

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY #JAVAPRO

30 YEARS OF JAVA SPECIAL EDITION

Save the Date



APRIL 20-23, 2026
www.jcon.one



06 **30 YEARS OF JAVA - HOW
THE LANGUAGE HAS EVOLVED**

43 **HITCHHIKER'S GUIDE
TO JAVA PERFORMANCE**

91 **A JOURNEY THROGH
30 YEARS OF API EVOLUTION**

67 **BUILD AI APPS & AGENTS IN JAVA:
HANDS-ON WITH LANGCHAIN4J**

108 **30 YEARS OF JAVA, 25 YEARS
OF ENTERPRISE JAVA**

116 **ECLIPSESTORE: THE EVOLUTION
OF JAVA PERSISTENCE**

JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



www.javapro.io

30 Years of Java

My Personal Java Experience

My journey with Java started a few years later, in 1998. Growing up with a Commodore 64 and, thanks to a good old friend who was obsessed with computers and programming and taught me coding, I went from gaming to programming with BASIC and assembly language.

When we embarked on developing a web design software, Java 2, with its vibrant Swing framework, was a game-changer. Here was a GUI framework that allowed us to build rich UIs and bring desktop-like experiences in the browser. Swing offered a level of UI richness that felt light years ahead. This conviction fueled our journey to build a full-fledged Java IDE, complete with GUI builders for Swing and later emerging Vaadin for building web UIs with Java. This eventually morphed into RapidClipse, an Eclipse distribution for visual Java development.

Our focus soon shifted to years of improving database programming with the Eclipse Hibernate tools and increasing query performance through caching optimization. Using the experience gained and a strong vision of Java-native persistence, we launched the Eclipse Serializer and the EclipseStore project. Time flew by, and suddenly we are celebrating 30 years of Java.

From JavaOne to JAVAPRO and JCON

For 20 years, the beating heart of the Java community was undoubtedly JavaOne in San Francisco. The Moscone Center was packed with developers, easily surpassing 25,000 at its peak 2021. It was like a pilgrimage, an annual gathering of the brightest minds in the Java ecosystem. The speaker lineups read like a who's who of the Java elite, sharing their knowledge, innovations, and passion. The energy was electric, a collective excitement of Java. This is where we came up with the vision of launching JAVAPRO magazine and the JCON conference to bring this spirit to the Java Communities in Europe.

In 2015, we launched JCON in Duesseldorf, Germany. To provide a first-class developer event experience, we deliberately chose a cinema. Thanks to giant screens, attendees can see the code even in the back row. During COVID, over 2,900 participants joined

JCON WORLD ONLINE. This year, JCON EUROPE returns to the Cinedom in Cologne for its third iteration.

The JCON family has even expanded, with JCON OpenBlend Slovenia, a collaboration with SIOUG, established in 2023, and the exciting prospect of the very first JCON USA at IBM TechXchange in Orlando this October. This year marks a significant milestone: ten years of JAVAPRO and JCON, a decade dedicated to the Java community.

Write Once, Run Anywhere

Looking back at the tapestry of Java's thirty-year history, certain threads stand out, both brilliant and perhaps, in hindsight, a little unfortunate. The initial vision of Sun Microsystems, to create a language for embedded systems, for set-top boxes, was perhaps less about the web initially and more about the general computing future. Yet, the „write once, run anywhere“ promise became the rallying cry for Java.

Graphical User Interfaces – Once a Java Domain

Java applets, though eventually fading, were undeniably instrumental in bringing dynamic content and interactivity to the early internet. The shift from the Abstract Window Toolkit (AWT) to Swing was a monumental leap forward. Swing provided developers with the tools to create truly first-class user interfaces, rich and platform-independent. There was a strong conviction within the Java community that complex application UIs belonged in Java, that HTML was not built for and was insufficient for the task. JavaFX should become the successor to Swing, but its arrival was too late. By then, HTML-based UIs were gaining significant traction, fueled by the rise of JavaScript. While JavaFX was technically revolutionary compared to Swing, UI development was increasingly becoming the domain of specialized front-end teams wielding JavaScript. The dream of Java dominating the client UI failed. The competing Microsoft Silverlight and the legendary Flash have even been discontinued completely.

Java Becomes Open Source

Sun's open-sourcing of Java under GPLv2 was a significant breakthrough, and the OpenJDK came with deciding benefits: increased community involvement and innovation, wider adoption and ecosystem growth, transparency and trust, and improved agility like the six-month release cycle.

Powerful Language Features

As Java matured, significant language enhancements arrived, such as generics in Java 5, which brought type safety and improved the developer experience with collections. Another pi-



Markus Kett
Editor in Chief
JAVAPRO

<https://linkedin.com/in/markuskett>
<https://twitter.com/MarkusKett>

votal evolution was the introduction of lambda expressions and the Streams API in Java 8. These features embraced functional programming concepts, leading to more expressive and efficient data manipulation. Project Jigsaw, released in JDK 9, promised modularity but sparked years of debate over classpath changes, leaving many developers indifferent. Oracle's IoT push in the 2010s, aiming to revive Java's embedded roots, gained little traction. Java dominates on servers, not sensors.

The Mobile Disaster

The mobile revolution presented another challenge. Apple's refusal to allow Java on the iPhone, dealt a significant blow to JavaFX's aspirations in that domain as well. Later, as Android rose to prominence, JavaFX was already only a niche, and other technologies had captured the attention of mobile developers. The Java community waited for a long time, hoping vainly that Oracle would solve this problem. Some community projects, such as Glueon and JPRO, are still working on this.

From Java EE to Jakarta EE

A significant shift for the enterprise Java world was the transition from Java EE to Jakarta EE, navigated new governance and branding. The open Eclipse Foundation enabled the Java community to have independent evolution and faster innovation. The platform has been modernized to better suit cloud-native environments and integrate with newer technologies.

The Big Challenge – Java in the Cloud

The cloud revolution initially seemed to present a hurdle for Java. The perception was that other languages, like Go, with their smaller footprint and faster startup times, were the preferred choice for microservices. However, the Java ecosystem has responded with remarkable agility. The evolution from monolithic application servers to lightweight microservice frameworks and runtimes like Quarkus, Micronaut, Spring Boot, and Helidon, as well as GraalVM native images to boost start-up time, has been transformative. Microservices architecture has fundamentally changed how we build applications, introducing both incredible flexibility but also the inherent complexities of distributed systems. The rise of serverless computing with platforms like AWS Lambda also presented new paradigms for Java developers to adapt to.

Reactive programming gained popularity, promising better resource utilization and responsiveness, but its inherent complexity often proved challenging. The advent of Virtual Threads in recent Java versions offers a compelling alternative, potentially simplifying concurrency and minimizing the need for complex reactive frameworks in many scenarios.

The Greatest Transformation Ever: AI in Java

Now we are already in the middle of another profound transformation: the integration of artificial intelligence into Java. This isn't a peripheral trend - it feels like a fundamental shift in how we will write, test, document, modernize, and migrate Java code. AI-po-

wered tools are already assisting developers, generating code, crafting tests, producing documentation, and aiding in the often arduous tasks of code modernization and migration.

The question on many minds is, of course, when will AI-generated code become sophisticated enough to potentially replace human developers entirely? While the near future will likely see AI as a powerful augmentation to our abilities, the long-term implications are still being written. Looking ahead, I envision a Java ecosystem even more deeply intertwined with AI. Code generation will become increasingly sophisticated, perhaps allowing developers to describe functionality at a higher level of abstraction, with AI filling in the implementation details. Testing might become largely automated, with AI generating comprehensive test suites based on code analysis and requirements. Documentation could become automated, enriched by AI insights. Code modernization and migration, often time-consuming and error-prone, will be significantly streamlined by intelligent AI tools. What is currently underestimated in AI projects, but is becoming increasingly important, is security, compliance requirements and ethical aspects.

The potential of AI is undoubtedly gigantic, and we are only at the beginning. However, the human element will remain crucial. Creativity, nuanced problem-solving, and the ability to understand and translate complex business needs into software solutions will still require human intellect. AI will be an incredibly powerful partner, but not a complete replacement, at least not in the foreseeable future.

Java – A Promise for the Future

Java's enduring dominance over the past three decades with 60 billion JVMs is a testament to its fundamental strengths. It represents a first-class investment for companies that rely on mission-critical applications with lifespans measured in decades. While other technologies may experience meteoric rises and subsequent fades, Java's vast ecosystem and large pool of million experienced developers provide stability and longevity that is invaluable. Time and again, we've heard the question: Is Java still relevant in the face of new paradigms like the web, mobile, cloud, and now AI? And each time, the Java community has not only adapted but thrived, integrating new concepts and technologies while retaining its core strengths.

From Java Applets to the AI-powered tools of today, Java's journey has been one of constant evolution. It is a language that has not just survived but has continued to shape the technological landscape. Here's to the next thirty years, to the continued growth and innovation of the Java community, and to the exciting possibilities that lie ahead in this ever-evolving world of software development. The story of Java is far from over - it is just entering its next, most fascinating chapter.

As we celebrate thirty years of Java, this special edition of JAVAPRO is a tribute to that enduring spirit of innovation and adaptation. Enjoy reading this special issue, and a big thank you to all our amazing authors who made this issue possible!

TABLE OF CONTENT

06

JAVAPRO

30 Years of Java - How the language has evolved

by Sebastian Hempel

50

CORE JAVA

Modern Java

by Ron Veen

11

JAVAPRO

Java through the decades: From Applets to Microservices

by Mihaela Ghorghie-Roman

57

AI

AI Tools for Jakarta EE

by Gaurav Gupta

14

JAVAPRO

Java: let's do a SWOT!

by Gireesh Punathil

62

AI

Securing the Future of AI: Authorization for Java RAG Systems using LangChain4j and OpenFGA

by Deepu Sasidharan

17

JAVAPRO

Revitalizing Legacy Code

by Chiara Civardi

67

AI

Build AI Apps and Agents in Java: Hands-On with LangChain4j

by Lize Raes

20

CORE JAVA

From Reactive Streams to Virtual Threads

by Adam Warski

78

AI

Langchain4J Musings

by Nicolas Fränkel

26

CORE JAVA

Brain-friendly programming: Why the latest language features matter

by Markus Westergren

83

FRAMEWORKS & API

Adapting Java for Modern APIs: REST, GraphQL, and Event-Driven Innovation

by Mihaela Ghorghie-Roman

37

CORE JAVA

Structured Concurrency

by Ron Veen

87

FRAMEWORKS & API

The long history of log4j

by Christan Grobmeier

43

CORE JAVA

Hitchhiker's Guide to Java Performance

by Ingo Düppe

91

FRAMEWORKS & API

A journey through 30 years of API evolution

by Vipin Menon

95

FRAMEWORKS & API

A Passwordless Future: Passkeys for Developers

by Deepu Sasidharan

120

DATA

Java and Databases: From SQL to Hibernate and NoSQL

by Franck Pachot

98

FRAMEWORKS & API

Best Practices for API Design in Java

by Muuath Bin Ali

125

DATA

A Look Back at Java's 30-year Journey with Databases

by Raphael De Lio & Brian Bodden

102

FRAMEWORKS & API

Under the Surface: An 'Iceberg' Journey through the Lesser-known Faces of Java

by Artur Skowronski

129

CLOUD

Platform Engineering 101 for Java developers

by Grace Jansen & Maaren Veandeperre

108

FRAMEWORKS & API

30 Years of Java, 25 Years of Enterprise Java

by Chiara Civardi & Steve Millidge

135

CLOUD

Greener Code: Sustainable Java Deployments with Native Builds and Knative Serverless on Kubernetes

by Mariu Stein & Vishal Shanbhag

111

FRAMEWORKS & API

Policy and Process: Thinking differently about Modern Java API design

by Steve Poole

142

PROJECT MANAGEMENT

From Silent Coder to Respected Leader

by Markus Westergren

116

FRAMEWORKS & API

EclipseStore: The Evolution of Jaca Persistence

by Markus Kett

151

PROJECT MANAGEMENT

Should we stop discussing about technical debt... with top management?

by Pablo Bouzada

JAVAPRO**Publisher:**JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
GermanyE-Mail: info@javapro.io
Website: <http://www.javapro.io>**Editor in Chief:**

Markus Kett (V.i.S.d.P.)

Editorial:info@javapro.io**Design, Layout & Print:**Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
GermanyCopyright (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."



#JAVAPRO

30 Years of Java - How the Language has Evolved

We are celebrating 30 years of Java. Time to look back at how Java as a programming language evolved since then. We will see how Java integrated different programming paradigms without losing its clear structure.

Author:



Sebastian Hempel is a self employed consultant and trainer from the north of Bavaria. Since 2003 he supports customers in developing and operating enterprise systems. His main focus lays in the development of systems in Java (JakartaEE and Quarkus) on Linux based systems.

In projects he is often responsible for the build- and deployment-systems. Beside his work as a software developer, he also is a trainer for everything about Java, Jakarta EE and the configuration management system Puppet.

<https://chaos.social/@sepp>

When Java arrived in the mid 90s, object-oriented programming was the dominant programming paradigm. This is reflected in Java, which was a „pure“ object-oriented language from the beginning. „Everything is an object“ can be taken literally. Except for primitives, everything is derived from the base class object. In the early days, the language strictly sticks to this paradigm. This changed over time as the language incorporated more and more elements from the functional programming paradigm. This is a general development that can also be seen in other programming languages.

This article will not describe each and every change that was made to Java since 1995. I will pick out the changes that are the most important for me and my daily work. This might leave out important development of the language in certain areas. I will also look at improvements in the standard library of Java. The library is an essential part and, for me, cannot be separated from the language itself.

Collections

The language itself supports the concept of an array. An array stores elements that can be accessed by an index. The size of an array is fixed and is given when creating the array. Arrays can be used to store primitives and objects.

To enable the work with „dynamic“ arrays that grow automatically when new elements are added, JDK 1.0 introduced the Vector class. The class is still available but should not be used in new projects. It was integrated into the collection framework introduced in Java 1.2 in 1998.

You create an instance of the Vector class and add elements with the addElement method. To add an element at a given index, the method insertElementAt can be used. The method ensures that the element is inserted at the given index and all existing elements are moved by one position.

To go through all elements of a Vector we get an Enumeration with the elements method. The enumeration can be used to walk through all elements using a while loop. With the method hasMoreElements we check if there are still elements not visited. With the method nextElement of the Enumeration we move to the next element.

```
package de.ithempel.java30;

import java.util.Enumeration;
import java.util.Vector;

public class VectorExample {

    public static void main(String[] args) {
        Vector v = new Vector();

        v.addElement („Duke“);
        v.addElement („JAVAPRO“);
        v.addElement („Java“);

        v.insertElementAt („Sebastian“, 1);

        v.remove(2);

        Enumeration elements = v.elements();
        while (elements.hasMoreElements()) {
            System.out.println(elements.nextElement());
        }
    }
}
```

Besides the Vector class, JDK 1.0 also introduced the class Dictionary to assign a key to the value. This allows the access of the element via a key. The class allows retrieving an Enumeration of all keys or all values of the dictionary. Reusing an existing key replaces the old with the new value. It is also possible to remove the value for the key.

```
package de.ithempel.java30;

import java.util.Dictionary;
import java.util.Enumeration;
import java.util.Hashtable;

public class DictionaryExample {
```

```
    public static void main(String[] args) {
        Dictionary dict = new Hashtable();

        dict.put („A“, „Duke“);
        dict.put („B“, „JAVAPRO“);
        dict.put („C“, „Java“);

        System.out.println („Value of B: „ + dict.get („B“));

        String oldValue = (String) dict.put („C“, „Sebastian“);
        System.out.println („Old Value of C: „ + oldValue);

        dict.remove („A“);

        Enumeration elements = dict.keys();
        while (elements.hasMoreElements()) {
            String key = (String) elements.nextElement();
            System.out.println (dict.get(key));
        }
    }
}
```

JDK 1.2 in 1998 introduced a completely new framework to work with collections. The framework supports different types of collections, like List, Set or Map to name only some.

The use of these new collections does not differ from the use of Vector or Dictionary. The method names are shorter. Instead of the Enumeration we now use an Iterator to walk through all elements of a collection. Internally, the implementation of the collections has a better performance. The Vector class synchronised every operation. This was dropped with the move to the collection framework.

```
package de.ithempel.java30;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ListExample {

    public static void main(String[] args) {
        List list = new ArrayList();

        list.add („Duke“);
        list.add („JAVAPRO“);
        list.add („Java“);

        list.add(1, „Sebastian“);

        list.remove(2);

        Iterator iter = list.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

Generics

We still have to cast elements we retrieve from a collection from the generic type Object to the concrete type we store in the collection. The type cast operation might lead to a ClassCastException at runtime. There is no possibility for the compiler to check for the correct type while creating the object code.

Java 5 in 2005 introduced the concept of generics. It allows extending the type system of Java. The collection framework supports

generics to specify the type of objects that are stored inside a collection. With generic, the compiler is able to check the type of the element at compile time.

We get a type safe iterator that returns the type of the element stored in a collection.

```
package de.ithempel.java30;

import java.util.ArrayList;
import java.util.List;

public class ListGenericExample {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();

        list.add(„Duke“);
        list.add(„JAVAPRO“);
        list.add(„Java“);

        list.add(1, „Sebastian“);

        list.remove(2);

        for (String elem : list) {
            System.err.println(elem);
        }
    }
}
```

But Java 5 not only brought us generics. It also introduced a new type of the for loop. The for each loop can iterate over each member of an array or an Iterable that every standard collection class implements. We don't have to check the hasNext method or use the next method to get the next element. Together with the Iterable interface now collections feel like a first class element of the language.

```
package de.ithempel.java30;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ListForEachExample {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();

        list.add(„Duke“);
        list.add(„JAVAPRO“);
        list.add(„Java“);

        list.add(1, „Sebastian“);

        list.remove(2);

        Iterator<String> iter = list.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

Java 7 in 2011 improved the type inference for the creation of generic instances. It is a small enhancement that eliminates some kind of duplication in the code. The type of the generic only has to be defined in the constructor. In the definition of the variable, the generic type could be left out.

```
List<> list = new ArrayList<String>();
```

Working with dates

Java has support for working with dates and times from the beginning. The class Date was introduced in Java 1.0. It is a kind of wrapper around the unix time value also taking into account time zones. It is also the class with the most long-lasting deprecated methods for decades. All constructors, getters and setters are deprecated in favour of the new class Calendar, that was introduced in Java 1.1 in 1997.

```
package de.ithempel.java30;

import java.util.Calendar;
import java.util.Date;

public class DateTimeExample {

    public static void main(String[] args) {
        Date now = new Date();

        Calendar cal = Calendar.getInstance();
        cal.setTime(now);

        int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
        int month = cal.get(Calendar.MONTH) + 1;

        System.out.println(„We have the „ + dayOfMonth + „ of the
        „ + month + „ month of the year“);
    }
}
```

The Calendar class can work with both timezones and the calendar system. After setting the time, we can use the get method to get parts of the timestamp like the month or the day of the month.

We had to wait until Java 8 in 2014 to get a complete new date and time API. Java integrates the previously available Joda-Time library into the JDK. The api introduces new classes for handling date and time in the current (local) timezone LocalTime, LocalDate and LocalDateTime. When working with timezones, we now have the classes ZonedDateTime. The class Instant can be used to work with exact timestamp in nanoseconds.

The classes support methods to do date calculation and calculation differences between date / times. There are also methods to get or set certain fields for a date or time. The interfaces of the new classes support chaining multiple operations (fluent interface).

```
package de.ithempel.java30;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;

public class DateTimeApiExample {

    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();

        LocalDateTime tomorrow = now.plusDays(1);
        LocalDateTime yesterday = now.minusDays(1)
    }
}
```

```

        .withHour(0).withMinute(0).withSecond(0);

    DateTimeFormatter dateTimeFormatter = DateTimeFormatter
        .ofLocalizedDateTime(FormatStyle.MEDIUM);
    System.out.println("Today: " +
        dateTimeFormatter.format(now) +
        ", yesterday: " + dateTimeFormatter
            .format(yesterday) +
        ", tomorrow: " + dateTimeFormatter.format(tomorrow));
}

```

The evolution of switch

The switch statement was there from the beginning. The first version could only be used with integers. It helped to better implement multiple if - else if - else constructs. The feature of the fall through was lent by other languages as C.

```

package de.ithempel.java30;

public class SwitchExample {

    public static void main(String[] args) {
        int value = 5;
        switch (value) {
            case 1:
                System.out.println("One");
                break;
            case 5:
                System.out.println("five");
                break;
            default:
                System.out.println("Unknown");
        }
    }
}

```

The first change was done in Java 7 in 2011. Beside integers, we can now use the switch statement to also select between different strings and enum values. This made the switch statement even more useful.

Many improvements of the switch statement and the now possible switch expression were introduced in Java 14 in 2020. Multiple case values can be provided in a single case. The biggest change was the possibility to use switch as an expression. You can return values from a switch block in multiple ways. There is a new statement yield to return a value, or you can use the arrow operator to define some kind of „mapping“.

```

package de.ithempel.java30;

public class SwitchExpressionExample {

    public static void main(String[] args) {
        String day = „Tuesday“;

        String typeOfDay = switch (day) {
            case „Monday“:
                yield „Weekday“;
            case „Tuesday“:
                yield „Weekday“;
            case „Wednesday“:
                yield „Weekday“;
            case „Thursday“:
                yield „Weekday“;
        }
    }
}

```

```

        case „Friday“:
            yield „Weekday“;
        case „Saturday“, „Sunday“:
            yield „Weekend“;
        default:
            yield „Unknown“;
    };

    System.out.println(typeOfDay);
}

```

Pattern Matching

Pattern matching was introduced in multiple steps. The feature of pattern matching was first implemented in function programming languages and soon implemented in other languages like Scala or F#. For Java, the first implementation of one kind of pattern matching was introduced with Java 14 in 2020.

Instead of only checking for a specific type with the instanceof operator you can now in one step assign the casted value to a new variable.

```

package de.ithempel.java30;

public class InstanceofExample {

    public static void main(String[] args) {
        Object obj = „Duke“;

        if (obj instanceof String) {
            String s = (String) obj;
            System.out.println(s.length());
        }

        if (obj instanceof String s) {
            System.out.println(s.length());
        }
    }
}

```

A much bigger step was to introduce pattern matching for switch with Java 21 in 2023. It allows cases to be selected based on the type of the argument. Beside the matching itself, the casted value is available as a new variable. After matching to a type, additional expressions can be used to refine the selection based on other conditionals.

```

package de.ithempel.java30;

public class PatternMatchingExample {

    public static void main(String[] args) {
        Object o = 42;

        String formatted = switch (o) {
            case null -> „Null“;
            case String s -> „String %s“.formatted(s);
            case Long l -> „long %d“.formatted(l);
            case Double d -> „double %f“.formatted(d);
            case Integer i when i > 0 // guarded pattern
                -> „positive int %d“.formatted(i);
            case Integer i when i == 0
                -> „zero int %d“.formatted(i);
            case Integer i when i < 0
                -> „negative int %d“.formatted(i);
            default -> o.toString();
        }
    }
}

```

```

    };

    System.out.println(formatted);
}
}

```

Not only with pattern matching but also with a normal switch statement we can add a case for null. This allows to work around a `NullPointerException` when giving a null value to a switch statement.

Records

In Java 16 from 2021 we get the possibility to define immutable data classes without having to write a lot of boilerplate code. With records, we only have to define the type and the name of the fields. All other code like equals, hashCode, toString, getters and public constructors are „created“ by the compiler for us.

Getters can be used like normal classes. The difference for getters is the missing get before each field name. We simply write the name of the field we want to access, like `name()` to get the name field of a record. Records can be enhanced by static variables and methods.

```

package de.ithempel.java30;

public class RecordsExample {

    public record Person(String firstName, String lastName, String address) {

        public String fullName() {
            return „%s, %s“.formatted(lastName, firstName);
        }
    }

    public static void main(String[] args) {
        Person duke = new Person(„Duke“, „Java“, „Javaland“);

        System.out.println(duke.lastName() + „“, „“ + duke.firstName());
        System.out.println(duke.fullName());
    }
}

```

As records are immutable - they have no setters - they are perfect to be used as data objects. In many current source code, the `lombok` library is used for that purpose. In newer projects, I myself try to replace `lombok` with records.

Stream Processing

One of the biggest changes in Java 8 from 2014 was the support of lambda expressions. Instead of defining anonymous inner classes, we now have a function as a new type. We can define functions / lambdas and pass them to other methods that call them.

Lambdas make use of functional interfaces. Interfaces of this type consist of one (abstract) function. There are many pre-defined functional interfaces in the package `java.util.function`. The annotations ensure, that we get a compilation error when we try to add another function to our interface.

```

package de.ithempel.java30;

public class LambdaExample {

    @FunctionalInterface
    public interface Foo {
        public String method(String s);
    }

    public static void main(String[] args) {
        Foo foo = parameter -> parameter + „ from lambda“;
        String result = add(„Duke“, foo);

        System.out.println(result);
    }

    private static String add(String s, Foo foo) {
        return foo.method(s);
    }
}

```

The Java library itself makes use of lambdas by providing a streaming api to work with a sequence of data. Streams are inspired by the map reduce programming model to process big data in parallel and distributed. The operations provided by streams are also known in functional programming. In java, a stream provides different operations. The operations are chained to a stream pipeline.

The streaming API provides operations like `filter` to select elements on certain conditions. With the `map` operation we can transform an element. Final operations can simply iterate over all elements like the operation `forEach`. There are also `Collectors` to collect all elements of a stream to a collection.

A pipeline can work on a stream in parallel. With the operation `parallel` we can distribute the work of the pipeline to multiple threads.

```

package de.ithempel.java30;

import java.util.stream.Stream;

public class StreamingExample {

    public static void main(String[] args) {
        Stream.of(„a“, „b“, „c“, „d“, „Duke“)
            .filter(e -> e.length() == 1)
            .map(e -> e.toUpperCase())
            .forEach(e -> System.out.println(e));
    }
}

```

The streaming api knows intermediate and terminal operations. Terminal operations trigger the execution of a stream pipeline. Intermediate operations are only executed, when a terminal operation is executed. This optimises the execution of a stream pipeline.

This was my short and quick evolution of the Java language. As already mentioned, I did not cover all changes in the language since the last 30 years. One big area I did not mention are the different mechanisms to manage multiple threads.

Looking at the current development I see a bright and interesting future for Java. There are a plenty of changes ahead. Since the change to a release train we constantly see previews of new features and changes in the language and the library.

Java through the Decades: From Applets to Microservices

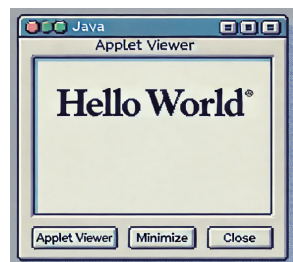
Java has undergone a remarkable transformation since its creation. It evolved from a language designed for interactive web-based Applets to a dominant force in enterprise development and cloud-native microservices. Java was originally created to enhance web interactivity. It soon evolved to support business applications, distributed systems, and modern architectures. This article explores Java evolution from Applets to microservices, highlighting key technological advancements and industry shifts. These changes, you are still able to program in Java the same way as 30 years ago.

Java Evolution from Applets to Microservices: A Technological Shift

Java's origins trace back to 1991 when Sun Microsystems initiated the Green Project to develop a platform-independent language. Officially released in 1995, Java introduced the concept of „Write Once, Run Anywhere“ (WORA), allowing applications to execute on any system with a Java Virtual Machine (JVM).

What were Applets?

Applets were small Java programs that ran inside web browsers. They made websites interactive, enabling animations, calculators, and dynamic graphics. Unlike standard web technologies at the time, which relied primarily on static HTML, Applets allowed rich, client-side functionality.



Evolution and Transition of Applets

By the late 1990s, Applets were widely used to enhance web functionality. However, they required the Java Plugin to run within browsers, which led to security vulnerabilities and performance issues. As browser vendors introduced stricter security policies, Applets faced increasing restrictions, making them less practical. By the early 2010s, advancements in web technologies such as JavaScript, HTML5, and CSS rendered Applets obsolete, prompting Java's evolution from Applets to microservices, shifting focus toward server-side and enterprise solutions.

Author:

Mihaela Gheorghe-Roman is a Software Architect at Systematic, Romania. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge at work, at the university, at different national and international IT conferences.



In her professional career, spanning over 13+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular.

<https://www.linkedin.com/in/mihaela-gheorghe-roman-6304915a/>

Java EVOLUTION in Enterprise Development: From Applets to Microservices

As Applets faded, Java emerged as a dominant force in backend enterprise applications. Java 2, released in 1998 and later rebranded as J2EE, provided a strong platform for scalable server-side development. Technologies like Servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJBs) helped businesses build powerful web applications without browser limitations.

Throughout the 2000s, Java became the backbone of enterprise computing. Major financial institutions, e-commerce platforms, and government agencies adopted Java for their critical applications.

Frameworks like Spring (2003) and Hibernate (2001) transformed enterprise development. They improved productivity, flexibility, and maintainability, paving the way for modern architectures.

Java EE 5, released in 2006, simplified development with annotations, removing the need for complex XML configurations. Jakarta EE replaced Java EE in 2017 after Oracle transitioned it to the Eclipse Foundation, ensuring continued innovation.

Open Source and Modernization

The rise of open-source software further propelled Java's evolution. The introduction of OpenJDK in 2007 fostered greater community-driven innovation, ensuring continuous enhancements in performance, security, and usability.

Significant advancements in Java's architecture followed, including improved memory management, concurrency handling, and functional programming features like Lambdas and Streams in Java 8 (2014). These enhancements laid the foundation for Java's next major transformation—microservices and cloud computing.

Further updates in Java 9 (2017) introduced the Java Module System, allowing better code encapsulation and modularization. Java 10 and later versions continued this trend with performance enhancements, garbage collection improvements, and new APIs for modern development needs.

From Monolith to Microservices: Java's Role in Cloud-Native Development

The shift toward cloud computing and distributed systems marked another pivotal milestone in Java's evolution. Traditional monolithic applications transitioned to microservices architectures, where independent services could be developed, deployed, and scaled separately. Java adapted to this paradigm with frameworks such as Spring Boot (2014), Micronaut (2018), and Quarkus (2019), simplifying microservices development.

Java's integration with containerization tools like Docker and orchestration platforms like Kubernetes reinforced its role in cloud-native applications. Performance optimizations, lightweight execution environments such as GraalVM, and reactive programming capabilities further cemented Java's relevance in modern software development.

Java has also embraced serverless computing, with support for AWS Lambda, Azure Functions, and Google Cloud Functions, allowing developers to create highly scalable applications without managing the underlying infrastructure.

From Microservices to Modern Distributed Applications

As microservices architectures became the norm, Java continued to evolve to meet new challenges in distributed computing. Large-scale applications required advanced data handling, high availability, and real-time processing capabilities, leading to the adoption of Java-based solutions for emerging fields such as:

- **Scalable Data Processing:** The transition from monolithic architectures to microservices led to the need for robust data streaming and processing solutions. Java's ecosystem embraced tools like Apache Kafka and Apache Spark, enabling real-time analytics and event-driven architectures that aligned with microservices.
- **Reactive Programming:** As cloud applications demanded increased responsiveness, frameworks like Project Reactor and Akka allowed Java developers to build non-blocking, event-driven microservices that efficiently handled asynchronous operations.
- **Edge and Serverless Computing:** With microservices forming the backbone of cloud applications, Java evolved to support edge computing environments, enabling lightweight, fast-startup applications using GraalVM's native image compilation.
- **Security in Microservices:** As Java moved from centralized monolithic applications to distributed microservices, security mechanisms like OAuth, JWT, and mutual TLS became essential in securing service-to-service communication in a decentralized environment.

The Future of Java Evolution: Cloud-Native and Microservices Advancements

As the software industry continues to evolve, Java remains at the forefront of technological advancements. The introduction of Project Loom, which aims to improve Java's concurrency model by introducing lightweight threads, promises better scalability and efficiency. Additionally, innovations in the JVM, such as Just-In-Time (JIT) compilation and Ahead-Of-Time (AOT) compilation, continue to enhance Java's performance.

The growing adoption of serverless computing has also led to new developments in Java's runtime environment. Frameworks like AWS Lambda and Azure Functions now support Java applications, enabling developers to build event-driven applications with reduced infrastructure concerns.

With continuous updates through Java's six-month release cycle, the language remains adaptable to emerging industry needs. Features such as pattern matching, records, and sealed classes improve code expressiveness, maintainability, and security.

Java's increasing support for cloud-native development through tools like OpenTelemetry for observability and GraalVM for faster startup times further strengthens its role in future architectures.

Conclusion

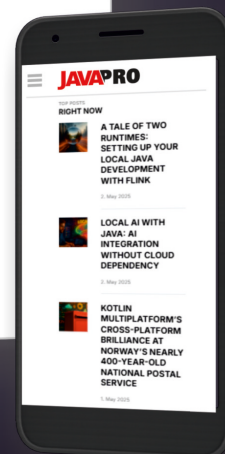
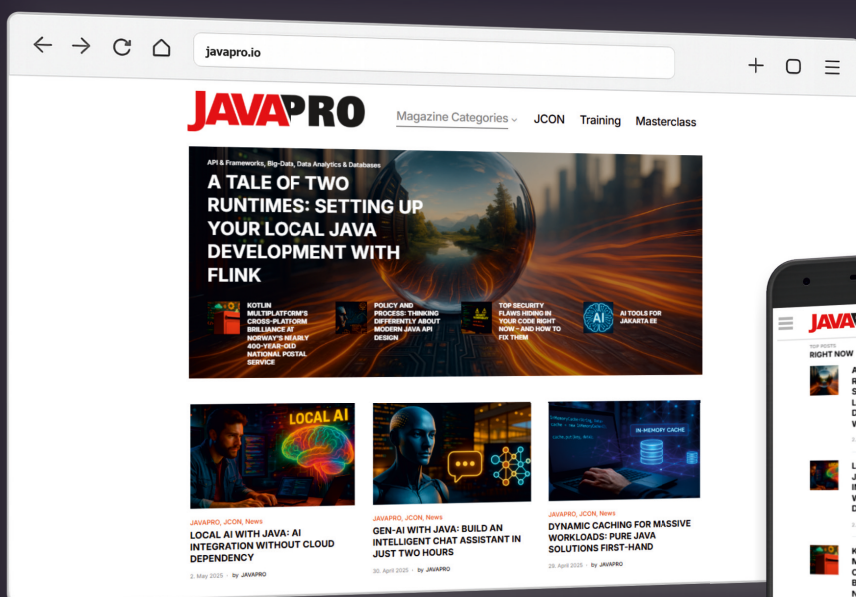
Java evolution from Applets to microservices underscores its adaptability to industry changes. By shifting focus to server-side applications, embracing open-source innovation, and optimizing for cloud-native architectures, Java continues to be a key player in modern software development.

The rise of microservices, serverless computing, and distributed architectures ensures Java's continued relevance. As businesses increasingly adopt cloud-first strategies, Java's role in shaping the future of software remains stronger than ever.

With continuous enhancements in performance, security, and scalability, Java is well-positioned to support the next generation of software development.

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



#JAVAPRO

Java: Let's do a SWOT!

„Java is to enterprise what electricity is to civilisation“.

Java has transcended “being just a programming language - it has become a cultural icon. From academic syllabuses to heated tech debates, and from viral social media memes to pop culture references in movies, Java’s influence is everywhere. Few languages have had such an incredible run, shaping not just software but also conversations, careers, and even family.

Java, a stalwart in the programming world, has not only stood the test of time but has also shaped the evolution of modern software

Author:



Gireesh Punathil is an IBM master inventor, Senior Technical Staff Member, committer or steering committee member for several leading open source projects. He is part of IBM Application Runtimes team that develops and supports websphere, java, eclipse and many other tools and frameworks. He spent a good share of his career in developing, porting and debugging platform tooling software: web servers, virtual machines, compilers, ide, monitoring agents and cloud components.

development. To the casual observer, it may seem like just another language, yet over the past three decades, Java has cemented itself as an integral part of the technology ecosystem. It has been the foundation for groundbreaking innovations and a cornerstone of academic research.

Despite its strengths, Java faces inherent challenges - particularly in adapting to modern workloads such as AI. However, with every challenge comes an opportunity. What are those? What hurdles does Java need to overcome to remain a dominant force in the ever-changing landscape of technology?

As Java marks an impressive 30-year milestone, let's conduct a candid SWOT analysis. While past assessments have explored selective and contextual aspects, this article takes a holistic approach - evaluating Java across multiple critical dimensions: language design, platform resilience, paradigm shifts, ecosystem evolution, community strength, and its broader business impact.

This comprehensive analysis provides a 360-degree perspective on Java's enduring relevance. The objective is to present an honest, data-driven evaluation that equips developers, organizations, and stakeholders with the insights needed for them to make informed decisions - helping them to leverage Java as a foundation for sustainable, long-term innovation in their focus areas.

Strength: the king of programming languages in the enterprise

„if luck favours the brave, natural selection favours the innovative!“

I would suggest Java was lucky that the timing of its birth and early adoption aligned perfectly with the rise of large-scale commercial software development. Born in 1995, right when software was starting to be used in large scale commerce, and enterprises were moving towards internet-based and distributed computing, Java became a natural choice. 1990s and early 2000s saw the rise of web applications that created demand for scalable, portable and robust programming frameworks. Java was the natural choice!

What do you mean by luck? Is it sheer coincidence that Java rose to fame? Are there other factors?

Yes. It is not just by chance. Its blend of abstraction and capabilities strike a perfect balance between developer productivity and system performance. Java sits at an interesting sweet spot: more abstraction than C/C++, such as intuitive object orientation, automatic memory manager, threading abstractions for improved concurrency management. At the same time it has more control than scripting languages such as strong typing.

Then comes the legendary mantra: „write once, run anywhere.“ In today's landscape, platform independence is no longer a differentiator - it's an expectation! Newer languages have adopted it as a given, but Java set the standard. Its seamless execution across platforms hasn't just been a benchmark; it has shaped the very foundation of enterprise software development.

Java's open and community-driven nature is a major strength, especially in the enterprise space. Initiatives like Jakarta EE, MicroProfile, OpenJDK, JCP, and Adoptium ensure continuous innovation, interoperability, and long-term stability. This collaborative model helps Java stay relevant, adaptable, and widely adopted across industries while providing vendor-neutral, high-quality, production-ready runtimes that reinforce trust and sustainability. Bottom line: When the ecosystem (developers, users, clients, tooling, frameworks ...) flourished in the enterprise greenfield through maintaining a symbiotic co-ordination, Java didn't look back, and surged forward with unstoppable momentum.

Weaknesses: Legacy baggage

„what do you call something caught between the two slices - existing workloads that demand stability and transformational workloads that require agility? Java sandwich“

Java's deep adoption in enterprise workloads was both a strength and a curse! While it provided the necessary stability for existing large workloads, it also slowed its ability to adapt to rapid, transformational shifts in workload demands. As cloud, AI, and startup-driven innovations pushed for paradigm changes, Java found itself torn between the need for agility and the pull of enterprise compatibility, making it struggle to fully meet the demands of either world.

Long release cycles and the rigid governance that was required to support the existing user base made Java slow to adopt features such as lightweight concurrency, fast startup, and efficient memory models rather than as extensions to the language. End result: Java was late to optimize for ephemeral (pipeline tasks) and heterogeneous functions (parallel tasks). Though there are recent efforts aiming to bridge these gaps, Java is still playing catch-up in an era where agility and adaptability are the success criteria.

Java's abstraction and garbage collection add overhead, making it slower than C++ in raw computational power. What does this mean? It makes the developer's life somewhat easy, but the processor feels the heat. Java's verbosity, inherent in the language semantics, creates a lot of boilerplate code; de-abstraction of such wrappers are in the way of computational efficiency. Ironically, what was once considered a great programming style now feels tedious to new developers, especially when compared to modern languages that are more efficient, intuitive, and lightweight. At runtime, where startup latency, memory efficiency, and resource consumption are critical, Java often loses out to modern alternatives that prioritize these factors. Instead, Java focuses more on steady-state throughput, which benefits certain workloads but limits its adaptability to modern, ephemeral, and resource-constrained environments.

Opportunities: reinventing for the future

„in nature, adversity drives mutation; in technology, it breeds innovation!“

The widespread adoption of Java in enterprises, such as in financial systems and large-scale business solutions, provides a strong foundation to build upon and a unique opportunity to expand its relevance. However, staying competitive in a rapidly evolving technology landscape requires more than just maintaining a presence; it demands groundbreaking and rapid innovation.

As Java already powers a significant portion of enterprise applications, organizations naturally prefer extending their existing infrastructure rather than migrating to entirely new ecosystems. For example, integrating AI capabilities into Java-

based business applications could open new frontiers. However, this requires fundamental shifts at both the language and platform levels. Going over and beyond functional programming elements, incorporating declarative paradigms could make Java more expressive while reducing boilerplate code.

Modern developers favor languages that offer high-level abstractions without runtime penalties. Innovations such as workload specific JIT optimizations, ahead-of-time (AOT) compilation of Java bytecode to native images, lightweight concurrency models, memory-efficient data structures, and efficient native data access are promising advancements in this space. Collectively, these enhancements can simplify development while ensuring high performance.

Despite these advancements, significant opportunities remain to establish Java as a first-class language for modern workloads. For example, JIT and GC could be further optimized to dynamically adapt to workload-specific characteristics. Additionally, ongoing efforts to improve resource efficiency and reduce startup times need further maturity and convergence to maximize impact. Technologies like fast startup using checkpoints, native images, and cloud-optimized JIT address performance and deployment challenges, making Java more competitive in modern workloads like cloud-native and serverless environments.

Moreover, debugging often requires more than twice the effort of initial development. Enhanced observability and diagnostic capabilities could serve as a critical differentiator, making Java more resilient and adaptable to highly complex modern workloads such as containerised applications with microservices.

Java's widespread adoption gives it a strong advantage - it has the trust and backing of enterprises. However, the real opportunity lies not just in maintaining its relevance but in leading the next wave of technological evolution.

Threats: the looming shadows

„Achilles - swift of foot but slow to wrath“

Newer programming languages are rapidly gaining traction in web development, cloud-native computing, and ai applications, rising stiff competition to Java. Many of these languages are designed with specific workload characteristics in mind, excelling in particular domains: whether it's Python for AI/ML, Javascript for web development, or Go for cloud-native microservices. Java, as a general-purpose language, faces increasing pressure from these specialized alternatives that offer more efficient, domain-specific solutions.

The declining mindshare in web development further threatens Java's relevance in that space. Modern frameworks in other languages provide greater flexibility, ease-of-use, and built-in support for large-scale automation, areas where Java has been slower to evolve. Additionally, the rise of user-friendly, low-code/no-code platforms - often built using non-java technologies - further reduces its appeal for rapid application development.

Another key concern is Java's licensing uncertainty. Frequent shifts in licensing policies by vendors have led to confusion and increased costs for enterprises, prompting many to explore entirely different technology stacks as alternatives. This uncertainty, combined with Java's perceived complexity relative to newer, more lightweight languages, contributes to its diminishing dominance in some areas.

The good news is that the Java community is actively engaged and giving the right attention and focus to address these concerns. Efforts like Project Panama for native integration, GraalVM for performance, CRUI/CRaC for fast startup, and AI-focused optimizations are gaining traction and attention. Conferences, Java user groups, and other developer meetups and hackathons provide excellent platform for discussing new features and gathering vital feedback, while working groups and forums define newer standards and propose enhancements to the language. This is indeed promising!

Conclusion: Java, the timeless warrior

Despite its strengths - platform independence, a vast ecosystem, and enterprise-grade stability - Java faces challenges in an evolving technology landscape. The rise of specialized languages, shifting development trends, and increasing competition in web, cloud, and AI domains are constantly challenging Java's dominance. While Java continues to adapt with new features, frameworks, and optimisations, its long-term relevance will depend on how well it balances innovation with its legacy constraints.

Java, much like a seasoned martial artist, has adapted through decades of change. While newer, flashier languages emerge, Java's reliability, robustness, and ecosystem ensure it remains a dominant force in software engineering. As the saying goes, "old is gold" - and Java, with its ability to evolve, is pure 24-karat.

The battle is far from over, and Java's next chapter might just be its most exciting yet. Will it rise to the challenge or fade into obsolescence? Only time will tell, but if history is any indication, Java isn't leaving the battlefield anytime soon.

"endure my heart, you have endured worse than this!"



JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



#JAVAPRO

Revitalizing Legacy Code - How to Modernize 30-Year-Old Java Apps for the Next 30 Years

Java has been the backbone of web and enterprise applications for 30 years, powering everything from banking systems to large-scale logistics platforms. Not only is the technology still widely used, but some of the earliest enterprise Java applications developed in the 1990s and early 2000s are still running, playing a key role in business operations. So, how can developers bring these essential 30-year-old enterprise Java applications into the future without disrupting critical business functions?

Author:

Chiara Civardi is a tech communicator with over 10 years of experience in producing technical content on a variety of topics related to the digital transformation. Her expertise includes automation engineering, robotics, software engineering and network technology. She holds a PhD from ETH Zurich and a MSc from the University of Southampton.



Over 10 years as technical writer, when doing the PhD presented at a lot of conferences (winning a number of presentation awards).

As we celebrate 30 years of Java, let's time travel back to the mid-1990s. I remember my father had just started a new job at a company providing IT outsourcing services to banks - or, as I used to explain to my classmates, "he works with computers." At the time, his new workplace was buzzing about the "cool Sun reps" and the new kid on the block: Java.

The promising „Write Once, Run Anywhere“ (WORA) magic was fascinating many software developers, who were eager to build robust enterprise applications based on this exciting new technology. Countless engineering teams across the world succeeded in establishing the Java language and its framework as the de facto standard for application development. These first applications set the foundation for digital business operations as we know them today.

Fast forward three decades, and my father is about to retire while these very same Java applications are still there, together with many other newer ones. In many cases, these earlier software solutions can be considered 'legacy,' as they were developed with older technologies or using outdated development paradigms. As such, these once cutting-edge applications are now often suffering from a number of issues, such as outdated architecture and security vulnerabilities. Moreover, they can be challenging and/or costly to maintain.

Bite the bullet: If it ain't broke... it may still need fixing

Legacy applications aren't necessarily broken. In fact, they remain critical to many day-to-day operations and are still functioning well—just not well enough for today's business and technology requirements.

An enterprise Java application written in the early 2000s on Java 2 Platform, Enterprise Edition (J2EE) or Java EE might still be effectively handling financial transactions or managing logistics. However, integrating it with modern tools and platforms, ensuring their security and next-level performance can be challenging if not impossible. Add to that the dwindling number of experts that were involved in the initial creation of the legacy applications and the limited pool of developers familiar with older Java versions, and you have a task that becomes harder to tackle with each passing year.

Because of the crucial role such legacy applications have, the question isn't whether they need modernization, it's how to do it while minimizing any interference with the entire system and associated business operations. Luckily, there are a number of ways to successfully upgrade.

The best approach to modernization should take into account business objectives, budget constraints and the risks involved. Some organizations may prioritize cost-effectiveness and opt for incremental updates, while others may focus on long-term scalability and choose a complete overhaul.

What to consider before modernizing

Understanding different migration approaches as well as how they can both address technical and business needs is at the backbone of any successful modernization. By having a clear picture of the current status and what's needed, companies can set up effective plans that align with strategic goals while maximizing returns on investment and mitigating potential disruptions.

When discussing the potential modernization of an enterprise Java application, it is important to inspect the software and define the potential technical challenges that may arise. These can be caused by legacy dependencies, outdated frameworks or monolithic architectures that can be difficult to decouple. Moving to modern frameworks can introduce complexities that require careful planning. Additionally, security vulnerabilities in older Java applications can present risks if they are not properly addressed during the modernization process.

Budget should also be discussed. While modernization comes with significant costs for infrastructure, licensing, training and potential downtime, there are multiple routes available, each involving different initial investments. In addition, organizations should weigh the short-term expenses against long-term gains, such as reduced maintenance costs, improved security and uptime as well as the ability to support future business growth.

Besides, cultural issues should be evaluated too. Developers may be keen to adopt the latest technologies, while business leaders may hesitate to invest in something that appears to be „working fine.“ Bridging this gap requires clear communication and extensive collaboration, so that all expectations are managed, and everyone understands the value of an up-to-date application.

In addition to assessing and building the right modernization culture, it is important to consider developer talent. Java remains one of the most popular programming languages and frameworks, but the expertise associated with enterprise Java applications written in the 1990s and early 2000s is dwindling. The original developers of legacy systems are no longer available to consult. Like my father, many of them are now enjoying their retirement.

This fact, combined with limited documentation from earlier systems, means that reverse engineering is often essential before updating an application. In addition, the software may rely on outdated frameworks and methodologies that current teams may not be experienced with. Having a clear picture of the existing capabilities and any potential gaps is fundamental to defining how to proceed.

By proactively discussing and planning these aspects, organizations can identify the most suitable modernization strategy and set the stage for a successful collaboration.

The many paths to modernization

For some businesses, the best approach is the least disruptive. In such cases, retention, also known as encapsulation, is ideal. It involves keeping the old system intact but wrapping its core features within one or more microservices that interact with newer applications offering more modern capabilities. As a result, it is possible to maintain the necessary functionalities without any rewrite while adopting innovative elements, such as cloud-based enhancements.

Another non-invasive option is rehosting, also known as lift-and-shift. It typically involves migrating an application from one physical environment, such as on-premises hardware, to another, namely a cloud infrastructure. The process focuses solely on the hosting environment and doesn't involve any significant changes to the application's underlying architecture or functionality.

While this approach offers a quick way to transition to a new environment and can partially help reduce the cost of maintaining aging hardware, it may not help developers and end users fully leverage the benefits of the cloud (or any other new platform). Therefore, they may not be able to benefit from key modernization opportunities, such as system optimization, new capabilities and further cost savings, that could be achieved through more extensive modifications.

A step in this direction can be achieved by opting to replatform legacy applications, which involves migrating the legacy application to a newer version of its runtime without making extensive changes to the code or its structure. This update helps companies take advantage of modern improvements while keeping the application largely intact. For instance, an application running on Java 5 and J2EE could be upgraded to Java 11 and Jakarta EE 8 with minimal modifications. While this might seem like a small change, the performance enhancements and security updates are substantial and can help minimize security risks.

Organizations can also combine replatforming with refactoring, which involves optimizing and restructuring an application internally, i.e. its source code, without impacting its external functionality and intended functions. The goal of this type of modernisation is to improve the internal performance and maintenance of the application without causing significant downtime. For example, moving J2EE applications to the Cloud can breathe new life into an aging system without having to invest in a complete rewrite, even if the updated solution doesn't take full advantage of the new environment.

When the goal of a modernization is shifting to a different architecture, e.g. from monolithic to microservices-based applications, then a rearchitecting is necessary. By breaking a monolithic Java application into microservices, companies can improve their systems' scalability, independence when it comes to deployment, as well as resilience in case of failures.

Rather than updating just part of a legacy enterprise Java application to adopt more modern architectures and innovations,

such as the cloud, organizations may have the resources needed to rebuild the application. This involves rewriting the entire code and it can help fully leverage the latest technologies at hand.

Finally, the most radical approach is the full replacement of the existing enterprise Java application. Starting from scratch with modern Java frameworks and technologies can certainly help companies interested in staying up to date with the latest solutions and trends. However, while this guarantees a fresh start with the latest capabilities, it also demands significant investments in terms of cost, time and resources. Organizations must carefully weigh whether the benefits of a ground-up replacement justify the disruption.

Although the modernization approaches discussed here vary, certain strategies can help ensure success regardless of the chosen path. For example, adopting a phased approach, even when rebuilding or replacing legacy applications, can help minimize disruptions. Gradually introducing new technologies, e.g. through incremental updates and proof-of-concepts, allows teams to adapt at a manageable pace and make sure that the entire system and associated applications continue to work smoothly.

Adopting a modernization mindset

Java and enterprise Java applications from the 1990s and early 2000s are the backbone of a multitude of today's digital systems. As such, keeping them up to date is paramount to ensure optimum operations, delivering high performance, efficiency and uptime while maximizing user satisfaction.

While application modernization is inevitable, the goal of such activity isn't to blindly update old code or adopt any new technology available. Successful modernizations are first and foremost strategic, as they balance innovation with pragmatism. Not every system needs a microservices overhaul, nor does every modernization effort require a full rewrite. With careful planning and the right strategy, even a 30-year-old Java application can be ready for the next 30 years.

From Reactive Streams to Virtual Threads

Virtual Threads deliver a fast, cheap (in terms of memory & switching speed) threading solution for the JVM. They promise the return of a direct-style, synchronous programming model. But is that enough to challenge the status quo regarding data streaming? Can we have the best of both worlds: the simplicity that Virtual Threads promise and the resilience and safety of Reactive Streams? Let's find out!

Virtual threads: the story so far and what's ahead

First, a little bit of history. Virtual Threads were released with Java 21 in September 2023, after 6 years of development. Some final limitations of that implementation will be fully lifted with the release of Java 24 in March 2025.

Virtual Threads are part of a broader initiative called Project Loom. This initiative also covers related features, specifically the Structured Concurrency and Scoped Values APIs, which are currently in the preview phase. Virtual Threads are predominantly a feature of the JVM (the virtual machine); however, there are also some API changes in the standard library. Hence, all JVM-based languages can use the new lightweight concurrency model, starting with Java but also Kotlin, Scala, Clojure, and others.

Why introduce Virtual Threads at all?

Let's explore the motivation for introducing Virtual Threads to understand their relation to reactive streams better.

A long time ago, Java had a synchronous programming model. For example, when implementing an HTTP server, each request was assigned a thread, which read the data, processed it, and wrote the response.

In older Java versions, every `java.lang.Thread` corresponded to an operating-system-level thread. These „platform threads,“ as they are called in Virtual Threads nomenclature, are a heavy-weight resource. First, they take a non-trivial amount of memory; second, creating a thread and switching the CPU's execution context to it takes a non-trivial amount of time. Hence, a limited number of threads can be created and used (in the order of thousands).

With the rise of the web, increased traffic and vast amounts of data to process, caused this model to run into scaling issues. With threads being a scarce, expensive resource, it just didn't make sense for them to sit idle and wait synchronously for responses from the database or other web services.

That's how thread pools, `ExecutionContexts` and the like came to life, with the aim to utilize threads better. Instead of writing code in a direct style, we shifted to working with various flavors of Futures. For example, a blocking, synchronous version of some car-selling business logic might look as follows:

```
var person = db.findById(id);
if (person.hasLicense()) {
    bankingService.transferFunds(person, dealership,
    amount);
    dealerService.reserveCar(person);
}
```

Author:



Adam Warski is one of the co-founders of SoftwareMill, where he primarily codes using Java, Scala, and other interesting technologies. He is actively involved in open-source projects, such as Ox, Tapir, sttp, Quicklens, ElasticMQ, and others. He has also been a speaker at major conferences, including JavaOne, Devvxx, GeeCON and ScalaDays.

In addition to writing closed- and open-source software, Adam spends his free time exploring various (functional) programming-related subjects.

<https://www.linkedin.com/in/adamwarski/>

The same code, but utilizing threads more efficiently, using `CompletableFutures`, looks rather different:

```
db.findById(id).thenCompose(person -> {
    if (person.hasLicense()) {
        return bankingService.transferFunds(person,
            dealership, amount)
            .thenCompose(transferResult -> dealerService.
                reserveCar(person));
    }
    return CompletableFuture.completedFuture(null);
})
```

Quite clearly, technical concerns (thread utilization) have taken priority over the code's readability. Project Loom set out to maintain the better thread utilization we achieved with Futures while retaining the readability and simplicity of the direct-style code.

More specifically, Project Loom aims to solve three problems that have been introduced with the shift to asynchronous code as above:

- Syntax: use direct-style, readable, synchronous syntax as the default way of using Java; this includes using Java's built-in control flow constructs instead of library code
- Virality: if we call a method that returns a Future, our method should probably return a Future as well (as blocking until the future is ready defeats its purpose!); this is also known as the function coloring problem
- Lost context: exceptions reported when using Futures often capture only part of the full stack trace, since the last future transformation; this makes debugging difficult, if not impossible at times

And it did so successfully!

Futures and Virtual Threads: under the covers

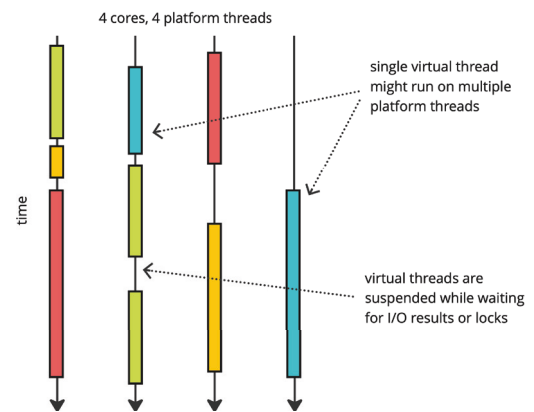
It might also be beneficial to understand how futures, and Virtual Threads, achieve better thread utilization.

Behind each executor is a pool of threads, which are created upfront. These threads run tasks from a task queue. When code is submitted to the executor service, a Future value is returned to the user, which will be completed once the submitted code is (eventually) run.

Whichever thread is free picks up and executes a task from the queue. Running such code might involve submitting more tasks, transforming other futures, and returning the futures created that way.

That's also precisely how Virtual Threads work under the covers. However, while the executors above are implemented at the library level, Project Loom shifts that and implements it at the VM level.

The JVM transparently manages a pool of threads for submitting and executing tasks. At the same time, the Thread API is kept intact: an instance might now represent a platform or a virtual thread.



However, the Virtual-to-Platform thread scheduler is not Project Loom's only contribution to the JVM. The second crucial component is retrofitting all blocking APIs to be Virtual-Thread aware. Any time you call a blocking method, such as acquiring a semaphore or a lock, sending data over a socket, or reading from an input stream, the underlying platform thread is not blocked.

Instead, only the virtual thread is blocked and put aside until a permit/lock is available or the operation's result is ready. The platform thread can then run other virtual threads that are ready for execution.

Some exceptions to the above retrofitting (specifically, synchronized methods) will be lifted in Java 24.

How did we arrive at Reactive Streams?

Let's now shift our attention to Reactive Streams. The initiative to develop a standard for asynchronous, non-blocking stream processing started in late 2013. The final version was released in 2015 and included as part of Java 9 in 2017.

Reactive streams aim to „govern the exchange of stream data across an asynchronous boundary.“ This exchange—or rather, a series of exchanges—should be done within bounded memory.

The JVM interfaces that are part of reactive streams (Publisher, Subscriber, and Subscription) are quite low-level and are not meant to be used directly by end-users. Instead, libraries that implement the reactive streams standard should be used.

Such libraries provide high-level interfaces for stream processing and tools for declaratively managing concurrency, interfacing with I/O operations, and safely handling errors.

Examples of reactive streams libraries include Akka Streams, Vert.x, Helidon, RxJava, Reactor (used in Spring) and others.

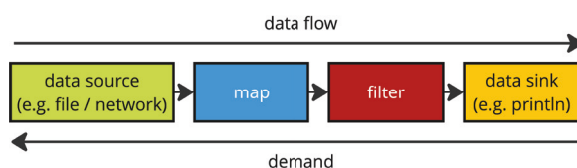
What problems do Reactive Streams solve?

As mentioned above, Reactive Streams allow the processing of large amounts of data in a streaming way. Libraries implementing reactive streams are often based on Futures or similar concepts, leveraging thread pooling and efficiently utilizing operating system resources as described above.

There are two classes of problems that are inherently hard, and reactive streams excel at providing robust and readable solutions for them.

The first area is concurrency. Conventional wisdom says that writing concurrent code using locks is hard and error-prone. Declarative concurrency helps, as the end-user only has to describe the „what,“ not the „how.“ Details of the (concurrent) execution are delegated to the runtime.

The second area is error management. Errors can happen in the most unexpected places, and ensuring proper resource management in case of an exception is often the centerpiece of many system implementations (or a source of bugs!). Reactive streams also help here, providing facilities for describing how errors should be handled, managing the lifecycle of resources safely, and recovering from errors, if possible.



Finally, a problem that reactive streams set to address is ensuring that data is processed within bounded memory. This is implemented using backpressure. An upstream component can only send data for further processing once it receives demand from downstream components. This bi-directional flow, data from upstream to downstream and demand from downstream to upstream, ensures that there is always a bounded number of elements that are being processed at any given time.

Implementing simple streams

While delivering a fast and powerful solution for data streaming, reactive streams implementations also suffer from the problems we have identified before: virality of Futures, lost context in case of errors, and high syntax overhead due to the need to compose asynchronous computations, not simple statements.

Can we leverage what Project Loom delivered, keeping the benefits of Reactive Streams? It turns out that yes, it is possible. We've set out to create such an implementation, to verify our approach, taking inspiration from Go and Kotlin.

The code lives in a project called Jox, and is available under the Apache2 OSS license. We'll briefly describe its architecture, but if you'd like to find out how direct-style „reactive“ streaming feels in practice, I encourage you to try the library!

To implement a „reactive“ data processing library in the spirit of Project Loom, we want to use as many constructs that are built into Java as possible. That is, ifs, fors, whiles, and try-catch-finally for error handling.

We also want to use direct syntax to describe the logic in the data processing stages. Operations should be composed using ;, without needing to be wrapped in a container.

A single data transformation pipeline processing stage is represented using a FlowStage class. A stage can be run, provided a sink to which data can be emitted:

```
public interface FlowStage<T> {
    void run(FlowEmit<T> emit) throws Exception;
}

public interface FlowEmit<T> {
    void apply(T t) throws Exception;
}
```

We also introduce a Flow wrapper class, which contains the last stage defined so far and a number of helper operations that we'll use to build the data processing pipeline:

```
public class Flow<T> {
    final FlowStage<T> last;
    // ...
}
```

It turns out it's all we need to describe quite complex flows! For example, here's a factory method which produces an infinite stream of values:

```
public class Flows {
    public static <T> Flow<T> iterate(T zero,
        Function<T, T> nextFn) {
        return new Flow(emit -> {
            T current = zero;
            while (true) {
                emit.apply(current);
                current = nextFn.apply(current);
            }
        });
    }
}
```

The Flow instance created by the method above only describes what should happen—no data is processed at its invocation. To start processing, we need to provide a FlowEmit instance to the last stage, which might vary depending on how we want to consume this stream.

But before we get to consumption, you might notice that we're already following some of Loom's principles: using Java's built-in control flow while construct and direct-style syntax.

Once we have that, here's a method that runs the stream and creates a list of all elements processed:

```
public class Flow<T> {
    public List<T> runToList() throws Exception {
        List<T> result = new ArrayList<>();
        last.run(result::add);
        return result;
    }
}
```

Looks simple enough! We also need some stream-transformation

methods. `map` is a popular one:

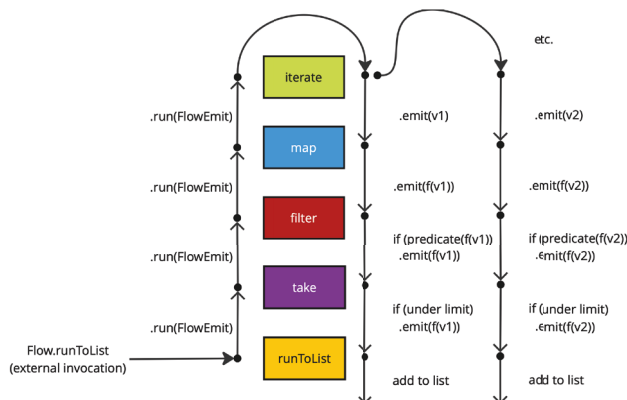
```
public class Flow<T> {
    public <U> Flow<U> map(ThrowingFunction<T, U>
        mappingFunction) {
        return new Flow<>(emit -> {
            last.run(t -> emit.apply(mappingFunction.
                apply(t)));
        });
    }
}
```

We simply emit the result of applying the mapping function to whatever the previous (then-last) stage produced! Similarly, we can implement methods such as `filter`, `take`, `mapStateful`, `sliding`, `runFold`, `runDrain`, etc. With these tools in place, we can now build synchronous, single-threaded data processing pipelines:

```
List<Integer> result = Flows
    .iterate(1, i -> i + 1)
    .map(i -> i*2)
    .filter(i -> i%3 == 2)
    .take(10)
    .runToList();
```

How does this work? As mentioned before, when we invoke the `iterate`, `take`, and `map` methods, we only create a description of the flow to run. `Flow` is a lazily-evaluated description of how to process a stream. Each new `Flow` being constructed contains a reference to a `FlowStage`, which encloses over the previous stage, and so on.

Only when we invoke `.runToList` is the data actually processed. This method creates a sink—a `FlowEmit` instance—and passes it to the previous stage. Here, the sink simply adds items to a list. The previous stage then passes a modified sink—another `FlowEmit` instance—to the `run` method of the previous stage.



Ultimately, the producing stage calls `FlowEmit.emit`. Each invocation of `emit` sends the data through the entire pipeline. Hence, each `emit` call only completes once that element has been fully processed (or discarded). We thus limit processing to at most one element at a time (for now)—trivially meeting the memory-boundedness requirement.

We do have a nice stream-processing API, but so far, we could have just used Java's streams to a similar effect:

```
List<Integer> result = Stream
    .iterate(1, i -> i + 1)
    .map(i -> i * 2)
    .filter(i -> i % 3 == 2)
    .limit(10)
    .collect(Collectors.toList());
```

It's when we want to introduce concurrency, multi-Flow operations, and I/O when it gets interesting.

Implementing asynchronous streams

As we already mentioned, concurrency is inherently hard. Using locks all too easily leads to deadlocks, and trying to run things in parallel often ends with a once-in-a-while race condition. That's why the best concurrency is when somebody else manages it for us. Then, it's no longer our concern.

Reactive streams have proven great in providing tools to avoid explicit concurrency by including a number of declarative operators. Can we implement something similar as well?

It turns out that—yes, we can! First, we'll need low-level primitives to communicate and transfer data between concurrently running virtual threads.

We might try using Java's blocking queues, however we'll quickly run into some limitations. Luckily, prior art is available, giving us just the tools we need. Specifically, the Go programming language has shown that using channels to communicate between goroutines (which resemble Virtual Threads in many aspects!) is a convenient and powerful way to solve concurrency problems.

Moreover, Kotlin has successfully implemented performant Go-like channels on the JVM, albeit using coroutines. Luckily, they published their algorithm in a paper, paving the way for a Virtual-Thread-based implementation, now available as part of Jox.

Compared to a blocking queue, a `Channel` has two major differences:

- It can be marked as done—meaning no more elements will be sent, or put in an „error“ state, which discards any buffered element short-circuits any send/receive operations.
- A select operation is available, which allows selecting exactly one value to be received from (or sent to) a list of channels.

The second infrastructural component, which will be crucial when discussing error handling, is structured concurrency scopes. Discussing structured concurrency would be an article in itself; hence, we'll provide a very brief overview.

There are several ways to implement structured concurrency, some of which are explored in JEP-453. This Java feature is currently in preview and has undergone a significant revision between Java 22 and Java 23. The community hopes it will be finalized in Java 25—we'll see!

Jox provides an alternative structured concurrency API (although behind the scenes, using the one from the JEP), which focuses on safety and simplicity of use. The base unit is a scope within which forks—Virtual Threads—can be spawned:

```
var result = supervised(scope -> {
  var f1 = scope.fork(() -> {
    Thread.sleep(500);
    return 5;
  });

  var f2 = scope.fork(() -> {
    Thread.sleep(1000);
    return 6;
  });

  return f1.join() + f2.join();
});
```

The core property of a structured concurrency scope is a guarantee that once the supervised scope completes, all forks started within it have finished either successfully, with an exception, or due to an interruption. Hence, from the outside world's point of view, threading, and concurrency become an implementation detail!

The fact that no threads can outlive the scope is crucial in case of an error in one of the forks. The default behavior is for the scope to shut down, interrupting any other forks still running and waiting until they are complete.

We can implement concurrent streaming operators now that we have those two tools. Let's start by merging two streams. Just as before, we are creating a lazily evaluated flow description. Once the merged flow is run, it creates a structured concurrency scope and, within that scope, two forks that run the two child flows. Each flow produces its output to a channel.

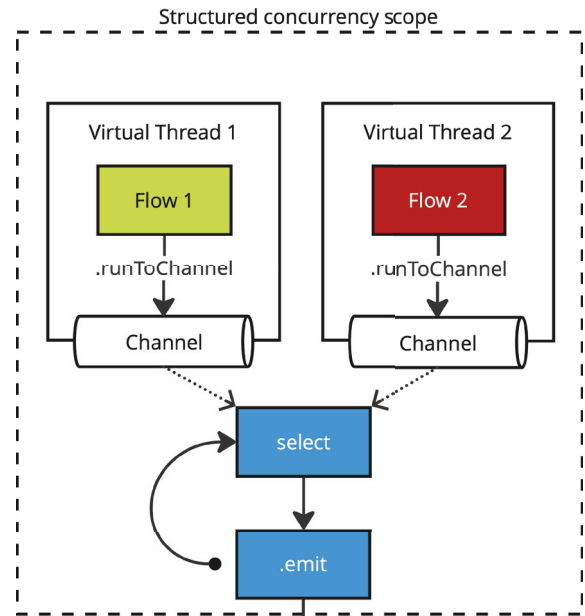
We can then use select to receive values from the channel that has data available (some simplifications are in place regarding handling stream completion):

```
public class Flow<T> {
  public Flow<T> merge(Flow<T> other) {
    return new Flow<>(emit -> {
      supervised(scope -> {
        Channel<T> c1 = this.runToChannel(scope);
        Channel<T> c2 = other.runToChannel(scope);

        boolean continueLoop = true;
        while (continueLoop) {
          switch (selectOrClosed(c1.receiveClause(), c2.receiveClause())) {
            case ChannelDone _ -> continueLoop = false;
            case ChannelError error -> throw error.toException();
            case Object r -> emit.apply((T) r);
          }
        }
      })
    })
  }
}
```

```
return null;
});
});
}
```

Note that we are once again using Java's built-in control flow constructs (while), Java's built-in pattern matching (switch), and Java's built-in error handling (throw).



In a similar way (albeit with more code), we can implement streaming operators such as mapPar, zip, buffer, flatMap, grouped, etc. These operators take inspiration from the features available in Reactive Streams libraries such as Akka Streams or Reactor.

Moreover, we now have the tools to interface with the external world, creating byte streams from files, parsing them as lines, or running to obtain an InputStream.

To conclude, here's a longer example using some of the functionalities mentioned above:

```
Flows.unfold(0, i -> Optional.of(Map.entry(i + 1, i + 1)))
  .throttle(1, Duration.ofSeconds(1))
  .mapPar(4, i -> {
    Thread.sleep(5000);
    var j = i * 3;
    return j + 1;
  })
  .filter(i -> i % 2 == 0)
  .zip(Flows.repeat("x"))
  .runForeach(System.out::println);
```

Backpressure

This all sounds great, but is our implementation on par when it comes to resiliency qualities of reactive streams, such as guaranteeing that memory usage is bounded? After all, in the implementations that

e've inspected, backpressure propagation is nowhere to be seen?

The truth is that backpressure is there; however, it's implicit; there's no need to create explicit code paths to manage it.

Backpressure arises from limited buffers (channels, queues) and virtual thread blocking. That is, if some stage runs in a background fork (thus asynchronously), and the downstream can't keep up with processing the data, eventually the channel's buffer will fill up, and the `.send` invocation will block the thread—naturally pausing production in the upstream stage.

Hence, backpressure is provided by the fact that we use thread-blocking operations.

Error handling

The final facet of virtual threads we must consider is error handling. In the case of a simple, synchronous stream, error handling is pretty straightforward: any exception thrown by a processing stage will propagate through the `.emit` call chain and be rethrown by whatever consumption method we use (such as `.runForeach` or `.runDiscard`).

Java's error-handling mechanisms are sufficient here. Convenience methods such as `Flow.onError` or `Flow.onComplete` simply create stages that use try-catch or try-finally.

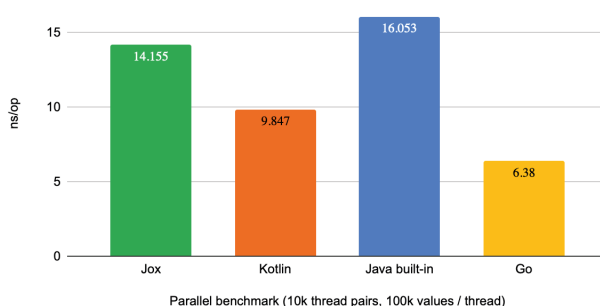
The situation is only slightly different for asynchronous streams, which spawn multiple forks (threads) and run flows in the background. That is thanks to the crucial structured concurrency principle we've discussed: if any fork throws an exception, the whole scope is shut down, and the exception (appropriately wrapped) is rethrown, but only when all forks are complete after being interrupted.

Hence, from the point of view of other flow stages, it makes no difference whether some other processing stage is asynchronous or not. Threading becomes an implementation detail (although an important one).

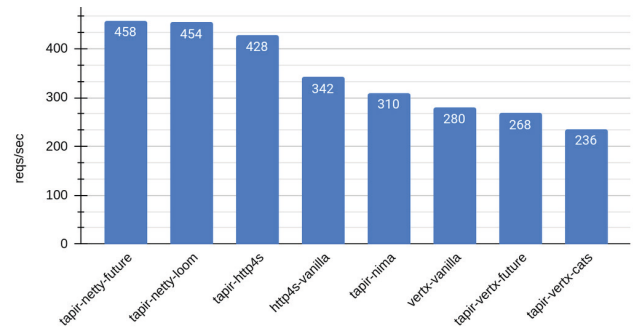
Performance

Functionality-wise, we can replicate at least large parts of reactive stream libraries. But what about performance?

We have yet to design benchmarks for both approaches to streaming, so we have only partial data at our disposal. First of all, we've benchmarked, profiled, and optimized the implementation of the channel that we're using for communicating between asynchronously running stages; as a result, it's on par with Java's concurrent data structures and Kotlin's implementation, slightly lagging behind Go's.



Secondly, we've looked at the performance of Loom-based and Reactive-libraries-based HTTP servers. In our tests, we didn't notice any significant differences.



That shouldn't come as a surprise: the core idea of using a pool of OS-level platform threads to execute multiple „virtual“ threads (coroutines, futures, etc.) is the same in the „reactive“ and „Loom“ approaches.

Summary

Can you implement a Virtual-Thread-based library with feature parity of Reactive Streams implementations? Yes. Jox is a case in point.

Will Virtual Threads replace Reactive Streams? Partially. The Reactive Streams specification is still very useful as an interoperability standard. I suspect that it will continue to play an essential role in implementing low-level, performant integration code. For example, Jox implements the reactive streams Publisher interface to integrate with any third-party library that is reactive-streams-aware.

However, when it comes to writing data processing pipelines, the popularity of reactive stream libraries will most likely diminish. The Virtual Threads model is simpler, both when writing and reading code. It allows using Java's control flow constructs and error model (exceptions) and avoiding function coloring (Future virality). It also doesn't need a library-level runtime, which manages the pool of threads on which execution happens, instead shifting that concern to the JVM.

Libraries such as Akka Streams or Reactor still have an edge when it comes to maturity, API versatility, and integration availability. But the Virtual Threads ecosystem will eventually catch up and, if my predictions are correct, take over. Virtual Threads provide a robust, performant platform that allows us to build applications leveraging the knowledge gained from operating „reactive“ systems and expanding upon it, providing a more programmer-friendly way of writing resilient systems.

In a way, the emergence of Virtual Threads is the ultimate validation of the ideas behind Reactive Streams, which proved so valuable that they've been incorporated into the JVM itself.



#JAVAPRO #CoreJAVA

Brain friendly Programming: Why the latest Language features matter

Author:



Markus Westergren is a staff engineer with over 20 years of experience. During his work with large backend projects, Markus has repeatedly seen the big impact code quality has on the outcome of the projects and the careers of the developers working on them. Because of that, Markus focuses on helping professional Java back-end developers write maintainable code so that they become sought-after experts in the software development industry.

<https://twitter.com/markuswesterg1>
<http://www.empatheticdev.com>

From Skepticism to Trust: A Developer's Journey

Remember your first encounter with Java? For me, it was 1998, wrestling with Java 1.1 in a university distributed systems course. Coming from a background in assembler and C, I was skeptical. Garbage collection? It felt almost too good to be true. After years of meticulously tracking every allocated memory block, the idea that the runtime would handle memory management seemed like dangerous magic. Java felt slow, memory-hungry, and, frankly, untrustworthy.

But that skepticism would transform into appreciation as Java evolved to become increasingly brain-friendly.



The Cognitive Evolution of Java

Interestingly, this evolution from „dangerous magic“ to trusted companion mirrors Java’s broader journey over the past three decades. While other languages have come and gone, Java has endured and thrived largely because of its commitment to cognitive ergonomics - making complex programming tasks more manageable for human minds.

Throughout this journey, each major release has introduced features that reduce mental overhead: from garbage collection freeing us from memory management, to generics eliminating type-casting cognitive burden, to records and pattern matching making data handling more intuitive. This consistent focus on developer cognition hasn’t just kept Java relevant; it has made it a language that grows with its developers, supporting them as they tackle increasingly complex challenges in cloud computing, microservices, and enterprise systems. As we celebrate Java’s 30th anniversary, this cognitive-friendly evolution stands out as perhaps its greatest contribution to software development.

Cognitive Benefits in Practice: A Real-World Example

Let me share a story that illustrates this evolution. In the mid-2000s, I joined a team maintaining a large enterprise application written in Java 1.4. One of my first assignments was to introduce generics as we migrated to Java 1.6. It was a massive undertaking that revealed a critical cognitive burden we had been carrying: Every time we encountered a List in the codebase, we had to hunt through the code to figure out what it contained.

```
// Before generics - cognitive burden
List users = getUserList();
for (Object obj : users) {
    User user = (User) obj; // Hope this doesn't
    throw ClassCastException!
    // Process user
}

// After generics - intent is clear
List<User> users = getUserList();
for (User user : users) { // Our brain can focus
    on the business logic
    // Process user
}
```

This migration wasn’t just about type safety—it was about making the code’s intent immediately obvious to our brains. We discovered several potential bugs during this process, hidden in the ambiguity of raw types. Each fix felt like removing a cognitive tax we hadn’t even realized we were paying.

The Science Behind Code Comprehension

Miller’s law in psychology states that the average number of items a person can hold in their short-term memory, working memory, is

7 ± 2 items. This fact has major implications on us programmers. Naturally, we have to take this into consideration when we write code so that it will be possible for a reader to understand it.

In „Pragmatic Thinking and Learning: Refactor Your Wetware“, Andy Hunt explores how our brains process information and learn new concepts. His insights on cognitive processing and the importance of understanding our mental models are particularly relevant to software development.

In „The Programmer’s brain“, Dr. Felienne Hermans talks about how our brain works when we are working with code.

She states that our working memory can only keep track of 3 - 5 chunks of information at the same time. How big a chunk is depends on how much experience we have with the current task. The more exposure we have had to a certain area, the more our brain is able to recognize patterns and offload our working memory.

Working Memory and Code Chunks

Have you ever looked at some code and instinctively felt that something felt off without being able to pinpoint the cause of that feeling? This is most likely your brain recognizing a pattern and identifying an anomaly. However, it’s only after you have done some debugging that the cause of the problem becomes clear.

We are writing code for others to read and occasionally for machines to execute. We have to be mindful of the fact that the person reading our code maybe don’t have the same experience as we do and will therefore have a higher cognitive load trying to make sense of what we have created.

A chunk can be an individual statement, lines or entire design patterns. Usually a chunk gets bigger the more experience we have. If you are new to programming, the for-loop might take several chunks of your working memory while a more experienced developer can recognize the whole thing as a single chunk.

Fortunately, we can make our code easier to understand, to lower the cognitive load of the reader, by limiting the number of chunks of information we have to keep track of.

There is much we can do as developers, use meaningful names, don’t mix levels of abstractions, keep methods and classes focused on one thing, and so on.

We can also take advantage of the evolution of Java and the new features that help us write code that lowers the cognitive load of the reader.

Evolution of Java Through a Cognitive Lens

Java’s evolution over the past three decades reveals a fascinating pattern: each major release has systematically reduced cognitive load while increasing expressive power. Let’s explore this evolution through the lens of cognitive science.

The Early Days: Simple But Verbose (Java 1.0 - 1.4)

Initially, early Java prioritized simplicity and clarity over conciseness. This matched the cognitive needs of its time:

```
// Java 1.1 - Explicit iteration
Vector elements = new Vector();
elements.addElement("first");
elements.addElement("second");

for (int i = 0; i < elements.size(); i++) {
    String element = (String) elements.
        elementAt(i);
    System.out.println(element);
}
```

While verbose, this code had benefits:

- Every operation was explicit
- Control flow was immediately obvious
- No hidden complexity

However, it imposed significant cognitive load:

- Type casting required mental tracking of actual types
- Error handling was manual and verbose
- Collections couldn't express their content types
- Boilerplate code cluttered the essential logic

The Type Safety Revolution (Java 5)

Subsequently, Java 5 introduced generics, a feature that dramatically reduced the mental overhead of type tracking:

```
// Before generics - Mental overhead
List users = getUserList();
Iterator it = users.iterator();
while (it.hasNext()) {
    User user = (User) it.next(); // Mental note:
    this list contains Users
    processUser(user);           // Hope we
    remembered correctly!
}

// After generics - Compiler enforces our mental
// model
List<User> users = getUserList();
for (User user : users) {       // Type safety
    processUser(user);          // No mental
    tracking needed
}
```

Key cognitive improvements:

- Enhanced for-loop aligned with natural thinking about iteration
- Generics moved type safety from runtime to compile time
- Autoboxing/unboxing removed mental conversion between primitives and objects
- Static imports reduced namespace mental overhead

The Expressiveness Era (Java 8)

Most notably, Java 8's streams and lambdas represented a quantum leap in cognitive alignment. Code that had been a maze of nested loops and conditional statements could now be written in a way that matched our mental model of the data flow:

```
// Traditional imperative style - Multiple mental
// contexts
List<Transaction> highValueTransactions = new
    ArrayList<>();
for (Account account : accounts) {
    if (account.isActive()) {
        for (Transaction transaction : account.
            getTransactions()) {
            if (transaction.getValue() > 1000) {
                highValueTransactions.
                    add(transaction);
            }
        }
    }
}

// Stream style - Matches our mental data flow model
List<Transaction> highValueTransactions = accounts.
    stream()
        .filter(Account::isActive)
        .flatMap(account -> account.getTransactions().
            stream())
        .filter(transaction -> transaction.getValue() >
            1000)
        .collect(toList());
```

While verbose, this code had benefits:

- Every operation was explicit
- Control flow was immediately obvious
- No hidden complexity

However, it imposed significant cognitive load:

- Type casting required mental tracking of actual types
- Error handling was manual and verbose
- Collections couldn't express their content types
- Boilerplate code cluttered the essential logic

The Type Safety Revolution (Java 5)

Subsequently, Java 5 introduced generics, a feature that dramatically reduced the mental overhead of type tracking:

```
// Before generics - Mental overhead
List users = getUserList();
Iterator it = users.iterator();
while (it.hasNext()) {
    User user = (User) it.next(); // Mental note:
    // this list contains Users
    processUser(user);           // Hope we
    // remembered correctly!
}

// After generics - Compiler enforces our mental
// model
List<User> users = getUserList();
for (User user : users) {       // Type safety
    // guaranteed
    processUser(user);          // No mental
    // tracking needed
}
```

Key cognitive improvements:

- Enhanced for-loop aligned with natural thinking about iteration
- Generics moved type safety from runtime to compile time
- Autoboxing/unboxing removed mental conversion between primitives and objects
- Static imports reduced namespace mental overhead

The Expressiveness Era (Java 8)

Most notably, Java 8's streams and lambdas represented a quantum leap in cognitive alignment. Code that had been a maze of nested loops and conditional statements could now be written in a way that matched our mental model of the data flow:

```
// Traditional imperative style - Multiple mental
// contexts
List<Transaction> highValueTransactions = new
ArrayList<>();
for (Account account : accounts) {
    if (account.isActive()) {
        for (Transaction transaction : account.
            getTransactions()) {
            if (transaction.getValue() > 1000) {
                highValueTransactions.
                add(transaction);
            }
        }
    }
}

// Stream style - Matches our mental data flow model
List<Transaction> highValueTransactions = accounts.
```

```
stream()
    .filter(Account::isActive)
    .flatMap(account -> account.getTransactions().
        stream())
    .filter(transaction -> transaction.getValue() >
        1000)
    .collect(toList());
```

Cognitive benefits:

- Declarative style focuses on „what“ not „how“
- Method chaining matches natural data transformation thinking
- Immutable operations reduce state tracking
- Common patterns (filter, map, reduce) become visual

The Data Modeling Renaissance (Java 14+)

The journey continued through Java 17's records and Java 21's sealed interfaces and pattern matching. These features enabled something I never imagined back in my C programming days: truly data-driven design that the compiler could verify. Consider this modern Java code:

```
// Traditional approach - High cognitive load
public class UserStatus {
    private final String status;
    private final String message;
    private final LocalDateTime timestamp;

    // Constructor, getters, equals, hashCode,
    // toString...
    // (40+ lines of boilerplate)
}

// Modern approach - Zero cognitive overhead
public record UserStatus(
    String status,
    String message,
    LocalDateTime timestamp
) {}

// Pattern matching with sealed interfaces
public sealed interface LoginResult {
    record Success(User user, String token)
        implements LoginResult {}
    record Failure(String reason) implements
        LoginResult {}
    record MfaRequired(String temporaryToken)
        implements LoginResult {}
}

// Data-driven flow control
String message = switch (loginResult) {
    case Success(var user, var token) ->
        „Welcome back, „ + user.name();
    case Failure(var reason) ->
```

```

        „Login failed: „ + reason;
    case MfaRequired(var token) ->
        „Please enter your MFA code“;
};

```

This code aligns perfectly with how our brains naturally categorize and handle different cases. The compiler ensures we've handled every possibility, reducing the mental overhead of „what if“ scenarios.

Furthermore, modern features reduce cognitive load through:

- Records eliminating data class boilerplate
- Pattern matching expressing natural categorization
- Switch expressions handling all cases explicitly
- Sealed interfaces ensuring exhaustive handling

Small Changes, Big Impact on Cognitive Load

Notably, while major features like records and pattern matching get most of the attention, Java's evolution has included numerous smaller enhancements that have significantly reduced cognitive load. Here are some examples:

Stream.toList() (Java 16)

```

// Before - Mental overhead:
// „Which collector should I use? What's
// the difference between toList() and
// toUnmodifiableList()?“
List<String> names = users.stream()
    .map(User::getName)
    .collect(Collectors.toList());

// After - Single chunk: „Convert stream to list“
List<String> names = users.stream()
    .map(User::getName)
    .toList();

```

Helpful NullPointerExceptions (Java 14)

Better error messages reduced debugging cognitive load:

```

// Given this code:
user.getAddress().getStreet().toUpperCase()

// Before - Mental overhead: „Which part was null?“
// NullPointerException: null

// After - Clear indication:
// NullPointerException: Cannot invoke „String.
// toUpperCase()“
// because the return value of „Address.
// getStreet()“ is null

```

Text Blocks (Java 15)

Simplified multi line string handling:

```

// Before - Mental overhead: Line endings,
// escaping, indentation
String query = „SELECT u.name, u.email „ +
    „FROM users u „ +
    „WHERE u.active = true „ +
    „ AND u.lastLogin > ?“;

// After - What you see is what you get
String query = „““
    SELECT u.name, u.email
    FROM users u
    WHERE u.active = true
    AND u.lastLogin > ?
    „““;

```

Enhanced Switch Expressions (Java 14)

Made switch statements more intuitive and safer:

```

// Before - Mental overhead: Break statements,
// fall-through cases
String message;
switch (status) {
    case PENDING:
        message = „Still waiting“;
        break;
    case APPROVED:
        message = „Good to go“;
        break;
    case REJECTED:
        message = „Not accepted“;
        break;
    default:
        message = „Unknown status“;
}

// After - Direct mapping
String message = switch (status) {
    case PENDING -> „Still waiting“;
    case APPROVED -> „Good to go“;
    case REJECTED -> „Not accepted“;
    case UNKNOWN -> „Unknown status“;
};

```

Type Inference for Local Variables (Java 16+)

Incremental improvements to var usage:

```

// Before - Mental overhead: Repetitive type
// information
HashMap<String, List<String>> map = new
    LinkedHashMap<String, List<String>>();

```

```
// After - Let the compiler handle it
var map = new LinkedHashMap<String, List<String>>();
```

Collections Factory Methods (Java 16+)

Simplified immutable collection creation:

```
// Before - Mental overhead: Multiple steps,
mutability concerns
List<String> list = Collections.
unmodifiableList(Arrays.asList("a", "b", "c"));

// After - Direct intention
List<String> list = List.of("a", "b", "c");
```

Case Rules for Switch (Java 17+)

Simplified case label rules:

```
// Before - Mental overhead: Remember colon syntax
switch (obj) {
    case Integer i:
        // handle integer
        break;
    case String s:
        // handle string
        break;
}

// After - Consistent, familiar syntax
switch (obj) {
    case Integer i -> handleInteger(i);
    case String s -> handleString(s);
    case null -> handleNull();
}
```

Collectively, these smaller changes demonstrate Java's commitment to reducing cognitive load across all aspects of the language. While each improvement might seem minor in isolation, their cumulative effect is substantial:

1. Reduced Mental State Tracking

- Fewer special cases to remember
- More consistent patterns
- Clearer error messages

2. More Natural Expression

- Code structure closer to mental models
- Reduced boilerplate

- More intuitive syntax

3. Better Development Flow

- Fewer context switches
- Reduced debugging time
- More focused problem-solving

The impact of these changes reminds us that cognitive load improvement isn't just about big features. Sometimes, it's the small refinements that make our daily coding experience more productive and enjoyable.

However, this journey hasn't been without challenges. Nevertheless, one persistent struggle has been convincing all team members to embrace these cognitive-friendly features. Developers comfortable with familiar patterns sometimes resist change, even when it would make their code more comprehensible. It's a reminder that the most significant barriers to better code aren't always technical—they're human.

The Future: Project Valhalla and Beyond

Looking ahead, project Valhalla represents perhaps the most significant attempt yet to align Java's type system with developers' mental models. Its features address several cognitive burdens that Java developers have carried for decades:

```
// Current Java - Mental overhead of reference vs
primitive types
Integer wrapperNum = 42;           // Is this on heap
or stack?
int primitiveNum = 42;             // Different
behavior, same concept
List<Integer> numbers = ...        // Boxing/unboxing
mental tax

// With Valhalla's primitive objects
primitive class Point {
    double x;
    double y;
}
List<Point> points = ...           // No mental
overhead - works like primitives
Point p1 = new Point(2, 3);
Point p2 = new Point(2, 3);
boolean same = (p1 == p2);        // True - just
comparing values
```

1. Identity-free Objects

- Current Java forces developers to constantly think about object identity versus value equality
- Valhalla's primitive classes eliminate this mental overhead - they're simply values, like integers

2. Specialization

- Currently, generic code forces developers to maintain two mental models: one for primitive types and another for reference types
- Valhalla's specialization unifies these models

3. Universal Generics

- Eliminates the cognitive overhead of remembering which types can be used with generics
- No more mental tracking of boxing/unboxing operations

This aligns perfectly with our brain's preference for consistent, predictable patterns. Just as records eliminated the cognitive overhead of writing data classes, Valhalla will eliminate the cognitive overhead of dealing with Java's type system fundamentals.

Impact on Development Practice

Consequently, this evolution has profoundly affected how we write and maintain code:

1. Code Review

- Less time spent on boilerplate
- Focus on business logic
- Easier to spot potential issues

2. Maintenance

- More self-documenting code
- Fewer places for bugs to hide
- Better alignment with business requirements

2. Specialization

- Currently, generic code forces developers to maintain two mental models: one for primitive types and another for reference types
- Valhalla's specialization unifies these models

3. Team Dynamics

- Shorter on-boarding time
- More productive code reviews
- Easier knowledge sharing

4. Testing

- More focused test cases
- Better expression of test intentions
- Reduced test boilerplate

The key lesson from Java's evolution is that cognitive-friendly features don't just make code easier to write—they make it easier to think about, discuss, and maintain. Each advancement has brought the language closer to how developers naturally reason

about their problems.

Real-world Impact

Our working memory can only handle 3-5 chunks of information simultaneously. This cognitive limitation directly impacts how we understand and write code. As a result, by applying cognitive-friendly practices, we reduce the number of chunks developers need to hold in memory at once.

Key Benefits

Faster On-boarding Through chunk reduction

- New developers need to maintain fewer concepts in working memory
- Each piece of code communicates its intent clearly
- Natural progression from simple to complex concepts

```
// Before: Multiple chunks (raw types, casting,
error handling)
List items = getItems();
for (Object item : items) {
    try {
        DataItem dataItem = (DataItem) item;
        process(dataItem);
    } catch (ClassCastException e) {
        handleError(e);
    }
}

// After: Single conceptual chunk (processing
items)
getItems()
    .forEach(this::process);
```

Improved Understanding Through Structured Information

- Code organized in logical, digestible pieces
- Clear separation of concerns
- Pattern matching aligns with natural thinking

```
// Before: Multiple mental contexts to track
public Response handleRequest(Request request) {
    Object result = processRequest(request);
    if (result instanceof String) {
        return new SuccessResponse((String)
result);
    } else if (result instanceof Error) {
        return new ErrorResponse(((Error) result).
getMessage());
    } else if (result instanceof Exception) {
        return new ErrorResponse(„Processing
```

```

failed");
    }
    return new ErrorResponse("Unknown result
type");
}

// After: Natural pattern matching
public Response handleRequest(Request request) {
    return switch (processRequest(request)) {
        case Success(var message) -> new
SuccessResponse(message);
        case Failure(var reason) -> new
ErrorResponse(reason);
        case Error(var message) -> new
ErrorResponse(message);
    };
}

```

Reduced Bug Occurrence Through Cognitive Load Reduction

- Fewer mental juggling acts means fewer mistakes
- Compiler can verify more of our intentions
- Clear state transitions and error handling

```

// Before: Multiple opportunities for bugs
public void processPayment(Payment payment) {
    if (payment.getAmount() > 0) { // Possible
NullPointerException
        if (payment.getMethod() != null) {
            if (payment.getMethod().
equals("CREDIT")) { // Magic string
                // Process credit payment
            } else if (payment.getMethod().
equals("DEBIT")) {
                // Process debit payment
            }
        }
    }
}

// After: Explicit, safe handling
public sealed interface PaymentResult {
    record Success(BigDecimal amount) implements
PaymentResult {}
    record Failure(String reason) implements
PaymentResult {}
}

public PaymentResult processPayment(Payment
payment) {
    return switch (payment.method()) {
        case CREDIT ->
processCreditPayment(payment);
        case DEBIT -> processDebitPayment(payment);
    };
}

```

Business Impact of Reduced Cognitive Load

1. Faster Development Cycles

- Less time spent understanding code
- Quicker code reviews
- Reduced debugging time

2. Higher Quality Code

- Fewer bugs making it to production
- More consistent implementations
- Better test coverage due to clearer structures

3. Improved Team Dynamics

- More confident developers
- Better knowledge sharing
- Reduced friction in code reviews

Ultimately, by consciously managing the number of chunks developers need to hold in memory, we create code that's not just easier to write—it's easier to understand, maintain, and evolve. This alignment with our cognitive limitations leads to measurable improvements in development efficiency and code quality.

Navigating the Pitfalls of Modern Java Features

While Java's modern features can significantly reduce cognitive load, their misuse can actually increase it. Here are key considerations when modernizing your codebase:

Records: Not for Every Class

Records are powerful, but they're not suitable for all data-holding classes. Watch out for:

```

// Problematic use of record - business logic
doesn't belong here
record User(String name, String email) {
    public String getDisplayName() {
        return name.split(", ")[0]; // Business
logic in a record
    }

    public boolean isValidEmail() {
        return email.contains("@"); // Validation
logic belongs elsewhere
    }
}

// Better approach - separate data from behavior
record UserData(String name, String email) {}

class UserService {
    public String getDisplayName(UserData user) {
        return user.name().split(", ")[0];
    }
}

```

```

    public boolean isValidEmail(UserData user) {
        return user.email().contains("@");
    }
}

// Or let User be a class that contains both the
// data and the behavior

```

Pattern Matching: Clarity vs. Complexity

Pattern matching can make code more readable, but overuse can lead to confusion:

```

// Overly complex pattern matching - trying too
// hard to use the feature
String result = switch (obj) {
    case String s when s.length() > 10 &&
s.startsWith("A") ->
        s.substring(0, 10);
    case String s when s.length() > 5 ->
        s.substring(0, 5);
    case String s ->
        s;
    case Integer i when i > 100 ->
        i.toString();
    case Integer i ->
        "0" + i;
    default ->
        "unknown";
};

// Clearer approach - separate complex conditions
String processString(String s) {
    if (s.length() > 10 && s.startsWith("A"))
        return s.substring(0, 10);
    if (s.length() > 5) return s.substring(0, 5);
    return s;
}

String result = switch (obj) {
    case String s -> processString(s);
    case Integer i -> i > 100 ? i.toString() : "0"
+ i;
    default -> "unknown";
};

```

Var: Readability vs. Brevity

Type inference can reduce clutter, but it can also hide important information:

```

// Problematic use of var - type information is
// valuable here
var result = service.processData(input);
var transformed = result.transform(); // What are

```

we working with?

```

// Better - explicit types for public APIs and
// complex transformations
ProcessResult result = service.processData(input);
TransformedData transformed = result.transform();

```

Streams: Readability vs. Conciseness

While streams can make code more declarative, complex chain operations can become harder to understand:

```

// Overly complex stream with multiline lambda
List<Order> results = orders.stream()
    .filter(order -> {
        if (order.getStatus() != Status.COMPLETED)
            return false;
        if (order.getTotal() < 100) return false;
        return order.getItems().size() > 2;
    })
    .sorted(Comparator.
        comparing(Order::getCreatedDate))
    .collect(toList());

// Better - extract complex predicate to named
// methods
List<Order> results = orders.stream()
    .filter(Order::isHighValue) // Intent is clear
    .filter(Order::isCompleted)
    .sorted(Comparator.
        comparing(Order::getCreatedDate))
    .toList();

// The predicate logic in well-named methods
public class Order {

    // rest of class omitted for brevity

    public boolean isHighValue() {
        return total >= 100 && items.size() > 2;
    }

    public boolean isCompleted() {
        return status == Status.COMPLETED;
    }
}

```

Remember:

- New features should reduce, not increase, cognitive load
- Clear code is better than clever code
- Not every new feature needs to be used everywhere
- Consider team experience and maintenance implications
- Maintain consistent patterns across your codebase

By being mindful of these pitfalls, you can ensure that your adoption of modern Java features truly serves its purpose: making code more maintainable and easier to understand.

Conclusion

By understanding how the brain processes information, developers can create applications that are more intuitive, efficient, and less cognitively demanding for users. This approach can improve the overall user experience, increase productivity, and promote greater accessibility for individuals of all cognitive abilities.

Embracing brain-friendly programming will not only benefit individual programmers and users but has the potential to revolutionize the way we interact with technology and shape the future of computing.

The goal isn't to make code as simple as possible, but to ensure that its complexity serves a purpose. Every line of code should justify its cognitive cost.

We can get help achieving this by upgrading to the latest Java versions and taking advantage of the features they provide.

Are your application running on the latest Java, or at least the latest LTS version? If not, try and find out the reason and if there is a way to upgrade.

Do you know about all the features that your current Java version offers? There are lots of smaller enhancements that don't get as much publicity as the bigger ones but they can be quite useful.

Immediate Actions

1. Audit Your Current Java Version

- Check your application's current Java version
- List the features you're missing out on by not being on the latest version or LTS
- Create a migration plan focusing on cognitive-friendly features first

2. Start Using Records

- Identify your data classes (classes that are primarily getters/setters)
- Convert one class to a record and measure the reduction in code
- Document the improved readability for your team

3. Implement Cognitive Checkpoints

- Add a „cognitive load“ section to your code review template
- Questions to ask:
 - How many levels of nesting does this code have?
 - How many variables need to be tracked at once?
 - Could this code be restructured to reduce mental overhead?

Team-Level Changes

1. Update Your Style Guide

- Add guidelines for maximum method complexity
- Define standards for when to use newer Java features
- Include examples of before/after refactoring for cognitive load

2. Knowledge Sharing

- Schedule regular team sessions to explore new Java features
- Create a „cognitive smells“ catalog specific to your codebase
- Share refactoring wins and their impact on maintainability

Long-term Goals

1. Measure and Monitor

- Track cognitive metrics in your codebase:
 - Cyclomatic complexity
 - Number of parameters per method
 - Depth of inheritance
- Set improvement targets for these metrics

2. Continuous Learning

- Follow JEP (JDK Enhancement Proposal) discussions
- Experiment with preview features in non-production code
- Share knowledge about cognitive impact with the broader Java community

Remember: The goal isn't to make code as simple as possible, but to ensure that its complexity serves a purpose. Every line of code should justify its cognitive cost.

Take Action Now

Audit Your Current Codebase (Week 1)

- Run `java -version` across all your applications and services
 - Create a spreadsheet documenting:
 - Current Java version for each application
 - Target upgrade version (latest LTS or latest version)
 - Estimated effort
 - Potential cognitive benefits (e.g., „Can convert 12 data classes to records“)
- Pick one small service as your pilot upgrade project

Create Your Feature Migration Checklist (Week 1-2)

Map out opportunities to use modern Java features:

```
// Find candidates like these:
List<String> names = users.stream()
    .map(User::getName)
    .collect(Collectors.toList());
// Convert to:
List<String> names = users.stream()
    .map(User::getName)
    .toList();

// Identify data classes like:
public class UserPreference {
    private final String userId;
    private final Theme theme;
    private final Language language;

    // Constructor, getters, equals, hashCode...
}
// Convert to:
public record UserPreference(String userId, Theme
theme, Language language) {}
```

Set Up a Learning Environment (Week 2)

- Create a sandbox project with the latest Java version
- Add examples of each modern feature you plan to adopt
- Include before/after comparisons for your team
- Document cognitive load improvements for each change

Measure Current Cognitive Load (Week 2-3)

Pick three complex classes in your codebase and:

- Count the number of fields team members need to track
- Measure cyclomatic complexity using tools like SonarQube
- Document the number of implicit assumptions in the code
- Create a „cognitive heat map“ highlighting areas needing attention

Implement Changes Incrementally (Week 3-4)

Start with high-impact, low-risk changes:

- Convert simple data classes to records
- Replace `Collectors.toList()` with `toList()`
- Enhance error handling with pattern matching
- Update switch statements to use the new syntax

Document and Share Results (Week 4+)

Create a „Modern Java Patterns“ guide for your team:

- Before/after examples from your own codebase
- Measured improvements in:
 - Lines of code
 - Cyclomatic complexity
 - Code review time
 - Bug density

- Guidelines for when to use each feature
- Common pitfalls to avoid

Create a Feature Adoption Schedule

Week-by-week plan for introducing modern features:

- Week 1: Records and compact constructors
- Week 2: Enhanced switch expressions
- Week 3: Pattern matching for instanceof
- Week 4: Text blocks and String enhancements
- Week 5: Sealed classes and interfaces

Remember to:

- Start small with contained changes
- Measure before and after metrics
- Document team feedback
- Share learnings in code reviews
- Celebrate cognitive load reductions!

By following these steps, you'll not only modernize your codebase but also create a more maintainable, understandable, and enjoyable development environment. The key is to make incremental improvements while continuously measuring their impact on cognitive load and team productivity.

www.javapro.io

www.javapro.io





#JAVAPRO #CoreJAVA

Structured Concurrency

Structured Concurrency was the second delivery that came out of Project Loom. It was first introduced in Java 19 as a preview API. It aimed to simplify multithreaded programming by treating multiple tasks running in different threads as a single unit of work, thereby streamlining makes the code more readable, streamlining error handling, improving reliability, and enhancing observerability.

In this article, we will look at:

- Look at what structured concurrency is.
- Look at an example using structured concurrency.
- Dive into the underlying API with StructuredTaskScope.

What is Structured Concurrency

Structured concurrency allows developers to reason about concurrent code by defining precise points where execution splits into multiple tasks and where those tasks subsequently

Author:

Ron Veen is a highly experienced software engineer who has seen it all, from Midrange Systems to Micro Services.



Through all this, he is guided by his passion for software engineering and architecture. For more than 15 years he has been working on the JVM and the Java ecosystem and has seen all the frameworks and libraries from Apache to ZK. He is an Oracle Certified Java Programmer (OCP) and Sun Certified Business Component Developer (SCBCD/OCPCD).

Ron is an avid fan of alternative JVM languages like Groovy, Scala, Clojure, and his favorite, Kotlin.

At Team Rockstars IT he is a Special Agent and senior developer who frequently speaks at international conferences.

<https://www.linkedin.com/in/ron-veen>

merge. It brings the same level of organization to concurrency that for loops and if branches bring to structured programming. It achieves this by restricting the lifespan of concurrent operations to specific scopes. Similar to how variables within a if statement in structured programming are bound to that block, concurrent operations within a structured scope are terminated upon exiting that scope. This ensures that threads do not linger, preventing memory leaks and wasted CPU resources.

Imagine a task that can be broken down into smaller subtasks and those can, in turn, be further divided. This creates a nested, tree-like structure of tasks requiring execution. The entire structure's lifetime is confined to the scope of the code block in which it is created. In Java's structured concurrency API, which is encapsulated in the `StructuredTaskScope` concept we'll dig into in the following section. Upon closing this scope, it's assured that all tasks and their respective subtasks have either finished or been canceled.

Basic example of StructuredTaskScope

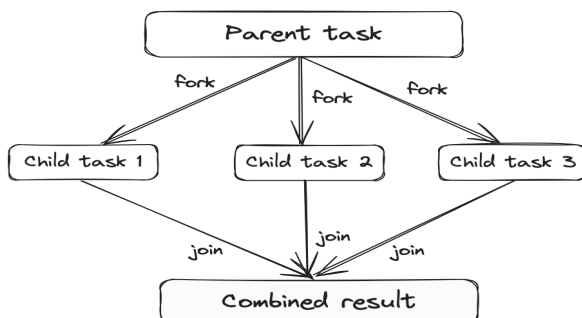
To demonstrate the fundamental use of `StructuredTaskScope`, consider this simple example. Keep in mind that this is primarily for educational purposes. Real-world applications frequently involve using specific subclasses that override the `handleComplete` method or potentially define your custom `StructuredTaskScope` subclass.

```
try (var scope = new StructuredTaskScope<String>())
{
    Supplier<String> currentSubtask = scope
        .fork(this::getCurrent);
    Supplier<String> forecastSubtask =
        scope    .fork(this::getForecast);
    Supplier<String> historySubtask =
        scope.fork(this::getHistory);

    scope.join();

    return new Result(currentSubtask.get(),
        forecastSubtask.get(),
        historySubtask.get());
}
```

If we put this code into a schema, it would be something like this



To start, we instantiate a `StructuredTaskScope` within a try-with-resources block. This is crucial as `StructuredTaskScope` implements `AutoCloseable`, ensuring the `close` method is automatically called when the block finishes.

We then launch three tasks concurrently using the `fork` method, which returns a `SubTask` object. For type safety and clarity, it's recommended casting this to a `Supplier`, given that `SubTask` extends it.

Following this, we invoke `join`, which blocks the current thread until all tasks within the scope are completed or the scope is shut down.

Once the tasks are finished, we can access their results by calling the `get()` method on each `SubTask`.

The core pattern of structured concurrency unfolds as follows:

1. Establish the task scope, the root for all subtasks.
2. Launch any number of subtasks within this scope.
3. The scope's creator thread joins the scope (and implicitly all subtasks).
4. The creator thread blocks until all subtasks finish.
5. The creator thread manages any errors.
6. The scope closes automatically upon completion.

Subtasks

We need to discuss the `SubTask` class, specifically its role as the return value of the join methods. While the join methods, due to `SubTask`'s implementation of `Supplier`, effectively return a `Supplier`, the language designers have purposefully steered developers towards casting the result to `Supplier`. This design choice encourages treating the result as a data provider, rather than relying on the specific `SubTask` implementation. It's important to note that `SubTask` is a sealed interface.

```
public sealed interface Subtask<T> extends
    Supplier<T> permits SubtaskImpl
```

The `SubTask` interface utilizes a `State` enumeration to track the completion status of its associated task. This enumeration has three states: `UNAVAILABLE`, indicating the task is either incomplete or was completed after the scope was shut down; `SUCCESS`, signifying successful completion with a result available; and `FAILED`, indicating completion with an exception and no result.

If a `SubTask`'s state is `SUCCESS`, the result can be obtained through the `get()` method. Attempting to call `get()` on a `SubTask` with a state of `UNAVAILABLE` or `FAILED` will result in an `IllegalStateException`. Conversely, if the state is `FAILED`, the exception thrown during execution can be retrieved using the `exception()` method. Calling `exception()` on a `SubTask` with a state of `UNAVAILABLE` or `SUCCESS` will also throw an `IllegalStateException`.

Structured Concurrency Policies

It's important to reiterate that direct use of `StructuredTaskScope` is discouraged. Instead, it serves as a foundation for two subclasses, each implementing a common concurrency design pattern: `Invoke All` and `Invoke Any`. These patterns are not novel; they were also implemented in Java 8's `ExecutorService`, which utilized collections of `Callable` objects. However, the `ExecutorService`'s execution was subject to variations based on the underlying thread pool's characteristics. Structured concurrency addresses this by providing predictable task behavior and scope. The same patterns can also be found in the `Fork/Join Framework` and in `CompletableFutures`.

InvokeAll

The `Invoke All` concurrency pattern, which is the more frequently used of the two built-in patterns, facilitates the creation of multiple subtasks and then waits until all subtasks have completed successfully or until any subtask has thrown an exception.

This pattern is provided by the `StructuredTaskScope.ShutdownOnFailure` static inner class. In the event that any subtask throws an exception, it is assumed that no valid result can be returned to the caller, and the entire scope is immediately shut down, hence the name `ShutdownOnFailure`. The previous example can be modified to employ the `ShutdownOnFailure` policy in the following manner:

```
try (var scope = new StructuredTaskScope.  
    ShutdownOnFailure()) { // 1  
  
    Supplier<CurrentWeather> currentSubtask = scope.  
fork(this::getCurrent); // 2  
    Supplier<Forecast> forecastSubtask = scope.  
fork(this::getForecast);  
    Supplier<History> historySubtask = scope.  
fork(this::getHistory);  
  
    scope.join(); // 3  
    scope.throwIfFailed(); // 4  
  
    return new Result(currentSubtask.get(), // 5  
                     forecastSubtask.get(),  
                     historySubtask.get());
```

Let us walk through the code:

7. We create a `StructuredTaskScope` of type `ShutdownOnFailure`
8. We fork three of subtasks
9. We call `scope.join` to make the thread that owns the scope wait until all subtasks are completed (or and exception is thrown)

10. Once the thread owning the scope continues, meaning all subtasks are completed or cancelled, we check if an exception was throws via `scopeIfFailed`
11. After this point, we can safely call `get()` on the subtasks as we can be assured now that no exception was thrown.

Notice how we define the scope as `StructuredTaskScope.ShutdownOnSuccess`. As said, `ShutdownOnSuccess` is a static final class defined within `StructureTaskScope` and extending it.

It uses much of the functionality of `StructuredTaskScope`, but overrides the protected `handleComplete(Subtask)` method to add specialized behaviour, that differs between `ShutdownOnSuccess` and `ShutdownOnFailure`.

This pattern is typically used if we need to retrieve multiple sets of data related to one another. Imagine a customer details screen that contains the customers address, a list of his outstanding orders, a list of his most recent completed orders, his outstanding balance, and a list of outstanding returns.

Instead of retrieving this information sequentially, we can retrieve it concurrently, while still waiting for all information to be retrieved before continuing. Structured concurrency provides a very easy to follow, sequential flow of code that is easy to read, understand, and reason about.

InvokeAny

The `Invoke Any` pattern is kind of the opposite compared to the `Invoke All` pattern. It is a race: the tasks that returns a value first wins. The idea is that multiple tasks are started simultaneously. As soon as any of these tasks returns a result, the other tasks that are still executing will be cancelled. If the first result is an exception, all other tasks will be cancelled as well, and the result of the execution will be this exception.

See the code sample below:

```
try (var scope = new StructuredTaskScope.  
    ShutdownOnSuccess<>()) { // 1  
  
    scope.fork(this::getWeatherData);  
    // 2  
    scope.fork(this::getOpenWeatherData);  
  
    scope.join();  
    // 3  
    return scope.result();  
    // 4  
}
```

In this scenario, we aim to fetch weather data from two distinct APIs. Our primary objective is to obtain the data as rapidly as possible, regardless of the API source. Consequently, the returned data might originate from different APIs on

each method call. The focus here is on minimizing response time, rather than relying on a particular data provider.

A comparable situation arises when retrieving driving directions from point A to point B. You could query multiple map services, such as Google Maps and Apple Maps, and choose the directions provided by the fastest responding service.

Lets us again step through this code:

1. We create a `StructuredTaskScope` of type `ShutdownOnSuccess`
2. We fork the subtasks. Note that we do not need to hold a reference to the results of the operation
3. The scope joins and waits for the first result to be returned
4. We return the result.

While resembling the preceding example, this one presents three significant differences. Firstly, it employs a different type of `StructuredTaskScope`. Secondly, we no longer need to retain references to the `SubTask` return values during the forking process. The API assumes responsibility for storing the single successful result, thereby eliminating the need for individual subtask queries. Lastly, the `scope.result()` method provides direct access to the result. If no subtasks complete successfully, an `ExecutionException` is thrown by default, but this behavior can be customized by overriding the overloaded result method

```
public <X extends Throwable> T
result(Function<Throwable, ? extends X> esf) throws
X
```

This overloaded method accepts a function that generates the desired exception. Notably, as soon as a subtask returns a successful result, the `StructuredTaskScope` automatically manages the cancellation of all other subtasks, relieving the developer of this responsibility.

Rolling your own strategy

Next to these two default policies, you are able to use your own policy targeted to the rules your business requires. Let us see how can create such a custom policy.

Picture this: you're planning a ski trip and want to ensure there's sufficient snow at your destination. We can develop a custom task scope, which we'll call `SkiPolicy`, to query a Weather API for snow conditions across a list of candidate cities.

We are using the `weatherapi.com` REST API here. To access the API you will need an API key. There is a free plan, which limits the number of requests you can you do. Still, 1,000,000 free requests per month should be more than sufficient to find you that ideal skiing resort.

We use the `Gson` library from Google to convert the JSON

responses coming from the API into Java objects.

We start off by defining Java records from the JSON responses

```
public record Condition(String text) { }

public record Current(Condition condition) { }

public record Location(String name, String country)
{}

public record CityWeather(Location location,
Current current) { }
```

Next, we define the main method of the application. Will will show this first and the later look at the details of the `run()` and `getForecast()` methods.

```
import com.google.gson.Gson;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.List;
import java.util.stream.Collectors;

public class CustomPolicyExample {

    private static final String API_KEY = „Add your
api key here“;

    public static void main(String... args) throws
Exception {
        if (args.length == 0) {
            System.out.println(„Usage: java
CustomPolicyExample <space-seperated list of
cities>“);
            System.exit(0);
        }
        new CustomPolicyExample().run(List.
of(args));
    }

    private void run(List<String> cities) throws
Exception{
        // discussed later
    }

    private CityWeather getForecast(String city)
throws Exception {
        // discussed later
    }
}
```


Next, we define the main method of the application. We will show this first and then later look at the details of the `run()` and `getForecast()` methods.

```
import com.google.gson.Gson;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.List;
import java.util.stream.Collectors;

public class CustomPolicyExample {

    private static final String API_KEY = „Add your
    api key here“;

    public static void main(String... args) throws
    Exception {
        if (args.length == 0) {
            System.out.println(„Usage: java
            CustomPolicyExample <space-separated list of
            cities>“);
            System.exit(0);
        }
        new CustomPolicyExample().run(List.
        of(args));
    }

    private void run(List<String> cities) throws
    Exception{
        // discussed later
    }

    private CityWeather getForecast(String city)
    throws Exception {
        // discussed later
    }
}
```

The main method takes the cities to examine for the correct skiing conditions as command line arguments. If there is at least one parameter specified, the `run()` method is called, passing on the command line parameters.

```
private void run(List<String> cities) throws
Exception{
    try(var scope = new
    SkiPolicy<CityWeather>()) {
        for (var city : cities) {
            scope.fork(() ->
            getForecast(city));
        }

        scope.join();
        scope.results().stream()
```

```
.filter(e ->
    e.current().
    condition().text().contains(„snow“)
    .collect(Collectors.toList())
    .forEach(System.out::println);
    }
}
```

The `run()` method forks a different thread for each city to retrieve the weather data. This is done by calling the `getForecast()` method. The `run()` method then joins the scope, effectively waiting until all threads have completed.

Once completed, the results of all the threads, that are collected in the Scope in the results variable, are filtered for the text „snow“. The ones matching the condition are then printed to console.

For completeness, here is the code from the `getForecast` method

```
private CityWeather getForecast(String city) throws
Exception {
    var uri = „http://api.weatherapi.com/v1/
    current.json?key=“
        + API_KEY
        + „&q=“
        + city“;

    var request = HttpRequest.newBuilder()
        .uri(new URI(uri))
        .GET()
        .build();
    try(var client = HttpClient.
    newHttpClient()) {
        var response = client.send(request,
        HttpResponse.
        BodyHandlers.ofString());
        CityWeather cw = new Gson().
        fromJson(response.body(),
        CityWeather.class);
        return cw;
    }
}
```

This URI for the REST call is made of the endpoint address and the parameters for the API key and the name of the city are added.

We then performed a simple GET request and upon receiving a result we use the GSON library to convert the JSON to a `CityWeather` Java object.

What is the future for structured concurrency?

You should now have a fairly good idea what structured concurrency is, how it works, and how you can apply it to your applications.

So should you use it in your production code? Based on functionality, I would certainly say „yes“. It is still a preview function, meaning that

the API could still change. It could even be withdrawn completely (as we saw with the StringTemplate JEP), but that is very unlikely.

I had expected the Structured Concurrency API to become final in Java 25, but now I am not so sure anymore. In fact, it is quite the opposite. There is a new Structural Concurrency JEP (Java Enhancement Proposal) that proposed quite some changes to the API. This proposal (<https://openjdk.org/jeps/8340343>) is still in draft, but I expect it to target Java 25.

It proposed the use a static open method on StructuredTaskScope and Joiner objects for implementing the policies.

```
String race(Collection<Callable<String>> tasks)
throws InterruptedException {
    try (var scope = StructuredTaskScope.
        open(Joiner.

<String>anySuccessfulResultOrThrow()) {
        tasks.forEach(scope::fork);
        return scope.join();
    }
}
```

Notice the use of the open method. And the use of the anySuccessfulResultOrThrow method, which is the ShutdownOnSuccess from above.

You can again custom implementations by creating custom joiners. For this, you need to implement the Joiner interface:

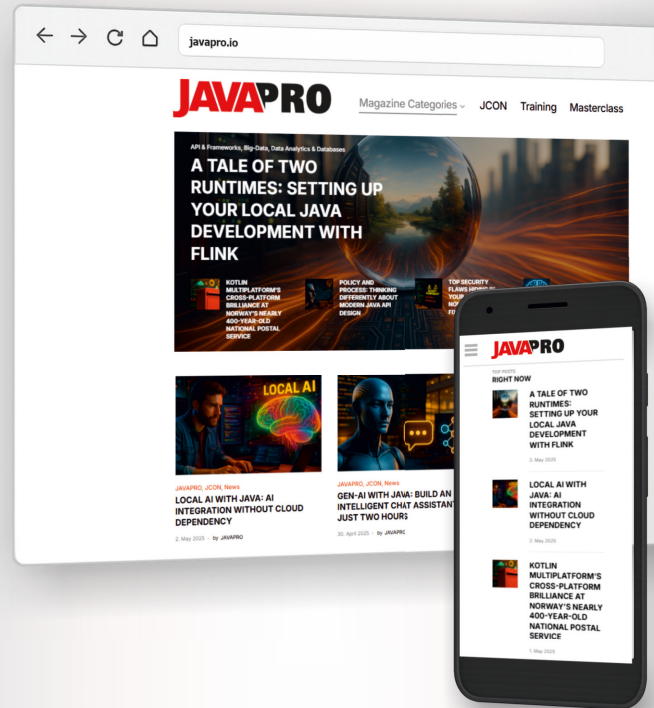
```
public interface Joiner<T, R> {
    default boolean onFork(Subtask<? extends T>
        subtask)
    default boolean onComplete(Subtask<? extends T>
        subtask)
    R result() throws Throwable
}
```

Conclusion

Structured concurrency is a very useful addition to the standard Java library. It takes the burden on concurrent programming away from the developer. It lets you write linear code that is easy to read, to understand, and to debug. As such, it offers solutions to concurrent programming that so far only could be found in Reactive Programming, but without the problems of Reactive (steep learning curve, added complexity, and hard to debug).

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

Hitchhiker's Guide to Java Performance



The Past, The Present and The Future

Over the last 30 years, Java has evolved from an exotic “write once, run anywhere” language to one of the dominant platforms for software development worldwide. In the early years, Java was justifiably considered slow compared to languages such as C/C++, which was mainly due to the initial interpreter approach. However, the last three decades have shown that the VM concept with HotSpot’s adaptive optimization is clearly the superior approach. by Ingo Düppe

Earlier JVM versions executed bytecode in a purely interpreted manner, which made Java programs 10 to 20 times slower than equivalent C code. So, performance was a challenge from the beginning, but continuous optimizations have drastically improved execution speed since then.

We take this opportunity to provide a detailed look at the evolution of Java performance. We look back at the past – from the first JVMs of the 90s to the introduction of the JIT compiler and early garbage collector strategies. We then look at the present of modern JVMs: the HotSpot engine, current tiered compilers (C1, C2 and GraalVM), advanced GCs such as G1, Shenandoah and ZGC, improvements in threading (e.g. Project Loom) and memory optimizations.

A look into the future shows what developers can expect with Loom, C2, native compilation and new projects. Finally, a few practical tips to avoid getting lost in the Java performance universe.

Past: The beginnings of Java performance

Java 1.0 and 1.1 in the years 1996 - 1997 relied exclusively on interpreters. The Java bytecode was therefore emulated command by command at runtime instead of being translated into native machine code. This approach ensured portability but led to considerable overhead. Average Java applications initially ran 10 to 20 times slower than programs written in C. Developers in the 90s therefore often derided Java as “too slow for serious applications”.

A just-in-time compiler (JIT) was first introduced in 1997 with Java 1.1. The JIT compiler dynamically translates frequently executed bytecode sequences into native code in order to significantly speed up execution. However, early JIT compiling was itself computationally intensive, so that only limited use could be made of it at first. The real turning point came from the HotSpot team: their technology was initially made available as an option for Java

Author:

Ingo Düppe is an IT consultant and founder of CROWDCODE. He advises in the field of architecture management and is responsible for the realization and conception of web, mobile and cloud applications. The challenge is to find the right combination of technologies, methods and tools to suit the individual context. As a coach, he supports teams in the use of Java EE and Spring technologies and their efficient integration into the individual software development process.



<https://www.linkedin.com/in/ingodueppe/>

1.2 in 1999 and became standard as of Java 1.3 (2000) as the HotSpot JVM. HotSpot introduced adaptive optimization: The JVM observes the code at runtime, identifies “hot spots” (frequently used code paths) and selectively compiles them with the JIT, while less critical code can remain in interpreted form. This selective approach minimized the compilation effort and led to massive speed gains – benchmarks showed up to 10-fold acceleration through HotSpot compared to purely interpreted code.

Also worth mentioning from this era is the threading model. Platform independence initially imposed restrictions on the use of operating system threads. Java 1.1 used green threads on some platforms – i.e. threads managed by the JVM runtime system in user space instead of native OS threads. Although this allowed threading even on systems without their own thread support and had low costs for thread switches, it did not scale on multi-processor systems. All green threads of a process shared one kernel thread, so one blocking operation (e.g. I/O) could freeze the entire VM.

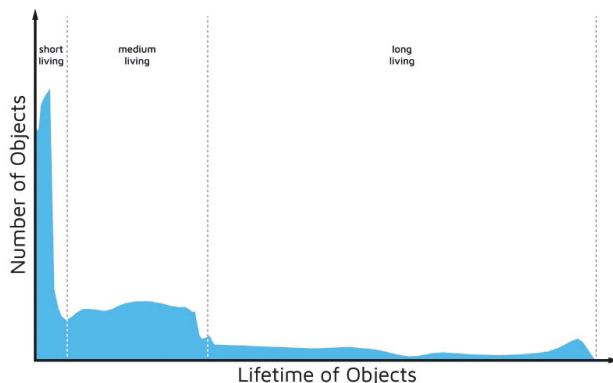
Java 1.3 therefore initiated the switch to native threads, which drastically improved parallelism on multi-core systems – albeit at the cost of slightly higher thread creation and context switching costs. Interestingly, Java has recently returned to the concept of ultra-light runtime-managed threads with Project Loom – more on this later.

Memory management and garbage collection (GC) of the 90s

Another key performance issue in the early days was memory management. Java freed developers from manual memory

allocation and deallocation, reducing programming errors but transferring the responsibility of memory cleanup to the JVM. The very first JVMs (1.0, 1.1) used a simple mark-and-sweep GC that periodically scanned the heap, marked unused objects as garbage and freed their memory. This procedure often led to heap fragmentation. In addition, the collection was performed with a stop-the-world scenario – in other words, the entire application paused while the GC was running, which led to noticeable lags. In addition, the GC did not scale with increasing memory.

With Java 1.2 (1998), the generational GC was introduced. The Weak Generational Hypothesis states that most objects are very short-lived and only a few objects are very long-lived. The JVM divided the heap into young and old generations accordingly and cleaned up the young generation (with the majority of the short-lived garbage) very frequently, and the old generation correspondingly less frequently. In addition, fragmentation could be reduced by copying collector approaches. This generational GC design proved to be significantly more performant. For example, it ensured that large heaps were better compacted and memory leaks were avoided. Despite these advances, GC pauses remained a problem in large Java applications of the late 1990s – concurrency in the GC itself was still rudimentary, causing the application to halt significantly during memory cleanup.



Weak Generational Hypothesis – most objects die young

The challenge of platform independence

All in all, the late 1990s were a time when fundamental performance techniques were just maturing for Java. Many challenges arose directly from platform independence: every advantage had to be achieved at runtime and without deep integration into the operating system. Early versions of Java suffered from slow GUI performance because AWT/Swing used platform-neutral code instead of native widgets. File and network accesses were noticeably slower than C APIs due to abstract streams and security checks. The JNI overhead made native libraries inefficient, which is why performance-critical code was often written in C/C++.

Differences in thread scheduling made optimal tuning difficult, as early JVMs did not map Java threads 1:1 to OS threads. Nevertheless, these years laid the foundation for later optimizations: With JIT, generational GC and native threads, Java became significantly faster and more scalable around 2000, a trend that still characterizes the platform today.

Present: Optimized Performance in Modern JVMs

Today's Java versions (Java 17 to 23+) feature highly optimized virtual machines that are the result of decades of research and development.

HotSpot Engine and Adaptive Optimization

For over 20 years, the HotSpot JVM has been the foundation of the Java SE platform. True to its name, it identifies hotspots in the code and aggressively optimizes them at runtime. It combines a bytecode interpreter with two JIT compilers: C1, which compiles quickly with limited optimizations, and C2, which generates highly optimized machine code for long-running server applications.

Tiered Compilation leverages both compilers in stages: methods start by being interpreted, C1 takes over after a few invocations, and frequently used code is eventually optimized by C2. This approach enables rapid warmup and maximum performance. During execution, HotSpot dynamically optimizes using techniques such as inlining, loop collapsing, and peephole optimizations, adapting based on runtime profiles. Speculative optimizations make assumptions about the code that are automatically rolled back if they prove invalid.

A milestone was the introduction of escape analysis in Java 6/7. This technique identifies objects that do not escape a certain scope and optimizes their memory management: objects can be allocated on the stack instead of the heap or completely eliminated through scalar replacement, thereby reducing the GC load.

Thanks to these optimizations, HotSpot Java achieves near-native performance in many scenarios. As early as 2008, benchmarks demonstrated that advanced JIT techniques allowed Java to lag behind C++ by only 10 - 30%, sometimes matching or even outperforming it due to runtime optimizations unavailable to static compilers.

Modern Garbage Collection: G1, ZGC, Shenandoah & Co.

JVM garbage collectors have made significant steps in recent years. In addition to the classic Serial and Parallel GCs, CMS introduced concurrent GC aspects for the first time, though it had shortcomings like heap fragmentation. With Java 7/8, G1 GC emerged as a modern replacement and became the standard in Java 9, while CMS was removed again in Java 14.

G1 GC operates on a regional basis, targeting the most memory-intensive regions for cleanup to keep configurable pause times (typically around 200 ms by default) within limits. Through concurrent marking and selective evacuation, G1 strikes a good balance between throughput, moderate pauses, and minimal tuning effort, making it especially suitable for medium to large heaps.

For even lower latencies, Shenandoah and ZGC were developed. Shenandoah reduces stop-the-world pauses by performing fully concurrent heap compaction using Brooks pointers, keeping

pauses in the sub-10-ms range even for 200-GB heaps – albeit without generational behavior in its early versions. ZGC follows a similar approach using colored pointers, which embed state information directly into pointers to efficiently manage object movements. It guarantees pauses under 10 ms, regardless of heap size. With Java 21, Generational ZGC was introduced to collect short-lived objects more efficiently and further boost throughput.

Today, the JVM offers a wide range of GC algorithms that can be chosen according to the specific application needs. While Oracle uses G1 as the standard and is looking to generational ZGC for the future, Red Hat often prefers Shenandoah. Thanks to these advances, Java can perform efficiently even with terabyte-sized heaps, without long stop-the-world pauses becoming an issue.

Threading and Concurrency: From Fork/Join to Loom

Since Java 5 (2004) and the `java.util.concurrent` package, a lot has happened in terms of parallelization: thread pools, locks, atomic variables and concurrent collections enable modern multi-core use. With Java 7, the fork/join framework (JSR 166y) for recursive parallelization was added, and from Java 8 onwards, parallel streams and lambdas further simplified this approach. At the same time, improvements such as lock striping and more efficient data structures (such as the `ConcurrentHashMap` with block-wise locking) were developed. This was followed in Java 11 by `VarHandle` and spin-wait hints via `Thread.onSpinWait()`, which facilitate high-performance concurrency. A typical example looks like this:

```
while (!lock.isAvailable()) {
    Thread.onSpinWait(); // Gives the CPU a hint for
    optimisation
}
```

In addition, thanks to lock elision via escape analysis, synchronization is no longer necessary if objects are only used thread-locally. Nevertheless, one hurdle remains: each Java thread corresponds to an operating system thread, which leads to high memory requirements (usually 1 MB per stack/thread) and expensive context switches for very large quantities (approx. over 100k). Scheduling then no longer scales linearly, and the limits of the OS are reached.

This was one motivation for Project Loom, which is arguably the most significant upheaval in Java's concurrency model in decades. Project Loom introduced Virtual Threads (also known as fibers), which have been available as a preview since Java 19 and were finalized in Java 21.

Virtual threads are ultra-lightweight threads managed by the JDK that are not permanently bound to kernel threads. They can be seen as a modern equivalent to green threads—but without their drawbacks.

The idea is that a Java program can create hundreds of thousands

of concurrent threads without overloading the operating system. Virtual threads are scheduled by the JVM and mapped onto a smaller number of kernel threads (so-called carrier threads). When a virtual thread blocks (for example, during I/O or `Thread.sleep()`), the entire carrier thread is not blocked too – the JVM can decouple the carrier and assign it to another virtual thread while the blocked thread waits in the background. As a result, blocking calls become virtually asynchronous automatically, without the programmer having to write complex callback or future logic.

In practice, Loom enables the well-known thread-per-request model to scale to high levels. Until now, handling thousands of simultaneous connections (e.g., in a web server) required resorting to asynchronous I/O or reactive programming (Netty, Vert.x, etc.) to avoid blocking an equal number of OS threads. With virtual threads, each request can now be handled in its own (virtual) thread – using simple, synchronous code – while the JVM ensures that the hardware is optimally utilized. Tests show that millions of sleeping virtual threads cause virtually no issues. IO-intensive applications also benefit enormously: In one experiment with 1 million parallel HTTP requests, the new virtual threads were able to handle the load with very little overhead, whereas 1 million traditional threads would have rendered the system unusable.

An example illustrates how virtual threads are handled in Java 21:

```
try (var executor = Executors.
    newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 100_000).forEach(i ->
        executor.submit(() -> {
            try {
                Thread.currentThread().
                    sleep(1_000);
                System.out.println("Thread #" + i +
                    " done");
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        })
    );
}
```

In this code, an `ExecutorService` is created that starts a new virtual thread for each submitted task. We submit 100,000 tasks, each of which sleeps for 1 second and then prints a message. This would be unfeasible with traditional threads – simply starting 100k OS threads would demand immense resources. With virtual threads, however, it is possible; the JVM multiplexes them onto a few dozen real threads, according to the number of available CPU cores.

For Java developers, Loom means that the complexity of concurrent programming can be significantly reduced: Many problems that were previously solved using asynchronous callbacks or reactive streams (to save threads) can now be written as simple sequential logic – and still scale to massive parallelism. It is important to note that virtual threads are not a magic bullet: They do not speed up CPU-bound workloads (here, more cores or better parallel algorithms are still necessary). However, IO-

bound workloads and systems with high concurrency demands will be considerably easier to write and maintain.

Java objects and memory consumption

A key aspect of Java performance is the memory requirements of objects and the memory organization (heap vs. stack).

Objects & Heap: Java objects have an overhead (class membership, synchronization state, etc.). On 64-bit JVMs, the header is typically 16 bytes for each object. There may also be padding, as objects are usually aligned to multiples of 8 bytes for alignment reasons. Java 6 introduced compressed OOPs (Ordinary Object Pointers): If the heap is smaller than ~ 32 GB, the JVM could use 32-bit pointers instead of 64-bit, which saves 4 bytes per object field. If more memory is required, you can switch to 16-byte alignment (-XX:ObjectAlignmentInBytes), for example, which means that even heaps up to 64 GB still use compressed pointers. As a rule, the switch from 32-bit to 64-bit pointers requires 1.7 times more memory.

String: Since Java 9, the JVM uses compact strings internally. This means that if a string only consists of pure ASCII characters, it saves the characters as byte[] (8-bit per character) instead of char[] (16-bit per character) as before. This saves nearly half of the storage space for the many strings that are typically ASCII (e.g. JSON texts, protocol messages). In individual applications, a performance gain of ~40% was observed simply by switching from Java 8 to 9.

- **Metaspace:** Java 8 replaced the “PermGen” with Metaspace, which is managed in the native memory (off-heap). Metaspace grows dynamically and eliminates the old problem of a fixed permanent heap, which could previously lead to OutOfMemoryError. On the performance side, this did not bring any direct acceleration, but increased stability and simplified tuning.
- **Heap vs. off-heap:** In addition to the regular Java heap (for objects), the JVM also uses native memory in various situations: e.g. the metaspace (see above), DirectByteBuffer allocations for NIO (to keep byte buffers outside the heap, which avoids expensive copy operations into kernel SPACE), as well as C heaps for JNI libraries. For performance, it can be useful to make targeted use of off-heap memory – this relieves the GC, but at the expense of more complex memory management.
- **Stack allocation:** Using escape analysis (see above), the JVM tries to avoid storing objects on the heap if it is not necessary. This effectively results in something like a form of value types for short-lived objects even today: the object lives implicitly only on the stack and is discarded with the method frame, completely without GC.

In summary, a lot has been done in the memory layout to reduce the footprint of Java applications. This is not just about saving memory: less occupied memory and less fragmentation also means less work for the GC and better use of CPU caches.

For example, more compact objects can improve cache locality, which leads to noticeable speedups, especially with large data structures. Developers should nevertheless make conscious decisions: Using primitive arrays vs. object lists, using string sparingly and knowing the cost of autoboxing. The JVM mitigates a lot, but a basic understanding of memory usage helps to avoid performance bottlenecks.

JVM Implementations: HotSpot, OpenJ9, Azul Prime, GraalVM

HotSpot (OpenJDK/Oracle JDK) is the most widely used JVM, but there are powerful alternatives with specific advantages:

- **Eclipse OpenJ9,** IBM’s open-source JVM, is optimized for fast startup and low memory usage. With IBM’s Testarossa JIT and its own GC, it’s ideal for cloud environments with limited RAM, though it slightly lags behind HotSpot in peak throughput.
- **Azul Platform Prime** (formerly Zing) is designed for low-latency applications. Its pauseless C4-GC minimizes pauses even with large heaps, while the LLVM-based Falcon JIT often produces more efficient code than HotSpot C2. Azul has been using these technologies in production for over a decade, making it a trusted solution for latency-critical systems.
- **GraalVM** is an extended JDK distribution from Oracle Labs with polyglot support for JavaScript, Python, R, Ruby, and WebAssembly. Its Native Image feature enables ahead-of-time compilation, allowing Java applications to start instantly and consume less RAM. While ideal for serverless and FaaS workloads, a well-optimized HotSpot VM often delivers better latency under sustained load.

These alternatives demonstrate that Java is no longer a monolithic system but an ecosystem of diverse VM technologies. GraalVM Native Image has gained traction in the cloud era, making Java more attractive for fast-starting CLIs and FaaS. Azul Prime remains relevant for high-frequency financial systems, while OpenJ9 is a strong option for memory-sensitive workloads. The diversity of JVMs fosters innovation and continuously drives performance improvements.

Milestones of the Java versions (performance-relevant)

Finally, in this present block, a compact overview of some important versions and their performance innovations:

Version	Year	Key Performance Features
Java 5 (Tiger)	2004	java.util.concurrent (thread pools, futures), 64-bit JVM, improved JIT, CMS GC (low-pause GC as an option)

Version	Year	Key Performance Features
Java 6 (Mustang)	2006	Faster JIT compilation, Escape Analysis (experimental), Compressed OOPs, Parallel Old GC, Improvements in synchronization
Java 7 (Dolphin)	2001	Fork/Join framework, G1 GC (experimental, official from 7u4), invokedynamic, NIO 2 (asynchronous channels), tiered compilation (C1+C2)
Java 8 (Spider)	2014	Lambdas & Streams (easier parallelisation), Metaspace instead of PermGen, Compact Strings, Tiered Compilation activated by default
Java 9 (Jigsaw)	2017	G1 GC as standard, Flight Recorder (OracleJDK commercial, from OpenJDK 11 free), AOT compilation (experimental), spin-wait hint, 8-byte-aligned heap
Java 11 (LTS)	2018	ZGC (experimental), Epsilon GC (No-Op), Flight Recorder + Mission Control in OpenJDK, new HTTP client (better performance than HttpURLConnection)
Java 15	2020	Shenandoah GC integrated into OpenJDK (JEP 379); ZGC now ready for production (no more experiment flag, JEP 377); text blocks (syntax, without performance impact); hidden classes (for frameworks such as bytecode generators, slightly more efficient)
Java 17 (LTS)	2021	Stronger encapsulation in the JDK (security vs. performance), new platform intrinsics (e.g. CRC32, GHASH), Shenandoah in Oracle JDK, consolidated updates from 11-16.
Java 19	2022	Virtual Threads (Preview) for high IO parallelism, Structured Concurrency (Incubator), Foreign Function & Memory API (Preview) for faster access to Native/Off-Heap
Java 21 (LTS)	2023	Virtual Threads GA (JEP 444), Generational ZGC (Preview), overall big step through Loom and improved GC.

Version	Year	Key Performance Features
Java 22	2024	Region Pinning for G1 (JEP 423): Stabilizes/improves GC by fixing certain heap regions. Foreign Function & Memory API (JEP 454): More efficient native interop, lower overhead. Vector API (JEP 460): Exploits SIMD instructions for data-parallel speedups. Stream Gatherers (JEP 461): Reduces I/O overhead via batched data gathering.
Java 23	2024	Vector API (JEP 469, 8th Incubator): Accelerates data-parallel tasks via SIMD Stream Gatherers (JEP 473, Second Preview): Improves throughput by batching multiple I/O operations. ZGC: Generational Mode by Default (JEP 474): More efficient GC, better handling of short-lived objects
Java 24	2025	It speeds up with generational Shenandoah and Compact Object Headers (both experimental), along with improvements to G1 that reduce GC overhead. It also offers faster AOT loading, refined streaming, and an extended Vector API for SIMD (9th incubator), among other enhancements.

You can see that the performance of Java has been improved in almost every version, either directly (faster JVM) or indirectly through new language/library features that enable more efficient code. The LTS versions (8, 11, 17, 21) in particular often included the experimental features of the intermediate releases in a stable form. It is always worth using the latest Java version in order to benefit from all optimizations – especially the performance boost has a direct effect without having to compile new byte code.

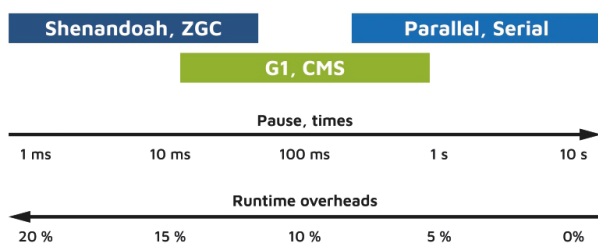
Future: What's Next for Java?

The Java platform continues to evolve, with several key projects shaping its future performance and scalability:

- Project Loom & Virtual Threads:** Now part of Java 21, Loom will take time to be fully adopted as frameworks like Tomcat, Jetty, Spring, and Quarkus integrate it. Virtual threads could eventually surpass reactive programming models by simplifying concurrency, allowing thread-per-request architectures to scale efficiently. Structured concurrency will further enhance clarity in concurrent code. While ideal for I/O-bound tasks, CPU-heavy parallel streams benefit less, as they remain OS-limited.

- **CRaC (Coordinated Restore at Checkpoint):** OpenJDK's CRaC project aims to dramatically reduce startup time by snapshotting a „warmed-up“ JVM state, which can be restored in milliseconds. This is highly relevant for cloud environments where rapid scaling is needed. Applications must adapt, handling resource cleanup before a snapshot. Currently, CRaC is Linux-only and still in development, but it presents an alternative to GraalVM Native Image by accelerating cold starts without losing JIT optimizations.
- **Native Compilation & Project Leyden:** Inspired by GraalVM Native Image, Project Leyden seeks to standardize ahead-of-time (AOT) compilation for Java. The goal is to tackle slow startup times, particularly for cloud-based applications, while addressing current limitations like restricted reflection and dynamic loading. Future Java versions may introduce separate JIT and AOT modes, making Java binaries more common in environments where fast startup is crucial.
- **Project Valhalla (Value Types):** Future Java releases will likely introduce value types, allowing objects to be stored flat in arrays or containers, improving cache efficiency and reducing pointer overhead. Though still in development, Valhalla will further optimize Java's memory footprint and performance.
- **Profiling with the Right Tools:** Identify performance bottlenecks before optimizing. Java Flight Recorder (JFR) and Async Profiler provide deep insights. Profiling early helps establish baselines, revealing top CPU consumers and allocation hotspots. For deeper analysis (e.g., lock contention, native bottlenecks), you should use specialized profilers or tools. Start broad (high-resolution CPU profiling) and refine focus gradually. Avoid guesswork – bottlenecks often lie where least expected (e.g., I/O instead of CPU, excessive GC instead of slow loops). The quickest way to find bottlenecks is with JProfiler [10] – my faithful travel companion for decades.
- **Efficient Data Structures & Memory Management:** Choose the right structures to reduce overhead. Use primitives over wrapper types (e.g., int instead of Integer) to avoid unnecessary boxing. For large collections, consider specialized libraries like Trove or FastUtil for primitive-backed lists. Avoid excessive String concatenation in loops – use StringBuilder instead. Minimize unnecessary object allocations, but avoid outdated object pools, as modern JVMs handle short-lived objects efficiently.
- **Concurrency & Parallelism:** Manage threads efficiently. For CPU-bound tasks, the optimal thread count is roughly equal to the number of CPU cores. For I/O-bound workloads, virtual threads (Java 21) allow scaling to thousands of tasks without excessive overhead. Minimize synchronization (Locks) and favor lock-free alternatives like Atomics, StampedLock, or LMAX Disruptor. When working with collections in multithreading, use Concurrent data structures or immutable snapshots instead of global locks. Always measure scalability – adding threads mostly doesn't improve performance linearly. Use JMH benchmarks [11] to determine optimal parallelism levels.
- **Stay Up-to-Date with JDK Versions:** New JDK releases offer significant performance improvements. Upgrading from Java 8 to Java 11 can reduce GC time by 20% due to G1 optimizations, while Java 17 improves string handling and GC efficiency further. Benchmarks show Java 17 increasing throughput by ~15% over Java 11 with lower latency, even without code changes. Regular updates ensure free performance gains without additional effort.

Java's future is focused on **faster startup, better concurrency, and reduced memory overhead**. Virtual threads will reshape how Java handles scalability, CRaC addresses startup delays, and Valhalla modernizes object handling. The last two are still a dream of the future, but these innovations would make Java more flexible, allowing applications to be fine-tuned for deployment needs – whether through native binaries or new concurrency models.



GC-Selection: Latency vs. Throughput

Practical Performance Tips for Hitchhiker's

To optimize Java application performance effectively, consider these key strategies:

- **Choosing the Right Garbage Collector:** The choice of GC impacts latency and throughput. G1 has been the default for most server applications since Java 11. For ultra-low pause times (real-time, UI), ZGC (stable since Java 17) or Shenandoah (preferred in Red Hat/SAP JDKs) is ideal. Batch workloads benefit from Parallel GC, while small tools with short runtimes may perform best with Serial GC to avoid parallelization overhead.

Keep It Simple - Avoid premature optimization—focus on architecture and algorithms first. The JVM is highly optimized, but when fine-tuning is needed, leverage profiling tools and modern features. With the right approach, Java can achieve exceptional performance.

Conclusion

Java has come a long way in the last 30 years. Thanks to highly developed JIT compilers, modern garbage collectors and innovative concurrency concepts, the platform that once had a reputation for being slow is now one of the most powerful environments for scalable server applications. The continuous improvements – whether in the HotSpot engine, in alternative JVMs such as

GraalVM or in forward-looking projects such as Loom and CRaC - show that performance in Java remains a lively and dynamic topic.

Developers should consistently use the many tools for analysis and optimization (JFR, JMC, Async Profiler, JMH) and orient themselves towards both proven and new technologies. In this way, they can get the best possible performance out of their applications - regardless of whether they are dealing with compute-intensive processes, latency-critical systems or modern cloud environments.

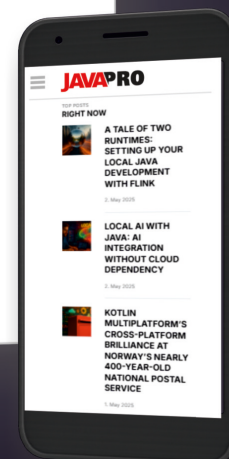
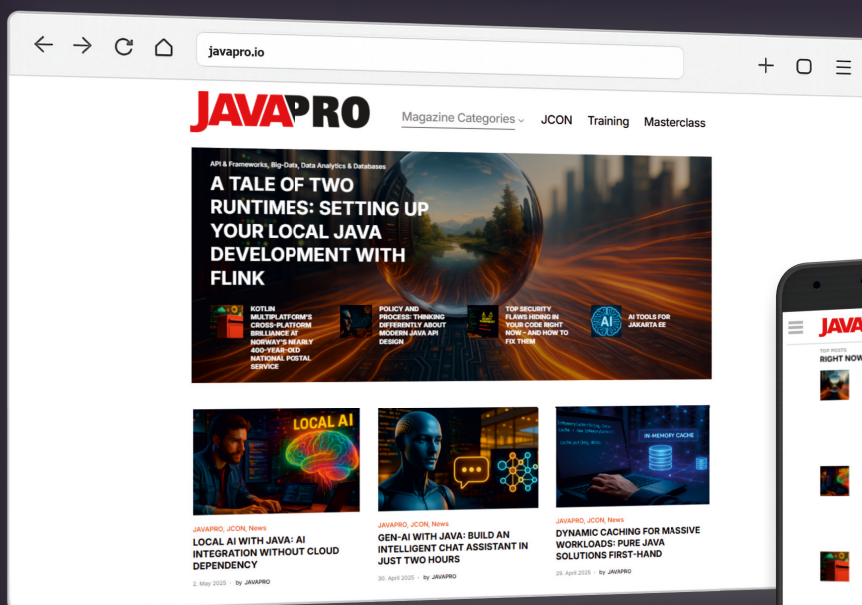
The future of the Java platform promises to become even more powerful through native compilation, optimized concurrency and energy-efficient approaches. Those who rely on the right optimization techniques today will not only benefit from higher speed, but also from better scalability and maintainability of the software. Java will therefore remain a central technology in the future - ready to master the challenges of modern applications.

Literature:

- [01] https://www.academia.edu/50669704/Escape_analysis_for_Java
- [02] <https://www.azul.com/blog/why-we-called-our-new-jit-compiler-falcon/>
- [03] <https://shipilev.net/talks/javazone-Sep2018-shenan-doah.pdf>
- [04] <https://openjdk.org/projects/leyden/>
- [05] <https://www.infoq.com/articles/java-virtual-threads-a-case-study/>
- [06] <https://webtide.com/jetty-12-virtual-threads-support/>
- [07] <https://www.infoq.com/news/2024/11/tomcat-11/>
- [08] <https://citeseerx.ist.psu.edu/document?doi=93b48e-ef8acfd110755b44ff1a346b65422a2bf4#>
- [09] <https://youtu.be/HCcq6VLuXe0>
- [10] <https://www.ej-technologies.com/jprofiler>
- [11] <https://github.com/openjdk/jmremain> a central tech-

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



www.javapro.io



#JAVAPRO #CoreJAVA

Modern Java

Author:



Ron Veen is a highly experienced software engineer who has seen it all, from Midrange Systems to Micro Services.

Through all this, he is guided by his passion for software engineering and architecture. For more than 15 years he has been working on the JVM and the Java ecosystem and has seen all the frameworks and libraries from Apache to ZK. He is an Oracle Certified Java Programmer (OCP) and Sun Certified Business Component Developer (SCBCD/OCPBCD).

Ron is an avid fan of alternative JVM languages like Groovy, Scala, Clojure, and his favorite, Kotlin.

At Team Rockstars IT he is a Special Agent and senior developer who frequently speaks at international conferences.

<https://x.com/ronveen>

<https://www.linkedin.com/in/ron-veen>

With Java's adoption of a six-month release cycle, new features are being introduced to the language rapidly. This article will explore the latest enhancements from Java versions 21 to 24. Our focus will be on language-specific changes, excluding updates to the JVM. Additionally, we will not cover the Foreign Functions and Memory API, which serves as a modern and safer alternative to JNI.

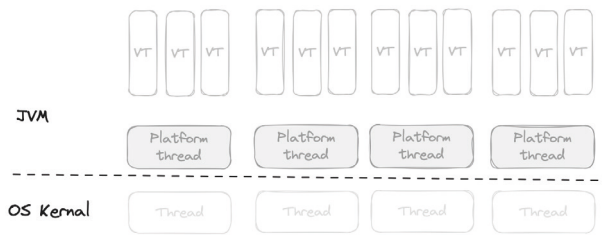
I would categorize the changes made to the language into three groups:

- Faster
- Better
- Easier

The first group would undoubtedly include the contributions from Project Loom, such as Virtual Threads, Structured Concurrency, and Scoped Values.

Virtual Threads

Virtual Threads (VTs) are lightweight alternatives to the traditional threads in Java. As illustrated below, they operate on top of the existing Platform threads.



Virtual Threads (VTs) are managed by the JVM and are designed to be lightweight in terms of memory usage and construction overhead. When a VT is ready to run, the JVM assigns it to an available Platform Thread. If a blocking operation, such as an I/O operation, occurs, the VT is removed from the Platform Thread and another VT takes its place. These context switches are inexpensive, and running a virtual thread has nearly the same overhead as a regular method call. This efficiency, combined with their low memory footprint, enables you to run millions of VTs in your application.

VT can be created via a factory method in the Thread class, like:

```
var vt = Thread.ofVirtual(runnable);
```

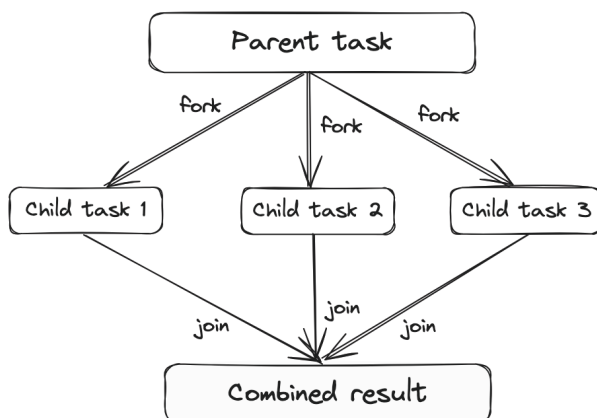
There is also a new ExecutorService that creates VTs:

```
var es = Executors.  
newVirtualThreadPerTaskExecutor();  
es.submit(runnable);
```

One final word, VTs should never be pooled, just discard them when you're done with them.

Structured Concurrency

Structured concurrency in Java simplifies managing multiple threads by organizing them into a structured task scope. This ensures that tasks are started, managed, and completed in a predictable manner. It uses policies like ShutdownOnFailure to cancel all tasks if one fails, and ShutdownOnSuccess to stop all tasks once one completes successfully. This approach enhances readability, reliability, and maintainability of concurrent code.



Essentially, you create a StructuredTaskScope and apply one of two default policies. The first policy, ShutdownOnFailure, cancels all forked tasks if any task fails. The second policy, ShutdownOnSuccess, shuts down all other tasks once a single task returns a result.

For a more detailed discussion, please refer to my dedicated article on Structured Concurrency in this magazine.

Scoped Values

Scoped Values act as an alternative, rather than a replacement, to ThreadLocals. Thread Locals have limitations such as mutability, potential exposure of sensitive information, scalability issues due to copying, and memory leaks.

In contrast, Scoped Values are immutable, passed by reference, and have a limited lifetime, which corresponds to the lifetime of the runnable they are assigned to.

```
public final ScopedValue<String> USER_PRINCIPAL =  
ScopedValue.newInstance();
```

```
ScopedValue.where(USER_PRINCIPAL, „sysadmin“).run(runnable);
```

During the execution of the runnable, the value for USER_PRINCIPAL is set and available. Accessing it outside of the runnable's code will throw an IllegalArgumentException. This ensures that only the specified runnable can access the value, preventing memory leaks caused by failing to remove the value.

This brings us to the second category of changes, which I would like to dub 'Better'.

In Java 17, the first pattern matching was introduced, the instanceof pattern. Since then, a considerable number of patterns have been added to the language.

Unnamed pattern and variables

The unnamed pattern and variables allows for not naming a local or pattern variable if you do not use them. You use the underscore (_) as the name.

Its usage is allowed in:

- For loop variables: for (Locale _ : Locale.getAvailableLocales())
- Catching exceptions: catch (Exception _)
- Return values: var _ = Class.getName();
- Try with resources: try (var _ = new FileInputStream(file))

There is one more usage, but more on that in the next section.

Record pattern

Java already supported switch pattern matching over object types, meaning that we switch over types, not over values.

Here is an example:

```
public String format(Object obj) {
    var result = switch(obj) {
        case null -> „null“;
        case Integer i -> String.
            format(„%d“, i);
        case Long l -> String.
            format(„%d“, l);
        case String s -> s;
        case Boolean -> b.toString();
        case Object o -> String.
            format(„%o“, o);
    }
    return result;
}
```

We also use this type of pattern matching for our object types. Assume we have the following sealed interface:

```
public interface FrequentFlyer permits BlueWing, SilverWing,
    RoyalWing {}
```

We could then write the following code to send a different email based on the type of FrequentFlyer being passed in:

```
public void sendEmail(FrequentFlyer ff) {
    switch (ff) {
        case BlueWing b -> sendEncouragingEmail(b);
        case SilverWing s ->
            SendNearlyThereEmail(s);
        case RoyalWing r -> SendMajesticEmail(r);
    }
}
```

Note that the default statement is not required because the compiler can determine that we handled all possible values in the switch.

We could take this a step further and actually deconstruct the record into local variables.

Assume that BlueWing, SilverWing, and RoyalWing all have the same structure:

```
public record BlueWing(Integer id, String name, String email,
    boolean active, int flights) {}
```

We could rewrite the above code as

```
public void sendEmail(FrequentFlyer ff) {
```

```
    switch (ff) {
        case BlueWing(var i, var n, var e, var b,
            var f) -> sendEncouragingEmail(n, e);
        case SilverWing(var i, var n, var e, var b,
            var f) -> SendNearlyThereEmail(n, e);
        case RoyalWing (var i, var n, var e, var b,
            var f) -> SendMajesticEmail(n, e);
    }
}
```

What happens here is that we ask the compiler to four local variables called i, n, e, b.

We did not need to specify their types as the compiler can infer it from their position in the record definition. So it will get the type of id, being an Integer. n will get the value of the 2nd element in the record, being a String. And so on.

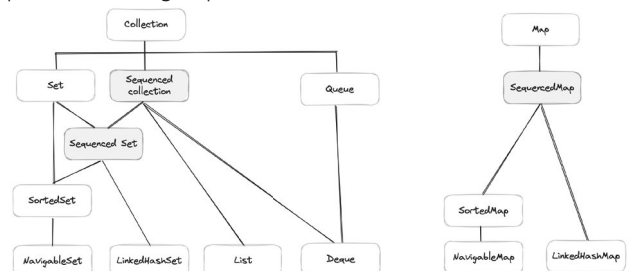
This creates local variables that are visible within the scope of the switch case.

But we can take this even one step further and apply the unnamed pattern here. We do not use the variables i (id) and b (active), so we just as well not define them.

```
public void sendEmail(FrequentFlyer ff) {
    switch (ff) {
        case BlueWing(var _, var n, var e, var _) ->
            sendEncouragingEmail(n, e);
        case SilverWing(var _, var n, var e, var _)
            -> SendNearlyThereEmail(n, e);
        case RoyalWing (var _, var n, var e, var _)
            -> SendMajesticEmail(n, e);
    }
}
```

Primitive patterns

A final pattern we should take a look at is the primitive patterns. So far, pattern matching could only be applied to objects. But with primitive patterns, added in Java 23, we can now apply pattern matching to primitive variables as well.



As an example, before if you wanted to get the last element of a list of an unknown size, you had to do some tricky operations:

```
var element = someList.get(someList.size() - 1);
```

This code is not only hard to read, but it also holds some danger, as you will get an `IllegalElementException` if the size of `someList` is zero.

Now, you can simply write this

```
var element = someList.getLast();
```

There are a whole bunch of new operations added:

```
var element = someList.getFirst();
var element = someList.getLast();
someList.addFirst(e);
someList.addLast(e);
var element = someList.removeFirst();
var element = someList.removeLast();

var reversed = someList.reverse();
```

For Maps, there are similar operations available, being that they have different names that use terminology that is more suitable for maps.

Flexible Constructor Bodies

As an example, before if you wanted to get the last element of a list of an unknown size, you had to do some tricky operations:

All Java programmers know this rule: in a constructor, the first statement should always be a call to `super()` or `this()`, to initialize the parent class or call another constructor in the same class. This is either explicit in the code or implicitly added by the Java compiler. The call to `super` is meant to initialize the parent class so that the child class can safely access them.

But this restriction is lifted within certain limits. You can now perform certain operations before calling `super()` or `this()`. There are certain scenarios where this could be very advantageous. One such example is validating parameters to ensure their validity. There is no use in initializing the parent class if the parameters passed in will be rejected.

```
public class PositiveBigInteger extends
    BigInteger {

    public PositiveBigInteger(long value) {

        super(value);           //
        Potentially unnecessary work

        if (value <= 0) throw new
        IllegalArgumentException(..);

    }
}
```

Another scenario would be if you try to prepare the parameters

required in the parent class:

```
public Sub(Certificate certificate) {

    var publicKey = certificate.getPublicKey();

    if (publicKey == null) throw ...

    byte[] certBytes = switch (publicKey) {

        case RSAKey rsaKey -> ...

        case DSAPublicKey dsaKey -> ...

        default -> ...

    };

    super(certBytes );
}
```

Stream Gatherers

Streams have been around since Java 8, which is almost 11 years now. During this period we as developers have become accustomed to using them, and for most of us, they are now the default way of writing code. Since their introduction, many functionalities have been added to streams by successive releases.

As an example of stream:

Breaking it down, a stream operation consists of:

- A stream source
- One or more intermediate operations
- A terminal operation

We have always had many ways to create stream sources, including many factory methods.

For terminal operations, we have several standard operations available. If these are not sufficient for the task at hand, we can create custom collectors. For intermediate operations, however, we did not have the option to create our own—until now!

It turned out that there was quite some demand for new intermediate operations. So the language designers had a choice: either add a large number of new intermediate operations to the standard language or allow developers to add intermediate operations. The solution was ... a bit of both!

First, a limited number of gatherers were added to the standard library:

- `mapConcurrent`
- `fold`

- scan
- windowFixed
- windowSliding

Here is an example of how to use them

```
public List<List<String>>
groupsOfThree(List<String> words) {

    return words.stream()
        .gather(Gatherers.windowFixed(3))
        .toList();

}
```

But the language designers did not stop there, they added a new API that allows developers to create gatherers themselves.

A gatherer consists of four parts:

- an initializer, creates an object to hold the intermediate state of the gatherer
- an integrator, integrates new elements and optionally pushes them downstream
- a combiner, combining two intermediate states into one
- a finisher, allowing for a final action to be performed at the end of the input stream

Only the integrator is required; the rest are optional and depend on the functionality you try to provide. Also, when 'downstream' is mentioned, the next stage in the stream pipeline is meant.

This might seem daunting at first, let's try to break it down with an example.

The first example, just for training purposes, provides an alternative for the map() intermediate operation. We accept an input element and output a transformed element.

```
public class AltMapGatherer {

    public static <T, R> Gatherer<T, ?, R>
altMap(Function<? super T, ? extends R> mapper) {
        Gatherer.Integrator<Void, T, R> integrator
= (_, current, downstream) -> {
            var result = mapper.apply(current);
            downstream.push(result);
            return true;
        };
        return Gatherer.of(integrator);
    }

    void main() {
        var list = List.of(„Duke“, „Duchess“);
        var upper = list.stream()
            .gather(altMap(String::toUpperCase))
            .toList();
        System.out.println(upper);
    }
}
```

```
        .gather(altMap(String::toUpperCase))
        .toList();
        System.out.println(upper);
    }
}
```

As mentioned, of the four parts of a gatherer, only the integrator is required. So we create an integrator here via a lambda expression. An integrator has three parameters. The first one is the intermediate state. Our alternative map operation does not hold any state between operations, so we simply ignore this parameter by specifying the underscore. The second parameter is the current stream element that is being processed, and the third and final parameter is a reference to downstream.

In the integrator's body, we apply the transformation operation to the stream element that was inputted. Next, we push this transformed element downstream to be processed by the next stage of the stream pipeline. Finally, we return true, signaling that we are ready to process more elements. Returning false would signal that we do not want to process more elements, allowing us to stop processing the remaining elements of the input stream.

In the main method, we will get to why it is so condensed in a bit, we can use our shiny new gatherer by adding it as an intermediate operation via the new Stream operation gather. In the constructor of altMap we specify the transformation function that is to be applied to each stream element.

As a next exercise, let us look at a gatherer that holds state. The gatherer mimics the behavior of the limit intermediate operation. You can specify how many input elements you want to process.

```
public class LimitingGatherer {

    public <T> Gatherer<T, AtomicInteger, T>
limiting(int maxSize) {
        return Gatherer.ofSequential(
            // Initializer
            AtomicInteger::new,

            // Integrator
            (state, element, downstream) -> {
                if (state.getAndIncrement() <
maxSize) {
                    downstream.push(element);
                    return true;
                } else {
                    return false;
                }
            });
    }

    public List<String> firstThreeWords(List<String>
words) {
        return words.stream()
            .gather(limiting(3))
            .toList();
    }
}
```

```

public void main () {
    var words = List.of(„The quick brown fox
jumps over the lazy dog“.split(„ „));
    List<String> firstThreeWords =
firstThreeWords(words);
    System.out.println(firstThreeWords);
}
}

```

We create a sequential gatherer here. This is the most common type of a gatherer, but there are parallel gatherers as well. This specific factory method of `Sequential` allows us to specify the initializer and the integrator. The initializer returns an object that will hold an intermediate state. In our case, we create an `AtomicInteger` to hold the number of elements already processed.

The integrator again receives three parameters, the state, the current element, and a reference to downstream. The first thing the integrator does is increment the `AtomicInteger`. It then verifies if we have not exceeded the maximum number of elements. If not, we push the element downstream and return true to indicate we are ready to process more elements. If we have reached the maximum of elements, we simply return false, indicating we do not accept any more elements. No more elements from the input stream will be sent to our gatherer.

So now we've come to the part of this article where we're gonna talk about things that make Java easier to use. These changes may not be so important for you and me, both of us being experienced Java developers. But to people new to the language, there are quite a lot of hurdles. Compiling, build tools, package imports, semantics, etc. Let us see where Java has become easier to use for novices.

Unnamed Classes And Instance Main Methods

One of the first things Java developers learned was how to run a Java application. We learned this by heart:

```

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(„Hello, World!“);
    }
}

```

We have become so accustomed to this code that we cannot even see what makes it so difficult to people new to the language. But they might have some questions like

- What does public mean? And what are the other possible values?
- What is static?
- What are args?
- Why do I have to define this within a class?
- And what is this `System.out.println`?

These are all good and fair questions. There are several changes made to the language that will allow you to turn the above code in this:

```

void main() {
    println(„Hello, World“);
}

```

That is right, a three-line Java application. You can leave out public and static from the method definition. You can even leave out the arguments to the main method.

We no longer have to use the `System.out.println` method to write data to the console. Instead, we can use `println` (and `print` and `readln`) from the new top-level class `java.io.IO`. Implicitly declared classes (more on that in a second) automatically import all the exported classes and interfaces from the `java.base` module, we do not even have to import this class.

As mentioned, if no outer class is defined around the main method, it is considered being an implicitly declared class. It is unnamed, and no package statement is allowed. Also, there can only be one unnamed class in your project.

All-in-all a pretty neat solution, isn't it?

Module Import Declarations

And while we are on the subject of importing packages and classes. There's something new there too. As you might recall, in Java 9, the JVM was modularized into what now are 95 different modules. Each module contains one to many packages.

The module `java.base` for example contains more than 60 packages like `java.lang`, `java.text`, `java.time` etc.

Before, we had to import each package when we needed it, but now we can simply import all packages in a module with the module import statement:

```

import module java.base;

```

Launch Multi-File Source-Code Programs

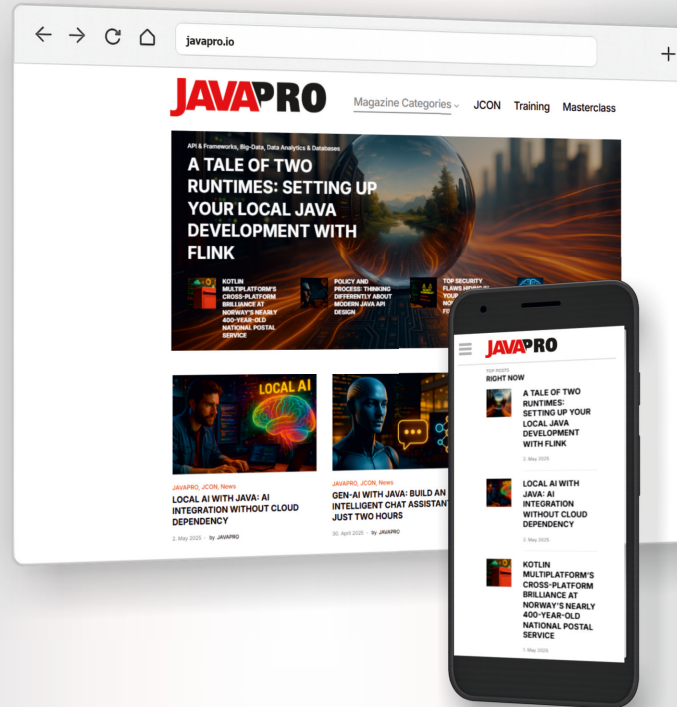
Did you know that you could run a Java program even without compiling it? This feature has been in Java since version 11. It allows you to code and run a simple Java program, without the need for compiling.

When you run Java and pass the name of the application, the JVM will look for a class with that name. If it cannot find it, it will look for a Java source file with the given name. If it is found, it is compiled into memory and then started by the JVM.

This again is a great example of Java has been made easier accessible for new developers.

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

But having only one source file has its limitations. If you want to write a small application with a few more classes, you still have to go through the code-compile-run circle. But since Java 22, we can now have multiple source-file applications. The only limitation is that they must reference each other directly.

Have a look at the following code.

```
void main() {
    Messenger messenger = new Messenger();
    println(messenger.getMessage());

    HelloMultipleWorlds.java

    public class Messenger {

        public String getMessage() {
            return „Hello, Multiple Worlds!“;
        }
    }
    Messenger.java
```

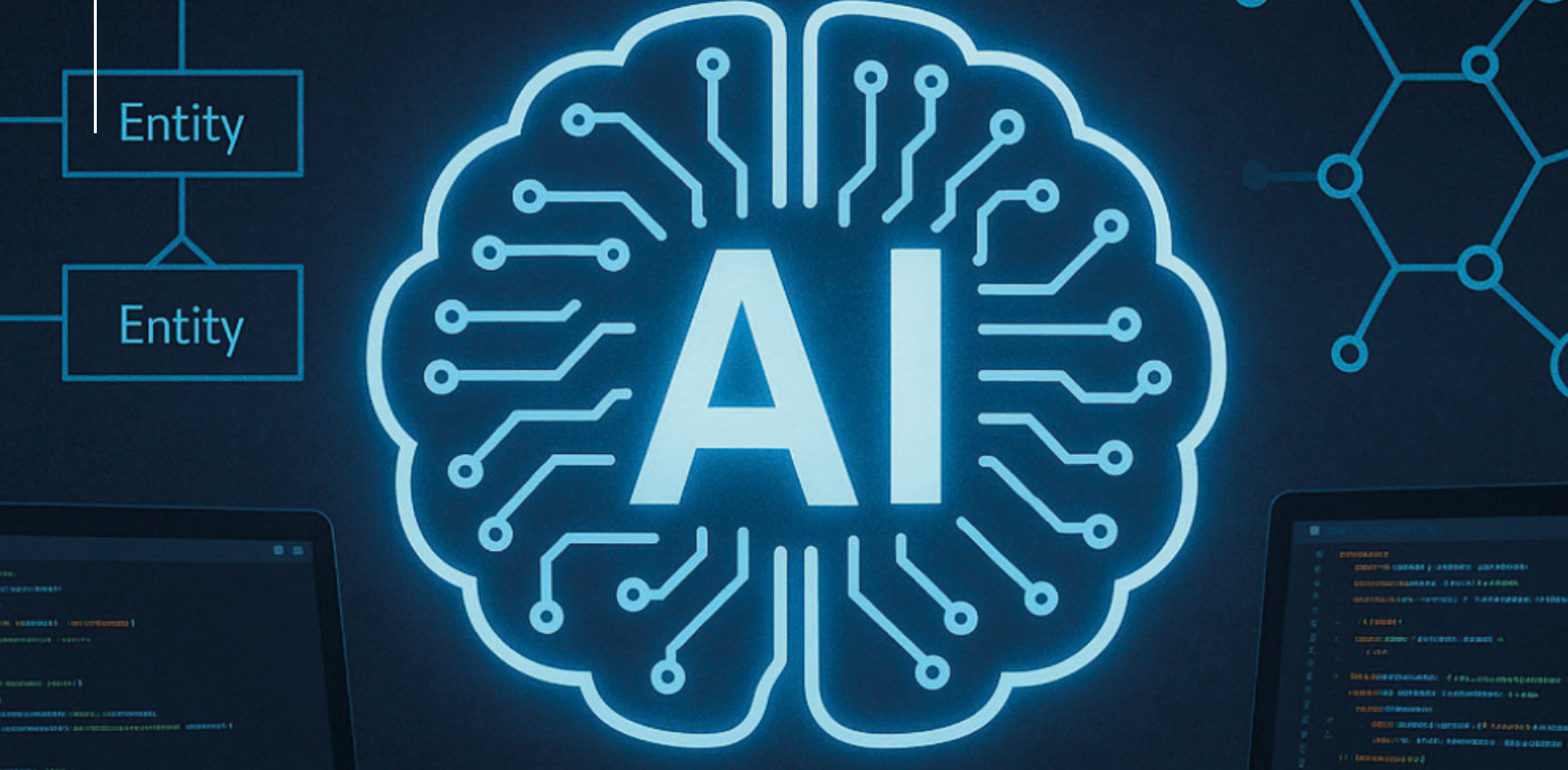
The first class, HelloMultipleWorlds.java is an implicitly declared class. It references the class Messenger in its code. As such, the JVM will compile both Java source files and then run HelloMultipleWorlds if we use the following command:

```
java --enable-preview --source 23
HelloMultipleWorlds.java
```

Allowing to run an application consisting of multiple source files without compiling them first means that novices do not have to get involved with tools like Maven or Gradle from the very beginning.

Conclusion

And there we are, we have gone through the most eye-catching changes in the Java language since version 21. With version 24 just out, and version 25, the next LTS, coming in September 2025, Java has a great future even after 30 years. And the language is only getting better.



#JAVAPRO #AI

AI Tools for Jakarta EE

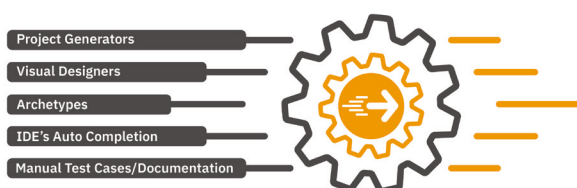


Introduction

Artificial Intelligence (AI) is transforming the world of software development, including Jakarta EE. Traditionally, developers relied on project generators, visual designers, and IDE auto-completion to build applications. However, these tools often lacked adaptability, context awareness, and efficiency. AI-powered tools such as Payara Starter and Jeddickt AI Assistant are now addressing these limitations, accelerating development workflows, and empowering developers to focus on innovation. This article explores how these tools enhance Jakarta EE development, using a conference application as a practical example. Additionally, it discusses future possibilities, including AI assistants integrated with Payara Server for server setup and diagnostics.

Traditional Development Challenges

Jakarta EE developers have long relied on conventional tools that, while helpful, come with limitations that can slow down development and increase the effort required for customization, testing, and documentation:



Author:

Gaurav Gupta is a Senior Software Engineer at Payara and the creator of Jeddickt, an innovative, open-source, AI-powered Jakarta EE application development platform. As an Apache NetBeans Committer and an Eclipse GlassFish Committer, Gaurav brings extensive expertise and dedication to advancing these key open-source tools. His focus on enhancing developer productivity and Jakarta EE standards through cutting-edge technologies highlights his commitment to the open-source ecosystem and his mission to empower developers worldwide.



<https://www.linkedin.com/in/jgauravgupta/>
<https://x.com/jGauravGupta>
<https://github.com/jGauravGupta>

- **Project Generators and Archetypes:** These tools help developers quickly scaffold applications using predefined templates. However, these templates are often rigid and require extensive customization to meet specific project requirements. This process can be time-consuming, especially for large-scale applications.

- **Visual Designers:** Visual tools allow developers to create database schemas and design UI layouts. However, these tools lack contextual awareness, meaning every component placement must be done manually. This manual process can lead to inconsistencies and slower development times.
- **IDE Auto-Completion:** IDEs provide auto-completion features that assist with basic syntax and code suggestions. However, these suggestions are limited because they do not fully understand the project's context, which means they cannot provide complex or high-level recommendations.
- **Manual Testing and Documentation:** Writing test cases and documentation by hand is time-consuming. Ensuring comprehensive test coverage and maintaining up-to-date documentation is essential for high-quality software, yet it often requires significant effort.
- **Automated Test and Documentation Generation:** AI can generate relevant test cases and comprehensive documentation by analyzing the code context. This automation ensures accuracy and completeness, saving developers significant time.
- **Intelligent Code Completion:** AI-powered code completion goes beyond syntax suggestions by understanding the entire project's context. Context-aware code completion results in smarter, more relevant code suggestions that help developers write cleaner, more efficient code.

These limitations can slow down development cycles, increase the risk of errors, and require developers to spend more time on repetitive tasks instead of focusing on innovation and core functionality.

By automating repetitive and time-consuming tasks, AI tools enable Jakarta EE developers to focus on building core functionality, enhancing user experiences, and driving innovation. This transformation leads to faster development cycles, higher-quality software, and greater overall productivity.

Let's explore some powerful AI tools that are transforming the Jakarta EE development landscape:

The AI Powered Tools for Jakarta EE Developer

Generative AI tools are transforming Jakarta EE development by addressing these challenges and offering smarter, context-aware

Payara Starter

Payara Starter (<https://start.payara.fish/>) is a powerful AI-powered tool designed to simplify Jakarta EE application development.



Using Entity-Relationship (ER) diagrams, developers can quickly design application architectures without manual setup. Generative AI in Payara Starter automatically creates ER diagrams based on natural language descriptions, allowing developers to refine

diagrams using live chat commands.

- **Adaptive Project Generators:** Unlike traditional project generators, AI-powered generators can create application scaffolds based on natural language prompts. These generators are adaptive, meaning they automatically select appropriate icons, titles, descriptions, and dynamic page content based on the application's context, reducing the need for manual adjustments.
- **Context-Aware Visual Designers:** AI-driven visual designers can operate using natural language commands. This allows developers to describe the desired UI and database structure, and the AI auto-designs the application based on the understanding of existing context and best practices. This capability eliminates the need for manual component placement and ensures consistent design.
- **Natural Language Archetypes:** Traditional archetypes are replaced with AI-driven code generation that responds to natural language commands. Developers can simply describe the functionality they need—such as „Create a REST endpoint for customer data”—and the AI generates the required code automatically, accelerating development.

Example: JCON EUROPE Conference Application

Payara Starter was used to generate an ER diagram for a conference application. By describing the application requirements (e.g., „JCON EUROPE Conference”), the tool created a diagram with entities such as Conference, Attendee, and Session, establishing relationships between them.

Source

JCON EUROPE Conference

1 erDiagram

2 CONFERENCE ||--o{ ATTENDEE : includes

3 CONFERENCE {

4 string conferenceId PK

5 string name

6 date startDate

7 date endDate

8 string location

9 }

10 ATTENDEE {

11 string attendeeId PK

12 string firstName

13 string lastName

14 string email

15 string ticketType

16 }

17 ATTENDEE ||--o{ SESSION : attends

18 SESSION {

19 string sessionId PK

20 string title

21 string speaker

22 date time

23 }

Create new ER Diagram

Live Preview

Tell us how you want to change the diagram (e.g., 'Add customer address') and press Enter

←

→

+

-

⌂

Using natural language commands, the diagram was expanded to include additional entities like Feedback and Registration, demonstrating the tool's context awareness.

Source

JCON EUROPE Conference

1 erDiagram

2 CONFERENCE ||--o{ ATTENDEE : includes

3 CONFERENCE {

4 string conferenceId PK

5 string name

6 date startDate

7 date endDate

8 string location

9 }

10 ATTENDEE {

11 string attendeeId PK

12 string firstName

13 string lastName

14 string email

15 string ticketType

16 }

17 ATTENDEE ||--o{ SESSION : attends

18 SESSION {

19 string sessionId PK

20 string title

21 string speaker

22 date time

23 }

24 ATTENDEE ||--o{ FEEDBACK : gives

25 FEEDBACK {

26 string feedbackId PK

27 string comments

28 int rating

29 }

30 ATTENDEE ||--o{ REGISTRATION : registers

Natural Language Command

Add Feedback and Registration

Live Preview

←

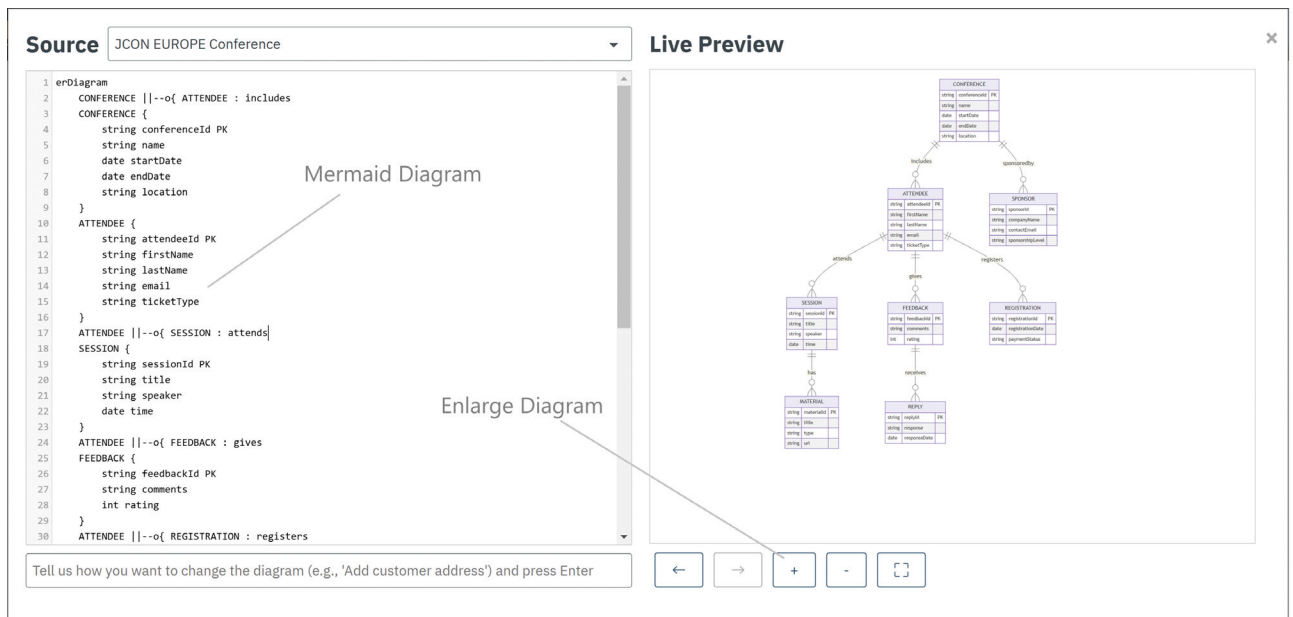
→

+

-

⌂

With the „+ enlarge“ button, the diagram was further expanded to include entities like Sponsor, Material, and Reply. This feature allows developers to iteratively refine their application design, ensuring comprehensive coverage of all business requirements.



Once the design was finalized and Jakarta EE, Payara, and JDK versions were selected, the application was generated with a single click.

Project Description

Build

●

Maven

○

Gradle

Group ID

jcon.europe.conference

Artifact ID

jcon-europe

Version

0.1-SNAPSHOT

Next

Jakarta EE

Payara Platform

Project Configuration

MicroProfile

Deployment Options

ER Diagram Designer

Security Configuration

Your request is in the queue. Please wait a moment while we process it.

Generate

The AI not only constructs the Jakarta EE backend structure with JPA entity classes and relationships but also auto-generates the frontend. User interfaces are designed with realistic titles, descriptions, and icons, tailored to each entity's function. This automation significantly reduces frontend development time while maintaining consistent UI design.

This capability represents a hybrid approach, combining the contextual adaptability of LLMs with the structured templates of traditional archetypes. While the LLM understands natural language descriptions and generates dynamic content, archetypes ensure consistent code scaffolding and design patterns. Together, they streamline both database schema creation and frontend UI design, accelerating development without compromising flexibility or consistency.

Conference Management

Search

Sign out

Dashboard

Conference

Attendee

Session

Feedback

Registration

Sponsor

Reply

Workshop

Material

OTHERS

About US

Session

Represents a session within the conference.

Add Session

Session Id	Title	Speaker	Time	Attendee	Actions
1	AI Tools for Jakarta EE	Gaurav Gupta	2025-05-13		Edit Delete

This significantly reduced development time, enabling developers to focus on core functionality.

Jeddict AI Assistant

Jeddict AI Assistant (<https://jeddict.github.io/>) is an AI-powered tool available for Apache NetBeans IDE, designed to streamline Java and Jakarta EE development. It offers intelligent code suggestions, context-aware variable and method naming, automated test generation, and more. With support for multiple LLM providers, developers can customize configurations to fit their workflow, ensuring efficiency and consistency.

Key features

- **Intelligent Code Completion:** Provides real-time, context-aware suggestions as developers type, reducing coding time and improving productivity.
- **Inline Hints in Code Editor:** Displays contextual hints directly in the editor, helping developers understand method usage, expected parameters, and best practices.
- **Context-Aware Naming:** Suggests meaningful variable and method names based on the surrounding code, aligning with standard naming conventions.
- **Inline SQL Completion:** Simplifies database interactions by offering real-time SQL suggestions as developers write queries.
- **Automated Logging and Documentation:** Generates clear, concise log messages and comprehensive Javadocs for classes and methods.
- **Test Case Generation:** Automatically creates relevant test cases based on the context of classes and methods, promoting best testing practices.
- **Multiple LLM Provider Support:** Allows developers to choose from different large language model providers with customizable configurations, including custom headers for secure API integrations.
- **Special Support for Jakarta EE and Java EE Projects:** Recognizes both javax and jakarta imports, ensuring compatibility with legacy and modern enterprise applications.
- **REST Endpoint Generation:** Quickly generates REST endpoints, including resource classes, annotations, and request/response models, reducing development time.
- **Context-Aware AI Chat:** Enables developers to select specific Java files or packages to provide context for AI-driven conversations, making it easier to get relevant suggestions and insights.
- **AI-Generated Commit Messages:** Automatically generates clear and meaningful commit messages based on code changes, ensuring better version control and collaboration.

Jeddict AI Assistant empowers developers to focus on building core functionality while automating repetitive tasks, enhancing productivity, and improving code quality across Java and Jakarta EE projects.

Conclusion

AI tools like Payara Starter and Jeddict AI Assistant are revolutionizing Jakarta EE development by automating repetitive tasks, accelerating coding, and enhancing code quality. With features like intelligent code completion, context-aware variable naming, automated test generation, and seamless REST endpoint creation, these tools empower developers to focus on building core functionality and delivering high-quality applications faster. Additionally, AI-generated documentation and commit messages streamline collaboration, ensuring teams can maintain consistency and share knowledge effectively.

Looking ahead, the potential for an AI Assistant specifically designed for Payara Server could further enhance productivity for developers of all skill levels. Beginners could benefit from simplified server setup and step-by-step guidance without needing advanced knowledge of asadmin commands. At the same time, experienced users could leverage AI for advanced diagnostics, log analysis, and performance optimization, ensuring efficient and reliable server operations. Automated server configuration through natural language commands would further streamline deployment and management, reducing the complexity of server administration.

As AI continues to evolve, its integration into Jakarta EE development promises smarter, faster, and more efficient workflows. By reducing manual effort and automating time-consuming tasks, AI empowers developers to focus on innovation, driving the creation of robust, scalable, and high-performing applications. With tools like Payara Starter, Jeddict AI Assistant, and future advancements in AI-driven server management, the future of Jakarta EE development is poised to be more productive and accessible than ever before.

JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



www.javapro.io



#JAVAPRO #AI

Securing the Future of AI: Authorization for Java RAG Systems using LangChain4j and OpenFGA

Author:



Deepu K Sasidharan is a Software Engineer by passion and profession. He is a Java Champion working as a Staff Developer Advocate at Okta. He is the co-chair of JHipster and the creator of KDash and JDL Studio. He is a polyglot programmer working with Java, Rust, JavaScript, Go, and so on. He is also a cloud technology advocate and an open-source software aficionado. He has authored books on Full-stack development and frequently writes about Java, Rust, JavaScript, Go, DevOps, Kubernetes, Linux, and so on, on his blog.

<https://www.linkedin.com/in/deepu05/>

<https://bsky.app/profile/deepu105.bsky.social>

<https://twitter.com/deepu105>

Artificial Intelligence is evolving at a breathtaking speed. AI agents will soon become the standard for building applications and they are going to be the new Web 2.0. With all that comes the critical need for robust security measures and checks. Retrieval-Augmented Generation (RAG) systems have become a cornerstone in building AI agents and applications. Agents that can access and utilize domain-specific knowledge are more accurate and more useful. However, as these systems become more prevalent, ensuring they only access and expose authorized information has become a critical challenge.

Sensitive Information Disclosure is a common issue that plagues RAG-based systems. We don't want the Large Language Model (LLM) to accidentally access or expose sensitive data from a database. Traditional Role-Based Access Control (RBAC) systems are not enough to secure RAG applications and agents. This is where Fine-Grained Authorization (FGA) or Relationship-Based Access Control (ReBAC) shines as an Authorization solution for RAG.

The Security Challenge in RAG Systems

RAG systems present some unique security challenges that traditional Role-Based Access Control (RBAC) systems struggle to address:

1. **Dynamic Context:** RAG systems need to make real-time decisions about which documents and tools can be used for context enhancement.
2. **Complex Relationships:** Documents often have intricate relationships with users, teams, and projects, forming a complex graph that simple role-based systems can't model effectively.
3. **Granular Control:** Different parts of documents may have different sensitivity levels requiring fine-grained control.
4. **Performance Requirements:** Authorization checks must be fast to maintain the AI system's responsiveness.

This is where FGA shines, it allows us to:

- Model complex relationships between users, groups, and documents
- Handle hierarchical and transitive permissions naturally
- Scale authorization rules efficiently

Let's look at some real-world scenarios where sensitive data could be at risk:

- A knowledge base assistant exposing confidential documents to employees
- A support AI revealing sensitive user information
- An AI assistant leaking proprietary information about a company
- A chatbot exposing private data to unauthorized users

Why OpenFGA?

OpenFGA is an open-source Fine-Grained Authorization system, created by Auth0, that allows you to manage access control in your applications. It's currently a CNCF Sandbox Project. OpenFGA is a powerful tool that allows you to define complex access control policies and enforce them in your application. It is a great choice for securing RAG applications as it solves most of the security challenges in RAG systems, including the ones mentioned above. You can also use a hosted version of OpenFGA from Auth0 if you don't want to self-host.

If you're new to RAG and access control, I recommend checking out this introductory post on RAG and Access Control: Where Do You Start?.

Why Java and LangChain4j?

Java isn't far behind when it comes to building AI applications. It has a large developer community and a rich ecosystem of open-source frameworks, and enterprises are adopting these frameworks.

LangChain4j is one of the popular Java AI frameworks. It is a powerful open-source framework that provides capabilities similar to LangChain and LlamaIndex used in the Python/JS ecosystem. It is becoming the Spring Boot of the Java AI ecosystem. The framework offers a comprehensive set of features for building AI applications. Some of the key features include:

- Easy integration with major commercial and open-source LLMs
- Unified API for different model providers
- Tools like prompt templating, chat memory management, and function calls.
- High-level patterns, abstractions, and implementations for building Agents and RAG applications

Building a Secure RAG System

LangChain4j makes it simple to get started with the „Easy RAG“ feature while providing advanced capabilities for complex use cases. Hence, in this guide, we will look at a secure Java RAG system using LangChain4j and OpenFGA. We'll focus on Relationship-Based Access Control (ReBAC), which prevents unauthorized access to sensitive information while maintaining the flexibility and power of RAG systems.

Prerequisites

This guide was created with the following tools and services:

- Java 21
- Gradle 8.12
- An FGA instance (self-hosted OpenFGA or create an Auth0 FGA store)
- A local Ollama instance or OpenAI API key

Setting up the project

To get started, clone the auth0-ai-samples repository from GitHub:

```
git clone https://github.com/oktadev/auth0-ai-samples.git
cd auth0-ai-samples/authorization-for-rag/
langchain4j-java
```

The application uses Gradle as the build system and is structured as follows:

- `src/main/java/rag/RagApplication.java`: The main entry point of the application that defines the RAG pipeline.
- `src/main/java/rag/FGARetriever.java`: The FGA-aware retriever that retrieves documents from the vector store based on the user's permissions.
- `src/main/java/rag/FGAInit.java`: Script to create the FGA store and models.
- `src/main/resources/docs/*.md`: Sample markdown documents that will be used as context for the LLM. We have public and private documents for demonstration purposes.
- `build.gradle`: The Gradle build file that defines the dependencies and tasks.

Let us look at the important bits and pieces before we run the application.

RAG pipeline

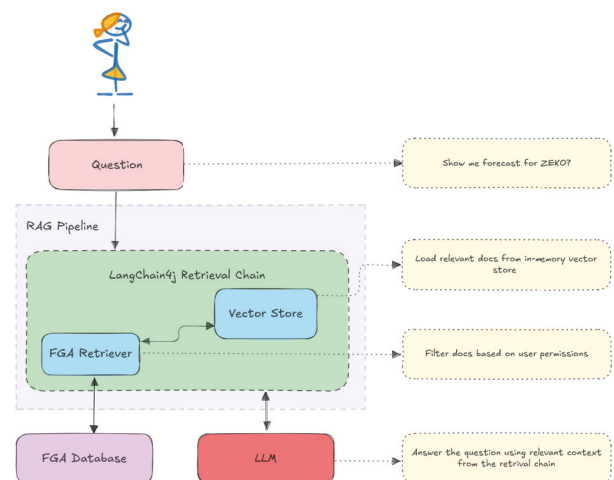
The RAG pipeline configures the underlying LLM and defines the retrievers for data. The pipeline is defined as follows:

```
/** src/main/java/rag/RagApplication.java */
final ChatLanguageModel CHAT_MODEL_OPENAI =
    OpenAiChatModel
        .builder()
        .apiKey(System.getProperty("OPENAI_
            API_KEY"))
        .modelName(GPT_4_O_MINI)
        .build();

var user = „user2“;
// 1. Read and load documents from the assets
folder
var documents = FileSystemDocumentLoader
    .loadDocuments("src/main/
        resources/docs");
// 2. Create an in-memory vector store from the
documents.
InMemoryEmbeddingStore<TextSegment> embeddingStore
    = new InMemoryEmbeddingStore<>();
EmbeddingStoreIngestor.ingest(documents,
    embeddingStore);
// 3. Create a base retriever
ContentRetriever baseRetriever =
    EmbeddingStoreContentRetriever.
        from(embeddingStore);
// 4. Create the FGA retriever that wraps the base
retriever
ContentRetriever fgaRetriever = FGARetriever.
    create(
```

```
        baseRetriever,
        // FGA tuple to query for the user's
        permissions
        content -> new ClientBatchCheckItem()
            .user(„user:“ + user)
            .relation(„viewer“)
            ._object(
                „doc:“ +
                content.textSegment().metadata()
                    .getString(„file_name“)
            )
            .split(„\\.“)[0]
        );
// 5. Create an assistant with the FGA retriever
var assistant = AiServices.builder(Assistant.class)
    .chatLanguageModel(CHAT_MODEL_OPENAI)
    .chatMemory(MessageWindowChatMemory.
        withMaxMessages(10))
    .contentRetriever(fgaRetriever)
    .build();
// 6. Query the retrieval chain with a prompt
var query = „Show me forecast for ZEK0?“;
var answer = assistant.chat(query);
```

If you want to use Ollama running locally, uncomment the `CHAT_MODEL_OLLAMA` variable and comment out the `CHAT_MODEL_OPENAI` line. Be sure to update the base URL and model name to match your local Ollama instance.



FGA retriever

The `FGARetriever` class implements `ContentRetriever` and filters documents based on the authorization model defined in the FGA store. This retriever is ideal for scenarios where you already have documents in a vector store and want to filter the vector store results based on the user's permissions. Assuming the vector store already narrows down the documents to a few, the FGA retriever will further narrow down the documents to only the ones to which the user has access.

Here is the important part of the FGARetriever class simplified for readability:

```
/** src/main/java/rag/FGARetriever.java */
// Create a client to interact with the FGA store
new OpenFgaClient(new ClientConfiguration()
    .apiUrl(/*...*/)
    .storeId(/*...*/)
    // credentials only required if using Auth0
    FGA,
    // comment out for local OpenFGA
    .credentials(/*...*/)
)

// Check permissions for the user and object
private Map<String, Boolean>
checkPermissions(List<ClientBatchCheckItem> checks)
{
    var options = /*...
    var request = /*...
    var response = fgaClient.batchCheck(request,
options).get();

    Map<String, Boolean> permissionMap = new
HashMap<>();
    for (var result : response.getResult()) {
        var checkKey = getCheckKey(result.
getRequest());
        permissionMap.put(checkKey, result.
isAllowed());
    }
    return permissionMap;
}

// retrieve the documents and filter them using
checkPermissions function
public List<Content> retrieve(Query query) {
    // First, get relevant documents from the base
retriever
    List<Content> relevantContent = baseRetriever.
retrieve(query);

    // Create data structures to track checks and
document mappings
    // var checks, documentToObject, seenChecks =
...

    // Process each document to build checks
    for (Content doc : relevantContent) {
        ClientBatchCheckItem check = buildQuery.
apply(doc);
        var checkKey = getCheckKey(check);
        documentToObject.put(doc, checkKey);

        // Skip duplicate checks for same user,
object, and relation
        if (!seenChecks.contains(checkKey)) {
            /*...*/
        }
    }
}
```

```
    }
}

var permissionsMap = checkPermissions(checks);
// filter based on permission
return relevantContent.stream()
    .filter(doc -> permissionsMap.
get(documentToObject.get(doc)))
    .collect(Collectors.toList());
}
```

Set up An FGA Instance

Run OpenFGA locally with Docker using the following command:

```
docker pull openfga/openfga && \
docker run -p 8080:8080 -p 8081:8081 -p 3000:3000
openfga/openfga run
```

If you are using Auth0 FGA, visit the dashboard, navigate to Settings, and in the Authorized Clients section, click + Create Client. Give your client a name, mark all three client permissions, and then click Create. Once your client is created, you'll see a modal containing Store ID, Client ID, and Client Secret. Click Continue to see the FGA_API_URL and FGA_API_AUDIENCE.

Add .env file with the following content to the root of the project.

```
# OpenAI key. Not required if using Ollama
OPENAI_API_KEY=<your-openai-api-key>

# Open FGA
FGA_STORE_ID=<your-fga-store-id>
# Required only for Auth0 FGA
FGA_CLIENT_ID=<your-fga-store-client-id>
FGA_CLIENT_SECRET=<your-fga-store-client-secret>
FGA_API_URL=https://api.xxx.fga.dev
FGA_API_AUDIENCE=https://api.xxx.fga.dev/
```

Create the FGA models and tuples

We will use a simple model that defines just an owner and viewer relation for docs:

```
model
  schema 1.1

  type user

  type doc

  relations
    define owner: [user]
    define viewer: [user, user:*]
```

Now, to have access to the public information, we will need the following tuple on FGA.

- User: user:*
- Object: select doc and add public-doc in the ID field
- Relation : viewer

A tuple signifies a user's relation to a given object. For example, the above tuple implies that all users can view the public-doc object.

Now, to have access to the private information, we will need the following tuple on FGA.

- User: user:user1
- Object: select doc and add private-doc in the ID field
- Relation : viewer

For OpenFGA, the store, models, and tuples can be created programmatically using the OpenFGA Java SDK. This is defined in `src/main/java/rag/FGAInit.java` and can be run using the following command:

```
gradle runFGAInit
```

Once done, copy the store ID from the console and update the `.env` file with the store ID.

If you are using Auth0 FGA, navigate to Model Explorer. You'll need to update the model information with the above schema. Next, navigate to the Tuple Management section and click + Add Tuple, fill in the details for both tuples detailed above.

Test the application

Now that you have set up the application and the FGA store, you can run the application using the following command:

```
./gradlew run
```

The application will start with the query, Show me forecast for Zeko? Since this information is in a private document, and user2 does not have access to this document, the application will not be able to retrieve it. The FGA retriever will filter out the private document from the vector store results and, hence, print a similar output.

```
The provided context does not include specific
forecasts or projections for Zeko Advanced Systems
Inc. ...
```

If you change the query to something that is available in the public document, the application will be able to retrieve the

information.

Now change the user to user1 and run the application again:

```
./gradlew run
```

This time, you should see a response containing the forecast information since we have a tuple that defines the viewer relation for user1 to the private-doc object.

Congratulations! You have run a simple RAG application using LangChain4j and secured it using OpenFGA.

Implementation Patterns for Securing RAG Systems

Here are some common patterns for securing RAG systems using FGA:

1. Pre-filtering Pattern: Check permissions before searching the vector database. This can be done using the OpenFGA list objects request and metadata filtering feature of supported vector databases.
2. Post-filtering Pattern: Perform a vector database search first and then filter the results by permissions. This can be done using the OpenFGA batch check request and custom retrievers or re-rankers. This is what we just did in the example above.
3. Hybrid Pattern: Combine pre-filtering and post-filtering patterns to achieve the best of both worlds.

Learn more about OpenFGA

In this post, you learned how to secure a LangChain4j-based RAG application using OpenFGA. We invite you to check out the OpenFGA code on GitHub. Visit Auth for GenAI to learn more about how Auth0 can help you secure your GenAI applications.

Build AI Apps and Agents in Java: Hands-On with LangChain4j



LangChain4j is a java library to make interactions with AI models and LLMs in Java easy, with unified APIs that wrap different models. It also provides tools to let you build more complex use cases on top of basic functionality. At this 2nd anniversary of LangChain4j, we are thrilled to give you a walkthrough of the functionality that enables you to build basic and advanced AI-powered apps in Java.



The goal of this article is to show you the most important features of LangChain4j, starting from the basics, building up to more advanced and cooler examples. Are you curious how to use LLMs in Java but never tried it yourself? Read on, this article is for you! Did you already build some apps but you're curious to learn more advanced features and use cases? Skip the first chapters and jump right to the topic of your fancy.

Intro to LangChain4j

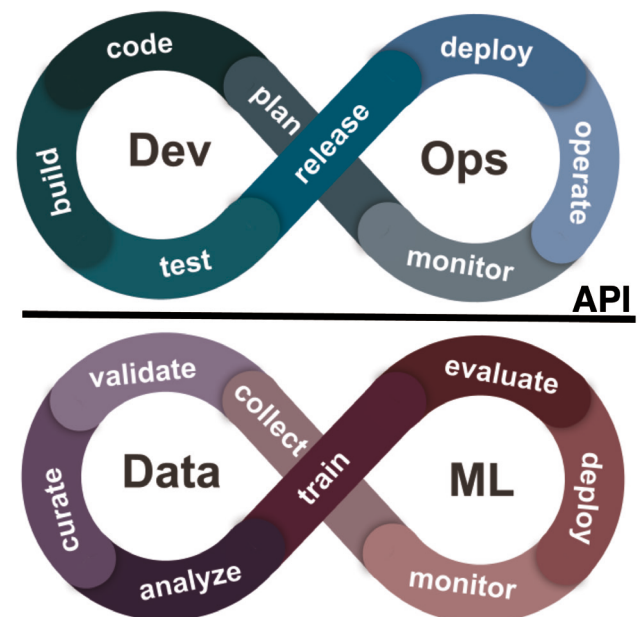
LangChain4j is all about interacting with AI models. Using a model is called inference. We can stay in our DevOps space and interact with the model via API, without caring too much about all the details in the layer below. What LangChain4j it NOT about:

- needing to understand how AI Models work under the hood: we can leave that to the data scientists.
- training or fine tuning models: if you want to do that, have a look at Tribuo for classic machine learning strategies like regression, or at DeepLearning4j for deep learning and neural networks.

Author:

Lize Raes is Product Manager and Developer Advocate at Naboo.ai, where she helps to build the developer productivity toolbox of the future. As collaborator at LangChain4j, she loves inspiring developers to apply AI in real-world applications. Committed to applying technology to societal challenges, Lize has embraced roles such as cochlear implant researcher at Ghent University, bioinformatics engineer for drug development software, and advisor to the Belgian government during COVID-19. In her free time, you will find her behind the piano or in her woodworking atelier.

<https://www.linkedin.com/in/lize-raes-a8a34110>



The main building blocks of LangChain4j are LLMs, but also image models, audio models and so called multi-modal models that do more than one of the aforementioned things at the same time (dealing with multiple kinds of input and output).

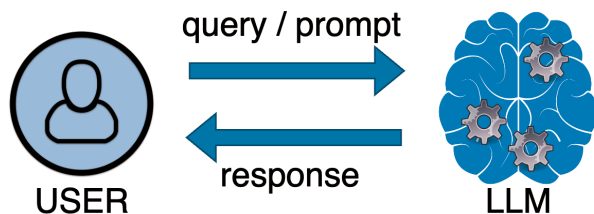
You can use vanilla LangChain4j, but we also have a smooth integration with Quarkus (see github.com/quarkiverse/quarkus-langchain4j) and Spring Boot (see [spring-boot-example in github.com/langchain4j/langchain4j-examples](https://github.com/langchain4j/langchain4j-examples)). Links to the relevant repositories can be found at the end of this article.

Architecture of AI-Powered Apps

Here we give a high-level overview of the components of LangChain4j. Later in the article we will dive in the details and the code for each of them.

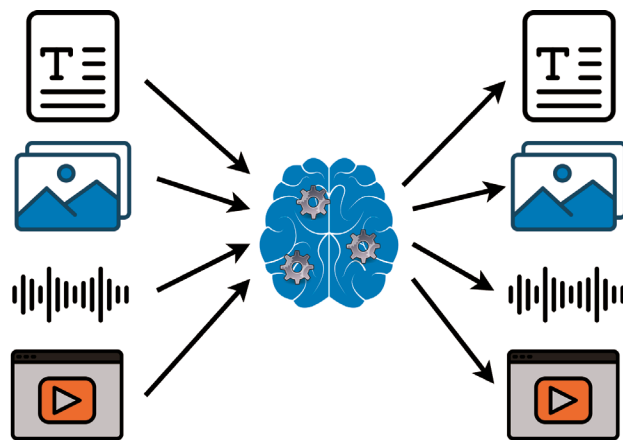
Basic Blocks

AI Models and Large Language Models (LLM): LangChain4j was initially built for LLMs as basic blocks.



The LLM itself is basically a huge calculation scheme that will take words in and produce an answer that (often sounds like it) makes sense. It's not important to understand how LLMs work under the hood to use LangChain4j. It is good to know that models vary in quality and capabilities, closely correlated with the model size.

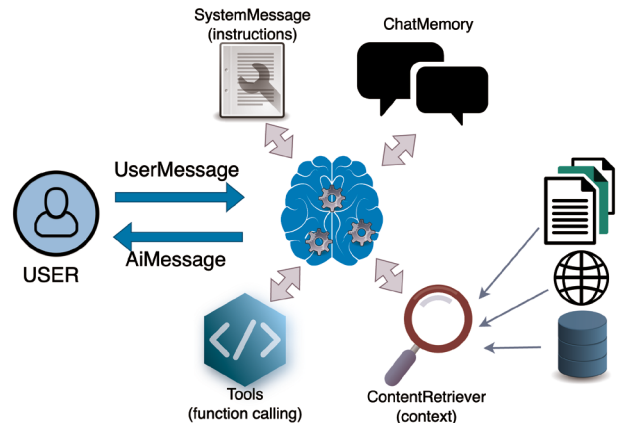
Other model types: LangChain4j supports specific image and audio models, as well as multimodal models that combine multiple input and/or output types. It is also possible to plug in and invoke any type of AI model, for example from HuggingFace (kind of Github for open source models).



Architecture of Augmented LLMs

An LLM can work with a number of surrounding blocks to

extend its functionality, making it very easy to turn them into very versatile and slightly risky agents. LLMs can be enhanced with general behavioral instructions, memory, extra context from content retrievers and the possibility to call tools to help it execute certain tasks. In the schema you see these components, along with their high level LangChain4j interface.



Hello world, how can I assist you today?

Now let's build something! To make a call to a model you will need to obtain an API key for the service you want to use. In this example we will be using OpenAI, you can obtain a key there really easily. I recommend storing the key in the environment variables to avoid accidentally pushing it to Github. You can also temporarily use LangChain4j's demo key, which we provide for free for demonstration purposes (does not support advanced functionality!)

We always import the langchain4j (core) functionality:

```
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j</artifactId>
<version>1.0.0-beta1</version>
```

You also need to import a model provider. For the code examples we are using `<artifactId>langchain4j-open-ai</artifactId>`, but you can easily use other model providers, for example `langchain4j-anthropic` etc. Now we're ready to code:

```
import dev.langchain4j.model.chat.
    ChatLanguageModel;
import dev.langchain4j.model.openai.
    OpenAiChatModel;
...
String apiKey = System.getenv("OPENAI_API_KEY");
// or String apiKey = "demo";

ChatLanguageModel model = OpenAiChatModel.builder()
    .apiKey(apiKey)
    .modelName("gpt-4o-mini")
    .build();

String answer = model.chat("Say ,Hello World");
System.out.println(answer);
```

This will sometimes return „Hello World“, and sometimes something else. Welcome to nondeterministic programming!

Using Models

Language Models and their parameters

Above we've seen the most basic invocation of OpenAI's (the provider) GPT4-0 mini (the model). We support many more commercial providers (where you need a key), such as Gemini, Azure OpenAI, Anthropic, Mistral, Amazon Bedrock, etc.

We also support running local models, via Ollama (this downloads OSS models for you and serves them on localhost) and JLLama (inference directly in Java, leveraging VectorAPI). Some reasons to use local models include privacy, cost, and also just using llama2-uncensored. An overview of all supported model integrations: docs.langchain4j.dev/integrations/language-models/

Each model comes with its own set of parameters that you can set to tweak its behavior. What parameters are available and what they do can be found on the model provider's website. This is how you set some of them for the OpenAI example:

```
ChatLanguageModel model = OpenAiChatModel.builder()
    .apiKey(ApiKeys.OPENAI_API_KEY)
    .modelName(GPT_4_0_MINI)
    .temperature(0.3)
    .timeout(ofSeconds(60))
    .logRequests(true)
    .logResponses(true)
    .build();
```

As you can see it allows us to set a timeout and define if requests and responses will be logged (try this in combination with log level debug to see the real calls to OpenAI). Higher values for temperature make the model more creative, lower values result in a more deterministic result.

The former example (`model.generate(String)`) returns a synchronous answer which includes waiting and then receiving lots of text at the same time. Models can also answer in streaming mode and return token by token, which allows for a much nicer user experience in chat applications. Tokens are chunks of text that are typically a couple of characters long. They are the model's minimal meaningful unit, but we don't really need to care, except that model prices are typically expressed per token.

```
StreamingChatLanguageModel model =
    OpenAiStreamingChatModel.builder()
        .apiKey(System.getenv(„OPENAI_API_KEY“))
        .modelName(GPT_4_0_MINI)
        .build();
```

```
String userMessage = „Write a poem about
NullPointers“;
model.chat(userMessage, new
```

```
StreamingChatResponseHandler() {

    @Override
    public void onPartialResponse(String
partialResponse) {
        System.out.print(partialResponse);
    }

    @Override
    public void onCompleteResponse(ChatResponse
completeResponse) {
        System.out.println(„\n\nDone streaming“);
    }

    @Override
    public void onError(Throwable error) {
        error.printStackTrace();
    }
});
```

Image Models

Image models generate images based on a text prompt. LangChain4j has integrations with Dall-E, Imagen and some more image models that you can find under docs.langchain4j.dev/category/image-models.

```
ImageModel model = OpenAiImageModel.builder()
    .apiKey(System.getenv(„OPENAI_API_
KEY“))
    .modelName(DALL_E_3)
    .build();

Response<Image> response = model.generate(„java
duke with loom, virtual threads and cat „);
System.out.println(response.content().url());
```

If you prefer storing the image directly, you can set the following parameters on the ImageModel:

```
ImageModel model = OpenAiImageModel.builder()
    .apiKey(System.getenv(„OPENAI_API_KEY“))
    .modelName(DALL_E_3)
    .quality(DALL_E_QUALITY_HD)
    .persistTo(Paths.get(„src/main/resources/
result-images“))
    .build();
```

The result (impossible to get the true Java Duke out of Dall-E :())



Funny cats, wasting computational resources since the invention of the internet

Make sure the folder is created beforehand. Image generation is calculation intensive. Generation can take a while and costs much more than text generation, for example \$0.02 for a Dall-E 2 image and \$0.08 for Dall-E 3.

Keep in mind that image generation is typically subject to rate limits that depend on your payment options, so before you launch your app on a big scale, make sure your subscription can keep up.

Audio models

LangChain4j supports speech-to-text conversion with Gemini (support for more models is coming). The input is a String of the audio file in base64 format. Gemini will typically take your audio instructions (like "who created Java" and immediately take action, by replying "James Gosling"). If instead you want to obtain a literal transcription of your audio, you can prompt it as in the code example below.

```
ChatLanguageModel model = GoogleAiGeminiChatModel.builder()
    .apiKey(System.getenv("GEMINI_TOKEN"))
    .modelName("gemini-1.5-flash-001")
    .build();

ChatResponse response = model.chat(
    SystemMessage.from("repeat the text from the audio message"),
    new UserMessage(AudioContent.from(base64, "audio/mp3")));

String message = response.aiMessage().text();
```

Message Roles

Before we continue, we need to introduce some important implementations of LangChain4j's ChatMessage API:

- **SystemMessage:** a message that we can optionally set to give general instructions to a model that will apply for the entire conversation. Eg. "You are a java programmer talking to a 3 year old".
- **UserMessage:** the user input (query, prompt)
- **AiMessage:** the reply of the model

They can be used like this:

```
UserMessage userMessage = UserMessage.from("How can I program my cat to fetch sticks for me?");
AiMessage firstAiMessage = model.chat(userMessage).aiMessage();
```

AI Services

Now we know how to perform basic interactions with models, I want to introduce you to a somewhat strange but very powerful concept in LangChain4j: AIServices, your companion for every problem. I introduce them early on because they make the syntax to plug in other components (memory, tools, ...) much more concise.

AIServices consist of an interface that describes what you want to happen. After which you can use a builder to create the implementation for you, specifying what components you want to add to it (model, RAG, tools, ...).

Some examples can explain this better than 1000 words. Here's a basic AIService:

```
interface Programmer {
    @SystemMessage("You are a java programmer talking to a 3 year old")
    String answer(String question);
}
```

The interface describes what we want: a Programmer object with one method: answer(String). We specify a SystemMessage (this is optional).

- The input is a String, so LangChain4j will infer that this is the UserMessage.
- The output is a String, so LangChain4j will automatically infer that this is the model output.

Now we want to create and use such a Programmer object (in the code below, model is a ChatLanguageModel that we created beforehand).

```
Programmer programmer = AIServices.create(Programmer.class, model);
```

```
String answer = programmer.answer(„How can
I program my cat to fetch sticks for me? Be
concise“);
```

Running this code will return something like (love this!):

Oh, little coder! You can't *program* a cat like a computer, but you can *train* it! Just like Java has loops, training takes **repetition**!

1. **Give treat** when kitty touches the stick.
2. **Move stick a little** and give treat again!
3. **Throw stick** and reward when kitty brings it back!

Like a while-loop:

```
`while(cat_fetches) { giveTreat(); }`
```

But remember, cats have their own „Garbage Collector“—they ignore stuff when bored!

Input parameters and structured outputs

Now starts the real fun! LangChain4j allows you to get java objects out of LLMs so you can just code on instead of parsing Strings all the time. Here are some examples of how to use these structured outputs, and what kind of combinations are possible in the input parameters with @UserMessage and @SystemMessage. TextUtils, our AIService, has 3 methods this time:

```
interface TextUtils {

    @SystemMessage(„You are a professional
translator into {{language}}“)
    @UserMessage(„Translate the following text:
{{text}}“)
    String translate(@V(„text“) String text,
@V(„language“) String language);

    @SystemMessage(„Summarize the user messag in
{{n}} bullet points. Provide only bullet points.“)
    List<String> summarize(@UserMessage String text,
@V(„n“) int n);

    @UserMessage(„Extract date and time from
{{it}}“)
    LocalDateTime extractDateTimeFrom(String text);
}
```

We create an instance, so we can actually use them:

```
TextUtils utils = AIServices.create(TextUtils.
class, model);
translate() will return a String, so the direct
```

output of the model.

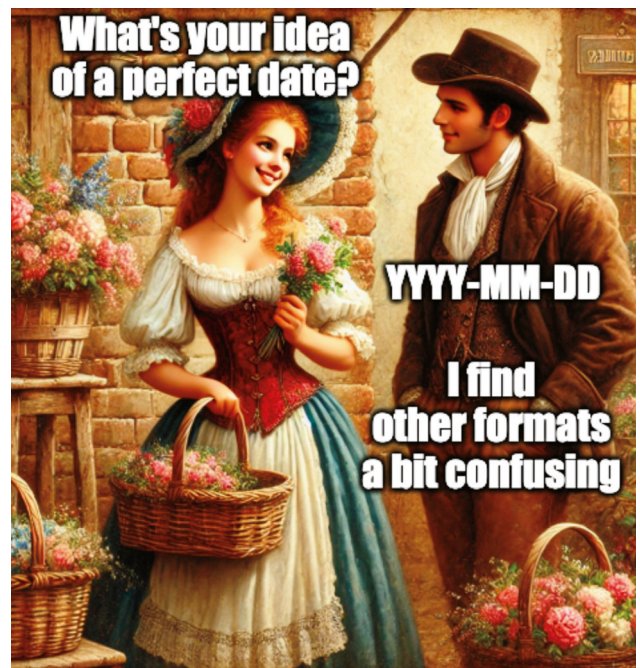
```
String translation = utils.translate(„Hello, how
are you?“, „italian“);
```

summarize() will return a Java object List<String>.

```
String text = „AI is a branch of computer science
that aims to create machines that mimic human
intelligence. E.g. recognizing patterns or making
decisions or predictions.“;
List<String> bulletPoints = utils.summarize(text,
3);
```

extractDateTimeFrom() is my favorite, done with endless parsing of date formats (if you can live with something that is mostly right but not always!)

```
text = „Three days before St. Valentine's Day of
2024, „ +
„just 15 minutes shy of midnight, „ +
„Jane discovered Jim was more in love with
his IDE than with her.“;
LocalDateTime dateTime = utils.
extractDateTimeFrom(text);
```



Good to know: you can also make them return a self-defined POJO 🧯 For example a Person object:

```
static class Person {
    @Description(„first name of a person“) //
optional description for LLM
    private String firstName;
    private String lastName;
```

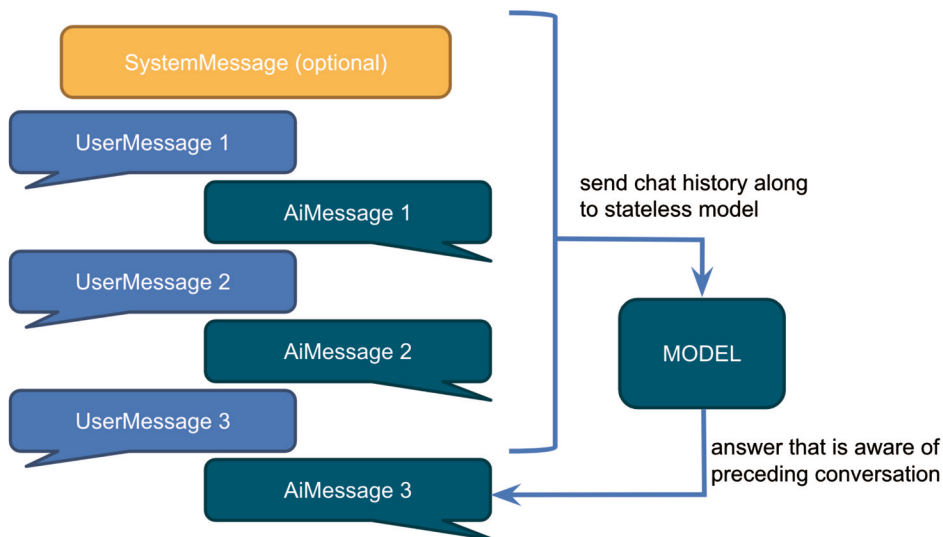
```
private LocalDate birthDate;
}
```

And an `AiService` to extract all Persons from text fragments (great for analyzing archives)

```
interface PersonExtractor {
    @UserMessage(„Extract all different persons from the following text: {{it}}“)
    List<Person> extractPersonFrom(String text);
}
```

Memory

Models are stateless, so if you want them to remember what has been said before, you have to send the history along every time. Most models' APIs provide a way to pass older messages from the conversation. LangChain4j wraps all the different syntaxes under one interface: `ChatMemory`. This is what memory looks like when we send it to an LLM



In theory we could keep the whole conversation in the memory, but it is sent along with every request. Memory eats tokens, and at some point we hit the size of the context window, the maximal number of tokens that a model accepts as input (model dependent). So we will need a strategy to reduce the memory size. LangChain4j comes with 2 default implementations:

- `MessageWindowChatMemory`, which will drop the oldest messages

```
ChatMemory chatMemory = MessageWindowChatMemory.
withMaxMessages(10);
```

- Or `TokenWindowChatMemory`, which will drop the oldest tokens

```
ChatMemory chatMemory = TokenWindowChatMemory.
withMaxTokens(1000, new OpenAiTokenizer());
```

`ChatMemory` is an interface, so you can write any implementation that makes sense for you, for example a `SummarizingChatMemory` that will keep track of important context based on your use case.

A `ChatMemory` is basically a list of `ChatMessages` (see before), and you can manually add the messages for every `UserMessage` you send and every `AiMessage` you receive back. But if you combine `ChatMemory` with an `AiService`, `LangChain4j` will take care of updating the memory for you. For example

```
Programmer programmer = AiServices.
builder(Programmer.class)
    .chatLanguageModel(model)
    .chatMemory(chatMemory)
    .build();
```

If you want to test if the memory is functioning properly, tell the model your name and ask 'what is my name' as the next question.

In a real application you will typically want to maintain and persist a memory per user. LangChain4j provides `MemoryId` functionality for this

```
interface Programmer {
    String chat(@
        MemoryId int memoryId,
        @UserMessage String
        userMessage);
}
```

In our `langchain4j-examples` repo (see link at the end), in the folder `tutorials`, you find

this example fully elaborated with the memory per user persisted to a database.

Few Shot Examples

Do you want your model to behave in a certain way, but it's not listening very well to your prompts and instructions? A more powerful technique is to pretend that the LLM behaved in the correct way by providing a fake memory pre-filled with examples, like this (if you need help setting boundaries ;))

```
fewShotHistory.add(UserMessage.from(„My mom
wants me to call every day.“)); fewShotHistory.
add(AiMessage.from(„You're not a podcast.“));
fewShotHistory.add(UserMessage.from(„My partner
always lets me do the laundry.“));
fewShotHistory.add(AiMessage.from(„Did their hands
fall off?“));
```

```
fewShotHistory.add(UserMessage.from(„Toddler is
crying bcs he didn’t get the pink cup.“));
fewShotHistory.add(AiMessage.from(„Tragic. For
them.“));
```

Now add this `fewShotHistory` as memory to your `AiService`, ask it anything and enjoy your boundary coach!

Tools

Until now the models have returned text, images and java objects. A lot of models have an extra capability that makes them really powerful (and dangerous): calling your code.

You can provide a method that a model can call, such as sending an email, calculating interest or writing something to the database. Models that support function calling (tool calling) will reply with a function call including method name and parameter values when they think the question requires using the tool. We will run the called method and send the result back to the model, which will use the answer in its final reply.

`LangChain4j` makes that super easy, by using the `@Tool` annotation as follows (the tools are part of class `ShoppingTools` in this example)

```
@Tool
public List<String> getProductList() {
    // custom code to obtain the list
}

@Tool(„returns product description, price and
availability“)
public String getProductDetails(@P(„5 digits
product ID“) int prodID) {
    // custom code to obtain the product details
}
```

If we have a `ShoppingAssistant` interface declared with a method `giveProductSuggestion(String userMessage)`, we can make the tools available as follows

```
ShoppingAssistant shoppingAssistant = AiServices.
builder(ShoppingAssistant.class)
    .chatLanguageModel(model)
    .tools(new ShoppingTools())
```

If we now run

```
shoppingAssistant.giveProductSuggestion(„I’m
looking for a statue of Yoda that can sing Xmas
songs“);
```

the model will (in most of the cases) call `getProductList()`, see if there’s anything in there that seems to match my request, and if yes, it will call `getProductDetails()` with the

`productID` that it got as a result from `getProductList()`. It will then use all that information to craft a reply, like “It seems like we don’t have a Yoda statue that sings Christmas songs, but I did find a singing Christmas goat in our catalog for 15.99\$”.

Good models are able to call tools sequentially and in parallel. In the example, it will first ask for the product list, and only then get specific product details. If it finds multiple singing Yoda’s that might carry my fancy, it will then call `getProductDetails` multiple times in parallel with the different product IDs.

Model calls and its subsequent tool calls are often high latency, so virtual threads come in handy to free resources while waiting for results.

You can do a lot more things with tools: adding tools from different classes, specifying tools dynamically, returning the tool result as final answer, ... you can find how to do all that in our documentation docs.langchain4j.dev/tutorials/tools.

Human as a tool

LLMs executing tools, placing orders and accessing the database is dangerous. Depending on the risks or resources involved in executing a tool call, you may want to have some hard code checks in place (for example authentication) or ask the user for permission to proceed. The latter is called human-as-a-tool or human-in-the-loop, and can be obtained by adding a request for permission to run the tool as first thing in the tool code. If the answer is yes, the code continues, if no, you can throw an error back to the model that the user (or the verification code) did not allow the operation.

MCP servers

If you’re anything like me, you’re probably hyperventilating by now and seeing all kinds of new automations that will make your life so much easier. Triage my emails, extract the TODOs, place them smartly in my agenda based on priorities and my existing calendar entries and and and... The new bottleneck for developers is writing all the tools that allow the LLM to execute these requests.

At the moment integrating with different services requires a different syntax and sometimes execution environment for almost anything you try to do. Github? git + ssh. Database? SQL + sqlplus. Local file system? Bash + terminal. Weather service? Custom REST API + curl. That’s a steep learning curve and quite hard to automate.

But the good news is, last November, Anthropic published the Model Context Protocol (MCP), an open standard for exposing LLM-readable tools. The protocol got a lot of traction, and by now you can find hundreds of MCP servers on Github, just google ‘MCP server PostgreSQL’ (or Grafana, Github, Atlassian Gmail, ... you name it). You can also create and publish your own MCP server really easily using Quarkus. Once you find a server, you can use it via `LangChain4j`’s integration like this:


```

McpTransport transport = new StdioMcpTransport.
Builder()
    .command(List.of(„/usr/local/bin/docker“,
„run“, „-e“, „GITHUB_PERSONAL_ACCESS_TOKEN“, „-i“,
„mcp/github“))
    .logEvents(true)
    .build();

McpClient mcpClient = new DefaultMcpClient.
Builder()
    .transport(transport)
    .build();

ToolProvider toolProvider = McpToolProvider.
builder()
    .mcpClients(List.of(mcpClient))
    .build();

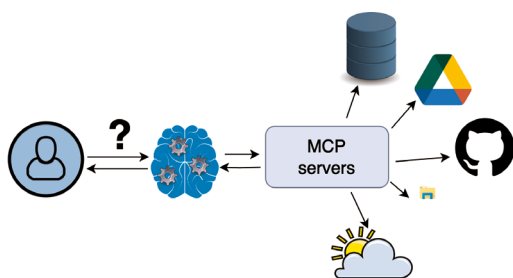
Bot bot = AiServices.builder(Bot.class)
    .chatLanguageModel(model)
    .toolProvider(toolProvider)
    .build();

```

You'll typically need some API key to connect to services giving access to personal information, like `GITHUB_PERSONAL_ACCESS_TOKEN` in the example.

This example uses `stdio` for communicating with your process. There are also servers that use `http`, see `langchain4j-examples` for the details.

The server exposes descriptions of available methods and parameters to the LLM. The server also takes care of translating parameters it is called with into the right syntax, and to execute the request, skipping all the difficulty of the previous diagram. The example above let's an LLM create new branches and inspect PRs. Beautiful. Dangerous.



Important note: LLMs can be manipulated and unpredictable. Watch out before granting your LLM the rights to wipe your database or send emails autonomously!

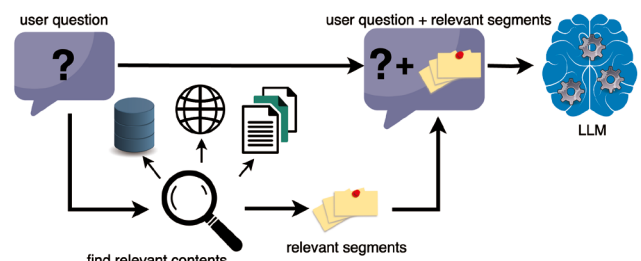
Career tip: Developers may be at risk of losing their job to AI, but if you go into (LLM) security, you're guaranteed to stay in demand.



RAG

By now we can build really interesting LLM-powered apps. One last thing to address is that each model was trained on a specific dataset. We can't hope for it to give us any information that was not present in its training data, such as the weather today, your company-specific procedures, your product database. In the best case it will say that it doesn't know, in the worst (and more prevalent) case, it will hallucinate something that sounds plausible. So we need a way to let the model know about all that extra information that was not part of its training data.

One solution would be to ask our question and send all the available data along. It would probably work but you will pay a lot for all those tokens. If everything would fit in the context window in the first place. That is why we want to only pass the extra information to the model that is relevant to our question. This is called Retrieval-Augmented Generation (RAG) and this is how it looks schematically:

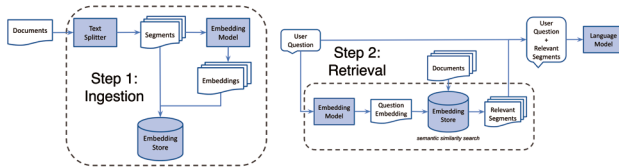


Side note: if you think RAG is an ugly name, so do its creators. Quote Patrick Lewis: "We always planned to have a nicer sounding name, but when it was time to write the paper, no one had a better idea."

The relevant fragments of information, we need to search in the context that we offer to the model. `LangChain4j` has an interface `ContentRetriever` where you can implement your own retriever. We also provide the 4 most used ways out of the box (examples for each can be found under `/rag` in `langchain4j-examples`):

- **WebSearchEngineContentRetriever:** the LLM turns the original prompt into a web search query and a number of search results are used as context
- **SqlContentRetriever:** the LLM is given the database schema and turns the original prompt into SQL to retrieve information that will be used as context
- **Neo4jContentRetriever:** the LLM is given the schema and turns the original prompt into Cypher (neo4j query) to retrieve information that will be used as context
- **EmbeddingStoreContentRetriever:** to retrieve relevant fragments from all documents that we provide (text, excel, images, audio, ...). To find relevant fragments, we need to prepare our sources in advance in an ingestion step: We first chunk our sources into fragments, then we calculate a numerical vector representation (embedding) that encodes the meaning of each fragment (or image, table, ...), using an embedding model. Finally, all embeddings are saved in an `EmbeddingStore` together with the original fragments. When the user asks a question, we will create

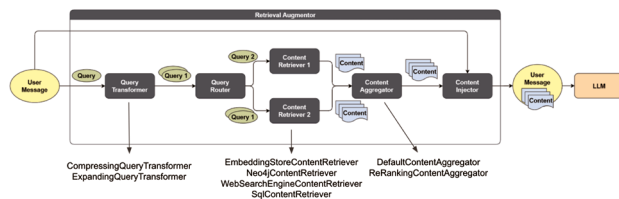
an embedding of the question and retrieve the closest matches in our EmbeddingStore. This is called semantic similarity search.



Lazy tip: In Quarkus, you can add an EmbeddingStoreContentRetriever to your AiService by simply adding the dependency quarkus-langchain4j-easy-rag and specifying quarkus.langchain4j.easy-rag.path in application.properties. This will use all documents stored in that folder as RAG sources.

Expert tip: Embeddings (vectors) can also be used to group topics or to find categories in data, by grouping the closest vectors together.

RAG works much better when we add some advanced RAG components around the ContentRetriever and its ContentInjector. Here are the available interfaces with their out-of-the-box implementations mentioned underneath:



- QueryTransformers: they are very important to make RAG work properly. Imagine your RAG system has access to web search. The LLM asks you if you want to proceed with, say, the selected products. You say 'yes please', and off it goes searching the internet for 'yes please'. You don't want that, especially not in production. You will want a CompressingQueryTransformer that will take the rest of the conversation into account when forming its query, and turn 'yes please' in eg. 'proceed with selection of products: singing Yoda and Christmas goat'. ExpandingQueryTransformer works similarly, but may create multiple queries to ensure an answer is found.
- If you wish to use different ContentRetrievers in parallel, you can store them in a map, like this

```
Map<ContentRetriever, String>
retrieverToDescription:
literatureDocsRetriever -> „Scientific literature on
antibodies“
webSearchContentRetriever -> „Web search“
sequenceDbContentRetriever -> „Protein database
with sequences“
```

- A QueryRouter will choose which ContentRetriever(s) the question should be sent to. You can write an own implementation, for example based on keywords, or use the LanguageModelQueryRouter that uses another LLM to route the question and transform it into queries for the

different retriever types.

- ContentAggregator: the DefaultContentAggregator will simply put all selected fragments one after the other. However, very often (especially with EmbeddingStoreContentRetriever) fragments are found that are not related to the question and will only confuse the LLM if we send them along. You can use a ReRankingQueryTransformer to remove unrelated fragments. It uses a tiny and super fast scoring model specialized in scoring how relevant a piece of information is for a question.

```
RetrievalAugmentor retrievalAugmentor =
DefaultRetrievalAugmentor.builder()
    .queryTransformer(queryTransformer)
    .queryRouter(queryRouter)
    .contentAggregator(contentAggregator)
    .contentInjector(DefaultContentInjector.
builder()
    .promptTemplate(
        PromptTemplate.
from(„{{userMessage}}\n“ + „\n“ +
„you can use the following information:\n“ +
„{{contents}}“)
    ).build())
    .build();
```

We can add this RetrievalAugmentor to our Programmer AiService like this

```
AiServices.builder(Programmer.class)
    .chatLanguageModel(model)
    .retrievalAugmentor(retrievalAugmentor)
    .build();
```

Good to know: In RAG everything happens in parallel and in one shot. This means it will fail for questions that require sequential searching like 'find the sequence of an antigen of disease xyz', because it will need to find the antigen first before it can query the database for it's sequence (that would be step 2). If you want to allow iterative searching, you can wrap your ContentRetrievers as Tools and the model will know which ones to call in which order.

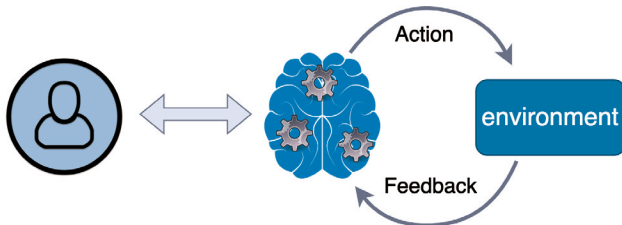
In github.com/langchain4j/langchain4j-examples/tree/main/rag-examples you can find examples of all the above, as well as examples where sources are retrieved and passed along, and an example of the use of metadata stored with the fragments (to store read permissions or sources for example).

Agentic Systems

AI Agents is a very wide term that can cover anything from an LLM augmented with tools to a full-blown complex workflow system with multiple LLMs, orchestrators, workers, gatekeepers and sophisticated interactions with the environment.

Basic Agent: LLM calls code

As a rule of thumb, build only the complexity that you need. For example, commercial LLMs can often perform quite complex tasks if you tell them the steps they need to run through, and the tools they need to use. If their tools include means to perform actions and observe the result, we have a basic AI Agent.



As you can see, a LangChain4j AiService can easily do that, the programmatic difficulty lies mainly in creating good tools. MCP servers made that part a lot easier.

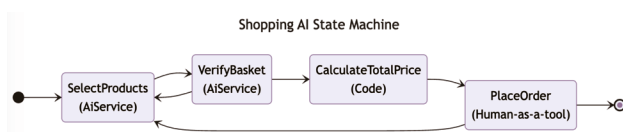
State Machine: Code calls LLM calls code

If you need more control over the workflow, you may consider structuring your process as a state machine, with the different steps (nodes) and transition criteria (edges) programmatically defined. The functionality inside of the nodes can be performed by AiServices (like determining which financial profile the customer has via guided chat) or nodes can be purely algorithmic (like running the calculations for the loan based on the financial profile - where using an LLM makes no sense).

The advantage here is more control and separation of concerns (and responsibilities) so LLMs don't overstep their boundaries. It makes it much easier to add security checks and limit permissions for each node. It also allows us to outsource to code what better happens in code, and to pick tiny models for simple tasks and specialized models for specialized tasks. For example, for node transitions after a chat, you often get a situation like

Model: Can you confirm your product selection?
User: I still want to add a singing goat doll!

To determine whether this means that you can move on to the next node (verify basket), or you stay in the current node a bit longer (select products), you can easily train a Bayesian classifier on 30 examples. It will do a great job and requires orders of magnitude less resources and latency.



It's easier to program than it sounds: state machines are easy to code, and you can call AiServices in the nodes or on the edges, wherever it makes sense. Often you'll also need an object to store the state, eg. the state of the shopping basket, the user profile, ...

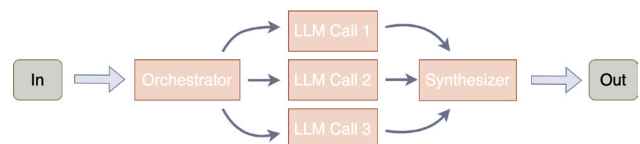
Combinations

It gets really interesting when we wrap AiServices in tools. For example:

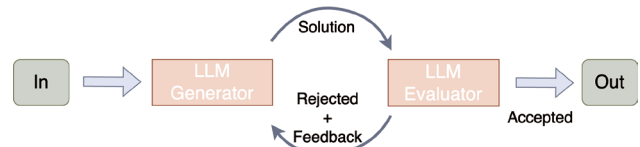


```
@Tool
void determineFinancialProfile(){
    FinancialProfileAiService.determineProfile(); //
    AiService
    // that has access to chat and tools
    // to complete a FinancialProfile object that
    keeps the state
}
```

Now we can let another AiService, for example BankChatBot (the orchestrator) determine which other AiServices (wrapped in tools) to call. When the user asks the BankChatBot for a loan for example, the bot will know to call determineFinancialProfile() and checkDebtRecords(String personID). That would be a variant of this architecture



If the quality of the result is important, you can implement a pattern where a second LLM or AiService evaluates the first one:



All these patterns are combinable, adaptable to your specific use case and surprisingly easy to program using AiServices and Tools. Where suitable tools can wrap other AiServices.

An important thing to keep in mind here is that model calls and tool calls introduce significant latency: depending on model and number of consumed/generated tokens it can be anything from 100s of milliseconds to 10s of seconds. So if your process allows it, keep the number of LLM calls low and use small (fast, cheap) specialized models.

Further reading: some more agentic architectures are covered in this article by Anthropic www.anthropic.com/research/building-effective-agents

Production Features

LLMs + production = nightmare? I feel you! But we can mitigate a lot with a number of more production-oriented features. In the second part of this JAVAPRO edition, we'll get a whole article dedicated to those, but for the people that can't wait,

JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



www.javapro.io

here are a couple of pointers to get you started (more details can be found online).

Moderation: some models like AzureOpenAI come with built-in moderation that will check the model output for aspects like violence, so it allows you to filter out those responses.

GuardRails: at the time of writing only available in quarkus-langchain4j, but coming soon to vanilla LangChain4j. InputGuardrails and OutputGuardrails are very powerful gatekeepers that will check if the prompt or the model input/output match certain criteria before sending it on. You can even use another LLM to do the check.

Observability: ChatModelListeners allow you to observe things like inputTokenCount to calculate cost, or finishReason to keep statistics of failed calls.

Testing non-deterministic apps: tests with retries come with Junit Pioneer, or even better (only in quarkus-langchain4j for now) @QuarkusTest allows you to give a score to your test output instead of having a clear pass vs. fail scenario. You can use scoring strategies like AiJudgeStrategy and SemanticSimilarityStrategy or create your own custom implementation.

Human feedback: at my company Naboo.ai, we noticed that having all the above in place is still not sufficient to guarantee an excellent user experience. If, like us, you deal with user queries as input ("ask anything regarding your project and integrated tools"), the variety of what users will ask beats anything you could have predicted upfront. It even beats what the overzealous testers could come up with!

We solved this by collecting user feedback (thumbs up, or more important for us: thumbs down). If you want to offer a great user experience, consider a pilot phase where you manually review the



creative caprices of your customers and extend your system to cover them. Once in production, you can continue gathering this feedback and use embeddings to find the most prevalent negative feedback categories and address those.

Important links

- LangChain4j repo: github.com/langchain4j/langchain4j
- The dreaded but beautiful documentation (It doesn't bite! It's trying to be likeable with tutorials): docs.langchain4j.dev
- Community incubator repo for new integrations: github.com/langchain4j/langchain4j-community
- Examples repo: github.com/langchain4j/langchain4j-examples
- Awesome LangChain4j examples: github.com/langchain4j/awesome-langchain4j
- LangChain4j Discord: discord.com/invite/JzTFvyjG6R

Thank You and FAQ

- LangChain4j was created in 2023 by Dmytro Liubarskyi, who is still its relentless lead engineer. Thank you so much for your awesome work!
- **Immense thanks to our 200+ contributors!** Proudly listing the top 20 here, but everyone that didn't make it to this list, we see you! Thank you Julien Dubois, Crutcher Dunnivant, Martin7-1, Konstantin Pavlov, jiangsier-xyz, Guillaume Laforge, Antonio Goncalves, Eddú Meléndez Gonzales, omarmahamid, Carlos Zela Bueno, Alexey Titov, Jan Martiska, xermaor, Georgios Andrianakis, kuraleta, bidek, Julien Perrochet, PrimosK, Eric Deandrea, Mario Fusco and all the other contributors
- **LangChain4j** is used in production in projects like MuleSoft by Salesforce, Advanced Coding Assistant by Deutsche Telekom, Apache Camel, MicroProfile, Wildfly, Micronaut and many more projects. We're proud ^^
- **Where are the chains in LangChain4j?** Well erm, just like RAG, names are sticky... we abandoned the chains for the more versatile AiServices
- **How do I contribute?** You are very welcome to contribute! Bug fixes may not be the most glamorous thing, but they actually help us a ton and we appreciate it tremendously. For more tips and tricks, the contributor guide is linked in the readme.
- **Aren't the swag and the model licenses for testing expensive?** Yes they are, so feel free to drop us some coins via opencollective.com/langchain4j for a swaggier future
- **How much does it cost to use GPT4, or Gemini, or...** ? Not that much, especially compared to the value they may bring to your product. Model pricing depends on the model quality (which you can assess using benchmarks, eg. www.livebench.ai) and are calculated in function of input tokens (less expensive) and output tokens (more expensive). For a pricing overview, check llm-price.com. Tokens typically are about 3 characters long, so to give an idea, with GPT4 you get around 12 A4 pages output for 1\$, and for claude-3.5-sonnet that would be 48 pages for the same price.
- **Just how good is a local model?** It depends a lot on how much GPU you have available to run it on! For 700GB GPU you can run deepseek-r1, which performs in the top 5 of best models at the moment of writing, or Llama-3.1-405B-Instruct which performs in the top 30. But you will need specialized, expensive hardware. Models of around 70GB can run reasonably fast on a laptop, for example a Macbook Pro with enough unified memory (GPU+CPU) and still perform reasonably well. If you want to serve multiple users simultaneously, you need to do some research on the latency though. Then there are the really small models like Microsoft's phi3, which is under 4GB. Some of those tiny models still perform very well for just language interaction with humans, but fail on more complex tasks and are very unreliable for tool calling.

Langchain4J Musings

I'm coming relatively late to the LLM party, but I rarely come very early in the hype cycle.



For example, I never bought into blockchain, the solution still searching for problems to solve, nor in microservices, the latest in the cargo cult IT trends. Despite my late arrival at the LLM party, I have been a regular user of LLMs. I use OpenAI for non-controversial questions outside my cone of knowledge, e.g., linguistics or legal; I use GitHub Copilot in my IDE to improve my code.

The main focus of this post is to integrate a chatbot into my application and explore its capabilities.

Choosing a LLM

A plethora of LLMs is available at the moment. I mentioned OpenAI, but plenty of others beg for your attention: Google Gemini, Cohere, Amazon Bedrock, and many others. Each has pros and cons, which are irrelevant to this introductory post.

My main requirement in the context of this post is that it needs to run locally. Besides, I want an abstraction layer over the LLM to learn the abstractions, not the specifics.

I chose LangChain4J and Ollama because they are well-known and meet my specific requirements for this project.

Quick introduction to LangChain4J and Ollama

Here's how LangChain4J introduces itself in its own words:

The goal of LangChain4j is to simplify integrating LLMs into Java applications.

Here's how:

- 1. Unified APIs:** LLM providers (like OpenAI or Google Vertex AI) and embedding (vector) stores (such as Pinecone or Milvus) use proprietary APIs. LangChain4j offers a unified API to avoid the need for learning and implementing specific APIs for each of them. To experiment with different LLMs or embedding stores, you can easily switch between them without the need to rewrite your code. LangChain4j currently supports 15+ popular LLM providers and 20+ embedding stores.
- 2. Comprehensive Toolbox:** Since early 2023, the community has been building numerous LLM-powered applications, identifying common abstractions, patterns, and techniques. LangChain4j has refined these into a ready to use package. Our toolbox includes tools ranging from low-level prompt templating, chat memory management, and function calling to high-level patterns like AI Services and RAG. For each abstraction, we provide an interface along with multiple ready-to-use implementations based on common techniques. Whether you're building a chatbot or developing a RAG with a complete pipeline from data ingestion to retrieval, LangChain4j offers a wide variety of options.
- 3. Numerous Examples:** These examples showcase how to begin creating various LLM-powered applications, providing inspiration and enabling you to start building quickly.

Ollama's introduction is even shorter:

Get up and running with large language models.

Run Llama 3.2, Phi 3, Mistral, Gemma 2, and other models. Customize and create your own

— <https://ollama.com/>

One runtime, multiple models.

Getting our feet wet

I'll split this section into the LangChain4j app and the Ollama infrastructure.

Author:



Nicolas Fränkel is a Developer Advocate with 15+ years experience consulting for many different customers, in a wide range of contexts (such as telecoms, banking, insurances, large retail and public sector). He usually works on Java/Java EE and Spring technologies, but with focused interests like Rich Internet Applications, Testing, CI/CD and DevOps. He also doubles as a trainer and triples as a book author.

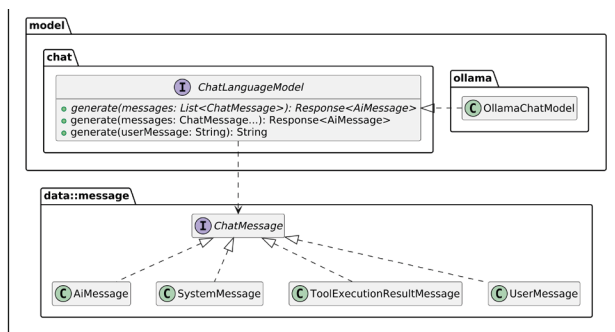
https://x.com/nicolas_frinkel
<https://mastodon.top/@frinkel>
<https://bsky.app/profile/frinkel.ch>

The LangChain4j app

LangChain4j provides a Spring Boot integration starter. Here's our minimal dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</
  artifactId>
</dependency>
  <dependency>
    <groupId>dev.langchain4j</groupId>
    <artifactId>langchain4j-ollama-spring-boot-
  starter</artifactId>
    <version>0.35.0</version>
  </dependency>
</dependencies>
```

LangChain4j offers an abstraction API over the specifics of



The fundamental API `model.generate(String)` passes the user's message to the Ollama instance and returns its response. We need to create an endpoint to wrap the call; the details are unimportant.

LangChain4j's Spring Boot starter automatically creates a `ChatLanguageModel` from the exact dependency set - here, Ollama. Furthermore, it offers lots of configuration options via Spring Boot.

```
langchain4j.ollama.chat-model:
  base-url: http://localhost:11434
#1
  model-name: llama3.2
#2
```

1. Point to the running Ollama instance
2. Model to use

When the app starts, LangChain4j creates a bean of type `ChatLanguageModel` and adds it to the context. Note that the concrete type depends on the dependency found on the classpath.

The Ollama infrastructure

For ease of use, I'll use Docker, and more specifically Docker Compose. Here's my Compose file:

```
services:
  langchain4j:
    build:
      context: .
    environment:
      LANGCHAIN4J_OLLAMA_CHAT_MODEL_BASE_URL:
        http://ollama:11434 #1
    ports:
      - „8080:8080“
    depends_on:
      - ollama
  ollama:
    image: ollama/ollama
#2
  volumes:
    - ./ollama:/root/.ollama
#3
```

3. Override the URL configured in the JAR to use the Docker container on Docker Compose
4. Use the latest images; it's not production
5. Keep a copy of the models on the host - see below

As mentioned above, Ollama is a runtime with switchable models. There's no model by default. To download a model, docker exec into the container and run the following command:

```
ollama run llama3.2
```

Be careful, llama3.2 is a whopping 20Gb; for this reason, you want to avoid downloading the model from each docker compose up. This is the reason for the volume mapping above.

Of course, you can substitute llama3.2 with any other smaller model, e.g., tinylama.

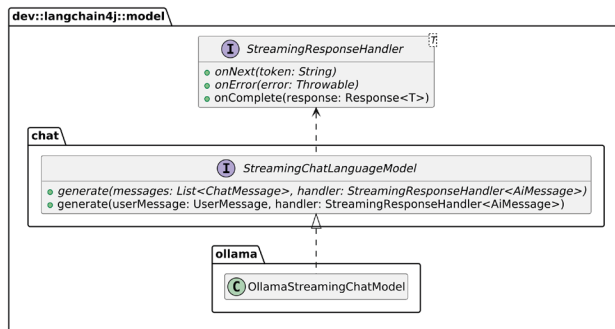
At this point, we can curl our app and see the results:

```
curl localhost:8080 -d „Hello I am Nicolas and I am
a DevRel“
```

Enhancing with streaming

The above solution works, but the user experience has room for improvement. The command hangs, and the response comes after several seconds, unlike the traditional OpenAI UI, which streams tokens back to the user.

We can readily replace `ChatLanguageModel` with `StreamingChatLanguageModel` to achieve this. Methods are slightly different:



We need to change the app configuration accordingly:

```
services:
  langchain4j:
    build:
      context: .
      environment:
        LANGCHAIN4J_OLLAMA_STREAMING_CHAT_MODEL_BASE_URL: http://ollama:11434 #1
      ports:
        - „8080:8080“
      depends_on:
        - ollama
```

1. Was formerly `LANGCHAIN4J_OLLAMA_CHAT_MODEL_BASE_URL`

In parallel, we must migrate from Spring Web MVC to Spring Webflux. Then, we pipe the LLM result stream to the app result stream like so:

```
class AppStreamingResponseHandler(private val sink: Sinks.Many<String>) :
  StreamingResponseHandler<AiMessage> {

    override fun onNext(token: String) {
      //1
      sink.tryEmitNext(token)
    }

    override fun onError(error: Throwable) {
      //1
      sink.tryEmitError(error)
    }

    override fun onComplete(response: Response<AiMessage>) {
      //2
      println(response.content()?.text())
      sink.tryEmitComplete()
    }
  }

class PromptHandler(private val model: StreamingChatLanguageModel) {
```

```
suspend fun handle(req: ServerRequest):
  ServerResponse {
    val prompt = req.awaitBody<String>()
    //3

    val sink = Sinks.many().unicast().
    onBackpressureBuffer<String>() //4

    model.generate(prompt,
    AppStreamingResponseHandler(sink)) //5

    return ServerResponse.ok().
    bodyAndAwait(sink.asFlux().asFlow()) //6
  }
}
```

1. Pipe tokens and errors to the sink
2. The function is not abstract and does nothing; hence, it won't close the stream. Remember to override it.
3. Get the request body asynchronously
4. Create a the sink
5. Call the model and pass the sink as a reference
6. Return the sink

We can now use curl in stream mode with the `-N` flag:

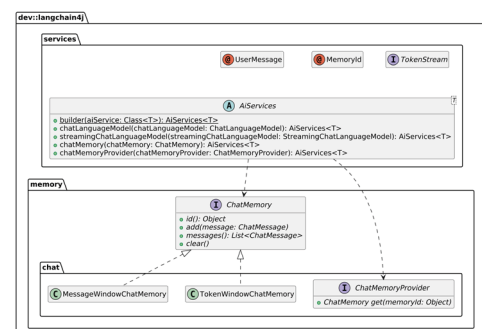
```
curl -N localhost:8080 -d „Hello I am Nicolas and I am a DevRel“
```

The result is already better!

Remembering history

Every chatbot request is independent of others at this stage - they don't keep a context. Chat history is an important feature that we miss from off-the-shelf AI assistants. We need to refactor the app in two directions: first, store each message from the user and the model, and second, compartmentalize users' histories from each other.

I started to store the history by myself in memory at first. If interested, check the commit history to see how I did it. However, LangChain4j offers an integrated approach via its `AiServices` class. `ChatLanguageModel` represents the basic request-response interface to the LLM, while `AiServices` wraps additional services: chat memory, RAG, and external function calls.



Here's the relevant code:

```
data class StructuredMessage(val sessionId: String,
val text: String)
//1
interface ChatBot {
//2 fun talk(@MemoryId sessionId: String, @User-
Message message: String): TokenStream //3-4-5
}

class PromptHandler(private val chatBot: ChatBot) {

    suspend fun handle(req: ServerRequest): Server-
Response {
        val message = req.awaitBody<StructuredMess-
age>()
        val sink = Sinks.many().unicast().onBack-
pressureBuffer<String>()
        chatBot.talk(message.sessionId, message.
text) //6
        .onNext(sink::tryEmitNext)
//7
        .onError(sink::tryEmitError)
//7
        .onComplete { sink.tryEmitComplete()
} //7
        .start()
        return ServerResponse.ok().bodyAndA-
wait(sink.asFlux().asFlow())
    }
}

fun beans() = beans {
    bean {
        coRouter {
            val chatBot = AiServices
//8
                .builder(ChatBot::class.java)
                .streamingChatLanguageMo-
del(ref<StreamingChatLanguageModel>())
                .chatMemoryProvider { MessageWind-
owChatMemory.
                    withMaxMessages(40) }
                .build()
            POST(,,/")(PromptHandler(chatBot)::hand-
le)
        }
    }
}
```

1. We need a way to pass a correlation ID to group messages with the same chat history. Given we are using curl and not a browser, we explicitly pass an ID along with the user messageThe function is not abstract and does nothing; hence, it won't close the stream. Remember to override it.
2. Define an interface with no hierarchy requirements. Functions are free-form, but you can set hintsCreate a the sink

3. @MemoryId marks the correlation ID
4. @UserMessage marks the message sent from the user to the model
5. TokenStream you can subscribe to
6. LangChain4j calls the configured model
7. Pipe the TokenStream to the sink as in our custom implementation
8. Build the ChatBot: AiServices will create the implementation at runtime

Here's how to use it:

```
curl -N -H ,Content-Type: application/json'
localhost:8080 -d ,{ „sessionId“: „1“, „message“:
„Hello I am Nicolas and I am a DevRel“ }'
curl -N -H ,Content-Type: application/json'
localhost:8080 -d ,{ „sessionId“: „2“, „message“:
„Hello I am Jane Doe and I am a test sample“ }'
```

Adding Retrieval-Augmented Generation

LLMs are only as good as the data they are trained on, and there's a high chance you want your chatbot to be trained on your own custom data. RAG is the answer to this problem. The idea is to index content ahead of time, store it somewhere, and add the indexed data to the search - called retrieval. For more details, LangChain4j does a great job of explaining RAG.

In this section, we will add an embryo of RAG to our app using data from my blog.

LangChain4j offers a dependency literally called Easy RAG. It provides two sources, files and URLs, and an in-memory embedding store. In a regular app, you would index offline and store embeddings in a regular database, but we will do it in memory at startup time. It's good enough for our prototyping purposes.

```
class BlogDataLoader(private val embeddingStore:
EmbeddingStore<TextSegment>) {

    private val urls = arrayOf(
        „https://blog.frankel.ch/speaking/“,
        // Other URLs
    )

    @EventListener(ApplicationStartedEvent::class)
//1
    fun onApplicationStarted() {
        val parser = TextDocumentParser()
        val documents = urls.map {
            UrlDocumentLoader.load(it, parser) }
        EmbeddingStoreIngester.ingest(documents,
embeddingStore)
```

```

    }
    fun beans() = beans {
        bean<EmbeddingStore<TextSegment>> {
            InMemoryEmbeddingStore<TextSegment>()
        } //2
        bean {
            BlogDataLoader(ref<EmbeddingStore<TextSegment>>()) //3
        }
        bean {
            coRouter {
                val chatBot = AIServices
                    .builder(ChatBot::class.java)
                    .streamingChatLanguageModel(
                        ref<StreamingChatLanguageModel>()
                    )
                    .chatMemoryProvider {
                        MessageWindowChatMemory.withMaxMessages(40)
                    }
                    .contentRetriever(EmbeddingStoreContentRetriever
                        .from(ref<EmbeddingStore<TextSegment>>())) //4
                    .build()
            }
        }
    }
}

```

1. Run the code when the application starts
2. Define the embedding store. Regular applications should use a persistent data store: LangChain4j supports more than a few.
3. Inject the store in the loader code
4. Configure the chatbot to retrieve data from the store

We can test the RAG by asking questions related to the documents ingested.

On OpenAI, I asked, „What books did Nicolas Fränkel write?“. It answered: Learning Vaadin (correct), Spring Security in Action (could be, but it's hallucinating), and Mastering Java EE Development with WildFly (no chance, and it's hallucinating again).

Let's do the same on the RAG'ed app:

```

curl -N -H ,Content-Type: application/json'
localhost:8080 -d ,{ „sessionId“: „1“, „message“:
„What books did Nicolas Fränkel write?“ }'

```

The answer is much better:

The provided information doesn't mention specific books written by Nicolas Fränkel. It only provides metadata for his blog, which has a section dedicated to his „Books“. ...

It's not really correct—I actually mentioned that I wrote the books mentioned, but it's at least not hallucinating.

We can test the RAG by asking questions related to the documents ingested.

On OpenAI, I asked, „What books did Nicolas Fränkel write?“. It answered: Learning Vaadin (correct), Spring Security in Action (could be, but it's hallucinating), and Mastering Java EE Development with WildFly (no chance, and it's hallucinating again).

Let's do the same on the RAG'ed app:

```

curl -N -H ,Content-Type: application/json'
localhost:8080 -d ,{ „sessionId“: „1“, „message“:
„What books did Nicolas Fränkel write?“ }'

```

The answer is much better:

The provided information doesn't mention specific books written by Nicolas Fränkel. It only provides metadata for his blog, which has a section dedicated to his „Books“. ...

It's not really correct—I actually mentioned that I wrote the books mentioned, but it's at least not hallucinating.

Conclusion

In this post, I showed how to start your Langchain4j journey in several incremental steps.

First, we used Langchain4j as a simple façade over Ollama.

Then, we switched to streaming tokens. We refactored the codebase to add chat history using Langchain4j's abstractions.

We finished the demo by adding RAG via a simple in-memory store and static links.

The complete source code for this post can be found on GitHub.

To go further:

- LangChain4j
- Ollama
- Streaming with REST API for LangChain Applications

Adapting Java for Modern APIs: REST, GraphQL, and Event-Driven Innovation

Introduction

In the rapidly evolving world of software development, new API standards like REST, GraphQL, and event-driven architectures are crucial for building scalable, efficient systems. Java for modern APIs, known for its stability in enterprise applications, has continually adapted to meet the demands of these modern paradigms. With an evolving ecosystem, Java for modern APIs remains a robust choice for developers creating flexible, scalable web services.

This article explores how Java for modern APIs has evolved to support REST, GraphQL, and event-driven architectures, highlighting the technical advancements, key frameworks, and best practices that power modern API development.

The Popularity of REST in Modern Web Services

Developers widely adopt REST (Representational State Transfer) as one of the most popular architectural styles for building web APIs. This is because its simplicity, statelessness, and ability to scale make it a natural fit for modern web services. Furthermore, Java has several frameworks that enhance and simplify the development of RESTful APIs, therefore making it easier for developers to build scalable and maintainable systems.

Key Frameworks for REST in Java

Below are some of the most widely used frameworks in the Java ecosystem:

- **Spring Boot**
- **JAX-RS**
- **Micronaut**

Spring Boot: Simplifying RESTful API Development

Spring Boot has become the most popular framework for building RESTful APIs in Java. Moreover, its ability to reduce boilerplate configuration through embedded servers like Tomcat and Jetty enables developers to get applications up and running quickly. In addition, with auto-configuration, production-ready defaults, and a comprehensive ecosystem of tools, Spring Boot simplifies the development of robust APIs.

Author:

Mihaela Gheorghe-Roman is a Software Architect at Systematic, Romania. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge at work, at the university, at different national and international IT conferences.



In her professional career, spanning over 13+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular.

<https://www.linkedin.com/in/mihaela-gheorghe-roman-6304915a/>

Spring Boot makes extensive use of annotations to define RESTful endpoints. For example, the `@RestController` annotation marks a class as a RESTful web service, whereas `@RequestMapping` maps incoming HTTP requests to methods, thereby ensuring seamless communication between the client and the server.

Here's a basic example of a Spring Boot RESTful service:

```
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/greeting")
    public String greeting() {
        return „Hello, World!";
    }
}
```

Spring Boot also offers comprehensive support for error handling through annotations like `@ExceptionHandler`, which simplifies the creation of consistent and informative error messages for clients. Furthermore, Spring Boot's monitoring and management features enable seamless integration of production-ready APIs into cloud environments, ensuring smooth deployment and operation.

JAX-RS: Jakarta EE's Approach to RESTful APIs

JAX-RS, a widely used specification for building RESTful APIs, is part of the Jakarta EE platform (formerly Java EE). JAX-RS simplifies the mapping of HTTP methods (GET, POST, etc.) to Java methods using annotations like `@GET`, `@POST`, and `@Path`. A basic example of a JAX-RS resource would look like this:

```
@Path("/api")
public class MyResource {

    @GET
    @Path("/greeting")
    @Produces(MediaType.TEXT_PLAIN)
    public String greeting() {
        return „Hello, World!";
    }
}
```

With implementations like Jersey and RESTEasy, JAX-RS provides flexibility in choosing the runtime environment. It integrates seamlessly with other Jakarta EE technologies, making it ideal for organizations already invested in the Jakarta EE ecosystem.

Micronaut: A Lightweight Framework for RESTful APIs

Micronaut is a relatively new framework specifically designed for building microservices and cloud-native applications. Notably, one of its key features is its focus on minimizing memory consumption and reducing startup time. This makes Micronaut particularly well-suited for serverless environments and applications that require quick startup times.

Unlike Spring, which relies on reflection for dependency injection, Micronaut, on the other hand, uses compile-time dependency injection. This results in faster startup times and lower memory usage, which is critical for applications running in resource-constrained environments like serverless functions.

Additionally, Micronaut also supports the creation of RESTful APIs through simple annotations like `@Controller`, `@Get`, and `@Post`.

A basic Micronaut RESTful service might look like this:

```
@Controller("/api")
public class MyController {
    @Get("/greeting")
    public String greeting() {
        return „Hello, World!";
    }
}
```

Its ability to integrate with serverless platforms such as AWS Lambda and Google Cloud Functions makes Micronaut an excellent choice for developers building cloud-native APIs that need quick startup times.

Java for Modern APIs: Best Practices for REST APIs

When building RESTful APIs, it's essential to follow best practices to ensure that your APIs are scalable, maintainable, and easy to use.

Here are some of the best practices for developing REST APIs in Java:

- **Statelessness**

RESTful services should be stateless, meaning that the server does not store any client-specific information between requests. Token-based authentication, such as JWT, is often used to manage sessions. Statelessness enables easy horizontal scaling, distributing the load across multiple servers without the complexity of managing session state.

- **Error Handling**

Well-structured error handling is critical for a smooth user experience. Java frameworks like Spring Boot and JAX-RS allow developers to manage errors globally using annotations like `@ExceptionHandler` and `@Provider`. Providing meaningful HTTP status codes (such as 400 for bad requests or 404 for not found) helps API consumers quickly diagnose issues.

- **API Versioning**

As APIs evolve, breaking changes may be introduced. To maintain backward compatibility, it's essential to version your APIs. The most common approach is URI-based versioning (e.g., `/api/v1/`), though header-based versioning is also supported by most Java frameworks.

- **OpenAPI/Swagger**

API documentation is an essential part of the development process, ensuring that API consumers understand how to interact with your services. Tools like `springdoc-openapi` allow developers to generate OpenAPI documentation automatically, providing interactive documentation that can be shared with teams and external partners.

Here's how to integrate `springdoc-openapi` into a Spring Boot project:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.5.9</version>
</dependency>
```

Java for Modern APIs: GraphQL for Flexibility and Efficiency

GraphQL, unlike REST, allows clients to specify exactly what data they need, preventing the over-fetching and under-fetching issues commonly found with REST. This flexibility has led to GraphQL's rising popularity in modern API design, and Java has several libraries and frameworks to support its implementation.

GraphQL Libraries and Frameworks in Java

Java offers several powerful libraries and frameworks to help developers implement GraphQL efficiently.

The following are some of the most widely used options:

- **GraphQL Java**

This library provides core GraphQL functionality, including schema definition, query execution, and mutation support. Furthermore, it integrates easily with both Spring and non-Spring applications, making it versatile for various use cases.

- **Netflix DGS Framework**

Netflix's DGS (Domain Graph Service) Framework, built on top of GraphQL Java, simplifies GraphQL development in Spring Boot applications. It supports schema-first development, query batching, and pagination, making it an ideal choice for building complex GraphQL APIs.

- **Spring GraphQL**

Spring GraphQL, introduced by Spring, provides tight integration between Spring Boot and GraphQL. It takes advantage of Spring's dependency injection, making it easier to build and secure GraphQL services.

Best Practices for GraphQL in Java

When working with GraphQL in Java, following best practices, in turn, ensures optimal performance, security, and maintainability.

As follows, here are key considerations to keep in mind when implementing GraphQL APIs in Java:

- **Schema-First Approach**

GraphQL revolves around well-defined schemas. Developers should focus on designing schemas that are clear, maintainable, and predictable for clients.

- **Efficient Data Fetching**

To avoid the N+1 problem, where multiple queries result in redundant data fetching, it is crucial to use tools like DataLoader. In addition, DataLoader batches and caches queries, which consequently improves performance for complex queries.

- **Query Security**

GraphQL's flexibility allows clients to execute expensive or malicious queries if not properly secured. Implement safeguards such as query depth limiting and complexity analysis to prevent abuse.

- **Pagination**

GraphQL allows clients to request large datasets, making pagination vital for API efficiency. Libraries like Netflix DGS offer built-in support for pagination and filtering, ensuring the server isn't overwhelmed by large data requests.

Event-Driven Architecture: Java's Path to Asynchronous Innovation

Event-driven architecture (EDA) is a powerful paradigm for building scalable, decoupled systems that react to real-time changes. Java provides excellent support for EDA, enabling developers to build reactive applications that respond to events asynchronously.

Tools and Frameworks for Event-Driven Systems in Java

Java provides a range of tools and frameworks for building event-driven systems that handle real-time data processing and asynchronous events.

Below are some of the most popular choices

- **Apache Kafka**

Kafka is one of the most popular platforms for building event-driven systems. Developers use it for real-time event streaming and processing, and Java's Kafka Streams API simplifies analyzing and processing event streams in real-time, offering features like fault tolerance, scalability, and exactly-once semantics.

- **Spring Cloud Stream**

Spring Cloud Stream effectively abstracts messaging systems like Kafka and RabbitMQ, allowing developers to easily build event-driven microservices. Furthermore, it simplifies the interaction between microservices and the messaging infrastructure by using annotations like `@StreamListener`.

- **Vert.x**

Vert.x is a lightweight, event-driven framework specifically designed for building reactive applications in Java. In addition, it provides an event bus for communication between services and supports high concurrency, making it an excellent choice for systems that need to handle large volumes of asynchronous events.

- **Akka**

Akka is a toolkit for building distributed, event-driven applications. By using an actor model to handle concurrency and message passing, it becomes highly suitable for large-scale, distributed systems that rely on real-time data processing.

Best Practices for Event-Driven Systems in Java

Adopting best practices is essential for building efficient and scalable event-driven systems in Java.

The following key practices are important for ensuring robust event handling and optimal system performance.

- **Idempotent Event Processing**

In event-driven systems, events may be processed multiple times due to retries or failures, which is why ensuring idempotent processing is critical to avoid unintended side effects. Ensuring idempotency - where processing the same event multiple times results in the same outcome - avoids unintended side effects.

- **Event Sourcing and CQRS**

Event sourcing involves storing all changes to an application's state as events, which makes the system auditable. Additionally, CQRS (Command Query Responsibility Segregation) separates the read and write sides of an application. Java's Axon Framework greatly simplifies the implementation of event sourcing and CQRS, therefore helping developers build scalable, maintainable systems.

- **Non-Blocking Communication**

Reactive programming is key to event-driven systems. For instance, Java's Project Reactor and RxJava offer tools for building non-blocking, reactive applications, thereby improving resource efficiency and scalability in systems with high concurrency.

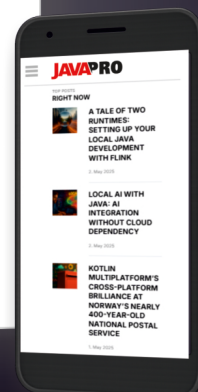
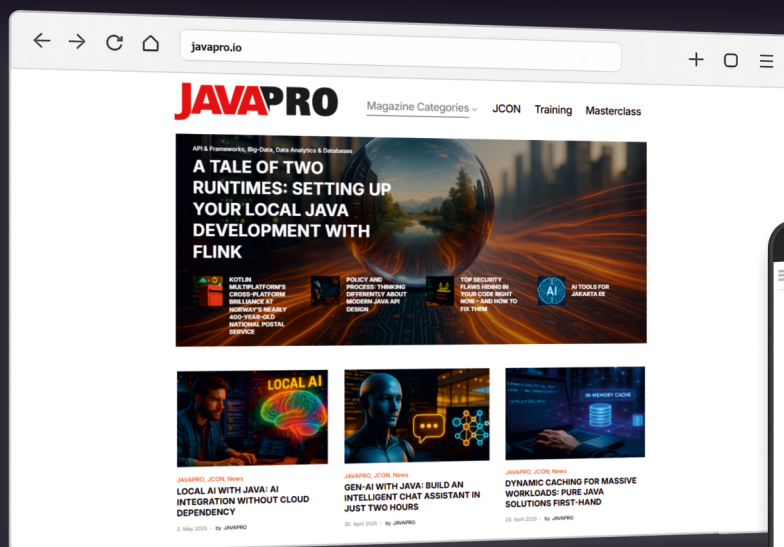
Conclusion

Java's flexibility and ability to evolve with modern API paradigms make it a leading platform for building scalable, efficient systems. Moreover, with robust frameworks for RESTful APIs, powerful libraries for GraphQL, and advanced tools for event-driven architectures, Java continues to be a reliable and versatile choice for developers building the APIs that drive today's applications.

By leveraging these technologies and best practices, developers can effectively create flexible systems that meet the growing demands of modern applications, all while harnessing Java's rich ecosystem of tools and frameworks.

DISCOVER MORE GREAT ARTICLES ONLINE:

JAVAPRO



The Long History of Log4j

Logging is the art of making sense of a system.

Software writes lines of information into log files, and developers sift through them, hoping to find what they need. The problem? Systems rarely log the right things. Yet without those files, we'd be flying blind, unaware that anything even went wrong.

This is the story of Apache Log4j—one of the oldest pieces of Java code still in use today. A tool that gave developers a way to listen to their systems.

Log4j is a European project

It all started in the mid-nineties when the European Union (EU) was in its early days. The SEMPER project was an early EU initiative to build a „secure toolkit for electronic commerce.“ It started in September 1995, ended three years later, and was funded with around €10,000,000. At that time, it was a huge investment—not just because of the amount but also because many doubted the internet itself would survive, let alone e-commerce. Prototypes were built and tested, and in the end, SEMPER became history. People often complain today that the EU is inflexible. But in 1995, SEMPER was proof that the EU could innovate.

In 1996, Ceki Gülcü, N. Asokan, and Michael Steiner introduced the idea of „hierarchical categories“. This was a groundbreaking idea at the time. Like all good programming jokes, logging was strictly binary: „on“ or „off.“ There was no way to log just database-related events or track user activity. Hierarchies changed everything. Developers could now enable logging at the package level, with child packages automatically inheriting their parents' settings. The SEMPER project funded the first version of Log4j, recognizing the importance of logging for system observability.

Transition to Open Source

Once SEMPER ended, Mr Gülcü brought the package to the IBM Zürich Research Laboratory, where he continued development. By 1998 or 1999, Log4j had found a new home at IBM Zürich. The project could have faded, but IBM Zürich kept it alive. Early records show that Log4j was first distributed through IBM's alphaWorks initiative. IBM used alphaWorks to foster emerging technologies, and Log4j aligned perfectly with that vision.

Wait, wait, emerging technologies?

Let's not forget Java launched as a beta version only a few years

Author:

Christian Grobmeier is an experienced Java developer, open-source advocate, and VP of Data Privacy at the Apache Software Foundation. He has contributed to key Apache projects, including Log4j, Commons, and Struts, and is actively involved in discussions around Java logging standards. As CEO of Grobmeier Solutions GmbH, he helps businesses build reliable, privacy-conscious software. Passionate about sustainable open-source practices and software security, he shares his insights at conferences, in publications, and through his writing on software ethics and leadership.



<https://www.linkedin.com/in/grobmeier/>
<https://bsky.app/profile/grobmeier.de>
<https://mastodon.social/@grobmeier>

earlier (in 1995) and hit 1.0 just one year later. Log4j emerged alongside J2SE 1.2 (1998) and 1.3 (2000). Java 1.2 brought us the Collections framework, and 1.3 followed with JNDI. In the era of Lambdas, working with Java 1.2 would send most of us running to woodworking. These upgrades made Java development massively easier.

As the millennium turned, many watched the skies in fear, wondering if Y2K would send planes plummeting. Meanwhile, Log4j's future was taking shape. Mr Gülcü moved Log4j to SourceForge, taking advantage of its public infrastructure, and registered log4j.org. Don't go there now—it leads to some weird places today.

Community over code

Open Source means something different today than it did in the past. Today, Open Source is often seen as a business model. We worry about funding, abandoned projects, and companies reclaiming their once-open code. Open Source takes many forms today, but back then, it was simple.

Money wasn't the driving force. Open Source was like punk—part political statement, part rebellion, a movement for software freedom.

When I was much younger, I met the CEO of a company. He told me how much he hated free software. In his opinion, he'd lose a lot of money because people would distribute their work for free. I was perplexed. How would a small company build Linux? How would they provide support for database servers and web servers? They'd charge, he told me. They would also charge African hospitals, small businesses, and charities. Without open source, only big corporations would exist. Open Source makes the world a better, fairer place.

Even today, the fight for software freedom continues, though in different forms. But freedom also comes with challenges—challenges that shaped where Log4j would end up.

The Free Software Foundation (FSF) was one of the strongest voices in early Open Source. Many Open Source projects found a home at the FSF, and Log4j could have been one of them. But there were a few reasons why this never happened: The biggest obstacle was licensing: FSF's GPL worked well for C and C++ but clashed with Java. Because Java loads libraries dynamically, it conflicted with the GPL's strict linking rules.

Another likely reason was an early commit of an Ant build file to Log4j. At some point, Log4j's maintainer reached out to the Apache Software Foundation (ASF), a rising community embracing Java projects. ASF's permissive license encouraged broad adoption.

So why wouldn't Log4j join ASF, where Ant was already at home?

Log4j joined the ASF and became part of the Apache Jakarta project. Yes, Apache Jakarta—the same name now known from the Eclipse Foundation, which develops Java Enterprise standards. By 2000, Apache Jakarta had become an umbrella project, hosting five major Java-related projects, including Apache Ant and Apache Tomcat.

Meanwhile, in 2001, Apache Commons Logging was proposed. Its goal was to provide a simple API compatible with both JDK logging and Log4j. It also introduced a pluggable API, allowing other logging frameworks to integrate seamlessly. Commons Logging aimed to unify logging under a single interface, positioning itself as the standard.

Log4j's time in Jakarta was brief. In 2003, it was promoted to a top-level project under the name „Logging Services.“ By then, Log4j had built a small community of six contributors. Growing from a solo project to a true community is rarely straightforward. As Log4j evolved, the ASF board tasked the project with integrating its clones, such as Log4net and Log4php.

The future looked bright—at least, for a while.

However, in 2005, technical disputes and personal conflicts

brought development to a halt. Momentum faded, and the community struggled to hold itself together. Eventually, Log4j's founder walked away and created Logback and Slf4j in 2006.

Code without community

You'd think the split would bring peace and stability to Log4j. But long disputes don't heal fast, and Log4j had no clear vision. The community was exhausted, and progress ground to a halt. Releases slowed to a crawl, taking years.

Version 1.2.15 came out in 2007. The next bugfix, 1.2.16, wouldn't arrive for another three years. Even sister projects like Log4php wound down and eventually stopped.

It was a dark time, with no sign of change. Log4j faded into obscurity, at risk of becoming dormant.

Comeback

By 2009, I stepped into the project at a crucial moment. For work, I needed Log4php—but it was barely functional. So, I started patching it—the first real work on the project in years.

Soon after, Ralph Goers jumped in to revive Log4j 2. By chance, our efforts aligned—sparking new life into the project. The community came back to life. Enthusiasm grew, and new contributors joined.

New contributors brought a fresh, unbiased perspective to the aging Log4j 1 codebase. While Logback positioned itself as a successor, Log4j 2 was built to learn from past mistakes. Log4j 2 was a complete rewrite, built on lessons learned from Log4j 1. Apache Logging Services might not have survived without breaking free from the old codebase.

Developers worldwide contributed to Log4j 2, shaping it into a community-driven project. Its open, community-driven approach encouraged developers to contribute patches and feedback.

Over time, it became clear that Log4j 2 was the future of Java logging and the days of Log4j 1 were numbered.

In 2015, Log4j 1 officially reached its end of life. Development ceased as the focus shifted to Log4j 2. The Log4j 2 team was confident that it was production-ready, and it delivered.

Positive feedback fueled adoption, drawing in more developers. Significant contributions included asynchronous logging and the LMAX disruptor, making Log4j 2 insanely fast.

Things looked bright again.

But not everyone welcomed the new logging landscape. Developers now had JUL, Slf4j, Logback, and Commons Logging to choose from. On top of that, both Log4j 1 and its successor, Log4j 2, coexisted.

Developers had to choose a framework and configure bridges to

connect different APIs. Log4j 2 wasn't just one product—it was two, packaged together. Like Slf4j and Logback, it combined an API and an implementation. It just never got the same attention.

Dark days: Log4shell

In July 2013, about a year before Log4j 2's first stable release, a developer contributed feature 313—enabling JNDI support. This feature integrated JNDI into Log4j 2. It was accepted without hesitation. Since JNDI is a core Java feature, adding it seemed like a logical step.

Log4j 2 was still in beta, so few questioned the change—or even noticed it.

Of course, we were wrong.

Log4j 2 shipped with feature 313—one of the most severe security flaws ever discovered.

The warning signs were there—we might have caught them at Black Hat 2016, a major security conference. But we didn't.

I hadn't even heard of Black Hat until Log4Shell went public. That year, security researchers at Black Hat detailed exactly how JNDI could be exploited.

And then came November 2021.

We communicate through e-mail lists, and every project has its own private channel. These lists are usually low-traffic, reserved for occasional personal matters or undisclosed security discussions. But in November 2021, the mailing list traffic suddenly spiked—something that almost never happened. It didn't take long to spot the words: „remote code execution.“ My first instinct? Close the e-mail client. I told myself I couldn't deal with this right now.

An hour later, I knew I had no choice—even though I wasn't actively developing. Fortunately, it wasn't my part of the code—others took the lead.

Say what you will about the issue—Log4j's team acted fast and handled it responsibly.

The first report came in on November 24, 2021.

A developer in China discovered the vulnerability. Within a day, we acknowledged the issue and realized it affected major projects like Apache Struts, Druid, and Flink—and likely countless others. After a few days of discussion, we fixed it by December 5. As usual, we initiated a three-day release vote. Despite the severity of the issue, the team remained confident—and relieved when everything worked out.

But just as the release vote wrapped up, two things happened. First, security groups started discussing the Log4j issue openly—on social media and beyond.

Then came the real shock: an email from the original reporter.

The fix didn't work.

That's when everything changed. Suddenly, I was in the middle of it—explaining the issue and fielding calls from companies. Panic set in, and we rushed to push out a new release. On December 9, we built a new release—cutting the usual three-day wait period.

Log4j 2.15 was released.

But the nightmare wasn't over.

More flaws emerged, and we patched them—one by one. Amid the chaos, some of us received e-mails filled with anger and blame. On forums, frustration exploded. Looking back, a handful of dedicated people worked tirelessly to fix this. None of us were paid for this work. Some even took days off, sacrificing their own vacation time. This was our issue, no doubt. But we didn't abandon the sinking ship—we fought to keep it afloat.

I often look back on this time and what it taught me. If there's one thing I learned, it's that this project was never just one or two people—it was a team.

With a strong team and dedicated people, we pulled through.

Aftermath

After the fix, Open Source was under a new kind of scrutiny. Governments and major newspapers picked up the story, fueling a deeper conversation about Open Source sustainability. For decades, developers worked for free, contributing their work for the public good—while organizations profited from it.

And then, the foundation of organizational code blew up. And once again, the same few people stepped up to fix it. Unpaid.

Twenty years ago, programming and software played a very different role in our lives. Back then, smartphones didn't even exist. But today, we use them to make payments. Your car is now more computer than machine. Almost every device in your home is now a tiny computer—even your water boiler connects to Wi-Fi.

Thankfully, things are starting to change. It's far from perfect, but organizations like the Sovereign Tech Agency (STA) are stepping in to support Open Source sustainability. In 2023 and 2024, the Log4j team received funding—and we gladly accepted. Well, part of the team, at least. I quickly learned that funding developers isn't as simple as it sounds. Some couldn't accept due to job contract restrictions. Others faced tax complications. And some didn't have time for a major project with limited funding. Three people worked full-time on Log4j for 1.5 years. We added security features such as SBOM, provided cleanups, and improved documentation and the website. We also made it possible to use Log4j with Graal VM and allowed it to run on Android. We learned a lot, and Log4j grew despite the crisis it faced.

Takeaways

What if there were only two of us?
What if we were on vacation at that time?
What if we didn't care?

Log4j could have been abandoned many times, but it survived. Sometimes, it was luck, and sometimes, it was the right people at the right time.

What did I learn?

Software engineers must take security more seriously. Security will be one of the biggest challenges in software's future. AI will help attackers discover vulnerabilities faster, and we must stay vigilant. As threats evolve, we must stay one step ahead.

Think about this: Log4Shell was introduced in 2013—and remained undetected until 2021. And we still don't know if it was exploited before then.

Community is the backbone of software development. High-impact projects can't be maintained alone—we need a reliable team.

Open Source has a sustainability problem. Tech giants profit from Open Source, yet without it, innovation would stall.

Open Source is everywhere. Let's make it sustainable.

Future

So where does that leave Log4j today?

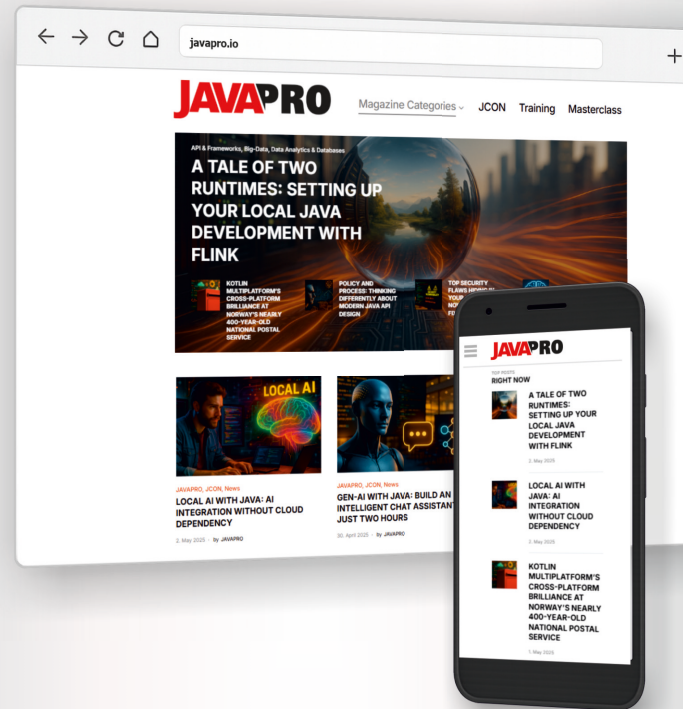
Because of Log4Shell, no other logging framework has been tested as thoroughly. The team has proven its responsibility and commitment to security. Yes, it is „safe.“ Nobody can predict the next Log4Shell, but we've strengthened our security and built a cleaner, more robust codebase.

Collaboration is everything. Some of us hope to one day work under a shared logging API, hosted within the Eclipse Jakarta project.

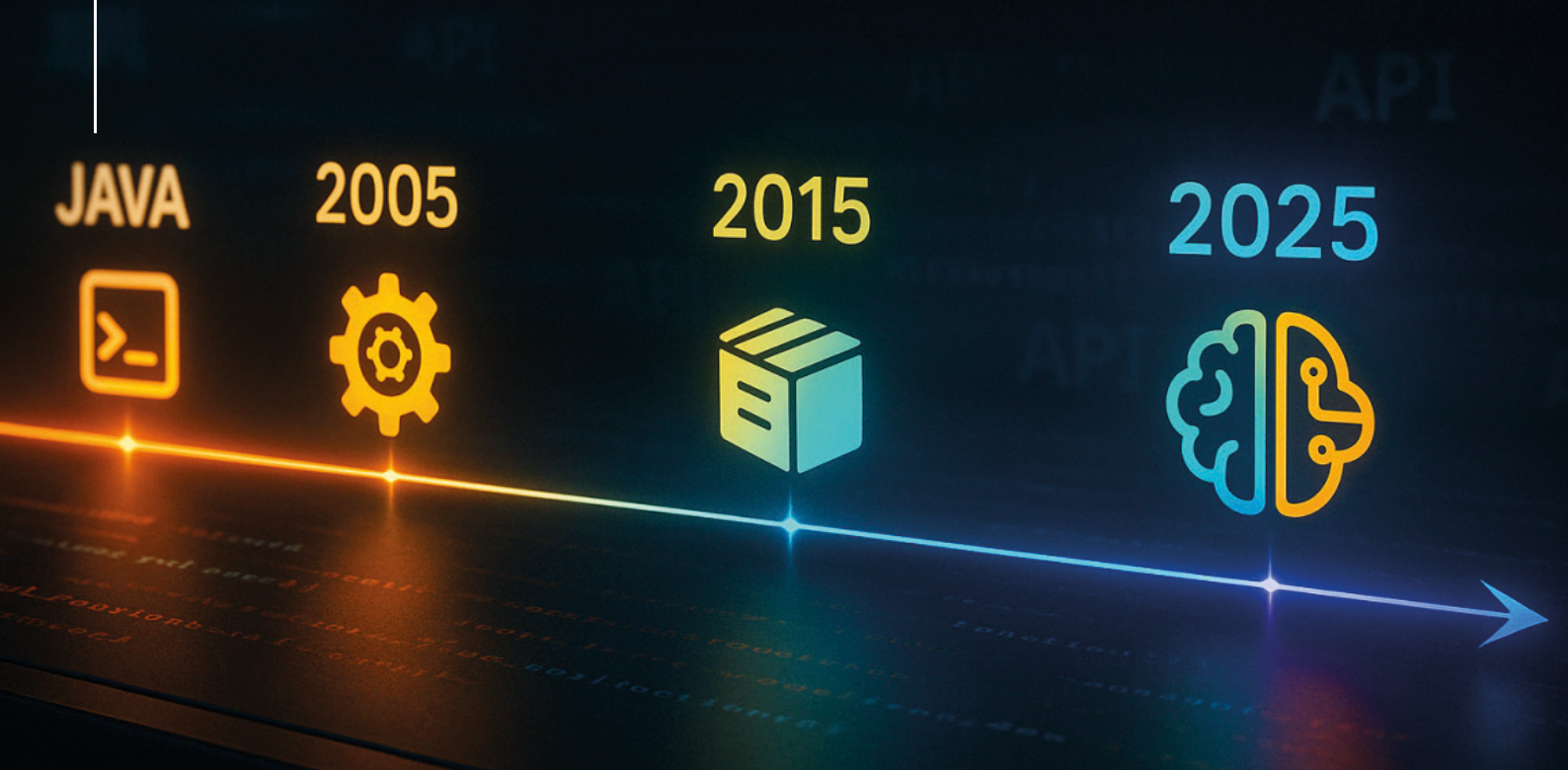
Until that day comes, we look forward to another 20 years of logging. And who knows? Maybe in twenty years, I'll write another article like this one. History is still written every day. You can be part of it. Leave your mark—we are just one commit away.

JAVAPRO

**DISCOVER MORE GREAT
ARTICLES ONLINE:**



**BECOME A JAVAPRO
AUTHOR & BLOGGER !**



#JAVAPRO #FRAMEWORK #API

A Journey through 30 Years of API Evolution

Introduction

Java, since its inception in 1995, has become one of the most influential programming languages. Its promise of “Write Once, Run Anywhere” (WORA) transformed the landscape by offering a platform-independent, object-oriented programming environment. Over 30 years, the evolution of Java APIs showcases constant innovation. From portability to cloud-native apps, machine learning, and distributed systems. This blog takes you how Java has evolved and shaped its API ecosystem as we know it today.

Packages

A Java package is a namespace or container that organizes related classes, interfaces, and sub-packages into a modular structure. Each package serves a specific purpose and improve code management. Java supports both built-in and user-defined packages. This discussion focuses on the evolution of built-in packages and their adaptation to industry requirements.

The `java.lang` package contains the fundamental classes. `java.util` provides utility classes. `java.io` is defined for input and output operations, while `java.nio` introduces asynchronous capabilities for I/O. `java.awt` enables graphical user interfaces (GUI), and `java`.

Author:

Vipin Menon is a software professional with over 13 years of experience in the industry, specialising in a diverse range of technologies, including Java, React, Node.js, JavaScript, and various web and cloud technologies. Currently, he is contributing to the design and development of Application Performance Monitoring for Instana at IBM.

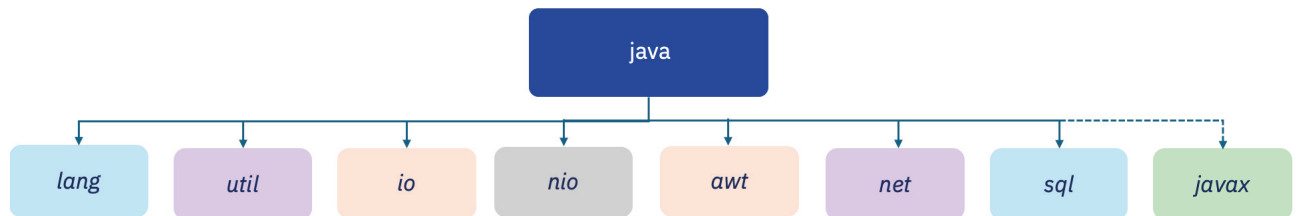


A passionate advocate for open-source software, he has made significant contributions to projects such as OpenJDK, Node.js, and Swift. He played a pivotal role in implementing key security algorithms in IBM Java 11 as part of the integration of TLS 1.3 within the security module. With a strong focus on innovation, he holds six patents, one publication, and shares his insights through numerous blogs on technology topics, featured on platforms like Medium and other blogging platforms.

<https://x.com/vipinaiswarya>

net supports network applications. java.sql facilitates database communication, and javax is introduced as an extension to the JDK focused on extending core functionalities, particularly for enterprise applications.

- **javax:** Classes like swing, xml that extended the functionality of the core java package.



The Foundation (Java 1.0 - 1.4)

When Java first appeared in 1995, it aimed to provide a platform-independent language. This allowed developers to build applications capable of running anywhere once built. Java 1.0 was simple, robust, and lightweight. It offered minimal yet essential APIs that formed the building blocks of the Java ecosystem.

- **java.lang:** classes like String, Math, Exceptions and Threads. String helped in manipulating the text. Math helped with mathematical calculations, Exceptions were one of the key feature providing a structured mechanism to handle unexpected errors during program executions. While threads provide a way for a Java program to perform multiple tasks at the same time
- **java.util:** Classes like List, Set, and Map as part of the Java Collections Framework. List for indexed data with duplicates, Set for unique elements and Map for <key, value> pairs ensuring each key is unique. This package also contained Logging classes which could be used in Java applications for logging messages such as errors, warnings, and informational messages.
- **java.io:** Classes mainly for file operations like InputStream, OutputStream, Reader and Writer developers to read and write data.
- **java.nio:** Classes for non-blocking I/O operations, including buffers, channels, and selectors.
- **java.awt:** Classes for creating windows, buttons, menus, text fields, labels and other common GUI interfaces. Later, Swing largely replaced it.
- **java.net:** Classes for network communication such as Socket and URL, enabling developers to create web-based applications.
- **java.sql:** Classes like Connection, statement, ResultSet and for jdbc interface for connecting and executing queries with relational databases, allowing Java applications to interact with databases in a platform-independent way.

The evolution of the APIs clearly followed industry demands and remained up to date. It started with basic language functionality and I/O support. It then expanded into a comprehensive set of libraries and tools. These offered powerful capabilities for GUI development, distributed systems, networking, and database access. These versions primarily focused on providing developers with the necessary tools to build cross-platform applications. This led to the introduction of JDBC (Java Database Connectivity). It served as a powerful interface to allow Java applications to interact with databases in enterprise applications.

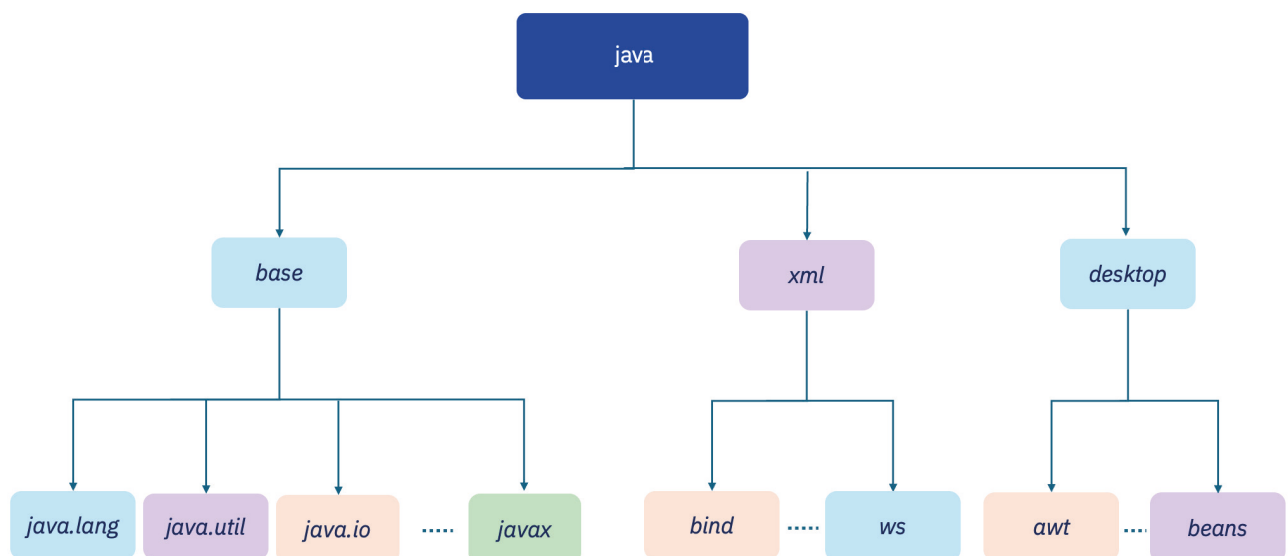
JAVA In Enterprise world(Java 1.5 - 1.8)

As the internet evolved, the need for scalable and robust enterprise applications became evident. This led Java to position itself as the go-to language for large-scale applications. Java 5 through Java 8 introduced a series of critical API, marking the golden era of Java.

- **java.lang:** features like Enum, StringBuilder, ThreadLocal, Lambda , Annotations. Enum allows collection of constants. StringBuilder handles mutable strings efficiently. ThreadLocal helps to store data that is specific to a particular thread. Lastly, Lambda introduced functional programming in Java. Annotations helped to associate metadata to program elements which later became a stepping stone for frameworks like Spring and Hibernate.
- **java.util:** Classes like ConcurrentHashMap, Streams, Generics, Optional. ConcurrentHashMap introduced concurrent read and write operations without locking the entire Map. Streams helps with a functional approach to process collections of objects. Generics introduced parameterised types to enhance type safety. Optional is used to represent a value that may or may not be present, helping to prevent NullPointerExceptions.
- **java.io:** Classes Console for reading from and writing to the console. BufferedReader and BufferedWriter enhancements with lines() method to work with file lines using functional programming there by allowing better integration with

streams API.

- **java.nio:** Introduced File class. Unlike io, it used platform independent path with support for nonblocking and memory mapped I/O.
- **java.awt:** Improved EventHandling due to Lambda expressions. Enhanced java webstart integration and javafx integration.
- **java.net:** try-with-resources, SocketChannel, HTTP Client API. try-with-resources manage the lifecycle of network resources automatically. SocketChannel allowed for non-blocking I/O operations. HTTP Client API was introduced as a preview for handling of HTTP requests and responses.
- **java.sql:** Introduced JDBC4.0, java.time types and Stream API support for ResultSet. 4.0 brought in automatic driver loading, connection pooling and better exception handling. time type helped to use LocalDate, LocalDateTime, ZonedDateTime directly within JDBC. Stream helped to enhance resultset with functional processing.
- **javax:** Addition of JAXB, JAX-WS, javax.tools, javax.script. JAXB helped easier conversion between Java objects and XML documents. JAX-WS introduced tooling for webservices. javax.tools applications to compile Java code
- **java.util:** Optional allows user-defined exception and can return an alternate value if empty. Stream.takeWhile() and Stream.dropWhile() can take or drop elements based on the predicate conditions. copyOf() can create immutable copies of collections.
- **java.io:** InputStream.transferTo() can efficiently transfer data from an InputStream to an OutputStream while properly managing the closure of the stream resources.
- **java.nio:** Files.writeString() and Files.readString() simplify reading from and writing to text files, and they automatically handle character encoding.
- **java.awt:** The package was moved under java.desktop package. A new rendering pipeline for AWT and Swing applications on macOS, utilizing Apple's Metal API instead of OpenGL. Improved java.awt.Font with better customisation.
- **java.net:** Enhanced HTTP/2 support, including the ability to manage multiple streams over a single connection. cached SecureRandom instances for faster communications over protocols like SSL/TLS.
- **java.sql:** Introduced java.sql module separate from core.



- **java.lang:** With StackWalker, we can now traverse specific stack frames of interest instead of working with the entire stack trace. The String class was enhanced with the strip() method to better handle whitespace. The var keyword allows for declaring variables without the need to explicitly specify their types. Additionally, NullPointerExceptions now provide more detailed information about which object was null. Introduction of Sealed classes and interfaces could restrict which classes can extend or implement them.
- **javax:** The most significant change in the javax package occurred after Java 11, with the transition of Java EE (Enterprise Edition) to the Jakarta EE project under the Eclipse Foundation. This transition led to the javax namespace being replaced with the jakarta namespace. This marked the deprecation of javax from JDK.

Launch Single-File Source-Code Programs(JEP 330) can help run database queries and interact with JDBC in a simple program. It is especially useful for quick tests or scripts.

Scalable Solutions (java 18-23)

Java addressed scalability and concurrency in these versions with the introduction of virtual threads, as part of Project Loom. Project Panama improves Java's ability to interface with native libraries. It enhances performance and reduces boilerplate code in native interactions. Java's core libraries were robust, but improvements in other areas of the language and JVM enhanced the overall ecosystem. These changes made it more efficient and better able to leverage the existing API.

- **java.lang, java.util:** RecordPatterns(JEP 405) introduced a pattern-matching feature that allows a more concise way to match and de structure records. Sequential Collections(JEP 431) Introduced new interfaces to represent collections with a defined encounter order. Virtual threads (JEP-444) were introduced which could execute large numbers of concurrent tasks, such as web servers or network applications.
- **java.io, java.nio:** The Foreign Function and Memory API (JEP 454) helps Java programs to interoperate with code and data outside of the Java runtime. UTF-8 by default(JEP 400) helped Java programs more predictable and portable.
- **java.net:** Structured concurrency improves the efficiency and manageability of handling concurrent network operations particularly when dealing with multiple network requests or responses happening simultaneously. jwebserver as a command line utility to serve static files.

Future and beyond (java 24 -)

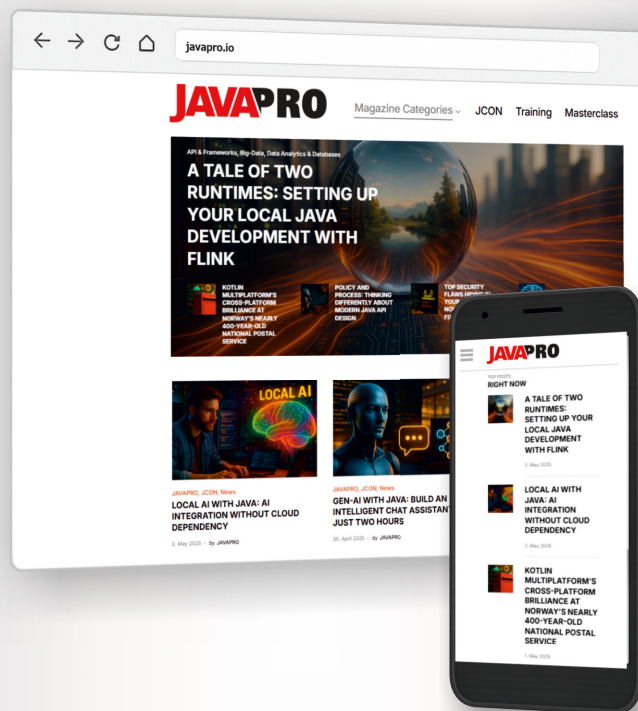
When Java 24 releases in March 2025, it will introduce several features. These will enhance cloud environments, improve startup performance, and increase resource management efficiency. Notable among these are Ahead of Time (AOT) class loading and linking (JEP 483) to boost startup times. Another feature is Compact Object Headers (JEP 450), which minimize the in-memory size of Java objects. With these improvements, Java is positioning itself as an ideal platform for cloud-native developers. It enables them to take full advantage of these new capabilities.

Conclusion

Over the past 30 years, the Java API ecosystem has evolved. It has catered to the diverse and ever-changing needs of the software development world. Java has shown an unparalleled ability to adapt to its surrounding demands. As we look ahead, this rich ecosystem of API will drive Java's evolution. This will ensure its place at the forefront of the programming world for years to come.

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

A Passwordless Future: Passkeys for Developers

Passwords have been around for thousands of years and we were all happily sharing our Netflix passwords. They are probably not going anywhere but they are not the best way to secure our digital lives. They are a hassle for users and a security risk for applications. The future of secure authentication is passwordless and passkeys are leading the way.

According to Verizon's 2024 Data Breach Investigations Report, stolen credentials and phishing account for over 65% of all data breaches.

Most of the password problems are human problems as they rely on us humans to remember them and to not share them and do a bunch of other stuff. The main problems with passwords are:

- **Knowledge-based:** People can be socially engineered to divulge passwords. If passwords are easy to remember they are also easy to guess. Sharing and reusing passwords makes them even more vulnerable.
- **Phishing & Remote Replay:** Phishing websites can easily harvest passwords from even the most tech-savvy. Accounts can be accessed remotely using harvested passwords.
- **Data Breach:** Applications become a target for data breaches when they store passwords.
- **Password management:** Passwords need password recovery and reset flows and multi-factor authentication flows to secure them further.

Did you know it could cost around 70\$ to reset a password?

Of course, password managers help with some aspects of this and everyone should use one. But they are still an overhead and not convenient for everyone, especially non-tech folks.

Passkeys

The obvious solution for the password problem is to go passwordless. This is where passkeys come into the picture.

"Passkeys are a password replacement that provides faster, easier, and more secure sign-ins to websites and apps across a user's devices. Unlike passwords, passkeys are resistant to phishing, are always strong, and are designed so that there are no shared secrets."
— FIDO Alliance

Author:

Deepu K Sasidharan is a Software Engineer by passion and profession. He is a Java Champion working as a Staff Developer Advocate at Okta. He is the co-chair of JHipster and the creator of KDash and JDL Studio. He is a polyglot programmer working with Java, Rust, JavaScript, Go, and so on. He is also a cloud technology advocate and an open-source software aficionado. He has authored books on Full-stack development and frequently writes about Java, Rust, JavaScript, Go, DevOps, Kubernetes, Linux, and so on, on his blog.



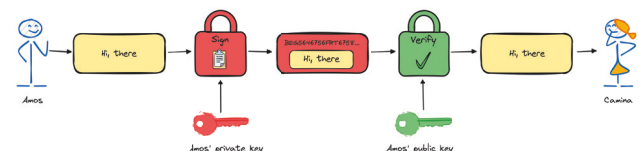
<https://www.linkedin.com/in/deepu05/>

<https://bsky.app/profile/deepu105.bsky.social>

<https://twitter.com/deepu105>

A passkey is a unique cryptographic key pair that allows you to access online services without using passwords. It is based on asymmetric public-key cryptography.

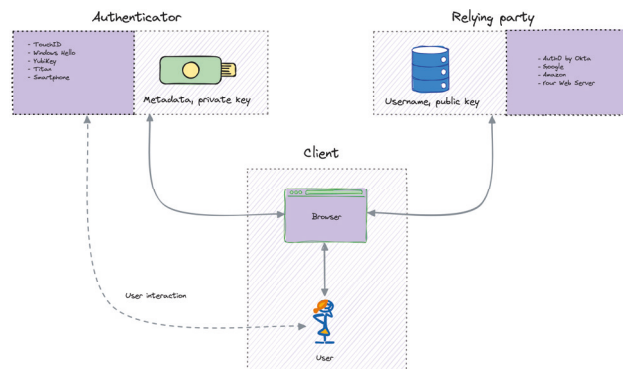
Before we dive deep into passkeys let's look at some of the underlying technologies that make passkeys possible.



Passkeys

An authenticator is a hardware or software entity that can create and store public-private key pairs. There are two types of authenticators:

- **Platform authenticators:** It is built into a user's device. For example, TouchID, smartphone authenticators, Windows Hello, and so on.
- **Roaming authenticators:** A removable authenticator



The WebAuthn architecture consists of three main entities:

- **Authenticator:** A device that lets a user authenticate by confirming their presence.
- **Relying Party:** A server or an Identity Provider (IdP) that requires authentication. It issues challenges and stores public keys.
- **Client:** A client consists of the user's browser. The client relays information between an authenticator and a relying party.

Client to Authenticator Protocol

The FIDO Client to Authenticator Protocol is used for communications with authenticators over a variety of transports like USB, NFC, and Bluetooth. It is used to send requests from WebAuthn to authenticators.

Types of passkeys

Passkeys have two variants. Synced passkeys and device-bound passkeys.

Synced passkeys

Synced passkeys have a better user experience since the private keys are end-to-end encrypted and synced to the cloud. For example, on the Apple ecosystem, the private key is synced on your iCloud Keychain and you can register on one device and log in to any synced Apple device. The same goes for the Google ecosystem using the Chrome browser and Google Passward Manager. Synced passkeys can be restored on new devices.

Device-bound passkeys

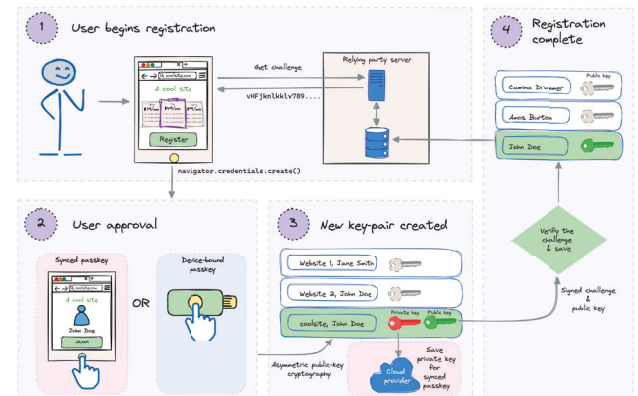
In the device-bound passkeys, the private key stays on the device itself and you need to authenticate using the same authenticator used for registration. It is slightly less convenient but more secure than synced passkeys. The relying party must support registering multiple credentials for a user as a backup.

How does it work?

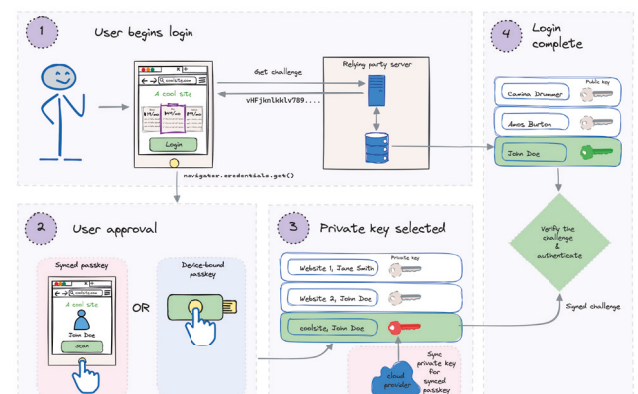
Let's see how user registration and authentication work with passkeys.

User registration

First, let's see how the registration flow works.



1. The user begins registration. The relying party provides a randomly generated challenge string.
2. The navigator.credentials.create() method of the WebAuthn API is invoked and the user provides approval using their authenticator.
3. The authenticator creates a private-public key pair which is unique for the relying party's domain and the user. The private key is used to sign the challenge.
 - The private key is stored on the authenticator.
 - For synced passkeys, the private key is also synced to a cloud service for backup and roaming.
 - An attestation object is created which contains the public key, signed challenge, credential ID, and certificate.
4. The attestation object and other metadata are passed to the relying party by the client-side implementation. The relying party verifies the signed challenge using the public key and registers the user by storing the public key and credential ID along with the user details.



1. The user begins the login flow. The relying party provides a randomly generated challenge string.
2. The `navigator.credentials.get()` method of the WebAuthn API is invoked and the user provides approval using their authenticator.
3. The authenticator retrieves the private keys for the relying party's domain name.
 - For synced passkeys, if the device is new, the private key is synced from a cloud service if available.
 - The user selects the private key for their username. The private key is used to sign the challenge.
 - An assertion object is created which contains the signed challenge and credential ID.
4. The assertion object and other metadata are passed to the relying party by the client-side implementation. The relying party verifies the signed challenge using the public key stored for the user and authenticates the user.

Why Passkeys?

Passkeys are superior to password + traditional OTP MFA in terms of security and usability and they are as secure and more convenient than password + FIDO MFA. Most importantly, you don't have to remember anything.

Passwords	Passkeys
🔒 Phishing	🔒 Phishing resistant
🔄 Remote replay	🔒 Remote attack resistant
📄 Data breach	🔒 Breach resistant
🔄 Reuse & share	🔒 Not Reusable & Shareable
🔑 Knowledge-based	🔒 Discoverable Credentials
🔑 Password management	🔒 Easier to maintain

- **Discoverable:** Passwords are knowledge-based but passkeys are discoverable credentials and the browser can autofill them for a service making it unnecessary for you to remember even usernames. It doesn't rely on something you know, instead, it relies on something you have or something you are which is more secure from hacking and social engineering. They cannot be reused as they are unique per service and user combination and cannot be shared.
- **Phishing & Remote attack resistant:** Passkeys cannot be phished as they rely on public key cryptography and are bound to the domain name of the website, making it impossible to work on a spoofed website. Passkeys rely on physical keys, like biometric sensors of platform authenticators or roaming authenticators like YubiKey, hence cannot be remotely breached.
- **Breach resistant:** The website only stores the public key of a user which is useless to an attacker on a data breach on the server side. This makes the server less attractive to hackers.
- **Easier management:** Synced passkeys are backed up

and replicated across your devices by services like iCloud Keychain and Google Password Manager. This makes recovery part of the platform rather than the application.

- **Cross-device authentication:** Passkeys can also perform cross-device authentication regardless of ecosystem or platform. For example, you can simply use your Android phone as an authenticator for your Apple laptop.

Challenges

There are still some challenges when it comes to passkeys:

- **OS/Browser support:** It is dependent on the OS and Browser to implement the specs, and we all know how that turns out right? This means the support may not be uniform and can become fragmented. Browser and OS compatibility is still catching up.
- **Cloud vendor reliance:** It relies on companies like Apple, Google, and Microsoft to save the private keys in their cloud securely and protect them from breaches.
- **Enterprise use cases:** Enterprise users might want more control and flexibility which could be a problem. For example, if an enterprise doesn't allow iCloud or Google Chrome on their computer, synced passkeys will not work there, only device-bound passkeys will work.
- **Reset & Recovery:** There are no default recovery flows for device-bound passkeys, and applications might still need to implement recovery & reset flows to accommodate all use cases.

WebAuthn and Passkeys for Java

Though Web Authentication's user experience is a client-side implementation using JavaScript, the backend or Relying party can be a Java server. Ideally using an IdP like Auth0 would be the best option since it takes care of all the heavy lifting for you. You can find the sample app using Auth0 on GitHub.

But if you want to implement it yourself and walk the harder path, you can use one of the below libraries.

- **WebAuthn4j:** A 100% FIDO2 conformant library with support for all attestation formats and validation. It is used by Keycloak and Spring Security.
- **Java-webauthn-server:** A library from Yubico that supports many attestation format. But it is not 100% FIDO2 conformant.

Conclusion

Passkeys are the future of secure authentication. They are more secure and more convenient than passwords and traditional MFA. They are also more secure and more convenient than other passwordless methods like magic links, SMS/Email OTP, and push notifications. The wider adoption of passkeys could finally solve the password problem and make the internet a safer place. Learn more on learnpasskeys.io.

JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



www.javapro.io

Best Practices for API Design in Java

Understanding The best practices of API Design in Java is crucial for Java developers aiming to create robust, scalable Microservices. In this article, we dive into 11 essential best practices for API development in Java, which will guide you toward professional and efficient API creation. By exploring Java-specific examples, you'll learn to distinguish between effective and inefficient approaches to API development.

1. Adherence to RESTful Principles

RESTful architecture is defined by its statelessness, cacheability, uniform interface, and client-server design. By adhering to these principles, APIs promote predictable and standardized interactions.

Good Example

A GET request to retrieve a resource by ID.

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@
    PathVariable Long id) {
        User user = userService.findById(id);
        return ResponseEntity.ok(user);
    }
}
```

Avoid Example

Performing an update action using a GET request goes against the principle that GET requests should be safe and idempotent.

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public ResponseEntity<Void>
    forceUpdateUserEmail(@PathVariable Long id,

    @RequestParam String email) {
        // This is bad, GET should not change
        state.
        userService.updateEmail(id, email);
        return ResponseEntity.ok().build();
    }
}
```

Author:



Muaath Bin Ali is a Principal Software Engineer with over 15 years of experience. As the leader of RiyadhJUG, he regularly hosts speakers and Java champions worldwide. Muaath shares his insights through blogs on MezoCode.com and fooja.io, assisting Java developers in building real-world software architectures and excelling in their careers.

<https://www.linkedin.com/in/muaath-bin-ali/>

<https://x.com/mezoCode>

2. Utilization of Meaningful HTTP Status Codes

Good Example

Using 201 Created for successful resource creation.

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody
    User user) {
    User savedUser = userService.save(user);
    return new ResponseEntity<>(savedUser,
    HttpStatus.CREATED);
}
```

Avoid Example

Returning 200 OK for a request that fails validation, where a client error code (4xx) would be more appropriate.

```
@PostMapping(„/users“)
public ResponseEntity<User> createUser(@
RequestBody User user) {

    if (!isValidUser(user)) {
        // This is bad, should be 4xx error.
        return new ResponseEntity<>(HttpStatus.
OK);
    }

    User savedUser = userService.save(user);
    return new ResponseEntity<>(savedUser,
HttpStatus.CREATED);
}
```

3. Strategic API Versioning

Good Example

Explicitly versioning the API in the URI.

```
@RestController
@RequestMapping(„/api/v1/users“)
public class UserController {
    // RESTful API actions for version 1
}
```

Avoid Example

Lack of versioning, leading to potential breaking changes for clients.

```
@RestController
@RequestMapping(„/users“)
public class UserController {
    // API actions with no versioning.
}
```

4. Graceful Exception Handling

Robust error handling enhances API usability by providing clear, actionable information when something goes wrong.

Good Example

A specific exception handler for a resource not found scenario, returning a 404 Not Found.

```
@ExceptionHandler(UserNotFoundException.class)
public ResponseEntity<Object>
handleUserNotFound(UserNotFoundException ex) {
    return new ResponseEntity<>(ex.getMessage(),
HttpStatus.NOT_FOUND);
}
```

Avoid Example

Exposing stack traces to the client, can be a security risk and unhelpful to the client.

```
@ExceptionHandler(Exception.class)
public ResponseEntity<Object>
handleAllExceptions(Exception ex) {

    // This is bad, don't expose stack trace.
    return new ResponseEntity<>(ex.getStackTrace(),
HttpStatus.INTERNAL_SERVER_ERROR);
}
```

5. Ensuring API Security

Security is paramount, and APIs should implement appropriate authentication, authorization, and data validation to protect sensitive data.

Good Example

Incorporating authentication checks within the API endpoint.

```
@GetMapping(„/users/{id}“)
public ResponseEntity<User> getUserById(@
PathVariable Long id) {

    if (!authService.isAuthenticated(id)) {
        return new ResponseEntity<>(HttpStatus.
UNAUTHORIZED);
    }
    // Fetch and return the user
}
```

Avoid Example

Omitting security checks, leaving the API vulnerable to unauthorized access.

```
@GetMapping(„/users/{id}“)
public ResponseEntity<User> getUserById(@
PathVariable Long id) {
    // No authentication check, this is bad.
    // Fetch and return the user
    User user = userService.findById(id);
    return ResponseEntity.ok(user);
}
```

6. Comprehensive API Documentation

Well-documented APIs facilitate ease of use and integration, reducing the learning curve for developers and encouraging adoption.

Good Example

Using annotation-based documentation tools like Swagger.

```

@Api(tags = „User Management“)
@RestController
@RequestMapping(„/api/v1/users“)
public class UserController {
    // RESTful API actions with Swagger annotations
    // for documentation
}

```

Avoid Example

Non-existent documentation makes it difficult to discover the usability of APIs.

```

@RestController
@RequestMapping(„/users“)
public class UserController {
    // API actions with no comments or documentation
    // annotations
}

```

7. Effective Use of Query Parameters

Query parameters should be employed for filtering, sorting, and pagination to enhance the API's flexibility and prevent the transfer of excessive data.

Good Example

API endpoints that accept query parameters for sorting and paginated responses.

```

@GetMapping(„/users“)
public ResponseEntity<List<User>> getUsers(
    @RequestParam Optional<String> sortBy,
    @RequestParam Optional<Integer> page,
    @RequestParam Optional<Integer> size) {

    // Logic for sorting and pagination
    return ResponseEntity.ok(userService.
        getSortedAndPaginatedUsers(sortBy, page, size));
}

```

Avoid Example

Endpoints that return all records without filtering or pagination potentially overwhelm the client.

```

@GetMapping(„/users“)
public ResponseEntity<List<User>> getAllUsers() {
    // This is bad, as it might return too much
    // data
    return ResponseEntity.ok(userService.findAll());
}

```

8. Leveraging HTTP Caching

Caching can significantly improve performance by reducing

server load and latency. It's an optimization that experts should not overlook.

Good Example

Implementing ETags and making use of conditional requests.

```

@GetMapping(„/users/{id}“)
public ResponseEntity<User> getUserById(
    @PathVariable Long id,
    @RequestHeader(value = „If-None-Match“, required =
        false) String ifNoneMatch) {
    User user = userService.findById(id);
    String etag = user.getVersionHash();

    if (etag.equals(ifNoneMatch)) {
        return ResponseEntity.status(HttpStatus.
            NOT_MODIFIED).build();
    }

    return ResponseEntity.ok().eTag(etag).body(user);
}

```

Avoid Example

Ignoring caching mechanisms leads to unnecessary data transfer and processing.

```

@GetMapping(„/users/{id}“)
public ResponseEntity<User> getUserById(
    @PathVariable Long id) {
    // No ETag or Last-Modified header used, this is
    // bad for performance
    return ResponseEntity.ok(userService.
        findById(id));
}

```

9. Maintaining Intuitive API Design

An API should be self-explanatory, with logical resource naming, predictable endpoint behavior, and consistent design patterns.

Good Example

Clear and concise endpoints that immediately convey their functionality.

```

@PostMapping(„/users“)
public ResponseEntity<User> createUser(@RequestBody
    User user) {
    // Endpoint clearly indicates creation of a user
}

@GetMapping(„/users/{id}“)
public ResponseEntity<User>
    getUserById(@PathVariable Long id) {
    // The action of retrieving a user by ID is clear
}

```

Avoid Example

Confusing or convoluted endpoint paths and actions that obfuscate their purpose.

```
@PutMapping(„/user-update“)
public ResponseEntity<User> updateUser(@RequestBody
User user) {
    // This is bad, as the path does not indicate a
    resource
}
```

10. Enable Response Compression

Enabling response compression is a smart move to optimize network performance. It reduces the payload size, which can significantly decrease network latency and speed up client-server interactions.

Good Example

Configuring your web server or application to use gzip or Brotli compression for API responses.

```
// In Spring Boot, you might configure application.
properties to enable response compression.

server.compression.enabled=true
server.compression.mime-types=application/
json,application/xml,text/html,text/xml,text/plain
```

This configuration snippet tells the server to compress responses for specified MIME types.

Avoid Example

Sending large payloads without compression leads to increased load times and bandwidth usage.

```
// No configuration or code in place to handle
response compression.

@GetMapping(„/users“)
public ResponseEntity<List<User>> getAllUsers()
{
    // This could return a large JSON payload that
    isn't compressed, which is inefficient.
    return ResponseEntity.ok(userService.
    findAll());
}
```

11. Embrace Asynchronous Operations

Asynchronous operations are essential for handling long-running tasks, such as processing large datasets or batch operations. They free up client resources and prevent timeouts for operations that take longer than the usual HTTP request-response cycle.

Good Example

Using asynchronous endpoints that return a 202 Accepted status code with a location header to poll for results.

```
@PostMapping(„/users/batch“)
public ResponseEntity<Void> batchCreateUsers(@
RequestBody List<User> users) {

    CompletableFuture<Void> batchOperation =
userService.createUsersAsync(users);
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation(URI.create(„/users/
batch/status“));

    return ResponseEntity.accepted().
headers(responseHeaders).build();
}
```

This example accepts a batch creation request and processes it asynchronously, providing a URI for the client to check the operation's status.

Avoid Example

Blocking operations for batch processing that keep the client waiting indefinitely.

```
@PostMapping(„/users/batch“)
public ResponseEntity<List<User>>
batchCreateUsers(@RequestBody List<User> users) {
    // This is a synchronous operation that may
    take a long time to complete.
    List<User> createdUsers = userService.
createUsers(users);
    return ResponseEntity.ok(createdUsers);
}
```

This example performs a synchronous batch creation, which could lead to a timeout or a poor user experience due to the long wait time.

By incorporating response compression and embracing asynchronous operations, API developers can greatly improve performance and user experience. These practices are essential when dealing with modern web applications and services that require efficient real-time data processing and transmission.

Conclusion

adhering to these practices is not just about following a checklist; it's about creating an API that stands the test of time and evolves without causing disruption. For developers with a wealth of experience, these guidelines serve as a reminder and a benchmark for excellence in API development.

References

- RESTful web API design
- GitLab API Design
- REST Architectural Constraints
- Standard Response Status codes



#JAVAPRO #FRAMEWORK #API

Under the Surface: An ,Iceberg‘ Journey through the Lesser-known Faces of Java

The „Iceberg“ meme is an internet phenomenon that humorously and sometimes unsettlingly, illustrates levels of knowledge or initiation into a given topic – from simple, widely known facts at the tip of the iceberg to the dark, esoteric depths comprehensible only to the most battle-hardened veterans. Picture an iceberg floating on water: what’s visible on the surface is just the beginning, while the real magic (or nightmare) lurks beneath, in increasingly inaccessible layers.

Personally, I love it. So I decided to create Java one:

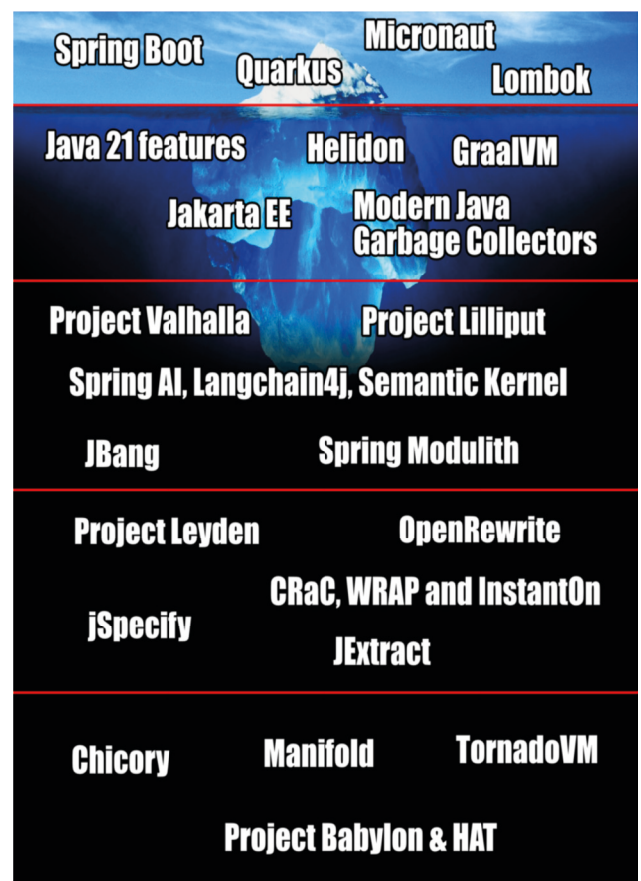
Author:



Head of Java/Kotlin Engineering at VirtusLab, Artur Skowronski has been in the industry for ten years. During this time, he has had the opportunity to work in various roles, such as Software Engineer, Tech Lead, Architect, and even Technical Product Manager. This diverse experience enables him to approach problems from a holistic perspective.

He still loves to get his hands dirty - for more than two years, he has been publishing weekly reviews of events in the JVM world

<https://www.linkedin.com/in/arturskowronski/>



That's why this article will take you on a Lovecraftian journey through the successive levels of understanding the Java ecosystem – from innocent beginnings to an abyss where code becomes something far more than mere instructions. Brace yourself, for the deeper you dive, the harder it will be to resurface!

PS: For using the word „pragmatic,” you'd be thrown overboard on this ship! Consider yourself warned.

Level 1: The Tip



At this level, you're like a newbie who just installed Java and thinks that „Hello World” is the pinnacle of programming achievement and your biggest drama is a missing semicolon at the end of a line.

At the same time, you start noticing the technologies that „professionals” talk about, and each one feels like a new level of initiation into some esoteric knowledge.

Spring Boot

Spring Boot is like a streamlined version of classic Spring. With traditional Spring, you get a toolbox and have to assemble everything yourself, like building LEGO without instructions. Boot was created to simplify this - no more manual wiring, just a setup that works out of the box.

The „need for speed” led to its creation. Classic Spring was powerful but slow to configure, making it frustrating for simple tasks. In 2014, Pivotal introduced Boot to automate setup and let developers focus on coding instead of configuration. It's still Spring, just with less hassle.

Quarkus

Quarkus was born to tackle the need for agile tools that fly on GraalVM and Kubernetes, where old Java frameworks lumbered like dinosaurs. Quarkus keeps it snappy and slim, perfect for microservices and serverless - none of that bloated nonsense. Unlike runtime-heavy frameworks, it shifts the grunt work to build time, launching quick and light, with a slick “Dev Mode” to keep coding smooth.

Started by Red Hat, it went big under their watch, then hopped over to the Commonhaus Foundation in 2024, joining a wider crew to keep rocking the Java world..

Micronaut

Like two previous, a Micronaut Java project emerges as a framework tailored for crafting nimble, high-performing applications, especially suited for microservices and serverless setups. Its creation was led by Graeme Rocher, co-creator of the Grails framework, poured their expertise into rethinking Java's potential. Unlike the clunky, reflection-heavy Java frameworks of yesteryear, Micronaut flips the script with ahead-of-time (AOT)

compilation, slashing startup delays and trimming memory bloat. Micronaut syncs effortlessly with GraalVM and can be treated as „official” solution for that technology.

Lombok

Project Lombok is a tool saving developers from the soul-crushing monotony of typing out getters, setters, and constructors till their fingers bleed. With annotations like @Data or @Getter, it whispers to your compiler, „Hey, let's generate that boilerplate nonsense behind the scenes,” leaving your code looking sleek and sassy. Simply speaking, it frees Java developers from Kotlin envy.

Sure, a few purists grumble it's like letting a wizard loose in your codebase, cranking „magic” and potentially turning debugging into a treasure hunt, but most folks just enjoy the ride nowadays.

Level 2: Just Below the Surface



Here, you start to realize that Java isn't just about writing code - it's also a battle with classpaths, as if you were Indiana Jones searching for a lost artifact. You've already had your first encounter with NullPointerException and are beginning to suspect that it's not a coincidence, but Java's revenge for your attempt to understand what was meant to remain hidden.

Java 21 features

In an era where many are still grappling with JDK 1.8 in production, last year's release of Java 21 introduced such a wealth of new features that its debut resonated beyond the usual JVM enthusiast circles. One of the most anticipated additions was Virtual Threads, which have revolutionized concurrent programming in Java. These lightweight threads allow developers to create and manage threads more efficiently by decoupling Java threads from OS threads, Virtual Threads enable the handling of numerous concurrent tasks without the overhead traditionally associated with thread management.

Another significant enhancement in Java 21 is Pattern Matching for switch statements. This feature enables more elegant and readable operations on various data types, allowing developers to write concise and type-safe code while avoiding excessive casting and complex conditional structures. Additionally, the introduction of Record Patterns facilitates easier deconstruction of objects, streamlining data handling and improving code clarity.

Jakarta EE

The first version of Jakarta EE, then known as Java EE was released in December 1999, bringing standards for Java-based enterprise applications. In the following years, the platform evolved, gaining popularity with the introduction of technologies such as servlets, Java Server Pages (JSP), and Enterprise JavaBeans (EJB).

Yes, this truly used to excite the community.

In 2017, Oracle announced it was open-sourcing Java EE to the Eclipse Foundation. Due to legal issues related to the name „Java,“ - which is a trademark - the new version of the platform was renamed Jakarta EE. The first release under the new name, Jakarta EE 8, was released in September 2019 and was fully compatible with Java EE 8, allowing a smooth transition for developers. Subsequent versions, such as Jakarta EE 9 in 2020, introduced significant changes, including the migration of the namespace from „javax“ to „jakarta.“ The latest version, Jakarta EE 10, was released in September 2022 – although by the time of this publication, we should already be enjoying Jakarta EE 11.

Helidon

Like Micronaut or Quarkus, Helidon emerges as a modern Java framework, particularly suited for microservices and cloud-native environments. Created by Oracle itself, it offers two distinct styles: Helidon SE (standalone version), and Helidon MP, which aligns with the MicroProfile project. Helidon integrates smoothly with tools like GraalVM and thrives in containerized setups such as Docker... yata, yata, yata, you already know the story.

However, the most interesting part of Helidon ecosystem is Helidon Níma. Built on Java's Project Loom and its virtual threads, Níma redefines concurrency by replacing the old thread-per-request approach with a lightweight, scalable model that handles thousands of tasks effortlessly. This makes it the most exciting piece of the Helidon puzzle, offering blazing-fast performance and a slim memory footprint, perfect for high-concurrency microservices and proving that Helidon isn't just keeping up with the times but pushing Java into the future.

GraalVM

As the knights of old sought the Holy Grail, so doth GraalVM rise as a mighty force within the Java realm, wrought by the hands of Oracle for deeds of high performance in the lands of microservices and cloud-native contrivances. It is a universal engine, not merely toiling at Java's behest, but enkindling it with the craft of ahead-of-time (AOT) compilation, whereby bytecode is transmuted into native executables. In fellowship with trusty tools such as Helidon and Quarkus, and flourishing within the bounds of Docker's vessels, GraalVM doth cleave the delays of startup and lessen the weight of memory's burden verily, thou knowest well this tale.

Yet, the chief wonder of GraalVM's domain lieth in its native image cunning. By forging Java works into solitary binaries, it casteth off the heavy mantle of the common JVM, granting speeds swift as the wind and a footprint scant as a miser's trove. This maketh it the most stirring piece of GraalVM's riddle, meet for microservices and serverless labours, shewing forth that GraalVM doth not only match the pace of present needs, but redefineth how Java doth fare in the wilds beyond.

Modern Java Garbage Collectors

One of the widely discussed is the ZGC, introduced in JDK 11. ZGC was designed for applications needing minimal interruptions, offering pauses in the millisecond range, even with multi-gigabyte memory heaps. A key innovation in ZGC is the use of so-called „colored pointers,“ which enable efficient tracking of objects in memory without requiring prolonged pauses in application threads.

Another garbage collector is Shenandoah, which also emerged around JDK 11 and is developed with a single focus - to minimize latency. Like ZGC, Shenandoah operates concurrently but stands out with its approach to heap compaction, which it performs without halting the application. This process traditionally caused significant pauses in older GCs like CMS or G1.



In this realm, the light fades, and project names sound like the forgotten names of ancient gods (Valhalla, huh). You feel like you're walking on the edge of madness, the code pulses with life and seems to guard hidden secrets. Something watches you from the darkness, something that knows.

Project Valhalla

Project Valhalla, kicked off in 2014, is Java's ambitious stab at fixing its performance quirks and healing the primitive-object schism. It's packing Value Classes - think lean, identity-free data types that ditch Object overhead, sitting inline in memory to slash heap usage and GC churn. Then there are Primitive Classes, improving int, double, etc., to play nice in generics. Add specialization to the mix, and generics stop erasing to Object, getting custom, type-specific bytecode instead. The payoff? Tighter memory layouts, fewer cache misses, and a VM that hums on modern hardware.

But here's the rub: after a decade, Valhalla's still a no-show. It's tangled in technical quicksand - retrofitting the JVM, preserving bytecode compatibility, and rejigging a crusty type system without sparking a revolt. Early drafts (JEPs 401, 402) and test builds tease us, but the finish line keeps shifting. It's a slog through backward compatibility hell and edge-case nightmares, all to make Java a leaner, meaner beast for data-crunching workloads.

Tonight, we (rather not) dine in Valhalla!

Project Lilliput

In 2021, Roman Kennke from Red Hat initiated Project Lilliput to reduce object header sizes in the HotSpot JVM from 128 bits to 64 bits, and potentially even to 32 bits. In the 64-bit HotSpot JVM, each object comprises a 64-bit „mark word“ (used for synchronization and hash codes) and a 64-bit class pointer -

for small objects (typically 5-6 words), this header constitutes a significant overhead. Project Lilliput introduced Compact Object Headers (JEP 450), which restructured the object header to 64 bits by relocating class metadata to a separate structure and optimizing lock management. This enhancement leads to tangible benefits, including reduced heap usage, less frequent garbage collection, and improved CPU cache utilization.

Unlike the perpetually deferred Project Valhalla, Project Lilliput has delivered on its promises. JEP 450 was integrated into JDK 24 (with a premiere in march of 2025) as an experimental feature, with ongoing efforts to further shrink headers to 32 bits. Key hurdles include maintaining compatibility with existing JVM code, optimizing lightweight locking mechanisms, and experimenting with further compression techniques. The outcome is a leaner Java memory footprint, with prospects for even greater efficiency in the future, this time without endless delays.

Spring AI, Langchain4j, Semantic Kernel

Alright, let's dive into the wild world of Spring AI, LangChain4j, and Semantic Kernel - the trio of Java frameworks for AI Bros.

Spring AI is part of the Spring ecosystem, strutting in like it owns the place with its mission to sprinkle AI magic on your Java apps. It's all about wrapping foundational models and vector stores in cozy Spring Boot autoconfiguration, complete with slick APIs for chatting up models and RAG (Retrieval-Augmented Generation) support for smarter answers.

LangChain4j, meanwhile, is the scrappy indie kid, porting the Python LangChain vibe to Java with a "we don't need no stinkin' Python". It's got AI Services - think declarative interfaces that hide LLM plumbing - plus tools for memory, embeddings, and chaining prompts.

Then there's Semantic Kernel, Microsoft's entry into the fray, supporting Java alongside C# and Python. Launched in 2023 too, it's got big dreams of multi-agent systems, but its Java support feels like it's still doing warm-up laps compared to the Python crew.

Spring Modulith

Spring Modulith is a clever little gem in the Spring ecosystem, helping you tidy up your Spring Boot project into neat, modular modules. Armed with structural validation and a ruleset for keeping things loosely coupled, it helps turn monolith into a maintainable project, all without forcing you to jump on the microservices bandwagon. Perfect for devs who want to keep things simple yet scalable.

JBang

JBang is a tool simplifying the process of running, managing, and creating Java scripts (pun intended), without the need for a traditional, heavyweight project setup. It allows developers to execute Java code as easily as a shell script by downloading

dependencies, compiling, and running the code in one seamless step—all from a single command.

With JBang, you can write a Java file, include dependencies using simple comments (like `//DEPS`), and run it instantly with `jbang script.java`, making it ideal for prototyping, scripting, or even building small utilities.

Level 4: Even deeper



Here, you lose all sense of time, just understanding why these initiatives were even started feels like a challenge in itself. The depth crushes your mind, yet you can't stop descending - something is calling you, something closer to the essence of knowledge (and the virtual machine) than yet another CRUD.

jSpecify

This initiative aims to standardize nullability annotations in Java, providing a consistent set of annotations that enable static code analysis to identify potential `NullPointerException` issues, thereby reducing such errors.

In July 2024, version 1.0 of the jSpecify project was announced. Project introduces four key annotations:

- **@Nullable:** Indicates that a variable, parameter, or return value can be null.
- **@NonNull:** Indicates that a variable, parameter, or return value cannot be null.
- **@NotNullMarked:** Applied at the package or class level, it designates all unannotated types within that scope as non-nullable by default, reducing the need for repeated use of the `@NonNull` annotation.
- **@NullUnmarked:** Allows disabling the effect of `@NotNullMarked` within a specific scope, facilitating a gradual adoption of annotations in large projects.

CRaC, WRAP and InstantOn

CRaC, WRAP, and InstantOn are technologies aimed at improving Java application startup and performance, each with roots in the broader concept of checkpoint/restore mechanisms, notably influenced by CRIU (Checkpoint/Restore In Userspace).

CRaC, or Coordinated Restore at Checkpoint, is an OpenJDK project that enhances Java startup times by taking a snapshot of a fully warmed-up JVM - after Just-In-Time (JIT) compilation and application initialization - and saving it as a checkpoint. Using CRIU under the hood, CRaC pauses the application, ensures resources like files and sockets are closed via its Java

API (with callbacks like `beforeCheckpoint` and `afterRestore`), and dumps the process state to disk - think Save States in video game emulators. Later, this snapshot can be restored rapidly, skipping the usual warmup phase, delivering near-instant startup and full performance from the get-go. Frameworks like Spring and Quarkus have integrated CRaC support, leveraging its ability to coordinate resources explicitly, a step beyond CRIU's more generic, uncoordinated process freezing.

WRAP (Warp Restore Acceleration Platform) and InstantOn build on similar checkpoint/restore ideas but cater to distinct use cases while sharing CRIU's foundational influence. WRAP, part of Azul's commercial offerings, extends CRaC-like functionality by optimizing the restoration process for large-scale deployments, focusing on reducing memory footprint and accelerating repeated startups - ideal for cloud-native or containerized environments. InstantOn, meanwhile, targets embedded systems and IoT, using CRIU to create a pre-initialized JVM image that boots in milliseconds, bypassing traditional JVM startup entirely. Unlike CRaC's runtime coordination, InstantOn emphasizes a one-time checkpoint baked into a deployable artifact, while WRAP refines the process for scalability.

All three trace their origins to CRIU's ability to freeze and restore Linux processes, but they diverge in scope: CRaC as an open-source JVM enhancement, WRAP as an enterprise optimization, and InstantOn as a lightweight, embedded solution, each tailoring CRIU's raw power to Java's unique needs.

JExtract

jextract is a tool developed as part of OpenJDK's Project Panama to simplify calling native C libraries from Java by automatically generating Java bindings from C header files. It works by parsing the header files using the Clang C API, then producing Java classes that leverage the Foreign Function & Memory API (FFM API, finalized in JDK 22) to interact with native code.

This eliminates the need to manually create method handles or memory layouts, letting developers call native functions as if they were regular Java methods, complete with type safety and resource management via arenas. It's particularly useful for integrating legacy C libraries or system APIs into Java applications, streamlining what was once a tedious process with JNI (Java Native Interface).

OpenRewrite

OpenRewrite is an automated refactoring tool designed to help developers eliminate technical debt in their codebases. Project features an engine that applies pre-built, community-driven „recipes“ to automate tasks like framework migrations, security vulnerability fixes, and stylistic consistency updates, reducing what might take hours or days into mere minutes. These recipes operate on a Lossless Semantic Tree (LST) representation of the source code, preserving original formatting while enabling precise, type-aware changes.

Initially focused on Java, OpenRewrite is now expanding to support other languages. While the project is maintained under an Apache 2.0 license, Moderne, the company behind OpenRewrite, is trying to scale this technology for enterprise use and build the proper business around it. In February 2025, they announced a \$30 million Series B funding round.

Project Leyden

Project Leyden is an OpenJDK initiative aimed at improving the startup time, peak performance, and memory footprint of Java applications by introducing a shift toward a more static, ahead-of-time (AOT) compilation model. Launched to address long-standing criticisms of Java's slow warmup and resource demands, Leyden seeks to evolve the JVM by exploring „closed-world“ assumptions - where the entire application is known at build time - allowing more aggressive optimizations than the traditional dynamic, „open-world“ approach of the JVM.

It builds on existing technologies like the GraalVM Native Image and the Ahead-of-Time compilation in the JDK, but with a focus on integrating these ideas directly into OpenJDK. By pre-computing class loading, linking, and JIT compilation steps into a single executable or a pre-initialized state, Leyden aims to deliver faster startup and reduced runtime overhead, making Java more competitive with languages like Go or Rust for serverless and containerized environments.

The project operates in phases, starting with concepts like „condensers“ that transform bytecode into optimized forms, potentially producing native binaries or cached runtime states, while preserving Java's portability and security guarantees. Unlike tools like CRaC or InstantOn, which snapshot a running JVM, Leyden emphasizes build-time transformations, avoiding runtime snapshot dependencies. It's still experimental, with no fixed release date as of early 2025, but it's guided by a collaborative effort within the OpenJDK community to balance performance gains with Java's flexibility.

Level 5: The Bottom



You've reached a place where there is no bottom, only an eternal void, writhing with the most ephemeral of projects, ready to devour you the moment you dare to ask yourself, „But how would I use this in production?“

And then, you will become part of the abyss, not a programmer anymore, but a vessel for something that should never have been awakened. Knowledge has consumed you, and you... you will never return to write another controller.

Chicory

Chicory serves as a Java-powered interpreter for WebAssembly, allowing Wasm modules to run natively inside the JVM. Rather than converting Wasm into machine-level instructions like conventional compilers, it processes the bytecode directly, bypassing any extra compilation phases for a streamlined approach.

Built entirely on Java's core library, Chicory avoids external dependencies altogether. This lean design boosts its portability across systems and slashes the chances of security flaws linked to third-party code, though it sacrifices some speed, as interpreted execution tends to lag behind the efficiency of pre-compiled binaries.

Manifold

On the top layer, we had Lombok, and now it's time to focus on its more hidden companion.

Manifold is a Java compiler plugin that extends the language with a slew of advanced features such as metaprogramming, extension methods, operator overloading, properties, and templates. As an example, it allows developers to integrate various data formats (e.g., JSON, SQL, XML) directly in Java code in a type-safe way, eliminating the need for separate code generation steps.

A key advantage of Manifold is increased productivity by simplifying code and reducing boilerplate. For example, you can add your own methods to existing classes (e.g., String), use native SQL in Java code with immediate type safety, or write templates with full Java expression support.

Manifold differs from other tools like Lombok because it functions as a compiler extension rather than a separate language, although technically, it is more fragile because it relies on internal mechanisms of javac, which ties it to specific JDK versions.

TornadoVM

TornadoVM was born as a research initiative aimed at accelerating Java code execution on modern accelerators – specialized hardware designed to accelerate specific computations, often through massive parallel processing, such as GPUs or FPGAs. The project started taking shape around 2018 as part of research at the University of Manchester.

The key challenge the developers set for themselves was to create a specialized just-in-time (JIT) compiler and multi-layered runtime logic that would automatically detect parts of the code suitable for parallel execution and offload them dynamically to the GPU (or other devices). Over time, TornadoVM has expanded to include additional backends (OpenCL, CUDA PTX, and SPIR-V), making it a universal environment for accelerating Java code on various types of hardware.

In simple terms: the goal of TornadoVM is to reduce the effort of developers who want to write in pure Java but still leverage

computational power beyond traditional CPUs.

Project Babylon & HAT

Babylon, a new OpenJDK project, enhances Java reflection with Code Reflection, enabling deep analysis of methods, lambdas, and runtime behavior beyond simple class and field inspection. This facilitates advanced metaprogramming in pure Java, allowing dynamic code generation and modification. While distinct from TornadoVM's hardware focus, Babylon's reflective innovations suggest potential synergies for optimizing Java applications across diverse platforms.

Its subproject, HAT (Heterogeneous Accelerator Toolkit), targets GPU acceleration with an abstraction layer supporting CUDA, OpenCL, and Vulkan APIs. HAT lets developers define „code models“ at compile time, which are translated into low-level OpenCL C or CUDA PTX at runtime. This enables parallel Java applications to harness GPU power efficiently, keeping development within the Java ecosystem, akin to TornadoVM's goals but driven by reflection-based techniques.

The goal of Babylon and HAT is to empower developers to write high-performance, GPU-accelerated Java applications using Code Reflection's flexibility and HAT's hardware integration. This combination supports optimized solutions for tasks like scientific computing or real-time data processing, all in pure Java, without low-level expertise. Still evolving, Babylon could complement projects like TornadoVM, strengthening Java's high-performance computing capabilities.

30 Years of Java, 25 Years of Enterprise Java

Over the last three decades, technology has been evolving at a breakneck pace, with innovations constantly redefining every aspect of application development. In such a fast-moving and dynamic landscape, few technologies stand the test of time, especially in computer science. Yet, Java has done just that. As we celebrate 30 years of Java and 25 years of enterprise Java, it's clear that these solutions have not only endured but thrived—adapting, advancing and proving their lasting value to software engineers worldwide.

The rise of personal computers transformed technology from a niche tool into something accessible to millions. This widespread adoption sparked collaboration and innovation, particularly through open-source projects, shaping the digital world we know today. Within this short yet prolific period of technological advancement, it's easy to assume that early developments have long been replaced. Yet, some technologies, namely Java and enterprise Java, have not only remained relevant but also essential. Their longevity isn't just a matter of persistence. It is the

result of solving fundamental challenges that developers have faced for decades.

A duke is born: Java

The first of such challenges became especially clear in the early 1990s. Back then, a common pain point was surfacing in the software industry: engineers involved in commercial programming, typically using C/C++, struggled with reliability and portability. As a result, software development was fragmented, with programs often requiring extensive modification to run on different platforms. Recognizing this gap, Sun Microsystems' „Green Team,” led by James Gosling, set out to create a solution that would solve the practical problems experienced by engineers, making the software development process a more platform-neutral process.

The answer was Java, which made its debut in 1995 as a processor-independent, object-oriented programming language for dynamic content delivered over the internet. Its defining feature, „Write Once, Run Anywhere” (WORA), made it an instant hit and helped Java quickly gain traction, as developers could write code once and deploy it across multiple systems without modification.

In particular, the technology offered a versatile, portable, easy to use yet powerful general-purpose solution for the newly born world-wide web, which was invented in 1989 and was beginning to gain momentum. The popularity of Java skyrocketed when the first popular internet browser, Netscape Navigator, incorporated Java support.

The availability, accessibility as well as the ability of Java to remain relevant were (and still are) enhanced by the Java Community Process (JCP) designed to support its development. Established in 1998, the JCP offers a formal mechanism for interested parties to propose new features to the Java platform (Java Specification Requests, JSRs) and develop standard technical specifications. This collaborative effort, which exists to this day, has been helping Java adapt to changing user and market needs while expanding the programming language into a comprehensive platform.

Thanks to all these elements, Java's reach soon began to extend and permeate all aspects of software development beyond web applications, encompassing even enterprise solutions. In effect, while the potential of the web to organizations was still untapped and enterprise web strategies didn't exist, pioneering companies were beginning to leverage Java.

Authors:



Chiara Civarni is a tech communicator with over 10 years of experience in producing technical content on a variety of topics related to the digital transformation. Her expertise includes automation engineering, robotics, software engineering and network technology. She holds a PhD for ETH Zurich and a MSc from the University of Southampton.



Steve Millidge is the founder, CEO and Technical Director of Payara Services. He is an expert in Java EE, Jakarta EE and MicroProfile, and a Project Lead for the Eclipse Foundation GlassFish and Jakarta EE platforms. Steve has spoken at a number of events about the performance and scalability of Java systems, including Java One, JBoss World, UK Oracle User Group Conference and Special Interest Groups, JaxLondon, GeeCon, EclipseCon, and Jakarta EE LiveStream. He regularly presents technical workshops and webinars on Jakarta EE, MicroProfile and big scale Java.

The rise of another star: J2EE

Recognizing Java's potential for enterprise computing as well as the demand for such a solution, Sun Microsystems introduced Java 2 Platform, Enterprise Edition (J2EE) in 1999. This continued to rely on the WORA approach of the Java programming language while including specifications designed to support the needs of enterprise frameworks, namely the design of large-scale applications and distributed systems.

Simultaneously, application server technology gained prominence as a valuable middleware solution for creating effective distributed applications, further supporting Java's adoption and platform independence. The nature of application servers and Java made the two technologies complementary, offering users a common, ubiquitous, platform-independent substrate upon which enterprise applications could be built. As such, this combination helped standardize enterprise application development, providing developers with tools for scalability, security and robustness while redefining how enterprises perceived and interacted with the web.

The next few years saw continued innovations in the Java and application server space, with a wealth of new APIs added to the language. In 2006, a number of key events influencing the future of the technologies took place.

Firstly, following the launch of Java version 5 and 6, J2EE was renamed Java Enterprise Edition (Java EE). Then, Red Hat acquired JBoss open-source middleware provider, which was seen as an expansion of the application server industry to support growing market needs. Finally, by releasing the core Java technologies under the GNU General Public License 2.0 (GPLv2) and formally open-sourcing the technology, Sun took a key step in helping Java become a collaborative, community-driven language.

In 2008, interest in Java grew further, as a new, popular platform using Java for its UI code and application development was released: Android. The following year, the first version of the videogame Minecraft further drove Java adoption, inspiring many young developers to learn the language in order to add new content to the game, like skins and mods.

A past of uncertainties and major changes

The story of Java and enterprise Java would be very boring without a proper hero's journey. In fact, we wouldn't be here celebrating if our beloved technologies didn't manage to win a decisive victory after a crisis. This happened in the 2010s.

Despite their success and large-scale adoption, Java and its enterprise counterpart began to face turbulent times, leaving many developers and industry leaders uncertain about the future of the technologies. In effect, the adoption of Java was declining. For example, on increasingly more popular mobile phones, Java was pushed aside by Apple, and although Google used Java as the primary app development language for Android, it also developed its own runtime, Dalvik.

The enterprise Java landscape was presenting equal challenges. A series of acquisitions, licensing changes, development slowdowns and commercial support endings for key products culminated in 2015, when contributions to the Java EE project stopped. This led to stagnation and industry-wide speculation about the platform's viability, support and sustainability.

Key initiatives were established to help ensure Java EE's longevity. In particular, a group of industry leaders formed the „Java EE Guardians“ in 2016. This group included leading industry experts, such as James Gosling and Java EE expert Reza Rahman, whose mission was to advocate for the continued development of Java EE and ensure its relevance in the enterprise landscape.

While the work from such organizations was successful, as it helped Java and enterprise Java survive this challenging period, uncertainties still permeated the industry. These, together with the simultaneous rise of microservices (in place of monoliths), cloud computing and DevOps, created a fertile ground for the development and adoption of key alternatives, such as the Spring framework.

Java and Jakarta EE today: stability and growth

The issues encountered in the 2010s made it clear that something had to change if Java and enterprise Java were going to survive in the tech world. They needed to address current challenges and find new, more suitable ways to support developers as well as the broader tech community.

The first major move was the shift in Java's release model. Instead of waiting years between updates, a new version of Java would be (and still is) released every six months. This was a game-changer, as it helped the technology evolve incrementally and consistently to stay relevant. The introduction of Java 9 in 2017, with its modular system (Java Platform Module System, JPMS), marked the beginning of Java's transformation into a more modern, flexible language.

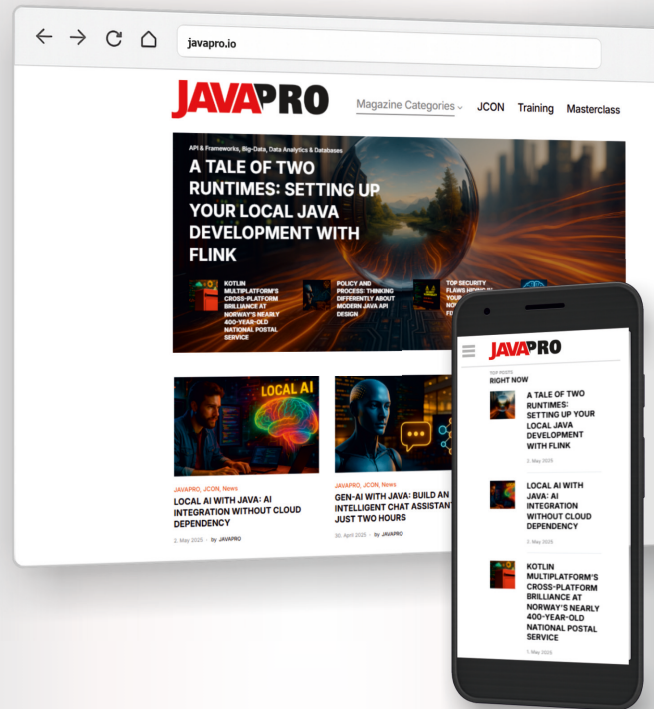
Meanwhile, Java EE was donated to the Eclipse Foundation in 2017 as Jakarta EE. Jakarta EE 8 was released in 2019 and ensured backward compatibility with Java EE 8. These two events represented a decisive step to help enterprise Java modernize and maintain its competitiveness.

At the same time, Java had to address its biggest disadvantage in the cloud-native era: performance. Compared to newer languages, Java applications were often slow to start and consumed too much memory, making them less suited for containerized environments. The solution came in the form of GraalVM, which allowed Java applications to be compiled ahead of time into native executables. This drastically improved startup times and reduced memory usage, making Java viable for cloud and Kubernetes deployments.

In addition, Jakarta EE leaned into microservices. The MicroProfile initiative provided lightweight APIs for cloud-native development.

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

Jakarta EE 10, released in 2022, delivered valuable features, such as modular architecture and microservices-first design.

As a result of these actions, Java and Jakarta EE have been able to re-establish a strong presence in the market. For example, Jakarta EE has seen a notable rise in usage, growing from 53% in 2023 to 60% in 2024, according to the Eclipse Foundation's 2024 Jakarta EE Developer Survey Report [1].

The code ahead: what's next for Java and enterprise Java

Just like in the best hero stories, our protagonists Java and enterprise Java managed to improve and recover, despite the challenges in the 2010s. In fact, we are here commemorating their 30th and 25th anniversaries and hoping to celebrate 30 more.

The latest developments with the two platforms hint at a positive future. Several key OpenJDK initiatives, such as Project Loom, are helping advance Java, so that it remains a robust and modern language as well as framework, well-suited for contemporary application development.

The Eclipse Foundation's Jakarta EE Future Directions Interest Group is designed to help the technology remain up to date by looking at emerging technologies, new industry use cases and integrations, among others.

In addition, the Open Regulatory Compliance Working Group (also from the Eclipse Foundation) will assist open-source participants with adhering to global regulatory requirements, such as the EU's Cyber Resilience Act.

Conclusions

In this highly technological era, it's all too easy to forget about the many computing technologies that once ruled our work and our lives but are now irrelevant. Only those that have been able to evolve to address changing needs have succeeded in remaining a mainstay. Java and enterprise Java stand as rare examples of longevity in the industry, driven by constant innovation, large-scale engineering investments and strong community support, such as the JCP.

Their ability to evolve has ensured their place as foundational technologies in application development. Even more, their continued growth and relevance suggest that their journey is far from over.

References

[1] Eclipse Foundation. (2024). 2024 Jakarta EE - Developer Survey Report, October 2024. Available at: <https://outreach.eclipse.foundation/jakarta-ee-developer-survey-2024> [Accessed: 5 February 2025].

Policy and Process: Thinking differently about Modern Java API design



There are many guides on achieving the right architectural balance and multiple thoughts about the 'right' way. In this article, I want to distil some of this thinking, apply it to Java API design, and use modern Java features. Historical patterns like AWT or RMI often influence older Java API design thinking, and it's time to remind readers that modern Java features offer new, more powerful, and frequently safer ways to design APIs.

History

We've seen many approaches to API design ever since the first computer. At their heart is the desire to create an architecture that balances simplicity over adaptability and future enhancement. As a Java developer, you'll often hear references to some standard terms:

Separation of Concerns: This fundamental design principle advocates separating code into distinct sections, each addressing a separate concern.

Strategy Pattern: In design pattern terminology, particularly from the "Gang of Four" book, the Strategy pattern encapsulates different strategies in separate classes and allows them to be swapped easily without affecting the context

Inversion of Control (IoC): While typically associated with dependency management, IoC is also about decoupling task execution. It creates a hard barrier between code, what the application does, and what the framework provides.

Command Pattern: This pattern separates the action's requester from the object that executes the action. Command objects encapsulate a request as an object, letting you parameterise clients with different requests, queue or log requests, and support undoable operations.

Delegation Pattern: This pattern involves two objects where one object handles a request by delegating to a second object (the delegate).

There are many other patterns. This is a rich vein of discussion, and there are many opinions.

Lines in the Sand

All the great concepts above have tremendous value, but where

Author:

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



<https://www.linkedin.com/in/noregressions/>

those architectural lines begin to melt away is when we talk about user 'participation'. Any design requires thinking about how the user interacts with the API. How do we draw hard lines and create firm boundaries when code can be malleable and developers are inventive?

Often, our design requires or invites participation in unexpected ways. Ways we didn't intend or expect.

Some programming languages make creating firm boundaries challenging, but Java has many features that can be used to create boundaries that are more than lines in the sand.

In this article, I want to extract an underlying principle or two and look at what we have available to create robust designs. Designs that allow just the right sort of participation without too much opportunity to directly or inadvertently compromise our API.

Classes and Interfaces

Whatever design pattern you follow or the framework you use, you still write code.

The Java API design toolkit we use essentially consists of Classes and Interfaces. We might use Annotations to signal requirements

to containing systems, but, as the name suggests, they are annotations and do not have to be honoured.

As an API designer, you will consider how someone consumes the API, what constraints you must apply, and where. Many functional and nonfunctional elements, such as performance, security, observability, etc., must be considered, and you must decide how the API consumer will participate in all of these factors.

Participation – it’s not a spectator sport

Participation means assessing how much the API user is a consumer rather than a provider. Historically, the way we use Java classes and interfaces has remained unchanged.

Common approaches include:

- Using the provider pattern to allow objects to be ‘magically’ instantiated. i.e. via IoC or a Java Service Provider mechanism.
- Providing an interface and Javadoc to explain expected behaviour for each method
- Providing an abstract class for others to subclass and fill-in-the-blanks.
- Providing a concrete class that someone can subclass to override behaviour
- Providing a concrete class that has a constructor with parameters
- Providing a concrete class and a selection of setter methods – the JavaBean pattern
- We might occasionally use the Builder pattern to allow the instantiation of a complex object.

Force Fitting

We assume (and occasionally even try to force) how the end user will participate in all these standard techniques.

IoC tries to force the user to be only a consumer. Builder patterns or final classes also attempt to force consumption only, albeit with some flexibility in configuration. Meanwhile, abstract classes and plain interfaces invite the user to participate in the API’s internal behaviour as both a producer and a consumer.

The JavaBean pattern is a dreadful mix of consumer and producer but highlights the other aspect of participation: understanding one’s place in the process.

Ideally, once an object is instantiated, it’s fully configured, and any other method calls are to retrieve data or transform the object’s state. What the JavaBean model does is blur configuration with use. Any setter method can be called at any time, so the implementation has to deal with the chance of its configuration being modified during use. That’s a challenge and a ready source of nasty bugs.

The API design invites a particular form of user participation. JavaBeans are bad practice, and so is using subclassing as a

consumption principle. Whether an abstract or concrete class, the requirement that the user use subclassing to participate must be revised.

This approach requires the user to know much more about the API’s work; the user code is now part of the API. The user code must understand its responsibilities and its place in the process. It needs to understand what state its parent is in, when it will get called, and what it’s allowed to do when accessing the parent state.

Overriding or implementing methods as an API design choice forces the user to become both producer and consumer, encouraging them to override other methods or dive into the parent class’s internals. The tie between user and API is now strongly coupled and personal.

Let’s talk dishwashing machines.

Examining one, we can see that simplistically we have a few buttons and knobs that form the control panel, and we interact with the machine by loading salt, coupling up power, water and drainage and then regularly loading the machine with the dishwasher tablets and, of course, dishes.

We can see three sections by distilling this physical design into software principles.

Configuration: Where the machine is plumbed in and connected to a power supply.

Policy: The control panel allows the user to describe what they want the machine to do.

Process: Loading, policy execution, washing the dishes, Unloading etc.

Note how the dishwasher users’ participation is controlled and anticipated. Of course, they must remember to load dishes or tablets and not overload the dishwasher. Many of the common mistakes are discoverable by the machine. Open the door, and the machine stops. No tablets or softener, etc., so a light goes on. Depending on how clever your machine is, it is more likely to defend against common mistakes.

At no point in these activities is there any expectation or ability for the user to step outside their part of the process. There is a clear separation between what the machine does and what the user does.

The user doesn’t get involved in the actual act of dishwashing. They support the machine in fulfilling its function. The user selects a policy to tell the machine what sort of washing to do, provides dishes to wash and then extracts the clean dishes.

Configuration, Policy and Process

This model is great for API designers to follow. Thanks to recent additions to the Java language, we now have the tools to follow this pattern to deliver robust, secure APIs and control user participation.

Configuration is where the environment is discovered, and elements that affect all usage are gathered. This might be via a pre-instantiated builder class, loaded by IoC, etc., or the parameters passed into a constructor. Think of them as 'rules of physics' that are fixed for the API during its lifetime in the JVM.

Policy is like the washing machine control panel, the rules that this particular instance of the API will follow. Policy declares what you want the API to do, not how it does it or what data it will process. A policy is a reusable but immutable object. Think of it akin to something like a declarative contract.

Process is the well, process of executing the policy on a particular data set. The process uses information from the policy and configuration to do its job. In Java terms, the policy might contain selection criteria for data as a predicate. The process uses this predicate internally but cannot interact with any other part of the process. It's isolated and contained.

Clean Design

The separation of configuration from policy and policy from process is an essential concept that promotes cleaner, more modular, and maintainable code. This principle contrasts with some of the older design paradigms in Java, where the lines between these elements were often blurred, leading to complicated codebases that needed to be revised to maintain and extend.

For example, older Java designs, such as those found in the Abstract Window Toolkit (AWT), encouraged a blending of policy and process. Developers often had to subclass abstract classes or override methods to integrate with an API. This design forces developers to engage deeply with the internal workings of the classes they are extending, leading to tightly coupled code that is difficult to manage and prone to errors.

Having to have special knowledge of the working of related classes - especially about method call order, expected responses, even what exceptions to throw when is not only nonoptimal it's both fragile and restrictive. Modularity and separation of concerns become diluted and the end result is a API in name only.

Practical Implementation in Java – expanding the designer's toolkit

I hinted a little about using predicates above, but that's only a part of what is available in modern Java. Java gives us classes, interfaces, lambdas, streams, enums and records today. Add in inner classes, sealed classes and even sealed jars, and we have an extensive palette of tools that allows us to deliver better and cleaner APIs.

There is too much to unpack in this article alone, so let's look at a simple worked example to give you a taste.

Scenario

The scenario is that of an API that can visit a tree structure

and stream the contents. There is an implicit visitor pattern here, but it's hidden away. The consumer's experience is more straightforward.

Configuration

What configuration might we have for a tree streamer? What elements would we expect to be needed to be fixed in place for the lifetime of a Java process?.

In general we're looking at non-functional aspects and default values.

A fundamental question for any API designer is how the first object is instantiated and populated. In Java frameworks it's common to see IoC patterns or service-loader approaches where a default constructor is called. At load time, these IoC or service loaded classes will have to discover their detailed configuration. That might come from system properties, envvars or other system configuration methods.

That means that the separation between configuration and policy is already blurred but even so, it's important to be clear about what is fixed and what is flexible. It may be that rather than using IoC or service loaders to instantiate API objects, it's feasible to instead instantiate an API builder.

For example, configuration will cover default values (such as rate limiting) and expensive discovered system configuration that should only be done once. It might be that our tree streamer has preferences to, say, the HTML parsers to use. Our code might be smart enough to detect what is available on the classpath and decide accordingly. That is certainly something to decide at configuration time. Whether this decision is flexible or not becomes a policy question.

Policy

Let's define the policy for navigating and traversing a tree structure. We need to know how to interpret a node in the tree as a parent and decide which nodes to visit. Beyond that, there might be particular types of trees we're going to visit that will have other policy requirements. If the tree is a view over a website, where we're streaming the URLs we discover, we'll want to add rate limiting and possibly a depth limit. If we're streaming a file system, we might want to keep the depth limit, but rate limiting is probably not useful.

Builder Pattern – Part of the API

Since this scenario is complex, our policy will be created via a builder pattern. Unless the policy is simple, using a builder gives us more flexibility to provide an interface that is easier to police for misconfiguration and easier for the consumer to understand.

```
import static NavigatorPolicyBuilder.builder;

NavigatorPolicy shortPolicy = builder()
```

```

        .rateLimit(100, Duration.ofMinutes(1)) //
play nice
        .defaultReader(new HTMLRefNavigator())
        .maxDepth(2)
        .build();

```

In this example, we're setting the policy on how to navigate a website. We've created a policy that we can reuse. The builder pattern (using fluent style) is easy to understand and validate, resulting in a reusable but immutable policy object. The builder is part of the API but is not a process object.

Secret Sauce

In this scenario, `NavigatorPolicy` is a Java Interface. Until recently, that would allow anyone to implement it and subvert any restrictions we might put in place via the builder. With the advent of Java 17 and the arrival of the 'sealed class' feature, we now have control over who can implement the interface.

This is important because we want the `NavigatorPolicy` object to be read-only and remove any public ways of changing its data. Therefore, we must separate the mutator methods from the interface and hide them.

Sealed classes let us define which classes can implement an interface and which can subclass it. Only those on the permitted list can be implementors or subclasses.

The code for the `NavigatorPolicy` interface looks like this.

```

public sealed interface NavigatorPolicy permits
AbstractNavigatorPolicy {

    LinkReader handler(Connection.Response r);
}

```

This code indicates that only the `AbstractNavigatorPolicy` class can implement it. The `AbstractNavigatorPolicy` class definition includes

```

public abstract sealed class
AbstractNavigatorPolicy implements NavigatorPolicy
permits NavigatorPolicyBuilder.MyUriPolicy {

```

This shows that it implements the sealed interface and is, in turn, itself sealed. Allowing only an inner class of the builder to subclass.

`MyUriPolicy` looks like this below. That's it. Note the final setting and private constructor.

```

public static final class MyUriPolicy extends
AbstractNavigatorPolicy {
    private MyUriPolicy() {
    }
}

```

The `MyUriPolicy` is an inner class inside the builder `NavigatorPolicyBuilder`. Thus only the builder can instantiate

this particular policy class since it alone has access to the private constructor. Making the class final ensures that there is no opportunity for the behaviour of the policy object to be subverted through subclassing.

What might have escaped your notice is that the `MyUriPolicy` class is hollow. It has no functionality. The `AbstractNavigatorPolicy` class implements the behaviour. This pattern allows us to keep control over who can implement and instantiate our API classes while allowing us to keep the functionality in a separate class file (`AbstractNavigatorPolicy.java`) and not clutter up the builder with long, complicated inner classes.

Class Relationships

This code pattern of a sealed interface + sealed abstract class + inner class concrete (hollow) class provides good code separation and strong controls on how they interact and can be used.

The API consumer has a `NavigatorPolicy` that can be interrogated but not overridden or replaced. Without using reflection, the consumer cannot discover policy internals.

From the API designer's POV, the builder is part of the API, which is evident in its intention, yet the implementation details are hidden away and can not be changed. The builder might return `MyUriPolicy` this time, but maybe a change in the future means that the next time `build()` is called, it returns `YourUriPolicy`.

The consumer is not affected, and the API is rigorous yet evolvable. Builders, especially using a fluent approach become a new form of API. One that is more open to change than traditional Java APIs expressed in interfaces and classes directly. Let's see that in action.

Process Time – STREAMS and things

Here's a related code set for using the API to do work.

'base' is the URI the API is going to visit. The policy object, `shortPolicy`, comes from above.

```

URITreeStreamVisitorBuilder.newInstance(base)
    .policy(shortPolicy)
    .select()
    .on(Link.class)
    .consume(this::handleLink)
    .visit();

```

The first thing to note is that we have a reasonable DSL-like interface using a fluent style with judicious inner classes. Next, the consumer's involvement with the process is controlled. The consumer is effectively on the end of a stream of data.

In this case, the API allows some inline filtering for architecture and performance reasons. Still, it could have been written to return a stream and have the filtering and processing to be done entirely by the consumer.

We did not need to ask the consumer to provide a custom class other than the callback for received data.

Let's expand our use of this builder to see how much we can do with this style of API design.

```
URITreeSteamVisitorBuilder.newInstance(base)
    .policy(shortPolicy)
    .select()
    .on(Link.class)
    .consume(this::handleLink)
    .on(Meta.class)
    .consume(this::handleMeta)
    .otherwise()
    .consume(o -> {System.out.
println(",other:"+o);})
    .visit();
```

Remember this is all valid Java code. Now, we have an additional `on()` clause and an `otherwise()` method. Hopefully, you can see how this mirrors a select expression. Again, the API provides ways for the consumer to participate in the process, but as an consumer, The internals are kept hidden away. The API is not just a simple class or interface. The API here is a fluent style, DSL-like mechanism that is flexible and extensible while allowing the API designer control over behaviour and interaction.

A bonus is that this style works well with IDEs; the same code seen in the IDE looks like this.

```
URITreeSteamVisitorBuilder.newInstance(base)
    .policy(shortPolicy)
    .select()
    .on(Link.class)
    .consume(this::handleLink)
    .on(Meta.class)
    .consume(this::handleMeta)
    .otherwise()
    .consume(o -> {System.out
        .println(*other:**o);})
    .visit();
```

Note all the additional info about the intermediate classes of this DSL-like construction. This really helps with understanding the available options as the DSL is used and gives you a hint of how the inner classes are stitched together internally.

DSL-Like is your new API

From an API design POV, using sealed classes to add extra controls on who can do what with your code is compelling. You could stop there, having already reduced the chances of your API being deliberately or accidentally compromised.

Including the builder approach (with the same sort of DSL-like structure) takes the API design into new territory and transforms it from a static Java interface or class into something more dynamic.

Now, your API can be a sophisticated builder that can be extended in ways that a simple Java interface or class would struggle to match.

Conclusions

I hope I've given you a taste of what can be achieved using Java's features.

Using Inner classes, Sealed Classes, Lambdas, Streams, Enums, etc., and by thinking again about how we separate configuration and policy from the process, it's possible to create APIs that provide clear, easy-to-use yet enforceable designs that are hard to compromise but can be extended and enhanced.

JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



www.javapro.io



#JAVAPRO #FRAMEWORK #API

EclipseStore: The Evolution of Java Persistence



Author:



Markus and his teams have been working on IDE tools for Java and database development, as well as on 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.

<https://linkedin.com/in/markuskett>
<https://twitter.com/MarkusKett>

Database development in Java has always been a challenge, often feeling like a battle against mismatched paradigms. From the start, developers wrestled with complex mappings and performance bottlenecks when integrating Java with databases. EclipseStore, an open-source project under the Eclipse Foundation, was created to change this. Its mission is to deliver a Java-native persistence layer that feels like an integral part of the language, not a workaround for database limitations. By storing Java objects directly, it aims to simplify and accelerate development, aligning persistence with Java's core strengths.

Whatever design pattern you follow or the framework you use, you still write code.

What We Love About Java

Java developers cherish the language for its object-oriented programming, type safety, and intelligent tooling. Features like code completion, compiler checks, and elegant APIs make coding intuitive and reduce potential errors. Clean and testable code is highly valued in the Java community. However, these strengths often vanish in database programming, where developers face clunky APIs and error-prone string-based queries statements that are neither type safe nor enable code completion, compiler checks, and DBUnit testing. In many aspects, the elegance of

Java's ecosystem and clean code standards are missing. At many Developers miss at various points the elegance of Java's ecosystem and clean code.

What We Get with Databases

Databases operate on fundamentally different principles from Java, supporting SQL, stored procedures, functions, or PL/SQL, and numerous additional data types unknown in Java. That's why the strengths of Java often vanish in database programming, where developers face clunky APIs and error-prone string-based queries statements that are neither type safe nor enable code completion, compiler checks, and DBUnit testing. In many aspects, the elegance of Java's ecosystem and clean code standards are missing. At many Developers miss at various points the elegance of Java's ecosystem and clean code.

Biggest Pain-Points with RDBMS

Relational Database Management Systems (RDBMS) pose significant challenges for Java developers. Mapping Java objects to relational tables requires intricate Object-Relational Mapping (ORM) frameworks, which introduce complexity and maintenance overhead. Query performance suffers due to latency from data conversions and network round-trips. Joins across multiple tables can cripple runtime performance, especially in high-load scenarios. Schema migrations are error-prone, often requiring manual intervention. Concurrency handling demands careful locking strategies, complicating multi-user applications. The mismatch between Java's object model and relational structures creates a persistent "impedance mismatch." These issues slow development and inflate costs, frustrating developers seeking seamless integration.

Persistence History in Java

In Java's early days, database programming was rudimentary, relying on JDBC with raw SQL queries that were error-prone and hard to maintain. Enterprise JavaBeans (EJB) 1.0, introduced in 1998, aimed to simplify persistence but brought heavy complexity, with verbose XML configurations and poor performance. Hibernate, launched in 2001, revolutionized the field by mapping Java objects to relational tables, abstracting SQL, and offering a more natural way to store objects. The Java Persistence API (JPA), standardized in 2006, built on Hibernate's ideas, providing a vendor and database-neutral ORM standard. The goal of these abstractions was to let developers focus on storing objects, not wrestling with database specifics. Spring Data, introduced in 2011, and Jakarta Data, evolving from Java EE, added higher-level abstractions, simplifying repository-based access. These tools made development more comfortable, but drawbacks persist: ORM frameworks still require mappings, introduce latency, and struggle with complex object graphs. Performance overhead from data conversions remains a challenge, and schema migrations can still disrupt workflows. Despite progress, the dream of truly Java-native persistence remained elusive.

NoSQL Databases

NoSQL databases emerged in the 2000s to address RDBMS limitations, prioritizing scalability and flexibility over strict consistency. They introduced new data models like key-value stores, JSON documents, and graphs, better suited for unstructured or semi-structured data. JSON document databases became very popular. However, Java developers face pain points also with NoSQL: APIs often require JSON conversions, breaking Java's type safety, and complex queries lack the elegance of Java's programming model. Key-value databases often force Java developers to manually handle object serialization/deserialization and lack built-in support for complex relationships and traversals that are natural in Java's object-oriented and graph-based models. This leads to more code for managing data structures and querying. Graph databases require Java developers to learn proprietary graph models and query languages, differing from Java's object-oriented approach. While OGMs exist, they introduce mapping complexity between Java object graphs and the database's node-relationship structure, leading to a different paradigm for data modeling and retrieval. The shift to NoSQL solved some scalability issues but introduced new integration challenges for Java applications.

Introduction to EclipseStore.io

EclipseStore is a Java-native persistence framework designed to work as developers expect from their language. It uses the native Java object model, supporting any Java type, including complex object graphs of any size and depth, with full support for inheritance, polymorphism, and circular references. The primary goal is to store any Java object graph, however intricate, and restore it in RAM, either fully or partially, as needed. Objects can remain in memory for as long as required, and developers can remove them when RAM is scarce, with lazy-loading ensuring only necessary data is loaded. By eliminating mappings and conversions, EclipseStore avoids the latency of ORM frameworks and JSON-based NoSQL systems.

To achieve this, EclipseStore serializes objects directly to disk using the Eclipse Serializer, a custom-built component addressing the flaws of Java's built-in serialization. The Eclipse Serializer is faster, handles complex object graphs effortlessly, and is highly secure, preventing malicious code injection by never executing code during deserialization. EclipseStore is open-source under the Eclipse Public License (EPL) 2.0, available on Maven Central, and integrates with frameworks like Micronaut, Spring Boot, Quarkus, and Helidon.

Java Object Graph

A Java object graph is a network of interconnected objects, where each object may reference others, forming a structure of arbitrary complexity. For example, a BookStore object might contain a List<Book>, where each Book has an Author and a List<Review>, with circular references (e.g., a Book referencing its Author, and the Author referencing their Books). EclipseStore requires a root object that serves as the entry point for the

persisted objects. All objects that are reachable from the root object can be persisted. Vice versa, when you want to retrieve your persisted data, you retrieve it by accessing this root object. All other objects in the graph are then reachable from the root. EclipseStore lets developers design such models freely, without restrictions like annotations, specific interfaces, or external configurations. Any object model that works in Java can be stored by EclipseStore, preserving the full object graph, including relationships and state, for seamless persistence.

Store Operation and Storage

EclipseStore stores objects in an open binary format using the Eclipse Serializer. Objects are appended to the storage files, never overwritten, ensuring data integrity. Each store operation is a single method call, a blocking, atomic all-or-nothing operation, with a transaction journal guaranteeing ACID compliance and full consistency. For example:

```
storageManager.store(myObjectGraph);
```

Developers can choose what to store - full graphs, subgraphs, or specific objects - offering flexibility. Storage targets include local disks, mounted volumes, cloud object storage like AWS S3, or even traditional databases; however, they just store binaries without mapping to their data models. As EclipseStore just writes binary data directly to a storage medium supporting multithreaded IO operations, write performance and data throughput are extremely high and almost equivalent to the medium's performance.

Loading Objects in RAM

At system startup, EclipseStore loads only the object IDs of the entire object graph into RAM, restoring the graph's structure without loading data. Developers can configure eager loading for instantly needed objects or lazy loading for on-demand access, making EclipseStore efficient even on infrastructure with limited RAM. Objects are written directly to the heap, under the JVM's control, ensuring standard garbage collection applies.

Access to EclipseStore

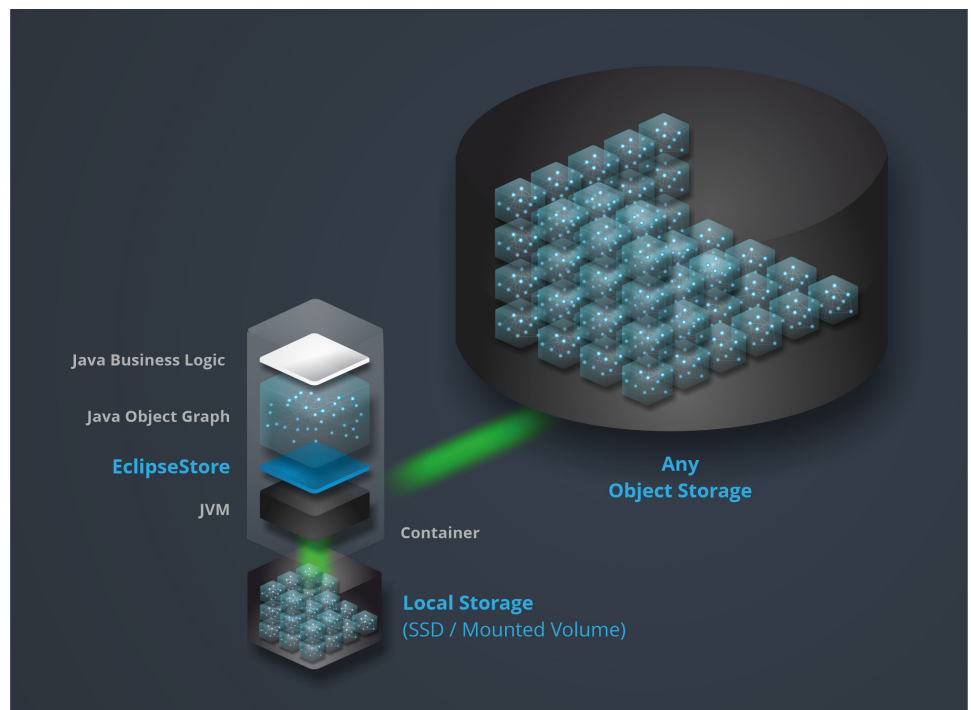
Access from external applications is possible through the implementation of REST endpoints or GraphQL interfaces. Both variants are supported by most applications, such as data analytics and BI tools.

EclipseStore provides a REST interface and a tool with an UI for accessing the persistent binary data in the storage. For easy data migration, the framework offers CSV import and export.

Exploring Potential Challenges and Solutions

EclipseStore addresses several challenges to deliver robust persistence:

- **Schema Migration:** When classes evolve, legacy type mapping ensures compatibility.
- **Storage Growth:** EclipseStore's housekeeping and garbage collection processes remove unreachable or corrupt objects from storage, preventing endless growth. Behind the scenes, a garbage collector constantly deletes corrupt or outdated files, maintaining efficiency.
- **Limited RAM:** Lazy-loading ensures only necessary objects are loaded, allowing EclipseStore to handle databases larger than available RAM. Developers can define which subgraphs load on-demand, optimizing memory usage.
- **Concurrency in Multi-User Apps:** In multi-user applications, all users access a single object graph in RAM. EclipseStore's Locking API provides fine-grained locking to prevent conflicts. Developers can also use any standard Java concurrency concept, like synchronized blocks, reentrant locks, or ConcurrentHashMap, for robust handling.
- **I/O Overhead:** Critics argue that I/O overhead could be high for large collections, but EclipseStore mitigates this with its optimized binary format and lazy-loading. Unlike traditional databases, which process queries on servers and transfer results, EclipseStore loads only required objects directly into RAM. Also, special pageable collections minimizing I/O. There is also an extension providing a powerful indexing concept comparable



to database indexing.

- **RAM Management:** Developers can manually remove objects from RAM or set timeouts, freeing memory without affecting storage. Objects are only removed from storage when no longer reachable by the JVM.
- **Indexing and GigaMap:** For massive datasets, MicroStream's enterprise extension offers indexing, enabling searches across billions of objects in milliseconds using off-heap bitmap indexes. The GigaMap, a high-performance map implementation, supports billions of entries with automated lazy-loading and minimal I/O, ideal for caching and real-time analytics.
- **Distributed Applications:** EclipseStore operates within a single JVM, but MicroStream's Cluster extension enables distributed applications. It supports load balancing, replication, and elastic scalability, available as SaaS or on-premise. Combined with Java Streams and indexing, it transforms EclipseStore into a powerful in-memory data processing and caching platform with ACID-compliant persistence.

Potential Use-Cases for EclipseStore

EclipseStore's Java-native persistence and high-performance capabilities make it versatile for a range of applications - from monoliths to microservices, GraalVM native images, and even Android devices.

It can replace Hibernate in projects needing simple, low-overhead Java object persistence, eliminating complex mappings and delivering microsecond response times. In microservices architectures, where each service requires its own lightweight persistence, EclipseStore shines due to its minimal footprint and seamless integration with frameworks like Quarkus and Spring Boot. It's ideal for UI data persistence, such as storing user session states or form data in web or mobile apps, where a lightweight storage solution ensures fast response times without database complexity. EclipseStore also enables graph analytics, enabling rapid traversal and querying of complex object graphs, like social network connections or supply chain networks, leveraging the Java's in-memory efficiency. For data analytics, it accelerates real-time processing of large datasets, such as financial transactions or IoT sensor data, with the GigaMap extension handling billions of records. Fraud detection is another strong use-case, as EclipseStore's microsecond queries and indexing enable real-time analysis of transaction patterns to flag anomalies instantly. Additional applications include caching for high-traffic APIs, event sourcing in domain-driven design, and embedded persistence in IoT devices, where its small footprint and ACID compliance ensure reliability and speed.

Challenges with EclipseStore

The biggest challenges with EclipseStore are not technical but human. EclipseStore represents a paradigm shift in database programming. There's no longer a relational data model, no

classic selects, no traditional database transactions, no SQL - even the entire database server no longer exists. Although EclipseStore is a pure Java approach, even Java developers often find it difficult to let go of the relational model and think completely object-oriented when it comes to data storage. Developers who have already worked with NoSQL databases may find this easier because they have already learned that data processing does not have to be relational. However, once you internalize this, the elegance of the approach becomes clear.

If the EclipseStore approach has to be discussed and defended with database specialists, things can become challenging. Relational databases have been leading the way for decades, ubiquitous, and their success proves them right. Database specialists can also usually point to many years of experience and projects. However, the disadvantages of relational databases and ORM frameworks - such as high complexity and development effort, data model limitations, performance issues, and lack of scalability - are often overlooked. Due to a lack of Java knowledge, you may not be able to fully grasp the advantages of EclipseStore. This can make the discussion about EclipseStore very difficult.

Conclusion

As Java celebrates 30 years, EclipseStore emerges as a transformative force in persistence, aligning database development with Java's core strengths. By storing native Java object graphs without mappings or conversions, it eliminates the pain points of RDBMS and NoSQL systems. Its microsecond query times, ACID compliance, and support for diverse storage targets make it ideal for cloud-native microservices, monoliths, and edge devices. With features like lazy-loading, secure serialization, and enterprise-grade indexing, EclipseStore empowers developers to build faster, simpler, and more cost-effective applications. As Java evolves, EclipseStore ensures persistence feels like a natural extension of the language, ready for the next 30 years of innovation.

Links:

- EclipseStore website: www.eclipsestore.io
- Code on GitHub: <https://github.com/eclipse-store/store>
- Getting started: <https://docs.eclipsestore.io/manual/storage/getting-started.html>
- Indexing and distributed EclipseStore apps: <https://microstream.one/>



#JAVAPRO #DATA

Java and Databases: From SQL to JPA and NoSQL

Over the past 30 years, Java has significantly changed the way applications interact with databases. In the early days, Java applications typically used a client/server model and a three-tier architecture, where SQL statements were often embedded as string literals within the application. These statements were sent to databases through straightforward interfaces such as JDBC.

As programming paradigms shifted towards Object-Oriented

Programming (OOP), Java developers began to integrate business logic more seamlessly into data structures. This move represented a shift from purely database-centric models to application-first designs. This transition enabled data abstraction through application programming interfaces (APIs), allowing SQL database tables and rows to be represented as classes and objects within the application. As a result, data sharing and persistence also evolved, moving from strictly normalized relational tables to document-oriented databases that align more closely with object-oriented structures and specific use cases.

The role of databases has evolved from being the central repository for enterprise information systems to becoming a component of a wider application ecosystem. Nowadays, databases are designed with specific services and access patterns in mind. They often function within more complex architectures, where data is transferred to other databases to support various distinct services.

Pre-Java Era: Interacting with Databases

Before Java's emergence, database interactions were mainly handled through SQL statements that were deeply integrated into applications. The SQL standard governing this has not changed since SQL-92, which outlines four primary methods for executing SQL statements.

Author:



Franck is a Developer Advocate for MongoDB and previously for YugabyteDB (distributed PostgreSQL). With 25 years of experience in database consulting for development and operations teams, Franck speaks at conferences, writes articles, and participates in social media to continuously learn and share his knowledge of all databases. He is recognized as an Oracle Certified Master and an AWS Data Hero.

<https://www.linkedin.com/in/franckpachot>
<https://twitter.com/FranckPachot>
<https://dev.to/franckpachot>

Two methods focused on end-user interactions through the Command Line Interface (CLI), today referred to as data scientists.

- **Direct Invocation:** SQL statements are executed directly via command-line interfaces with literal values. This method allows for instant command execution without additional code.
- **Dynamic SQL Statements:** These are prepared with placeholders (bind variables or parameters), allowing specific values to be bound before execution. This approach enables reusability and efficiency by preparing a statement once and executing it multiple times with different values but the same compiled execution plan.

Two methods were developed to program and execute SQL queries from a procedural language. Today, we might refer to it as a persistence layer API:

- **Stored Procedures:** These are predefined SQL statements packaged into procedures within the database module, which can be deployed alongside database tables. Stored procedures are optimal for encapsulating complex SQL logic and are executed within the database server environment.
- **Embedded SQL** is an extension of a host programming language that requires preprocessing by a precompiler to translate it into stored procedures or low-level database calls. This method was typical in database-centric deployments using limited programming languages but fell out of favor with more modern architectures.

By the late 1980s and early 1990s, the shift towards client-server architectures broadened development possibilities, introducing graphical user interfaces (GUIs) and more flexible runtime environments. This change encouraged developers to select their preferred programming languages, rendering stored procedures proprietary languages and embedded SQL precompilers less practical. Applications increasingly relied on dynamic execution of SQL text sent via lightweight APIs like Open Database Connectivity (ODBC).

ODBC standardized the use of SQL in applications by sending SQL statements as character strings through a limited set of methods to connect, prepare, bind, execute, and fetch. This abstraction enabled rapid development but introduced risks, such as inefficiencies and vulnerabilities like SQL injection when user inputs were mishandled. Hence, prepared statements, which separate SQL syntax from data inputs through placeholders (e.g., question marks), became the best practice. This approach safeguards against injection attacks and enhances performance by parsing and compiling SQL statements only once upon preparation.

This employed dynamic methods designed for user interaction rather than those meant for programming languages, resulting in increased agility. The database client is the application runtime instead of a user, yet the transaction dynamic uses the same chatty protocol with an interactive sequence of statements and responses. ODBC does not present the database as a data persistence service. Instead, it introduces a protocol to facilitate program communication, similar to what a command-line interface would send following the user interactions.

Using SQL as character strings in the code challenges application development and evolution. Such practices restrict compile-time error detection and necessitate synchronization between the application code and any modifications to the database structure. The schema is defined in the database using Data Definition Language (DDL) but utilized in the application with Data Manipulation Language (DML). This process involves different tools and teams. Unlike stored procedures or pre-compilers, no syntax or type is checked before runtime.

Java and the Advent of JDBC

JDBC, introduced in 1997 as part of JDK 1.1, defined how Java applications connect with database systems. It brought a standardized mechanism similar to ODBC, providing Java with a versatile toolset for accessing different kinds of databases using SQL. This innovation was key in abstracting the complexities of database protocols, thereby maintaining Java's principle of cross-platform compatibility in data access.

One of JDBC's fundamental contributions was providing a consistent set of methods to establish connections, execute SQL statements, retrieve results, and manage transactions. This standardization enabled application developers to focus more on logic and less on the underlying protocols of various database systems. Vendors subsequently created specific JDBC drivers, which were crucial to this ecosystem. These drivers varied significantly, ranging from layers on top of ODBC to dedicated Java implementations of the database's client interface. Java was multi-platform, and JDBC introduced a client/server API standard for all databases. However, like many such abstractions, it had to be confined to the lowest common denominator of all databases, resulting in a limited interface for navigation through cursors or mapping data types between the application runtime and the database engine. It maintained the illusion of an SQL standard that allowed for building database-agnostic applications, but all SQL dialects are different, and JDBC transmits SQL as-is.

Java's orientation toward object classes representing business entities conflicted with the relational database model of tables and rows. This paradigm difference, „impedance mismatch,“ underscored the challenges of communicating with the database using dynamic SQL through JDBC. Developers were often required to craft SQL queries as hardcoded strings with placeholders, binding scalar values to execute them, and processing a flat result set where rows are fetched and columns are indexed. This method lacked semantic consistency between the object and relational models and provided no compile-time checking, increasing the likelihood of runtime errors and SQL injection vulnerabilities. The tedious process of mapping two-dimensional result sets to complex object graphs further revealed these limitations. To obtain results from multiple tables in relational databases, a developer had to decide between joining tables into a single two-dimensional table and using additional application code to break it into various application objects or executing several queries, which would increase database round trips and impact performance and scalability.

Database vendors, cautious about preserving established SQL paradigms, emphasized stored procedures and embedded SQL as efficient ways to co-locate logic and data within databases. However, these approaches conflicted with modern development practices that advocate a more apparent separation between business logic and data persistence. Consequently, they did not adapt SQL and RDBMS to keep pace with advancements in Java and object-oriented analysis and programming. They opted to adhere to the two methods outlined in the SQL standard: in-database stored procedures and embedded SQL. They argued that the data logic must be deployed to the database. The impedance mismatch was not only between two technologies using different data types and structures. It also impacted the communication between database administrators, who considered only their traditional practices good practices, and application builders, who viewed the SQL database as a barrier to their development agility.

Efforts like SQLJ, developed in the late 1990s, aimed to seamlessly integrate embedded SQL with Java, like it did with COBOL and C in the past, to improve database interactions within Java applications. SQLJ offered several benefits, including compile-time verification, which helped catch errors early in the development cycle, and performance tuning, which sought to optimize database interactions. However, SQLJ pre-compilers faced challenges in gaining widespread adoption, as they relied heavily on older procedural programming patterns that often conflicted with Java's more modern object-oriented approach and design patterns.

As databases advanced, they increasingly adopted features to handle complex data types, such as Object-Relational Database Management Systems (ORDBMS), XML, and JSON support. ORDBMS systems aimed to incorporate elements of object-oriented programming, such as inheritance, polymorphism, and encapsulation, into traditional relational databases. Oracle and PostgreSQL extended their SQL capabilities to support user-defined types, complex data structures, and custom functions. They allowed the schema to be denormalized by adding new datatypes for nested tables, collections, arrays, or documents.

Despite these advancements, ORDBMS systems struggled with scalability and complexity. Maintaining a database-centric approach often resulted in monolithic systems that were challenging to scale and upgrade without significant downtime. This complexity sometimes conflicted with the evolving requirements of application developers, who needed systems that could adapt to changing business needs. In another effort to reintegrate application code into the database, database vendors such as Oracle added a JVM to the database instance to execute stored procedures written in Java. This further intensified the monolithic nature of the RDBMS and made database upgrades more challenging.

Instead of evolving to better integrate with modern applications, SQL databases maintained a database-centric approach. They became more monolithic and challenging to scale and upgrade. They failed to meet the demands of application developers and became more cumbersome to operate. Software development has advanced with the capability to run CI/CD and scale horizontally. However, this didn't apply to the database, which

became the primary problem for application builders.

The ORM Revolution: Hibernate and JPA

The early 2000s marked a significant transition in how Java applications managed database interactions, driven by the emergence of Object-Relational Mapping (ORM) frameworks. At the forefront of this revolution was Hibernate, which significantly eased the complexities of integrating Java objects with database tables. By automating the translation of JDBC result sets into domain-specific objects, Hibernate reduced the tedious manual coding efforts associated with traditional database operations. It also addressed the long-standing impedance mismatch between object-oriented and relational paradigms.

Hibernate's standout feature was its ability to automatically translate actions performed on Java objects into corresponding database operations through a Hibernate session that synchronizes the state of objects and table rows. This allowed developers to work directly with domain models, as Hibernate tracked object changes and generated SQL statements to persist updates as needed. Its design was flexible enough to accommodate different database capabilities, re-ordering the statements to avoid deadlocks, allowing configurations to adjust prepared statement use or optimize dynamic updates.

Building on the promise of efficient ORM, the Criteria API and Hibernate Query Language (HQL) provide robust query capabilities without the need to write SQL queries. The Criteria API enables developers to construct queries programmatically using Java criteria, offering type safety, flexibility, and reduced syntax errors. Meanwhile, HQL allows developers to write SQL-like queries operating over the Java object model rather than directly on database tables and views. This allows for different fetching strategies without defining another mapping model. Use cases that know in advance all tables and columns to read can do so in one call to the database, avoiding multiple round trips that hinder performance.

For the numerous use cases that do not require further optimization, Hibernate also introduced features like lazy loading, which deferred data loading until it was explicitly accessed. Although this technique could lead to the notorious N+1 problem by causing multiple database roundtrips, it provided developers with flexibility and simplicity in coding scenarios that lacked critical performance requirements.

In addition to Hibernate, alternatives like iBATIS (now MyBatis) were available. These options enabled developers to retain precise control over SQL while facilitating object mapping of result sets. This capability empowered developers to enhance query performance and utilize database-specific features, striking a significant balance. Nonetheless, Hibernate presented a further benefit: it reduced the growing number of SQL statements needed as additional use cases and scenarios evolved within the application lifecycle.

Later, jOOQ (Java Object-Oriented Querying) offered

another approach to using the data model declared in the database from Java. It follows the SQL semantics by generating type-safe SQL queries directly from Java code. jOOQ bridges the gap between ORM and direct SQL, offering a fluent API that provides compile-time verification of SQL syntax and enhanced control over database interactions.

Other programming languages have implemented similar approaches, but solutions have often been driven by Java first. Meanwhile, Hibernate became so popular that a need for a standardized ORM framework emerged. The Java Persistence API (JPA) was introduced, providing a set of standard annotations and interfaces that allowed for ORM integration without a commitment to a specific implementation. While Hibernate quickly became a leading JPA implementation, others like EclipseLink and OpenJPA provided developers with diverse choices.

Data Modeling in ORM

With the Java Persistence API (JPA), the paradigm has fundamentally shifted from database-centric to application-first. Traditionally, the database-centric or bottom-up approach concentrated on defining the data model within the database catalog, where applications would query directly using SQL commands on catalog views or cursor results. This method positioned the database at the core of schema management and application interaction. It represented a continuation of the conventional data modeling approach for monolithic information systems: the data model (Entity-Relationship Diagram) was created first, static, modeling the business entities before analyzing the dynamic access patterns of the application use cases.

However, with the advent of JPA, the location of schema definition shifted to the application itself. Developers utilize Java objects annotated with JPA-specific metadata as the authoritative source for schema generation. This ensures that the Java classes define the database schema, enhancing synchronization between application logic and data structures. With this approach, validation can occur during the application code compilation and deployment of the database objects. There is no risk that a change in the database will break the application at runtime.

The database must still understand the data it stores to validate data integrity and optimize data access. However, the Data Definition Language (DDL) is generated from the application using a top-down, or JPA-first, approach. The previously used bottom-up, or database-first, approach is now rarely employed, typically only to merge with the JPA-generated DDL to create migration scripts. You can read more about it in an article by Andrey Belyaev: <https://jpa-buddy.com/blog/db-first-vs-jpa-first/>

Data modeling is essential when using ORM frameworks. Developers must carefully design their entity classes to accurately map the underlying database schema while taking advantage of ORM features. Implementing JPA annotations for a relational database involves establishing one-to-many, many-to-one, and many-to-many relationships and applying relational model concepts to linked or embedded objects. It also requires evaluating access

patterns and deciding which data will be retrieved together through eager or lazy loading. There are two types of developers. One group uses ORM as an abstraction of the database and does not consider what is generated. This approach works for applications without critical performance requirements, as the generated queries may not be optimal. This allows rapid application development with teams that do not need to be experts in each component used in the stack. The other group logs the generated SQL queries and analyzes their execution plans to ensure performance scalability. Those in this latter group may wonder why they must do the work twice: design and query an object model while validating what is executed on a relational model. Once the data is fully described in a model with linked or embedded objects, why not use a database that can store and retrieve it using the same model?

The Rise of NoSQL and Java's Adaptation

While ORM frameworks provide many benefits, they also have notable drawbacks. When avoiding complications from native SQL or employing one mapping per use case, mapping Java objects to relational tables can lead to significant performance issues, especially when managing object relationships (e.g., one-to-many or many-to-one) in large applications with complex object graphs. A common concern is the N+1 query problem, where retrieving related entities results in excess database queries, severely affecting application performance. To counter this inefficiency, developers often need to implement optimization strategies such as batch fetching, join fetching, and caching techniques. They must avoid that one transaction involves multiple interactions with the database, which incur network latency, context switches, and memory to keep the state of the transactions. They should adopt a document-centric approach when designing their code, ensuring they query related objects in one SQL or JPQL statement to fetch them together instead of navigating on a per-object or per-table basis and maintaining a transaction between them.

A database-centric approach also presents scalability, flexibility, and adaptability limitations, primarily due to the central catalog containing metadata about the application schema. Although SQL and relational databases allow for agile schema modifications using data definition language (DDL), and many support transactional or online DDL, this agility diminishes when synchronization with the application is required. Adapting to rapidly changing application needs can become challenging. Some applications had to adopt an Entity-Attribute-Value model to avoid any schema on the database. However, this is disastrous for performance as the database has no information to optimize the access patterns. Furthermore, as web applications grow to accommodate millions of users, relational databases often struggle with horizontal scalability, necessitating complex sharding techniques to distribute data across multiple nodes. While the problem has been solved for data with distributed SQL databases, the metadata, holding the SQL catalog or dictionary, must still be shared and be a single point of truth.

To resolve these issues, NoSQL databases have emerged, offering alternative data storage models tailored for applications where the schema is application-defined. In contrast to SQL databases, they retain minimal information, such as index definitions and schema validations, while storing most application data in documents defined by the application. Some NoSQL databases, like key-value stores, only need to know the sharding key, delegating all data access optimizations to the application that explicitly defines and queries the indexes, which is opposite to

the data independence principle of relational databases. Other document databases, such as MongoDB, retain more information for transparent index usage, supported by a query planner optimizer. The significant advantage over Object-Relational Mappers lies in the absence of mapping and transformation from objects to multiple tables and matching the transaction bounds to one document.

With an API closer to their objects, Java developers are not required to understand the relational version of the application schema nor examine execution plans, index definitions, or join method algorithms to grasp the complexities behind their query access patterns. A Java object embedding another object is stored as a single document, with subdocuments, all colocated physically and fetched together without requiring multiple random I/O operations.

Additionally, the database uses the same syntax and semantics as the application. For instance, handling null values in Java differs significantly from SQL, which applies tree-valued logic to nullable datatypes. In summary, Java developers can utilize the database similarly to an ORM but without the complications associated with mapping to a different storage paradigm regarding persistence.

As document databases became popular for adding a persistence layer for business objects defined in Java, Java adapted and proposed ORM for NoSQL databases, like Hibernate OGM. However, ORMs were made for relational databases and made little sense with a document database. The main reason was to unify the definition of persistence, like with JPA, but another framework became popular to simplify interactions with all databases.

Spring Data: Simplifying Data Access in Java

Spring Data, introduced in 2011, is an integral part of the Spring Framework that addresses data access complexities in Java applications. It provides a unified programming model for interacting with various data sources, including relational databases (via JPA, JDBC) and NoSQL databases such as MongoDB, Cassandra, and Redis. By abstracting the details of database operations, Spring Data allows developers to concentrate on business logic rather than the intricacies of transaction persistence mechanics.

The launch of Spring Data JPA revolutionized ORM persistence by automatically generating repository implementations. This automation significantly reduces boilerplate code and simplifies CRUD operations, thus improving the development speed and maintainability of simple use cases. Over time, Spring Data has expanded to support various NoSQL databases, permitting flexible data modeling across document, key-value, and graph database paradigms.

Key features like pagination, sorting, customizable query methods, and integration with QueryDSL enhance the framework's ability to manage data retrieval and manipulation efficiently. It implements the best design patterns to avoid inefficient interactions with the database. The 2017 introduction of Spring Data Reactive marked a transition to non-blocking, event-driven architectures, addressing the requirements of modern microservices and

cloud-native applications. By promoting scalable application development and aligning with Java's evolution, Spring Data is crucial in bridging the gap between relational and NoSQL data storage and supports best practices in domain-driven design within contemporary application architectures.

Conclusion

Over the years, Java has dramatically improved the way applications work with databases, shifting from focusing mainly on databases to being more about the applications themselves. At first, Java programs relied heavily on SQL commands directly in their code. The arrival of ORM frameworks like Hibernate and JPA has changed how applications use databases. These frameworks have helped reduce the problems that arise from the differences between relational databases and object-oriented Java programs. They allow developers to work directly with the structure of objects related to their applications and their business logic. One of Hibernate's key advantages is its ability to optimize database interactions by reducing the number of queries to write, especially for CRUD operations, through caching, lazy loading, and batching.

JDBC drivers have failed to provide a convenient approach, which gave rise to ORMs, but Java has also evolved for developers who prefer better control and write native SQL statements. They can use JDBC templates, introduced in Spring Framework 1.0 (2004), and Spring's JdbcTemplate, introduced in Spring Framework 6.1 (2023), to map their results to Java objects. This approach fits between JDBC and ORM. They can also fetch the query result to Java records, standardized in Java 16 in 2021, instead of objects, similar to the rows from tables and views.

As the need for more complex data handling grew, especially regarding scalability and flexibility, NoSQL databases emerged to address the limitations of traditional database systems and their monolithic schema definition. Document-oriented databases, in particular, better matched application-focused structures for Java developers, making it easier to manage data without complicated transactions. Java's strong ecosystem has continuously adapted to these changes. Frameworks like Spring Data provide a unified way to access data, allowing the smooth integration of SQL and NoSQL systems within one programming style. These changes have helped developers focus more on business rules with greater efficiency while maintaining the ability to change and grow their systems with modern technology.

Java's important role in this change highlights its ability to adapt and stay in sync with the fast-changing tech world. It effectively links application logic and various types of data storage. As a result, Java has significantly influenced the industry's move towards more application-centered ways of interacting with data, addressing the growing complexity and variety of modern software development demands. From a database perspective, the evolution of Java over the past 30 years parallels the transition from database-centric monolithic applications through client-server models and finally to more flexible software design and deployment where the data schema is integrated within the application services.

A Look Back at Java's 30-Year Journey with Databases



When Java 1.0 launched in 1996, both the language and the internet were very different from today. At the time, the web was mostly static, and Java's promise to "caffeinate" the web by supercharging it with interactive games and animations was promising. According to David Banks, who wrote to Wired in 1995, Java was the hottest thing since Netscape.

To put it in a better perspective, Java's origins date back to 1990 when Sun Microsystems struggled to find its place in the consumer market. At the time, they developed Oak, a new programming language designed to be platform-independent, lightweight, and ideal for networked environments. Their first attempt was to commercialize Oak for interactive televisions.

By 1994, the Web started popularizing and Sun recognized Oak as its perfect application. Unlike traditional software, Oak allowed applets (small, portable programs) to run on any machine with an interpreter, making it ideal for web applications. Oak was rebranded as Java, its interpreter was named Java Virtual Machine, and a prototype browser, HotJava, was created to demonstrate its capabilities.

However, Java 1.0 lacked standardized database connectivity. The web was still in its infancy, and databases were largely the domain of enterprise applications rather than web-based software. As a result, Java's initial design focused on creating dynamic applications that could run seamlessly across different platforms, rather than integrating with backend data storage systems. This made sense given the landscape at the time - the database world was dominated by heavyweight relational database systems like Oracle 7, IBM DB2, Sybase SQL Server, and Informix. These systems were designed for traditional client-server architectures and enterprise workloads, with sophisticated transaction processing capabilities and complex query optimizers. While object-oriented databases like ObjectStore and GemStone existed, they remained niche products, primarily used in specialized engineering and telecommunications applications. The concept of „web-scale“ databases was still years away.

Java's portability and architecture-neutral design made it appealing for programmers and businesses. Eli Javier, who worked for IBM, wrote to Toronto's User Group Magazine back in 1997 that programmers loved Java for being the first true web programming language and that businesses loved it because it could help them reduce costs by simplifying the implementation and distribution of applications since it could run anywhere and could be deployed from centralized servers.

Authors:

Raphael De Lio is a passionate software engineer who loves to think about solutions and ways to improve anything he touches. With over seven years of experience across multiple roles, industries, and countries, Raphael brings a rich perspective to solving technical challenges and connecting with developer communities.



Brian Sam-Bodden is a Senior Applied AI Engineer at Redis as well as an author, instructor, speaker, developer advocate and open source contributor and Java Champion who has spent over twenty years crafting software systems. He holds dual bachelor's degrees from Ohio Wesleyan University in computer science and physics. Brian is a frequent speaker at user groups and conferences nationally and abroad and is the author of "Beginning POJOs: Spring, Hibernate, JBoss and Tapestry", co-author of the "Enterprise Java Development on a Budget: Leveraging Java Open Source Technologies" and a contributor to O'Reilly's "97 Things Every Project Manager Should Know".



Even IBM recognized Java as the foundation for network computing in 1996, shortly after Java 1.0 was released. This shift was publicly reinforced in 1997 when Bill Zeitler, IBM's General Manager for the AS/400 Division, explicitly stated that Java was the key enabler of the AS/400 business on network computing.

Despite its rapid adoption, Java needed a unified way to interact with databases. Without a standard, developers had to write database-specific code, threatening Java's portability. In response, Sun Microsystems introduced the Java Database Connectivity API (JDBC) just three months after Java 1.0's release. Inspired by Microsoft's ODBC, JDBC allowed applications to interact with databases using a single interface. By mid-1996, JDBC 1.0 was finalized, marking a major step toward enterprise computing.

With JDBC, Java became more than just a language for applets and interactive web applications, it was now a viable option for enterprise software development. By the late 1990s, major database vendors like Oracle, IBM, Sybase, SAS, and Borland had fully embraced Java and provided official JDBC drivers for their databases.

While JDBC was a breakthrough, it wasn't a perfect solution—as Java applications grew in complexity, developers started to hit limitations. Developers had to write a lot of repetitive code just to execute simple queries. Every database operation required manually managing connections, statements, and result sets. At the same time, exception handling was verbose and cumbersome, requiring try-catch-finally blocks for proper cleanup.

The perils of the Object-Relational impedance mismatch started rearing its ugly head. Database Tables and Objects are two very different mediums for our data. While relational databases focused on data normalization and integrity through relationships and constraints, object-oriented programming emphasized encapsulation, inheritance, and polymorphism. This fundamental disconnect meant developers had to constantly translate between these two paradigms. Objects with rich hierarchies and complex relationships had to be flattened into tables, while normalized database schemas had to be reassembled into interconnected object graphs. Simple concepts in one world became complicated in the other - a basic inheritance hierarchy in Java might require multiple joined tables in the database, while a straightforward database view might need multiple interrelated classes to represent it properly in the object world. The impedance mismatch not only made development more complex but also impacted performance, as each translation between paradigms added overhead to database operations.

To address these challenges, Java 2 Enterprise Edition (J2EE) introduced Enterprise JavaBeans (EJB) in 1998. As Cliff Berg wrote to Info World in 1999, the main goal of EJB was to simplify enterprise application development to allow software developers to focus on business logic rather than building custom infrastructure. EJB simplified enterprise development with built-in transaction management, security, and scalability.

The model distinguished between different types of beans: session beans, which managed business logic and user interactions, and entity beans, which represented persistent data. A major innovation in EJB was container-managed persistence (CMP), which allowed database interactions to be handled automatically by the application server, rather than requiring developers to write explicit SQL queries.

With CMP, developers could define entity beans that represented database records, and the EJB container would automatically handle the persistence, updates, and retrieval of these objects. This not only reduced the amount of boilerplate code but also improved database portability, as applications no longer had to be tightly coupled to a specific SQL dialect or database vendor.

EJB also introduced declarative transaction management, which was a significant improvement over traditional database transaction handling. Instead of requiring developers to manually manage transactions (such as calling `commit()` and `rollback()` within their code), EJB allowed transactions to be configured declaratively through deployment descriptors.

Another database-related challenge that EJB aimed to solve

was connection pooling. Establishing a database connection is a costly operation, and opening a new connection for every request led to severe performance bottlenecks.

But despite its advancements, EJB's database management approach was not without issues. The entity bean model was often criticized for performance inefficiencies, particularly in how it handled large-scale persistence. CMP, while convenient, introduced significant overhead, as the container had to generate SQL dynamically and manage database operations behind the scenes. Developers sometimes found that bean-managed persistence (BMP)—where database operations were explicitly coded—was necessary for performance tuning.

Frustrated by EJB's complexity, Gavin King developed Hibernate in 2001, an Object-Relational Mapping (ORM) framework. His goal with Hibernate was to create an open-source Object-Relational Mapping (ORM) framework that would provide transparent persistence, allowing developers to work with plain Java objects (POJOs) instead of managing manual SQL queries and database transactions.

Unlike JDBC, where database queries were embedded as strings in code, Hibernate introduced a declarative mapping system that allowed developers to define database relationships in XML configuration files. This approach separated database logic from application code, making it easier to modify schemas without breaking application logic.

One of Hibernate's most significant contributions, as written by Mario Aquino back in 2003, was how it enabled one-to-one, one-to-many, and many-to-one relationships through mapping configurations, removing the need for developers to manually handle joins and foreign keys in their code. Additionally, Hibernate supported lazy loading, cascading deletes, and outer join fetching, allowing applications to optimize database interactions by reducing the number of roundtrips to the database.

Another major advantage of Hibernate was code generation and automation. Hibernate could generate Java source files based on object-relational mapping definitions. This eliminated much of the manual coding required in traditional JDBC applications. Tools like Middlegen, which is integrated with Hibernate, could even analyze database schemas and automatically generate Hibernate mapping files. In contrast, EJB entity beans required manual deployment descriptors and verbose XML configurations, making them difficult to maintain and deploy.

To further simplify application architecture, Hibernate also introduced a lightweight and flexible transaction model. It provided a built-in transaction management, allowing developers to persist, update, and delete objects without directly managing JDBC transactions. This was in stark contrast to EJB's container-managed transactions, which forced developers to work within the constraints of the Java EE server. With Hibernate, transactions could be handled programmatically or declaratively, offering more control and flexibility.

By 2003, Hibernate had already become a preferred alternative to EJB entity beans for data persistence. And its success influenced Java's official ORM standard.

In 2006, recognizing the success of ORM frameworks like Hibernate, the Java Community Process (JCP) formed JSR 220, which introduced the Java Persistence API as part of EJB 3.0, which in turn, was part of Java EE 5.

JPA standardized ORM across different implementations like Hibernate, TopLink, and JDO. It replaced verbose XML configurations with annotations, introduced the Java Persistence Query Language (JPQL), and allowed developers to use lightweight POJOs instead of heavyweight entity beans. JPA quickly made EJB entity beans obsolete. Annotations replaced the need for XML configuration, making persistence mapping much cleaner. EJB 2.1 required lengthy XML descriptors to define entity mappings, but with JPA, simple annotations like `@Entity`, `@Id`, and `@OneToMany` made defining persistence rules much easier. Additionally, JPA introduced the Java Persistence Query Language (JPQL), an object-oriented query language that improved upon traditional SQL and the limited EJB QL.

Unlike EJB entity beans, which required a Java EE container, JPA could be used in standalone Java SE applications. This flexibility allowed developers to use JPA in both enterprise applications and smaller Java projects without the overhead of an application server. Additionally, JPA provided automatic transaction management, freeing developers from manually handling commit and rollback operations.

Another key advantage of JPA was its vendor independence. The API defined a standard persistence model, but developers could choose any JPA provider, such as Hibernate, EclipseLink, or OpenJPA. This decoupling allowed for more flexibility and reduced dependency on specific ORM implementations.

Although Hibernate was already widely used, JPA's standardization brought consistency to enterprise applications. Many projects continued using Hibernate as a JPA provider, but they could now rely on a common API instead of being locked into a single ORM framework. Many of Hibernate's innovations made their way into the JPA standard: its transparent lazy loading through proxy objects, its solution for handling entity lifecycle states (transient, persistent, and detached), the concept of persistence contexts for tracking entity changes, and its sophisticated caching mechanisms.

Even JPA's query language (JPQL) was heavily influenced by Hibernate's HQL (Hibernate Query Language), adopting its object-oriented approach to querying. Other key Hibernate features that shaped JPA included its cascading operations for managing entity relationships, its optimistic locking strategy for concurrency control, and its session-per-request pattern which evolved into JPA's `EntityManager` concept.

With JPA's introduction in 2006, EJB entity beans quickly became obsolete. Developers now had access to a simpler, more intuitive persistence model without the heavyweight nature of EJB. JPA

also accelerated the adoption of Java EE 5, making enterprise development more accessible.

JPA had simplified database access, but managing repositories and writing queries was still tedious. Developers are still required to write boilerplate for repository management. Fetching, updating, and deleting records required defining custom queries or writing extensive code.

Besides that, by the late 2000s, the rise of NoSQL databases was reshaping the data storage landscape. While relational databases had been the dominant choice for decades, the explosion of web applications, big data, and real-time processing led to new requirements that traditional relational databases struggled to meet. NoSQL databases like MongoDB, Cassandra, Redis, and Neo4j gained popularity because they offered better scalability, flexibility, and performance for specific use cases. Unlike relational databases, which enforced strict schemas, NoSQL databases allowed for more dynamic data models, making them ideal for cloud-native applications and distributed systems.

Recognizing the need for a simpler and more unified way to interact with both relational and NoSQL databases, Spring Data was introduced in 2010 as part of the Spring ecosystem. Its goal was to provide a consistent, high-level abstraction for data access, eliminating the need for repetitive boilerplate code and allowing developers to focus on business logic rather than infrastructure. Instead of manually defining queries or implementing repository patterns, Spring Data introduced the concept of repositories that automatically generated queries based on method names, reducing the amount of code developers had to write.

The first module released under the Spring Data umbrella was Spring Data Neo4j in April 2011 and just a few months later, in July 2011, Spring Data JPA 1.0 was released.

While JPA had already simplified ORM, Spring Data JPA took it a step further by introducing automatic query generation, pagination, and declarative transaction management. Developers no longer had to write SQL or JPQL for simple queries—Spring Data JPA could derive queries from method names, making database access more intuitive and reducing the need for boilerplate code.

In October 2011, Spring Data MongoDB 1.0 was released. By this time, MongoDB had become the dominant NoSQL document database, providing developers with a flexible, schema-less data model. Unlike relational databases, where changes to schemas required complex migrations, MongoDB allowed applications to evolve dynamically.

The next major milestone came in May 2012, when Spring Data Redis 1.0 was released. Unlike traditional databases, Redis is an in-memory first key-value database, known for its exceptional speed and versatility. Initially developed in 2009, Redis quickly became a critical component in modern architectures, powering caching layers, real-time analytics, session storage, and even message brokering systems.

Along the upcoming years, new Spring Data integrations continue to be released and the landscape continues to evolve rapidly with the rise of multi-model databases, serverless databases, and edge computing. This leads us to ask: Where will Java go next when dealing with the ever-changing world of data?

The future isn't about choosing between SQL or NoSQL, it's about choosing the right data model for the right job. This sentiment is reflected in modern databases like CockroachDB and FaunaDB that blur the lines between traditional categories, offering both SQL interfaces and distributed NoSQL-like scalability.

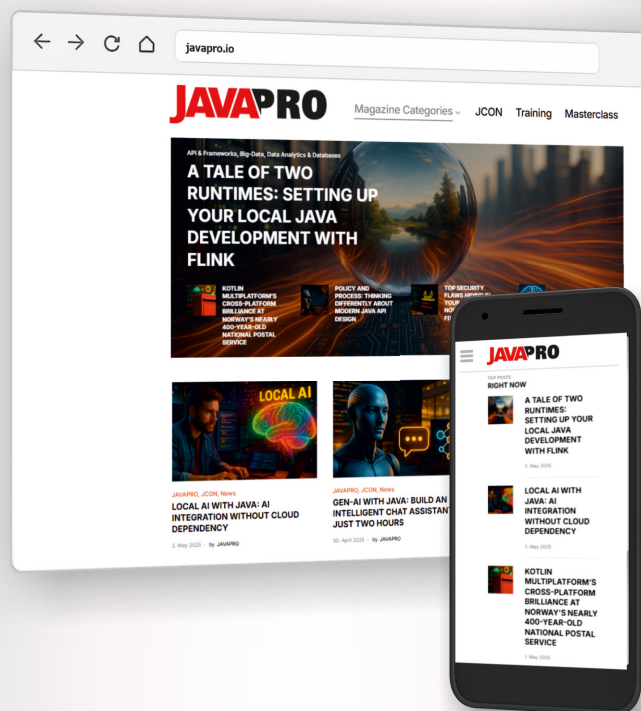
The evolution of Java data access libraries seems to be following suit. Project Loom's virtual threads promise to revolutionize how Java handles database connections, potentially making reactive programming patterns optional rather than necessary for high-throughput applications. We can expect that data access on the Java platform will evolve to provide a consistent experience across different data models while leveraging Java's new concurrency features.

The rise of AI and machine learning is also pushing Java to adapt. The new kid on the data block is the Vector Database. Although Vector Databases have been around for a while, the newly gained ability to find an AI model to create vector representations for any unstructured data has elevated the need for databases to effectively handle vectors and vector operations (such as semantic searches via KNN/ANN).

We're seeing a growing need for Java applications to seamlessly interact with vector databases and large language models while maintaining the type safety and reliability that Java developers expect. Will we see new standards emerge for these use cases, similar to how JPA standardized ORM? How will Java evolve to handle the increasing demand for real-time data processing at the edge? As serverless databases gain popularity, how will Java's traditional connection pooling and transaction management patterns adapt to these new paradigms? This progression is already visible in projects like Spring AI.

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !



#JAVAPRO #CLOUD

Platform Engineering 101 for Java Developers



Combating the struggles of modern cloud-native Java developers

Moving our applications to the cloud and cloud-native infrastructure can offer many benefits. These can include flexibility, reliability, robustness, access to innovative technologies and current trends, speed, cost efficiencies and more. As developers, it is an exciting environment to work in. However... despite the excitement to move to the cloud, actually completing this move and effectively creating cloud-native applications can be extremely challenging, frustrating, and difficult! Why is this, and what can we do about it? In this article, we'll explore the tools and technologies that can help us combat the struggles of cloud-native Java developers. We'll delve into Platform Engineering, Internal Developer Platforms and Portals, and tools like Backstage and Red Hat Developer Hub.

Struggles of a cloud-native developer

There are many challenges that developers face when creating effective modern cloud-native applications.

Challenge 1: Increasing cognitive load

Increasing responsibilities and tools has led to what The State of Platform Engineering Report describes as "increased cognitive load on engineers." As developers, we now have to focus on more than just app development. In fact, there is what seems to

Authors:

Grace Jansen is a Java Champion and Developer Advocate at IBM, working with Open Liberty, MicroProfile and Cloud Technologies. She has been with IBM since graduating from Exeter University with a Degree in Biology. Grace enjoys bringing a varied perspective to her projects and using her knowledge of biological systems to simplify complex software patterns and architectures. As a developer advocate, Grace builds POC's, demos and sample applications, and writes guides and tutorials.



Maarten Vandepierre is an experienced software professional and AppDev & AI Specialized Solutions Architect at Red Hat, focusing on application development and AI with OpenShift. Passionate about clean architecture, he emphasizes modular, maintainable, and scalable software design while aligning infrastructure solutions to support these principles. With deep expertise in integration technologies like API Gateways, Keycloak, Kafka, service mesh, and Camel, he ensures seamless system connectivity.



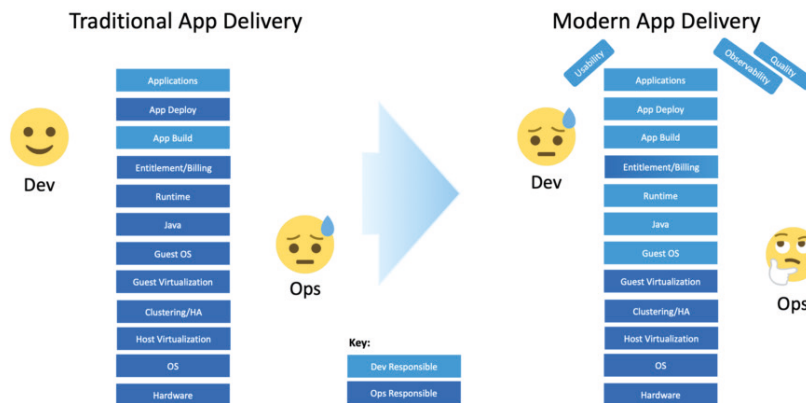
be an ever-expanding list of accumulated-word-descriptions to capture the growing responsibilities developers now have. What started as Dev and Ops and became DevOps has now expanded to FinOps, MLOps, GitOps, DataOps, DevSecOps, BizDevOps, BizDevSecOps, BizDevSecFinMLOps.... The list goes on. As developers, we now face a world of increasing complexity where we are taking on even more tasks and responsibilities.

responsibilities that were more traditionally for ops teams and roles, they are faced with even more challenges associated with effectively managing their distributed applications in the cloud.

Utilizing cloud native infrastructure offers the opportunity to take advantage of increased portability and flexibility for our applications. It enables us to develop applications that utilize containers and can be deployed to any cloud platform, be it

public, private or hybrid cloud, or even on the edge, with an abundance of different tools and technologies in servers available across the world. However, as well as offering portability, this cloud-native infrastructure introduces yet another barrage of key decisions - including making decisions on which tools to use, how to configure environments, how best to build applications within a container, and how to ensure consistency and best practices for containers.

Organizational Impact & Cognitive Load



Challenge 2: Tools – too much choice

A significant contributing factor to the increasing cognitive load we face is the ever-increasing number of complex tools we now have to know and use. We only have to look at the CNCF landscape, or as I affectionately call it, the cloud-native grocery store, to see the vast array of different tools and technologies available for developers to use for each stage of the software delivery lifecycle (SDLC).

From choosing the right database, streaming or messaging tools, to what tools to use for building application images, CI/CD stacks, scheduling and orchestration, service proxies, API gateways, container registries, observability and analysis, and more, how do developers choose the best tools for their specific needs and requirements when there's so much choice? Even knowing where to get started seems impossible looking at this landscape!

Challenge 3: Steep learning curve

Even once the tools have been selected, it's often a very steep learning curve to get comfortable with these technologies or master them. And, with this ever-changing landscape and the rate of innovation within these tools, this learning curve never seems to stop growing!

Not only this, but developers are often "expected" to be experts from day 1, getting limited or no time for training on the job, which makes this challenge even more difficult.

Challenge 4: Difficult to manage

As developer's role starts to include more and more tasks and

As well as making these many critical decisions, developers now also have to contend with "bookmark hell" to be able to manage these build processes, running containers, production images, etc... Developers now have to know how to reach a dozen CI pipelines, countless CD tools, applications, ticketing systems, documentation pages on different source systems (often not findable if you do not bookmark them), have guidelines per application, technology and more!

This is all compounding the issues of Challenges 1, 2 and 3, putting further strain on development teams!

Challenge 5: Security and compliance

Arguably one of the most significant challenges for any application and especially those deployed in distributed cloud-native infrastructure is security and regulatory compliance. We only need look to recent examples (e.g. Log4Shell/Log4J) to grasp the significant risks to our applications when security and compliance is not carefully considered.

Moving towards greater use of open-source libraries, like Log4Shell, means that we need to ensure we are effectively vetting any open source dependencies our applications use and storing an accurate and up-to-date list of these dependencies (like an SBOM – Software Bill Of Materials), so that if a vulnerability is discovered we can act immediately if we know our application is affected. Use of open-source libraries and standards or even adopting architecture styles like hexagonal architecture or clean architecture designs, also enable us to keep our options open if we need to swap out any existing dependencies or elements of our applications

with minimal code disruption or effort from developers.

Developers now need to ensure that security is a part of every aspect of the SDLC and is baked into their practices when developing the applications (i.e. eliminating credentials in code, ensuring proper authentication and authorisation is enabled, making use of role-based access control principles, etc). Couple these security concerns with the growing list of compliance regulations (e.g. data privacy laws and regulations, data storage regulations, etc) and developers have even more to keep in mind!

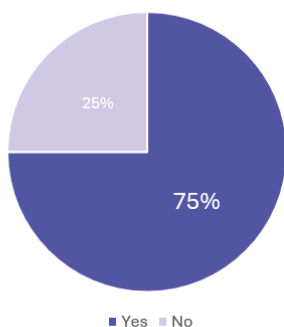
The result of these challenges

Developers are being asked to do more and more, with a confusing landscape of tools and libraries to select from, while contending with the risks and dangers posed by security and compliance.

Put simply, developers are burnt out!

And this isn't just us claiming this; recent studies back up these findings... For example, JetBrains' 2023 State of the Developer Ecosystem report showed that nearly three-quarters (73%) of developers have experienced burnout at some point in their career. Similar research from Haystack Analytics in 2021 found that 83% of developers suffered from work-related burnout.

HAVE YOU EVER EXPERIENCED BURNOUT?



When it comes to explaining the cause of this burnout, many of these studies cite the challenges we've detailed in this article. For example, Salesforce's Mulesoft Study in April 2022 showed that the top three causes of developer burnout were: Increasing workload/demand from other teams, pressure of digital transformation, and requirement to learn new skills. Harness and Haystack's research also supported the role of heightened workloads in developers' burnout.

But what can we do about it? How can we help relieve developers of some of the pressures they are under and reduce developer burnout? This is where platform engineering and the use of Internal Developer Platforms (IDPs) can help...

What is Platform Engineering?

According to platformengineering.org:

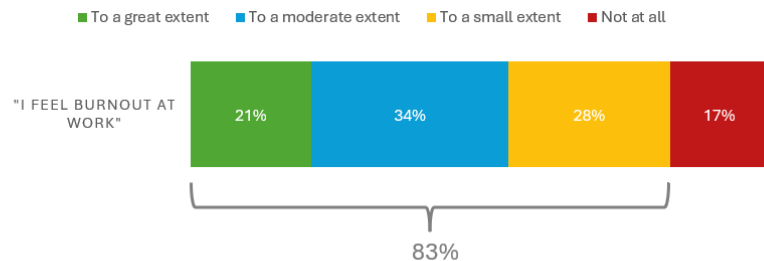
"Platform engineering is the discipline of designing and building toolchains and workflows that enable self-service capabilities for software engineering organizations in the cloud-native era. Platform engineers provide an integrated product most often referred to as an "Internal Developer Platform" covering the operational necessities of the entire lifecycle of an application."

It is an emerging discipline focused on enhancing developer productivity. Platform Engineering aims to reduce complexity and overheads while enabling frictionless, self-service capabilities for organizations trying to develop cloud-native applications. It involves designing and creating toolchains, workflows and automated infrastructure operations that

help to address some of developers' biggest challenges: e.g., reducing the burden of managing multiple tools, technologies and infrastructure and aligning development practices with business priorities and compliance requirements.

You may be wondering, but how does this fit in with DevOps? Does this now mean that DevOps is dead and that platform engineering is the future? Well, not quite. Many in the community believe that platform engineering doesn't necessarily supplant DevOps, rather, they view it as the next evolution of DevOps. In fact, Daniel Bryant, a platform engineering advocate from

TO WHAT EXTENT, IF AT ALL, DOES THE FOLLOWING STATEMENT APPLY TO YOU?



Syntasso, has highlighted that platform engineering is positively grounded in the three ways of DevOps (as described in The DevOps Handbook):

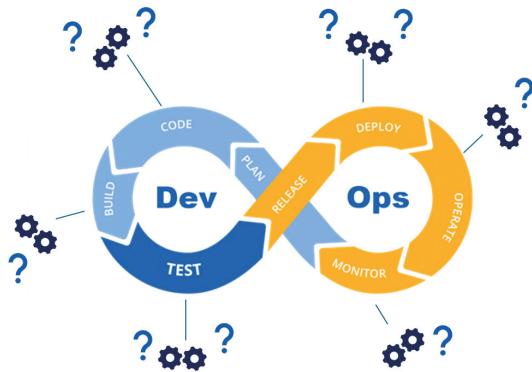
- Concentrating on flow and systems thinking to deliver end-to-end business value
- Even shorter feedback loops
- Continuous experimentation, learning and improvement

How can we use platform engineering?

As the definition provided above suggests, platform engineering is less of a tangible, physical thing and more of a discipline that is enabled by a platform engineering team. Platform engineering teams can implement the principles of this discipline by designing and creating Internal Developer Platforms to provide approved golden work paths (curated

sets of tools and workflows) for developers and Internal Developer Portals for developers to access these golden paths.

It's a bit like creating a recipe book and working out the best ingredients for your organization to use... like baking your own platform. When it comes to selecting ingredients, the platform engineering team would need to consider the majority of use cases for each stage in the SDLC and map out which tools would be best for their organization for each of these stages. This can include making choices about which IDEs to use, version control systems, resources, container registries, observability tools, vulnerability scanners, CI/CD tools, GitOps tools, etc.



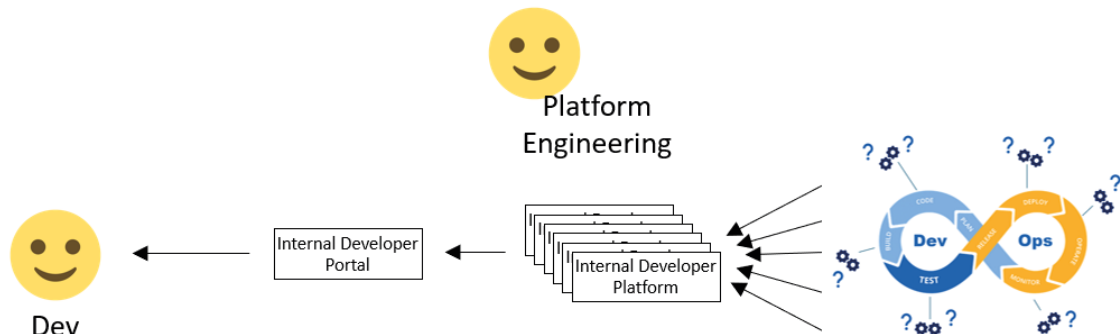
Once this list of ingredients has been selected, the recipes can be created as golden paths. A golden (or paved) path consists of a defined set of tools and workflows for developers to pick up and use through an Internal Developer Platform and Portal.

IDPs integrate into existing workflows which usually remain a git-push deployment workflow but add further automation. The entire deployment process is now at the disposal of the developer. They can request resources, spin up fully provisioned environments, rollback, deploy and set deployment automation ruling autonomously. Essentially, a modern developer uses an IDE to code, git to merge, and an IDP to ship.

What is an Internal Developer Portal?

An Internal Developer Portal provides a single interface to your developer platform – it serves as the „face“ or UI of the Internal Developer Platform. It allows developers to perform self-service operations and actions and offers visibility into the infrastructure. Depending on the maturity of the IDP, it can provide several interfaces and access points for developers to make use of. That can be a CLI, different kind of User Interfaces (UIs), or a developer portal with a service catalog to unify the developer experience.

So, it's important to understand and acknowledge the difference between these two acronyms. Internal developer portals serve as the interface through which developers can discover and access internal developer platform capabilities. An IDP on the other hand, is the sum of all tech, tools and processes that a platform engineering team binds into a golden path for developers. Golden paths reduce cognitive load and drive standardization by design. So, an IDP doesn't necessarily need a developer portal to deliver value to developers. However, using the two together effectively can help to deliver one cohesive platform and UI to ensure ease of use for developers.



What is an Internal Developer Platform (IDP)?

An Internal Developer Platform (IDP) is a central collection of tools, services and automated workflows that enable development teams to be able to rapidly deploy and deliver software. These platforms provide a service layer that abstracts away the complexities of infrastructure management and application configuration.

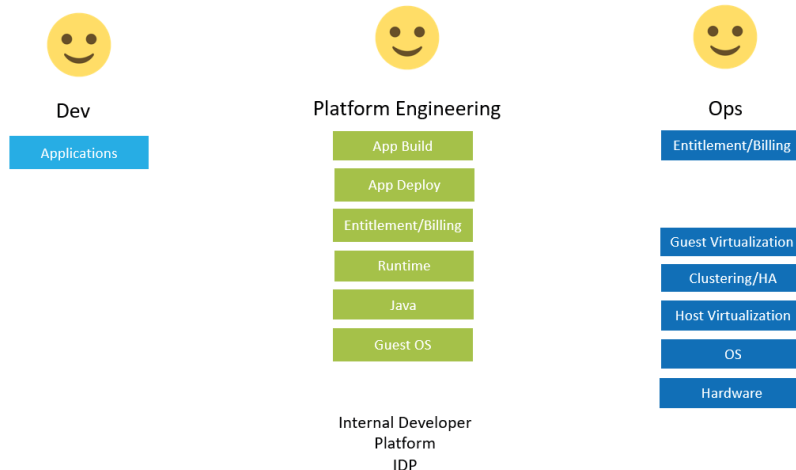
Internal Developer Platforms (or IDPs for short), as the name suggests, are platforms that are designed for internal use by developers (their primary users) within an organization. IDPs are custom-built by a platform team by combining different tools and technologies (services, open-source, self-developed, etc.), those ingredients we mentioned earlier, with the intent to increase developer productivity and velocity.

Benefits for Developers

Qualitative Benefits

So, why use platform engineering and IDPs? What benefits can it bring to developers?

From an organisational perspective, utilising IDPs and having a dedicated Platform Engineering team can help to spread out the responsibilities and concerns of cloud-native application development in a healthier manner across development, ops, and platform engineering teams, without (importantly) compromising on security, compliance, or best practices. Allowing each of these teams to focus on where their strengths lie and having clear distinction between responsibilities. It's important to note here that this platform engineering team does not necessarily have to be new hires, but can instead be a reshuffle of the current organisation to help separate out these responsibilities clearly.



Reference architectures

There are reference architectures available from companies like Humanitec and McKinsey. These companies researched hundreds of real-life platform setups, uncovering common patterns in IDP architectures. They then distilled these patterns into five architectural planes: Developer control plane, integration and delivery, monitoring and logging,

Other qualitative benefits gained include:

- Standardizing and securing key delivery processes (Golden Path), reducing the risk of shadow IT and the burden of operational complexity placed on developers.
- Enabling Internal Developer Platforms, resulting in self-sustainable and self-responsible teams.
- Simplicity of DevX, reducing cognitive load and complexity for development teams and enabling greater productivity.

Quantitative Benefits

External studies showing measured quantitative benefits further supports the actual impact of these qualitative benefits. For example, a recent Forrester Opportunity Snapshot, showed that 74% of DevOps leaders saw that improving their developer experience with an IDP boosted developer productivity, 77% reported that it can shorten time to market, and 85% that it drives revenue growth. Studies by Humanitec also showed that developers were able to deploy 4x more frequently with 0 minutes of waiting time, platform engineers had a 95% reduction in config files and 75% less maintenance overheads, and organizations saw a 30% increase in time to market.

However, that's not to say that platform engineering and the creation of IDPs is something that every company should be doing. As with all methodologies, guidelines, architecture types, etc. there are both pros and cons to utilizing these principles and tools and you should carefully consider these for your own organization and applications to determine if it would be suitable. If, for example, you are part of a small organization and it would not be feasible to create a dedicated platform engineering team, then this perhaps may not be something you would want to implement.

Tools to help

Thankfully, there is already a range of tools, architectures, libraries and references you can use to start creating your own IDP and introduce platform engineering into your own organization.

security plane, and the resource plane. Starting from a blueprint like this could be a big help to rapidly construct IDPs.

IDP tooling landscape

PlatformEngineering.org expanded McKinsey's original IDP reference architecture to create a tooling landscape that adopts the same umbrella subgroupings but includes other pluggable tools.

Backstage

The CNCF incubating project Backstage is arguably one of the most popular platform engineering tools. In fact, this project has the second largest number of contributors within the CNCF, behind Kubernetes! Backstage provides an open platform for building developer portals, powered by a centralized software catalogue, customizable with plugins. This is a fantastic OSS project that has really gained momentum and adoption lately, with over 3,000 companies having adopted Backstage for their own developer portals, including companies like Capitol One, CVS Health, Siemens, LinkedIn, REI, Vodafone, Lego, and more.

As standard, Backstage provides the following core components:

- Backstage Software Catalog for managing all your software (microservices, libraries, data pipelines, websites, ML models, etc.)
- Backstage Software Templates for quickly spinning up new projects and standardizing your tooling with your organization's best practices
- Backstage TechDocs for making it easy to create, maintain, find, and use technical documentation, using a „docs like code“ approach
- Plus, a growing ecosystem of open source plugins that expand Backstage's customizability and functionality

Traditionally, Backstage plugins are static (i.e. code-based). So, when you want to integrate any new plug-ins and features, it

involves modifying the core codebase of Backstage and then re-compiling, re-packaging and re-releasing the entire application stack. This process can be time-consuming and suffers from drawbacks including introducing increased complexity, dependency headaches, and increased time to production. However, this is starting to change and the Backstage community is currently working on enabling dynamic plugins.

Red Hat Developer Hub (RHDH)

Building on the success of Backstage, Red Hat launched the Red Hat Developer Hub (RHDH), a downstream project of Backstage.io. It offers an enterprise-grade, open developer platform for building developer portals, containing a supported and opinionated framework.

So, what the difference between RHDH and Backstage?

RHDH extends the functionality of Backstage through a set of predefined supported dynamic plug-ins providing extensions and components to make it easier for developers to design and build applications. The „dynamic“ plugin approach taken by Red Hat Developer Hub addresses the challenges of static plugins head-on by employing a “configuration over code” approach. This allows the loading of plugins dynamically at runtime and negates the need for complex code recompilation or container rebuilds. This approach to plugins enables many benefits, including immutability by design, enhanced flexibility and customization, simplified plug-in management and improved scalability and maintenance. However, as mentioned, this difference is changing as Red Hat have been contributing to Backstage with implementations and suggestions of how to enable dynamic plugins in this portal too.

One other difference to note is that when using RHDH, your app's configuration is now in YAML definitions, and not in the source code anymore (which is the case for Backstage). This offers much closer synergy with the principles of methodologies like the 12 and 15 factor app methodologies, that state that config should be separated from code for effective cloud native applications.

Kratix

Taking a slightly different approach to IDPs, Kratix is another open-source project (contributed by Syntasso) that can be used. It is an open-source cloud-native framework for building internal platforms, this time, specifically on Kubernetes. It enables the creation of a flexible platform with your paved paths as 'Promises' - any kind of abstraction that increases developer productivity, efficiency and security (e.g. developer portals, Databases as a Service, Kubernetes namespaces, etc). Kratix enables co-creation capabilities by providing a clear contract between application and platform teams through the definition and creation of these Promises. Where Kratix stands out from other OSS projects in this space, like Backstage, is that it supports platform teams to easily deliver a curated Kubernetes-native platform.

Other proprietary offerings

There are, of course, also a range of proprietary offerings, tools and services that can be used to create IDPs within an organization. Helpfully there is a list of these on the internal developer platform website: <https://internaldeveloperplatform.org/developer-portals/>. Many of these tools are SaaS offerings.

Summary

In summary, what we've tried to capture in this article is the fact that cognitive load and demands on developers are ever-increasing as we move towards cloud-native development, leading to burn-out, lower productivity, and demoralized teams. However, the use of Platform Engineering, Internal Developer Platforms (IDP), Internal Developer Portals, and Golden/Paved Paths can help developers with the challenges that they now face.

What this looks like will be specific to your organization because the ingredients you use to bake together your platform will be unique to the tools, libraries and technologies needed by your applications and your teams. But, the good news is that there are many examples, references, architecture maps and recommended tools and technologies out there that you can use if you're diving into this in your own organization

Next Steps

Now that we've covered the theory, if you're interested in getting hands-on with these technologies, you can follow our workshop on GitHub. It takes you through how you can set up an IDP with the portal, Red Hat Developer Hub, on OpenShift using the free developer sandbox environment. In this workshop, we've provided you with an example software template you can use, utilizing GitHub actions and Open Liberty. Try it for yourself: <https://github.com/grace-maarten/platform-engineering-101>

In this GitHub repository you can also access a list of real-world examples of companies implementing platform engineering and their own IDPs into their organizations and what impact this has had for them. We've included a range of companies of differing sizes and structures in this list, so hopefully you'll be able to learn from or relate to them, to help with your own journey potentially implementing this in your own organization.

Alternatively, you can access a whole community of support focused on platform engineering with the following links...

- <https://www.infoq.com/minibooks/platform-engineering-guide/>
- <https://platformengineering.org/slack-rd>
- <https://platformweekly.com/>
- <https://www.youtube.com/@PlatformEngineering>

www.javapro.io

www.javapro.io





#JAVAPRO #CLOUD

Greener Code: Sustainable Java Deployments with Native Builds and Knative Serverless on Kubernetes

Data Center Migration- Any techie who has spent a significant amount of time in Information Technology and working with large enterprises has heard of the word Data Center Migration. Quite likely you would have been a part of a migration project to the Cloud. Data Center Migration projects have been around for a long time and long before the word Cloud Data Center burst into the scene.

Mostly migration projects talk about cost optimization or cost savings. Cost savings is what drives most of these projects among decision makers. There is however a second "Sustainability" aspect as well. Data Centers consume a lot of electricity. Like so much that the world wide consumption of electricity by data centers was estimated to be about 415 TWH in 2022. For context, the total Electricity consumption by the entire nation of France in the same year was about 426 TWH. And this is before the AI boom led by ChatGPT and the like. This usage is only slated to increase.

So anything that we as technologists can do to reduce the energy consumption of applications would be of interest. It will

Authors:

Marius Stein is a freelance IT consultant and cloud engineer. He has many years of experience in advising companies on building cloud platforms and developing cloud-native systems. Additionally, he is a member of the program committee of the Cloudland Festival and co-organizer of the Cloud Native Computing Rheinland Meetup group.



Vishal Shanbhag
Entrepreneur | Startup Mentor | Blogger
| Technology Consultant | Tech Lead and
Programmer | Trekker
20+ years with a Proven track record as Senior
Technical Consultant, Software Architect, Lead
Developer and Pre-Sales Consultant.



of course also help with reducing running costs for applications. In the remainder of this article we will shed light on how three crucial technologies have evolved (namely Java, virtualization and containerization). We will start with a review of the early days of java and virtualization and end with modern concepts like GraalVM native builds and running serverless applications on kubernetes. On this journey we will showcase how a simple (yet enterprise ready) java application deployed as a simple kubernetes deployment can be turned to a native executable deployed as a serverless container on kubernetes with knative. We will fine tune this deployment to achieve sub-second cold start times when scaling up from zero and ultimately conclude how such an architecture can lead to more sustainability.

History - The 2000s

In the early 2000s, Java was hailed as a language that allowed creation of portable applications. One could write code once and compile it into a Java Byte Code representation. This byte code (aka .class files) could be packaged into a compressed file format (aka .jar) and run anywhere so long as the system had a Java Runtime Environment (JRE) available.

This meant that developers could create code and run builds on Windows or Mac, the OS that were more popular amongst end users, and still the same package could be used to run the final program on Linux or Unix. This worked so long as the target runtime OS had a compatible JRE available.

This along with many other factors such as availability of libraries and language features around object oriented programming and memory management allowed the language to reach a large scale adoption. Developers could now describe software data models in terminology similar to real word through use of Object Oriented Features. They could write complex programs without having to worry about keeping track of each and every byte of memory allocated to the program. So long as one followed some memory management best practices the system would take care of allocation and cleaning up.

While Java offered platform independence, enterprise software was still commonly deployed on physical servers following a one application - one machine approach. Benefits of this approach were a strong isolation between different applications running on different server and the ability to have different OS-Level dependencies in different applications. The obvious drawbacks were scalability and resource overprovisioning. Running one application on one physical server naturally meant that all resources that are available on the server could only be used by the application hosted on it. Resource sharing was not simple with this approach.

Virtual Machines - JRE and Hypervisors

But how did Java achieve platform independence? Put simply the java application does not interact directly with the OS. Any interaction with the OS is abstracted through use of a Java Virtual Machine (JVM). For all practical purposes the programmer uses

the JVM as if it is an actual Machine. It offers access to machine resources such as File System, CPU and Memory through the Java libraries. However, there is no need to perform any OS specific actions since there is only the need to know how to work with Java system libraries.

Java programs are compiled into a platform agnostic lower level representation known as the Byte Code. The byte code in itself cannot be run on any system. However a JRE (Java Runtime Environment) will spin up a JVM (Java Virtual Machine) at runtime which knows how to interpret the byte code instructions.

This is where Just In Time (JIT) compilation comes into play. Not all libraries needed by the application are linked in the Byte Code. Any library that is required by the application is only loaded and linked into the memory when it is used the very first time.

This basically means that a Java package is a loose assembly of different byte codes required by the program and the JIT compiler does the stitching together and optimization at runtime.

In a time when hardware procurement was time consuming (months) this portability of Java was helpful because applications could be written irrespective of what hardware they were to be deployed on. So long as the target data center had excess hardware available, one could deploy the application. As a result of this portability, in the decade between 2000 and 2010 we saw rapid adoption of Java.

Another technology that found broad adoption in this timeframe were hypervisors. Hypervisors benefit from the same concept of virtualization as java does, it just applies it on the OS and kernel level and not on the application level. Through virtualization, one physical server could be divided into multiple virtual ones, allowing to share the CPU and memory resources between those virtual servers and thus allowing for a more efficient resource allocation between workloads.

Virtualization works through simulating the hardware interface on a physical server through a software component called hypervisor. This hypervisor provides a virtualized hardware interface for multiple virtual machines running on one physical server. Each virtual server has its own encapsulated kernel for the interaction between the operating system and virtualized hardware. This allows for efficient resource allocation while strictly separating processes running on different virtual machines, maintaining a similar level of security and encapsulation compared to physical servers. Although virtualization has a positive impact on resource utilization, it still introduces overhead since each virtual machine runs their own OS and kernel which requires resources.

Also when applied on the application level virtualization introduces overhead. Anything running inside a JVM does:

- consumes more resources than a natively compiled executable. The JVM itself requires a finite amount of memory.
- has a slower start up time. Again JVM itself requires some

time to start up.

- Initial run of any application features tend to be a little slower than running the same feature a second time. This is due to JIT.

To tackle the problem introduced by JIT, libraries like Enterprise Java Beans and Spring were introduced. They allowed preloading of all necessary libraries through creation of “beans”. Basically any library that was going to be used by the application often would be preloaded into memory into a runtime instance called a “bean”.

While this improved the execution time of each code path after the application finished startup, it wasn’t without drawbacks. The main drawback was even slower application start up times since in addition to the normal JVM startup the application would also require a “Warm up” time to load its necessary resources. However, the runtime performance is superior since there is no time spent in JIT beyond the startup. Every library (class) that is needed by the application at run time is warmed up and ready to use.

Containers, Serverless and GraalVM native images

In 2013, Docker popularized a technology called containerization, which has since revolutionized the way applications are built, shipped, and deployed. Unlike virtual machines, containers do not simulate an entire hardware environment. Instead, they share the host operating system’s kernel, allowing for lightweight and efficient isolation of processes. Each container packages the application along with its dependencies and libraries, ensuring that it runs consistently across different environments, from development to production. This approach eliminates the need for a full operating system within each instance, resulting in faster startup times and lower resource overhead compared to traditional virtual machines.

Containers also enable greater scalability and flexibility. With orchestration tools like Kubernetes, organizations can deploy and manage thousands of containers across clusters, dynamically scaling resources based on demand. This makes container technology ideal for microservices and cell based architectures, where applications are broken into smaller, independently deployable components. Additionally, the portability of containers simplifies workflows, enabling developers to focus on writing code without worrying about compatibility issues across different environments

Another modern development in software engineering is the emergence of serverless computing, a paradigm that abstracts infrastructure management and allows developers to focus solely on writing and deploying code. Although the term „serverless“ can have varying definitions, we will define it here as a deployment model for workloads that satisfies the following criteria:

1. **Dynamic Scalability:** A serverless workload automatically scales to accommodate demand, ensuring optimal performance without requiring manual intervention.

2. **Zero-Scaling:** In periods of no demand, serverless workloads scale down to zero, eliminating idle resource costs.

3. **Abstraction of Server Management:** Developers are freed from managing the underlying infrastructure, including provisioning, patching, and scaling servers.

Through Dynamic Scalability and Zero-Scaling we can improve the resource utilization of deployed applications. Applications experiencing little or no traffic can release their allocated CPU and memory resources, making them available for other workloads in the deployment environment. As traffic increases, a serverless deployment environment scales applications up horizontally, to meet the increased demand, ensuring resilience in high workload scenarios without overprovisioning.

While serverless computing was initially popularized by public cloud platforms like AWS Lambda, Azure Functions, and Google Cloud Functions, the paradigm has also made its way into on-premises and hybrid environments through open-source solutions. Tools like Knative or OpenFaaS enable serverless deployment models on Kubernetes clusters, extending the benefits of serverless to organizations already invested in container orchestration.

However, for Apps built within the JIT paradigm, this means that the advantages offered by the JIT are no longer relevant. In fact in the Serverless world they are counter productive.

For one, with Docker and other containerization options, the application developer can control the full application stack including the OS without needing to worry about the underlying hardware. This means that the portability advantage of JVM startup is no longer necessary since developers can simply choose one target OS runtime inside the container.

Secondly, java still works within a JVM and therefore the time cost of the JVM startup is a real thing to contend with. And if one is using libraries like spring to create a large number of beans for use in monolithic applications, it makes the applications too large to be started and stopped at will. The startup times can range from a couple of seconds to sometimes even minutes.

Refactoring such applications into several smaller Microservices is already a thing and Java developers around the world have been adopting Microservices for a while. Despite such refactoring, sub second startup times are still a challenge.

It is not all gloomy though and the Java community is up to the challenge. One of the solutions is actually existent in history already with the older compiled languages like C, C++ which can produce “native” executable binary for the target OS platform.

Unlike the Java to Byte code compilation these languages compile the code into lower level instructions for the operating system. This binary executable contains the code from the developer’s program, its associated libraries, system libraries

and operating system features all linked together into a single optimized binary executable. This process is called Ahead of Time (AOT) compilation. In AOT, every library that is required for the program to run is known “ahead of time” and therefore at run time there is nothing extra to be done (like Class loading).

In essence the solution to the challenge of reducing startup and warmup time, lay in going back to the history and coming up with a “Java” solution to AOT. We will not dive into too much detail about the inner workings of AOT, but simply mention that the GraalVM offers the solution.

For the most part GraalVM can work with existing Java code without any modifications. It simply does static code analysis to identify all dependencies used by a Java program and then generates the optimized native library or executable version of the Java Program including all the necessary classes from its classpath dependencies and the JVM. This way anything that is not explicitly used by the Java program is simply left out from the compiled version.

A native binary thus generated can be potentially much faster than a regular Byte code compile because the startup times are cut down drastically. There isn’t a need to start a JVM, then do Class loading etc. All of the linkages are there in the native binary already and it can get straight to business.

Of course there are some limitations. GraalVM does static analysis and therefore if the program being compiled uses dynamic features such as loading libraries through Reflection, JNI, Serialization then the static analysis might not pick up all the relevant dependencies. In such a case there is explicit configuration required to inform GraalVM about the additional dependencies that will be used at runtime. The GraalVM documentation does offer tools and support to do this but that detail is beyond the scope of this article.

Suffice to say here that with the availability of AOT, Java applications can be more easily moved to the cloud and run inside Kubernetes or even AWS Lambda runtimes.

In the remainder of this article we will demonstrate how an enterprise ready Spring Boot Java application running on kubernetes can be migrated to a serverless application with knative. We will outline in what scenarios such a migration is feasible and what steps to take. Moreover we will demonstrate common pitfalls and how to overcome them. The demo application used here can be found on github: <https://github.com/stein-solutions/java-knative-demo>.

The Demo Application

Our demo application is a Java Spring Boot application consisting of a simple REST API allowing us to manage Product- and Category-Entities. Although the application is stateless, meaning that it does not persist any state on a harddrive that is attached to the application,

it initializes an H2 in memory database with a product and a

category table through JPA. Having a data layer leads to startup times that are more akin to real world scenarios. We have used H2 here purely to demonstrate database connectivity. In real applications H2 would likely be replaced by persistent databases like MySQL, PostgreSQL etc.

Besides that, the application integrates with the typical cloud native monitoring ecosystem by offering application metrics in the prometheus format. These can be scrapped by prometheus in regular intervals. The required prometheus configuration will be automatically applied through the use of the kubernetes Prometheus operator and the Service-Monitor Custom Resource.

The demo application is deployed to kubernetes as a Deployment and is accessible via an http-endpoint which is configured through a kubernetes Ingress Resource. There is no autoscaling configured for demo applications.

Creating a native Build on our local machine

After installing the required dependencies for GraalVM, creating a native executable with GraalVM from a spring boot application is straightforward. Recent Spring Boot versions already have a native maven profile and the native-maven-plugin (org.graalvm.buildtools) as part of their parent pom. With the following command our application is built into a native GraalVM executable.

```
mvn -Pnative native:compile
```

The resulting executable is an ahead of time compiled native executable of our application that is runnable on our host operating system and architecture. All uses of reflections or JNI calls were registered by the library implementing them. For libraries where this has not been done by the library itself (e.g.: H2 Database), there is a community project (<https://github.com/oracle/graalvm-reachability-metadata>) offering the required runtime hints. For cases where neither community nor library-provided runtime hints are available, these runtime hints need to be supplied manually by the application developer. One example of such are the runtime hints required for dynamically generated code by Project Lombok (<https://projectlombok.org/>).

Dockerized builds and execution environments

Next we dockerize the native executable. We suggest using a multistage build, where in a first stage we build the native executable and in a second stage we have the final container image just containing the built executable and possibly its dependencies. For the build stage we recommend using a base image provided by GraalVM. This base image already includes all required dependencies to successfully create a native executable of our application. We only need to instruct the container build process to create the native executable by calling the above mentioned mvn goal. The complete build stage can be seen below:

```
FROM ghcr.io/graalvm/native-image-community:23 AS
```

```

builder

WORKDIR /build
# Copy the source code into the image for building
COPY . /build

# Build
RUN ./mvnw --no-transfer-progress -e native:compile
-Pnative

RUN chmod +x /build/target/demo

```

The second stage of our container build process prepares the runtime environment of our native executable. If we use a base image that offers the same OS-Level dependencies as the build stage base image, we only need to copy over the created executable and define an entry point command that tells the container runtime what to execute when starting the container. In this example we use "ubuntu:jammy" as the base image. Moreover, it is good practice to define the ports that our application is listening on here, this however does not directly influence the way the container is executed and only serves descriptory purposes here. The full docker file can be seen below.

```

FROM ubuntu:jammy

# Copy the native executable into the containers
COPY --from=builder /build/target/demo /usr/local/
bin/app

EXPOSE 8080
ENTRYPOINT ["/usr/local/bin/app"]

```

The resulting container image is around 220mb "heavy".

Optimizing away

The native executable still contains a lot of OS-level dependencies through the use of ubuntu:jammy as base image. We can further reduce the file size by creating a fully statically linked executable of our application. GraalVM supports this out of the box, we just need to pass the parameters --static and --libc=musl to the build command and the resulting executable is statically linked with musl as standard c library implementation as part of the executable.

It is recommendable to create a separate maven profile for statically linked builds. Below is the code for that

```

<profiles>
  <profile>
    <id>nativelinked</id>
    <build>
      <plugins>
        <plugin>

```

```

          <groupId>org.graalvm.
buildtools</groupId>
          <artifactId>native-maven-
plugin</artifactId>
          <configuration>
            <buildArgs combine.
children="append">
              <buildArg>--
verbose</buildArg>
              <buildArg>--static</
buildArg>
              <buildArg>--
libc=musl</buildArg>
            </buildArgs>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>

```

This code extends the native-maven-plugin configuration to add the required parameter. To execute the build we run `mvn -Pnative,nativelinked native:compile`. It is important to note here, that the above command will fail on ARM macs. This results in an executable file which contains all required dependencies to be executed on a specific CPU architecture.

With having all OS-Level dependencies linked into the static executable, we can get rid of our base image for the application container and instead build the container image from scratch. The resulting docker image only contains our executable and no other files or folders. For the embedded tomcat inside Spring Boot to work, we need to create a /tmp directory, but since our scratch base image does not have any tools to create directories we simply copy over an empty directory from the build stage.

```

FROM ghcr.io/graalvm/native-image-community:23-
muslib AS builder

WORKDIR /build

# Copy the source code into the image for building
COPY . /build

# Build
RUN ./mvnw --no-transfer-progress -e native:compile
-Pnative,nativelinked

RUN chmod +x /build/target/demo
RUN mkdir /custom-tmp-dir

# The deployment Image

```

```
FROM scratch

EXPOSE 8080

# Copy the native executable into the containers
COPY --from=builder /build/target/demo /usr/local/bin/app
# Spring embedded Tomcat fails to start if /tmp is not present
COPY --from=builder /custom-tmp-dir /tmp

ENTRYPOINT [„/usr/local/bin/app“]
```

The resulting container image is around 90mb “light”.

Deploying on Kubernetes with knative

Knative is an open source solution for kubernetes that allows the deployment of serverless applications that can scale to zero on kubernetes. To deploy an application onto an knative enabled kubernetes cluster the following yaml configuration can be used:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: test-java-knative-demo
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/window: „6s“ # Set
a custom stable window
    spec:
      automountServiceAccountToken: false
      containers:
        - image: „ghcr.io/stein-solutions/java-knative-demo:nativelinked-f5b208448747baf69e9afe06f0cac1d0b86a265e“
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
          resources:
            limits:
              cpu: 125m
              memory: 256Mi
            requests:
              cpu: 125m
              memory: 256Mi
```

This will create a knative service that dynamically scales based on the amount of http requests that are sent to the application. If there is no http traffic directed at our application, knative will scale down our application to zero replicas, freeing all CPU and memory requests from the environment. In our configuration the time frame after which knative will scale down if no requests

are received is 6 seconds (the minimum possible time frame), in production environments it is probably reasonable to choose a timeframe that is longer to avoid cold starts.

Coldstarts - the bane of serverless and how to solve it

For serverless applications to be feasible, reasonable cold start times are a necessity. When operating user facing APIs, response time of many tens of seconds are unacceptable.. This is where native builds come into play.

“Normal” (read non-native builds) spring boot applications have quite slow startup times. While running our demo application application on an M2 Pro Macbook, the application startup takes around 1.6 Seconds, however, in more resource constrained environments the startup time easily exceeds 10 seconds and more. When restricting the available CPU for our containerized spring boot application to 200 Milicors (20% of one CPU core), the application needs around 30 seconds until being fully started with knative. Even when allowing for one full CPU it still takes around 10 Seconds and when not restricting CPU usage (on a Azure Standard_D2s_v3 Worker Node - 2 CPU available), we can achieve startup times of around 5 seconds.

The problem with assigning one full CPU or more to our Demo application is that those resources will be blocked for the entire lifecycle of the application and not only for application startup when they are needed. After application startup only around 200 Milicors or less will be needed to answer http requests and during idle time there is almost no need to reserve resources to our application at all. However, our application will still block 100% of the requested CPU core.

To address the issue of resource overprovisioning, Kubernetes provides the ability to configure resource requests and limits for containers. Resource requests define the minimum guaranteed CPU and memory that a container will have access to, ensuring stability and performance. Limits, on the other hand, cap the maximum resources a container can use, preventing it from consuming more than its allocated share and affecting other workloads.

By setting a lower CPU request (e.g., 200 millicores) and a higher CPU limit (e.g., 1 full core) for our Spring Boot application, we can strike a balance. This approach ensures the application gets sufficient resources during startup while reducing idle resource usage. The application can briefly utilize additional resources when needed, without permanently blocking them for other workloads. The main drawback of this approach is that the application is not guaranteed the one full CPU core, if there is not enough CPU time available the application will get less (but at least 200 millicores). Moreover, during times with no requests the 200 millicores will be reserved for our application and unavailable to other applications.

By optimizing resource allocation with Kubernetes and dynamically adjusting CPU requests and limits, we can reduce the need for overprovisioning, thereby limiting the number of CPUs required in the cluster. This approach supports a more sustainable IT infrastructure by minimizing the environmental

impact of maintaining excess hardware. We did tests with our demo application deployed through knative on an Azure Kubernetes cluster. As worker nodes we used Azure Standard_D2s_v3 instances with 2 Cores and 6 GB of memory. The below table shows metrics about the request duration and app start times of cold starts. For each configuration, as defined by CPU and Memory allocated, we measured 80 cold starts. All measurements had the container image of our application already downloaded on the respective worker node. Since not only the application needs to start but also knative needs to route the request to the started pod, the full request takes

Config	Avg. Req. Duration	Avg. App Start	Median Req. Dur.	Median App Start	2 sigma Req. Dur.	2 sigma App Start
50m / 128Mi	3.989s	2.765s	2.961s	2.758s	3.291s	2.908s
75m / 128Mi	2.013s	1.887s	2.005s	1,877s	2.170s	1.993s
125m / 256Mi	1.177s	1.021s	1.163s	1.019s	1.32s	1.115s
250m / 256Mi	1.002s	808.958ms	1.006s	798.205ms	1.079s	924.783m
750m / 512Mi	988.207ms	575.536ms	1.003s	554.031ms	1.085s	692.777ms
1000m / 512Mi	990.191ms	530.500ms	1.005s	524.964ms	1.060s	594.912ms

Process for persistent volumes usually takes time (at least a couple of seconds), this leads to cold start times that would be unacceptable. Even only mounting the kubernetes service account token to the application will increase the average request time of a cold start by about 200ms.

From a sustainability perspective it is necessary to deploy multiple applications onto our infrastructure before we can see a reasonable improvement in sustainability. Running knative itself on a kubernetes cluster requires

When giving 1 full CPU core to the application a cold start takes on average 990ms, however it only takes around 530ms for our application to be started and ready to serve traffic. The remaining time the request is processed by the knative activator component before being redirected to the actual application pod.

One remarkable thing we found is that we can reduce the CPU and memory allocation to about 125 Millicores and 256 mb of RAM before the request time starts to significantly drop. With this configuration we still achieve an average request duration of 1.177 seconds (only around 180ms slower than with one full core). However, the app start time is significantly higher compared to the first configuration. Here the app takes 1.021s on average to start up, almost 500ms more than in the first configuration. This gives us as application developers some wiggle room. App start of around one second will lead to similar request durations as app start of about 500ms seconds.

In configurations with 75 Millcors (1/16 of a CPU), requests take significantly more time. This increase in request duration is mainly driven by an increase in the app start time, only an overhead of about 100ms is introduced by knative request routing.

Limitations

The architecture behind knative only allows autoscaling based on http requests, not based on TCP connections. Because of this, websocket-connection can be initialized but will not be considered for the scaling decisions of knative.

Moreover, for the cold start times to be reasonable, it is necessary that the application is stateless. Although possible, it is not recommended to mount persistent volumes. The mounting

at least 6 CPU and 6Gi of memory according to the official documentation. So for our setup to be sustainable, we need to deploy a critical mass of knative services onto our cluster.

Additionally, the load pattern that is handled by our application should be volatile, so that longer periods without requests actually exist. Knative shines for applications that are rarely requested, the more constant the load actually is, the less resources can be saved through this approach.

Outlook

In this article we have shown that it is possible to run stateless, enterprise-ready java applications as serverless containers on top of kubernetes. The cold starts of those containers are comparable to other serverless technologies like AWS lambda and others. This approach can lead to a more sustainable IT system in certain load scenarios. Applications that only are invoked rarely but require swift response times would benefit from this architecture.

However, as with everything in IT, this architecture is not a silver bullet solution to all applications and architecture needs.

References

- <https://www.linkedin.com/pulse/managing-forecasting-your-cloud-consumption-prakash-a/>
- <https://www.cesarsotovalero.net/blog/aot-vs-jit-compilation-in-java.html>
- <https://www.statista.com/chart/32689/estimated-electricity-consumption-of-data-centers-compared-to-selected-countries/>
- <https://www.graalvm.org/jdk21/reference-manual/native-image/dynamic-features/Reflection/>

From Silent Coder to Respected Leader

Java technical leadership is a journey that transforms developers from focused coders into community contributors. With „silent coder“ I mean a person who is focused on writing software but has no intention or interest in sharing what they have learned. Productivity is measured in how much work I as a person can get done. This mirrors Java's early days when it was a closed-source language, its evolution hidden behind Sun Microsystems' walls, its growth limited by what a single company could envision.

I still remember the moment everything changed. It was 2014 at JavaOne in San Francisco, which was the biggest and most famous Java conference in the world for decades, and I was watching Venkat Subramaniam present Java 8's new features. As he transformed traditional for-loops into elegant stream operations, engaging with the audience and making complex concepts approachable, I realized something profound: technical knowledge alone wasn't enough – how we share that knowledge matters just as much.

This was my first international Java conference, coinciding with one of the most revolutionary changes in Java's history. The entire community was buzzing with excitement about Java 8's new direction, with developers eagerly learning and teaching each other about lambda expressions, the Stream API, and the Optional class. The energy was infectious, but what truly changed my perspective was watching Venkat present. Here was Java 8, a complex and transformative update that could have been intimidating, being explained with such clarity and enthusiasm that even skeptics were won over.

Author:



Markus Westergren is a staff engineer with over 20 years of experience. During his work with large backend projects, Markus has repeatedly seen the big impact code quality has on the outcome of the projects and the careers of the developers working on them. Because of that, Markus focuses on helping professional Java back-end developers write maintainable code so that they become sought-after experts in the software development industry.

<https://twitter.com/markuswestergren1>
<http://www.empatheticdev.com>



„We write code for others to read and occasionally for machines to execute“

-Venkat Subramaniam

I remember going back to my hotel room that evening, my mind racing with possibilities. For the first time, I could envision myself not just learning from speakers, but potentially becoming one. The next morning, I even worked up the courage to approach Venkat after his session on functional programming patterns. To my surprise, he was welcoming and encouraging, spending several minutes discussing how he prepares his presentations. This brief interaction showed me that even renowned speakers in our community are approachable and willing to help others grow.

But my journey to that transformative moment - and beyond it - wasn't straightforward. I had been working with software development for 25 years, paralleling much of Java's own journey. Just as Java started as a language meant for embedded systems and television set-top boxes, I began my career focused narrowly on embedded systems and C programming. And like Java, my journey has been one of continuous evolution.

The Silent Years (1990s-2010)

We don't think you are suited to work in a team, that you are more of a person who wants to sit in a corner working on your own thing

-Feedback from first interview

The Early Days

My journey as a developer mirrors Java's early transformation in fascinating ways. Just as Java began as Oak, a language designed for embedded systems and interactive television, I started my career focused on embedded systems and C programming. And just as Oak's creators discovered their technology had potential far beyond its original scope, I would eventually learn that my capabilities extended beyond solitary coding.

My journey started with a bachelors degree in electronics with a major in computer engineering. Like Oak's initial closed development within Sun's „Green Project,“ I was totally focused on the technology and dreading standing in the spotlight. During my education, I was mainly working with embedded systems, assembler and C, and I enjoyed it - much like how Oak was originally designed for embedded consumer devices.

This was the late 1990s, a pivotal time when Java was making its own transition. Just as Java's team realized that „Write Once, Run Anywhere“ could revolutionize general-purpose programming, I got my first taste of Java in a course. But while Java was embracing openness and cross-platform compatibility, I retreated back to C and systems programming, choosing isolation over collaboration - much like how Oak might have remained a niche technology had it not evolved into Java.

The idea of writing software for end users and working in teams didn't appeal to me - I wanted to focus on the low-level technology, just me and the problem, no other people involved. The irony isn't lost on me now: while Java was transforming from a closed project into an open, community-driven platform that would connect millions of developers worldwide, I was deliberately choosing paths that would keep me isolated.

I actually got rejected in one of my first job interviews with the feedback, „We don't think you are suited to work in a team, that you are more of a person who wants to sit in a corner working on your own thing.“ Looking back, this feedback perfectly captured where I was in my journey - still focused on the machine level while the industry, led by Java, was moving towards more collaborative, high-level software development. Just as Java had to break free from its initial constraints to reach its full potential, I too would need to step out of my comfort zone to grow.

Reflect: Think about your current relationship with coding and community. Are you, like I was, more comfortable with the technical aspects than the human connections? What holds you back from sharing your knowledge?

Action Items:

1. Schedule 15 minutes this week to write down one technical problem you solved recently
2. Share your solution in your team's Slack/Discord channel or during the next team meeting
3. Find and join your local Java User Group on Meetup.com -

even if you just observe the first few meetings

4. Set up a technical blog (e.g., on dev.to or Medium) - write your first draft about that problem you solved

First Steps Toward Community (2000-2010)

While Java developers were building a robust ecosystem of shared libraries and frameworks, I was discovering the power of community in the Linux world - though I remained just a consumer of information, afraid that I had nothing unique to contribute.

Finding My Place in the Linux World

I managed to get a job at a small company working mainly with systems programming and device drivers in Linux. That's where I first came into contact with a community. While Java developers were building a robust ecosystem of shared libraries and frameworks through the newly-created Java Community Process (JCP), I was discovering the power of community in the Linux world.

But I was still just a consumer of information, not sharing what I knew because I felt that I didn't know anything that the community didn't already know. I was also afraid to make mistakes that would make me look incompetent. The sometimes harsh tone in the Linux community - quite different from the more welcoming Java community I would later discover - didn't help my confidence.

For those who, like me then, are solely consuming community content - take that first step toward participation. Find your local Java User Group (JUG). Even if you just attend one meeting, you'll be surprised how welcoming these communities are. I wish I had done this sooner in my journey.

The Unexpected Bridge: From Fitness to Speaking

Around 2010, as Java was evolving from its enterprise-focused Java 6 era toward the more modern Java 7, something unexpected happened that would later prove crucial to my transformation. I joined a local gym to address my health issues, and discovered a passion for fitness that would ultimately help me overcome my fear of public speaking.

When the gym needed instructors, I volunteered - a decision that seemed unrelated to my technical career at the time. But leading fitness classes taught me fundamental lessons that would later prove invaluable:

- How to read a room and adjust based on participant feedback
- Breaking down complex movements into digestible steps
- The importance of authenticity over perfection
- Managing energy levels and engagement

These skills would soon find an unexpected application in my technical career. Just as I learned to break down complex workout

routines into manageable sets, I would later apply the same principle to explaining intricate Java concepts. The confidence I gained from standing in front of a fitness class would become the foundation for my future technical presentations.

Reflect: What experiences outside of tech have taught you valuable skills? How might these translate to your development career?

Action Items:

1. List three non-technical activities you enjoy or excel at
2. For each activity, identify one skill that could apply to tech leadership (e.g., teaching music = breaking down complex concepts)
3. Volunteer to lead your next team's retrospective, applying these transferable skills
4. Ask your manager for a 15-minute meeting to discuss growth opportunities that align with these skills

The Transformation Begins (2010-2014)

As we debated language features and solved practice problems together, I discovered something surprising: each of us, despite working with the same language, brought unique insights and approaches to the table.

Finding Community in Java

When I joined a company building applications for cell phones using Java and Java ME, I found myself naturally applying these presentation skills in team discussions. The same techniques I used to guide people through challenging workouts helped me explain technical concepts to colleagues. During our code reviews, I started noticing how my experience reading the energy of a fitness class translated surprisingly well to gauging team members' understanding of complex code.

Those study sessions became my first real taste of collaborative learning in the Java ecosystem. As we debated language features and solved practice problems together, I discovered something surprising: each of us, despite working with the same language, brought unique insights and approaches to the table.

This realization was preparing me for that transformative moment at JavaOne, though I didn't know it yet. Each small step toward community involvement was building the foundation I would need.

If you see yourself in this phase of my journey, start small. Next time your team has a technical solution to discuss, volunteer to present it. Even a 5-minute walkthrough of your code during a team meeting can build your presentation muscles. The key is to begin with an audience that already knows and supports you.

After JavaOne - Putting Inspiration into Action (2014-2022)

If I could lead a room full of people through a challenging workout, surely I could guide fellow developers through a technical topic I knew well.

From Inspiration to Implementation

After returning from JavaOne, inspired by Venkat's presentation style, I started small. I began with lunch-and-learn sessions about Java 8 features for my colleagues. Just as developers were discovering the elegance of functional programming in Java, I was discovering the joy of helping others learn.

My experience as a fitness instructor proved unexpectedly valuable. The same skills I used to guide people through complex workout routines now helped me explain technical concepts. If I could lead a room full of people through a challenging workout, surely I could guide fellow developers through a technical topic I knew well.

Reflect: Who are the speakers or community members that have inspired you? What specifically about their style resonates with you?

Action Items:

1. Follow three Java community leaders on BlueSky/LinkedIn who inspire you
2. For each activity, identify one skill that could apply to tech leadership (e.g., teaching music = breaking down complex concepts)
3. Volunteer to lead your next team's retrospective, applying these transferable skills
4. Ask your manager for a 15-minute meeting to discuss growth opportunities that align with these skills

The Transformation Begins (2010-2014)

I could lead a room full of people through a challenging workout, surely I could guide fellow developers through a technical topic I knew well.

Reflect: Who are the speakers or community members that have inspired you? What specifically about their style resonates with you?

Action Items:

1. Follow three Java community leaders on BlueSky/LinkedIn who inspire you
2. Write down three presentation techniques you admired from recent tech talks you've watched
3. Pick one upcoming Java conference (virtual or in-person)

and submit a proposal for a lightning talk

4. Record yourself explaining a Java concept for 5 minutes - watch it back and note areas for improvement

The Next Level: Oracle Code One 2022

Just as my JavaOne experience with Venkat had opened my eyes to the power of technical presentation, a session at Oracle Code One 2022 would take my journey to the next level. I attended a session about „Growing beyond senior“ with Bruno Souza, memorable for wearing a Brazilian flag as a cape. His message about technical leadership resonated deeply with where I was in my journey.

After that session, I did something that would have been unthinkable to my „silent coder“ self - I approached Bruno to continue the conversation. This time, unlike at JavaOne, I wasn't just seeking inspiration; I was ready to take action. I became his mentee, and that relationship would prove instrumental in preparing me for my biggest challenge yet: presenting at Devovx Belgium.

Building Bridges in the Community

„NCAY - Nobody Cares About You, they care how you can help them solve their problems“

-Bruno Souza

As I worked with Bruno, I began to see how my various experiences were connecting. The confidence I had gained from fitness instruction, the technical depth from my years of coding, and the inspiration from Venkat's presentations - all these pieces started fitting together.

My journey to international speaking began with small steps in the local community. I started attending more JUG meetings, not just as a passive listener but as an active participant. Each small interaction built my confidence incrementally.

Finding a mentor transformed my journey, but I had to take the first step by approaching potential mentors. Look for experienced developers in your community who inspire you. Many are happy to provide guidance - they remember their own journey and want to pay it forward.

Through my mentor's guidance, I learned that technical leadership isn't just about speaking at conferences or writing blog posts - it's about identifying opportunities to solve problems and share solutions in ways that benefit the whole team. This lesson proved especially valuable when we faced a significant challenge with application performance monitoring.

Through a beta testing program with a new APM tool, our team discovered how powerful development-time telemetry could be for improving application quality.

By instrumenting our application with OpenTelemetry, we gained insights that transformed how we approached performance and debugging:

Understanding N+1 Query Patterns: One of our most valuable learnings came from analyzing trace data during development. We discovered that a seemingly innocent REST endpoint was causing significant database load. The telemetry data revealed a classic N+1 query pattern hidden in our layered architecture - for each parent record, we were making separate database calls to fetch child records. By identifying this pattern early in development, we refactored our code to use join fetches, significantly reducing database load.

Cache Effectiveness Analysis: Telemetry data also revealed gaps in our caching strategy. While we had implemented caching throughout our application, the trace data showed we weren't getting the hit rates we expected. In some cases, we were repeatedly fetching the same data despite having a cache in place. This led us to discover subtle bugs in our cache key generation and helped us optimize our cache eviction policies.

Shifting Left with Observability: Perhaps the most important learning was how integrating observability into our development workflow changed our team's approach to performance. Instead of treating performance as a production-only concern, we made it part of our daily development practice:

- We started reviewing trace data during code reviews, catching potential performance issues before they reached production
- Team members began annotating their code with custom spans to provide better context for complex operations
- We established baseline performance metrics during development, making it easier to identify regressions

This experience taught me that sharing knowledge about observability patterns and practices was just as important as sharing code solutions. I organized a company-wide presentation about our learnings, demonstrating how early performance visibility had fundamentally changed our development workflow.

The presentation sparked valuable discussions about observability patterns across different teams, leading to the creation of shared instrumentation guidelines and performance monitoring standards. This collaborative approach to solving performance challenges became a model for how we could turn individual technical discoveries into team-wide improvements.

Your experiences, even if they seem ordinary to you, can provide valuable insights for others. Consider starting a technical blog or sharing your learnings on social media. That debugging breakthrough you had last week? Someone else is probably struggling with the same issue right now.

The presentation sparked valuable discussions about observability patterns across different teams, leading to the creation of shared instrumentation guidelines and performance monitoring standards. This collaborative approach to solving performance challenges became a model for how we could turn individual technical discoveries into team-wide improvements.

Your experiences, even if they seem ordinary to you, can provide valuable insights for others. Consider starting a technical blog or sharing your learnings on social media. That debugging breakthrough you had last week? Someone else is probably struggling with the same issue right now.

Reflect: What technical challenges have you solved that others might benefit from hearing about? How could sharing your solution help others avoid similar problems?

Action Items:

1. Start a „TIL“ (Today I Learned) document to track your daily technical discoveries
2. Write a detailed Stack Overflow answer to one Java question this month
3. Contribute to one open source Java project's documentation
4. Schedule a lunch-and-learn session with your team about a recent technical challenge you overcame

Coming Full Circle - From Audience to Speaker

„Remember, your audience wants you to succeed“

-Frank Delporte

The Path to Devoxx Belgium

As I prepared for Devoxx Belgium, every lesson from my journey came into play. The presentation skills from my fitness instruction days helped me structure my talk. The technical depth from my years as a developer gave me confidence in my material. The inspiration from Venkat's JavaOne presentation guided my delivery style.

When I reached out to the Foojay community for advice, Frank Delporte responded with invaluable tips. „Remember,“ he wrote, „even though the conference is big, it's a friendly crowd. They want you to succeed.“ These words echoed what I had learned from Venkat years ago - that our community grows stronger when we help each other.

Standing on the Other Side

Finally, I found myself standing on that sloped stage at Devoxx Belgium, facing hundreds of developers. In that moment, I thought back to watching Venkat at JavaOne. Now I was the one sharing knowledge, helping others grow. The nervous developer who once sat in the audience was now the one inspiring others.

After my talk, I had the chance to thank Frank in person, and he invited me to join some speakers at a local pub. There, I met Gerrit Grunwald and other community members. As we shared experiences and stories, I realized how far I'd come from the days when I thought, „why would a speaker want to talk to me?“

As you grow in your journey, remember to create opportunities

for others. Whether it's helping a junior developer understand a concept, reviewing someone's first conference proposal, or sharing speaking opportunities - you can be the bridge for others just as others were for me.

Reflect: What would giving your first tech talk feel like? What topic could you speak about for hours if given the chance?

Action Items:

1. Create an outline for a 30-minute presentation on your chosen topic
2. Practice the first 5 minutes of your talk and record it
3. Share the outline with a trusted colleague for feedback
4. Submit the talk proposal to a local JUG or virtual Java meetup

The Virtual Revolution: Community Beyond Borders

The evolution of our community took an unexpected turn when global events forced us to change how we connect. What initially seemed like a setback revealed new possibilities for knowledge sharing and community building. Virtual platforms removed geographical barriers, making Java expertise accessible to developers worldwide.

My first virtual conference talk was an eye-opening experience. While I missed the energy of a physical room, I found that the chat was buzzing with questions and discussions, sometimes more actively than in-person sessions. Developers who might have been too shy to raise their hand in a crowded room were now actively participating through text.

Tools like Discord, Twitter Spaces, and virtual JUG meetups have created persistent spaces for continuous learning and mentorship. I've had enlightening discussions with developers from Bangalore to Buenos Aires, each bringing their unique perspective to Java development. These platforms have become particularly valuable for developers in regions without active local Java communities.

Some key lessons I've learned about virtual community participation:

- Async interactions can be as valuable as live ones - sometimes more so. A thoughtful GitHub discussion or Stack Overflow answer can help countless developers over time.
- Screen sharing during pair programming sessions can be more effective than looking over someone's shoulder. Tools like VS Code Live Share have transformed how we collaborate on code.
- Recording and sharing technical content has become more accessible. My blog posts and recorded presentations now reach developers I could never have connected with through traditional channels.

Virtual platforms have also democratized knowledge sharing. You don't need to wait for a major conference to share your insights. Starting a technical blog, contributing to open source discussions, or hosting online workshops can help you build your voice in the community.

Bridging Physical and Virtual Worlds

The future of our community isn't about choosing between physical or virtual engagement - it's about leveraging both. When I mentor developers now, we might meet in person monthly while maintaining regular contact through Discord. Local JUG meetings are streamed to reach wider audiences, with remote participants contributing actively to discussions.

The key is understanding the strengths of each medium. Complex technical discussions often benefit from asynchronous, written communication where people can think through their responses. Live sessions, whether virtual or in-person, excel at building connections and handling dynamic Q&A.

Reflect: How has virtual community participation changed your learning journey? What opportunities could you create by combining physical and virtual engagement?

Action Items:

1. Join two Java-focused Discord servers or Slack communities
2. Schedule one virtual pair programming session with a colleague
3. Create or join a virtual Java study group
4. Start a weekly technical discussion thread in your team's communication channel

Lessons for the Next Generation

Looking back on my transformation from silent coder to community leader, I've learned several crucial lessons that I believe are valuable for any developer on a similar journey:

The Power of Small Steps

Remember how terrified I was of public speaking? What I didn't realize then was that presenting isn't just about standing on a stage - it's about impact. Each small step built confidence:

- Leading study groups for certification
- Giving lunch-and-learn sessions
- Teaching fitness classes
- Speaking at local meetups
- Writing blog posts

- International presentations

The Importance of Mentorship

My journey has been shaped by various forms of mentorship:

- Indirect mentorship through speakers like Venkat
- Formal mentorship with Bruno
- Peer mentorship in the Java community
- Now, becoming a mentor myself

Technical Excellence Isn't Enough

What I've learned is that technical expertise alone doesn't make a leader. It's the ability to:

- Share knowledge effectively
- Build bridges between people and technology
- Create opportunities for others to grow
- Contribute to the community's collective knowledge

AI and the Evolution of Developer Learning

Transforming How We Learn and Share

The emergence of AI coding assistants has added a fascinating new dimension to how we learn and share knowledge in the Java community. When I first started programming, learning meant poring over documentation and books. Today, developers have AI-powered tools that can explain code, suggest implementations, and even help debug issues.

But rather than replacing community learning, these tools have enriched our discussions in unexpected ways. We are raising the level of abstraction. AI tools can help us discover alternative implementations that we haven't considered. The focus is shifted from „what's the right way?“ to „what are the tradeoffs between these different approaches?“

AI tools are like having a junior developer who has read all the documentation but needs your expertise to make good decisions.

New Challenges, New Opportunities

This shift brings both opportunities and challenges for our community:

- Code reviews have evolved. Instead of focusing solely on syntax and style (which AI tools can check), we spend more time discussing architecture, maintainability, and business logic.

- The „why“ has become more important than the „how.“ When AI can suggest multiple valid implementations, understanding the principles behind choosing one approach over another becomes crucial.
- Knowledge sharing has become more nuanced. Rather than just sharing solutions, we're sharing decision-making frameworks and architectural thinking.

In my mentoring sessions, I've noticed a change in the questions I receive. Instead of „How do I implement this pattern?“, developers are asking „The AI suggested these three approaches - how do I evaluate which is best for my use case?“ This shift toward higher-level thinking is making our community discussions richer and more valuable.

Community Role in the AI Era

The rise of AI tools hasn't diminished the importance of community - it's made it more vital than ever. Here's why:

- AI tools can provide answers, but they can't share experiences. When I talked about the Digma observability implementation earlier, the value wasn't just in the technical solution, but in the real-world context and lessons learned.
- Community discussions help develop critical thinking about AI suggestions. During our team's lunch-and-learn sessions, we now regularly discuss how to effectively collaborate with AI tools while maintaining code quality and architectural integrity.
- The community helps navigate the rapid evolution of AI capabilities. Through our JUG meetings and online forums, we share best practices for integrating these tools into our development workflows.

Reflect: How has AI changed your learning and knowledge-sharing patterns? What aspects of community learning become more important, not less, in an AI-assisted world?

Action Items:

1. Join two Java-focused Discord servers or Slack communities
2. Create a team guide for effectively using AI pair programming tools
3. Organize a team discussion about AI best practices in your development workflow
4. Start a repository of interesting AI-human pair programming sessions and their outcomes

I've found that AI tools, much like the IDEs and frameworks that came before them, are becoming part of our shared toolkit. But they're not replacing the human elements that make our community special - the mentorship, the shared experiences, and the collective wisdom that comes from years of solving real-

world problems together.

Looking Forward: The Next 30 Years of Java

The future I envision isn't just about technical innovations - though those are important. It's about creating an ecosystem where every Java developer feels empowered to contribute their unique perspective. Whether you're a developer working on enterprise applications in Stockholm, a student learning Java in Lagos, or a startup engineer building microservices in Singapore, your experiences and insights matter.

A Call to Moving Forward

As Java enters its fourth decade, your voice matters more than ever. Throughout this article, we've explored various ways to grow from a silent coder to an active community member - from those first tentative steps of sharing in team meetings, to joining local JUGs, to sharing knowledge through blogs and presentations, to finding and becoming mentors.

Remember that becoming a technical leader isn't about dramatic transformations - it's about taking small, consistent steps forward while bringing others along with you. Every experienced speaker in our community started exactly where you are now. The strength of Java's future lies not just in its technical innovations, but in the willingness of developers like you to share your unique perspectives and experiences.

What will your next step be in this journey?

Your Journey Begins Now

As I write this on Java's 30th anniversary, I can't help but reflect on my own transformation from that nervous developer who couldn't imagine speaking to others, to someone who now regularly shares knowledge at international conferences. But more importantly, I think about all the developers reading this who might see themselves in my „silent coder“ phase.

Remember: I didn't transform overnight from a silent coder into a conference speaker. My journey began with a single small step - explaining a Java 8 stream operation to a colleague during lunch. Your journey can start just as simply:

Today:

- Open your IDE and find a piece of code you're proud of. Write a brief comment explaining not just what it does, but why you designed it that way. This is your first step in documenting knowledge for others.
- In your next team meeting, speak up about one technical challenge you solved recently. It doesn't need to be revolutionary - even small solutions can help others.

This Week:

- Find your local Java User Group on Meetup.com. Just knowing when they meet is a start. When you're ready, attend one meeting - you don't need to speak, just observe.
- Write a „TIL“ (Today I Learned) note about something you discovered while working with Java. Keep it in a private repository for now. This builds the habit of documenting your learning journey.

This Month:

- Share one of your „TIL“ notes with your team. It could be as simple as a useful method you discovered in the Java Collections API.
- Comment thoughtfully on one open source Java project you use. Even pointing out unclear documentation helps the community.

This Year:

- Give a 10-minute presentation at your company about a Java feature you recently mastered.
- Write your first technical blog post. Start with something you wish you'd known a year ago.
- Submit a proposal for a lightning talk at a local or virtual Java meetup.

The Java ecosystem thrives because developers like you eventually choose to share their experiences. Your unique perspective matters - whether you're wrestling with Spring Boot configurations in Stockholm, optimizing Android apps in Lagos, or building microservices in Singapore. The solutions you've found, the mistakes you've made, the workarounds you've discovered - these are all valuable pieces of our community's collective knowledge.

Think back to a moment when you were stuck on a problem, and someone's blog post or Stack Overflow answer helped you break through. Now you have the opportunity to be that helper for someone else. Your voice - yes, yours - could be exactly what another developer needs to hear.

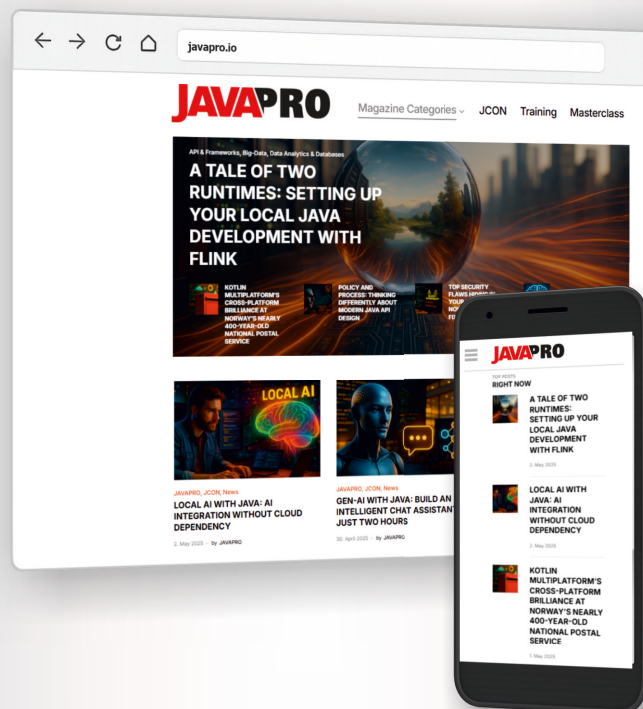
So, what will your first step be? The Java community has been building for 30 years, and we're just getting started. Your chapter in this story begins now.

Remember: every Java expert you admire started exactly where you are today. The only difference is they took that first step.

Take yours today.

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:

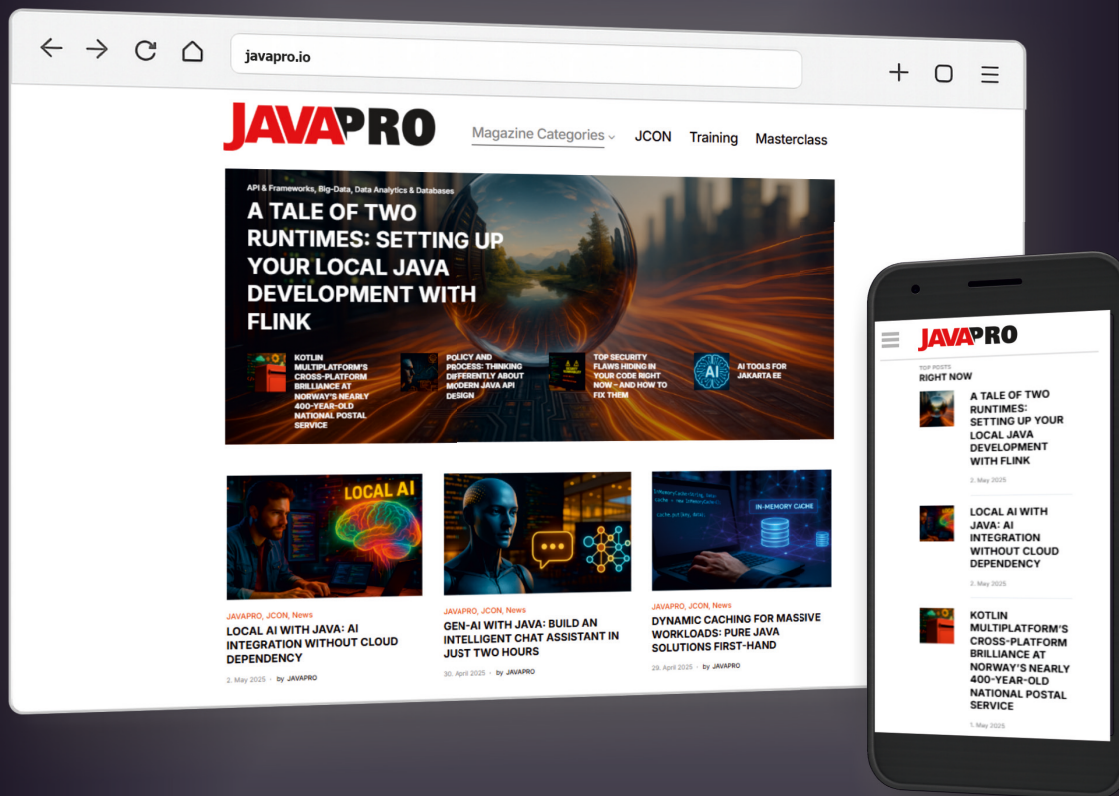


BECOME A JAVAPRO AUTHOR & BLOGGER !

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY **#JAVAPRO**

DISCOVER MORE GREAT ARTICLES ONLINE:



BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

Should we Stop Discussing about Technical Debt... with Top Management?

Cyclomatic complexity, coupling and cohesion, orthogonality, big O... are concepts that technical teams use when analyzing complex software systems and can establish why it is difficult to change them. Technical language refers to the technical aspects of those systems. But when we have to explain to non-technical people in the organization why an application has bugs, why a feature is not on time, or why we need a database expert to make a simple change to a service, we cannot use those concepts. To do that, we have to speak in a language that those non-technical people understand the intrinsic nature that exists in systems developed with software.

Ward Cunningham (among other things, author of the first wiki) found himself in this same dilemma when he had to explain to his boss the need to invest in refactoring the financial product they were working on. And he came up with the metaphor of "Technical debt". Technical debt refers to the implied cost of additional work in the future, resulting from choosing an expedient solution over a more robust one. While Technical debt can accelerate development in the short term, it may increase future costs and complexity if is left unresolved/unpaid.

Cunningham presented the metaphor to the world in this short article [1] in 1992, (I guess at that time James Gosling was already working on the first version of Java, but it was still three years away from being released). So, 33 years later, we are still using the metaphor, because it is a very good one. It is an easy way to explain technical issues to the people who know about the business, using a common language that everyone can understand quickly.

Are we overusing the metaphor or using it as an excuse?

Before continuing, let me explain a personal story. Some years ago, working in a consultancy, we were hired for a project to improve an application and a team that had been stagnating for years due to a huge technical debt. After a quite thorough analysis of their systems, their technology, and their teams, we proposed to make an incremental migration of their monolith to smaller and more manageable services, breaking the coupling and allowing the systems to be divided into independent domains. Also accompanied by an up-skilling of their teams because we had also opted to change the technology they worked with, which was quite obsolete. Of course, we were going to explain that the reason why changes could not be made, why new features could not be added, why the performance

was so poor, and why there was no way to test the application was... the technical debt they had accumulated over the last 10 years, and which had not grown exponentially in recent years.

When I showed the plan's draft to the company's CTO, he asked me not to mention the technical debt at all. The CEO was freaking out every time someone mentioned that there was technical debt in the code and that was why none of the plans were being met. He told me that the first time technical debt was explained to him, he paid attention and accepted the schedule slippage. The second time, he made a face and asked for more commitment. And the third time, he interrupted the person who was talking and said, "Stop, stop, stop, stop. Let me guess, it's technical debt again, right? Don't give me that bullshit excuse again." From then on, it was not allowed to talk about technical debt in any meeting with this CEO.

In the last years some voices, usually from C-level, have expressed their discontent with the technical debt metaphor, even referring to it as a way technical teams have to avoid explaining their work or an excuse to ship faulty code. Also, in some cases when someone talks about technical debt, it sounds like a complaint and an excuse to use new technology that may not add anything to the business. Even Ward Cunningham reales this video [2] to explain misconceptions behind the metaphor.

Author:

With over 20 years of hands-on experience in programming, Pablo Bouzada is a well-versed professional who has successfully transitioned into leadership roles, currently serving as an Engineering Manager in Viaplay Group. He is dedicated to fostering collaborative environments and helping individuals discover more effective ways of working together, navigating challenges such as remote team collaboration and conflict resolution. Pablo provides a fusion of technical knowledge and leadership insights, based on years of practical experience, exploring the intersection of technology, management, and the human diversity that drive successful teams.



<https://www.linkedin.com/in/pablo-bouzada/>

In my experience, there are three main reasons for this dealignment between technical and non-technical people:

First and foremost, usually when we are discussing with top-management about technical debt, it is too late. Because the team, the product, and the company are already suffering the consequences. Those issues are shown in some, of all, of the following:

- The most obvious issue is that technical debt increases the cost of ongoing maintenance, making it more difficult to predict release schedules.
- Carrying technical debt into production raises the risk of outages, security breaches, financial losses, and potential legal issues due to violations of Service Level Agreements (SLAs).
- Future refactoring becomes riskier and more expensive, as modifications to production code carry a higher chance of causing disruptions.
- Productivity declines and delivery slows down. This adds stress on teams and can lead to reduced motivation and higher staff turnover.
- The cumulative effects of technical debt result in increasingly fragile systems, making significant improvements challenging. Inconsistent design and poor UI/UX can cause users to experience degraded performance and limited functionality.

We can agree that those consequences are really bad for team morale, company reputation, and users' experience with the company's products. So, it's difficult to simplify all to a "debt" we should pay.

Second, the analogy between software and finance breaks down when it comes to how repayments work. When you borrow money, repayments are typically time-based, making it easy to track them (for example, €100 per month for 5 years at a 3% interest rate). If you don't make your payments on time, there are clear consequences.

In contrast, repaying technical debt operates differently. There is rarely a set timeframe for repayment, and the cost of that debt can vary significantly depending on the context in which it was incurred.

And third, technical debt is linked to huge costs in refactoring or rewriting projects that can take years to finish, often providing little value to the company.

Those issues make C-level people get mad because executives expect to see diagrams with lines going up and to the right. With green numbers and positive deltas. Moreover, executives expect that if someone brings a problem, they also propose a solution, a plan, objectives, and the budget to carry it out.

When we are in this situation, we have to go beyond the metaphor, because technical debt is a simplification in non-

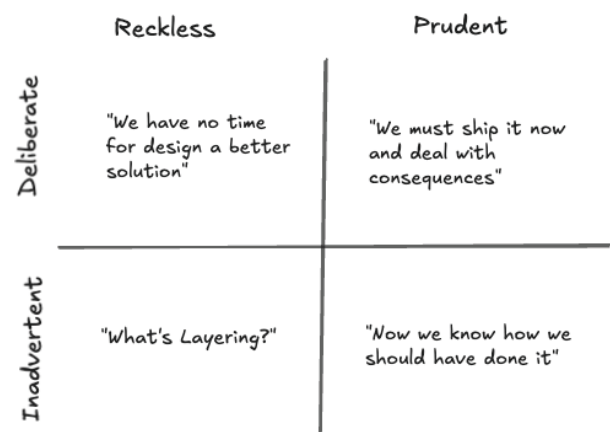
technical language. It is not the whole picture, but a way to explain the result of some decisions made in the past and how that affects current and future work. However, technical debt is not the main problem; it is an effect of deeper issues that need to be addressed. But to go deeper we have to understand the causes.

Not all technical debt is the same

Martin Fowler [3] set an interesting point of view on the technical debt metaphor adding an extra ingredient, the internal quality. In his words, "Software systems are prone to build up of cruft (deficiencies in internal quality) that make it harder than it would ideally be to modify and extend the system further. Technical debt is the extra effort that it takes to add new features, is the interest paid on the debt."

And, you know what, internal quality does not matter to customers, nor investors and sometimes neither to top management. But customers, investors and managers do care that new features come quickly. Similar to financial debts, if the debt in your code is not addressed, it can accumulate "interest," making it increasingly difficult to implement changes. Eventually, you may reach a point of "bankruptcy" where the code becomes so hard to maintain that you spend more time (and money) fixing issues than you do developing new features that add value.

We still do not have any causes here, but Fowler was the author of another brilliant reference in this field: the Technical Debt Quadrant [4], which explains the causes of different types of technical debt more explicitly.



Starting from the top/right:

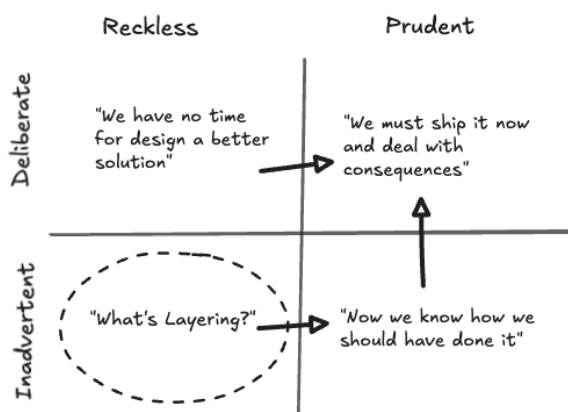
- **Prudent/Deliberate:** perhaps this is the best example of technical debt as originally defined by Cunningham. It is typically used to accelerate time to market, often going unnoticed by the organization. High-performing teams usually operate this way, testing their hypotheses and refactoring as soon as possible. In the words of Kent Beck: "Make it work. Make it right. Make it fast."
- **Prudent/Inadvertent:** it primarily occurs in a team that is growing and improving. This might be the most natural

way of learning, but it requires us to take responsibility for the mistakes the team made and learn from them. Instead of blaming others, we should focus on understanding the lesson, adding it to the team's knowledge base, and moving forward.

- **Reckless/Deliberate:** the main cause would be too much pressure to release something new. Teams instead of challenging business requests back, lower the bar on technical aspects. Eventually, they fall into a spiral because they do not have time to refactor and improve the code because the next new thing is in the backlog with a tight deadline already set.
- **Reckless/Inadvertent:** This is the worst-case scenario. The team lacks knowledge in both technical aspects and business topics, leading them to make questionable decisions. They are not learning from their mistakes and are simply adding more and more code in an attempt to fix the bugs.

How to pay technical debt properly?

We should assume some kind of technical debt, even in the best case: Prudent/Deliberate. And set up it as the standard of the internal quality we want to achieve in every team. But let's be honest, there is no way to move from the bottom left (Reckless/Inadvertent) to the top right (Prudent/Deliberate), we have to transition from the other cases in the quadrant to achieve it. Also is crucial to take into account that technical debt is the result of socio-technical issues in the organization. You have to fix them first, or at least at the same time, you are paying the technical debt.



The easy one is to move from Prudent/Inadvertent to Prudent/Deliberate:

- Assume some technical debt is part of the journey.
- Protect the team from blaming by making mistakes.
- Encourage the team to challenge business requests that compromise technical solutions.
- Avoid solutions driven by individual efforts.
- Run away from over-engineering.

How to move from Reckless/Deliberate to Prudent/Deliberate:

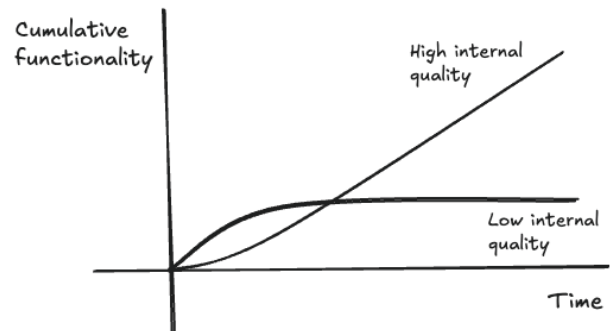
- Reduce the WIP (Work In Progress).
- Set % of time dedicated to clean off/refactor.
- Educate stakeholders about the impact of technical debt on the product in the long term.
- Follow the "Boy Scout rule": Leave the campground cleaner than you found it.

Moving from Reckless/Inadvertent to Prudent/Inadvertent:

- Invest in hiring and training.
- Set and ensure the teams are following good engineering practices.
- Set guidelines to refactor legacy code.
- Improve the feedback loops inside the team (pull request, Pair programming, Mob programming) and outside the team (reviews, audits, and feedback from other teams).

Discuss about internal quality instead

Now we have arguments to talk with the C-level. The issue we are trying to solve is not paying technical debt, but increasing the internal quality of the systems. The goal is to maintain the ability to easily and safely implement changes in a system that is continually evolving. This system should exist in a context that is nearly free of technical debt. And then, technical teams could provide new functionality in a shorter time and with fewer bugs.



Rather than continually repeating the phrase „technical debt“ like an old, broken record, we should talk business language and refer to technical debt as operational risk: the risk of losses caused by faulty or failed processes, policies, systems, or events that disrupt business operations. We should focus on discussing:

- Systems most prone to critical failures and their potential costs (e.g., loss of sales, time spent fixing errors, loss of trust, low team morale, burnout).
- The impact of maintaining a system with low quality, performance, or maintainability.
- Actions to improve system resilience and prevent cost overruns.

All of the above will definitely cost money and require investment. So, be ready to do your numbers and present your plan based on strong arguments.

Extra-ball: Will AI help us to deal with technical debt?

In my opinion, it could help as a tool used to understand old legacy code, but if we use it for everything, it could be like adding more gasoline to the fire. As Kevlin Henney mentioned in his excellent talk: Technical neglect [5]: "Automating the creation of legacy code using LLMs, developers have redefined their job role as debugging and fixing such code."

Even more, if junior developers rely too heavily on LLMs to complete tasks without learning and understanding the work (or the code), they will never develop expertise or critical thinking. They will never be able to improve the code because they do not have the necessary skills.

Use AI with caution!

With the kind support of our partners. JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors



Silver Sponsors



Bronze Sponsors



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!

trusted by

ATU

EMPORIO ARMANI

CONRAD



Allianz

**BEST
{BUDDYS}**
IN CODE



Arrange a free discovery call here!

www.xdev.software

XDEV

MicroStream In-Memory Data Platform:

Next Generation Caching Boost Any Database Application

Caching - Searching - Indexing - ACID Persistence

Why Caching With MicroStream:

- Native Java Object Model. No Mappings or Conversions.
- Searching, Filtering & Indexing On Your Cache Layer.
- Uniquely Fast Microsecond Response & Query Time.
- Implement Any Java Logic in Your Cache Layer.
- ACID-Compliant Persistence.
- Distributed. SaaS. On-Prem.
- Integrates With Any Existing Application.
- Pure Java & Open Source.

Next Steps, We'll Guide You:

1. Book a Free Intro Call.
2. Get Started With a POC.
3. Train Your Team.
4. Build Your In-Memory Data Layer.

www.microstream.one/intro



www.microstream.one

JAVAPRO

**SUBSCRIBE TO OUR FREE
PDF EDITIONS.**

www.javapro.io



www.javapro.io