client Lib

Cardano Client Lib is the core Java SDK for building on Cardano. If you've ever used a fluent builder API in Java, you'll feel right at home. It covers wallet creation, transaction building and signing, native token minting, smart contract interaction, and NFT metadata standards, all available as a standard Maven dependency.

Getting Started: Adding Dependencies

Adding Cardano Client Lib to your project is no different from adding any other Java library:

```
1 <dependency>
2   <groupId>com.bloxbean.cardano</groupId>
3   <artifactId>cardano-client-lib</artifactId>
4   <version>0.7.1</version>
5 </dependency>
```

No special plugins, no custom build steps, no native binaries to install. It is a standard Maven (or Gradle) dependency that works with Java 11 and above.

Creating Accounts and Building Transactions

Creating a wallet takes two lines — instantiate an Account with the target network (mainnet or testnet) and call `baseAddress()`. The class handles HD wallet derivation, mnemonic generation, and key management behind the scenes. You get a fully functional blockchain account without needing to understand elliptic curve cryptography or derivation paths, though those details are accessible if you want them.

```
1 Account account = new Account(Networks.testnet());
2 String baseAddress = account.baseAddress();
```

Where the library truly shines is the QuickTx API, which lets you specify the *intent* of a transaction declaratively. You describe what you want to happen: pay this address, attach this metadata, mint these tokens, and QuickTx handles the input selection, fee calculation, and transaction balancing to make sure the declared intent is fulfilled and the transaction is valid. All the complexity is wrapped behind a clean, easy-to-use Java API.

```
1 QuickTxBuilder quickTxBuilder = new QuickTxBuilder(backendService);
2
3 Tx tx = new Tx()
4     .payToAddress(receiverAddress, Amount.ada(1.5))
5     .payToAddress(anotherAddress, Amount.ada(3.0))
6     .from(senderAddress);
7
8 Result<String> result = quickTxBuilder
9     .compose(tx)
10    .withSigner(SignerProviders.signerFrom(senderAccount))
11    .completeAndWait();
```

Read that as a Java developer: Even without blockchain knowledge, the intent is clear. You are composing a transaction that pays ada – the native currency on Cardano, where the smallest unit is called a Lovelace – to specific addresses, detailing who sends it, signing it, and waiting for confirmation. The pattern is the same as building an HTTP request with WebClient or constructing a query with a JPA Criteria builder. The Result object wraps the outcome with success/failure status, so your error handling works the same way it does in any Java application.

Modular Architecture and Backend Providers

The library is organized into over 20 focused modules: core, crypto, address, plutus, Cardano Improvement Proposal (CIP) implementations, and more. You only pull in what you need, keeping your application lean.

It also supports multiple backend providers: Blockfrost (managed API), Koios (community-driven API), and Ogmios/Kupo (self-hosted). You choose your backend the same way you might choose between different database drivers: swap the provider, keep the same application code.

If you want to get hands-on straight away, the official docs have [a simple ada transfer tutorial](#) to get you started, and also [a multi-sig transfer tutorial](#) once you're ready to go further.

Your Local Blockchain in Seconds: Yaci DevKit

One of the real pain points with blockchain development is testing. Cardano has public test networks, and you can even submit real transactions to them, but public testnets aren't ideal for fast iteration. Block times are fixed, the network is shared with everyone else, and the immutable nature of blockchain means you can't just roll back a bad test. Waiting anywhere from seconds to minutes for a testnet confirmation on every small change is not how anyone wants to develop.

As Java developers we're used to running everything locally. Docker made that easy: spin up your database, your message broker, your cache, all in one command. Yaci DevKit does the same thing for Cardano. You get a fully private Cardano network running on your machine in seconds, and you can create or reset it with a single command. You can even dial the block time down to milliseconds for ultra-fast iteration.

```
1 yaci-cli> create-node -o --start
```

That's it. A private Cardano blockchain, made for you, and running locally.

The kit also comes with Yaci Viewer, a browser-based block explorer at localhost:5173. Think of it as your own personal blockchain explorer for your own devnet. You can see every block, every transaction, every UTXO in real time, the same way you'd browse Etherscan or Cardanoscan on mainnet, except this is your chain, with your test data.

And it ships with a built-in indexer that exposes all the REST API endpoints that Cardano Client Lib needs to build and submit transactions. So your local development setup is fully self-contained. No external API keys, no Blockfrost account, no public network dependency. Just point your BackendService at localhost:8080 and start coding.

Commands

The CLI mirrors tools you already know. Fund test addresses like seeding a test database, query UTXOs like inspecting rows in a table, reset the chain like truncating your test DB:

```
1 devnet:default> topup addr_test1... 50000 # fund a test address
2 devnet:default> utxos addr_test1...      # check UTXOs
3 devnet:default> tip                       # current chain tip
4 devnet:default> reset                     # wipe and start fresh
```

Distributions

Yaci DevKit comes in three flavours depending on your setup: a Docker Compose based distribution for local development, an NPM package for CI/CD pipelines, and a standalone ZIP. Whichever you choose, you go from zero to a running blockchain in a single command.

For full documentation and setup guides, visit devkit.yaci.xyz.

Under the Hood: Yaci Mini-Protocols

Most developers won't need to touch Yaci directly. But it's worth knowing it exists and what it does.

Cardano nodes communicate with each other using a set of low-level protocols called mini-protocols. These handle everything: synchronizing chain data, fetching blocks, submitting transactions, watching the mempool. Yaci is a pure Java implementation of those protocols. It's what sits underneath Yaci Store and Yaci DevKit and handles all the actual node communication.

If you're building something custom, like a chain indexer, a real-time monitoring tool, or your own block explorer, Yaci gives you direct access to the raw blockchain data stream without any of the network complexity. You don't write socket code, you don't handle protocol framing, you don't deal with connection management. You create a regular Java object, attach listeners, and start receiving data.

Listening to new blocks as they arrive is a few lines of code. Replaying historical blocks from a specific point is the same API. Watching the mempool for unconfirmed transactions in real time is just another listener. Everything that would normally require deep knowledge of Cardano's network layer is hidden behind a clean Java abstraction.

If you've worked with message consumers or event listeners in Java, the programming model will feel immediately familiar. The [getting started guide](#) covers the Fetcher API, Reactive API, and Local Client Provider with simple examples for each.

Indexing Blockchain Data: Yaci Store

Yaci Store is a chain indexer built on top of Yaci. It reads block data from the Cardano blockchain using those same mini-protocols and stores it in a standard relational database: Postgres, H2, or MySQL. From there, it exposes a REST API layer that tools like Cardano Client Lib use to query data, build transactions, and submit them to the network.

Out of the box, Yaci Store works as a standalone product. You download it, configure it like any other Spring Boot application through `application.properties` or `application.yml`, point it at a Cardano node, and it starts indexing. The Cardano node connection is just a few lines of configuration:

```
1 store:
2   cardano:
3     host: backbone.cardano.iog.io
4     port: 3001
5     protocol-magic: 764824073
```

The storage layer is built with DDD in mind. Each module in Yaci Store is called a store, and each one represents a specific domain of blockchain data: UTXO data, block data, transaction data, governance data, and so on. By default Yaci Store keeps full historical data, but pruning configurations are available if your application only needs recent state. If you don't need a particular type of data at all, you just disable that module:

```
1 store:
2   blocks:
3     enabled: true
4   transaction:
5     enabled: true
6   utxo:
7     enabled: true
8   metadata:
9     enabled: false
10  script:
11    enabled: false
```

Yaci Store exposes the Blockfrost-compatible API endpoints that transaction-building SDKs like Cardano Client Lib need, so you can develop locally without a Blockfrost account and switch to Blockfrost in production without changing your code. Develop against your locally running instance and point your app at Blockfrost's hosted service in production without changing a line of application code.

That's the product. But here's where it gets interesting for application developers:

Spring boot Starters

Each store module is also published as a separate library with its own Spring Boot starter. So instead of running Yaci Store as a standalone product, many developers pull in only the starters they need and build a narrow, application-scoped indexer directly inside their own Spring Boot app. Your application just becomes a standard Spring Boot app that listens to blockchain events using plain `@EventListener` annotations:

```
1 @Component
2 public class MyBlockchainListener {
3
4     @EventListener
5     public void onTransaction(TransactionEvent event) {
6         event.getTransactions().forEach(tx -> {
7             System.out.println("New tx: " + tx.getTxHash());
8             // your custom logic here
9         });
10    }
11
12    @EventListener
13    public void onBlock(BlockEvent event) {
14        System.out.println("Block #" + event.getMetadata().getBlock()
15            + " with " + event.getMetadata().getNoOfTx() + " transactions");
16    }
17 }
```


No blockchain-specific framework to learn; just Spring events. For more on this approach, see the [usage guide](#).

One more thing worth knowing: Yaci Store ships with a plugin framework backed by GraalVM's polyglot support. If you have team members who don't write Java, they can write plugins in JavaScript, Python, or MVEL to create filters and event listeners to process blockchain data. It's a practical feature for teams with mixed language backgrounds. There's a hands-on tutorial on [tracking address UTXOs with the plugin framework](#) if you want to see it in action.

Quick Start: Your First ada Transfer

Here's how the pieces fit together for your first ada transfer. This is the actual flow, not a simplified demo:

- Add cardano-client-lib to your pom.xml
- Install the Docker-based Yaci DevKit following the instructions at devkit.yaci.xyz, then start it:

```
1 devkit start
```

- Create a new local devnet with 1-second block times:

```
1 yaci-cli> create-node -o --start
```

- Create sender and receiver accounts with `new Account(Networks.testnet())`
- Fund the sender via DevKit CLI: `topup <senderAddress> 1000`
- Build and submit:

```
1 BackendService backendService =
2     new BFBBackendService("http://localhost:8080/api/v1/", "");
3
4 QuickTxBuilder quickTxBuilder = new QuickTxBuilder(backendService);
5
6 Tx tx = new Tx()
7     .payToAddress(receiver.baseAddress(), Amount.ada(5))
8     .from(sender.baseAddress());
9
10 Result<String> result = quickTxBuilder
11     .compose(tx)
12     .withSigner(SignerProviders.signerFrom(sender))
13     .completeAndWait();
```

- Open `localhost:5173` to see your transaction confirmed in the block explorer

That's it: From zero to a confirmed on-chain transaction in under 20 lines of Java and one Docker command. No waiting for testnet, no external API keys needed, and all in your preferred language.

The Broader Java Ecosystem on Cardano

BloxBean provides the core development stack, but it is worth noting that the Java ecosystem on Cardano extends beyond it. The community and the [Cardano Foundation](#) maintain additional libraries, including the [Koios Java Client](#) for REST-based node interaction, the [Ogmios Java Client](#) for WebSocket communication via Ouroboros mini-protocols, and the [Cardano Foundation Java Rewards Calculation library](#) for computing staking rewards among others. These complement BloxBean's stack and signal a broader, ecosystem-wide commitment to making Java a first-class citizen on Cardano.

Java in Production on Cardano

The BloxBean toolchain is already powering production-grade, Java-written applications on the Cardano blockchain. Here are some notable examples:

Reeve — Financial Data Integrity

Reeve is a platform developed by the Cardano Foundation that connects Enterprise Resource Planning (ERP) systems with the Cardano blockchain to create immutable, auditable financial records. Built entirely in Java using Spring Boot with a Spring Modulith architecture, Reeve allows organizations to record accounting transactions on-chain, creating a tamper-proof trail of financial activities.

The platform is designed with modular Java components — a reporting core, blockchain publisher and reader services, and ERP adapters — all operating independently within a Spring dependency injection context. Use cases include NGOs seeking to bolster donor transparency, publicly listed corporations managing ESG disclosures, and governmental bodies modernizing procurement audits. Reeve is open source under the Apache 2.0 license.

OriginateNavio — Supply Chain Verification

OriginateNavio is a blockchain-based solution that helps enterprises prove product origin and quality. It digitizes supply chain and certification data at the source using unique QR codes, uploads it for secure tracking with unique identifiers, and permanently records verified information on the Cardano blockchain. The result is instant, trustworthy verification of product provenance — from raw materials to finished goods.

USDM by Moneta — Fiat-Backed Stablecoin

USDM is a Cardano fiat-backed stablecoin issued by Moneta Digital LLC (FinCEN-registered MSB). Each USDM token is backed 1:1 by USD reserves, making it a foundational building block for decentralized finance (DeFi) on Cardano. USDM enables instant on-chain payments, serves as a base asset for DeFi protocols, and provides a stable store of value within the Cardano ecosystem. Parts of its infrastructure leverage the Cardano Client Lib to simplify interactions with the Cardano blockchain.

These are production platforms developed by established organizations and regulated financial entities to solve everyday problems in supply chain, finance, and identity. Several are built in Java, using Spring Boot, Maven, and the same architectural patterns you use in your day job.

If you are a Java developer who has been curious about blockchain, BloxBean gives you the most practical place to get started. Your existing knowledge becomes your blockchain foundation — the very building blocks you use from day one.

- **BloxBean:** <https://www.bloxbean.com>
- **Cardano Client Lib:** [GitHub](#)
- **Yaci DevKit:** [GitHub](#) | [Documentation](#)
- **Yaci:** [GitHub](#)
- **Yaci Store:** [GitHub](#)
- **Koios Java Client:** [GitHub](#)
- **Ogmios Java Client:** [GitHub](#)
- **CF Java Rewards Calculation:** [GitHub](#)
- **Reeve:** [Cardano Foundation](#) | [GitHub](#)
- **OriginateNavio:** [Cardano Foundation](#)
- **USDM by Moneta:** <https://moneta.global>
- **Cardano Developer's portal:** <https://developers.cardano.org/>

› **Back to the Table of Contents** ‹



Connecting Java Reinforcement Learning to Python Gymnasium

API & Frameworks

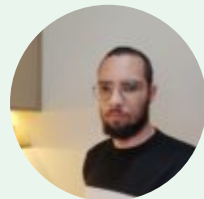
AI

MachineLearning

ReinforcementLearning

Author: Guilherme Alves Silveira

He documents his journey in Artificial Intelligence through technical articles on Computer Vision, Robotics, and Games and Simulation. Guilherme Alves Silveira learns by writing and shares so others can learn too, trying to respect technical rigor as much as possible without giving up creativity. He uses techniques like Memory Palace and Anki to retain complex concepts. He works primarily with Java and Python, exploring creative solutions to technical problems and constantly seeking new challenges. Martial arts are part of his lifestyle.



LinkedIn

This three-part series builds a Java Reinforcement Learning Python Gymnasium pipeline from scratch. Part 1, this article, explains why the project `java-rl-dqn-to-rainbow`[1] starts with Gymnasium integration. The main reason is testability. CARLA[2] is a very large and complex environment, so the algorithm needs simple environments such as CartPole[3] or Mountain Car[4] first, allowing shorter testing cycles. The implementation starts with DQN[5], one of the simplest algorithms combining Reinforcement Learning and Deep Neural Networks. Each subsequent step refines and upgrades the algorithm until the second article delivers the complete Rainbow DQN implementation. The last article integrates CARLA and trains the agent with Rainbow DQN[6]. The full series can be found online at www.javapro.io

What's Reinforcement Learning? A Simple Introduction

Reinforcement learning algorithms require environments to interact with. In practical terms, an environment in Gymnasium is a deterministic or stochastic simulator. It exposes a state, accepts an action, and returns a reward and the next state, as shown in the diagram below:

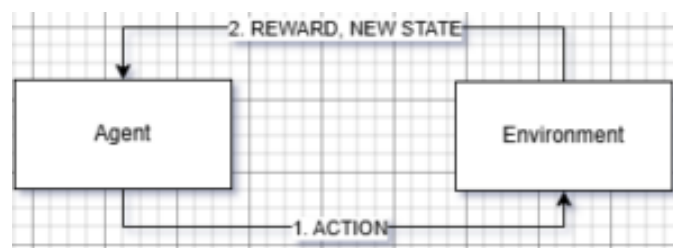
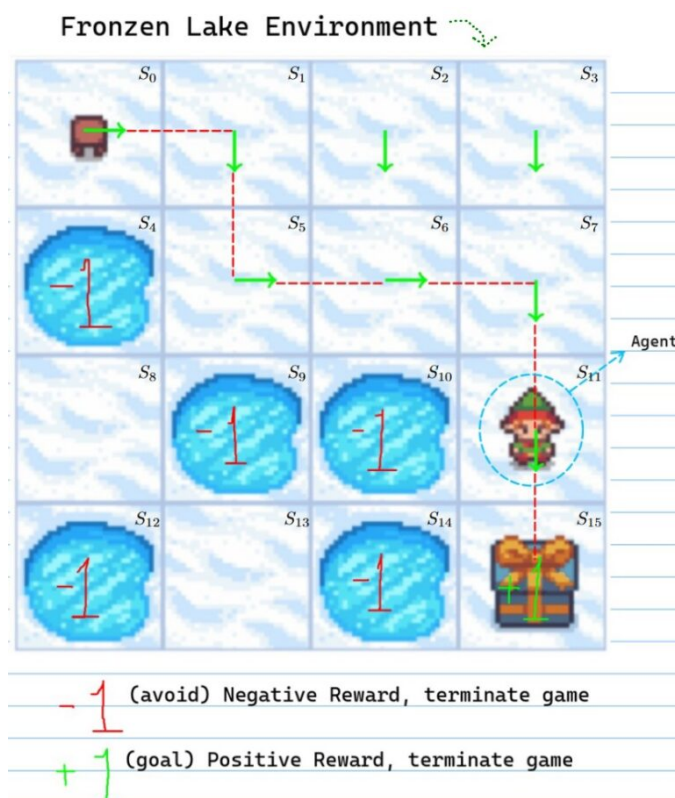


Abb. 3: Environment and Agent

A 4x4 grid, where each state marked as Sindex starting with 0 and finishing at 15, has holes with a **-1** reward (negative) and the present has a **+1** reward (positive). In the original implementation, the rules just terminate the game. The image only illustrates the concept.



Originally published: This article first appeared on Substack[7] as part of a three-article series on building a Java-based Rainbow DQN agent that connects Python Gymnasium and CARLA.

Gymnasium in the Java Reinforcement Learning Ecosystem

The Java ecosystem lacks native training environments for reinforcement learning. Python fills this gap with Isaac Lab[8] (Robotics mainly), the now-unmaintained OpenAI Gym[9] and the fork Gymnasium[10], which is an ecosystem on its own with many environments under the Farama[11] umbrella.

Example of the two environments that are covered in Video Games and Reinforcement Learning[12] and How to Never Forget Deep Q-Networks: Memory Palaces Meet Reinforcement Learning[13] articles:

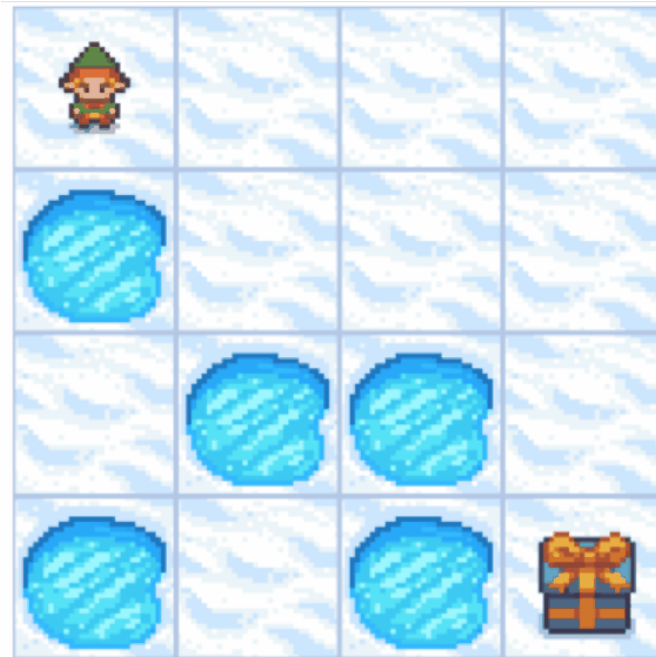


Abb. 4: Gymnasium Frozen-Lake environment

Trained agent running in Frozen-Lake environment



Abb. 5: Gymnasium Pong environment

Trained agent running in Pong environment

Evaluating Java Reinforcement Learning Integration Options

Two integration candidates stood out: Py4J[14] (used in PySpark[15]) and ZeroMQ[16]. ZeroMQ proved to be the better choice, as it is a brokerless messaging library with zero-copy optimizations. Py4J, by contrast, uses a gateway (broker) that translates calls between the languages.

The initial trial started a Python process from Java and used the Request and Response pattern for communication. Synchronizing the two platforms proved problematic; any ordering issue left the socket in an invalid state.

For example, sending two consecutive Java requests breaks the socket. The same occurs if Python sends a response twice. The following order must always be respected: req→resp→req ... and so on[17].

Even restarting the socket requires restarting the server or client as well. The project then switched the protocol to the Pirate Patterns to make the communication reliable[18].

Even so, the goal was a seamless integration between Java and Python. The evaluation covered two options: the FFM API[19] (Foreign Function and Memory API) and JavaCPP[20]. Both offer efficient integration with native code.

I chose JavaCPP for two reasons: it includes CPython[21] integration and works easily with C++ code[22], useful for future CARLA integration. The FFM API requires creating a C-API first, then generating bindings with jextract[23] for any library not written in pure C.

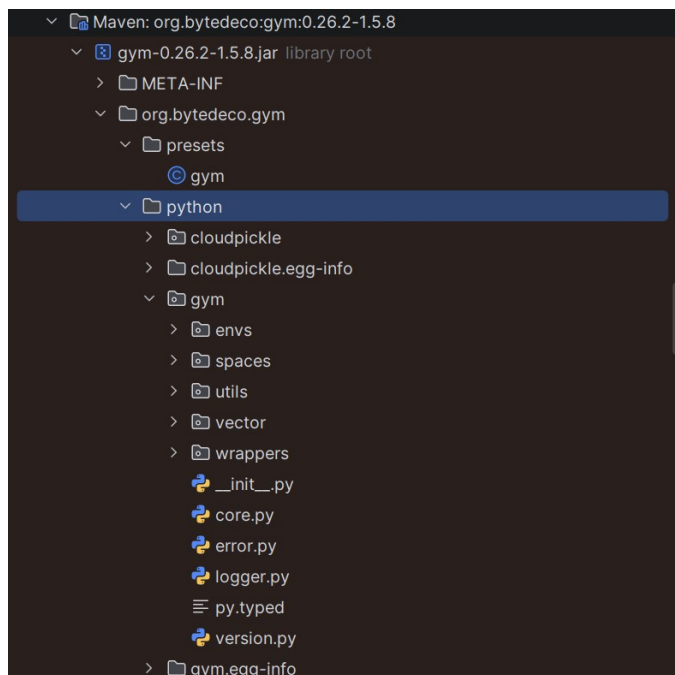
JavaCPP

JavaCPP calls native Python functions by implementing most of the CPython C-API[24], with some macro exceptions due to parser limitations[25].

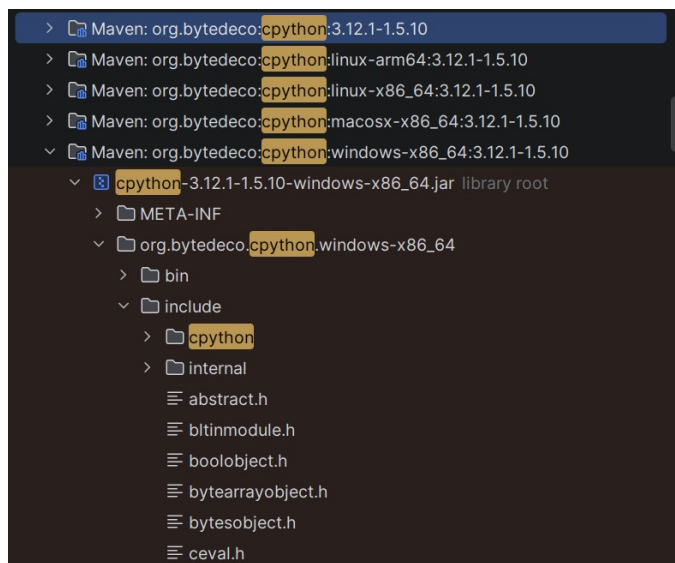
The javacpp-presets[26] already includes the Gym 0.26.2[27] library and ALE 0.8.0[28] (Arcade Learning Environment), and the gym module worked successfully. Inside the library, each platform bundles an embedded Python version within the jar to execute the gym code. Example:

```
1 // omitted code
2 public class gym {
3     private static File packageFile;
4     public static synchronized File cachePackage() throws IOException {
5         if (packageFile != null) {
6             return packageFile;
7         } else {
8             // This is the location of the jar that will be loaded
9             packageFile = Loader.cacheResource("/org/bytedeco/gym/python/");
10            return packageFile;
11        }
12    }
13 // omitted code
```

The physical location[29]:



Similarly, this can be observed with CPython[30], but separated by platform:



Setting Up the Python Environment

To control the version and libraries, the project creates a gymnasium folder in the root with the following content:

```
1 README.md
2 requirements.txt
```

README.md contains the instructions to install uv[31] and the correct Python version, along with the environment variables that Java exports for its use:

```

1 // omitted
2 pip install uv
3 uv python install cpython-3.12.1-windows-x86_64-none
4 uv venv --python 3.12.1
5 .venv/Scripts/activate
6 uv pip install -r requirements.txt
7 // omitted
8
9 - `export JAVA_RL_SITE_PACKAGES=/path/to/java-rl-dqn-to-rainbow/gymnasium/.venv/include/
   site/python3.12`

```

The requirements.txt contains the libraries to be installed:

```

1 numpy==2.3.5
2 pygame==2.6.1
3 matplotlib==3.10.8
4 gymnasium[classic-control,box2d]==1.2.2

```

Java Reinforcement Learning Python Gymnasium: Integration Roadmap

With the integration options evaluated, the implementation planning can proceed:

- Python starts as a fully managed embedded process.
- Code executes inside this process and the output returns to Java.
- Java creates the object inside Python and invokes its methods.
- Java reads Python arrays with zero-copy or memory copy depending on the scenario.
- `env.render()` displays the environment output in Java using JFrame.
- The environment executes actions using `env.step(action)` and returns the method output.
- The `step` method output converts to DJL[32]
- The project creates multiple tests for different environments to validate usefulness.

During development, many segfaults occurred, as the lifecycle of Python requires manual management. For an alternative approach to Java and Python integration, see [Bridging Java and Python for AI/ML in Production](#).

Embed Python Lifecycle

Calling `Py_Initialize()` and `Py_FinalizeEx()` more than once does not guarantee correct behavior due to external modules, as is the case of this project.

That is why the project avoids `Py_FinalizeEx()`, as Gymnasium depends mainly on NumPy and PyGame[33]; calling it can trigger a segmentation fault.

To work safely with Python in a multi-thread environment where Java has its own thread state, the integration requires `PyGILState_Ensure()` and `PyGILState_Release()`, e.g.:

```

1 // insideGil() wraps this block, accepting a Runnable or Function
2 var gstate = PyGILState_Ensure();
3 try {
4     result = callSomeFunction();
5 } finally {
6     PyGILState_Release(gstate);
7 }

```

Without this, race conditions can leave objects in an invalid state and cause a segmentation fault[34], crashing the JVM. For example, multiple threads decrementing the reference count variable simultaneously can produce a reference count of 0 instead of 1, causing a double-free[35].

However, if only the thread that called `Py_Initialize()` is used, in most cases the code block above is not needed. The method `insideGil()` encapsulates this block to avoid the boilerplate.

To understand why this matters, it helps to know about the GIL (Global Interpreter Lock), a Python mechanism that controls thread access to the interpreter. `Py_Initialize()` shares it across all threads by default, which is a limiting factor.

NumPy mitigates this by releasing the GIL during intensive C operations, which is why it remains fast.

To circumvent this limitation and better utilize the machine's multiprocessing, the project can adopt the vectorization[36] feature from the Gymnasium library, which runs environments in parallel and returns multiple values at once. If necessary, Gymnasium4j will include these features in the future.

Multiple Python Interpreters

An alternative is `Py_NewInterpreterFromConfig()`, which allows its own GIL per interpreter, introduced in Python 3.12[37].

Another useful method is `PyGILState_Check()`, which returns 1 if the thread holds the GIL; useful for debugging the application[38].

Memory management[39] outside Python also requires attention. Since the project embeds Python in Java, it must manage reference counting manually. Below is a list of the available methods:

Returns a new reference (you own it; you must decrement it):

- `PyObject_CallObject()`
- `PyObject_Call()`
- `PyObject_GetAttrString()`
- `PyImport_AddModule()`
 - The project usually creates and forgets the new module, as the Python process releases it on finish. This is the case here, used in `initPython()`, explained later.
- `PyUnicode_FromString()`
- `PyLong_FromLong()`
- `PyFloat_FromDouble()`
- `PyBool_FromLong()`
- `PyBytes_FromStringAndSize()`
- `PyByteArray_FromStringAndSize()`
- `PyDict_New()`
- `PyTuple_New()`
- `PyList_New()`
- `PyObject_Str()`
- `PyUnicode_AsUTF8String()`
- `PyDict_Items()`

Returns a borrowed reference (you don't own it, do **not** decrement reference):

- `PyTuple_GetItem()`
- `PyList_GetItem()`
- `PyDict_GetItem()`
- `PyModule_GetDict()`
- `PyErr_Occurred()`

Steals a reference (takes ownership from you):

- `PyTuple_SetItem()`
- `PyList_SetItem()`

PythonRuntime: Java Reinforcement Learning Python Bridge

Proceeding to the PythonRuntime, the main methods are `initPython()`, `exec()`, and `eval()`:

The `initPython()` method is called only once when the Java process starts. It is synchronized to ensure the code is initialized only once. `cachePackages()` returns the location of the javacpp CPython and custom libraries installed using [uv](#) (the environment/property `JAVA_RL_SITE_PACKAGES` must be filled).

The `globals` variable holds all variables created in Python by the integration:

```

1  @SneakyThrows
2  public static synchronized void initPython() {
3      if (initialized) return;
4      if (Boolean.getBoolean("python.initialized")) {
5          initialized = true;
6          return;
7      }
8      initialized = Py_Initialize(cachePackages());
9      globals = PyModule_GetDict(PyImport_AddModule("__main__"));
10     if (!initialized) {
11         throw new IllegalArgumentException("PythonRuntime is not initialized!");
12     }
13     System.setProperty("python.initialized", "true");
14 }

```

The `exec()` method is where Python functions are executed in the interpreter:

```

1  public static void exec(String code) {
2      PyErr_Clear();
3      try (var _ = PyRun_StringFlags(
4          code,
5          Py_file_input,
6          globals,
7          globals,
8          null
9      )) {
10         checkError();
11     }
12 }

```

Every variable is saved inside `globals`. For isolated executions, where variables do not live outside the scope, `execIsolated()` can be used as an alternative. Below an example of its usage:

```

1  // Code inside unit test
2  exec("""
3      import numpy as np
4      arr_little = np.array([1, 2, 3], dtype='<f4') # explicit Little-endian
5      arr_big = np.array([1, 2, 3], dtype='>f4')    # explicit Big-endian
6      """);

```

The `eval()` is very similar to `exec()`, the difference lies in the input parameter:

`Py_eval_input` expects an expression like `(1 + 1)` or `NewInstanceExample()`, while `Py_file_input` executes any code without returning a value:

```

1  public static PyObject eval(String expression) {
2      PyErr_Clear();
3      try (var result = PyRun_StringFlags(
4          expression,
5          Py_eval_input,
6          globals,
7          globals,
8          null
9      )) {
10         checkError();
11         return result;
12     }
13 }

```


The `PyErr_Clear()` clears any non-fatal error that was not cleared and could interfere with `PyErr_Occurred()`. The `checkError()` method checks if any error occurred after execution, using `PyErr_Occurred()`. The pattern is to clear possible old errors and check again at the end.

The other utility classes are `PythonDataStructures` and `PythonTypeChecks`, explained as needed, but the names are intuitive.

Java Reinforcement Learning Python Gymnasium: NumPy Integration

The `NumPyByteBuffer` class is used to make an efficient transfer between the `ndarray` data structure in Python and Java's `ByteBuffer` (or other languages that support the Buffer Protocol).

NumPy arrays implement the Buffer Protocol[40], This allows zero-copy communication because their memory is contiguous (a single block, like a primitive array)[41].

The static initializer is responsible for ensuring endianness alignment (the way bits are ordered in the hardware)[42] between Java and NumPy. It is implemented this way to perform the check only once:

```

1  static {
2      initPython();
3      BYTE_ORDER = insideGil(() -> {
4          exec("import numpy as np");
5          exec("_test_arr = np.array([1], dtype=np.float32)");
6          try (var testArr = eval("_test_arr")) {
7              String byteOrder = attrStr(attr(testArr, "dtype"), "byteorder");
8              return switch (byteOrder) {
9                  case ">" -> ByteOrder.BIG_ENDIAN;
10                 case "<" -> ByteOrder.LITTLE_ENDIAN;
11                 default -> ByteOrder.nativeOrder();
12             };
13         } finally {
14             exec("del _test_arr");
15         }
16     });
17 }

```

The `fillFromNumpy()` method handles this communication by making a copy of the Python `ndarray`. If receives a native array, it reuses it, avoiding array copying. However, since no efficient cycle management is yet implemented in this project (such as an Object Pool pattern), a temporary copy is made, as incorrect deallocation can also cause segmentation faults.

```

1  public static void fillFromNumpy(PyObject ndarray, ByteBuffer buffer) {
2      buffer.clear();
3      try (var view = new NumPyBufferView(ndarray)) {
4          int size = view.capacity();
5          if (size > buffer.capacity()) {
6              throw new IllegalArgumentException(
7                  "Buffer too small: capacity=" + buffer.capacity() + ", required="
8                      + size
9              );
10         }
11         buffer.put(view.buffer());
12     }
13     buffer.flip();
14 }

```

The `NumPyBufferView` uses the native Python C-API to bridge this communication. `@Delegate` from Lombok[43] is used to expose the exact same public API of `ByteBuffer` without requiring a manual implementation, establishing a standard to use the Python object as closely as possible to a `ByteBuffer`.

`AutoCloseable` is implemented to ensure the view is deallocated by the caller using the correct C-API:

```

1  public class NumPyBufferView implements AutoCloseable {
2      private final Py_buffer view;
3      @Delegate
4      private final ByteBuffer buffer;
5      public NumPyBufferView(@NonNull PyObject ndarray) {
6          this.view = new Py_buffer();
7          int rc = PyObject_GetBuffer(ndarray, view, PyBUF_SIMPLE);
8          if (rc != 0) {
9              throw new IllegalStateException("PyObject_GetBuffer failed (array not
              contiguous?), return code: " + rc);
10         }
11         long size = view.len();
12         this.buffer = view.buf().capacity(size).asByteBuffer();
13     }
14     // Used internally!
15     ByteBuffer buffer() {
16         return buffer;
17     }
18     @Override
19     public void close() {
20         PyBuffer_Release(view);
21     }
22 }

```

The method `PyBuffer_Release()` deallocates only the view. The original `PyObject` must be deallocated separately.

When the `ByteBuffer` is modified, the changes are reflected in the Python object:

```

1  exec("import numpy as np; arr = np.array([1.0, 2.0, 3.0], dtype=np.float32)");
2  try (var pyArr = eval("arr"));
3      var view = new NumPyBufferView(pyArr)) {
4      var buffer = view.asFloatBuffer();
5      assertEquals(1.0f, buffer.get(0), 0.0001f);
6      assertEquals(2.0f, buffer.get(1), 0.0001f);
7      assertEquals(3.0f, buffer.get(2), 0.0001f);
8      buffer.put(0, 4f);
9      buffer.put(1, 5f);
10     buffer.put(2, 6f);
11     assertEquals(new double[]{4., 5., 6.}, toDoubleArray(pyArr), 0.0001);
12     IO.println(toStr(pyArr)); // Output: [4. 5. 6.]
13 }

```

Gym Class: Building Gymnasium Environments in Java

To manage the environments, the `Gym` class is provided, making reference to the `gymnasium` as `gym` common import standard, and the `Env` class, the actual result of invoking the `make()` method, inspired by the original Python method:

```

1  @Slf4j
2  public class Gym {
3      @SneakyThrows
4      public static IEnv make(String name, NDManager ndManager) { // omitted code }
5      public static final class EnvBuilder {
6          // omitted code
7          String generatePyEnvScript() {
8              var makeCall = generateMakeCall();
9              var importLibsPy = generateImportLibsPy();
10             if (wrappers.isEmpty()) {
11                 return ""
12                 %s
13                 env_%s = %s
14                 """.formatted(importLibsPy, varEnvCode, makeCall);
15             }
16             var importPy = generateImportFromPy();
17             var wrappedEnvPy = generateWrappedEnvPy();
18             return ""
19             %s
20             %s
21             env_%s = %s
22             %s
23             """.formatted(importLibsPy, importPy, varEnvCode, makeCall, wrappedEnvPy);
24         }
25         public EnvBuilder add(@NonNull IWrapper wrapper) { // omitted code }
26         public Env build() {
27             return new Env(varEnvCode, envName, generatePyEnvScript(), ndManager);
28         }
29         // other methods omitted
30     }
31     public static final class PyMap {
32         // omitted code
33         public String toPyDict() {
34             return params.entrySet()
35                 .stream()
36                 .map(entry -> "" + entry.getKey() + ": " + entry.getValue())
37                 .collect(Collectors.joining(", ", "{", "}"));
38         }
39         // omitted code
40     }
41 }

```

The `EnvBuilder` is responsible for customizing the gymnasium environment, such as using ALE and other libraries.

`Wrappers` can be applied to modify the environment's step and observation space, such as skipping frames, converting to grayscale, resizing and concatenating (also known as stacking) the states: `MaxAndSkipObservation`, `GrayscaleObservation`, `ResizeObservation` and `FrameStackObservation`[44], which are the classes passed to the `add` method from `EnvBuilder`.

The `generatePyEnvScript()` method from `EnvBuilder` is the bridge to build the Python code, returning the configuration as a `String`. `PyMap` is a parameter configuration dictionary, equivalent to a `HashMap` in Java. Below are some examples of usage presented in the unit tests using [JUnit 6](#):

```

1 // omitted code
2 class GymTest {
3     @Test
4     void shouldTestGeneratedWrappers() {
5         var envId = "CarRacing-v3";
6         var script = Gym.builder()
7             .envName(envId)
8             .importLib("ale_py")
9             .params(Gym.builderMap()
10                 .put("domain_randomize", true)
11                 .put("continuous", true))
12             .add(new DelayObservation(1),
13                 new GrayscaleObservation(false),
14                 new NormalizeObservation(),
15                 new MaxAndSkipObservation(4),
16                 new FrameStackObservation(4),
17                 new ReshapeObservation(new int[] {1, 84, 84}),
18                 new ResizeObservation(new int[] {50, 50, 1}))
19             .generatePyEnvScript();
20         // omitted code
21     }
22     @Nested
23     @DisplayName("PyMap Tests")
24     class PyMapTest {
25         // omitted code
26         @Test
27         void shouldConvertToPyDictFormat() {
28             var pyMap = Gym.builderMap().put("seed", 42).put("render_fps", 60);
29             assertEquals("{\"seed\": 42, 'render_fps': 60}", pyMap.toPyDict());
30         }
31         // omitted code
32     }
33 }

```

Possible output of `generatePyEnvScript()`:

```

1 import gymnasium as gym, ale_py
2 from gymnasium.wrappers import DelayObservation, GrayscaleObservation,
   NormalizeObservation, MaxAndSkipObservation, FrameStackObservation,
   ReshapeObservation, ResizeObservation
3 env_2931ac287e0f4fdfae8a4ed7b75347dc = gym.make('CarRacing-v3', render_mode='rgb_array',
   domain_randomize=True, continuous=True)
4 env_2931ac287e0f4fdfae8a4ed7b75347dc = DelayObservation(
   env_2931ac287e0f4fdfae8a4ed7b75347dc, delay=1)
5 # omitted code

```

Env Class: Managing Gymnasium in Java

The `render_mode='rgb_array'` is a default value, as the main purpose is to display the environment visually when `env.render()` is called.

Below the main methods of the `IEnv` interface, **implemented** by the `Env` class:

```

1  public interface IEnv extends AutoCloseable {
2      boolean closed();
3      boolean scalarObservation();
4      ActionSpaceType actionSpaceType();
5      String actionSpaceStr();
6      String observationSpaceStr();
7      ActionSpaceType.ActionResult actionSpaceSample();
8      Pair<NDArray, Map<Object, Object>> reset();
9      EnvStepResult step(ActionSpaceType.ActionResult action);
10     EnvStepResult step(ActionSpaceType.ActionResult action, NDManager manager);
11     BufferedImage render();
12     NDManager manager();
13     @Override
14     void close();
15 }

```

The `Env` constructor contains the native `PyObject` wrapper, used to call methods in Python. `NDManager` is used in DJL to create the architecture, tensors and numerous Deep Learning utilities. `manager.newSubManager()` is called to ensure that when the environment is closed on the Java side, the objects allocated inside DJL are deallocated.

`ActionSpaceType` is a type-safe wrapper of Python's `ActionSpace`, providing an encapsulated execution of the main environments. `varEnvCode` is a mechanism to avoid overwriting Python's global variables (e.g.: `env_cf59c3da7e24499fa9f1d6860a534cd7`), as the Python interpreter uses the `globals`[45] for each process.

Another reason is that if `execIsolated` were used, the same variable would need to be recreated every time, requiring a scoped design that would differ significantly from the original API.

```

1  public final class Env implements IEnv {
2      // omitted code
3      private final NDManager manager;
4      @Getter private final String varEnvCode;
5      @Getter private final String envName;
6      private final PyObject pyEnv;
7      private final PyObject pyActionSpace;
8      private final PyObject pyObservationSpace;
9      private final PyObject pyRender;
10     private final PyObject pyStep;
11     private final PyObject pyReset;
12     private final ActionSpaceType actionSpaceType;
13     // omitted code
14     Env(@NonNull String varEnvCode, @NonNull String envName,
15         @NonNull String generatedScript, @NonNull NDManager manager) {
16         initPython();
17         this.varEnvCode = varEnvCode;
18         this.envName = envName;
19         this.manager = manager.newSubManager();
20         exec(generatedScript);
21         this.pyEnv = eval("env_" + varEnvCode);
22         this.pyActionSpace = attr(pyEnv, "action_space");
23         this.actionSpaceType = detectActionSpaceType(pyActionSpace);
24         this.pyObservationSpace = attr(pyEnv, "observation_space");
25         this.pyRender = attr(pyEnv, "render");
26         this.pyStep = attr(pyEnv, "step");
27         this.pyReset = attr(pyEnv, "reset");
28     }
29     // omitted code
30 }

```

The methods `eval` and `attr` are from the `PythonRuntime` class. `eval` evaluates and executes Python's interpreter, returning a `PyObject`, an object that contains a pointer to the native Python object, without passing through any gateway. `attr` follows the same idea, but for accessing attributes and methods, since in Python both are objects and can be retrieved by name from any `PyObject`.


```

1 // PythonRuntime.class
2 // omitted code
3 public static PyObject attr(PyObject obj, String attr) {
4     var result = PyObject_GetAttrString(obj, attr);
5     if (isPyNull(result)) {
6         PyErr_Print();
7         throw new IllegalArgumentException("Attribute not found: " + attr);
8     }
9     return result;
10 }
11 // omitted code

```

An equivalent example in Python, for the `Env`'s constructor, would be:

```

1 pyActionSpace = env_cf59c3da7e24499fa9f1d6860a534cd7.action_space
2 pyRender = env_cf59c3da7e24499fa9f1d6860a534cd7.render
3 # omitted code

```

`Env.render()`: Visualizing Gymnasium in Java

The `Env.render()` method is used to get the visual state of the environment, enabling the possibility to debug and verify each step taken as the state changes, from `s0` to `s1`, `s1` to `s2` and so on. The `PythonRuntime.callFunction()` executes the render method from the Python runtime's `env` variable. The `fillFromNumpy()` method transfers bytes to the native `ByteBuffer`, which as shown above, is created only once since it holds an image.

```

1 // Env.java
2 public BufferedImage render() {
3     try (var ndarray = callFunction(pyRender)) {
4         if (renderMetadata == null) {
5             renderMetadata = EnvRenderMetadata.fromNumpy(ndarray);
6             imageBuffer = ByteBuffer
7                 .allocateDirect(renderMetadata.size())
8                 .order(ByteOrder.nativeOrder());
9         }
10        fillFromNumpy(ndarray, imageBuffer);
11        return ImageFromByteBuffer.byteBufferToImage(
12            imageBuffer,
13            renderMetadata.width(),
14            renderMetadata.height(),
15            renderMetadata.channels() == 4
16        );
17    }
18 }

```

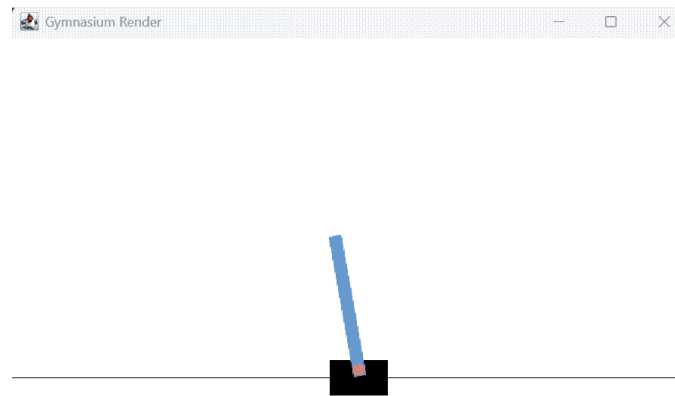


Abb. 6: CartPole execution inside Java, ignoring the `done()` method.

CartPole execution inside Java, ignoring the `done()` method.

`EnvRenderMetadata.fromNumpy()` handles the conversion of the NumPy array to `EnvRenderMetadata`, applying the appropriate treatment to produce the correct type and shape for DJL's `NDArray`.

```

1  public class EnvRenderMetadata extends EnvStateMetadata {
2      // omitted code
3      static EnvRenderMetadata fromNumpy(PyObject arr) {
4          var base = EnvStateMetadata.fromNumpy(arr);
5          return new EnvRenderMetadata(
6              base.shape, base.dtype,
7              base.djlShape, base.djlType,
8              base.size
9          );
10     }
11     // omitted code
12 }

```

Env.reset(): Starting a Gymnasium Episode

Returning to the `Env` class, `reset()` must be called before invoking `render()` and `step()`, and is required after each finished episode[46]. The code handles scalar and array observations, as each environment has its own rules, covering most cases.

```

1  // Env.java
2  @Override
3  public Pair<NDArray, Map<Object, Object>> reset() {
4      this.stateMetadata = null;
5      this.stateBuffer = null;
6      try (var result = callFunction(pyReset)) {
7          var pyState = getItem(result, 0);
8          var infoMap = getItemMap(result, 1);
9          if (!hasAttr(pyState, "shape")) {
10             this.scalarObservation = true;
11             long observationValue = toLong(pyState);
12             var state = manager.create(observationValue);
13             log.debug("Discrete observation: {}", observationValue);
14             return new Pair<>(state, infoMap);
15         }
16         this.scalarObservation = false;
17         this.stateMetadata = EnvStateMetadata.fromNumpy(pyState);
18         this.stateBuffer = onHeapBufferNumpy(stateMetadata.size());
19         fillFromNumpy(pyState, stateBuffer);

```

```

20         var state = manager.create(
21             stateBuffer,
22             stateMetadata.djlShape,
23             stateMetadata.djlType
24         );
25         return new Pair<>(state, infoMap);
26     }
27 }

```

Env.step(): Executing Actions in Gymnasium

The `Env.step()` is responsible for advancing the state (e.g.: `s1` to `s2`) and executing the action in the environment. Note that each value must be retrieved separately from the result using the `getItem*` methods, as the result is a Python tuple:

```

1  // Env.java
2  @Override
3  public EnvStepResult step(ActionResult action, NDManager manager) {
4      try (var result = callFunction(pyStep, action.pyObj)) {
5          NDArrary state;
6          if (scalarObservation) {
7              var pyState = getItem(result, 0);
8              long observationValue = toLong(pyState);
9              state = manager.create(observationValue);
10             log.debug("Discrete observation after step: {}", observationValue);
11         } else {
12             if (stateBuffer == null) {
13                 throw new IllegalStateException("You should call reset() first!");
14             }
15             fillFromNumpy(getItem(result, 0), stateBuffer);
16             state = manager.create(
17                 stateBuffer,
18                 stateMetadata.djlShape,
19                 stateMetadata.djlType
20             );
21         }
22         double reward = getItemDouble(result, 1);
23         boolean terminated = getItemBool(result, 2);
24         boolean truncated = getItemBool(result, 3);
25         var infoMap = getItemMap(result, 4);
26         return new EnvStepResult(reward, terminated, truncated, infoMap)
27             .state(state);
28     }
29 }

```

The following unit tests demonstrate that `Env` can handle multiple types of environments:

```

1  // GymActionSpaceTest.java
2  @Test
3  @DisplayName("MountainCar-v0 should have Discrete(3) action space")
4  void testMountainCarActionSpace() {
5      try (var env = Gym.make("MountainCar-v0", ndManager)) {
6          // omitted code
7          try (var action = env.actionSpaceSample()) {
8              assertEquals(DISCRETE, action.spaceType());
9              var result = env.step(action);
10             assertNotNull(result);
11             assertNotNull(result.state());
12             assertFalse(result.state().isReleased());
13             // omitted code
14         }
15     }
16 }
17

```

```

18     @Test
19     @DisplayName("LunarLanderContinuous-v3 should have Box(2,) action space")
20     void testLunarLanderContinuousActionSpace() {
21         try (var env = Gym.make("LunarLanderContinuous-v3", ndManager)) {
22             // omitted code
23             try (var action = env.actionSpaceSample()) {
24                 assertEquals(BOX, action.spaceType());
25                 var result = env.step(action);
26                 assertNotNull(result);
27                 assertNotNull(result.state());
28                 assertFalse(result.state().isReleased());
29                 // omitted code
30             }
31             // omitted code
32         }
33     }
34
35     @Test
36     @DisplayName("FrozenLake-v1 should have Discrete(4) action space")
37     void testFrozenLakeActionSpace() {
38         try (var env = Gym.make("FrozenLake-v1", ndManager)) {
39             // omitted code
40             try (var action = env.actionSpaceSample()) {
41                 assertEquals(DISCRETE, action.spaceType());
42                 var result = env.step(action);
43                 assertNotNull(result);
44                 assertNotNull(result.state());
45                 assertFalse(result.state().isReleased());
46                 // omitted code
47             }
48             // omitted code
49         }
50     }

```

With the Gymnasium integration established and validated, the Rainbow DQN algorithm will be implemented in the next article, using these simple environments as a stepping stone before training the agent in the CARLA environment. The complete Java Reinforcement Learning Python Gymnasium integration code is available at [java-rl-dqn-to-rainbow](#).

1. [Link](#) to the project.
2. CARLA is an open-source simulator to train autonomous driving agents: [link](#).
3. CartPole environment: [link](#).
4. Mountain Car environment: [link](#).
5. DQN is Deep Q-Network.
6. Improvements in the Deep Q-Network algorithm: [link](#).
7. Substack [link](#).
8. Isaac Lab: [link](#).
9. OpenAI Gym: [link](#).
10. Gymnasium: [link](#).
11. Umbrella Farama projects: [link](#).
12. Article [link](#).
13. Article [link](#).
14. Py4J [link](#).
15. PySpark [link](#).
16. ZeroMQ [link](#).
17. ZeroMQ is excellent, but you have to consider the trade-off between each scenario.
18. Reliability pattern for ZeroMQ that handles reconnection and message retries, found at: [link](#).
19. FFM API [link](#).
20. JavaCPP [link](#).
21. JavaCPP's CPython binding [link](#).
22. The FFM API tends to be faster because it doesn't have the overhead of JNI, usually not relevant, but depends on your needs.
23. jextract documentation [link](#).

24. CPython C-API documentation [link](#).
25. You can check the source code at [link](#) and limitations at [link](#).
26. javacpp-presets [link](#).
27. JavaCPP Gym binding [link](#).
28. JavaCPP ALE biding [link](#).
29. The gym library is platform independent, unlike CPython which is distributed as OS-dependent binaries.
30. CPython github [link](#), not JavaCPP's bindings.
31. The uv Python's package manager's [link](#).
32. DJL, acronym of Deep Java Library, a framework used in Deep Learning, with PyTorch, TensorFlow and JAX bindings, found at: [link](#).
33. [NumPy](#) is a numerical computing library and [PyGame](#) is for game development, both for Python.
34. [Segmentation fault](#): error raised when a program accesses invalid or unauthorized memory.
35. [Double free](#): error where a program frees the same allocated resource more than once, causing undefined behavior.
36. [Vector API](#): interface for running multiple independent environments in parallel and processing batched observations, actions, and rewards.
37. [CPython commit](#) introducing a per-interpreter GIL via `Py_NewInterpreterFromConfig()` as part of PEP 684.
38. More details at: [link](#).
39. The Python docs explain this in more detail: [link](#).
40. Buffer Protocol detailed explanation: [link](#).
41. Explanation of what's contiguous memory: [link](#).
42. Endianness explanation: [link](#).
43. Lombok can be found at: [link](#).
44. These wrappers can be found at the original documentation: [link](#).
45. Further evidence can be found at the links from official [documentation](#) and [real-python site](#).
46. An episode is the period from when the environment starts until a terminated or truncated state returns. After that, call `reset()` to restart from the beginning.

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

Vector Search with Pure Java

No External Database. Extremely fast. No Compromises.

NEW:

EclipseStore 4 - Java Embedded Vector Database

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

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

Project: www.eclipsestore.io

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

NEW:

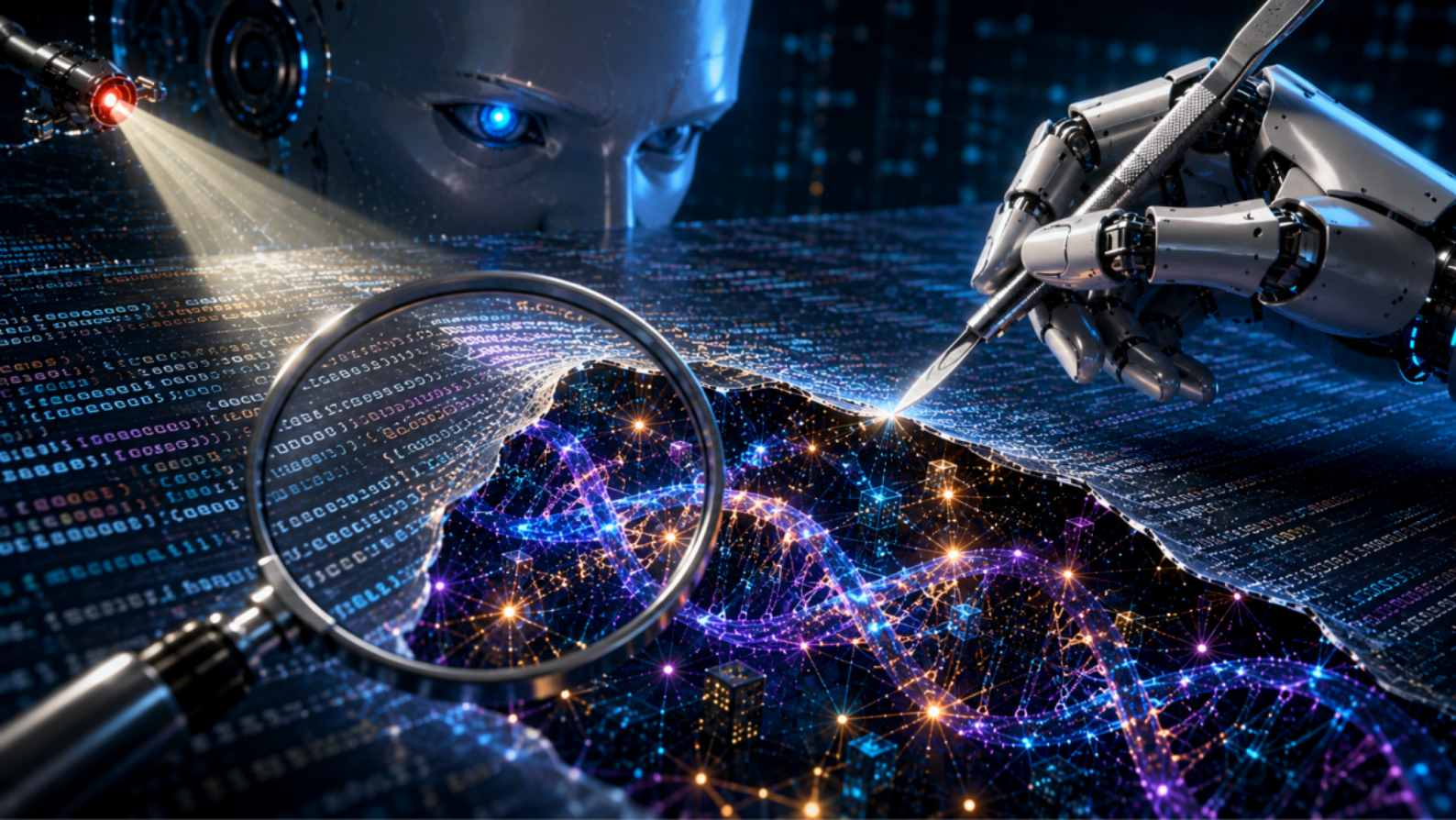
Eclipse Data Grid 4 - Distributed In-Memory Vector Search

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

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

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

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



Beyond Text Search: A Smarter Way to Understand Your Code

API & Frameworks

AST

OpenRewrite

Author: Bryan Friedman

Bryan Friedman is passionate about building and shipping software, educating users, and enabling customers for success. With a love for technology and problem solving, he has spent the past several years as a product leader with a focus around application platforms, cloud computing, and infrastructure automation, working in roles across Product Management, Technical Product Marketing, and Developer Relations. Now a Technical Marketing Lead at Moderne, he previously led the Customer Learning and Developer Engagement teams at VMware. Before that, he was a Product Manager and Director at CenturyLink Cloud, which was the culmination of over ten years prior to that working as a software developer and in a number of other various information technology areas.



LinkedIn

Every nontrivial change to a codebase starts the same way: someone asks a question about the code. The answer ultimately determines what gets read, what gets edited, and what eventually ships. For most of the history of programming, the tool that answered it was some flavor of text search. From `grep` to `find-in-files` to hosted code search platforms, it's all the same fundamental operation: where does this string appear?

That has worked well enough for a long time, but two pressures push against it. The first is shape: Java systems sprawl across dozens or hundreds of repositories, and the question “where is this used?” no longer has a single-repo answer. The second is who is asking. AI coding agents now read code constantly, and they read it by grepping for a string and opening every candidate file to figure out what it actually means. The cost of imprecise search used to be just an annoyance. Now, with agents doing the reading too, it's tokens, latency, and the potential for wrong answers.

The good news is that code search is not really just one capability. It's a spectrum, from raw text matching at one end to full type-aware analysis at the other, with several useful stops in between. Knowing what's at each point in the spectrum, and which questions belong where, is what makes it possible to work safely across a sprawling estate, with or without an agent in the loop.

Why Text Search Keeps Showing Up

Code search is something developers (and now the agents working alongside them) do dozens of times a day, often without thinking of it as search at all. Looking for the definition of a method, the call sites of a class, every place a configuration value appears, the spot where a particular log line gets emitted—text search handles a lot of these well enough.

What text-based tools have in common is that they are matching strings. `grep` matches literals. `Regex` bumps that up a level, allowing for character classes, alternation, and quantifiers, which are genuinely useful for things like “every method whose name starts with `find` and ends with `By` followed by something.” That's a real step up from string matching, but it's still pattern matching over text. `Regex` doesn't know what a method is. It doesn't know whether the match is in code or a comment or a string literal. For navigation and casual exploration, that's fine. The problems start when text matching is asked to stand in for understanding.

A few places text matching tends to break down, all of them variations on the same theme: text doesn't know what code is.

False positives (and false completeness). This is the failure mode most people learn to live with. `grep` happily returns everything that matches your spelling: imports, comments, string constants, unrelated identifiers that happen to collide. That may be okay in some scenarios where false positives aren't a problem. But for use cases that require precision, like impact analysis or security remediations, you can't always be sure you've found everything that matters, and you can't always tell what you've missed.

Overloads, inheritance, annotations. In Java, false positives appear in several different ways. Method names get reused, so a search for `process()` returns dozens of unrelated implementations across unrelated classes, while changing a base class can affect subclasses that never explicitly mention the modified method. Also, frameworks like Spring and Jakarta wire behavior through annotations and reflection, so a method annotated `@EventListener` may never appear in any explicit call chain at all. `grep` will find the annotation, but it cannot tell you what fires it, when, or with what payload.

Scale. `grep` reads every file line by line. At a few thousand lines that may be fine; at five billion lines across an enterprise's many repositories, it's not as reasonable. With agents running `grep` frequently and following up with file reads, the cost compounds and the time adds up.

Java's reliance on type attribution and runtime composition makes these failures more common than they look. The deeper the abstraction, the less the source text tells you about what the code actually does.

The Model Underneath: Lossless Semantic Trees

Type-aware search needs a richer model of the code. Most parsers produce an abstract syntax tree (AST), a structural representation of the source that captures the grammar but throws away things that aren't strictly part of it, like comments, formatting, and type information. [Tree-sitter](#), the parser library underlying a lot of modern editor and IDE tooling, is a good example. It's fast, incremental, and structurally accurate, but it's still an AST.

OpenRewrite's model is the [Lossless Semantic Tree \(LST\)](#), which powers type-aware search and transformation capabilities. It extends an AST with the things ASTs typically discard:

- Full type attribution across dependency graphs, including transitive types
- Symbol resolution and method binding
- Annotations, modifiers, and generics
- Formatting, whitespace, and comments preserved exactly

The formatting preservation is what allows automated changes to land in a codebase without erasing developer intent or producing messy pull requests that are difficult to review. The type attribution is part of what makes semantic search possible. Consider:

```
1 public interface PaymentProcessor {  
2     void process(Payment payment);  
3 }
```

A query against the LST can find every implementing class, every call site bound to this interface, and every overridden variant all while excluding any unrelated `process()` methods elsewhere. The query doesn't depend on strings at all; it depends on the type model the LST carries.

This model supports finding things that text search can't: every public method returning `ResponseEntity`, every method that declares `throws IOException`, every concrete implementation of a given interface. (If you're interested in what the LST actually contains and why it matters for transformation, you can find more details in the [OpenRewrite documentation](#) or [my earlier JavaPro piece on OpenRewrite and LSTs](#).)

To be fair, this capability isn't unfamiliar to developers. IDEs have been doing search like this for decades. Tools like IntelliJ, Eclipse, and VS Code all carry a model of the project that supports things like find usages, call hierarchy, go-to-definition, and find implementations. The catch is scope: the IDE's model lives inside the IDE, limited to one project at a time and one developer's local view. The LST is built to support multiple repositories. It can be serialized, indexed, and queried in bulk, enabling type-aware search across an entire estate. The model is also open and, as you'll see, the query layer (called a recipe) is programmable, so the questions you can ask aren't limited to a fixed set of operations the way IDE search is. Anyone can write a recipe to answer a new kind of question. And the same model can be projected onto more than one surface, each fit for a different shape or level of question.

Structural and Semantic Aren't the Same

Structural and semantic search sometimes get used interchangeably, but they shouldn't be. The distinction tells you what kinds of questions a particular search approach can actually answer.

Structural means the search understands the *grammar* of the language. It knows that `foo(bar)` is a method invocation, that `class Foo extends Bar` is a class declaration with a superclass, and that a parenthesized expression is one unit. Structural matching can find every method call whose first argument is a string literal, or every class declaration with at least three annotations. It does not know what those classes mean, or what types those expressions resolve to. It's operating on the shape of the source only, not its meaning.

Semantic means the search understands the resolved program, like the kind of information an LST carries. It knows that the `Foo` in one file is the same `Foo` as the one declared in another file. It knows that one particular method invocation binds to a specific overload of another method declared elsewhere. It can find every call site that invokes this specific method, including the ones text search would miss (like in the case where the static type is an interface and the binding only resolves through the type hierarchy).

Structural search can find every method named `save`. Semantic search can find every call site of `org.springframework.data.repository.CrudRepository#save`, distinct from `MyOwnService#save` and from a string literal that contains the word "save" in a comment. The first is a syntactic question, but the second requires the program's resolved type information. Structural search returns more matches, with more noise. Semantic search returns fewer, but each one can be trusted to be precise.

A Spectrum of Search, Not a Single Tool

The useful frame is not "text search vs. semantic search" but a spectrum of search functionality, each with a different cost, a different scope, and a different kind of certainty.

No method is necessarily “better” than any other. Each one has its own individual depth of understanding of what it’s able to reason about. They aren’t strictly substitutes; each is best at different kinds of questions, and some earn their keep through scale rather than depth. They can also build on each other in productive ways: a broader search that returns many false-positives can scope down what an exact search has to look at, so the more precise approach only runs against the candidates worth examining. Many hosted platforms already do this, like running structural matching on top of a trigram index, with the broad pass narrowing what the deeper pass has to parse.

1. Text and regex. Basic text matching, like `grep` does, is simple and ubiquitous, but with no awareness of code structure or meaning. It scales by brute force. If you point it at any number of files, it’ll dutifully scan them all. This can be fine locally, but it doesn’t scale to portfolios the way an indexed search does.

2. Trigram search. This is what hosted code search platforms like GitHub’s code search, Zoekt, and others bring to the table. [Trigram indexing](#) decomposes text into overlapping three-character sequences and intersects postings to find candidates fast. While trigram indexing is a decades-old information retrieval technique, [Russ Cox’s 2012 piece on Google Code Search](#) showed how to apply it to code search at scale and open-sourced a reference implementation that downstream projects built on, like the open-source [Zoekt](#) project. The trigram search method is still used by several commercial code search products because the technique scales well across thousands of repositories. The result is exactly what you need for portfolio-scale scoping: “does this pattern exist anywhere in the estate?” But it’s still text underneath, with all the same problems that text search can run into, like the potential for false positives.

3. Structural search. Beyond what regular expressions can do, structural search matches code by its parsed shape, typically working from an AST or an AST-like representation of the source. This kind of pattern matching respects the grammar of the language, with balanced parentheses, nested expressions, and statement boundaries. Tools like [Comby](#) let you write a query like `logger : [level] ( : [message] )` and have it match log calls regardless of argument shape, in a way regex can’t. It’s a step up in expressiveness, but it’s still operating on syntax rather than meaning.

4. IDE search. Integrated developer environments like IntelliJ, Eclipse, and VS Code, have parsed the code, built an index, and offer search operations like “find usages,” “go to definition,” and “find implementations.” It’s genuinely semantic, with a fully resolved model of the project’s code, bindings, overloads, and the type hierarchy all available. But it’s limited to one project per IDE session. These tools are also typically only accessible to a developer or a plugin running inside the IDE, and not necessarily readily available to an agent working in the repository.

5. Multi-repo semantic search. To precisely answer type-aware queries that span the whole estate, you need full semantic search capabilities across every repository. Getting information like every public method that returns `ResponseEntity`, every concrete implementation of `PaymentProcessor`, or every subclass of `AbstractEntity` across every service requires resolved types and a clear understanding of the symbols and relationships in the code. This is when search needs a richer model of the code than a mere syntax tree.

That fifth method is the one most teams don’t have. IDEs answer semantic questions inside a single project, but extending that across hundreds of repositories requires something most code search platforms don’t carry: a model of the code rich enough to resolve types across an entire estate. The LST is that model, and its existence is what makes type-aware search across a portfolio possible at all.

Between broad text search and deep semantic resolution, one useful intermediary has also emerged: a trigram index built from LSTs instead of text. Because the index also carries some additional data derived from the LST, queries can get at some deeper semantic information like inheritance, return types, and method visibility. Moderne calls that intermediary `Trigrep`.

Trigrep: Trigram Search, Made Smarter

[Moderne Trigrep](#) search uses a trigram index, but it’s built from LSTs rather than raw source. That changes what a “match” can be qualified by: alongside the usual literal and regex matching, the index carries enough information from the LST to understand symbols and distinguish things like a class declaration from a reference to one, a method definition from a call site, and code from a comment. The query syntax is intentionally similar to the dialects developers already know from GitHub search and Zoekt (literal matches, regex, symbol search, boolean composition) but with LST-backed, type-aware predicates available on top.

A few examples using the [Moderne CLI](#) on a working set of public Spring repositories:

```
1 # Literal match: finds every token that reads "RequestMapping"
2 mod search . RequestMapping
3
4 # Regex: finder method names
5 mod search . '/find\w+By\w+/'
6
7 # Symbol lookup: class declarations only
8 mod search . ClinicService type:symbol
```

Those queries can more or less be answered by any code search tool. The LST-backed predicates are what change the conversation:

```
1 # Every public method returning ResponseEntity, across the working set
2 mod search . visibility:public type:method returns:ResponseEntity
3
4 # Every method that declares throws IOException
5 mod search . throws:IOException type:method
6
7 # Every class that extends Person
8 mod search . extends:Person
```

The disambiguation case is the one that makes the difference concrete. `grep -rn "StringUtils"` returns hundreds of lines: imports, calls, comments, every reference of any kind. The same question, phrased semantically:

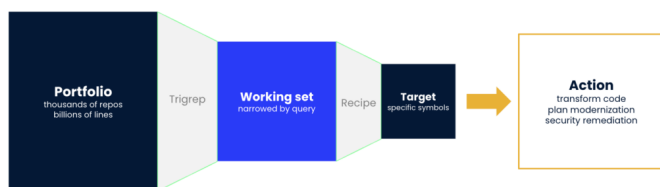
```
1 mod search . StringUtils type:symbol
```

This returns only the handful of actual `StringUtils` classes that exist in the working set, several of which happen to share a name. The index understands that “a class named `StringUtils`” is a different question from “the eleven characters `S-t-r-i-n-g-U-t-i-l-s` appearing somewhere in a file.”

Recipes: the Deep End of the Spectrum

The top of the spectrum is where full, multi-repo semantic search actually lives, and in OpenRewrite that’s a search recipe. An [OpenRewrite recipe](#) is a deterministic program that walks an LST to find or transform code. Recipes can be as simple as “find every usage of this method” or as complex as a full framework migration. They prioritize precision, reading every file, resolving every type, and producing results that are exact by construction. They’re also the same primitive that performs the transformations themselves—find and change use the same model.

Recipes are powerful, but they earn that power by reading every file and resolving every type. What’s interesting is exploring how to use them in combination with the rest of the spectrum. Trigrep and recipes are two different projections of the same LST model: one casts a broader net with some noise, the other returns exact results. Trigrep can scope a question across an estate down to the candidates worth looking at; the recipe then returns precise answers on that smaller set.



Each stage shrinks the scope. Action can be taken on smallest set.

Bridging Broad and Precise: Scoping a Recipe Run

A broad Trigrep search can scope what a more precise search has to look at. A Trigrep search for `RestController` is approximate since it matches imports and comments along with declarations. The OpenRewrite [FindAnnotations](#) recipe is heavier work, but it’s precise because it walks the LST and reports only actual `@RestController` usages.

Composing them is the productive move:

```

1 # 1. Quick pass: which repos even mention RestController?
2 mod search . RestController type:symbol
3
4 # 2. Precise pass: only on the repos that matched
5 mod run . --last-search \
6     --recipe=org.openrewrite.java.search.FindAnnotations \
7     -PannotationPattern='@org.springframework.web.bind.annotation.RestController'

```

The recipe runs on a smaller set than the full portfolio, using only the results from the Trigrep search (with the `--last-search` flag). The precise pass costs less because the broader pass eliminated everything that could not match.

Code Search for Agents

Most of what's been described so far is a developer workflow: a person scoping a portfolio with a broader query, and possibly drilling in with a precise one. The same methodology actually becomes even more valuable when it's an AI agent doing the searching.

When a developer greps and gets a noisy result, the cost is mostly just annoyance. They can scroll past the imports, ignore the comments, and click into the file that looks right. When an agent greps and gets a noisy result, the cost is real: every candidate file has to be opened and read in full to confirm whether it matters. Grep reads every line looking for a match, so it's not the most performant solution in the world, especially with massive codebases where the number of files it has to scan increases considerably. There are faster alternatives like ripgrep, but truthfully the search call itself isn't the biggest issue. It's the read loop after the search that burns tokens, runs into context limits, and slows the agent down.

The fix is to give the agent a search tool whose results are structured enough that the extra read loops are unnecessary. If the search returns "this is a method declaration named `process`, in `com.example.PaymentService`, with `public` visibility and a return type of `void`," the agent doesn't need to open the file to find out what it's looking at.

The same LST model serves both audiences. A developer searching across a portfolio uses the CLI or web interface; an agent working in a single repository calls the same index [via MCP](#). Side-by-side comparisons of agents working with and without this kind of search consistently show order-of-magnitude differences in tokens consumed and tool calls made. Trigrep search is faster than regular grep on average, but again, most of the savings aren't in the search calls themselves but rather in the file reads that structured results render unnecessary.

A Practical Search Workflow

A modernization pass through this model might look like:

1. **Trigram pass to scope.** Which repositories even reference the deprecated API? At portfolio scale this eliminates most of the work before any potentially expensive analysis runs.
2. **Structural or semantic pass to qualify.** Among the candidates, which usages match the patterns that need attention? This is where predicates like `visibility:public`, `returns:`, and `extends:` are most helpful.
3. **Recipe-driven analysis and/or transformation.** A deterministic OpenRewrite recipe narrows the search even more, or applies changes accurately, preserving formatting and semantics, on the repositories that actually need it.

Each step reduces uncertainty and concentrates the work where it matters. And it's a repeatable pattern: the same query against the same LSTs returns the same results, just as the same recipe applied twice produces the same output. So any of it can be re-run as the codebase evolves. For AI-assisted workflows, the structure is the same, only it's the agent orchestrating the steps.

Search Isn't Just a Feature Anymore

Programming has always been mostly an act of understanding what's already there well enough to add to it without breaking it. Search is how that understanding gets built, and it always has been. The tools we use to do it have always shaped what kinds of changes feel safe.

For a long time, `grep` was enough because the questions were small enough. A single repo, a single developer, a few hundred files. As estates grew, the questions grew with them, and platforms like GitHub's code search emerged to handle the multi-repo case. Now agents are doing the searching too, and they're asking more questions per task than any human ever did. The economics of imprecise search have changed.

The kinds of search described here aren't really new. Trigram search predates most of those in the industry. Structural matching has been around since the original AST-rewriting research. And type-aware search has existed in IDEs for decades. What's new is that they're all available together, across an entire estate, callable by humans and agents from the same surface, and grounded in a model rich enough to give precise answers when the question demands one. Text search will keep being useful for the questions text can answer. For everything else, having the rest of the spectrum available is the difference between guessing and knowing.

› **Back to the Table of Contents** ‹



The 3 A.M. Deadlock

API & Frameworks

Deadlock

Hibernate

JPA

Persistence

SpringBoot

Transactions

Author: Yasin Baytürk

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



LinkedIn

A Field Guide to Diagnosing and Fixing Real Production Deadlocks in Spring Boot + JPA

The pager goes off at 03:14. The nightly batch, the one that closes the day for a few thousand customers, has rolled back. Your phone shows three Slack messages already, the first one a single word: “again?”. By 03:18 you are at the kitchen table with a laptop, dependency tree in one tab, the previous incident’s post-mortem in another, and a half-formed suspicion that the fix you shipped two weeks ago did not actually fix anything.

This article is for the engineer at that kitchen table. It is a field guide to the deadlocks that actually occur in long-lived Spring Boot applications backed by JPA. The kind you do not see in textbook examples. The kind that hide for months and surface at the worst possible time. The patterns described here come from production incidents on real enterprise systems: nightly batch jobs, planning engines, integration adapters between Spring Boot and ERP backends.

If you have been doing this for more than a few years, you have probably met some of these. If you are newer, this might save you a few of those 03:14 mornings.

Why This Article Exists

The Java publishing landscape has plenty of material on transactions in the abstract: isolation levels, MVCC primers, “what your ORM does not tell you” tutorials. There is much less material on what to actually do when a relational deadlock has just rolled back a 47-minute batch and you have less than an hour to decide whether to retry, patch, or escalate.

There is one notable JAVAPRO piece on deadlocks: Adelina Simion’s “No Deadlocks in MongoDB: Atomic Documents and Retries with Spring Data MongoDB”. That article makes a clear, valid argument for the document-database side of the trade-off space. The piece you are reading is the relational counterpart. Spring Boot teams running PostgreSQL, MySQL/MariaDB, Oracle, or SQL Server still hit deadlocks regularly, and the diagnostic and architectural moves on the relational side are different enough to deserve their own treatment.

What a Deadlock Actually Is, Without the Hand-Waving

A relational deadlock occurs when two or more transactions hold locks that the others need, in a cycle the database cannot break by waiting. Modern databases detect this and pick a victim, but the codes they raise are not identical, and the distinctions matter when you are writing exception handlers:

- **PostgreSQL** raises `SQLSTATE 40P01 (deadlock_detected)` for deadlocks. `SQLSTATE 40001 (serialization_failure)` is a different condition. It surfaces under `SERIALIZABLE` or `REPEATABLE READ` isolation when the engine can no longer guarantee a serial schedule. Both are typically retryable, but they are not the same.
- **MySQL/InnoDB** raises error 1213 with `SQLSTATE 40001` for deadlocks. The victim transaction is rolled back automatically, and you can inspect the deadlock via `SHOW ENGINE INNODB STATUS`.
- **Oracle** raises `ORA-00060`. **SQL Server** raises error 1205 and exposes a deadlock graph in the `system_health` Extended Events session.

In a Spring JDBC or Spring Data JPA stack, these vendor codes will usually surface as a transient data access exception, most often `CannotAcquireLockException` or `PessimisticLockingFailureException`. Both are subclasses of `TransientDataAccessException`. The exact mapping depends on your driver, JPA provider, and the configured `SQLExceptionTranslator`, so the right move once in your codebase is to write a small integration test that triggers a real deadlock and confirms which exception type your stack actually produces. That is the type your retry logic should target.

A few things follow from this that matter for the rest of the article:

- A deadlock is not a bug in the database. The database is doing its job by detecting the cycle and rolling one transaction back. A deadlock is a signal that *your application’s locking pattern* contains a cycle.
- A deadlock is **transient**. The classification matters because it determines what your code is allowed to do automatically. A transient failure can be retried; a logical failure (constraint violation, business rule breach) cannot.
- The database tells you exactly which locks were involved if you ask it. PostgreSQL’s `pg_locks` view, MySQL’s `SHOW ENGINE INNODB STATUS`, and Oracle’s `V$LOCK` are the canonical sources. The thread dump tells you what your *application* was doing. Only the database tells you what the *transactions* were doing.

That last point is where most relational deadlock investigations go wrong. The reflex is to reach for `jstack` first, find a thread parked on a JDBC socket read, and conclude that the application is “stuck on the database”. That is true but useless. The thread dump shows the symptom; the database log and lock views show the cause.

The First 15 Minutes: A Triage Playbook

When the page comes in, you have a small number of decisions to make quickly. Make them in this order.

- 1. Confirm it is actually a deadlock, not a long lock-wait.** A timeout (`org.postgresql.util.PSQLException: ERROR: canceling statement due to lock timeout`) is *not* a deadlock. It means a transaction held a lock longer than your `lock_timeout` and you killed it. The fixes are different. Look at the SQLState in the exception. PostgreSQL 40P01, MySQL 1213 (SQLSTATE 40001), Oracle ORA-00060, and SQL Server 1205 are deadlocks. PostgreSQL 57014 (query cancellation) or a Spring `QueryTimeoutException` is a timeout. PostgreSQL 40001 on its own is a serialization failure: also retryable, but a different condition with a different fix.
- 2. Decide whether to retry or escalate.** If the rolled-back transaction is idempotent and the load is not pathological, a single automatic retry is usually safe and almost always correct. If the same transaction has now deadlocked three times in a row, retry is masking a real lock-ordering problem. Escalate.
- 3. Capture the evidence before it disappears.** PostgreSQL writes the deadlock detail to its log when `log_lock_waits` and `deadlock_timeout` are configured. MySQL keeps only the *latest* deadlock in `SHOW ENGINE INNODB STATUS`, which means the next deadlock will overwrite the one you need. Pull both immediately:

```
1 -- PostgreSQL: which queries are blocking which RIGHT NOW?
2 SELECT blocked.pid AS blocked_pid,
3        blocked.username AS blocked_user,
4        blocking.pid AS blocking_pid,
5        blocking.username AS blocking_user,
6        blocked.query AS blocked_query,
7        blocking.query AS blocking_query
8 FROM pg_stat_activity AS blocked
9 JOIN pg_stat_activity AS blocking
10  ON blocking.pid = ANY(pg_blocking_pids(blocked.pid))
11 WHERE NOT blocked.pid = blocking.pid;
```

A caveat that catches teams off-guard: this query is only useful while a blocking situation is *still active*. By the time PostgreSQL has detected a deadlock and aborted the victim, the cycle is already gone, and `pg_blocking_pids` will return nothing. For an already-detected deadlock, the database log is the primary evidence, not the live catalog views. The catalog views are for live lock-waits that have not yet escalated.

```
1 -- MySQL/MariaDB: most recent deadlock graph
2 SHOW ENGINE INNODB STATUS;
3 -- Look at the LATEST DETECTED DEADLOCK section.
```

- 4. Correlate with application logs.** Your Spring Boot app should be logging the SQLState and the transaction name. If it is not, fix that today. The single most useful log line during an incident is the one that says: “Transaction `OrderBatchService.closeDay` failed with SQLState 40P01 on attempt 2 of 3”. You can build that from a custom `@Around` advice on `@Transactional` methods, or by configuring `spring.jpa.properties.hibernate.session.events.log` and `logging.level.org.springframework.transaction=DEBUG` for the duration of the incident.

Once you have the deadlock graph from the database and the call stack from the application, you can usually identify the pattern within minutes.

Five Deadlock Patterns That Cover Most Real Incidents

Across enough production systems, the same handful of patterns turn up again and again. None are exotic. All of them are subtle enough to survive code review.

Pattern 1: Inconsistent Lock Ordering Across Two Entities

Symptom. Two transactions touching the same pair of rows from opposite directions. The textbook example is funds transfer. The production example is something less obvious, like a “merge customer” job that touches customers in pairs and is invoked concurrently from two different triggers.

```

1 // Looks innocent. Is not.
2 @Transactional
3 public void merge(long sourceId, long targetId) {
4     Customer source = customerRepo.findById(sourceId).orElseThrow();
5     Customer target = customerRepo.findById(targetId).orElseThrow();
6     target.absorb(source);
7     customerRepo.delete(source);
8 }

```

If thread A calls `merge(1, 2)` and thread B calls `merge(2, 1)` at the same moment, you have a textbook lock-ordering deadlock. The fix is the textbook fix: impose a total order on the locks.

```

1 @Transactional
2 public void merge(long sourceId, long targetId) {
3     long firstId = Math.min(sourceId, targetId);
4     long secondId = Math.max(sourceId, targetId);
5
6     // Acquire locks in deterministic order.
7     Customer first = customerRepo.findByIdForUpdate(firstId).orElseThrow();
8     Customer second = customerRepo.findByIdForUpdate(secondId).orElseThrow();
9
10    Customer source = (firstId == sourceId) ? first : second;
11    Customer target = (firstId == sourceId) ? second : first;
12
13    target.absorb(source);
14    customerRepo.delete(source);
15 }

```

```

1 public interface CustomerRepository extends JpaRepository<Customer, Long> {
2
3     @Lock(LockModeType.PESSIMISTIC_WRITE)
4     @Query("select c from Customer c where c.id = :id")
5     Optional<Customer> findByIdForUpdate(@Param("id") Long id);
6 }

```

The substantive change here is not the addition of `@Lock`. It is the ordering of the locks. Without ordering, pessimistic locking just moves the deadlock around without preventing it.

Lesson. Whenever a transaction acquires write locks on more than one row of the same table, define an ordering and stick to it. The simplest ordering, sort by primary key, is almost always correct. Document the rule in the entity’s class comment so the next engineer who adds a method does not break it.

Pattern 2: The Shared Hot Row Hidden Inside a Parallel Batch

This is the pattern most likely to wake you at 03:14. It scales with batch size and only manifests once the data is large enough. Suppose you have orders and order items, with a foreign key from `order_item.order_id` to `order.id`. Your nightly batch closes orders in parallel chunks:

```

1 @Transactional
2 public void closeOrder(long orderId) {
3     Order order = orderRepo.findById(orderId).orElseThrow();
4     order.markClosed();
5     // ... and somewhere down the call chain ...
6     auditService.recordClosure(order); // inserts into order_audit
7     metricsService.bumpCounter(order); // updates a small "stats" row
8 }

```


When two threads close two different orders, they are not contending on `order` rows. They *are* contending on the `stats` row that `metricsService` updates. Every closing transaction wants to write to it, and the write lock on a single shared row serialises an otherwise parallel batch. The deadlock arrives when chunk A is holding a write lock on `order` row 100 and waiting for the `stats` row, while chunk B is holding the `stats` row and waiting on a foreign-key path back to data chunk A has not yet committed. Whether that FK path actually produces a wait depends on your database, your isolation level, and your indexes. With insufficiently indexed FKs and parallel writers, the cycle is depressingly easy to construct.

The fix is not better locking. It is a separation of concerns:

1. Move the hot “stats” update out of the per-order transaction. Maintain it asynchronously, or batch it once at the end.
2. If the audit must be transactional with the close, write to a dedicated audit table that does not feed back into the same lock graph.
3. When batches do touch the same tables, configure Hibernate to make insert order deterministic:

```
1 # application.properties
2 spring.jpa.properties.hibernate.jdbc.batch_size=50
3 spring.jpa.properties.hibernate.order_inserts=true
4 spring.jpa.properties.hibernate.order_updates=true
5 spring.jpa.properties.hibernate.batch_versioned_data=true
```

`hibernate.order_inserts` and `hibernate.order_updates` make Hibernate order batched DML deterministically, primarily by entity type and primary-key value. This both improves batching and reduces the probability of two parallel sessions producing a cycle when they touch the same tables. Hibernate’s own documentation recommends them for highly concurrent workloads, and they have resolved more nightly-batch deadlocks than any clever lock annotation. Set them as a baseline for batch-heavy systems, but treat them as a configuration decision, not free magic. Verify behavior with your entity graph, your batch size, and your database before declaring victory.

Lesson. Anything that every transaction in a workload writes to is a deadlock magnet. Counters, “last updated” rows, monotonically increasing audit metadata: none of these belong in the hot path of a parallel batch. Get them out.

Pattern 3: The Long Transaction That Crossed a Hot Row

This is the pattern that survives code review because it does not *look* like a transaction problem at all. A service method is annotated `@Transactional`, does a bit of work, makes a remote call, then writes a row. The remote call takes 800 ms on average and 9 seconds at the 99th percentile. While you are waiting, you are holding every lock you have already acquired, including, crucially, any row you wrote to inside this transaction.

```
1 // The 800ms remote call holds your DB locks the entire time.
2 @Transactional
3 public void enrichAndSave(long orderId) {
4     Order order = orderRepo.findById(orderId).orElseThrow();
5     order.markEnrichmentStarted(); // UPDATE; lock held
6     EnrichmentDto dto = httpClient.fetch(orderId); // 800ms to 9s
7     order.applyEnrichment(dto);
8     // implicit commit at method exit
9 }
```

Under load, the long tail of that HTTP call combined with the held lock on `order` produces a pathology that looks like a deadlock from above. The database reports it as one when a second transaction joins the cycle. Underneath, it is a *lock duration* problem.

The fix is structural:

```

1  public void enrichAndSave(long orderId) {
2      EnrichmentDto dto = httpClient.fetch(orderId);    // OUTSIDE any transaction
3
4      enrichmentTx.persist(orderId, dto);                // short, local transaction
5  }
6
7  @Component
8  class EnrichmentTransaction {
9
10     @Transactional
11     public void persist(long orderId, EnrichmentDto dto) {
12         Order order = orderRepo.findById(orderId).orElseThrow();
13         order.applyEnrichment(dto);
14     }
15 }

```

The transaction now spans only the database work. The remote call happens with no locks held. This is sometimes summarised as “never hold a database lock across a network call you do not control”, a rule that is easier to state than to enforce, but worth enforcing anyway.

Lesson. A `@Transactional` method is a contract about lock duration as much as about consistency. Audit your transactional methods for hidden I/O: HTTP calls, message broker publishes, file system reads, calls to other services. Each one is a candidate to either move outside the transaction or to wrap in a hard timeout.

Pattern 4: Mixed Optimistic and Pessimistic Locking on the Same Entity

This pattern is rare but devastating when it happens, because the cause is not visible at any single call site. It occurs when two code paths through the same entity disagree on the locking strategy: one path uses `@Version` optimistic locking, another uses `SELECT ... FOR UPDATE` (pessimistic). The optimistic path increments the version, the pessimistic path holds a row lock, and an unfortunate concurrent ordering produces a wait cycle.

The diagnostic clue is a deadlock graph showing one transaction blocked on a row-level write lock while the other is blocked on what looks like an `UPDATE ... WHERE version = ?` predicate.

There is no clever way to mix these implicitly. The dangerous state is not the mere presence of both mechanisms. `@Version` plus an occasional `PESSIMISTIC_WRITE` escalation is a defensible design. The dangerous state is *undocumented and inconsistent* use across code paths. If you keep `@Version` on an entity, decide whether pessimistic locks are an exceptional escalation mechanism (for one specific contended operation, say) or part of the normal update path, and write that decision down on the entity itself.

```

1  // Decide and document the policy on the entity itself.
2  @Entity
3  public class Order {
4
5      @Id @GeneratedValue
6      private Long id;
7
8      // Concurrency policy: optimistic. Pessimistic locks are allowed
9      // only via OrderRepository.findByIdForUpdate for the close-order path.
10     @Version
11     private long version;
12
13     // ...
14 }

```

The reconciliation is rarely fun, but it is finite. The alternative, leaving both strategies in place and hoping, is not.

Pattern 5: Hibernate Flush Order Surprises

The last pattern is the most Hibernate-specific. By default, Hibernate flushes statements at the end of a transaction (or at query boundaries, depending on `FlushMode`), and the order it chooses for those statements is not the order in which you wrote them in your code. With `hibernate.order_inserts=false` (the default), Hibernate may flush inserts in a different order across two sessions. That alone is enough to produce a deadlock when two sessions are inserting into the same tables in parallel.

Two specific incidents from real systems:

- A `saveAll` of a few hundred entities of two types (`Header` and `Detail`, with an FK from `Detail` to `Header`) deadlocks under load. Diagnosis: Hibernate is interleaving header and detail inserts; the FK constraint check inside one transaction is waiting on a header row another transaction has not yet committed.
- A bulk reconciliation job updates rows in a hash-partitioned table. Two workers process disjoint partitions, but their statements share index pages, and the gap-locking behavior of MySQL InnoDB produces a cycle.

The first incident is fixed by the same property block as Pattern 2: `order_inserts=true, order_updates=true`. The second often requires changing the index strategy itself (adding a covering index, or in extreme cases switching the isolation level to `READ_COMMITTED` for that specific operation to avoid gap locks).

```
1 // Per-method isolation override; use sparingly and document why.
2 @Transactional(isolation = Isolation.READ_COMMITTED)
3 public void reconcilePartition(int partitionId) {
4     // ...
5 }
```

Lesson. Hibernate's default flush order is an implementation detail. When you depend on it implicitly, a Hibernate upgrade or a small entity-graph change can flip the order and produce a regression. Set `order_inserts` and `order_updates` to `true` as a baseline. Treat it as posture, not optimization.

Building a Regression Test You Actually Trust

A deadlock that has happened once will happen again. The most expensive thing you can do after fixing one is to ship the fix without a regression test, then watch the same deadlock recur six months later because somebody refactored the code path.

The reason teams do not write deadlock tests is that they look hard to write. They are not. They just need Testcontainers and two threads.

```
1 @Testcontainers
2 @SpringBootTest
3 class CustomerMergeDeadlockTest {
4
5     @Container
6     static PostgreSQLContainer<> postgres = new PostgreSQLContainer<>("postgres:16");
7
8     @DynamicPropertySource
9     static void datasource(DynamicPropertyRegistry r) {
10         r.add("spring.datasource.url", postgres::getJdbcUrl);
11         r.add("spring.datasource.username", postgres::getUsername);
12         r.add("spring.datasource.password", postgres::getPassword);
13     }
14
15     @Autowired CustomerMergeService merge;
16     @Autowired CustomerRepository customers;
17
18     @Test
19     void concurrent_merges_in_opposite_order_must_not_deadlock() throws Exception {
20         long a = customers.save(new Customer("A")).getId();
21         long b = customers.save(new Customer("B")).getId();
22
23         ExecutorService pool = Executors.newFixedThreadPool(2);
```

```

24     CountDownLatch start = new CountDownLatch(1);
25
26     // Each task blocks on the latch, so they really start together.
27     Callable<Void> ab = () -> { start.await(); merge.merge(a, b); return null; };
28     Callable<Void> ba = () -> { start.await(); merge.merge(b, a); return null; };
29
30     Future<Void> f1 = pool.submit(ab);
31     Future<Void> f2 = pool.submit(ba);
32     start.countDown();
33
34     Throwable t1 = catchThrowable(() -> f1.get(10, SECONDS));
35     Throwable t2 = catchThrowable(() -> f2.get(10, SECONDS));
36     pool.shutdownNow();
37
38     // The fix is specifically about deadlocks. We assert that no thread
39     // failed with a deadlock-class exception. Other failures (e.g. one
40     // thread losing the race and finding the entity already merged) are
41     // expected and should be asserted separately according to the business
42     // semantics of merge().
43     assertThat(Stream.of(t1, t2)
44         .filter(Objects::nonNull)
45         .map(Throwables::getRootCause))
46         .noneMatch(ex -> ex instanceof CannotAcquireLockException
47             || ex instanceof PessimisticLockingFailureException);
48 }
49 }

```

The test is valuable for two reasons. It runs against the *real* database engine, since Testcontainers gives you a throwaway PostgreSQL and the deadlock detection logic you are testing is the engine's, not a mock. And the `CountDownLatch` ensures the threads start at the same instant. Without it, one thread routinely finishes before the other starts, and the test passes for the wrong reason.

When you write this test before the fix, it should fail (or be flaky, which is the same thing). When you write it after, it should pass reliably. Pin it to the CI pipeline and let it die loudly if anyone reintroduces the regression.

A Word About Retries

A pragmatic deadlock strategy includes retries, but retries are not a substitute for fixing the underlying pattern. Spring Retry composes cleanly with Spring's transaction management:

```

1  @Retryable(
2      retryFor = {
3          CannotAcquireLockException.class,
4          PessimisticLockingFailureException.class
5      },
6      maxAttempts = 3,
7      backoff = @Backoff(delay = 50, multiplier = 2.0, random = true)
8  )
9  @Transactional
10 public void closeOrder(long orderId) {
11     // ...
12 }
13
14 @Recover
15 public void recover(DataAccessException ex, long orderId) {
16     log.warn("close-order gave up after retries for order {}", orderId, ex);
17     // surface to monitoring; do not silently swallow
18 }

```

A few things matter in that snippet. Target the specific exception types that *your* stack actually produces, and verify them with an integration test, since the mapping varies across drivers, JPA providers, and Spring versions. Resist the temptation to retry on the broad `TransientDataAccessException`: that family includes connection-acquisition and timeout failures whose root causes you do *not* want to mask behind a retry loop. The `random = true` jitter on the backoff matters

too. Without it, two retrying transactions tend to re-collide on the same tick. And the `@Recover` method must surface the failure to a metric, a dead-letter queue, or a human. Never silently swallow it. A retry that always recovers without anyone noticing is how a deadlock spike becomes an outage.

The retry annotation goes on a method that is the transaction boundary, not inside it. You cannot retry from inside the same transaction, because the transaction is already poisoned. If your `@Transactional` and `@Retryable` are on the same method, Spring's proxy ordering must put the retry advice outside the transaction advice. Modern Spring Boot gets this right by default, but the ordering has historically required `@Order` adjustments in some configurations. Verify it for your version with a deliberate failing test.

The Architectural Takeaway: Discipline Beats Cleverness

Across these five patterns, the common thread is not “use this annotation” or “tune that property”. It is a small set of architectural disciplines that, applied consistently, eliminate most deadlocks before they ever reach production:

1. **Order locks deterministically.** When a transaction touches multiple rows of the same table, sort by primary key before acquiring write locks.
2. **Keep transactions short.** No HTTP calls, message-broker publishes, or file I/O inside a transaction. If you absolutely must, set a hard timeout and treat exceeding it as a failure mode.
3. **Get hot rows out of the parallel hot path.** Counters, “last update” rows, and audit metadata that every transaction touches do not belong in the per-record transaction. Aggregate them asynchronously.
4. **Pick one locking strategy per entity.** Either optimistic with `@Version` or pessimistic with `@Lock`, never both by accident. Document the choice on the entity.
5. **Make Hibernate's flush order deterministic.** `order_inserts=true` and `order_updates=true` are a sensible baseline for batch-heavy systems and prevent a class of incidents. Verify them against your own entity graph and batch size, not by faith.

None of these is a clever trick. All of them are habits. The teams that do not have 03:14 deadlock pages are the ones that internalised these habits years ago and now apply them without thinking.

The 3 a.m. deadlock is rarely a story about the database doing something exotic. It is almost always a story about the application asking the database to do something that *looks* fine in isolation and produces a cycle in concert. The database is not the antagonist of the story. It is the witness, and if you read the lock graph carefully, it will tell you exactly which line of your code put the cycle there.

Sleep well.

Want to Dive Deeper?

- **PostgreSQL documentation, Explicit Locking and `pg_locks`.** Authoritative reference for lock modes and the catalog views you need during incident response.
- **MySQL Reference Manual, InnoDB Locking and `SHOW ENGINE INNODB STATUS`.** The “Latest Detected Deadlock” section is your single most important production diagnostic on MySQL.
- **Hibernate ORM User Guide, Locking, Flushing, and Batch Operations.** The chapters on flush ordering and batched DML are short and worth re-reading once a year.
- **Spring Framework Reference, Transaction Management.** Pay particular attention to the section on isolation levels, propagation, and how `SQLExceptionTranslator` maps vendor codes to Spring's `DataAccessException` hierarchy.
- **Adelina Simion, “No Deadlocks in MongoDB: Atomic Documents and Retries with Spring Data MongoDB” (JAVAPRO).** The complementary view from the document-database side, worth reading alongside this article to see where the trade-offs differ.

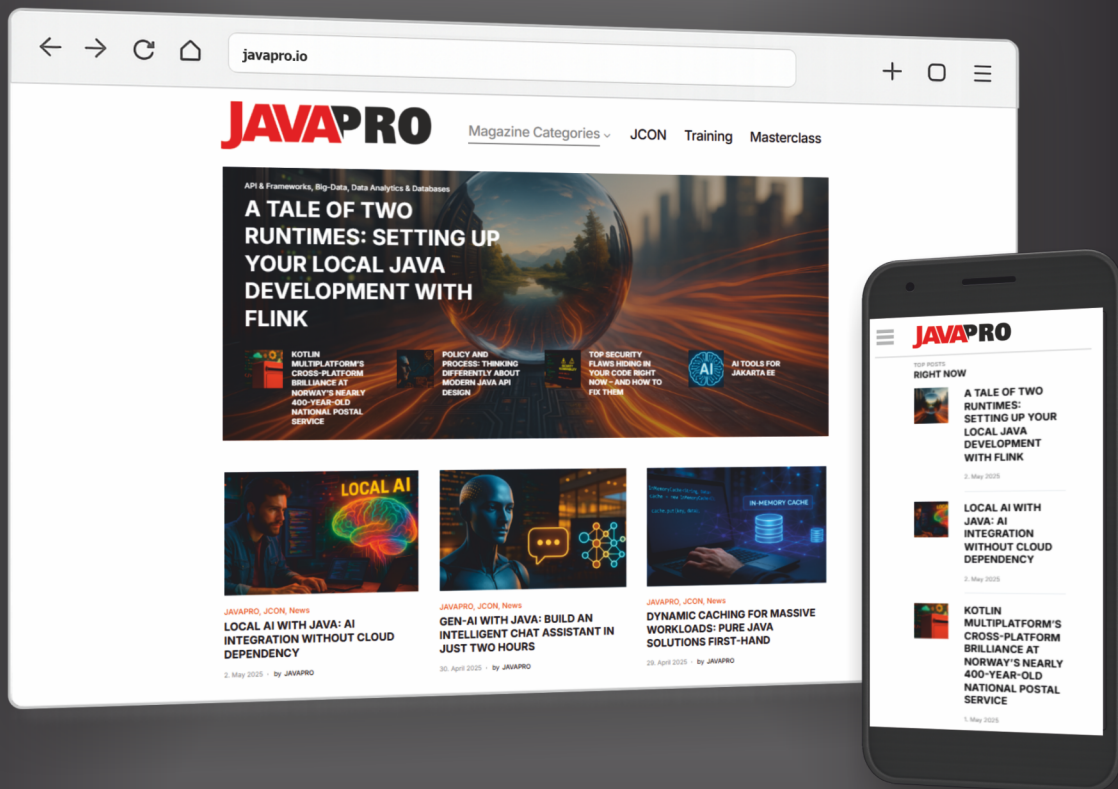
› [Back to the Table of Contents](#) ‹

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



Agentic AI Patterns for Enterprise Software

AI & ML

Langchain4j

Microservices

Quarkus

Authors: Kevin Dubois & Laura Cowen

JCON SPEAKER

Kevin Dubois is a software architect and platform engineer with a career spanning over 20 years. He is often featured as a keynote speaker at conferences around the world where he shares his experience and knowledge about cloud native & AI software development, developer experience, open source and Java. Kevin is also an author and Java Champion. He currently works as a Senior Principal Developer Advocate at IBM, and is also Technical Lead for the CNCF Developer Experience Technical Advisory Group.

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

Laura is a technical editor and technical content strategist for Java products at IBM, including open source projects Open Liberty and Quarkus. She has previously been a usability researcher, a technical writer, a user experience (UX) designer, and a developer advocate. She has a PhD in Cognitive Psychology. Laura was, for several years, a regular presenter on the popular Ubuntu Podcast, and was a founding organiser of OggCamp, a UK-based community unconference about open source software and open culture. She works at IBM Hursley in Winchester, UK.



LinkedIn



LinkedIn

It is no surprise that software development is increasingly shaped by the integration of AI. As the technology matures, we are moving beyond simple, monolithic AI models or single shot API calls. Modern enterprise architectures are rapidly adopting *agentic AI* systems, which orchestrate multiple independent reasoning agents to accomplish complex, real-world tasks. This article explores these powerful patterns, covering the core concepts of agents, their coordination, orchestration models, and crucially how to implement them effectively in enterprise Java applications.

Note that while there are many types of AI, this article focuses on interacting with Large Language Models (LLMs), commonly referred to as generative AI.

It All Starts with a Single AI Service

The first step in adopting AI in applications is typically to interact with a single LLM. This is already a nontrivial task because it requires a shift from deterministic to probabilistic thinking as LLMs inherently produce variable outputs. Many developers initially struggle with this because they are used to knowing exactly what their software will do for a given input.

Despite this shift, AI adoption in software has steadily increased due to the many use cases where a probabilistic LLM can augment applications. Early adoption often occurred in experimental Python applications, but demand for production-ready AI in established enterprise systems has exploded over the last few years. Projects like **LangChain4j** and **Spring AI** emerged to address this need in the Java ecosystem, providing idiomatic libraries that abstract the complexity of interacting with LLMs.

Thanks to these projects, enriching Java applications with AI capabilities has become much more straightforward. Prompting an LLM can now be almost as simple as adding an annotation.

Of course, things become more complex in a production setting. Even a single model interaction probably requires:

- maintaining conversational memory
- enforcing safety guardrails and other security features
- augmenting responses with proprietary data by using Retrieval Augmented Generation (RAG)
- enabling the LLM to call functions or tools, local or by using MCP for external actions
- streaming capabilities to stream responses in real time
- additional considerations like logging, tracing, (cloud-native) deployments, etc.

These “traditional” AI integrations involve calling a single LLM within a rigid programmatic scope. This works well for many use cases but reaches its limits when tasks are complex, require specialized knowledge, or involve multiple steps and decisions. This is where *agentic AI* shifts the paradigm.

The Path from a Single AI Service to Agentic Systems

We’ve established that an AI service is essentially a specialized autonomous software entity that connects to an LLM and can execute actions. *Agents* are nearly identical in functionality but conceptually distinct: they are designed to participate in an *agentic system* composed of multiple agents orchestrated by deterministic workflows or even coordinated and supervised in a probabilistic way by another LLM.

Each agent contributes a specialized capability, and together they work toward a shared objective. One additional distinction is that, while agents typically do interact with an LLM, they don’t *have* to. Some agents are simply a traditional software entity that can be executed by an agentic system. A ‘non-LLM backed’ agent would simply execute deterministic code in its body instead of relying on an LLM to provide a response.

Tools, Function Calling, and MCP

One way to build an agentic system is by providing a list of *tools* to an LLM. Tools allow the LLM to invoke external functions capable of taking action. You can add descriptions to clarify their purpose or provide parameters to supply input data. The LLM decides which tools to call and with which parameters.

Anthropic introduced **Model Context Protocol (MCP)**, a protocol to enable remote tool calling in a standard way. MCP allows you to create servers that expose deterministic or probabilistic functionality as tools. These MCP servers can be consumed by any MCP-compatible client, e.g. LangChain4j, Spring AI, Goose, LM Studio, or AI code assistants, without requiring knowledge of implementation details.

Here is an example of a simple Quarkus-based weather MCP service:

```

1 public class Weather {
2     @RestClient
3     WeatherClient weatherClient;
4
5     @Tool(description = "Get weather forecast for a location.")
6     String getForecast(
7         @ToolArg(description = "Latitude of the location") double latitude,
8         @ToolArg(description = "Longitude of the location") double longitude
9     ) {
10         return weatherClient.getForecast(
11             latitude,
12             longitude
13         );
14     }
15 }

```

This service can be registered as an MCP server and used by any MCP-compatible client. The LLM can then discover and instruct the AI client to call the weather tool as part of its invocation.

Tools enable a basic agentic system: the LLM receives a list of tools and autonomously decides which to call and in what order, and tools may invoke other AI services.

This works fairly well with an advanced LLM, a well-crafted prompt, and a *limited* set of tools. However, as the number of tools grows, the LLM becomes increasingly less effective at determining the correct sequence of actions or which tools to call.

Agentic AI Systems and Workflows

Agentic workflows are a relatively new concept in enterprise AI. Unlike the previous approaches, workflows enable much more control over which agents should run, in what sequence, and under what conditions.

Agentic orchestration patterns range from highly controlled **deterministic workflows** to flexible **autonomous systems**. Each pattern offers different tradeoffs between control, flexibility, and complexity.

1 Note: The examples below use a car rental fleet management system as a running scenario. The source code **for this** scenario as well as a further practical deep dive into **this** material can be found in the Quarkus LangChain4j Workshop.

Rigid Workflows: Predictable and Controlled

Rigid workflows offer maximum control and predictability. You define exactly which agents run, in what order, and under what conditions. This approach is especially valuable when execution sequences must be guaranteed.

Although these patterns may seem novel in their implementation, they mirror the same familiar paradigms used in traditional software development.

Sequential Execution

Sequential execution runs agents one after another in a predetermined order. Each agent's output becomes available to subsequent agents.



Example:

```

1 @SequenceAgent(
2     outputKey = "carConditions",
3     subAgents = { CleaningAgent.class, CarConditionFeedbackAgent.class }
4 )
5 CarConditions processCarReturn(
6     String carMake,
7     String carModel,
8     Integer carYear,
9     Long carNumber,
10    String carCondition,
11    String rentalFeedback,
12    String cleaningFeedback
13 );

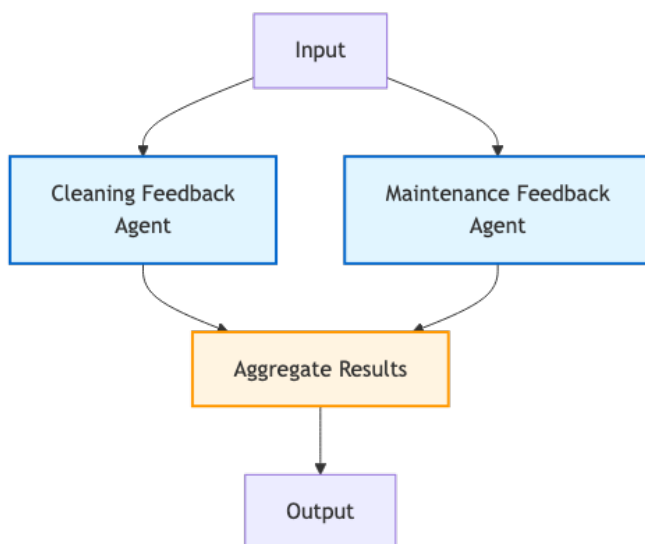
```

Here, `CleaningAgent` is invoked first, then `CarConditionFeedbackAgent`.

This pattern is deterministic and easy to understand, but it lacks flexibility: every invocation follows the same sequence regardless of actual needs.

Parallel Execution

When tasks are independent, running agents in parallel improves efficiency. You can then potentially aggregate results and use the output for further processing.



Example:

```

1 @ParallelAgent(
2     outputKey = "feedbackResult",
3     subAgents = { CleaningFbAgent.class, MaintenanceFbAgent.class }
4 )
5
6 String analyzeFeedback(
7     String carMake,
8     String carModel,
9     Integer carYear,
10    Long carNumber,
11    String carCondition,
12    String rentalFeedback,
13    String cleaningFeedback,
14    String maintenanceFeedback
15 );

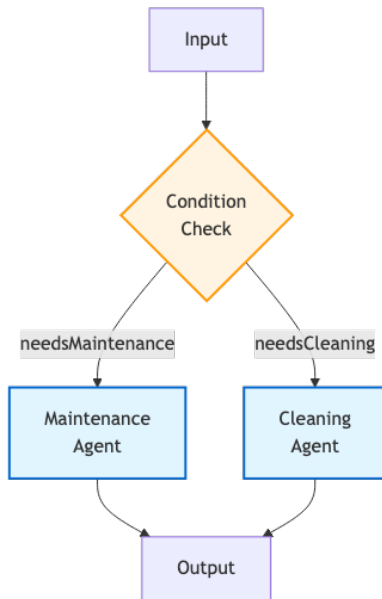
```

In this example, `CleaningFbAgent` and `MaintenanceFbAgent` are called at the same time, and the combined result is stored in the `feedbackResult` key.

Parallel execution significantly reduces processing time for LLM-based tasks, where network latency often dominates.

Conditional Routing

Conditional routing adds flexibility by selecting which agent to call at *runtime* based on given conditions:



Example:

```

1  @ConditionalAgent(
2      outputKey = "analysisResult",
3      subAgents = { MaintenanceAgent.class, CleaningAgent.class }
4  )
5  String processAction(
6      String carMake,
7      String carModel,
8      Integer carYear,
9      Long carNumber,
10     String carCondition,
11     String cleaningRequest,
12     String maintenanceRequest
13 );
14
15 @ActivationCondition(MaintenanceAgent.class)
16 static boolean assignToMaintenance(Boolean needsMaintenance) {
17     return needsMaintenance;
18 }
19
20 @ActivationCondition(CleaningAgent.class)
21 static boolean assignToCleaning(Boolean needsCleaning) {
22     return needsCleaning;
23 }
  
```

In this example, `MaintenanceAgent` is 'activated' if `needsMaintenance` is true, and otherwise `CleaningAgent` is activated if the `needsCleaning` variable is set to true.

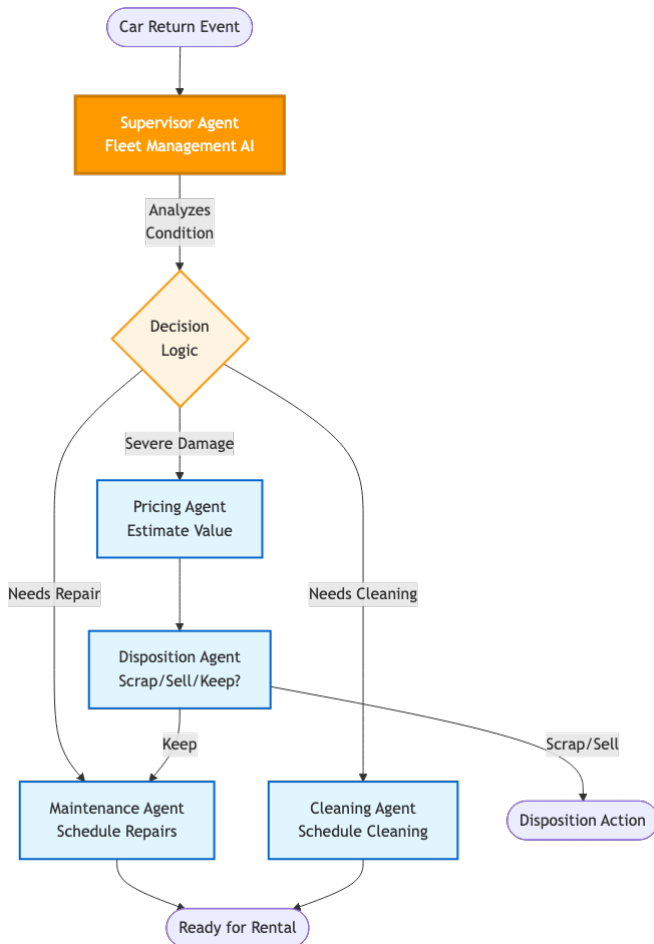
This pattern balances flexibility with explicit control and is ideal when clear business rules dictate actions.

Autonomous Agentic AI Patterns for More Flexibility

So far, we've seen rigid workflows where we determine which agents should be called and in what order. When the flow cannot be fully predetermined, autonomous agentic AI patterns allow an AI agent to call other agents based on its own LLM-backed determination.

Supervisor Pattern

The supervisor pattern strikes a balance between rigid control and autonomy. A supervisor agent receives a high-level goal and a list of worker agents, and then decides which agents to invoke, in what order, and with what parameters. In the following diagram, you can see that the decision logic needs to be more flexible to determine the flow of agents to be called and the arguments to be passed.



In code, it would look like this:

```

1  @SystemMessage("""
2      You are a fleet supervisor for a car rental company.
3      DECISION LOGIC:
4      <<< omitted for brevity >>>
5      Explain your decision-making clearly, including which agents
6      you invoked and why.
7      """)
8  @SupervisorAgent(
9      outputKey = "supervisorDecision",
10     subAgents = {
11         PricingAgent.class, DispositionAgent.class,
12         MaintenanceAgent.class, CleaningAgent.class
13     }
14 )
15 String superviseCarProcessing(
16     String carMake, String carModel,
17     Integer carYear, Long carNumber,
18     String carCondition, String rentalFeedback,
19     String cleaningFeedback, String maintenanceFeedback,
20     String cleaningRequest, String maintenanceRequest,
21     String dispositionRequest
22 );

```

In this example `SupervisorAgent` receives feedback analysis results and decides which action agents to invoke. For a car with severe damage, it might:

1. Invoke `PricingAgent` to estimate the vehicle's value
2. Invoke `DispositionAgent` to decide whether to scrap, sell, donate, or keep the car
3. If keeping the car, invoke `MaintenanceAgent` and/or `CleaningAgent` as needed

A detailed system message defines the decision logic. The supervisor can explain its reasoning, providing transparency. New worker agents can be added without modifying orchestration logic.

However, this pattern is less deterministic, and prompt engineering becomes more important because the instructions in the system and/or user message are critical for the agent to determine which actions should be taken and in what order.

Custom Agentic Patterns: Goal-Oriented Workflows

While earlier patterns rely on LangChain4j annotations, sometimes a more custom or flexible flow is needed. In that case, you can create custom planners that compute actions dynamically based on goals and state.

Goal-oriented workflows flip the traditional model of defining exact steps. Instead, you define the goal and let the system figure out how to achieve it. This approach is particularly powerful when dealing with complex state spaces where manually defining all paths becomes impractical.

The Planner Interface

A core building block for creating custom orchestration patterns in LangChain4j is the `Planner` interface. It allows you to decide at runtime how agents should behave, which ones to invoke, and when the workflow should conclude. By implementing methods such as `nextAction()`, which selects the appropriate agent based on the current state, and `done()`, which signals when the objective has been achieved, developers gain fine-grained control over complex interactions. This low-level approach opens the door to advanced techniques such as goal-oriented planning and peer-to-peer agent communication, enabling highly adaptable agentic systems.

Think of the planner as the conductor of an orchestra: it doesn't play the instruments (agents) itself but decides when each should perform and how they should work together. This separation of orchestration logic from agent logic makes complex workflows manageable and testable.

GOAP (Goal-Oriented Action Planning)

GOAP is a planning technique where you define the current state, the desired goal state, and a set of available actions. The planner then determines the sequence of actions needed to reach the goal using a dependency graph-based approach.

Using the Planner interface, each action's preconditions and effects are evaluated to construct this graph, and then the graph search is used to determine the most efficient sequence of actions to achieve the goal.

PeerToPeer (P2P) Agent Communication

Another goal-oriented approach is peer-to-peer agent communication, where agents can directly invoke other agents as needed. Rather than following a predetermined workflow, agents make autonomous decisions about which other agents to consult.

This pattern is a custom implementation that requires you to design how agents discover and communicate with each other. It's particularly useful for complex domains where the required expertise depends on the specific situation. An agent might start working on a task, realize it needs specialized knowledge, and delegate to an appropriate peer agent.

The key difference from rigid workflows is that the routing decisions happen at runtime based on the agents' reasoning, not based on predefined rules. This provides maximum flexibility but requires careful design to prevent infinite loops or inefficient agent chains.

Additional Agentic Features

So far in this article we have covered quite a few patterns to control the flow of which agents should be called, and when they should be called. In addition to these agentic workflow patterns, there are a few more patterns that are important to

keep in mind as you build your agentic AI system. Two of these are the *AgentToAgent* protocol to create distributed agents, and the *HumanInTheLoop* pattern, allowing for user input during the execution of the workflow.

A2A Communication: Distributed Intelligence

As systems grow, especially in an enterprise context, distributing agents into separate services becomes useful. For this, Google introduced the Agent2Agent (A2A) protocol. Just like in ‘classic’ distributed architectures, it allows for running agents remotely that can be developed and scaled independently, with advantages such as better isolation, reduced coupling, and reusability across different agentic systems.

Example:

```
1 @A2AClientAgent(
2     a2aServerUrl = "http://localhost:8888",
3     outputKey = "dispositionAction",
4     description = "Car disposition specialist."
5 )
6 String processDisposition(
7     String carMake, String carModel,
8     Integer carYear, Integer carNumber,
9     String carCondition, String carValue,
10    String rentalFeedback
11 );
```

In this case, the remote agent simply runs locally on a different port but behaves exactly the same as if it were a regular agent.

Human-In-The-Loop: Keeping USERS in Control

Human oversight is essential for high stakes decisions, regulatory compliance, or simply to get user input to determine what to do next. LangChain4j’s `@HumanInTheLoop` annotation allows workflows to receive user input during their invocation. These can be either synchronous/blocking, or asynchronous.

Example:

```
1 @HumanInTheLoop(
2     description = "Request human approval for vehicle disposition",
3     outputKey = "humanDecision",
4     async = false
5 )
6 static void requestApproval(
7     Long carNumber, String carMake,
8     String carModel, Integer carYear,
9     String proposedDisposition, String dispositionReason,
10    String carValue
11 ) {
12     request.set(String.format(
13         """
14         DISPOSITION APPROVAL REQUIRED
15         Vehicle: %d %s %s (Car #%d)
16         Estimated Value: %s
17         Proposed Action: %s
18         Reason: %s
19         Do you approve this disposition? (APPROVED/REJECTED)
20         """,
21         carYear, carMake, carModel, carNumber,
22         carValue, proposedDisposition, dispositionReason
23     ));
24 }
```

The workflow pauses until the user has decided whether to approve or reject the disposal of a car based on its value and the recommendation from the AI model.

LangChain4j and Quarkus

LangChain4j provides a comprehensive framework for implementing these agentic AI patterns in Java applications. The library offers two approaches: a declarative API using annotations and a programmatic API using builders. Both approaches are effective, and you can choose based on your preferences and use case.

While LangChain4j already provides a powerful and flexible toolkit for building AI-powered applications, it requires manual wiring, configuration, and lifecycle management. This is where the Quarkus LangChain4j extension is useful for enterprise Java developers. The extension transforms LangChain4j from a library into a first-class Quarkus citizen, providing enterprise-grade integration that dramatically improves developer productivity.

The Quarkus extension offers declarative AI services through CDI and annotations, eliminating boilerplate code and making AI integration as simple as defining an interface. It provides native support for multiple model providers including OpenAI, Gemini, Hugging Face, Ollama, and others, with centralized configuration through the `application.properties` configuration file. Build-time optimization validates your AI service definitions and configurations, catching errors early in the development cycle.

Built-in observability features provide comprehensive logs, metrics, and distributed traces out of the box, while Dev Services automatically provision LLM infrastructure during development. The Dev UI offers real-time insights into your AI interactions, and automatic chat memory management handles conversation context without manual intervention.

Importantly, you're not locked into a purely declarative approach. The extension allows you to seamlessly mix declarative `@RegisterAiService` annotations with low-level LangChain4j APIs when you need fine-grained control. This gives you the best of both worlds: high productivity through conventions and declarative configuration, with the flexibility to drop down to programmatic APIs when your use case demands it.

Conclusion

Agentic AI represents a fundamental shift in how we build intelligent systems. Instead of relying on a single monolithic model, we orchestrate specialized agents working collectively toward shared goals. This approach offers flexibility, maintainability, and the capability to solve complex real-world problems.

LangChain4j makes these patterns accessible to Java developers, enabling enterprise-grade agentic applications with familiar tools and strong ecosystem support.

Start simple with rigid workflows. As requirements evolve, introduce more autonomous patterns. The goal is not maximum autonomy but the *right* level for your use case.

The future of enterprise software will increasingly involve collaborative AI systems—and by understanding these patterns today, you're preparing your applications for that future.

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

**JAVAPRO**

SUBSCRIBE TO OUR FREE PDF EDITIONS & JAVAPRO UPDATES

www.javapro.io



Semantic Versioning Done Automatically

DevOps # API # MavenPlugin # Semver

Author: Jago de Vreede

JCON SPEAKER

Jago de Vreede is a full-stack software engineer at TenneT, as a software engineer, he has seen a broad spectrum of projects. He also likes to take Java to the extreme and do things with are not normally done with Java, like playing an automated guitar. He is passionate about his work and likes to share his knowledge and has therefore spoken at conferences worldwide.

Want to Dive Deeper?

Jago de Vreede is a speaker at [JCON](#)! If you can't attend live, The [session videos](#) will be available after the conference – it's worth checking out!



LinkedIn

Software evolves over time. Bugs are fixed, features are added, and occasionally design mistakes are corrected in ways that are incompatible with earlier releases. Version numbers exist to identify these distinct states of a software artifact and to allow consumers to reason about whether an upgrade is safe. Without versioning, it is impossible to distinguish between two builds of the same library or to express expectations about their compatibility.

Semantic Versioning

Software versions come in many forms, ranging from simple incrementing numbers to schemes based on release dates. One of the most widely adopted approaches is Semantic Versioning, commonly abbreviated as SemVer. SemVer defines a structured MAJOR.MINOR.PATCH version format with explicit rules about how version numbers must change in response to different kinds of software changes.

History

Before SemVer, version numbers were largely arbitrary. Consumers could not reliably infer whether upgrading from one version to another was safe or would break their builds. A change from 1.2 to 1.3 might be a small bugfix, or it might silently remove an API your application depended on. The only way to know was to read release notes, scan commit logs, or try the upgrade and see if anything failed.

As dependency graphs grew larger in the late 2000s, this uncertainty became a scaling problem. Package managers were becoming more common, applications depended on dozens of libraries, and automated upgrades were increasingly desirable but risky.

Semantic Versioning was introduced around 2009–2010 by Tom Preston-Werner, one of the founders of GitHub, as a formal specification to address this. The goal was simple: use version numbers to clearly communicate compatibility guarantees, so both humans and tools could make safer decisions about upgrades.

SemVer set out to solve the core problem of ambiguity by making the impact of a release explicit, without requiring every consumer to read documentation or reverse-engineer intent from changes.

Problem

While the SemVer specification defines clear and well-documented rules, it provides no mechanism to enforce them. In practice, adherence to SemVer relies almost entirely on developer discipline, common sense, and code reviews. As projects grow and teams change, this manual enforcement becomes inconsistent and error-prone.

This lack of enforcement leads to a familiar situation: version numbers claim semantic meaning, but consumers cannot fully trust them. A supposedly non-breaking release may still introduce incompatible API changes, undermining the very guarantees SemVer was designed to provide.

Solution

To address this problem, I created a SemVer checker as a Maven plugin. The goal was straightforward: automatically verify that version changes align with actual API changes, making semantic versioning enforceable. The plugin analyzes bytecode, compares APIs using reflection, and flags any mismatch between the declared version and the actual changes. There is no Gradle plugin available yet, but as the project is open source, contributions are welcome.

Rules

The SemVer specification consists of a set of rules that define how version numbers must change based on the nature of code changes. The most relevant rules from a consumer and compatibility perspective are listed below, along with how they are enforced by the SemVer plugin.

Many rules in the specification concern the structure of version strings themselves, such as pre-release identifiers and build metadata. These are outside the scope of the plugin. For the complete and authoritative list of rules, refer to the official SemVer specification at <https://semver.org/>

Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation.

<https://semver.org/#spec-item-1>

From an automation perspective, this is already part of Java. Public and protected classes, methods, and fields make a machine-readable public API. Beyond that, anything packaged into a JAR, configuration files or SPI descriptors for example, can also be considered part of the API, whether intended or not.

Major version MUST be incremented if any backward incompatible changes are introduced to the public API. It MAY also include minor and patch level changes.

<https://semver.org/#spec-item-8>

To enforce this rule automatically, we need to identify which changes are considered breaking and how they can be detected.

One class of backward-incompatible changes is the removal of files from the published JAR. While some files may be irrelevant, such as documentation, others may be required at runtime, for example, configuration files or service descriptors. Since the actual usage cannot be determined reliably, the plugin treats any removed file as incompatible. This can be implemented by comparing the contents of the previous and current artifacts and flagging any missing entries.

Another category consists of changes to the public API itself. Removing a public or protected class, method, or field clearly breaks compatibility, as consumers relying on those elements will no longer be able to compile or run their code. To detect these changes, the public API of the previously released version is compared against the current one using reflection. Both versions are loaded with separate class loaders to avoid class name collisions. Without this isolation, the JVM would reject loading version 2.0.0 of `com.example.MyClass` if version 1.0.0 is already loaded. The JVM enforces that each fully qualified class name can only be loaded once per class loader. By using separate class loaders, the plugin can load both versions simultaneously and compare them side by side.

Method signature changes deserve special attention. Changing parameter types or return types is generally a breaking change, but there are exceptions. Consider the following example:

```
1 public void doStuff() { } // Before
2 public String doStuff() { return null; } // After
```

If the return type of `doStuff` is modified, this is usually incompatible. However, if the original method returned void and is later changed to return a value, existing consumers can continue calling the method without modification and simply ignore the return value. In this case, the change can be classified as minor rather than major.

Finally, increasing the required Java version is also a backward-incompatible change. If the library is compiled for a newer Java release, all consumers must upgrade their runtime before they can use it. This can be detected by inspecting the class file version. Each class file starts with the magic number `0xCAFEBAFE`, followed by the minor and major version numbers. By reading the major version and mapping it to a Java release (for example, version 52 for Java 8 and 69 for Java 25), the plugin can determine whether the required Java version has increased.

Minor version MUST be incremented if new, backward-compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated.

<https://semver.org/#spec-item-7>

The most straightforward example of a minor change is the addition of new public or protected API elements. This includes new classes, methods, or fields. While scanning for removed API elements to detect breaking changes, the plugin can simultaneously look for additions. If a new public or protected element is present in the current version but not in the previous one, the change is classified as minor, as it represents new functionality that consumers may choose to use.

Another case is a change from a void return type to a non-void return type. As discussed earlier, this change does not break existing consumers. Calls to the method remain valid, and consumers who are interested in the new behavior can start using the return value. As a result, this change is classified as a minor version increment.

Annotations require more nuanced handling. Marking a public API element as deprecated using the `@Deprecated` annotation is explicitly covered by the Semantic Versioning specification and must result in a minor version bump. This can be detected reliably through reflection.

Other annotation changes are more ambiguous. Adding an annotation often introduces new behavior, but it may also be purely informational. Removing or changing an annotation can fix a bug, add functionality, or, in some cases, introduce backward-incompatible behavior. Since the semantic meaning of an annotation cannot be determined automatically, the plugin cannot reliably classify these changes. For this reason, annotation additions and removals are left configurable. The default behavior of the plugin is to classify such changes as patch-level, but developers can override this if their use case requires stricter handling.

Patch version MUST be incremented if only backward compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.

<https://semver.org/#spec-item-6>

From an automation perspective, patch changes are detected implicitly. If no major or minor changes have been identified, but the compiled output differs from the previous release, the change is classified as a patch. A practical way to detect this is by comparing the checksums of the class files. If the checksum differs, some internal implementation detail must have changed, even though the public API remains the same.

This approach works well in most cases, but it comes with important caveats. Code generators such as MapStruct or Lombok often add metadata to the generated bytecode, for example, a `@Generated` annotation containing a timestamp. Even if the generated code is functionally identical, this metadata causes the class file to differ, resulting in a different checksum and a patch classification.

Similarly, compiling the same source code with a different compiler version, or even with different compiler settings, can produce bytecode with a different layout. Again, the behavior of the code may be unchanged, but the checksum comparison will still detect a difference.

For this reason, checksum comparison is used as a pragmatic fallback: once major and minor changes have been ruled out, any remaining bytecode differences are classified as patch-level changes.

Dependencies

Most libraries have dependencies, and many have transitive dependencies as well. The SemVer specification does not provide clear guidance on how dependency changes should be versioned. The plugin currently treats all dependency changes as patch-level modifications.

This approach is based on the assumption that consumers follow the best practice of explicitly declaring all dependencies they directly use in their own projects. Under this model, if a library removes or changes a dependency, well-structured consumer projects should remain unaffected, as they declare their own requirements independently.

However, this assumption does not always hold in practice. If a consumer relies on a transitive dependency without declaring it explicitly (a form of implicit coupling) then removing that dependency could break their build. More problematic still is the reality of dependency version conflicts in the JVM ecosystem. If your library depends on Jackson 2.14 and a consumer uses Jackson 2.12, runtime failures can occur even when both projects follow best practices. The JVM's flat classpath means only one version of a dependency can be present at runtime, and resolution strategies don't always choose the compatible one.

Similarly, upgrading a dependency to a version with breaking changes could introduce incompatibilities, even if the library's own API remains unchanged. A consumer might compile successfully but encounter `NoSuchMethodError` or `ClassNotFoundException` at runtime due to version mismatches.

Incorporate in Your Project

There are multiple ways to incorporate the SemVer checker into a project. The simplest approach is to run it manually after creating your first release. You can check semantic versioning compliance by running:

```
1 mvn io.github.jagodevreede:semver-check-maven-plugin:check
```

With the default setting this will output something like:

```
1 Class io.github.jagodevreede.semver.example has been changed on byte level
2 Determined SemVer type as patch and is currently none, next version should be: 1.0.3
```

Additionally, a file named `nextVersion.txt` is generated in the target directory containing the suggested next version number.

If you want more control then you can also add the plugin to your maven POM, for example with:

```
1 <build>
2   ...
3   <plugins>
4     ...
5     <plugin>
6       <artifactId>semver-check-maven-plugin</artifactId>
7       <groupId>io.github.jagodevreede</groupId>
8       <version>VERSION_NUMBER</version>
9       <configuration>
10        <haltOnFailure>true</haltOnFailure>
11        <outputFileName>nextVersion.txt</outputFileName>
12      </configuration>
13      <executions>
14        <execution>
15          <id>check</id>
16          <phase>verify</phase>
17          <goals> <goal>check</goal> </goals>
18        </execution>
19      </executions>
20    </plugin>
21    ...
22  </plugins>
23  ...
24 </build>
```

With this configuration, the plugin runs during the verify phase and fails the build if the version number defined in the POM does not match the version required by Semantic Versioning. This makes versioning rules enforceable as part of the build process. For a full list of configuration options, consult the [README.md](#) in the project repository at <https://github.com/jagodevreede/semver-check>

Conclusion

Semantic Versioning promises clarity in dependency management, but that promise only holds if version numbers accurately reflect the changes they contain. Manual enforcement of these rules has repeatedly proven to be fragile.

By automating SemVer compliance checking, this plugin transforms versioning from a discipline problem into a build-time guarantee. Integration is straightforward as shown above. The few seconds it adds to your build time are outweighed by the potential reduction in frustration for downstream consumers later on.

The project is open source and available at <https://github.com/jagodevreede/semver-check>. Contributions and bug reports are welcome. If you've ever released a major change disguised as a patch, this tool is for you. It's designed to prevent it from happening again.

References

<https://semver.org>



SHARE TRANSPARENCY. PROTECT RISKS.
Disclose SBOMs securely and according to need

Sharing SBOMs Securely Without Giving Too Much Away

Security # Exodos # SBOM

Author: Sven Ruppert

CONSPEAKER

Sven Ruppert has been developing in Java since 1996 and has worked as a Developer Advocate for both JFrog and Vaadin. He has been speaking at conferences worldwide for over ten years. He specialises in Core Java and Secure Coding, with a focus on RAG systems and AI security. He also writes regularly for it-portals and magazines and maintains his personal website at svenruppert.com; furthermore, he serves as an instructor for Arctic winter expeditions.



LinkedIn

Why controlled transparency is the real goal — and how a tiered disclosure model achieves it

1. SBOMs Create Transparency — but Not Without Risk

The Software Bill of Materials, or SBOM for short, has undergone a remarkable shift in meaning over the past few years. What was long regarded as a technical aid for internal inventory management has since become a regulatorily mandated piece of evidence. The European Cyber Resilience Act will oblige manufacturers of digital products to document the composition of their software in a dependable manner; the NIS 2 Directive increases the pressure on operators of essential entities to secure their supply chains in a traceable way; and the United States Executive Order 14028 elevated the bill of materials to a component of government procurement as early as 2021. The bill of materials has thus risen from a voluntary artefact to an expected, mandatory disclosure.

With this elevation, however, a conflict of objectives comes to light that cannot be resolved, only managed. The very bill of materials that is meant to establish trust, enable verifiability and permit a swift response to vulnerabilities in an emergency simultaneously reveals the inner constitution of a software product. It names the third-party components in use, their version levels and therefore potential attack surfaces, and it permits inferences about architectural decisions and competitively relevant knowledge. A complete bill of materials is consequently evidence and blueprint in one. Anyone who publishes it carelessly mistakes transparency for surrender. This article therefore advances the thesis that it is not the sharing itself but its controlled design that determines whether benefit or harm results.

2. Why Complete SBOMs Contain Sensitive Information

To gauge the significance of the conflict, it is worth taking a closer look at what a complete bill of materials actually holds. Beyond the mere enumeration of the libraries used, it regularly contains their precise version levels, the full chain of transitive dependencies, occasionally internal package names, references to private artefact sources, as well as metadata about the generators used and the build process itself. Each of these particulars is harmless in isolation; taken together, however, they amount to a precise profile of a product's technical substance.

For the readership of a Java magazine, this circumstance is particularly vivid. Applications built with Maven or Gradle typically exhibit deeply branched transitive dependency chains, in which a single directly declared library pulls in dozens of further components. A complete bill of materials renders these chains visible in their entire depth. It reveals to an attentive observer not only which known vulnerabilities may potentially affect a product, but also which internal building blocks a manufacturer favours, which frameworks it deliberately avoids and where it depends on outdated versions. What was intended as a contribution to security can thereby turn into a manual for attackers. The sensitivity of a bill of materials is accordingly not a marginal phenomenon but its constitutive property.

3. Least Disclosure: Sharing as Controlled Disclosure

From this insight grows the guiding principle that forms the conceptual core of the subject: minimal disclosure, commonly known as *least disclosure*. Its substance can be captured in a single sentence: what is to be disclosed is solely that information which a particular recipient genuinely requires for a particular purpose — no more, and demonstrably so.

This formulation clears up a widespread misunderstanding that tacitly equates SBOM sharing with complete publication. Sharing a bill of materials precisely does not mean making all the information it contains available unfiltered and accessible to everyone. Rather, it means a controlled act in which content, recipient and context are always considered together. The question is not whether to share, but what reaches whom under which conditions. In this shift of perspective lies the entire difference between controlled transparency and inadvertent overexposure. Whoever takes the principle of minimal disclosure seriously understands the bill of materials not as a single document to be distributed, but as a data foundation from which differing views can be derived as the occasion demands. With this, the technical architecture to be developed in the following sections is already prefigured.

4. Different Recipients, Different Information Needs

Whoever wishes to disclose minimally must first know whom they are addressing, for the information need varies considerably between the groups of addressees. Four groups can usefully be distinguished, whose requirements determine the entire design that follows.

The broad public generally needs only a reduced view. For it, component name, licence and project reference suffice to satisfy the legitimate claim to transparency — particularly in the realm of open-source software — without surrendering the inner structure. Customers, by contrast, demand deeper insight, since they must conduct risk analyses and justify procurement decisions; they depend on version levels and dependency metadata. Auditors and supervisory authorities, in turn, require less abundance of detail than dependability: what matters to them is verifiable, complete and unfalsified evidence. Suppliers and internal teams, finally, work with the operational depth data, since they request, receive, validate, version and jointly edit bills of materials. This differentiation leads to an uncomfortable but unavoidable realisation: a single, universal SBOM view is too coarse, both in substance and in terms of security. The attempt to serve all addressees with the same mechanism, experience shows, ends in a relapse to the email attachment — that is, to the very uncontrolled practice that is to be overcome.

5. Public Transparency versus Private Exchange

From the diversity of addressees arises a fundamental structural separation that ought to underlie any well-considered disclosure model: the distinction between public transparency and private exchange. Public transparency means a deliberately reduced, publicly defensible view of the bill of materials that is accessible without further barriers. Private exchange, by contrast, denotes the controlled transfer of more detailed particulars to specifically authorised parties. The two modes must not be conflated, technically or organisationally, since otherwise the boundary between what anyone may see and what only a few should see becomes blurred.

The model described by Exodos Labs exemplifies this separation and is therefore drawn upon here as an illustration. It distinguishes expressly between a public “[SBOM Trust Center](#)” and a private “[Secure Exchange](#)”. The Trust Center provides the continuously updated, publicly defensible view, while the Secure Exchange maps the controlled exchange with specific organisations. The gain of this architecture lies less in any single function than in the clear governance boundary it draws: inadvertent overexposure is made structurally more difficult, because public and private data are assigned to separate channels from the outset. The original account of this model is found in the article *Secure SBOM Sharing Without Oversharing* by Exodos Labs ([Blogpost](#)), to which this article refers.

6. Redaction: Several Secure Views from One Bill of Materials

The structural separation of public and private does not yet answer the question of how the various views technically arise from a single complete data foundation. This is where the principle of redaction comes into play, which encompasses far more than the mere deletion of individual fields. Redaction means the purposeful reduction, masking, aggregation or recipient-dependent concealment of information.

In concrete terms, this may mean that internal package sources and references to private artefact registries are removed entirely, that transitive dependencies are condensed into an aggregated statement rather than listed individually, or that sensitive build and generator metadata remain hidden from certain recipients. From a complete bill of materials, several tiered views, each secure in itself, arise in this way: a minimal public view confined to component name, licence and project reference; an extended view for authorised customers that includes version levels and dependency metadata; and, where sensitivity requires it, a contractually safeguarded view that is released only after the conclusion of a non-disclosure agreement. The example model mentioned supports such selective redaction and provides recipient-specific views. The decisive point of this consideration is the reversal of the direction of thought: it is not the complete bill of materials that is distributed and subsequently trimmed; rather, deliberately designed views are generated from it, each of which possesses the right measure of openness for its circle of recipients.

7. Access Control Beyond Simple Roles

Once the views have been defined, the question arises by which criteria access is granted to a requester. Simple role models, as prevail in many systems, quickly prove inadequate here. Mere membership of a role — “customer” or “partner”, say — does not automatically justify access to every piece of information, for whether a particular customer may see a particular extended view depends on far more than their abstract category.

More appropriate is attribute-based access control, which brings together a range of characteristics before it releases a view. Among these characteristics are the associated organisation, the product-related entitlement, the contractual status and, where applicable, the status of a non-disclosure agreement, the assignment to a concrete release, the regulatory context

of a request, the release status, as well as any temporal limitation of access. Only the interplay of these attributes decides which view a requester actually receives. The example model drawn upon relies precisely on such attribute-based control in combination with the redaction described earlier. The conceptual added value lies in scalability: whereas rigid roles become unmanageable as the number of recipients and special cases grows, attribute-based rules can be enforced consistently even across large circles of recipients. With this, the question of who decides on disclosure is settled — complementing the previously treated question of what is concealed in the first place.

8. Demonstrability: Auditability and Release Binding

Controlled disclosure demands not only that the right measure of information be given to the right party, but also that this be capable of proof. Demonstrability here comprises two sides that belong together, because both answer the same fundamental question: what can the parties involved rely upon?

The first side concerns auditability. SBOM sharing is controllable only if it can be traced without gaps who requested access, who granted it, which view was displayed and which version was exported. The status of a non-disclosure agreement, its revocation and temporal limitations likewise belong in this audit trail. An immutable audit trail transforms sharing from a fleeting file transfer into a provable transaction; in the event of dispute, it replaces assertion with evidence. The second side concerns the binding of a bill of materials to a concrete artefact. A bill of materials is dependable only when it is unambiguously established to which release, to which build, to which JAR or WAR file, to which container image, Git tag, artefact hash or container digest it belongs. In the case of a security incident in particular, this assignment decides the capacity to act: without it, it remains open whether the bill of materials at hand actually describes the delivered artefact or a long-superseded state. Auditability thus proves who saw what and when; release binding proves what this view refers to in the first place. The two together establish the trust that a bill of materials is meant to instil at all.

9. CI/CD Integration and Conclusion

However demanding the mechanisms described may appear, in practice they most frequently founder on a plain circumstance: manual maintenance. Bills of materials that are compiled by hand, updated after the fact and published on static pages inevitably grow stale and thereby lose their value. The consequence is evident: the generation, validation, versioning and publication of a bill of materials belongs in the build and release pipeline. For Java teams this means a close interlocking with the Maven, Gradle and CI/CD processes, so that a current bill of materials arises with every build, is automatically checked against quality criteria and transferred into the released public view. The example model drawn upon illustrates this by feeding the Trust Center continuously from the supply chain, so that the public disclosure always corresponds to the actual state and the recurring question of which bill of materials is now the current one does not even arise.

Against this background, the typical mistakes that a well-considered approach avoids can be named. They range from complete public publication, through dispatch by email, the mixing of public and private views and the absence of a redaction strategy, to missing release processes, deficient auditing, absent release binding, manually maintained disclosure pages and unprovided-for means of revocation. Each of these mistakes is ultimately a variation on the same fundamental error of equating transparency with maximal disclosure.

It is precisely this equation that must be overcome. Bills of materials are indispensable for trust, regulatory compliance and the security of the software supply chain, yet maximal disclosure does not automatically mean better transparency. What is decisive is to provide information correctly, currently, appropriately for the target group and demonstrably. Secure bills of materials arise not through complete publication, but through suitable views for the right recipient in the right context.

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

Secure Coding in Java

`// from threat model to hardened production`

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

`// curriculum [11]`

01 Supply Chain Security

03 Threat Modelling — STRIDE & DREAD

05 Core Definitions — SBOM · CWE · CVE · EPSS

07 Obfuscation Techniques

09 Mutation Testing

11 Hardening a Web App — Deep Dive

02 Fundamental Security Principles

04 Security Testing — SAST · DAST · IAST · RASP

06 Attack Vectors in the Wild

08 Selected CWEs

10 Cryptography in the JDK

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

Vaadin 25.2

Security by Default

URL Protection • Safelist HTML • Clickjacking Protection • npm Hardening



The Security Innovations in Vaadin 25.2

Security

Vaadin

Secure Coding

Author: Sven Ruppert

JCONSPEAKER

Sven Ruppert has been developing in Java since 1996 and has worked as a Developer Advocate for both JFrog and Vaadin. He has been speaking at conferences worldwide for over ten years. He specialises in Core Java and Secure Coding, with a focus on RAG systems and AI security. He also writes regularly for it-portals and magazines and maintains his personal website at svenruppert.com; furthermore, he serves as an instructor for Arctic winter expeditions.



LinkedIn

With version 25.2, Vaadin presents the second feature release within the 25.x line. The release gathers a range of additions that span AI-assisted data views, access to native browser capabilities from plain Java, and tooling for load testing. As noteworthy as these enhancements may be for everyday development work, the present article deliberately directs its attention to an aspect that is easily overshadowed by the more conspicuous features, yet whose importance can scarcely be overstated: the hardening of the security-related defaults.

The guiding thesis of this article is that the security-related changes in 25.2 do not constitute a collection of isolated, individual measures, but rather follow a common design principle. They embody a shift towards secure defaults, commonly referred to as “secure by default”. Protective mechanisms that previously required explicit activation by the developer—and whose omission was, in practice, a widespread source of error—now take effect without any further intervention. The security posture of an application therefore depends less on the diligence of individual configuration decisions and rests more firmly on a foundation that the framework itself provides.

This shift is of particular relevance to organisations subject to regulatory or contractual security and compliance requirements. Where protective measures need not be set up deliberately but are already in effect in the delivered state, the likelihood that an application is operated in an unprotected condition by oversight is reduced. It is no coincidence that Vaadin published the second feature release simultaneously with the Vaadin Enterprise Edition, a separate edition aimed at organisations building business-critical Java applications. The hardened defaults address precisely this audience, for whom demonstrable security that does not hinge on individual configuration carries immediate value.

The article proceeds to examine three runtime-side innovations that integrate into the existing security model of Vaadin Flow and that the official documentation locates at the respectively appropriate points within its structure. These are the validation of URL schemes, by which script-capable and thus potentially abusable schemes such as `javascript:` are rejected by default; the controlled rendering of untrusted HTML by means of a jsoup `SafeList`, which is introduced directly into the `Html` component; and the automatically transmitted `X-Frame-Options: SAMEORIGIN` header, which prevents the abusive embedding of the application within foreign frames. A fourth measure is to be distinguished from these three mechanisms, and the article treats it separately: the hardening of the supply chain, whereby newly published npm packages are withheld for a defined grace period. It operates not at runtime but at the level of the build tooling, and for that reason alone warrants a distinct classification.

Before these innovations are discussed individually, it is worth considering the architectural foundation to which Vaadin Flow owes its reputation as an inherently secure framework. Only against this backdrop can one properly appreciate which gaps the innovations close and to which already established protective mechanisms they connect.

The Foundation: Why Vaadin's Server-Side Architecture Is Considered Inherently Secure

Before the individual innovations are discussed, it is worth considering the architectural foundation on which they rest. Vaadin Flow is a server-side framework: application state, business logic, and UI logic remain entirely on the server and never leave it. Unlike client-driven frameworks, a Flow application does not expose its internals to the browser, where an attacker might exploit vulnerabilities. The framework handles all communication between server and client through a single, secured endpoint that carries several built-in safeguards. It is from this design principle that Flow derives its reputation as an inherently secure framework—a reputation that nonetheless does not absolve the developer from observing established good practice.

How this principle plays out in the interplay between client and server can be traced through a typical operation, such as an authenticated user editing their own master data (see diagram 02-01). When the user activates a button, the framework's client-side JavaScript intercepts the event and translates it into a few precisely delimited details: the identifier of the component in question, previously assigned by the framework, together with the type of interaction and its associated event details. These details reach the server by way of the single server endpoint, which uses the standard mechanisms of the servlet session to locate the relevant user session. Vaadin then checks the session information and confirms that a component with the transmitted identifier actually exists before the server-side handler is executed. If that handler loads confidential data from storage, the developer alone decides which parts of it are presented; everything else remains on the server and is discarded once processing is complete. Only the value intended for display is sent to the client—the browser is not even aware that a complete user object exists on the server.

Request lifecycle through the single secure endpoint

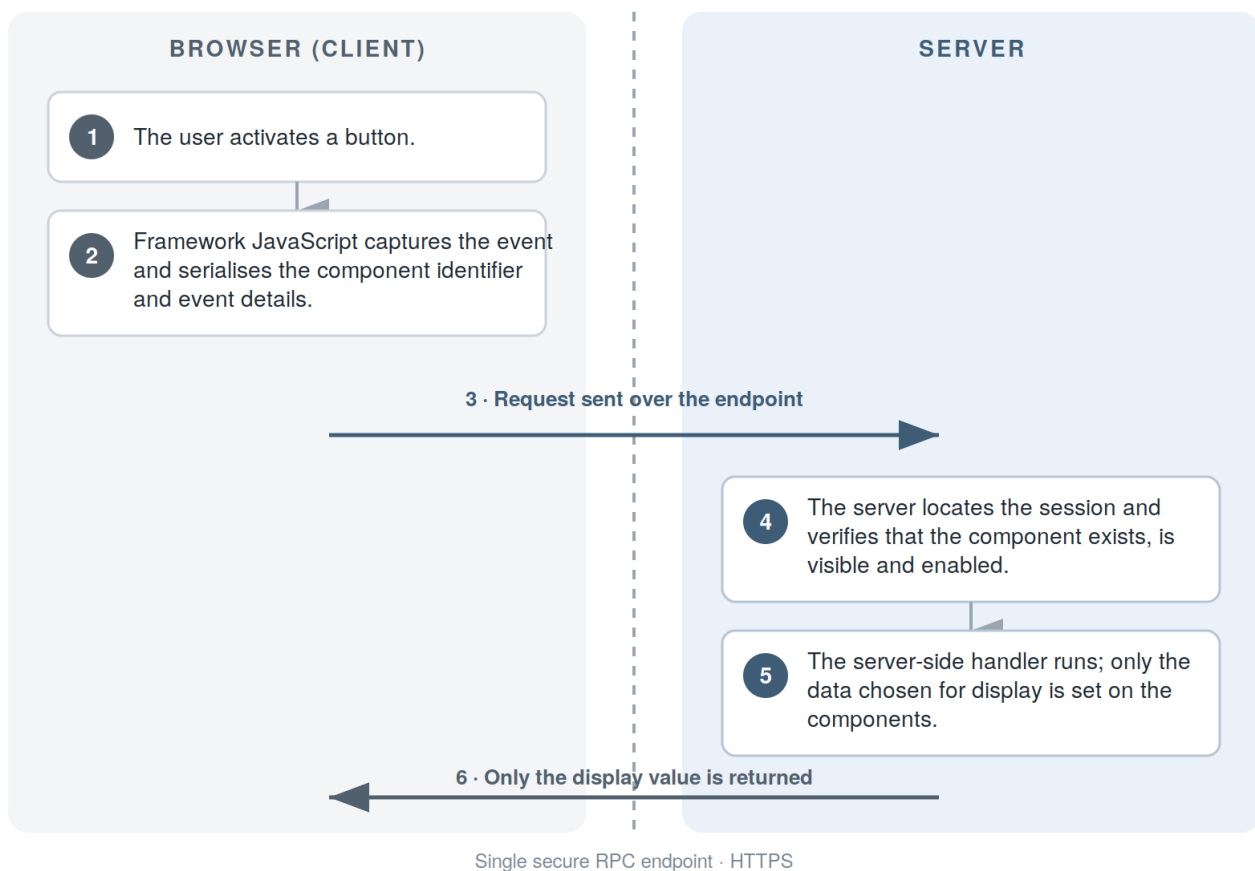


Abb. 7: Diagram 02-01: request lifecycle through the single secure endpoint

A Single Secured Endpoint and Server-Side State

In a Flow application, no public web services arise. All communication runs through a single HTTP request handler, which processes remote procedure calls via the standard servlet interface. Because the business logic is therefore not opened up to the outside as a web service, the number of entry points available to an attacker is reduced.

Beyond this, the server is aware of the application state at all times and, in particular, knows what is currently visible on the user's screen. From this follows an effective protective property: Vaadin refuses any interaction with components that are not visible or that have been disabled on the server. When the developer sets a component to disabled, this takes effect both on the server and on the client. In the browser, an attacker may well lift this disabling, since they have full control over everything that happens in the browser; the server, however, blocks any interaction with the component and records a corresponding warning in the server log.

```

1 Button button = new Button("Click me for effect!");
2 button.setEnabled(false);
3 button.addClickListener(event -> {
4     // If the button is disabled, this listener does not run,
5     // even if an attacker re-enables the button on the client side.
6 });

```

In addition, Vaadin recommends conducting all communication exclusively over HTTPS. Flow works with HTTPS without any additional configuration effort in the application code; setting up the secured endpoint is a matter for the respective servlet container.

Server-Side Validation That Cannot Be Bypassed on the Client

Vaadin's data binding API supports validation on the server that cannot be circumvented by client-side attacks. The components do also offer client-side checking, which improves the responsiveness of the application; this, however, serves convenience alone and can be defeated in the browser. As with any web application, the governing principle is therefore that all data originating from the client must be checked again as soon as it reaches the server. Relying on client-side validation alone would not be safe. To this end, Vaadin provides ready-made server-side validators; beyond these, the developer is free to draw on any Java interface for validation, up to and including connecting to external services.

```
1 TextField nameField = new TextField("Name");
2 Binder<Account> binder = new Binder<>(Account.class);
3
4 binder.forField(nameField)
5     // This validator runs on the server and cannot be
6     // bypassed from the browser.
7     .withValidator(name -> name != null && !name.isBlank(),
8         "The name must not be empty")
9     .bind(Account::getName, Account::setName);
```

A related consideration concerns data that originates from storage and is inserted as HTML into DOM elements. Such data must be escaped; the mechanisms pertinent to this are addressed in the chapter on cross-site scripting, into which the first of the three innovations fits directly. With that, the benchmark against which the following innovations are to be measured has been named.

Protection Against Unsafe URL Schemes (New in 25.2)

With the first of the three runtime-side innovations, the article turns to an area that the official documentation locates within its section on common vulnerabilities and that connects directly to the protection model against cross-site scripting named in the previous chapter. The matter at hand is the validation of the schemes with which URLs are set in the application. What previously fell to the developer, Vaadin takes on by default in version 25.2: URLs that are passed in are checked against a list of safe schemes, and an unsafe scheme leads to an immediate halt.

The javascript: Attack Vector and the New Validation

URLs that use a script-capable scheme constitute a common vector for cross-site scripting. The most prominent example is the `javascript:` scheme, which carries executable code in place of a link target; when such a URL is set as the target of a link or as the source of an embedded frame, it can, upon being triggered in the browser, bring arbitrary code to execution. It is precisely here that the innovation takes effect. The methods `Anchor.setHref`, `IFrame.setSrc`, and `Page.open` now check the URL passed to them against a list of schemes classified as safe. If the scheme proves to be unsafe, they throw an `IllegalArgumentException` and thereby prevent the problematic URL from entering the document at all.

```
1 Anchor anchor = new Anchor();
2
3 // A script-capable scheme such as "javascript:" is now rejected:
4 // this call throws an IllegalArgumentException instead of
5 // inserting the dangerous URL into the document.
6 anchor.setHref("javascript:alert('XSS')");
```

The advantage of this behaviour lies in the fact that the check takes effect not only at runtime in the browser, but already when the URL is set on the server. An unsafe URL therefore never reaches the client in the first place.

Permitted Schemes, Configuration, and Deliberate Exceptions

By default, the schemes `http`, `https`, `mailto`, `tel`, and `ftp` are considered safe. Relative URLs have no scheme and are always accepted, so that navigation within the application remains untouched by the check. The set of permitted schemes can be adjusted by way of the configuration parameter `com.vaadin.safeUrlSchemes`, which takes a comma-separated list. Depending on the operating environment, the parameter may be set, for instance, as a system property or as a context parameter of the application.

```

1 # Comma-separated list of schemes considered safe.
2 # An entry of "*" marks every scheme as safe and thereby
3 # disables the validation entirely.
4 com.vaadin.safeUrlSchemes = http,https,mailto,tel,ftp

```

A single entry of `*` marks every scheme as safe and disables the check entirely; this possibility should be used only with care, as it undoes the protection that has been gained. For the converse case, in which an individual URL is demonstrably under one's own control and may be regarded as safe, dedicated setters are available that bypass the check in a targeted manner: `Anchor.setUnsafeHref`, `IFrame.setUnsafeSrc`, and `Page.openUnsafe`. They are intended as a deliberately chosen exception and carry the lack of safety in their very name.

```

1 Anchor anchor = new Anchor();
2
3 // Use only for URLs that are fully under your control and
4 // known to be safe; this deliberately bypasses the validation.
5 anchor.setUnsafeHref(trustedUrl);

```

Placing the Measure Within the Existing XSS Protection Model

The measure fits seamlessly into the overarching protection model against cross-site scripting that already characterised Vaadin beforehand. Content is rendered as text rather than as HTML by default, and permitting executable content remains, at all times, an explicit decision by the developer. The scheme validation now closes a gap that had hitherto remained open within this model: while the textual treatment of content prevented the injection of markup, the setting of a script-capable URL persisted as an independent avenue. By barring this avenue by default whilst still providing for a deliberate exception, the innovation follows the very principle of secure defaults that runs throughout this article: the protection takes effect without any action by the developer, and lifting it demands an explicit decision that is visible in the source code.

Safe Rendering of Untrusted HTML with `Safelist` (New in 25.2)

The second runtime-side innovation remains within the domain of XSS protection and takes up a thread already hinted at in the previous chapter: the handling of data that is to enter the document as HTML. Whereas the setting of a script-capable URL is barred by the scheme validation, the deliberate rendering of HTML content remains an independent operation, one that hitherto required manual sanitisation. Vaadin 25.2 moves this sanitisation into the component itself and thereby relieves the developer of an error-prone obligation.

From the "Text Rather Than HTML" Default to Explicit HTML Allowance

Vaadin renders content as text rather than as HTML by default, by drawing on browser interfaces that treat the value passed in as plain text. In this way no markup can be injected; a value set as text is not interpreted even when it contains markup such as `<script>` elements. Some components nevertheless explicitly permit HTML for certain content. Permitting unsafe HTML content is never the default, but always a deliberate decision by the developer. Those who made it previously had to sanitise the content themselves, for instance with the aid of jsoup and a suitable `Safelist`.

```

1 Div div = new Div();
2
3 // Rendered as plain text – the markup is not interpreted:
4 div.setText("<b>This will not be bold.</b>");
5
6 // The previous approach: sanitise the content manually before
7 // adding it as HTML.
8 String safeHtml = Jsoup.clean(dangerousText, Safelist.relaxed());
9 div.add(new Html(safeHtml));

```

This approach was effective, yet it placed the responsibility entirely on the developer. If the sanitisation step was forgotten, or omitted at one of several points at which content is assigned, an entry point opened up once again.

The New Html Constructors and the Serialisation Trap

Since version 25.2, the `Html` component can perform the sanitisation itself. The new constructors take a supplier for a jsoup `Safelist` and thereby sanitise both the initial content and any content assigned later. The responsibility for sanitisation thus migrates from the call site into the component, which applies it reliably on every content assignment.

```
1 Div div = new Div();
2
3 // Since 25.2: the Html component sanitises the content itself,
4 // for both the initial content and any content assigned later.
5 div.add(new Html(dangerousText, Safelist::relaxed));
```

One important pitfall deserves particular attention. What is to be passed is a supplier in the form of a method reference or a lambda expression that creates the `Safelist` afresh each time—not, however, a pre-built `Safelist` instance that the supplier merely captures. The reason lies in serialisation: a jsoup `Safelist` is not serialisable, and capturing such an instance within the supplier would corrupt the serialisation of the session. Passing a method reference such as `Safelist::relaxed` avoids this problem, since it is not the instance but the instruction for creating it that is stored. Finally, it must be noted that the safelist has to permit the root element of the fragment to be rendered; otherwise the very element that carries the content would be removed.

Clickjacking Protection Through X-Frame-Options: SAMEORIGIN (New in 25.2)

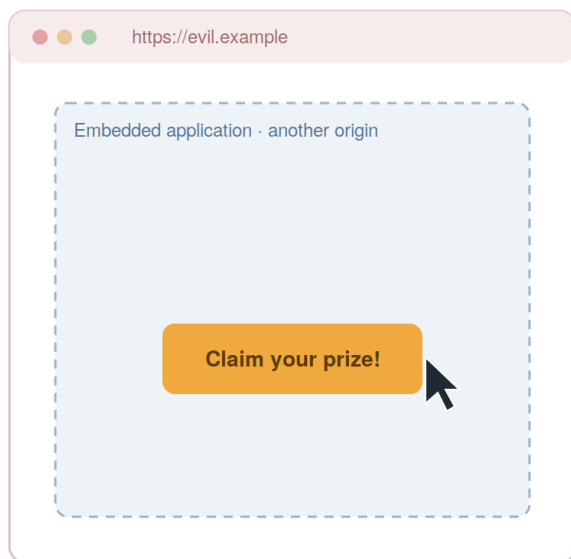
The official documentation lists the third runtime-side innovation within its section on frequently reported issues, since the absence of the relevant header was repeatedly raised as an objection in the past. With version 25.2, Vaadin now sends this header of its own accord and thereby counters, by default, a form of attack that aims at deceiving the user.

Threat Model and New Default Behaviour

In clickjacking, an application is embedded within a foreign frame and overlaid with a manipulated surface in such a way that the user unwittingly triggers actions within the embedded application whilst believing they are activating a harmless button on the surrounding page. The `X-Frame-Options` header is the established means by which a web page tells the browser that it should not be executed within a frame of another page. In this way it can be prevented that the application is embedded within a malicious page in which an attacker intercepts user actions (see figure 05-01).

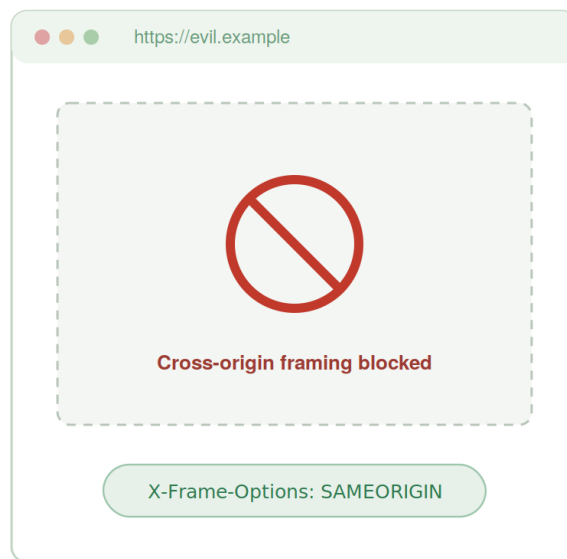
Clickjacking through frame embedding — and its prevention

1 · Without the header: clickjacking



The user aims at the decoy; the click lands on the hidden action framed beneath it.

2 · With X-Frame-Options: SAMEORIGIN



The browser refuses to display the application in a frame that originates from another origin.

Abb. 8: Figure 05-01: clickjacking through frame embedding and its prevention

In version 25.2, Vaadin sends this header by default with the application page and assigns it the value `SAMEORIGIN`. As a result, the application can be displayed only within frames of the same origin, whilst embedding by foreign origins is suppressed.

```
1 X-Frame-Options: SAMEORIGIN
```

The `frameOptions` Parameter and the Interaction with Spring Security

The default behaviour can be adjusted by way of the configuration parameter `frameOptions`. If the value is set to `DENY`, it forbids embedding within frames entirely, including by the same origin. If, on the other hand, an application is intentionally to be embedded within a frame of a foreign origin, the parameter must be set to an empty value, whereupon the header is omitted.

```
1 # Send "X-Frame-Options: DENY" to forbid framing entirely,
2 # or leave the value empty to omit the header for an application
3 # that is intentionally embedded in a cross-origin frame.
4 com.vaadin.frameOptions = DENY
```

Of particular importance is an interaction that also constitutes the most significant behavioural change upon upgrading: Vaadin sets the header only if the response does not already contain an `X-Frame-Options` header. A header that was set by another mechanism—for example, by Spring Security or a servlet filter—is not overwritten. Those who previously provided the protection by such a mechanism therefore retain the accustomed behaviour; in all other cases the new default takes effect, without displacing an existing configuration that has been set differently. Precisely because this change directly affects the behaviour of deployed applications, it warrants particular attention when moving to version 25.2.

Supply-Chain Hardening Against Compromised npm Packages (New in 25.2)

With the fourth security-relevant innovation, the article deliberately departs from the runtime-side Flow security model. It resides not in the realm of the application that processes requests at runtime, but at the level of the build tooling that assembles the application in the first place. This shift in the level of effect is the reason the measure is treated separately here; the reader should not confuse it with the three runtime mechanisms discussed previously.

The background is a risk that affects any application whose build draws on packages from a public registry. If such a package is compromised—for instance, because an attacker obtains a maintainer’s credentials and publishes a malicious version—the harmful code can enter the build, and thereby the deployed application, before the community has even noticed the incident. Such attacks on the supply chain unfold their effect precisely in the short window immediately after publication, during which the compromised version is already available but the abuse has not yet been detected.

Vaadin 25.2 counters this risk with a simple but effective default: by default, the framework does not install npm packages that were published within the last day (see diagram 06-01). The ecosystem is thereby granted the time to detect a compromised release and pull it before it reaches the build. The measure therefore deliberately delays the arrival of freshly published packages by a grace period and denies the attacker the very short window on which they depend.

Supply-chain grace period for npm packages

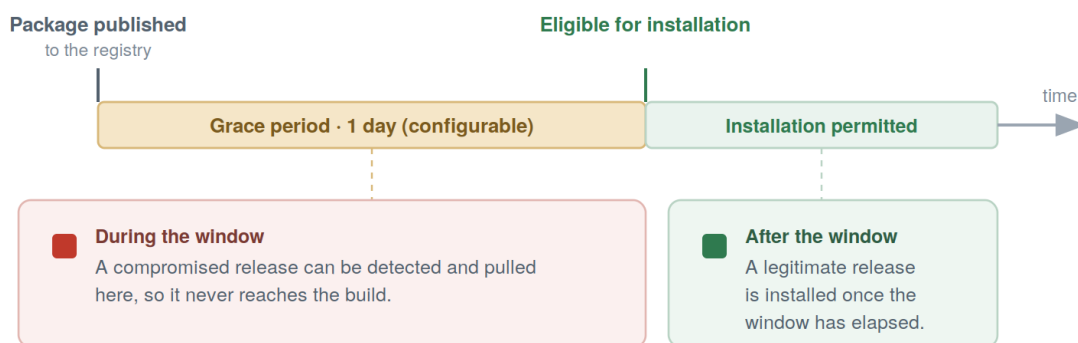


Abb. 9: Diagram 06-01: supply-chain grace period for npm packages

The length of this grace period is configurable and can be shortened or lengthened as required; the precise means of configuration is set out in the official documentation. As a default of the build tooling, the measure takes its place within the overarching leitmotif of the article, yet modifies it in one essential respect: whereas the three runtime-side innovations secure the application in operation, the supply-chain hardening already protects the application’s process of creation. Both levels complement one another, but take effect at different points—a distinction that matters for an understanding of the innovations as a whole.

The Existing Shield at a Glance, Into Which the Innovations Fit

The innovations discussed so far unfold their effect not in a vacuum, but fit into a web of established protective mechanisms that Vaadin already provides without any action by the developer. Only this overview makes visible which existing safeguards the innovations connect to and which gaps they close. Three areas deserve particular attention: the automatic protection against cross-site request forgery, the access control of navigation, and finally those fields in which responsibility deliberately remains with the developer.

Automatic CSRF Protection Following the Synchronizer Token Pattern

All requests between client and server carry a session-specific token for protection against cross-site request forgery. Because Vaadin handles all communication itself, the developer need neither insert these tokens manually nor verify them;

the protection takes effect without their involvement. In doing so, Vaadin follows the Synchronizer Token Pattern, an established design pattern for warding off forged requests.

Two tokens are in use. Requests of the UIDL protocol are protected by the `Vaadin-Security-Key` token, connections over WebSocket by the `Vaadin-Push-ID` token. The security key is generated per UI instance and sent to the client exactly once per opened browser tab, as part of the bootstrap response; on a refresh of the page or the opening of a new tab, Vaadin generates a new key. The push identifier is generated per user session and serves to check whether the identifier transmitted in the WebSocket request matches that of the session. If it is missing, the request is discarded and the WebSocket connection is closed.

```
1 {  
2   "appConfig": {  
3     "uidl": {  
4       "Vaadin-Security-Key": "f0ef03d7-0cf4-4f32-834d-47b88a1034b7",  
5       "Vaadin-Push-ID": "ad3744de-2280-4531-84df-6a70a5fe958e"  
6     }  
7   }  
8 }
```

Navigation Access Control with Annotation-Based and Path-Based Checkers

Navigation access control steps into the place of a single checking instance as the successor to the former `ViewAccessChecker`, and considerably expands its capabilities. It intercepts navigation events, evaluates the configured rules, and then decides whether the target view is displayed or access is denied. Its structure rests on three building blocks (see diagram 07-01): `NavigationAccessControl` forms the entry point, which observes the navigation events and evaluates the rules; the checkers, in the form of implementations of the `NavigationAccessChecker` interface, embody the individual security rules; and the `AccessCheckDecisionResolver` arrives at the final decision from their results.

Navigation access control

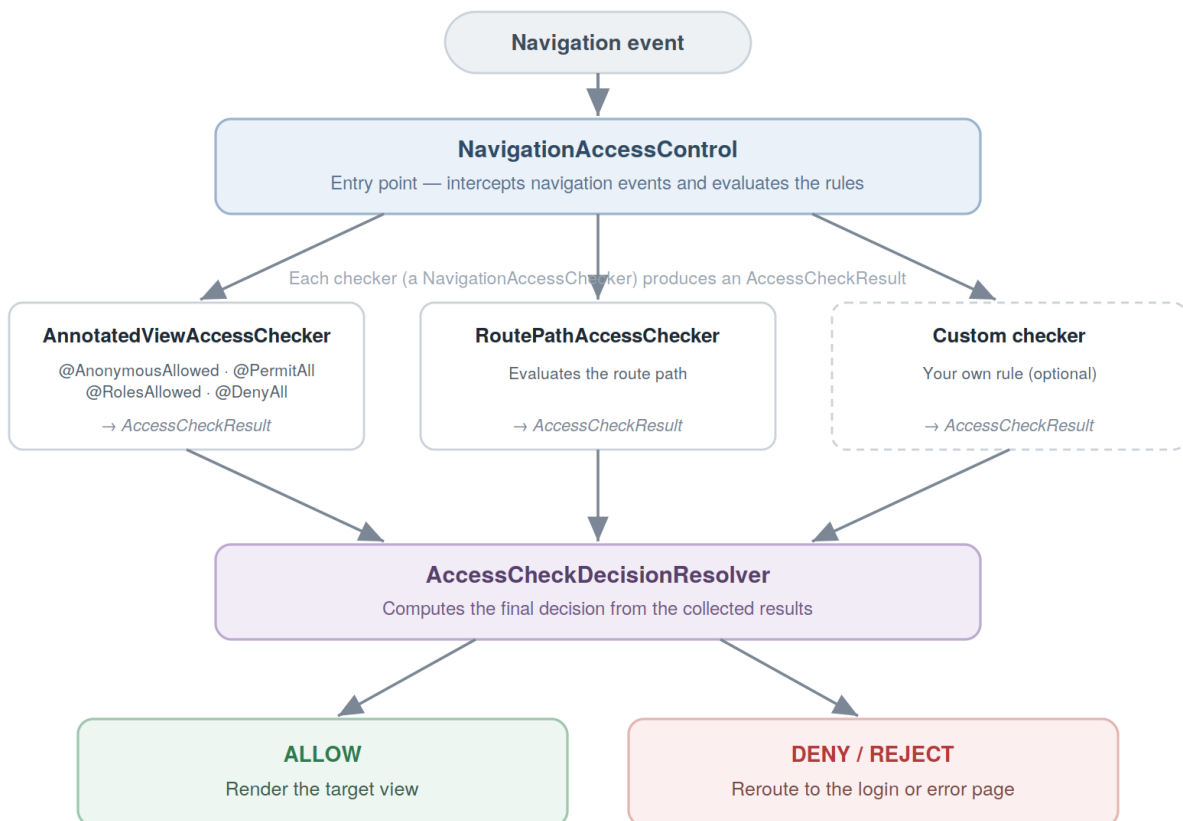


Abb. 10: Diagram 07-01: navigation access control

Vaadin ships two checkers. The annotation-based checker evaluates the security annotations `@AnonymousAllowed`, `@PermitAll`, `@RolesAllowed`, and `@DenyAll` on the target view; here, `@AnonymousAllowed` originates from Vaadin itself, whilst the role-based annotations stem from the standardised annotation interface that Vaadin recognises. The path-based checker, by contrast, evaluates the path used for navigation and, in Spring projects, relies on an implementation that connects to the Spring Security configuration.

```

1 @Route("admin")
2 @RolesAllowed("ADMIN")
3 public class AdminView extends Div {
4     // Only users with the ADMIN role may navigate here.
5 }
6
7 @Route("")
8 @AnonymousAllowed
9 public class PublicView extends Div {
10     // Open to everyone, including unauthenticated users.
11 }

```

When several checkers are in use, each contributes a result, and the decision resolver derives the final decision from them. The default implementation follows the rules set out below in the regular navigation case; each individual result may read **ALLOW**, **DENY**, **NEUTRAL**, or **REJECT**, where **NEUTRAL** denotes an abstention for want of sufficient information, and **REJECT** signals a critical configuration error.

Checker results	Decision
only ALLOW	ALLOW
ALLOW and NEUTRAL	ALLOW
only DENY	DENY
DENY and NEUTRAL	DENY
only NEUTRAL	DENY
ALLOW and DENY	REJECT
ALLOW, DENY, and NEUTRAL	REJECT

If the checkers contradict one another—remaining abstentions aside—the default resolver therefore rejects the navigation, which throws an exception in development mode and so draws attention to a faulty security configuration. In the error-handling phase, almost the same rule applies, with the one deviation that uniform abstention there grants access, since the navigation target in that case is an error-handling view without information worthy of protection.

If the supplied checkers do not suffice, one's own `NavigationAccessChecker` can be implemented, whose single method takes a `NavigationContext` and returns an `AccessCheckResult`. In plain Java projects there is no out-of-the-box path checking; the desired behaviour is instead set up by way of a `VaadinServiceInitListener`, which creates the `NavigationAccessControl` with the desired checkers and the resolver. For an orientation without Spring, it is therefore above all the annotation-based checker that is of relevance, as it requires no further frameworks.

Developer Responsibility for SQL Injection, `executeJs`, and Templates

In three fields, responsibility deliberately remains with the developer, since Vaadin, as a UI-side framework, can take no universally applicable protective measure here. The first concerns SQL injection. Vaadin does not access storage directly, so securing it falls to the chosen backend and the developer. If prepared statements are used, however, this form of attack can be excluded entirely, since the value passed in can no longer alter the structure of the query.

```

1 // Use a prepared statement so the user-supplied value cannot
2 // alter the structure of the SQL query.
3 String sql = "UPDATE app_users SET name = ? WHERE id = ?";
4 try (PreparedStatement statement = connection.prepareStatement(sql)) {
5     statement.setString(1, value);
6     statement.setLong(2, user.getId());
7     statement.executeUpdate();
8 }
```

The second field concerns the execution of one's own JavaScript by way of `executeJs`. Running arbitrary script is inherently unsafe, since the script has full access to the entire client side; it is especially delicate when the script does not reside in the application code but is loaded dynamically. Parameters can, however, be passed safely by supplying them as arguments rather than concatenating them into the script string.

```

1 // Parameters are passed safely as arguments; never concatenate
2 // untrusted values into the script string itself.
3 String script = "window.alert($0)";
4 UI.getCurrent().getPage().executeJs(script, untrustedValue);
```

The third field concerns templates. When using templates, heightened care is called for in inserting data into the DOM as well as in the use of JavaScript. Vaadin handles string values from the server-side model safely, but has no control over what happens within the template itself with HTML or JavaScript. As already with server-side validation, the governing principle is that values originating from the client must never be trusted.

Secure Configuration and Operation

The security level of a deployed application is decided not by the mechanisms of the framework alone, but also by operational aspects that lie within the responsibility of the operator. This chapter singles out two of them: the storage of sensitive data outside the project files, and the possibilities and limits of a strict Content Security Policy.

Storing Sensitive Data Outside the Project Files

It is considered bad practice to place sensitive details such as the address, the username, or the password of a database in the application's configuration file. If such secrets enter project files or source code, they are liable to escape as soon as the project is taken into version control. The governing principle is therefore: passwords and other secrets must under no circumstances be committed to the version control system.

The documentation describes three ways to externalise sensitive data. For an orientation without additional dependencies, the first is the immediately viable one: setting the values as system environment variables. They belong to the operating system, require no library, and can be read from plain Java.

```
1 // Read secrets from the environment rather than from project files.
2 String dbUrl = System.getenv("DB_URL");
3 String dbUser = System.getenv("DB_USER");
4 String dbPassword = System.getenv("DB_PASSWORD");
```

On Unix-like systems the variables are set roughly as follows; for lasting effect, the commands are to be added to the profile file of the login shell.

```
1 export DB_URL=jdbc:postgresql://localhost:5432/postgres
2 export DB_USER=postgres
3 export DB_PASSWORD=secret
```

The two further documented ways are tied to Spring Boot and are therefore of secondary importance for a Spring-free application. One imports an external properties file by way of the Spring property `spring.config.import`, so that the secrets remain outside the project directory. The other stores the values encrypted by means of Jasypt: the plain-text values marked as `DEC(...)` are encrypted in place to `ENC(...)` and may then enter version control, since they are present only in encrypted form. Decisive here is the caveat that the master password used for encryption must itself never enter version control; should it fall into the wrong hands, all encrypted values are compromised as soon as the attacker also has read access to the encrypted file.

Strict Content Security Policy: Possibilities and Limits

The Content Security Policy is a browser standard that counters certain types of attack by allowing the application to define rules for how JavaScript may be loaded and executed. A strict form offers an additional layer of protection against cross-site scripting, data injection, data theft, defacement of the page, and the distribution of malware.

Vaadin Flow, however, is in general not readily compatible with strict CSP rules. The reason lies in the nature of the client-server communication, which makes use of dynamically generated JavaScript functions and of `eval` calls—both of which are incompatible with strict CSP rules. With some effort, a nonce-based strict CSP can nevertheless be employed. A nonce is a random identifier, generated afresh by the server for each HTTP request, that is included in the response headers; once this has been done, the browser loads only those script elements that bear this nonce, whilst dynamic functions and `eval` calls remain prohibited. This approach is available exclusively in production mode.

The generation of the nonce is undertaken by an `IndexHtmlRequestListener`, which creates the identifier and adds it both to the response of the index file and to all script elements.

```
1 String nonce = UUID.randomUUID().toString();
2
3 // Require the nonce on every script tag via the CSP header.
4 response.getVaadinResponse().setHeader("Content-Security-Policy",
5     "script-src 'nonce-' + nonce + '");
6
7 // Add the nonce to all script tags in the host page.
8 response.getDocument().getElementsByTag("script").attr("nonce", nonce);
```

That alone, however, does not suffice. In order to handle the dynamic functions, the `eval` calls, and the on-demand loading of code in a compliant manner, the application must provide its own JavaScript file that overrides the `Function` constructor and `eval` and offers equivalent, predefined functions for all calls made by the application. This effort is considerable and, in practice, justifiable only where a strict CSP is strictly required. The chapter thus makes both points plain: that a strict CSP is attainable with Vaadin Flow, and that it comes at a price.

Migration Notes: Security-Driven Behavioural Changes When Upgrading to 25.2

Since 25.2 is a feature release within the 25.x line, most applications take the version jump without difficulty. Nevertheless, the hardening of the defaults brings with it a number of security-driven behavioural changes that ought to be considered before raising the version. They are not shortcomings of the upgrade, but the direct consequence of the principle of secure defaults: where a protection now takes effect without any action, the behaviour changes relative to a state in which it was previously absent.

Three changes deserve attention. First, Vaadin now sends the `X-Frame-Options: SAMEORIGIN` header by default; applications that are deliberately embedded within a frame of a foreign origin must permit this explicitly by emptying the parameter. Second, Vaadin validates the schemes of URLs that are set; this becomes relevant for teams that previously used unconventional schemes, and there calls either for an adjustment of the permitted schemes or for recourse to the deliberate exception setters. Third, the build tooling withholds freshly published npm packages for a grace period, which can affect builds that depend on only recently published packages.

The following overview summarises the changed defaults and the exception or adjustment mechanisms provided for each.

Innovation	New default behaviour	Exception or adjustment mechanism
URL scheme validation	Unsafe schemes such as <code>javascript:</code> are rejected	Adjust the permitted schemes via <code>com.vaadin.safeUrlSchemes</code> ; exempt individual URLs via <code>setUnsafeHref</code> , <code>setUnsafeSrc</code> , or <code>openUnsafe</code>
Clickjacking protection	<code>X-Frame-Options: SAMEORIGIN</code> is sent	Tighten to DENY via <code>frameOptions</code> , or empty it for cross-origin embedding; a header already set by other means is left untouched
Supply-chain hardening	npm packages younger than one day are not installed	Shorten or lengthen the grace period as required

A fourth innovation, the sanitising rendering of untrusted HTML by way of the new `Html` constructors, is not listed here, since it is additive: it adds a new, explicitly chosen possibility without altering existing behaviour, and thus constitutes no migration risk.

Before raising the version, the review of the official release notes and the upgrade guide is advisable in every case; the guide walks through the individual changes in detail—including those beyond the security area, such as those to the Signals API and to the build and tooling configuration. Those who are aware of the behavioural changes named here, and who deploy the provided exception mechanisms in a targeted manner, will, as a rule, carry out the upgrade smoothly.

Conclusion: Security as a Default and Its Significance for Compliance Requirements

At the end of this examination, the individual innovations come together into a coherent picture. Vaadin 25.2 augments the already robust server-side security model of Flow—whose inherent strength was outlined in the introductory architecture chapter—with three effective runtime-side defaults: the validation of unsafe URL schemes, the sanitising rendering of untrusted HTML by way of the new `Html` constructors, and the automatically transmitted clickjacking protection. Beyond these, it extends the model by a measure that secures the supply chain itself. None of these innovations reinvents the security model; each, rather, closes a hitherto open gap at precisely the point where the existing model anticipated it.

The real gain lies less in the individual mechanisms than in the principle that connects them. Secure defaults reduce that source of error which weighs most heavily in practice: the omission of a necessary configuration. Where a protection already takes effect in the delivered state, and lifting it requires an explicit decision visible in the source code, the burden of proof shifts in favour of security. An application is then protected not because someone remembered to make it so, but because the opposite would require a deliberate intervention.

It is precisely this reliability that lends the innovations significance beyond their purely technical value. For organisations subject to regulatory or business requirements, what counts is not merely that a protection is possible, but that it demonstrably takes effect independently of the diligence of individual configuration decisions. It is therefore consistent that Vaadin assigns the hardened defaults to the same audience addressed by the Enterprise Edition released alongside them: both aim at an operation in which security is understood not as a subsequent addition, but as a property of the foundation.

Those who put 25.2 into operation should be aware of the few security-driven behavioural changes and deploy the provided exception mechanisms in a targeted manner; for the rest, they may rely on the defaults doing their work without needing to set them in motion themselves. For a deeper engagement with the individual mechanisms and their configuration, the official security documentation remains the authoritative source.

› **Back to the Table of Contents** ‹



How to Write Your Own DbUnit

Testing & Quality

CoreJava

Frameworks

Author: Alexei Bratuhin

Alexei is senior software engineer at OpenValue. In course of his over 20 years specializing in Java career he collected vast experience in application design, development, operations and support in multiple domains. He is also a passionate contributor to open-source whenever he can - both in his private time and as part of his project work.



LinkedIn

You need to reach that (at least) 80% test coverage demanded either by your outer customer or your inner perfectionist. To achieve that noble goal you need a lot of tests – which usually means that you need a lot of test data.

Moment after moment, sprint follows sprint, and now you are suddenly left alone in the dark (mode UI) debugging the inner depths of some hundreds or thousands LOCs of the dreadful `TestData` class, trying to understand the dependencies between helper and builder methods all trying to simplify creation of complex interwoven data objects.

Or worse, you need to dive into some persistence mock layer someone thought would be a good idea to introduce a while ago (shortly before leaving the project, obviously).

Or you notice that your tests were accidentally working simply because all the database interaction in the test was running in the same transaction and on production systems nothing actually was persisted.

Or you are caught in the urgent need to exactly order the test method executions due to reuse of residue data.

This list of horror scenarios can be continued forever, and I am almost certain each and every developer faced above challenges in one or another form.

So – are we doomed or is there a way out of these torment circles?

The First Step to Dealing With a Problem Is Admitting That You Have a Problem. — Jase Robertson

How do you spot the problem? Some of the indicators were already mentioned above, but here is a list of what I have already seen in the wild:

```
1 TestUtil#createTestData(Connection) Testdata#createX()
2
3 EntityManager#unwrap(Session.class)
4
5 @BeforeEach void setUp() {
6     datasource.getConnection()
7         .execute("src/test/resources/some-test.sql");
8 }
```

not using Testcontainers/using a different in-memory database as a replacement for the real one

- H2
- embedded versions of MySQL/Redis/etc.

Is There a Better Way?

Yes!

The core problem stated above is either

- a) lack of separation between test logic and test data, or
- b) cumbersome boilerplate to fill data into the database before test execution/ to clean up the database afterwards

Let's briefly look into DbUnit and Database Rider (starting with Database Rider) and check their offering to see how they can help us.

In a nutshell, Database Rider offers you the possibility to easily preseed your database before the test and assert on the database after the test by means of (mostly) only 3 annotations:

- `@DBRider` ~ to tell the test that it's going to use
- `@DataSet` ~ to preseed the database
- `@ExpectedDataSet` ~ to assert on the database

So instead of using `State \+ Action \= Reaction`, asserting on Reaction and hoping the whatever is right was indeed persisted, you now do `State \+ Action \= Reaction \+ NewState`, also verifying the persistence layer itself.

A typical test could look like ([example test](#))

```

1  @QuarkusTest
2  @DBRider
3  public class QuarkusDBUnitTest {
4      @Test
5      @DataSet("book-empty.yml")
6      @ExpectedDataSet("book-expected.yml")
7      public void shouldCreateBookViaRestApi() {
8          final Book book = new Book("Joshua Bloch", "Effective Java (2nd Edition)", 2001,
9                                     "Tech", "978-0-3213-5668-0");
10
11          given()
12              .body(book)
13              .contentType(MediaType.APPLICATION_JSON)
14              .when()
15              .post("/api/books")
16              .then()
17              .statusCode(CREATED.getStatusCode());
18      }
19  }

```

with `book-empty.yml`

```
1 book:
```

and `book-expected.yml`

```

1 book:
2   - id: "regex:[0-9]{1,3}" #any number with 1 to 3 digits
3     author: "Joshua Bloch"
4     genre: "Tech"
5     isbn: "regex:^[0-9]{4}-[0-9]{1,3}-TEST$" #ex: 1234-291-TEST or 1234-29-TEST or
6         1234-2-TEST
7     title: "Effective Java (2nd Edition)"
8     year: 2001

```

respectively.

And in the background, Database Rider would use DbUnit – defining some abstraction layer to compare datasets and to read/ write datasets from/to databases – all utilizing standard JDBC (+ database-specific extensions) features.

How Come I Have Never Heard of It Before?

Surprisingly (at least from my experience), neither DbUnit nor Database Rider seem to be widely known in the developer community – are those libraries mature enough to be used in production, is there support and/or community?

DbUnit was first released to public ([DbUnit](#)) in 2002 to assist in writing better quality tests covering the persistence layer (quick reminder: JDBC was released in 1997, see [JDBC](#)). It offered a framework to seed the database before the tests and be able to compare the contents of the database against an expected dataset. Back then stored procedures still played a huge role in implementing the business logic, so being able to also validate all the side effects was very precious.

Still it was very verbose, so Arquillian, which was already simplifying application server developers' lives added Arquillian Persistence Extension ([Arquillian Persistence](#)) to ease developers pains and actually encourage them write more tests, separate test logic from test data and de-boilerplate code.

Well as much as we all love our application servers (cough!) with introduction of Spring and later SpringBoot a similar simplified integration was needed and founded – SpringTest DbUnit ([SpringTest DbUnit](#)).

Also – and that's what I love about open-source – DbUnit Rules project ([DbUnit Rules](#)) was started based on Arquillian Persistence Extension, DbUnit Rules would later become Database Rider ([Database Rider](#)) and my go-to database testing weapon of choice in almost every project involving relational databases as persistence layer.

Refresher: Test Pyramid vs. Test Trophy, Testcontainers

Who has not been asked at least once about the test pyramid ([test pyramid](#)) in the job interviews during the course of the career? Historically the differentiation between unit and integration/component tests made a lot of sense:

- unit tests are faster and require less external dependencies compared to integration tests
- database instances are limited and subject to strict licensing and hardware requirements

So the databases would be either mocked or replaced by in-memory ones, thus reducing the reliability of automated tests and reducing functionality coverable by tests (surprise-surprise, all those stored procedures and feature/syntax differences between the databases are still present!).

But with Docker (or other container runtimes) and Testcontainers to support their usage programmatically, many databases (even the “enterpriser” ones) allow for quickly spinning up short-lived database instances for CI purposes, thus reducing the need to fiddle with the persistence layer and allowing for more robust and reliable testing also of side effects happening directly in the database.

The industry wizards have long recognized the shift (see [Kent C. Dodds, test shapes](#)) in the ratio between different test types and so should we! Assisted with modern tooling, testing becomes a breeze!

What About X?

Thankfully, we’ll always have DbUnit for relational databases (until maybe someday license changes do us part). We also have neat integrations (Database Rider has grown to support not only Spring/SpringBoot, but also Quarkus, Micronaut and many others). But what about non-relational databases? Mongo, Redis, Cassandra, Neo4j? More esoteric and rarely used like ExistDB or maybe even Eclipsestore?

Although there are several projects, that try to close the gap – e.g. NoSqlUnit ([NoSqlUnit](#)), CassandraUnit ([CassandraUnit](#)) or MongoUnit ([MongoUnit](#), [MongoUnit GitHub](#)) – they mostly look either stale or abandoned.

So how hard is it **really** to write (and support) a framework like DbUnit or extension like Database Rider? Let’s try to find out.

Problem Formulation and Initial Design

We need a database to implement support for – we take etcd ([etcd](#)), which is a distributed key-value store. Selection seems to be appropriate due to its similarity to other non-relational databases/ stores (Redis, Zookeeper), no frameworks to support it (yet) and manageable amount of features.

It also comes with a Java client out-of-the-box, which is a great plus for us.

Now to what we want to achieve in our `EtcdUnit`. We want a Database Rider similar setup with one class-level annotation to indicate the use of `EtcdUnit` – let’s name it `@EtcdUnit`. We also need `@EtcdDataset` and `@EtcdExpectedDataset` to preseed the database before the test method execution and assert once the test method completes. We only plan for support of JUnit 5 – so we’ll use JUnit 5 Extension. Last but not least, we want to support YAML datasets only.

Note: we will use words “database”, “storage” and “store” interchangeably when addressing etcd.

Quick Introduction to etcd and Its Java Client jetcd

Etcd ([etcd](#)) is a strongly consistent, distributed key-value store. Most of us know it from Kubernetes, where it is used for all API server data ([Kubernetes components](#)).

Although it offers several core features – watches, leases (also supported by the jetcd client) – we will concentrate our efforts of the key-value store part of it.

To get ourselves familiarized with the API let us first write a small test not relying on using future `@EtcdUnit`.

First, some helper methods

```

1 public class BsUtils {
2     public static ByteSequence s2bs(String s) {
3         return ByteSequence.from(s, StandardCharsets.UTF_8);
4     }
5
6     public static String bs2s(ByteSequence bs) {
7         return bs.toString(StandardCharsets.UTF_8);
8     }
9 }

```

Now the test itself

```

1 public class ClientTest {
2
3     @Test
4     void valueUpdateIsSuccessful() throws ExecutionException, InterruptedException {
5         try (EtcdCluster cluster = Etcd.builder().withDebug(true).withNodes(1).build()) {
6             cluster.start();
7             Client client = Client.builder().endpoints(cluster.clientEndpoints()).build();
8
9             // setup
10            ByteSequence key1 = s2bs("key1");
11            ByteSequence value1 = s2bs("value1");
12            client.getKVClient().put(key1, value1).get();
13
14            // business logic
15            ByteSequence value2 = s2bs("value2");
16            client.getKVClient().put(key1, value2).get();
17
18            // assertions
19            assertThat(bs2s(client.getKVClient().get(key1).get().getKvs().getFirst().getValue())
20                .isEqualTo("value2");
21            assertThat(DatasetReader.from(client).getRows()).hasSize(1);
22
23            client.close();
24            cluster.stop();
25        }
26    }
27
28    @Test
29    void deleteAllIsSuccessful() throws ExecutionException, InterruptedException {
30        try (EtcdCluster cluster = Etcd.builder().withDebug(true).withNodes(1).build()) {
31            cluster.start();
32            Client client = Client.builder().endpoints(cluster.clientEndpoints()).build();
33
34            // setup
35            ByteSequence key1 = s2bs("key1");
36            ByteSequence value1 = s2bs("value1");
37            client.getKVClient().put(key1, value1).get();
38
39            // business logic
40            ByteSequence key = s2bs("\0");
41            client.getKVClient().delete(key, DeleteOption.builder().withRange(key).build()).get();
42
43            // assertions
44            assertThat(DatasetReader.from(client).getRows()).isEmpty();
45
46            client.close();
47            cluster.stop();
48        }
49    }
50 }

```

Notice, how clumsy and bloated the test looks, how much boilerplate is there and how it's masking the actual test! And all that already in the simplest possible scenario of key-value store, where we don't have to create any complex linked structures!

There is definitely room for improvement, so let's proceed.

Implementation

Note: we omit **import** statements for brevity, full source code can be found at [source code](#)

Let's start with defining the extension skeleton first.

```

1  public class EtcdUnitExtension
2      implements BeforeTestExecutionCallback,
3         AfterTestExecutionCallback, BeforeEachCallback, AfterEachCallback,
4         BeforeAllCallback, AfterAllCallback {
5
6      @Override
7      public void afterAll(ExtensionContext context) throws Exception {
8          System.out.println("afterAll");
9      }
10     @Override
11     public void afterEach(ExtensionContext context) throws Exception {
12         System.out.println("afterEach");
13     }
14     @Override
15     public void afterTestExecution(ExtensionContext context) throws Exception {
16         System.out.println("afterTestExecution");
17     }
18     @Override
19     public void beforeAll(ExtensionContext context) throws Exception {
20         System.out.println("beforeAll");
21     }
22     @Override
23     public void beforeEach(ExtensionContext context) throws Exception {
24         System.out.println("beforeEach");
25     }
26     @Override
27     public void beforeTestExecution(ExtensionContext context) throws Exception {
28         System.out.println("beforeTestExecution");
29     }
30 }
```

Then we go over to the annotations.

```

1  @Target({ ElementType.TYPE })
2  @Retention(RetentionPolicy.RUNTIME)
3  @ExtendWith(EtcdUnitExtension.class) // <-- magic happens here, automatically including
   @ExtendWith whenever @EtcdUnit is used.
4  public @interface EtcdUnit {
5  }
```

```

1  @Target({ ElementType.METHOD })
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface EtcdDataset {
4      String value() default ""; // <-- path to the dataset file
5  }
```

```

1  @Target({ ElementType.METHOD })
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface EtcdExpectedDataset {
4      String value() default ""; // <-- path to the expected dataset file
5  }
```


Now let's create the first tests (that will inevitably fail at the moment). You've already seen the `setUp` and `tearDown` parts of bootstrapping the etcd cluster and stopping it before and the three tests simply test the happy path of different dataset with expected dataset combinations.

```

1  @EtcdUnit
2  public class ExtensionTest {
3      private static EtcdCluster cluster; private static Client client;
4      @BeforeAll
5      static void setUp() {
6          ExtensionTest.cluster = Etcd.builder().withDebug(true).withNodes(1).build();
7          cluster.start();
8          ExtensionTest.client = Client.builder().endpoints(cluster.clientEndpoints()).build();
9      }
10     @AfterAll static void tearDown() { client.close(); cluster.stop(); }
11
12     @EtcdExpectedDataset(value = "datasets/expected_put.yml")
13     @Test
14     void withEmptyPutSuccessful() {
15         client.getKVClient()
16             .put(
17                 ByteSequence.from("hello".getBytes(StandardCharsets.UTF_8)),
18                 ByteSequence.from("world".getBytes(StandardCharsets.UTF_8)));
19     }
20     @EtcdDataset(value = "datasets/single.yml")
21     @Test
22     void withPreseedGetSuccessful()
23         throws ExecutionException, InterruptedException, TimeoutException {
24         GetResponse getResponse = client.getKVClient()
25             .get(ByteSequence.from("hello".getBytes(StandardCharsets.UTF_8)))
26             .get(1L, TimeUnit.SECONDS);
27         assertNotNull(getResponse);
28         assertEquals(1, getResponse.getKvs().size());
29         assertEquals(1, getResponse.getKvs().getFirst().size());
30         assertEquals("world", getResponse.getKvs().getFirst().getValue().toString(StandardCharsets.UTF_8));
31     }
32
33     @EtcdDataset(value = "datasets/single.yml")
34     @EtcdExpectedDataset(value = "datasets/expected_overwritten.yml")
35     @Test
36     void withPreseedOverwriteSuccessful()
37         throws ExecutionException, InterruptedException, TimeoutException {
38         client.getKVClient().put(
39             ByteSequence.from("hello".getBytes(StandardCharsets.UTF_8)),
40             ByteSequence.from("javapro".getBytes(StandardCharsets.UTF_8)));
41         GetResponse getResponse = client.getKVClient()
42             .get(ByteSequence.from("hello".getBytes(StandardCharsets.UTF_8)))
43             .get(1L, TimeUnit.SECONDS);
44         assertNotNull(getResponse);
45         assertEquals(1, getResponse.getKvs().size());
46         assertEquals(1, getResponse.getKvs().getFirst().size());
47         assertEquals("javapro", getResponse.getKvs().getFirst().getValue().toString(StandardCharsets.UTF_8));
48     }
49 }

```

Now the real work begins, let's roll up the sleeves.

For starters we definitely need to configure an instance of jetcd `Client` in our JUnit 5 extension to read from and write to the database, but how are we going to do it? There are multiple options:

- passing the client endpoints via the annotation directly
- system properties
- environment variables
- some kind of client connection holder

- class lookup
- (framework-dependent, for later considerations) cdi, bean injection

Since we spin up the etcd cluster dynamically at runtime (using Testcontainers under the hood), let's use the quick-and-dirty passing of client endpoints via system properties:

```

1 public class EtcdUnitExtension
2     implements BeforeTestExecutionCallback, AfterTestExecutionCallback,
3         BeforeEachCallback, AfterEachCallback,
4         BeforeAllCallback, AfterAllCallback {
5     public static String PROP_CLIENT_ENDPOINTS = "etcd-unit-extension.client-endpoints";
6     //...
7     private Client newClient() {
8         return Client.builder()
9             .endpoints(System.getProperty(PROP_CLIENT_ENDPOINTS).split(","))
10            .connectTimeout(Duration.ofSeconds(1L))
11            .build();
12     }
13 }

```

and simultaneously update our `EtcdExtensionTest`

```

1 @EtcdUnit
2 public class ExtensionTest {
3     //...
4     @BeforeAll
5     static void setUp() {
6         //...
7         System.setProperty(
8             EtcdUnitExtension.PROP_CLIENT_ENDPOINTS,
9             cluster.clientEndpoints().stream().map(URI::toString).collect(Collectors.joining(
10                ",")));
11     }
12     //...
13 }

```

Now some helpers to read and write datasets to/from file/store (note that we purposefully only include key and value, omitting all the advanced features, e.g. `version`, `revision`, `lease`, etc.)

```

1 public class Data {
2     private String key; private String value;
3     public Data() {}
4     public Data(String key, String value) {
5         this.key = key; this.value = value;
6     }
7     public String getKey() { return key; }
8     public void setKey(String key) { this.key = key; }
9     public String getValue() { return value; }
10    public void setValue(String value) { this.value = value; }
11 }

```

Note: usage of records might be complicated by choice of de-/serialization library and/or Java version.

```

1 public class Dataset {
2     private List<Data> rows;
3     public Dataset() {}
4     public Dataset(List<Data> rows) { this.rows = rows; }
5     public List<Data> getRows() { return rows; }
6     public void setRows(List<Data> rows) { this.rows = rows; }
7 }

```

```

1 public class DatasetReader {
2     public static Dataset from(String url) {
3         Yaml yaml = new Yaml(new Constructor(Dataset.class, new LoaderOptions()));
4         InputStream inputStream = DatasetReader.class.getClassLoader().getResourceAsStream(
5             url);
6         return yaml.load(inputStream);
7     }
8     public static Dataset from(Client client) throws ExecutionException,
9         InterruptedException {
10        // see jetcd issue: https://github.com/etcd-io/jetcd/issues/266
11        ByteSequence key = ByteSequence.from("\0", StandardCharsets.UTF_8);
12        return new Dataset(
13            client.getKVClient()
14                .get(
15                    key,
16                    GetOption.builder()
17                        .withSortField(GetOption.SortTarget.KEY)
18                        .withSortOrder(GetOption.SortOrder.DESCEND)
19                        .withRange(key)
20                        .build())
21                .get().getKvs().stream()
22                .map(
23                    keyValue ->
24                        new Data(
25                            keyValue.getKey().toString(StandardCharsets.UTF_8),
26                            keyValue.getValue().toString(StandardCharsets.UTF_8)))
27                .toList());
28    }
29 }

```

```

1 public class DatasetWriter {
2     public static void to(Client client, Dataset dataset) {
3         if (dataset == null || dataset.getRows() == null) { return; }
4         dataset
5             .getRows().forEach(
6                 data -> {
7                     try {
8                         client.getKVClient()
9                             .put(
10                                ByteSequence.from(data.getKey(), StandardCharsets.UTF_8),
11                                ByteSequence.from(data.getValue(), StandardCharsets.UTF_8))
12                             .get();
13                     } catch (InterruptedException | ExecutionException e) {
14                         throw new RuntimeException(e);
15                     }
16                 });
17     }
18 }

```

And we ready to return to implementing our extension.

As agreed in the design phase, we want to preseed the database before the test method execution iff it's annotated with `@EtcdDataset`:

```

1 public class EtcdUnitExtension
2     implements BeforeTestExecutionCallback, AfterTestExecutionCallback,
3         BeforeEachCallback, AfterEachCallback,
4         BeforeAllCallback, AfterAllCallback {
5     //...
6     @Override
7     public void beforeTestExecution(ExtensionContext context) throws Exception {
8         EtcdDataset datasetAnnotation =
9             context.getElement().get().getAnnotation(EtcdDataset.class);
10        if (datasetAnnotation != null) {
11            String datasetUri = datasetAnnotation.value();
12            Dataset dataset = DatasetReader.from(datasetUri);
13            try (Client client = newClient()) {
14                DatasetWriter.to(client, dataset);
15            }
16        }
17    }
18    //...
19 }

```

Similarly, once test method execution completes iff the test method is annotated with `@EtcdExpectedDataset` we want to check that the content of the database is exactly the same as expected **and** we want to prepare the next test method execution by wiping the database clean.

```

1 public class EtcdUnitExtension
2     implements BeforeTestExecutionCallback, AfterTestExecutionCallback,
3         BeforeEachCallback, AfterEachCallback,
4         BeforeAllCallback, AfterAllCallback {
5     //...
6     @Override
7     public void afterTestExecution(ExtensionContext context) throws Exception {
8         EtcdExpectedDataset expectedDatasetAnnotation =
9             context.getElement().get().getAnnotation(EtcdExpectedDataset.class);
10        if (expectedDatasetAnnotation != null) {
11            // compare content
12            String expectedDatasetUri = expectedDatasetAnnotation.value();
13            Dataset expectedDataset = DatasetReader.from(expectedDatasetUri);
14            try (Client client = newClient()) {
15                Dataset actualDataset = DatasetReader.from(client);
16                DataComparator.compare(actualDataset, expectedDataset);
17            }
18            // cleanup all
19            try (Client client = newClient()) {
20                ByteSequence key = ByteSequence.from("\0", StandardCharsets.UTF_8);
21                client.getKVClient().delete(key, DeleteOption.builder().withRange(key).build()).
22                    get();
23            }
24        }
25        //...
26    }

```

Note the shortcut in `DataComparator` again – we are keeping it extremely simple and simply throw a `RuntimeException` with minimal information about the cause. Definitely something to improve on if you intend to use in production!

```

1 public class DataComparator {
2     public static void compare(Dataset x, Dataset y) {
3         if (x == null && y == null) { return; }
4         if (x == null || y == null) {
5             throw new DataComparisonException("One of datasets is <null>.");
6         }
7         if (x.getRows() == null && y.getRows() == null) { return; }
8         if (x.getRows() == null || y.getRows() == null) {
9             throw new DataComparisonException("One of datasets has <null> rows.");
10        }
11        if (x.getRows().size() != y.getRows().size()) {
12            throw new DataComparisonException("Different row size.");
13        }
14        boolean xInY =
15            x.getRows().stream()
16                .map( dataInX ->
17                    y.getRows().stream()
18                        .anyMatch( dataInY ->
19                            dataInY.getKey().equals(dataInX.getKey())
20                                && dataInY.getValue().equals(dataInX.getValue()))
21                    )
22                .reduce(Boolean.TRUE, (b1, b2) -> b1 & b2);
23        boolean yInX =
24            y.getRows().stream()
25                .map( dataInY ->
26                    x.getRows().stream()
27                        .anyMatch( dataInX ->
28                            dataInX.getKey().equals(dataInY.getKey())
29                                && dataInX.getValue().equals(dataInY.getValue()))
30                    )
31                .reduce(Boolean.TRUE, (b1, b2) -> b1 & b2);
32        if (!xInY || !yInX) {
33            throw new DataComparisonException("Different datasets.");
34        }
35    }
36 }

```

```

1 public class DataComparisonException extends RuntimeException {
2     public DataComparisonException(String message) {
3         super(message);
4     }
5 }

```

And just like that we are now ready to execute our test again!

Down the Rabbit Hole

Are we done yet? Do we feel lucky? Well, it depends... During the implementation we have definitely taken some shortcuts and now it's exactly the right time to revise them and sketch the roadmap for next iterations.

So if you feel adventurous and up for the challenge, here is a thing or two to add, based on Database Rider features:

Better Diff Messages

Obviously, nobody is going to like the `Different datasets.` exception message, longing for more – it's up to you to define a suitable format. Some kind of 3-way diff (left-only, both, right-only) should be appropriate.

Multiple Schemas and Multiple Databases

Usually, one (micro)service is supposed to operate on a single database, but the enterprise reality is often way harsher. The necessity to operate on multiple database of same kind could be caused by migration, company policies, write vs. read replicas and many more.

Same reasoning could be applied to use of different schemas within the same database (if supported), though (potentially, if fully qualified table names could be used) with slightly different consequences.

Although Database Rider does offer some support for multiple schemas and connections out of the box using the annotation-based approach above, the recommendation (particularly if trying to combine multiple datasources in the **same** test) is using RiderDSL (see [RiderDSL](#) and [issue 248](#))

```

1  Connection getConnFromDs1() {
2  //...
3  }
4  Connection getConnFromDs2() {
5  //...
6  }
7  @Test
8  void useMultipleDatasources() throws SQLException, DatabaseUnitException {
9      withConnection(getConnFromDs1())
10         .withDataSetConfig(new DataSetConfig("dataset-ds1.yml"))
11         .createDataSet();
12      withConnection(getConnFromDs2())
13         .withDataSetConfig(new DataSetConfig("dataset-ds2.yml"))
14         .createDataSet();
15      // ... test logic ...
16      withConnection(getConnFromDs1())
17         .withDataSetConfig(new DataSetConfig("expected-ds1.yml"))
18         .expectDataSet();
19      withConnection(getConnFromDs2())
20         .withDataSetConfig(new DataSetConfig("expected-ds2.yml"))
21         .expectDataSet();
22  }

```

Repeatable Annotations, Dataset Merging

Sometimes (in particularly in case of larger databases with many tables) your datasets contain both fixed and variable parts:

- fixed being entries that you copy over and over again between your datasets, e.g. some basic configuration, list of predefined users with predefined roles, VATs, units, etc.
- variable being entries you need specifically for the one test, e.g. a user without roles or a posts with comments nested over newly introduced maximum nesting level.

In these cases it makes sense to split the dataset and re-use the fixed part over and over again, thus embodying the DRY principle. To enable it we could make `value` in `EtcdDataset` be an array (also see [DataSet](#))

```

1  @Target({ ElementType.METHOD })
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface EtcdDataset {
4      String[] value() default "";
5  }

```

Define the annotation to be `@Repeatable`, define a “container” annotation (e.g. `@EtcdDatasets`); define the overwrite/merge rules and we’re good to go (still relatively easy due to locality of dataset configuration).

Those who want to go the extra mile, might consider merging `@EtcdDataset` annotations defined on class (`@Target({ ElementType.TYPE })`) level with the `@EtcdDatasets` on `@BeforeAll`, `@BeforeEach` and test method levels as well.

Partial Expectations

For now, we have only implemented a very strict dataset comparison strategy – both the actual and expected dataset must match exactly, all attributes of all rows. But what if we only want to assert on parts of data, what if simply cannot know all the exact values (think e.g. `now()` as default value and `auto increment` or `nextval()` for ids)?

Applied to our case, what if would only like to test for a specific key being stored in the database?

Well, in these cases we would need the ability to specify a comparison strategy. In case of Database Rider those are `EQUALS` (default), `CONTAINS` and `PROLOG` (more on it below).

Smarter Datasets

Going further, once you have familiarized yourself with the basic features, you will most probably want to add some dynamics to your datasets, e.g. value generation (e.g. “set last logged in date to yesterday” or “include contents of a large file here”). If you would want to skip creating those datasets programmatically, it” where “dataset replacers” and “scriptable dataset” features (as in Database Rider, see [dataset replacers](#) and [scriptable datasets](#)) could come in very handy.

```
1 rows:
2   - key: "[DAY,YESTERDAY]"
3     value: "Weather was OK"
```

```
1 rows:
2   - key: "groovy:java.util.UUID.randomUUID()"
3     value: "This could be your personal identifier"
```

Obviously, defining the set of available replacers and possibilities to extend them with custom implementations; as well as defining the list of supported scripting languages might be quite time-consuming.

Even Smarter Assertions

As you might have noticed, most of the improvements above were enhancing out ability to do something with single row of a dataset. But what if would need to check the dependencies between the rows, e.g. the relations between dynamically created tree nodes, where we do not know the exact ids of children and parent nodes?

Specifically for this case, Database Rider offers `PROLOG` ([PROLOG](#)) kind assertions. By using specific placeholders, formulating the state of the database as “theory” and our expectations as “query” we could assert on the dataset as a whole!

DatabaseRider:

```
1 USER:
2   - ID: $$x$$
3     NAME: "@realpestano"
4 TWEET:
5   - ID: abcdef12345
6     CONTENT: "dbunit rules!"
7     USER_ID: $$x$$
```

etcd-unit:

```
1 rows:
2   - key: "$$r$$"
3     value: '{"name": "root"}'
4   - key: "$$n_1$$"
5     value: '{"name": "root", "parent": "$$r$$"}'
```

Multiple Data Format Support

We all know the “competing standards” comic ([xkcd 927](#)), so why limit the options? Initial choice of YAML is obviously very opinionated, to conquer the audience you might need to support TXT, CSV, XML, TOML, even EXCEL!

And Many More

- data types and custom data types (think enums, arrays, json, geometric, etc.) which can be very different across different databases
- data cleanup strategies (e.g. clean all tables, clean only dataset tables)
- wildcard/regexp support (e.g. skipping data cleanup for all QUARTZ_* tables)

Conclusion

You use relational database in your project – give Database Rider a try, if you are stuck – the database rider community is there for you!

You use a non-relational database – try one of the tools listed or try creating your own.

And if you want to work on EtcdUnit – drop me a line or a PR at [EtcdUnit repo](#)!

› **Back to the Table of Contents** ‹

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

You need a **{BUDDY}**.



Richard Fichter
CEO @ XDEV



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

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

Let us move your Java project forward together!

**BEST
{BUDDYS}**
IN CODE

trusted by

ATU

EMPORIO ARMANI

CONRAD

Fraport

Red Bull

Allianz



Arrange a free discovery call here!

www.xdev.software

XDEV