Edition DE 01 / 2025
JAVAPRO

Viele weitere Artikel online:
www.javapro.io

CORE JAVA • FRAMEWORKS & APIs • ARCHITEKTUR • CLOUD • KI

JAVAPRO

MAGAZIN FÜR PROFESSIONELLE JAVA ENTWICKLUNG

GenAI
Projekte mit Java

Super Earlybird!

JCON
GenAI
NOV 20, 2025 in Ljubljana
www.genai.jcon.one

38 KI-TOOLS FÜR JAKARTA EE

49 ENTWICKLUNG VON KI-ANWENDUNGEN MIT SPRING AI

59 LOKALE LLMS MIT OLLAMA & OPEN WEBUI

88 VON REACTIVE STREAMS ZU VIRTUAL THREADS

151 NATIVE BUILDS & SERVERLOSE DEPLOYMENTS AUF KUBERNETES

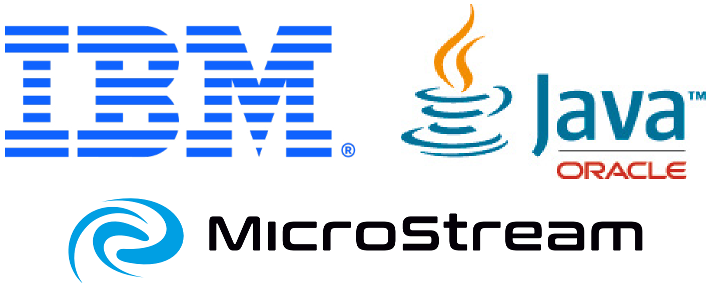
172 MODULARER MONOLITH MIT SPRING BOOT & MAVEN

www.javapro.io

Mit freundlicher Unterstützung durch unsere Partner.

JAVAPRO PARTNER NETZWERK

Platin Sponsoren



Gold Sponsors

Silber Sponsoren

Bronze Sponsoren

JAVAPRO

Verlag:
JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
Deutschland

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

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

Redaktion:
info@javapro.io

Design, Layout & Druck:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
Deutschland

Copyright (c) 2025
Impuls Mediengruppe GmbH

Alle Rechte vorbehalten.

Java(TM) ist eine eingetragene Marke der Oracle Corporation.

Javapro ist ein unabhängiges Magazin und ist nicht gefördert durch die Oracle Corporation.

Benannte Artikel reflektieren nicht zwingend die Meinung der Redakteure.

Die in der Ausgabe verwendeten Grafiken stammen von lizenzfreien Seiten wie Pixabay und Unsplash, wurden von den Autoren zur Verfügung gestellt oder mit der Hilfe von künstlicher Intelligenz generiert.

Java im Wandel – Kontinuität trifft Innovation

Java steht seit fast drei Jahrzehnten für Stabilität, Plattformunabhängigkeit und eine außergewöhnlich aktive Community. In dieser Zeit hat sich die Sprache kontinuierlich weiterentwickelt, ohne ihre Grundprinzipien zu verlieren. Heute befindet sich das Ökosystem in einer seiner dynamischsten Phasen. Neue Sprach- und Plattformfeatures, Veränderungen in Architektur- und Deployment-Strategien sowie die Integration von Künstlicher Intelligenz prägen die Diskussionen – und eröffnen Entwicklerinnen und Entwicklern neue Möglichkeiten.

Architektur jenseits von Trends

Ein zentrales Thema dieser Entwicklung ist die Verschmelzung bewährter Konzepte mit modernen Paradigmen. Die Java-Welt profitiert von einer breiten Auswahl an Werkzeugen, Frameworks und Bibliotheken, die gleichermaßen robuste wie flexible Systeme ermöglichen. Ob klassischer Monolith, modulare Architektur oder verteilte Microservices – die Wahl hängt mehr denn je von den konkreten Anforderungen ab, nicht von Trends. Zukunftsfähigkeit entsteht, wenn Innovation und betriebliche Realität im Einklang stehen.

Performance, Effizienz und Nachhaltigkeit

Performance bedeutet heute weit mehr als kürzere Ausführungszeiten. In einer Zeit flexibel skalierbarer Cloud-Ressourcen treten auch Startzeiten, Infrastrukturkosten, Energieverbrauch und Entwicklerfreundlichkeit in den Vordergrund. Moderne JVM-Features, optimierte Deployments und serverlose Architekturen bieten hier großes Potenzial. Trotz immer besserer IDE-Tools und Frameworks wie Spring, bilden auch weiterhin Erfahrung und die Bereitschaft, Technologien zu verstehen und zu hinterfragen, die Grundlage für eine erfolgreiche und nachhaltige Anwendungsentwicklung mit Java.

KI als integraler Bestandteil

Ein weiterer, kaum zu übersehender Treiber ist die Integration von KI in Unternehmensanwendungen. Die Java-Plattform hat sich hier schnell angepasst: Frameworks und Toolchains ermöglichen es, große Sprachmodelle lokal oder in der Cloud zu nutzen, in bestehende Workflows einzubetten und mit Sicherheits- sowie Qualitätsanforderungen zu verbinden. Damit rückt ein Ziel in greifbare Nähe, das noch vor wenigen Jahren wie Zukunftsmusik klang: KI nicht nur als Prototyp oder Zusatzfunktion, sondern als festen Bestandteil produktiver Systeme einzusetzen.

Innovation auf solidem Fundament

Trotz aller Neuerungen bleibt Qualität unverzichtbar: wartbarer Code, verlässliche Sicherheitsmechanismen und eine klare Architektur sind das Fundament jeder Innovation. Statische Analyse, automatisierte Tests und die Lehren aus Sicherheitsvorfällen der Vergangenheit mahnen, dass Stabilität und Vertrauen erarbeitet werden – und dass Fortschritt nur dann nachhaltig ist, wenn er auf solidem Unterbau steht.

Die Zukunft in unseren Händen

Diese Ausgabe ist kein Streifzug durch einzelne Technologien, sondern ein Blick auf die Strömungen, die Java aktuell prägen. Sie zeigt, Java Developer die Sprache und Plattform nutzen können, um robuste, performante und zukunftssichere Anwendungen zu schaffen – in einer Zeit, in der sich Anforderungen schneller ändern als je zuvor. Java steht 2025 an einem Punkt, an dem Tradition und Innovation nicht im Widerspruch stehen, sondern sich gegenseitig beflügeln. Die Plattform bleibt ihrem Kern treu, öffnet sich aber konsequent für Neues. Für alle, die mit Java entwickeln, bedeutet das: Die Möglichkeiten waren selten so groß – und die Verantwortung, sie sinnvoll zu nutzen, ebenso.

Markus Kett
Chefredakteur
JAVAPRO

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

07
JAVAPRO
Kino, Code, Community: Die JCON EUROPE 2025 setzt neue Maßstäbe für Java-Events
von JAVAPRO Team

13
CORE JAVA
30 Jahre Java – Wie sich die Sprache entwickelt hat
von Sebastian Hempel

27
CORE JAVA
Die lange Geschichte von Log4j
von Christian Grobmeier

38
AI & ML
KI-Tools für Jakarta EE
von Gaurav Gupta

49
AI & ML
Entwicklung von KI-Anwendungen mit Spring AI
von Timo Salm

59
AI & ML
Lokale LLMs mit Ollama und Open WebUI
von Jean-Claude Branschen

70
PERFORMANCE
Hitchhiker's Guide to Java Performance
von Ingo Düppe

88
PERFORMANCE
Von Reactive Streams zu Virtual Threads
von Adam Warski

108
PERFORMANCE
So beschleunigen sie existierende Deployments mit den richtigen JVM-Features
von Dmitry Chuyko

137
ARCHITECTURE & MICROSERVICES
Überwinde 20 Jahre mit Migration Engineering
von Merlin Bögershausen

151
ARCHITECTURE & MICROSERVICES
Nachhaltige Softwareentwicklung in Java – Native Builds und serverlose Deployments auf Kubernetes
von Marius Stein & Vishal Shanbhag

172
ARCHITECTURE & MICROSERVICES
Modularer Monolith mit SpringBoot & Maven
von Max Beckers

179
TESTING & QUALITY
Die Kunst der statischen Codeanalyse
von Martin Toshev

#JAVAPRO #JCON

Kino, Code, Community: Die JCON EUROPE 2025 setzt neue Maßstäbe für Java-Events

Von praxisnahen Workshops über Live-Coding im Kinosaal bis zur Mentoring-Revolution – die JCON EUROPE 2025 in Köln war ein Fest für Java-Fans aus aller Welt.

Ein globales Treffen der Java Community

Vom 12. bis 15. Mai 2025 wurde Köln zum Epizentrum der internationalen Java-Community. Zur zehnten Ausgabe der JCON EUROPE pilgerten Entwickler aus über 60 Ländern und fünf Kontinenten in den Cinedom, um gemeinsam 30 Jahre Java zu feiern. Die Stimmung? Euphorisch, gemeinschaftlich – und immer mit einer Prise Humor.

„30 years of Java. 10 years of JCON. The spirit of celebration was everywhere," schrieb ein Teilnehmer auf Social Media treffend. Oder wie ein anderer es formulierte: „JCON feels more like a reunion of old friends."

Ehrung durch Oracle: JCON feiert 10 Jahre – und wird gefeiert

Ein besonderer Moment der Konferenz: Oracle zeichnete das JCON-Team für zehn Jahre kontinuierliches Engagement in der Java-Community

aus. Die Auszeichnung wurde persönlich von Sharat Chander (Java Community Lead bei Oracle) an Markus Kett (Founder JCON) und Richard Fichtner (Co-Organisator JCON) überreicht.

Diese Ehrung unterstreicht die zentrale Rolle, welche die JCON in der weltweiten Java-Community einnimmt – nicht nur als Konferenz, sondern als Plattform für Innovation, Austausch und Weiterbildung.

Workshops: Intensiver Einstieg in die Themenwelt

Bereits am Montag, dem ersten Konferenztag, startete die JCON mit einem vollen Tag praxisorientierter Workshops. In kleinen Gruppen wurde konzentriert gearbeitet, programmiert, getestet und diskutiert. Die Themen reichten von Testing (Christian Stein, Marc Philipp), Java-Performance und Caching (Florian Habermann, Christian Kuemmel), KI-gestützter Anwendungsentwicklung (Marta Tolosa, Sydney Nurse, Sven Ruppert) bis hin zu effizientem Deployment und Startup-Verhalten (Mark Stoodley). Parallel dazu fand der ganztägige Java Luminaries Summit, ein Treffen führender Java Spezialisten, in einem eigenen Raum statt.

Die Workshops boten eine ideale Gelegenheit, tiefer in aktuelle Technologien einzutauchen und direkt mit den führenden Köpfen der Szene zu arbeiten. Besonders geschätzt: die Mischung aus Theorie, praktischen Übungen und Raum für individuelle Fragen.

Viele Teilnehmende nutzten den Workshop-Tag als Einstieg in die Konferenzwoche – und berichteten von „wertvollen Aha-Momenten“ und „direkt umsetzbaren Learnings für den Alltag im Team“.

KI & GenAI: Das große Zukunftsthema für Java

KI ist jetzt auch in der Java-Community angekommen – und das Potenzial für Java ist gigantisch. Gerade im Enterprise-Umfeld, wo Java traditionell führend ist, treffen modernste KI-Technologien auf riesige, gewachsene Datenmengen – ein Schatz, der nun gehoben wird.

Kein Wunder also, dass Generative AI eines der dominierenden Themen auf der JCON 2025 war. Zahlreiche hochkarätige Vorträge und Sessions zeigten, wie sich GenAI in bestehende Java-Landschaften

integrieren lässt: von LangChain4J über Vektor-Embedding-Modelle bis hin zu AI-gestützter Anwendungsentwicklung und Optimierung von Geschäftsprozessen.

Die Botschaft war klar: Java und GenAI ergänzen sich perfekt – und bieten gemeinsam enormes Potenzial für die nächste Generation von Enterprise-Anwendungen.

Konferenz mit Kino-Glamour

Statt nüchterner Konferenzräume flimmerten Live-Coding-Sessions über riesige Kinoleinwände. Die Idee, Tech-Talks mit Q&A über Slido im Kinosaal zu inszenieren, hat sich längst etabliert – und begeistert weiterhin.

Die JCON EUROPE 2025 bot an jedem Konferenztag eine inspirierende Keynote, welche die Vielfalt und Zukunftsfähigkeit der Java-Welt unterstrich.

Am Dienstag eröffnete Markus Kett die Konferenz mit einer Keynote über „Java's Ignored Potential". Darin zeigte er, wie sich Performance-Probleme in datenlastigen Anwendungen durch Java-native In-Memory-Datenverarbeitung drastisch reduzieren lassen – ein Thema, das bei vielen auf großes Interesse stieß. Am Mittwoch nahm Sharat Chander, das Gesicht der internationalen Java Community, die Bühne mit „Happy Birthday, Java!" ein – ein ebenso emotionaler wie informativer Rückblick auf drei Jahrzehnte Java-Geschichte, die mit Projekten wie dem Mars Rover oder den Olympischen Spielen längst globale Spuren hinterlassen hat. Den Abschluss bildete am Donnerstag Mark Stoodley (Chief-Architekt Java bei IBM) und Markus Kett, mit „Rethinking Microservice Persistence". Ein revolutionäres Konzept für Datenbank-Architekturen, Datenbanksysteme basierend auf einer Microservice-Architektur - weg vom Datenbankmonolithen, mit bis zu 80% Einsparungspotentialen bei Rechenleistung, Energieverbrauch, CO2 Emissionen und Cloud-Kosten.

Darüber hinaus überzeugte das Programm durch technische Tiefe, starkes Live-Coding und eine durchweg engagierte Community. Cay Horstmann begeisterte mit tiefen Einblicken zu Virtual Threads und Project Valhalla, Alina Yurenko führte in praxisnahen Sessions durch

die Welt von GraalVM, während François Martin, Lize Raes und Andres Almiray Themen wie Testing, Security und generative KI mit Java lebendig und praxisorientiert aufbereiteten. Ergänzt wurde das Programm durch Sessions von Szenegrößen wie Brian Vermeer, Sandra Ahlgrimm, Simon Martinelli, Ana-Maria Mihalceanu und vielen weiteren, die das Kinoprogramm in ein echtes Java-Festival verwandelten und von den Teilnehmern gefeiert wurden.

Für alle, die nicht live dabei sein konnten: Alle Sessions und Keynotes sind auf dem offiziellen JAVAPRO-YouTube-Kanal verfügbar.

Mentoring Hub: Tiefergehende Gespräche statt Frontalvortrag

Ein echtes Highlight war das neue Mentoring-Format: Im Mentoring Hub trafen erfahrene Entwickler auf Nachwuchstalente und Berufseinsteiger. In kleinen Runden wurde diskutiert, gecoacht und inspiriert – ein Format, das über das klassische Konferenzmodell hinausging. Statt passivem Zuhören ermöglichte es direkte Gespräche mit erfahrenen Java-Profis.

Ob bei Sessions mit Bruno Souza, Gesprächen zu „Next Steps for Developers“ oder der Frage, wie man ein „mature developer“ wird - die Resonanz war durchweg positiv. „It added a kind of depth and connection that went beyond the usual conference experience“, so eine Teilnehmerin.

Ein Statement in Print: JAVAPRO zurück auf Papier

Ein weiteres, kaum übersehbares Highlight war das Comeback der JAVAPRO in Printform. Die Sonderausgabe „30 YEARS OF JAVA“ war in kürzester Zeit restlos vergriffen – nach nur zwei Stunden war kein einziges Exemplar mehr am Stand verfügbar. Die Rückkehr der JAVAPRO als Printmedium wurde von vielen Teilnehmenden nicht nur als Überraschung, sondern als starkes Zeichen gewertet: Fachjournalismus für Entwickler ist relevant wie eh und je.

Wer leer ausging, kann die Artikel unter javapro.io nachlesen – sie werden sukzessive online veröffentlicht.

Die limitierte Special Editioin wird zudem auf ausgewählten Konferenzen verteilt und beinhaltet Artikel zu Core Java, GenAI, Microservices, Architektur, Frameworks& API, Testing, Security, Cloud, Rückblicke auf 30 Jahre Java-Entwicklung und Zukunftsthemen wie „GenAI mit Java" oder neue JVM-Sprachen. Wer darauf nicht warten möchte kann die kostenlosen PDF Ausgaben abonnieren.

Community-Spirit & Gespräche bis spät in die Nacht

Nicht nur fachlich, auch menschlich setzte die JCON 2025 Maßstäbe. Der „Hallway Track", das spontane Networking zwischen Sessions, sowie die 1:1 Speaker Meetings wurde rege genutzt. Viele sprachen von einer „Konferenzfamilie" statt reinem Networking.

Ein gesellschaftlicher Höhepunkt war das VIP-Dinner am Mittwochabend – mit Vertretern der Java-Community, Speakern, Sponsoren und dem Orga-Team. In entspannter Atmosphäre wurde diskutiert, angestoßen und über die Zukunft von Java philosophiert. Das Dinner unterstrich den verbindenden Charakter der Veranstaltung – und bot Raum für persönliche Gespräche auf Augenhöhe.

Natürlich wurde auch gefeiert: 30 Jahre Java – mit allem, was dazugehört. In bester Stimmung wurde auf drei Jahrzehnte Innovation und Community angestoßen – inklusive einer eigens gestalteten Jubiläumstorte, die beim VIP-Dinner angeschnitten und gemeinsam genossen wurde. Ein süßer Moment, der das Gemeinschaftsgefühl noch einmal spürbar machte.

Neben Fachvorträgen und tiefgründigen Diskussionen sorgten auch kleine, verspielte Details für Begeisterung: Besonders beliebt waren die bunten Ribbons. Wer mit einem Regenbogen aus Badge-Erweiterungen wie z.B. „JVM", „Star Trek" und „Maven" oder ähnlichen unterwegs war, hatte entweder Humor, ein breites Technikwissen – oder einfach eine gute Woche.

Jedes Jahr mit Spannung erwartet: die neuen JCON T-Shirts. Auch das Jubiläumsshirt zum zehnjährigen Bestehen war heiß begehrt und wurde von vielen Teilnehmenden direkt angezogen. Für manche haben diese längst Sammlerwert, und wer sich ein limitiertes Exemplar gesichert hat,

besitzt nun ein Stück JCON-Geschichte.

JCON goes to the USA

Doch das war längst nicht alles. Erstmals offiziell angekündigt wurde es auf der Bühne in Köln und sorgt bereits für Vorfreude: Mit der ersten US-Ausgabe JCON @ IBM TechXchange vom 6. bis 9. Oktober 2025 in Orlando, Florida geht das Erfolgsformat JCON erstmals über den Atlantik. Ein weiterer Meilenstein in der Entwicklung – und ein klares Zeichen dafür, dass der Spirit der Community keine Grenzen kennt.

Ein Jubiläum, das verbindet – und Lust auf mehr macht

Zehn Jahre JCON, 30 Jahre Java, vier intensive Tage. Die JCON EUROPE 2025 war ein voller Erfolg – sowohl organisatorisch als auch inhaltlich. Sie zeigte eindrucksvoll, wie modern, vielfältig und zukunftsgewandt die Java-Community agiert. Ein riesiges Dankeschön an alle Teilnehmer, Volunteers und das gesamte Orga-Team sowie ganz besonders an unsere fantastischen Speaker, Partner, Aussteller und Sponsoren, welche die JCON EUROPE 2025 möglich gemacht haben.

Die Mischung aus Innovation, Praxisnähe, Mentoring und echter Community-Bindung macht die JCON zu weit mehr als einer Entwicklerkonferenz – sie ist Heimat für Java-Begeisterte. Wer nicht dabei war, hat definitiv etwas verpasst.

Gute Nachrichten: Das Recap-Video ist online, und die Session-Videos werden nun schrittweise veröffentlicht. Viele Vorträge können bereits gestreamt werden – eine hervorragende Gelegenheit, die Highlights noch einmal anzusehen oder verpasste Sessions zu entdecken.

Save the Date

Die nächste JCON EUROPE findet vom 20. bis 23. April 2026 erneut im Multiplexkino Cinedom in Köln statt.

> Zurück zum Inhaltsverzeichnis

#JAVAPRO #COREJAVA

30 Jahre Java - Wie sich die Sprache entwickelt hat

Autor:

Sebastian Hempel ist ein selbstständiger Berater und Trainer aus dem Norden Bayerns. Seit 2003 unterstützt er Kunden bei der Entwicklung und dem Betrieb von Unternehmenssystemen. Sein Hauptfokus liegt auf der Entwicklung von Systemen in Java (JakartaEE und Quarkus) auf Linux-basierten Systemen.

https://www.linkedin.com/in/sebastian-hempel-07a50a224/

Wir feiern 30 Jahre Java. Zeit für einen Rückblick wie sich Java als Programmiersprache entwickelt hat. Wir werden sehen, wie Java verschiedene Paradigmen der Softwareentwicklung integriert hat, ohne seine klare Struktur zu verlieren. Es ist faszinierend, dass man trotz all dieser Veränderungen in Java immer noch genauso programmieren kann wie vor 30 Jahren.

Als Java Mitte der 90er Jahre aufkam, war die objektorientierte Programmierung das dominierende Paradigma. Dies spiegelt sich in Java wider, das von Anfang an eine „rein“ objektorientierte Sprache war. „Alles ist ein Objekt“ kann wörtlich genommen werden. Mit Ausnahme der Primitiven ist alles von der Basisklasse „Objekt“ abgeleitet. Anfangs hielt sich die Sprache strikt an dieses Paradigma. Dies änderte sich im

Laufe der Zeit, als die Sprache immer mehr Elemente des funktionalen Programmierparadigmas übernahm. Diese Entwicklung lässt sich auch in anderen Programmiersprachen beobachten.

Dieser Artikel wird nicht jede einzelne Änderung beschreiben, die seit 1995 in Java vorgenommen wurde. Ich werde die Änderungen herausgreifen, die für mich und meine tägliche Arbeit am wichtigsten sind. Daher sind nicht alle wichtigen Änderungen der Sprache in diesem Artikel enthalten. Ich werde auch Verbesserungen in der Standardbibliothek von Java betrachten. Die Bibliothek ist ein wesentlicher Bestandteil und kann für mich nicht von der Sprache selbst getrennt werden.

Collections

Die Sprache selbst unterstützt das Konzept eines Arrays. Ein Array speichert Elemente, auf die über einen Index zugegriffen werden kann. Die Größe eines Arrays ist unveränderlich und wird bei der Erstellung des Arrays festgelegt. Arrays können zum Speichern von Primitiven und Objekten verwendet werden.

Um die Arbeit mit dynamischen Arrays zu ermöglichen die automatisch wachsen, wenn neue Elemente hinzugefügt werden, wurde mit JDK 1.0 die Klasse Vector eingeführt. Diese Klasse ist weiterhin verfügbar, sollte aber nicht in neuen Projekten verwendet werden. Sie wurde 1998 in das mit Java 1.2 eingeführte Collection-Framework integriert.

Man erstellt eine Instanz der Klasse Vector und fügt Elemente mit der Methode addElement hinzu. Um ein Element an einem bestimmten Index hinzuzufügen, kann die Methode insertElementAt verwenden. Diese Methode stellt sicher, dass das Element an dem angegebenen Index eingefügt wird und alle vorhandenen Elemente um eine Position verschoben werden.

Um alle Elemente eines Vektors zu durchlaufen, erhält man mit der Methode elements eine Enumeration. Diese Enumeration kann verwendet werden, um alle Elemente mithilfe einer while-Schleife zu durchlaufen. Mit der Methode hastMoreElements prüft man, ob noch nicht besuchte Elemente vorhanden sind. Mit der Methode nextElement der Enumeration gelangen wir zum nächsten 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());
        }
    }
}
```

Neben der Vector-Klasse wurde mit JDK 1.0 auch die Dictionary-Klasse eingeführt, um einem Wert einen Schlüssel zuzuweisen. Dies ermöglicht den Zugriff auf das Element über einen Schlüssel. Die Klasse ermöglicht das Abrufen einer Enumeration aller Schlüssel oder aller Werte des Dictionarys. Die Wiederverwendung eines vorhandenen Schlüssels ersetzt den alten durch den neuen Wert. Es ist auch möglich, den Wert für den Schlüssel zu entfernen.

```
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 führte 1998 ein völlig neues Framework für die Arbeit mit Collections ein. Das Framework unterstützt verschiedene Collection-Typen, wie beispielsweise List, Set oder Map.

Die Verwendung dieser neuen Collections unterscheidet sich nicht von der Verwendung von Vector oder Dictionary. Die Methodennamen sind kürzer. Anstelle der Enumeration verwenden wir nun einen Iterator, um alle Elemente einer Collection zu durchlaufen. Intern ist die Implementierung der Collections performanter. Die Vector-Klasse synchronisierte alle Operationen. Diese Funktion wurde mit der Umstellung auf das Collection-Framework abgeschafft.

```
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

Wir müssen Elemente, die wir aus einer Collection abrufen, weiterhin vom generischen Typ „Object“ in den konkreten Typ umwandeln, den wir in der Collection speichern. Die Typumwandlung kann zur Laufzeit zu einer ClassCastException führen. Der Compiler kann beim Erstellen des Objektcodes nicht auf den korrekten Typ prüfen.

Java 5 führte 2005 das Konzept der Generics ein. Es ermöglicht die Erweiterung des Java-Typsystems. Das Collection-Framework unterstützt Generics, um den Typ der in einer Collection gespeicherten Objekte festzulegen. Mit Generics kann der Compiler den Typ des Elements zur Kompilierzeit prüfen.

Wir erhalten einen typsicheren Iterator, der den Typ des in einer Collection gespeicherten Elements zurückgibt.

```
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);
        }
    }
}
```

Java 5 brachte uns jedoch nicht nur Generics. Es führte auch einen neuen Typ der For-Schleife ein. Die For-Each-Schleife kann über jedes Element eines Arrays oder eines Iterable iterieren, das jede Standard-Collection-Klasse implementiert. Wir müssen weder die hasNext-Methode überprüfen noch die next-Methode verwenden, um das

nächste Element zu erhalten. Zusammen mit der Iterable-Schnittstelle wirken Collections nun wie ein direktes Element der Sprache.

```
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 aus dem Jahr 2011 verbesserte die Typinferenz für die Erstellung generischer Instanzen. Es handelt sich um eine kleine Erweiterung, die Duplikate im Code beseitigt. Der Typ des generischen Typs muss nur noch im Konstruktor definiert werden. Bei der Definition der Variablen kann der generische Typ weggelassen werden.

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

Arbeiten mit Datumswerten

Java unterstützt die Arbeit mit Datum und Uhrzeit von Anfang an. Die Klasse Date wurde in Java 1.0 eingeführt. Sie ist eine Art Wrapper für den Unix-Zeitwert und berücksichtigt auch Zeitzonen. Sie ist zudem die Klasse mit den seit Jahrzehnten am längsten veralteten Methoden. Alle Konstruktoren, Getter und Setter wurden zugunsten der neuen Klasse Calendar, die 1997 in Java 1.1 eingeführt wurde, verworfen.

```
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“);
    }
}
```

Die Klasse Calendar kann sowohl mit Zeitzonen als auch mit dem Kalendersystem arbeiten. Nach dem Einstellen der Zeit können wir mit der Methode get Teile des Zeitstempels wie Monat oder Tag abrufen.

Wir mussten bis Java 8 (2014) warten, um eine komplett neue API für Datum und Uhrzeit zu erhalten. Java integriert die bisher verfügbare Joda-Time-Bibliothek in das JDK. Die API führt neue Klassen für die Verarbeitung von Datum und Uhrzeit in der aktuellen (lokalen) Zeitzone ein: LocalTime, LocalDate und LocalDateTime. Für die Arbeit mit Zeitzonen stehen uns nun die Klassen ZonedDateTime zur Verfügung. Die Klasse Instant ermöglicht die Arbeit mit exakten Zeitstempeln in Nanosekunden.

Die Klassen unterstützen Methoden zur Datumsberechnung und zur Berechnung von Differenzen zwischen Datum und Uhrzeit. Es gibt auch Methoden zum Abrufen oder Setzen bestimmter Felder für ein Datum oder eine Uhrzeit. Die Schnittstellen der neuen Klassen unterstützen die Verkettung mehrerer Operationen (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));
    }
}
```

Die Entwicklung von Switch

Die Switch-Anweisung war von Anfang an vorhanden. Die erste Version konnte nur mit Integern verwendet werden. Sie trug dazu bei, Strukturen mit if-else if-else besser zu implementieren. Die Fallthrough-Funktionalität wurde von anderen Sprachen wie C übernommen.

```
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“);
        }
    }
}
```

Die erste Änderung erfolgte 2011 in Java 7. Neben Integern können wir mit der Switch-Anweisung nun auch zwischen verschiedenen Strings und Enumerationswerten auswählen. Dies machte die Switch-Anweisung noch nützlicher.

Viele Verbesserungen der Switch-Anweisung und des nun möglichen Switch-Ausdrucks wurden 2020 in Java 14 eingeführt. Mehrere Case-Werte können in einem einzigen Case bereitgestellt werden. Die größte Änderung war die Möglichkeit, Switch als Ausdruck zu verwenden. Werte aus einem Switch-Block können auf verschiedene Arten zurückgeben werden. Es gibt eine neue Anweisung „yield", um einen Wert zurückzugeben, oder man verwendet den Pfeiloperator, um eine Art „Mapping" zu definieren.

```
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);

    }

}
```

Musterverarbeitung / Pattern Matching

Die Verarbeitung von Pattern wurde in mehreren Schritten eingeführt. Die Funktionalität wurde zunächst in funktionalen Programmiersprachen implementiert und bald auch in anderen Sprachen wie Scala oder F#. Für Java wurde die erste Implementierung einer Mustervergleichsart mit Java 14 im Jahr 2020 eingeführt. Anstatt nur mit dem Operator „Instanceof" auf einen bestimmten Typ zu prüfen, kann der gecastete Wert nun in einem Schritt einer neuen Variable zugewiesen werden.

```
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());
        }
    }
}
```

Ein wesentlich größerer Schritt war die Einführung des Mustervergleichs für Switch mit Java 21 im Jahr 2023. Er ermöglicht die Auswahl von Fällen basierend auf dem Typ des Arguments. Neben dem eigentlichen Vergleich steht der gecastete Wert als neue Variable zur Verfügung. Nach dem Vergleich mit einem Typ können zusätzliche Ausdrücke verwendet werden, um die Auswahl basierend auf anderen Bedingungen zu verfeinern.

```
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);
    }
}
```

Nicht nur beim Mustervergleich, sondern auch bei einer normalen Switch-Anweisung kann ein Fall für Null hinzugefügt werden. Dadurch kann eine NullPointerException umgangen werden, wenn einer Switch-Anweisung ein Nullwert zugewiesen wird.

Neue Strukturen - Records

Java 16 (2021) ermöglicht die Definition unveränderlicher Datenklassen ohne umfangreichen Boilerplate-Code. Bei Records müssen lediglich Typ und Name der Felder definiert werden. Der restliche Code wie equals, hashCode, toString, Getter und öffentliche Konstruktoren werden vom Compiler erstellt.

Getter können wie normale Klassen verwendet werden. Der Unterschied ist das fehlende get vor jedem Feldnamen. Man schreibt einfach den Namen des Felds, auf das wir zugreifen möchten, z. B. name(), um das Namensfeld eines Records abzurufen. Records können durch statische Variablen und Methoden erweitert werden.

```
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());
    }
}
```

Da Records unveränderlich sind – sie haben keine Setter – eignen sie sich ideal als Datenobjekte. In vielen aktuellen Quelltexten wird hierfür die Lombok-Bibliothek verwendet. In neueren Projekten versuche ich selbst, Lombok durch Records zu ersetzen..

Verarbeitung von Daten-Streams

Eine der größten Neuerungen in Java 8 aus dem Jahr 2014 war die Unterstützung von Lambda-Ausdrücken. Anstatt anonyme innere Klassen zu definieren, gibt es nun Funktionen als neuen Typ. Wir können Funktionen/Lambdas definieren und an andere Methoden übergeben, die sie aufrufen.

Lambdas nutzen funktionale Interfaces. Schnittstellen dieses Typs bestehen aus einer (abstrakten) Funktion. Das Paket java.util.function enthält viele vordefinierte funktionale Schnittstellen. Die Annotationen stellen sicher, dass beim Hinzufügen einer weiteren Funktion zu unserer Schnittstelle kein Kompilierungsfehler auftritt.

```
package de.ithempel.java30;
public class LambaExample {
    @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);
    }
}
```

Die Java-Bibliothek selbst nutzt Lambdas, indem sie eine Streaming-API für die Verarbeitung von Datensequenzen bereitstellt. Streams sind vom Map-Reduce-Programmiermodell inspiriert, um große Datenmengen parallel und verteilt zu verarbeiten. Die von Streams bereitgestellten Operationen sind auch aus der funktionalen Programmierung bekannt. In Java bietet ein Stream verschiedene Operationen. Diese Operationen sind an eine Stream-Pipeline gekoppelt.

Die Streaming-API bietet Operationen wie Filter, um Elemente unter bestimmten Bedingungen auszuwählen. Mit der Map-Operation können wir ein Element transformieren. Finaloperationen können einfach über alle Elemente iterieren, wie die Operation „forEach". Es gibt auch Collector-Funktionen, um alle Elemente eines Streams in einer Collection zu sammeln.

Eine Pipeline kann parallel an einem Stream arbeiten. Mit der Operation „parallel" können wir die Arbeit der Pipeline auf mehrere Threads verteilen.

```
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));
    }
}
```

Die Streaming-API unterstützt Zwischen- und Terminaloperationen. Terminaloperationen lösen die Ausführung einer Stream-Pipeline aus. Zwischenoperationen werden nur ausgeführt, wenn eine Terminaloperation ausgeführt wird. Dies optimiert die Ausführung einer

Stream-Pipeline.

Abschließende Worte

Dies war meine kurze und schnelle Beschreibung der Entwicklung von Java. Wie bereits erwähnt, habe ich nicht alle Änderungen der Sprache der letzten 30 Jahren berücksichtigt. Ein wichtiger Bereich, den ich nicht erwähnt habe, sind die verschiedenen Mechanismen zur Handhabung von mehreren Threads.

Beim Blick auf die aktuelle Entwicklung sehe ich eine vielversprechende und interessante Zukunft für Java. Es stehen viele Änderungen bevor. Seit der Umstellung auf einen Release Train sehen wir ständig Vorschauen auf neue Funktionen und Änderungen in der Sprache und der Bibliothek.

> Zurück zum Inhaltsverzeichnis

#JAVAPRO #COREJAVA

Die lange Geschichte von Log4j

Autor:

Christian Grobmeier ist ein erfahrener Java-Entwickler, Open-Source-Verfechter und Vice President für Datenschutz bei der Apache Software Foundation. Er hat zu wichtigen Apache-Projekten beigetragen, darunter Log4j, Commons und Struts, und ist aktiv an Diskussionen über Java-Logging-Standards beteiligt. Als Geschäftsführer der Grobmeier Solutions GmbH unterstützt er Unternehmen beim Aufbau zuverlässiger, datenschutzbewusster Software. Mit großer Leidenschaft für nachhaltige Open-Source-Praktiken und Software-Sicherheit teilt er seine Erkenntnisse auf Konferenzen, in Fachveröffentlichungen und durch seine Texte über Software-Ethik und Führung.

https://www.linkedin.com/in/grobmeier/

„Logging ist die Kunst, ein System zu verstehen."

Software protokolliert was gerade passiert in Log-Dateien, und Entwickler durchsuchen sie in der Hoffnung, das zu finden, was sie brauchen. Das Problem dabei? Systeme protokollieren selten die richtigen Dinge. Doch ohne diese Dateien würden wir im Blindflug fliegen, ohne zu wissen, dass überhaupt etwas schiefgelaufen ist.

Dies ist die Geschichte von Apache Log4j - eines der ältesten Java-Frameworks, das heute noch verwendet wird. Ein Tool, das Entwicklern die Möglichkeit gab, ihre Systeme zu verstehen.

Log4j ist ein europäisches Projekt

Alles begann Mitte der 1990er-Jahre, als die Europäische Union (EU) noch in den Kinderschuhen steckte. Das SEMPER Projekt war eine frühe EU-Initiative zum Aufbau eines „sicheren Instrumentariums für den elektronischen Geschäftsverkehr". Es begann im September 1995, endete drei Jahre später und wurde mit rund 10 Millionen Euro finanziert. Damals war dies eine enorme Investition. Nicht nur wegen des Betrags, sondern auch, weil viele bezweifelten, dass das Internet selbst überleben würde, geschweige denn der elektronische Handel. Es wurden Prototypen gebaut und getestet, und am Ende war SEMPER Geschichte – und hatte Geschichte geschrieben. Heute wird oft beklagt, die EU sei unflexibel. Aber 1995 war SEMPER der Beweis dafür, dass die EU innovativ sein kann.

Im Jahr 1996 stellten Ceki Gülcü, N. Asokan und Michael Steiner die Idee der „hierarchischen Kategorien" vor. Damals war dies eine bahnbrechende Idee. Wie alle guten Programmierwitze war auch Logging damals streng binär: „ein" oder ‚aus'. Es gab keine Möglichkeit, nur datenbankbezogene Ereignisse zu protokollieren oder Benutzeraktivitäten zu verfolgen. Hierarchien änderten alles. Entwickler konnten nun Logging auf Paketebene aktivieren, wobei untergeordnete Pakete automatisch die Einstellungen ihrer Eltern übernahmen. Das SEMPER-Projekt finanzierte die erste Version von Log4j und erkannte die Bedeutung des Loggings für die Beobachtbarkeit des Systems.

Der Weg zu Open Source

Nach dem Ende von SEMPER brachte Ceki Gülcü das Paket in das IBM-Forschungslabor in Zürich, wo er die Entwicklung fortsetzte. 1998 oder 1999 hatte Log4j dort ein neues Zuhause gefunden. Das Projekt hätte in der Versenkung verschwinden können, aber IBM Zürich hielt es am Leben. Frühe Aufzeichnungen zeigen, dass Log4j zuerst durch IBMs alphaWorks Initiative verbreitet wurde. IBM nutzte alphaWorks, um aufstrebende Technologien zu fördern, und Log4j passte perfekt zu

dieser Vision. Moment, Moment, aufstrebende Technologien? Vergessen wir nicht, dass Java nur ein paar Jahre zuvor (1995) als Betaversion auf den Markt kam und nur ein Jahr später die Version 1.0 erreichte. Log4j wurde zusammen mit J2SE 1.2 (1998) und 1.3 (2000) entwickelt. Java 1.2 brachte uns das Collections-Framework, und 1.3 folgte mit JNDI. In der Ära von Lambdas würde die Arbeit mit Java 1.2 die meisten von uns zur Holzbearbeitung treiben. Diese Upgrades machten die Java-Entwicklung massiv einfacher.

Als das neue Jahrtausend anbrach, beobachteten viele den Himmel in Angst und fragten sich, ob Y2K Flugzeuge zum Absturz bringen würde. Währenddessen nahm die Zukunft von Log4j Gestalt an. Herr Gülcü verlegte Log4j zu SourceForge, nutzte die Vorteile der öffentlichen Infrastruktur und registrierte log4j.org. Die Domain sollten Sie jetzt aber nicht unbedingt in Ihren Browser eintippen–es führt heute zu einigen seltsamen Orten.

Community over code

Open Source bedeutet heute etwas anderes als in der Vergangenheit. Heute wird Open Source oft als Geschäftsmodell betrachtet. Wir machen uns Gedanken über die Finanzierung, verwaiste Projekte und Unternehmen, die ihren einst offenen Code zurückfordern. Open Source nimmt heute viele Formen an, aber damals war es ganz einfach.
Geld war nicht die treibende Kraft.
Open Source war wie Punk - teils politisches Statement, teils Rebellion, eine Bewegung für Softwarefreiheit.

Ich erinnere mich an ein Gespräch mit dem CEO eines Unternehmens, als ich noch recht jung war. Er erklärte mir, wie sehr er freie Software verabscheue. Seiner Meinung nach würde sie ihn Geld kosten, weil Menschen ihre Arbeit kostenlos weitergäben. Ich war sprachlos. Wie sollte ein kleines Unternehmen ein Betriebssystem wie Linux entwickeln? Oder Support für Datenbank- und Webserver leisten? Er meinte, sie würden es tun – aber gegen Bezahlung. Auch afrikanische Krankenhäuser, kleine Firmen und gemeinnützige Organisationen sollten zur Kasse gebeten werden. Ich war erschüttert. Ohne Open Source wären nur große Konzerne in der Lage, Software-Innovation voranzutreiben. Open Source macht die Welt zu einem besseren und gerechteren Ort.

Auch heute noch geht der Kampf für Softwarefreiheit weiter, wenn auch in unterschiedlichen Formen. Aber Freiheit bringt auch Herausforderungen mit sich - Herausforderungen, die den Weg von Log4j bestimmten.

Die Free Software Foundation (FSF) war eine der stärksten Stimmen der frühen Open-Source-Bewegung. Viele Open-Source-Projekte fanden ein Zuhause bei der FSF, und Log4j hätte eines von ihnen sein können. Aber es gab ein paar Gründe, warum dies nie geschah:
Das größte Hindernis war die Lizenzierung: Die GPL der FSF funktionierte gut für C und C++, machte aber Probleme mit Java. Da Java Bibliotheken dynamisch lädt, geriet es in Konflikt mit den strengen Linking-Regeln der GPL.

Ein weiterer wahrscheinlicher Grund war vielleicht, dass Log4j schon früh Ant zum Bauen seiner Software eingesetzt hatte. Irgendwann wandte sich der Betreuer von Log4j an die Apache Software Foundation (ASF), eine wachsende Gemeinschaft, die Java-Projekte unterstützt. Die freizügige Lizenz der ASF förderte eine breite Akzeptanz und war auch für Java-Software geeignet.

Warum also sollte Log4j nicht der ASF beitreten, wo Ant bereits zu Hause war?

Log4j trat der ASF bei und wurde Teil des Apache Jakarta-Projekts. Ja, Apache Jakarta - derselbe Name, den man heute von der Eclipse Foundation kennt, die Java Enterprise Standards entwickelt. Im Jahr 2000 war Apache Jakarta zu einem Dachprojekt geworden, das fünf große Java-bezogene Projekte beherbergte, darunter Apache Ant und Apache Tomcat.

In der Zwischenzeit, im Jahr 2001, wurde Apache Commons Logging vorgeschlagen. Ziel war es, eine einfache API bereitzustellen, die sowohl mit JDK Logging als auch mit Log4j kompatibel ist. Außerdem wurde ein Plugin-System eingeführt, die die nahtlose Integration anderer Logging-Frameworks ermöglicht.
Commons Logging zielte darauf ab, das Logging unter einer einzigen Schnittstelle zu vereinheitlichen und sich selbst als Standard zu positionieren.

Die Zeit von Log4j bei Jakarta war kurz. Im Jahr 2003 wurde es unter dem Namen „Logging Services" zu einem Top-Level-Projekt befördert. Bis dahin hatte Log4j eine kleine Gemeinschaft von sechs Mitwirkenden aufgebaut.
Die Entwicklung von einem Einzelprojekt zu einer echten Gemeinschaft ist selten einfach. Als sich Log4j weiterentwickelte, beauftragte das ASF-Board das Projekt außerdem mit der Integration von Schwester-Projekten wie Log4net und Log4php.

Die Zukunft sah rosig aus - zumindest für eine Weile.

Doch im Jahr 2005 brachten technische Streitigkeiten und persönliche Konflikte die Entwicklung zum Stillstand. Der Schwung ließ nach, und die Gemeinschaft hatte Mühe, sich zusammenzureißen. Schließlich zog sich der Gründer von Log4j zurück und gründete 2006 Logback und Slf4j.

Code ohne Community

Man hätte meinen können, dass die Trennung Frieden und Stabilität für Log4j bringen würde. Aber lange Streitigkeiten heilen nicht schnell, und Log4j hatte keine klare Vision. Die Community war erschöpft, und der Fortschritt kam zum Stillstand. Neue Versionen wurden nur sporadisch veröffentlicht.

So kam Version 1.2.15 im Jahr 2007 heraus. Der nächste Bugfix, 1.2.16, ließ weitere drei Jahre auf sich warten. Sogar Schwesterprojekte wie Log4php wurden zurückgefahren und schließlich eingestellt.

Es war eine düstere Zeit, in der es keine Anzeichen für Veränderungen gab. Log4j geriet ins Hintertreffen und drohte in Vergessenheit zu geraten.

Comeback

Im Jahr 2009 stieg ich in einer entscheidenden Phase in das Projekt ein. Für meine berufliche Arbeit war ich auf Log4php angewiesen – doch das Projekt war in einem desolaten Zustand. Also begann ich, es zu patchen – die erste nennenswerte Weiterentwicklung seit Jahren.

Kurz darauf engagierte sich Ralph Goers, um die Idee von Log4j 2 entwickeln. Zufällig ergänzten sich unsere Bemühungen, und das Projekt gewann wieder an Dynamik. Die Community erwachte zu neuem Leben, der Enthusiasmus wuchs – und mit ihm die Zahl der Mitwirkenden.

Die neuen Entwickler brachten eine frische, unvoreingenommene Perspektive in die alternde Log4j 1-Codebasis ein. Während Logback sich als Nachfolger positionierte, wurde Log4j 2 entwickelt, um aus den Fehlern der Vergangenheit zu lernen.
Es war eine komplette Neufassung. Die Apache Logging Services hätten ohne diesen Neuanfang wohl nicht überlebt.

Weltweit trugen Entwickler zur Weiterentwicklung von Log4j 2 bei und formten daraus ein lebendiges, community-getriebenes Projekt. Sein offener, gemeinschaftsorientierter Ansatz ermutigte Patches und Feedback beizusteuern.

Mit der Zeit wurde klar, dass Log4j 2 die Zukunft des Java-Logging ist und die Zeit von Log4j 1 vorbei war.

Im Jahr 2015 erreichte Log4j 1 offiziell sein Lebensende. Die Entwicklung wurde eingestellt, und der Schwerpunkt verlagerte sich auf Log4j 2. Das neue Team war überzeugt, dass die Lösung produktionsreif sei – und das war sie auch.

Positive Rückmeldungen förderten die Akzeptanz und zogen weitere Entwickler an. Zu den wichtigsten Beiträgen gehörten die asynchrone Protokollierung und der LMAX-Disruptor, der Log4j 2 wahnsinnig schnell machte.

Die Dinge sahen wieder gut aus - aber nicht jeder begrüßte die neue Logging-Landschaft.

Die Entwickler hatten nun die Wahl zwischen JUL, Slf4j, Logback und Commons Logging. Darüber hinaus existierten sowohl Log4j 1 als auch sein Nachfolger Log4j 2 nebeneinander.

Die Entwickler mussten sich für ein Framework entscheiden und „Brücken" konfigurieren, um verschiedene APIs zu verbinden. Bei Log4j 2 handelte

es sich nicht nur um ein Produkt, sondern um zwei, die zusammen verpackt waren. Wie Slf4j und Logback kombinierte es eine API und eine Implementierung. Das wurde nur nie so kommuniziert.

Dunkle Tage: Log4shell

Im Juli 2013, etwa ein Jahr vor der ersten stabilen Version von Log4j 2, steuerte ein Entwickler die Funktion 313 bei, die JNDI-Unterstützung ermöglicht. Mit dieser Funktion wurde JNDI in Log4j 2 integriert.
Es wurde ohne zu zögern vom Log4j Team angenommen. Da JNDI ein zentrales Java-Feature ist, schien es ein logischer Schritt zu sein, es hinzuzufügen.

Log4j 2 befand sich noch in der Betaphase, so dass nur wenige die Änderung in Frage stellten - oder sie überhaupt bemerkten.

Natürlich haben wir uns mit unserer Arglosigkeit geirrt.

Log4j 2 wurde mit der Funktion 313 ausgeliefert - eine der schwerwiegendsten Sicherheitslücken, die jemals entdeckt wurden.

Die Warnzeichen waren da - wir hätten sie auf der Black Hat 2016, einer großen Sicherheitskonferenz, bemerken können. Aber das haben wir nicht.

Bis Log4Shell an die Öffentlichkeit ging, war mir persönlich Black Hat kein Begriff. Dabei hatten Sicherheitsexperten dort bereits in jenem Jahr detailliert aufgezeigt, wie sich JNDI gezielt ausnutzen lässt.

Dann kam der November 2021.

Wir kommunizierten über E-Mail-Listen, und jedes Projekt hatte seinen eigenen privaten Kanal. Diese Listen sind in der Regel wenig frequentiert und für gelegentliche persönliche Angelegenheiten oder nichtöffentliche Sicherheitsdiskussionen reserviert.
Aber im November 2021 stieg der Verkehr auf der Mailingliste plötzlich sprunghaft an - etwas, das fast nie passierte. Es dauerte nicht lange, bis ich die Worte entdeckte: „Remote Code Execution".
Mein erster Instinkt? Den E-Mail-Client schließen. Ich sagte mir, dass ich

mich jetzt nicht damit befassen könnte.
Eine Stunde später wusste ich, dass ich keine andere Wahl hatte - auch wenn ich zu dem Zeitpunkt nicht aktiv an der Entwicklung beteiligt war. Glücklicherweise war es nicht mein Teil des Codes - andere übernahmen die Führung.
Man kann über das Problem sagen, was man will - das Team von Log4j hat schnell gehandelt und ist verantwortungsvoll damit umgegangen.

Die erste Meldung ging am 24. November 2021 ein.

Ein Entwickler in China entdeckte die Sicherheitslücke. Innerhalb eines Tages bestätigten wir das Problem und erkannten, dass es wichtige Projekte wie Apache Struts, Druid und Flink - und wahrscheinlich unzählige andere - betraf. Nach einigen Tagen der Diskussion hatten wir das Problem bis zum 5. Dezember behoben. Wie üblich hatten wir eine dreitägige Abstimmung über die Veröffentlichung eingeleitet. Trotz der Schwere des Problems blieb das Team zuversichtlich - und erleichtert, als alles funktionierte.

Doch gerade als die Abstimmung über die Freigabe beendet war, geschahen zwei Dinge.
Erstens begannen Sicherheitsgruppen, das Log4j-Problem offen zu diskutieren - in den sozialen Medien und darüber hinaus.
Dann kam der eigentliche Schock: eine E-Mail vom ursprünglichen Reporter.

Der Fix funktionierte nicht.

Ab diesem Zeitpunkt änderte sich alles. Plötzlich war ich mittendrin - ich verbreitete Informationen und Workaround über das Problem und nahm Anrufe von Unternehmen entgegen. Panik machte sich breit, und wir taten alles, um eine neue Version herauszubringen. Am 9. Dezember erstellten wir eine neue Version und verkürzten die übliche Wartezeit von drei Tagen auf nur einen Tag.

Log4j 2.15 wurde veröffentlicht.

Aber der Alptraum war noch nicht vorbei.

Weitere Schwachstellen tauchten auf, und wir flickten sie, eine nach der anderen.
Inmitten des Chaos erhielten einige von uns E-Mails voller Wut und Schuldzuweisungen. In den Foren entlud sich die Frustration. Im Nachhinein betrachtet hatte eine Handvoll engagierter Leute unermüdlich daran gearbeitet, das Problem zu beheben und wurden dafür beschimpft.
Keiner von uns wurde für diese Arbeit bezahlt. Einige nahmen sich sogar Tage frei und opferten ihren Urlaub.
Das war unser Problem, kein Zweifel. Aber wir haben das sinkende Schiff nicht aufgegeben - wir haben dafür gekämpft, es über Wasser zu halten.

Ich denke oft an diese Zeit zurück und daran, was sie mich gelehrt hat: Dass dieses Projekt nie nur aus einer oder zwei Personen bestand, sondern aus einem Team. Mit einem starken Team und engagierten Leuten haben wir es geschafft!

Aftermath

Nach der Korrektur wurde Open Source auf eine neue Art und Weise betrachtet. Regierungen und große Zeitungen griffen die Geschichte auf und entfachten eine Diskussion über die Nachhaltigkeit von Open Source.
Jahrzehntelang arbeiteten die Entwickler kostenlos und stellten ihre Arbeit dem Gemeinwohl zur Verfügung, während die Unternehmen davon profitierten.

Und dann flog die Grundlage der Firmen und ihres Profits in die Luft.
Und wieder meldeten sich dieselben wenigen Leute, um das Problem zu lösen. Unbezahlt.

Vor zwanzig Jahren spielten Programmierung und Software noch eine ganz andere Rolle in unserem Leben. Damals gab es noch nicht einmal Smartphones. Aber heute benutzen wir sie, um Rechnungen zu zahlen. Ihr Auto ist heute mehr Computer als Maschine. Fast jedes Gerät in Ihrem Haushalt ist heute ein winziger Computer–sogar Ihr Wasserkocher ist vielleicht mit dem Wi-Fi verbunden.

Zum Glück beginnt sich etwas zu ändern. Es ist noch lange nicht alles

perfekt, aber Organisationen wie die Sovereign Tech Agency (STA) setzen sich für die Nachhaltigkeit von Open Source ein.
In den Jahren 2023 und 2024 erhielt das Log4j-Team Fördermittel, die wir gerne angenommen haben. Nun, zumindest ein Teil des Teams.
Ich habe schnell gelernt, dass die Finanzierung von Entwicklern nicht so einfach ist, wie es klingt. Einige konnten aufgrund von arbeitsvertraglichen Einschränkungen nicht zusagen. Bei anderen gab es steuerliche Komplikationen. Und einige hatten keine Zeit für ein großes Projekt mit begrenzten Mitteln und ungewisser beruflicher Zukunft.
Trotzdem: drei Personen arbeiteten 1,5 Jahre lang Vollzeit an Log4j. Wir haben Sicherheitsfunktionen wie SBOM hinzugefügt, Bereinigungen vorgenommen und die Dokumentation und die Website verbessert. Außerdem haben wir es möglich gemacht, Log4j mit Graal VM zu verwenden und es auf Android laufen zu lassen. Wir haben viel gelernt, und Log4j ist trotz der Krise, in der es steckte, gewachsen.

Takeaways

Was wäre, wenn wir nur zu zweit wären?

Was wäre, wenn wir zu dieser Zeit im Urlaub wären?

Was wäre, wenn es uns egal wäre?

Log4j hätte viele Male aufgegeben werden können, aber es hat überlebt. Manchmal war es Glück, und manchmal waren es die richtigen Leute zur richtigen Zeit.

Was habe ich gelernt?

Software-Ingenieure müssen Sicherheit ernster nehmen. Sicherheit wird eine der größten Herausforderungen für die Zukunft der Software sein. KI wird Angreifern helfen, Schwachstellen schneller zu entdecken, und wir müssen wachsam bleiben. Da sich die Bedrohungen weiterentwickeln, müssen wir immer einen Schritt voraus sein.

Überlegen Sie mal: Log4Shell wurde im Jahr 2013 eingeführt - und blieb bis 2021 unentdeckt. Wir können nicht wissen, ob es nicht schon vorher ausgenutzt wurde.

Die Gemeinschaft ist das Rückgrat der Softwareentwicklung. Weit verbreitete Projekte können nicht allein gewartet werden - wir brauchen ein zuverlässiges Team.
Open Source hat ein Nachhaltigkeitsproblem. Tech-Giganten profitieren von Open Source, doch ohne sie würde die Innovation zum Erliegen kommen.

Open Source ist überall. Lassen Sie uns dafür sorgen, dass es eine nachhaltige Zukunft hat.

Zukunft

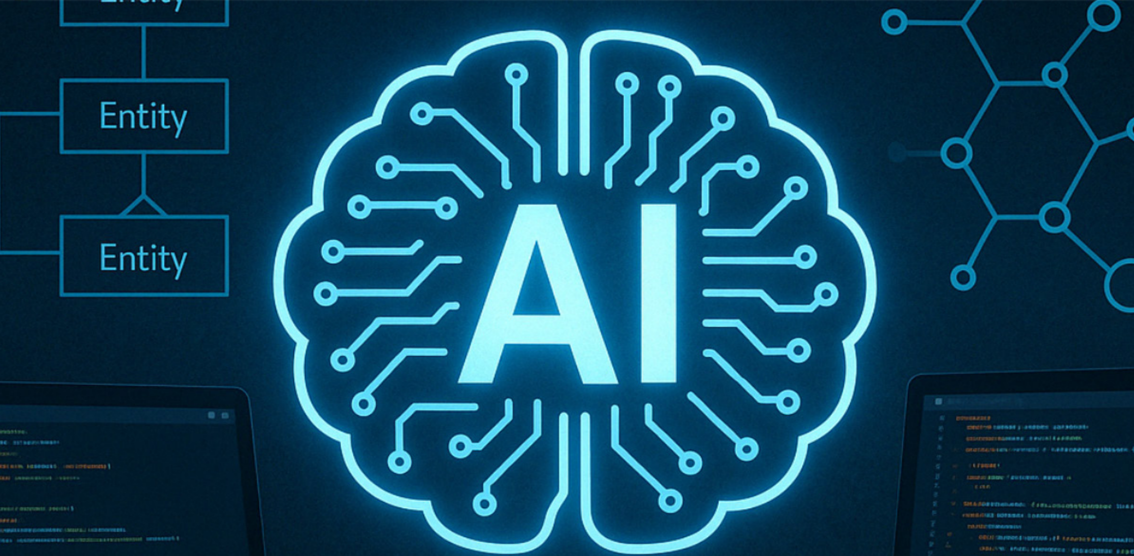
Was bedeutet das für Log4j heute?

Aufgrund von Log4Shell wurde kein anderes Logging-Framework so gründlich getestet. Das Team hat seine Verantwortung und sein Engagement für die Sicherheit unter Beweis gestellt. Ja, es ist „sicher". Niemand kann das nächste Log4Shell vorhersagen, aber wir haben unsere Sicherheit verbessert und eine sauberere, robustere Codebasis geschaffen.

Zusammenarbeit ist alles. Einige von uns hoffen, eines Tages unter einer gemeinsamen Logging-API zu arbeiten, die im Rahmen des Eclipse Jakarta-Projekts gehostet wird.

Bis es so weit ist, freuen wir uns auf weitere 20 Jahre Logging. Und wer weiß? Vielleicht schreibe ich in zwanzig Jahren einen weiteren Artikel wie diesen. Geschichte wird immer noch jeden Tag geschrieben. Sie können ein Teil davon sein. Hinterlassen Sie Ihre Spur – Ihr nächster Commit wartet schon.

> Zurück zum Inhaltsverzeichnis



#JAVAPRO #AI #ML

KI-Tools für Jakarta EE

Autor:

Gaurav Gupta ist Senior Software Engineer bei Payara und der Schöpfer von Jeddict, einer innovativen, quelloffenen und KI-gestützten Entwicklungsplattform für Jakarta EE-Anwendungen. Als Committer bei Apache NetBeans und Eclipse GlassFish bringt Gaurav umfangreiche Fachkenntnisse und großes Engagement in die Weiterentwicklung dieser wichtigen Open-Source-Werkzeuge ein. Sein Fokus liegt auf der Steigerung der Entwicklerproduktivität und der Weiterentwicklung der Jakarta EE-Standards durch moderne Technologien – ein Ausdruck seines Engagements für das Open-Source-Ökosystem und seiner Mission, Entwickler weltweit zu stärken.

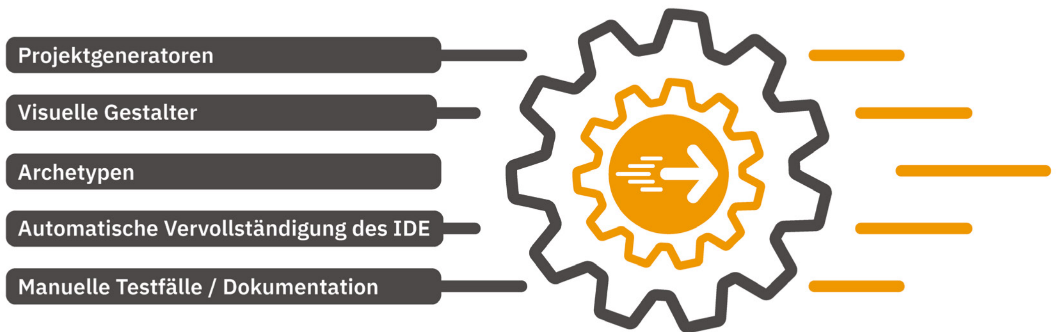
https://www.linkedin.com/in/jgauravgupta/

Künstliche Intelligenz (KI) verändert die Welt der Softwareentwicklung, und so auch Jakarta EE. In der Vergangenheit verließen sich Entwickler bei der Erstellung von Anwendungen auf Projektgeneratoren, visuelle Gestalter und IDE-Autovervollständigung. Es mangelte diesen Instrumenten jedoch häufig an Anpassungsfähigkeit, Kontextbewusstsein und Effizienz. Dank KI-gestützter Tools wie Payara Starter und Jeddict AI Assistant sind diese Einschränkungen jetzt kein Problem mehr; sie beschleunigen

außerdem Entwicklungsabläufe und ermöglichen es Entwicklern, sich auf Innovationen zu konzentrieren. In diesem Artikel wird anhand einer Konferenzanwendung als praktisches Beispiel untersucht, wie diese Werkzeuge die Entwicklung von Jakarta EE verbessern. Darüber hinaus werden zukünftige Möglichkeiten erörtert, darunter in Payara Server integrierte KI-Assistenten für die Einrichtung und Diagnose von Servern.

Herkömmliche Herausforderungen bei der Entwicklung

Jakarta EE-Entwickler haben sich lange Zeit auf konventionelle Tools verlassen, die zwar hilfreich sind, aber auch gewisse Einschränkungen mit sich bringen, die die Entwicklung verlangsamen und den Aufwand für Anpassungen, Tests und Dokumentation erhöhen können. Obwohl diese Instrumente wirksam sind, haben sie auch ihre Grenzen:



1. **Projektgeneratoren und Archetypen:** Diese Tools unterstützen Entwickler bei der schnellen Erstellung von Anwendungen mithilfe vordefinierter Vorlagen. Diese Vorlagen sind jedoch oft starr und erfordern umfangreiche Anpassungen, um den spezifischen Projektanforderungen gerecht zu werden. Insbesondere bei umfangreichen Anwendungen kann dieser Prozess kann sehr zeitaufwändig sein.

2. **Visuelle Gestalter:** Mit visuellen Tools können Entwickler Datenbankschemata erstellen und UI-Layouts entwerfen. Diese Werkzeuge haben jedoch kein kontextbezogenes Bewusstsein, weshalb jede Komponentenplatzierung manuell vorgenommen werden muss. Dieser manuelle Prozess kann wiederum zu Unstimmigkeiten und langsameren Entwicklungszeiten führen.

3. **IDE Auto-Vervollständigung:** IDEs bieten Funktionen zur automatischen Vervollständigung, die bei grundlegenden Syntax- und Codevorschlägen helfen. Diese Vorschläge sind jedoch nur begrenzt

aussagekräftig, da sie den Kontext des Projekts nicht vollständig verstehen und daher keine komplexen oder Empfehlungen auf hoher Ebene geben können.

4. **Manuelle Prüfung und Dokumentation:** Das Schreiben von Testfällen und Dokumentationen von Hand ist zeitaufwändig. Die Sicherstellung einer umfassenden Testabdeckung sowie die Pflege einer stets aktuellen Dokumentation sind für qualitativ hochwertige Software unerlässlich, erfordern jedoch oft einen erheblichen Aufwand.

Diese Einschränkungen können die Entwicklungszyklen verlangsamen, das Fehlerrisiko erhöhen und Entwickler dazu zwingen, mehr Zeit für sich wiederholende Aufgaben aufzuwenden, anstatt sich auf Innovationen und Kernfunktionen zu konzentrieren.

Der KI-Vorteil bei der Entwicklung von Jakarta EE

Generative KI-Tools verändern die Entwicklung von Jakarta EE, indem sie diese Herausforderungen angehen und intelligentere, kontextabhängige Lösungen anbieten:

1. **Adaptive Projektgeneratoren:** Im Gegensatz zu herkömmlichen Projektgeneratoren können KI-gestützte Generatoren Anwendungsgerüste basierend auf natürlichsprachlichen Eingabeaufforderungen erstellen. Diese Generatoren sind adaptiv, d. h. sie wählen automatisch geeignete Symbole, Titel, Beschreibungen und dynamische Seiteninhalte auf der Grundlage des Anwendungskontextes aus, so dass weniger manuelle Anpassungen erforderlich sind.

2. **Kontextbewusste visuelle Gestalter:** KI-gesteuerte visuelle Gestalter können mit Befehlen in natürlicher Sprache arbeiten. So können Entwickler die gewünschte Benutzeroberfläche und Datenbankstruktur beschreiben, und die KI entwirft die Anwendung automatisch auf der Grundlage des Verständnisses des bestehenden

Kontexts und der bewährten Verfahren. Diese Funktion macht die manuelle Platzierung von Bauteilen überflüssig und gewährleistet ein konsistentes Design.

3. **Archetypen der natürlichen Sprache:** Traditionelle Archetypen werden durch KI-gesteuerte Codegenerierung ersetzt, die auf Befehle in natürlicher Sprache reagiert. Entwickler können einfach die benötigte Funktionalität beschreiben, z. B. „REST-Endpunkt für Kundendaten erstellen", und die KI generiert den erforderlichen Code automatisch, was die Entwicklung beschleunigt.

4. **Automatisierte Test- und Dokumentationserstellung:** KI kann durch die Analyse des Codekontexts relevante Testfälle sowie umfassende Dokumentationen erstellen. Diese Automatisierung gewährleistet Genauigkeit und Vollständigkeit und spart Entwicklern viel Zeit.

5. **Intelligente Code-Vervollständigung:** Die KI-gestützte Codevervollständigung versteht den gesamten Projektkontext und geht so über Syntaxvorschläge hinaus. Die kontextabhängige Codevervollständigung führt zu intelligenteren, relevanteren Codevorschlägen, wodurch Entwickler saubereren und effizienteren Code schreiben können.

Durch die Automatisierung sich wiederholender und zeitaufwändiger Aufgaben können sich Jakarta EE-Entwickler auf die Entwicklung von Kernfunktionen, die Verbesserung der Benutzerfreundlichkeit und das Vorantreiben von Innovationen konzentrieren. Dieser Wandel führt zu schnelleren Entwicklungszyklen, qualitativ hochwertigerer Software und einer höheren Gesamtproduktivität.

Wir betrachten nun einige leistungsstarke KI-Tools näher, die die Jakarta EE-Entwicklungslandschaft verändern:

Payara Starter

Payara Starter (https://start.payara.fish/) ist ein leistungsstarkes, KI-gestütztes Tool, das die Entwicklung von Jakarta EE-Anwendungen vereinfacht. Mithilfe von Entity-Relationship (ER)-Diagrammen können Entwickler schnell und ohne manuelles Setup Anwendungsarchitekturen entwerfen. Die generative KI in Payara Starter erstellt automatisch

ER-Diagramme auf der Grundlage von Beschreibungen in natürlicher Sprache und ermöglicht es Entwicklern, Diagramme mittels Live-Chat-Befehlen zu verfeinern.

Beispiel: JCON EUROPE Konferenz Anmeldung

Payara Starter wurde für die Erstellung eines ER-Diagramms für eine Konferenzanwendung verwendet. Durch die Beschreibung der Anwendungsanforderungen (z. B. „JCON EUROPE Conference“) erstellte das Tool ein Diagramm mit Entitäten wie Konferenz, Teilnehmer und Sitzung und stellte Beziehungen zwischen ihnen her.

```
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

Mithilfe von Befehlen in natürlicher Sprache wurde das Diagramm um zusätzliche Entitäten wie Feedback und Registrierung erweitert, was das Kontextbewusstsein des Tools verdeutlicht.

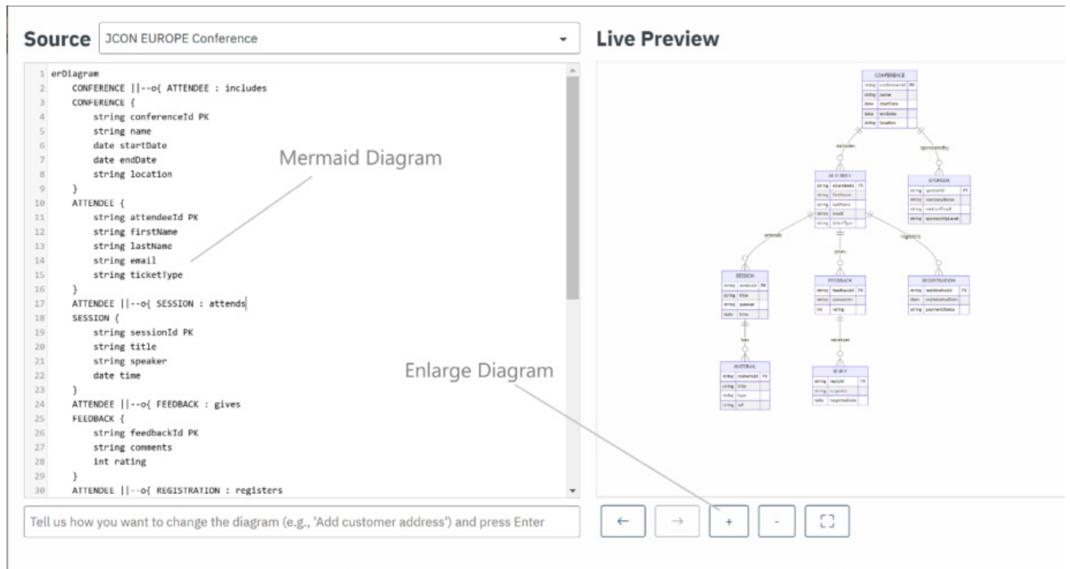
```
Source  JCON EUROPE Conference
1  erDiagram
2      CONFERENCE ||--o{ ATTENDEE : includes
3      CONFERENCE {
4          string conferenceId PK
5          string name
6          date startDate
7          date endDate
8          string location
9      }
10     ATTENDEE {
11         string attendeeId 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

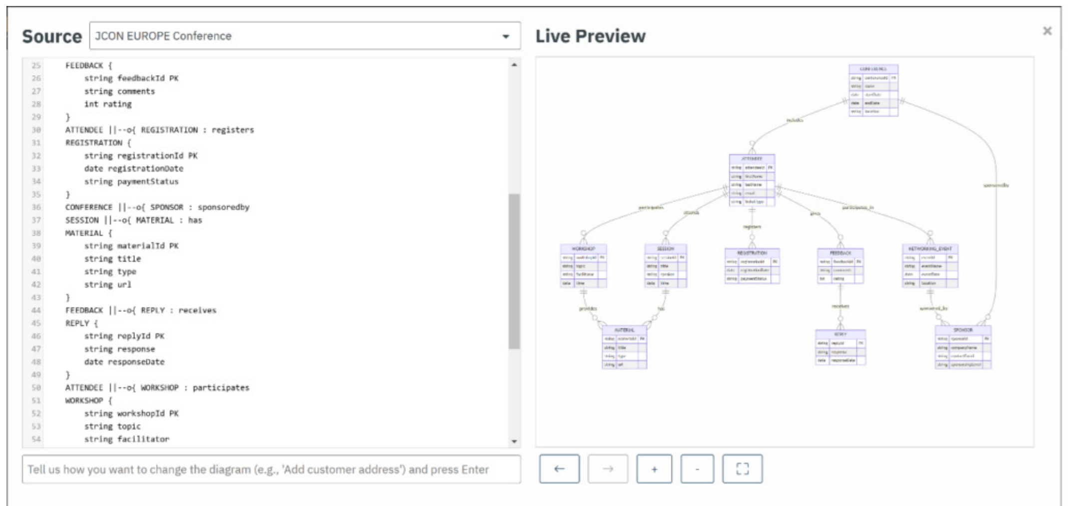
Live Preview

Add Feedback and Registration

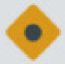
Mit der Schaltfläche „+ Vergrößern“ wurde das Diagramm um Entitäten wie Sponsor, Material und Antwort erweitert. Diese Funktion ermöglicht es Entwicklern, ihr Anwendungsdesign iterativ zu verfeinern und so eine umfassende Abdeckung aller Geschäftsanforderungen zu gewährleisten.



Wenn Sie das Diagramm mit der Schaltfläche „+ Vergrößern“ weiter ausbauen, werden dem Diagramm neue Einheiten wie Networking Event und Workshop hinzugefügt.



Nachdem der Entwurf fertiggestellt und die Versionen von Jakarta EE, Payara und JDK ausgewählt waren, wurde die Anwendung mit einem einzigen Klick erstellt.



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

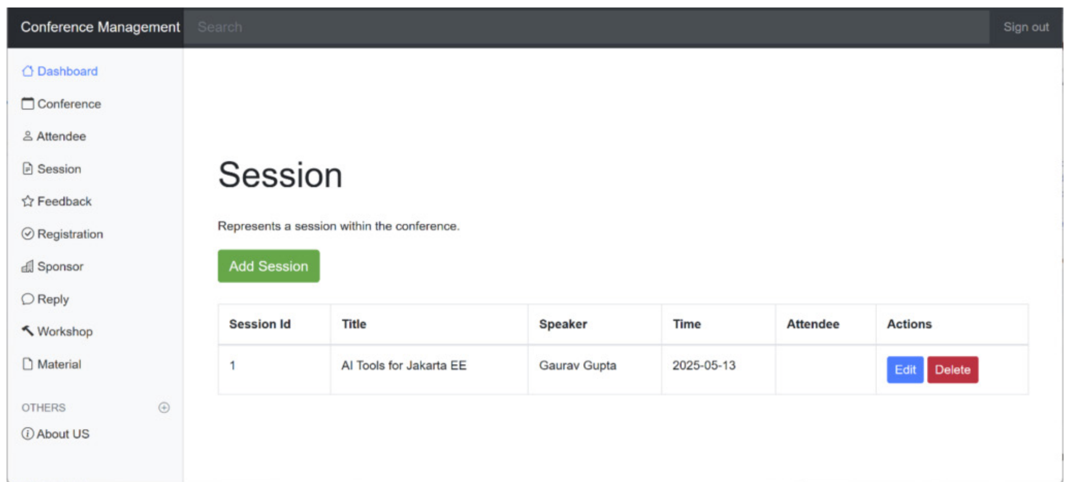
Your request is in the queue. Please wait a moment while we process it.

Generate

Die KI konstruiert nicht nur die Jakarta EE-Backend-Struktur mit JPA-Entitätsklassen und Beziehungen, sondern generiert auch automatisch das Frontend. Die Benutzeroberflächen werden mit auf die Funktionen der einzelnen Einheiten zugeschnittenen, realistischen Titeln,

Beschreibungen und Symbolen gestaltet. Durch diese Automatisierung wird die Zeit für die Frontend-Entwicklung erheblich verkürzt und gleichzeitig ein konsistentes UI-Design gewährleistet.

Diese Fähigkeit stellt einen hybriden Ansatz dar, der die kontextbezogene Anpassungsfähigkeit von LLMs mit den strukturierten Vorlagen traditioneller Archetypen kombiniert. Während das LLM Beschreibungen in natürlicher Sprache versteht und dynamische Inhalte generiert, sorgen Archetypen für ein konsistentes Codegerüst und Entwurfsmuster. Zusammen rationalisieren sie die Erstellung von Datenbankschemata als auch das Design der Benutzeroberfläche des Frontends und beschleunigen gleichzeitig die Entwicklung, ohne dabei Flexibilität oder Konsistenz zu beeinträchtigen.



Dies führte zu einer erheblichen Verkürzung der Entwicklungszeit, wodurch sich die Entwickler auf die Kernfunktionen konzentrieren konnten.

Jeddict AI-Assistent

Jeddict AI Assistant (https://jeddict.github.io/) ist ein in Apache NetBeans integriertes KI-gestütztes Tool, das die Java- und Jakarta EE-Entwicklung optimieren soll. Es bietet intelligente Code-Vorschläge, kontextabhängige Variablen- und Methodenbenennung, automatische Testerstellung und vieles mehr. Dank der Unterstützung für mehrere LLM-Anbieter können Entwickler die Konfigurationen an ihre Arbeitsabläufe anpassen und so Effizienz und Konsistenz gewährleisten.

Wesentliche Merkmale

- **Intelligente Code-Vervollständigung:** Bietet kontextabhängige Vorschläge in Echtzeit, während der Entwickler tippt, was die Programmierzeit verkürzt und die Produktivität erhöht.
- **Inline-Hinweise im Code-Editor:** Zeigt kontextbezogene Hinweise direkt im Editor an und hilft Entwicklern, Methodenanwendung, erwartete Parameter und bewährte Verfahren zu verstehen.
- **Kontextabhängige Benennung:** Schlägt sinnvolle Variablen- und Methodennamen vor, die auf dem Umgebungscode basieren und sich an einheitlichen Namenskonventionen orientieren.
- **Inline-SQL-Vervollständigung:** Vereinfacht Interaktionen mit Datenbanken durch SQL-Vorschläge in Echtzeit, während Entwickler Abfragen schreiben.
- **Automatisierte Protokollierung und Dokumentation:** Erzeugt klare, prägnante Protokollmeldungen sowie umfassende Javadocs für Klassen und Methoden.
- **Testfallgenerierung:** Erstellt automatisch relevante Testfälle auf der Grundlage des Kontexts von Klassen und Methoden und fördert so optimale Testverfahren.
- **Unterstützung mehrerer LLM-Anbieter:** Ermöglicht Entwicklern die Auswahl aus verschiedenen großen Sprachmodellanbietern mit anpassbaren Konfigurationen, einschließlich benutzerdefinierter Header für sichere API-Integrationen.
- **Spezieller Support für Jakarta EE und Java EE Projekte:** Erkennt sowohl javax- als auch jakarta-Importe und gewährleistet so die Kompatibilität mit älteren und modernen Unternehmensanwendungen.
- **REST-Endpunkt-Generierung:** Schnelle Generierung von REST-Endpunkten, einschließlich Ressourcenklassen, Anmerkungen und Anfrage-/Antwortmodellen, was wiederum die Entwicklungszeit verkürzt.
- **Kontextabhängiger KI-Chat:** Ermöglicht Entwicklern die Auswahl bestimmter Java-Dateien oder -Pakete, um den Kontext für KI-gesteuerte Konversationen zu liefern und so leichter relevante

Vorschläge und Erkenntnisse zu erhalten.

- **KI-generierte Commit-Nachrichten:** Erzeugt automatisch klare und aussagekräftige Commit-Nachrichten auf der Grundlage von Codeänderungen und sorgt so für eine bessere Versionskontrolle und Zusammenarbeit.

Jeddict AI Assistant ermöglicht es Entwicklern, sich auf die Entwicklung von Kernfunktionen zu konzentrieren, während sich wiederholende Aufgaben automatisiert, die Produktivität gesteigert und die Codequalität in Java- und Jakarta EE-Projekten verbessert werden.

Schlussfolgerung

KI-Tools wie Payara Starter und Jeddict AI Assistant revolutionieren die Jakarta EE-Entwicklung, indem sie sich wiederholende Aufgaben automatisieren, die Codierung beschleunigen und die Codequalität verbessern. Mit Funktionen wie intelligenter Code-Vervollständigung, kontextsensitiver Variablenbenennung, automatischer Testgenerierung und nahtloser Erstellung von REST-Endpunkten ermöglichen diese Tools den Entwicklern, sich auf die Entwicklung von Kernfunktionen zu konzentrieren und qualitativ hochwertige Anwendungen schneller zu erstellen. Darüber hinaus optimieren KI-generierte Dokumentationen und Commit-Nachrichten die Zusammenarbeit, so dass Teams Konsistenz wahren und Wissen effektiv austauschen können.

Mit Blick auf die Zukunft könnte ein speziell für Payara Server entwickelter KI-Assistent die Produktivität von Entwicklern aller Qualifikationsstufen weiter steigern. Anfänger können von einer vereinfachten Servereinrichtung und einer schrittweisen Anleitung profitieren, ohne dass sie fortgeschrittene Kenntnisse der asadmin-Befehle benötigen. Gleichzeitig können erfahrene Anwender die KI für erweiterte Diagnosen, Protokollanalysen und Leistungsoptimierung nutzen und so einen effizienten und zuverlässigen Serverbetrieb sicherstellen. Eine automatisierte Serverkonfiguration durch Befehle in natürlicher Sprache würde die Bereitstellung und Verwaltung weiter vereinfachen und die Komplexität der Serververwaltung reduzieren.

Mit der ständigen Weiterentwicklung der KI verspricht die Integration

von KI in die Jakarta EE-Entwicklung intelligentere, schnellere und effizientere Arbeitsabläufe. Durch die Verringerung des manuellen Aufwands und die Automatisierung zeitraubender Aufgaben ermöglicht KI den Entwicklern, sich auf Innovationen zu konzentrieren und robuste, skalierbare und hochleistungsfähige Anwendungen zu erstellen. Mit Tools wie Payara Starter, Jeddict AI Assistant und zukünftigen Fortschritten in der KI-gesteuerten Serververwaltung wird die Zukunft der Jakarta EE-Entwicklung garantiert produktiver und zugänglicher sein als je zuvor.

> Zurück zum Inhaltsverzeichnis

JCON2026
www.jcon.one

JAVAPRO

APR
20-23

JCON
EUROPE 2026
COLOGNE

CINEDOM COLOGNE

JCON EUROPE 2026
International Java Community Conference

#JAVAPRO #AI #ML

Entwicklung von KI-Anwendungen mit Spring AI

Autor:

Timo Salm ist Solution Engineer für Developer Experience bei Tanzu von Broadcom in der EMEA-Region und konzentriert sich auf interne Entwicklerplattformen sowie kommerzielle Spring-Produkte. In dieser Rolle ist er dafür verantwortlich, Kunden den Nutzen dieser Produkte zu vermitteln und ihren Erfolg sicherzustellen – indem er eng auf verschiedenen Abstraktionsebenen moderner Anwendungen und moderner Infrastruktur zusammenarbeitet. Zuvor war er als Softwarearchitekt und Full-Stack-Entwickler für Beratungsunternehmen in der Automobilbranche tätig.

https://www.linkedin.com/in/timosalm/

Künstliche Intelligenz (KI) wird für moderne Anwendungen immer wichtiger. Während KI unterschiedliche Technologien umfasst, ist der Fokus derzeit aufgrund der jüngsten Fortschritte bei großen Sprachmodellen (LLMs) auf Generative KI (GenAI).

Traditionell ist Python die dominierende Sprache für KI. Doch für Java-Entwickler, die Generative KI nutzen möchten, bietet das Spring AI

Projekt eine attraktive Alternative. Es vereinfacht die Entwicklung von KI-gestützten Enterprise-Anwendungen erheblich und ermöglicht es, so mit der rasant fortschreitenden KI-Landschaft Schritt zu halten.

Spring AI abstrahiert komplexe Interaktionen mit verschiedenen KI-Anbietern, die REST APIs bereitstellen, darunter OpenAI, Anthropic, Microsoft, Google, Amazon und sogar lokalen LLMs. Durch diese Abstraktionen ist ein einfacher Wechsel zwischen verschiedenen Modellen möglich, gleichzeitig ist aber, wie in Spring üblich, gewährleistet, auf spezifische Funktionen und Konfigurationen einzelner Modelle zugreifen zu können.

Das Framework bietet eine Vielzahl an Funktionen, wie die Konvertierung von Modellausgaben in Java Objekte, Multimodalität, KI-bezogene Observability und Testunterstützung zur Bewertung von Modellausgaben, bereit.

Darüber hinaus unterstützt Spring AI fortgeschrittene Techniken wie Tool Calling, Retrieval-Augmented Generation (RAG) und das Model Context Protocol (MCP), um den Kontext von LLMs anzureichern.

Spring AI baut auf den Kernfunktionalitäten des Spring Frameworks und weiteren Spring Projekten wie Spring Data zur Integration von Vektor-Datenbanken auf. Durch die automatische Konfiguration von Spring Boot, wird die Entwicklung von KI-gestützten Funktionen erheblich vereinfacht und beschleunigt.

Erste Schritte mit Spring AI

Nach dem groben Überblick zu Spring AI, widmen wir uns nun der Umsetzung im Code! Um Spring AI in deiner Spring Boot Anwendung zu nutzen, muss zunächst die entsprechende Bibliothek für den gewünschten KI-Anbieter hinzugefügt werden. In diesem Beispiel wird Ollama verwendet, das die lokale Ausführung von verschiedene LLMs unterstützt. Es wird außerdem empfohlen, die Spring AI Bill of Materials (BOM) hinzuzufügen, bis Spring Boot kompatible Versionen der Spring AI-Abhängigkeiten verwalten.

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

Die Standart-Konfiguration für Ollama geht davon aus, dass Mistral als LLM verwendet wird und die API zur Interaktion mit dem Modell auf demselben Host unter Port 11434 zu erreichen ist.

```
ollama pull mistral
```

Die automatische Konfiguration lässt sich einfach per Konfigurationsdatei oder Code anpassen. In diesem Beispiel wird z.B. über Ollama anstatt Mistral das Llama 3.2 Modell bereitgestellt.

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

Die ChatClient API

Mit der ChatClient API kann mit wenigen Zeilen Code mit dem KI-Modell der Wahl kommuniziert werden. Es wird sowohl die in diesem Beispiel verwendete synchrone Kommunikation, als auch Streaming unterstützt.

```
String answer = this.chatClient.prompt()
        .user(„Was ist die Hauptstadt von Deutschland?")
        .call()
        .content();
```

Die prompt() Methode initialisiert eine Interaktion mit einem KI-Modell und ermöglicht die Zusammenstellung der Anweisungen für dieses. Mit der call() Methode werden die Anweisungen, die als "Prompt" bezeichnet werden, an das KI-Modell bzw. die API des Anbieters gesendet, das eine Antwort generiert und zurücksendet. Die content() Methode extrahiert schließlich die vom Modell erzeugte Antwort von den restlichen Daten und gibt sie als String zurück.

Eine ChatClient Instanz wird mithilfe des Builder-Patterns von einem ChatClient.Builder erstellt. Durch die automatische Konfiguration steht dieser als Spring Bean für Dependency Injection zur Verfügung.

```
@Component
class MyComponent {

  private final ChatClient chatClient;
  // Konstruktor-basierte Dependency Injection
  public MyComponent(ChatClient.Builder chatClientBuilder) {
    this.chatClient = chatClientBuilder.build();
  }
...
}
```

Anweisungen können Platzhalter enthalten, die durch geschweifte Klammern gekennzeichnet sind und zur Laufzeit mit dynamischen Werten befüllt werden. Statt einer Zeichenkette wird hierfür eine Funktion vom Typ Consumer<PromptUserSpec> and die user() Methode übergeben, um die Werte für die Platzhalter zu definieren. Ein Consumer ist eine funktionale Schnittstelle in Java, die eine Eingabe, in diesem Fall ein Objekt des Typs PromptUserSpec, verarbeitet, ohne einen Rückgabewert zu liefern.

```
String country = „Deutschland“;
String answer = this.chatClient.prompt()
        .user(promptUserSpec -> promptUserSpec
                .text(„Was ist die Hauptstadt von {country}?“)
                .param(„country“, country))
        .call()
        .content();
```

Für KI-Modelle, die multimodale Eingaben unterstützen, also neben Text auch Bilder oder Audiodateien verarbeiten können, bietet die PromptUserSpec zusätzlich die Methode media(). Damit lassen sich Bild- und Audiodateien in den Prompt integrieren und gemeinsam mit der Anweisung in Textform an das Modell senden.

```
var imageResource = new ClassPathResource(„berlin.png“);
String answer = this.chatClient.prompt()
        .user(promptUserSpec -> promptUserSpec
                .text(„Was ist der Name dieser Stadt?“)
                .media(MimeTypeUtils.IMAGE_PNG, imageResource))
```

```
        .call()
        .content();
```

Strukturierte Ausgaben

Die ChatClient API bietet mehrere Möglichkeiten, mit KI-generierten Ausgaben umzugehen. Im vorherigen Beispiel wurde die Methode content() verwendet, um die Antwort des Modells als String bereitzustellen. Die API unterstützt jedoch auch die direkte Umwandlung der Ausgabe in Java Objekte. Dies wird durch Systemanweisungen erreicht, bei denen es sich um spezielle Anweisungen handelt, deren Zweck es ist, das Verhalten des KI-Modells zu steuern. Mit der system() Methode der ChatClient API können solche Systemanweisungen ähnlich wie Benutzeranweisungen in den Prompt integriert werden

```
String answer = this.chatClient.prompt()
        .user(„Was ist die Hauptstadt von Deutschland?“)
        .system(„Du bist ein Geschichtsexperte. Verwende ausführliche
                Erklärungen.“)
        .call()
        .content();
```

Zusätzlich kann über die Methode defaultSystem() in der ChatClient.Builder Klasse eine Systemanweisung für alle ChatClient API Interaktionen festgelegt werden.

Um die automatische Umwandlung von KI-generierten Ausgaben zu Java Objekten im ChatClient zu ermöglichen, fügt ein StructuredOutputConverter im Hintergrund eine Systemanweisung zum Prompt hinzu. Diese Anweisung fordert das Modell auf, die Antwort in einem bestimmten Format, beispielsweise JSON, zurückzugeben, das anschließend in ein Java Objekt überführt wird.

```
record City(String name, String zipcode) {}
```

```
City capitalOfGermany = this.chatClient.prompt()
        .user(„Was ist die Hauptstadt von Deutschland?“)
        .call()
        .entity(City.class);
```

Fortgeschrittene KI-Techniken

Während Prompt Engineering hilft, die Antworten eines KI-Modells zu steuern, hat es dennoch seine Grenzen. Egal wie gut ein Prompt formuliert ist, ein LLM kann nur Antworten auf Basis seines vorab trainierten Wissens und des im Prompt bereitgestellten Kontexts generieren.

Um diese Einschränkungen zu überwinden, kommen in KI-Anwendungen zunehmend Agenten zum Einsatz. Dies sind autonome, intelligente Systeme, die bestimmte Aufgaben ohne menschliches Eingreifen ausführen können. Diese Agenten erweitern die Fähigkeiten eines LLMs, indem sie planen, Aktionen ausführen und dynamisch neue Informationen abrufen oder generieren. Spring AI stellt die wesentlichen Bausteine für die Implementierung und Nutzung von KI-Agenten bereit.

Tool Calling

Ein solcher Bausteine ist Tool Calling, auch bekannt als Function Calling. Dabei können LLMs externe APIs nutzen, um bestimmte Aufgaben auszuführen und aktuelle Informationen bei der Generierung ihrer Antwort einzubeziehen. Wenn ein Modell eine Benutzeranweisungen erhält, prüft es, ob eine vorab definierte externe Funktion aufgerufen werden muss, um diese zu erfüllen. Falls dies der Fall ist, generiert das Modell eine strukturierte Antwort mit den Details zur aufrufenden Funktion sowie den benötigten Parametern. Das Client System verarbeitet diese Antwort, führt die entsprechende Funktion aus und gibt das Ergebnis an das LLM zurück, das auf dieser Grundlage dann die finale Antwort generiert.

Spring AI bietet mehrere Möglichkeiten, aufrufbare Funktionen zu definieren. Eine davon ist der deklarative Ansatz mit der @Tool Annotation, die ein Beschreibungselement enthält, um dem LLM den Zweck und die Funktionalität dieser Funktion zu erklären. Ebenso kann die @ToolParam Annotation verwendet werden, um einzelne Parameter der Funktion zu beschreiben.

```
@Service
class WeatherService {


  @Tool(description = „Liefert die aktuelle Temperatur in einer
```

```
Stadt“)
  Double fetchCurrentTemperature(
    @ToolParam(description = „Name der Stadt“) String city) {
    ...
  }
}
```

Um eine definierte Funktion in einer bestimmten Anweisung zu nutzen, wird eine Instanz der Klasse, welche die Funktion enthält, mit der tools() Methode an die ChatClient API übergeben. Um Funktionen für alle ChatClient API Interaktionen verfügbar zu machen, können sie in der defaultTools() Methode des ChatClient.Builder registriert werden.

Intern sendet Spring AI die Funktionsdefinitionen zusammen mit dem Prompt an das LLM. Das Modell entscheidet dann, ob ein Funktionsaufruf erforderlich ist. Falls ja, gibt es den Namen der Funktion sowie die individuellen Werte der Parameter zurück. Spring AI führt die Funktion mit den übergebenen Eingaben aus und gibt das Ergebnis an das LLM zurück.

Die Tool Calling Funktionalität von LLMs ist außerdem ein zentraler Bestandteil des Model Context Protocols (MCP). Dieses Protokoll standardisiert die Kommunikation zwischen KI-Modellen und verschiedenen Datenquellen sowie externen Funktionen und vereinfacht es so Agenten auf Basis von LLMs zu erstellen. MCP folgt einer Client-Server Architektur, bei der eine KI-Anwendung (MCP-Host) über eingebettete MCP-Clients mit MCP-Servern kommuniziert, um auf spezifische Ressourcen oder Funktionen zuzugreifen.

Spring AI bietet seit Kurzem Unterstützung für die Implementierung von MCP-Clients und MCP-Servern basierend auf dem offiziellen MCP Java SDK. Da dieses Thema den Rahmen dieses Artikels sprengt, wird es hier nicht weiter behandelt.

Retrieval-Augmented Generation

Eine weitere fortgeschrittene Technik zur Verbesserung der Antworten von LLMs durch externe Daten ist Retrieval-Augmented Generation

(RAG). Hierbei werden externe Daten durch Embedding-Modelle als Vektoren repräsentiert und in einer Vektor-Datenbank gespeichert. Um relevante Informationen abzurufen, wird auch die Benutzeranweisung in Vektoren konvertiert. Anschließend werden semantisch passende Daten aus der Datenbank abgerufen und in den Kontext des LLMs integriert, um die Antworten zu verbessern.

Um RAG in seiner einfachsten Form zu implementieren, müssen für den Use-Case relevante Daten zuerst in eine Vektor-Datenbank integriert werden. Spring AI erleichtert diesen Prozess durch eine Extract-Transform-Load (ETL) Pipeline. Zunächst extrahiert ein DocumentReader Inhalte aus verschiedenen Quellen wie PDFs und wandelt sie in strukturierte Document Objekte um. Anschließend werden diese Dokumente mit einem DocumentTransformer unterteilt, um den Kontextfensterbeschränkungen der KI-Modelle zu genügen. Schließlich speichert ein DocumentWriter, der durch das VectorStore Interface erweitert wird, diese Dokumentenfragmente in einer Vektor-Datenbank.

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

Das VectorStore Interface stellt eine Abstraktionsschicht bereit, die eine flexible Integration verschiedener Vektor-Datenbanken mit minimalen Code-Änderungen ermöglicht. Jede unterstützte Vektor-Datenbank verfügt über eine eigene Spring Boot Starter Bibliothek. In diesem Beispiel wird PGvector, eine Erweiterung für die Speicherung von Vektoren in PostgreSQL, verwendet.

```
implementation ‚org.springframework.ai:spring-ai-pgvector-store-
spring-boot-starter‘
```

Eine VectorStore Instanz in Spring AI benötigt ein Embedding-Modell, das über die EmbeddingModel API verfügbar ist. Da die meisten KI-Anbieter bereits Embedding-Modelle anbieten, sind in der Regel keine zusätzlichen Abhängigkeiten erforderlich. Dank der automatischen Konfiguration von Spring Boot ist sowohl eine VectorStore als auch eine EmbeddingModel Instanz mit minimalem Setup verfügbar.

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

Die Unterstützung von RAG in Spring AI basiert auf der Advisor API, mit der Entwickler die Kommunikation zwischen der Anwendung und LLMs abfangen und anpassen können. Für gängige RAG Workflows bietet Spring AI den QuestionAnswerAdvisor und, mit einer modularen Architektur, den RetrievalAugmentationAdvisor an. Mit ihm lassen sich fortgeschrittene RAG-Workflows realisieren, die z.B. Routing zwischen mehreren Vektor-Datenbanken nutzen.

Advisors können für eine spezifische ChatClient API Interaktionen über die advisors() Methode konfiguriert werden oder global über den ChatClient.Builder.

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

In diesem Beispiel wird ein QuestionAnswerAdvisor mit einer automatisch konfigurierten VectorStore Instanz verwendet. Über einen optionalen SearchRequest Parameter kann feinjustiert werden, wie relevante Daten innerhalb der Vektor-Datenbank gesucht werden. Ein weiterer Parameter ermöglicht es, dem LLM zusätzliche Anweisungen zu geben, wie es die abgerufenen Daten im Kontext interpretieren und nutzen soll.

KI-Modelle für Bilder und Audio

Die ChatClient API basiert auf der Model API, die unterschiedliche Arten von KI-Modellen, wie neben Text z.B. Embedding-, Bild- und Audiomodelle, unterstützt. Während die ChatClient API sinnvoll ist, um komplexe Prompts für Chat-Modelle zu vereinfachen, erfordern Bild- und Audiomodelle in der Regel weniger aufwändige Prompts. In diesen Fällen reicht die direkte Interaktion mit der Model API aus. Wie gewohnt werden durch die automatische Konfiguration von Spring Boot Model API Instanzen für alle Modelltypen vorkonfiguriert und so für Dependency Injection bereitgestellt.

```
String prompt = new PromptTemplate(„Generate a picture of {city}“)
        .render(Map.of(„city“, „Berlin“));
ImageGeneration imageGeneration = imageModel.call(prompt).
getResult();
Image image = imageGeneration.getOutput();
String imageUrl = image.getUrl();
```

Fazit

Zusammenfassend ermöglicht Spring AI es Ihnen Gen-AI-Funktionen, ähnlich wie beispielsweise Datenbanken, mit minimalem Aufwand in Spring Boot Anwendungen zu integrieren. Es unterstützt alle gängigen KI-Modelle mit einem Fokus auf Portabilität und bietet grundlegende Funktionen wie strukturierte Ausgaben und Multimodalität. Darüber hinaus stellt Spring AI die wesentlichen Bausteine für Agenten bereit und unterstützt das Model Context Protocol, das eine reibungslose Integration mit Drittanbieter Tools und Ressourcen ermöglicht. Durch diesen Funktionen stellt Spring AI sicher, dass Sie mit der Geschwindigkeit, der sich stetig weiterentwickelnden KI-Technologien mithalten können.

> Zurück zum Inhaltsverzeichnis

ENTDECKE WEITERE SPANNENDE ARTIKEL ONLINE:
JAVAPRO
www.javapro.io

www.javapro.io

#JAVAPRO #AI #ML

Lokale LLMs mit Ollama und Open WebUI

Autor:

Jean-Claude Brantschen ist ein erfahrener Softwareentwickler mit nachweislicher Erfolgsbilanz in der Softwarebranche. Kompetent in Software Crafting, Java, Python und SQL.

https://www.linkedin.com/in/jean-claude-brantschen-427738114/

Seit dem kometenhaften Aufstieg von ChatGPT sind AI (Artificial Intelligence) und LLMs (Large Language Models) in aller Munde. Es gibt kaum jemanden, der ChatGPT (oder einer seiner Kollegen) nicht im beruflichen oder privaten Umfeld nutzt. Was weniger bekannt ist, dass man LLMs auch herunterladen und lokal laufen lassen kann.

Diese hat folgende Vorteile:

- Verschiedene LLMs und Versionen lassen sich flexibel testen.
- Da alles lokal läuft, können auch Anfragen mit geschäftsrelevanten Informationen gestellt werden, ohne dass Daten auf fremde Server gelangen.

Eine Möglichkeit dafür bietet das Open-Source-Projekt Ollama: Es erlaubt, kleinere LLMs herunterzuladen und direkt auf dem lokalen Rechner auszuführen. Dieser Artikel zeigt zunächst die Installation von Ollama. Anschließend wird ein benutzerfreundliches Web-UI eingerichtet, das den Zugriff auf Ollama deutlich komfortabler gestaltet.

Lokale Installation von Ollama

Auf https://ollama.com/download stehen Installationspakete für Mac, Windows und Linux bereit. Eine Übersicht der unterstützten LLMs findet sich unter https://ollama.com/library.

Es gibt LLMs verschiedener bekannter Tech Konzerne. Unter anderem:

- **Llama** von Meta/Facebook
- **Phi** von Microsoft
- **Gemma** von Google

Nach der Installation von Ollama lassen sich LLMs direkt über die Kommandozeile herunterladen. Hier gezeigt am Beispiel von Llama (in der Version 3.2):

```
>> ollama pull llama3.2
```

Welche LLMs bereits heruntergeladen wurden und wie groß sie sind, lässt sich über die Kommandozeile anzeigen:

```
>> ollama list
```

Eine heruntergeladene LLM lässt sich direkt in der Konsole starten und dort für Anfragen verwenden:

```
>> ollama run llama3.2
```

Beim Aufruf von run kann man zusätzlich die Anfrage mitgeben. Ein Beispiel: Die Frage, wie sich in Java eine Collection umdrehen lässt, lässt sich mit folgendem Aufruf formulieren:

```
>> ollama run llama3.2 how can I reverse a collection in java
```

Funktioniert gut. Via Docker können wir uns noch ein UI installieren, so dass die Anfragen noch komfortabler gestellt werden können.

Installation UI zu Ollama via Docker

Open WebUI https://docs.openwebui.com bietet ein Web UI für Ollama an. Details zur Installation findet man unter: https://docs.openwebui.com/getting-started

Folgender Command installiert das UI und startet einen lokalen Webserver:

```
>> docker run -d \
-p 3000:8080 \
--add-host=host.docker.internal:host-gateway \
-v open-webui:/app/backend/data \
--name open-webui \
--restart always \
ghcr.io/open-webui/open-webui:main
```

Hier ein paar Erklärungen zu den Parametern beim Docker Aufruf:

- docker run -d

- Startet den Container (mit dem UI) im Hintergrund.

- p 3000:8080

- Mappt den Port 8080 im Docker Container auf Port 3000 auf dem lokalen Rechner.

- add-host=host.docker.internal:host-gateway

- host.docker.internal ist der standardmäßige Container-Hostname, der durch Docker definiert wird. Dieser wird nun an die gleiche IP-Adresse des Hosts-Gateways gebunden.

- v open-webui:/app/backend/data

- Ein kurzer Einschub: was sind Docker Volumes und warum brauchen wir diese. Docker Volumes sind ein Mechanismus zur persistenten Datenspeicherung in Docker (eine Art „virtueller" Ordner auf dem Host-System, der für die Daten der Anwendung im Container verwendet wird). Diese Daten bleiben bestehen, wenn man einen

Container stoppt oder löscht.

- Erstellt ein Docker Volume mit dem Namen open-webui auf dem lokalen Rechner und mappt dieses auf den Ordner /app/backend/data im Container. Dies dient zum Austausch von Daten zwischen lokalem Rechner und Docker. Durch Aufruf von docker volume inspect open-webui bekommt man mehr Informationen zum Volume.

- name open-webui

- Name des Containers

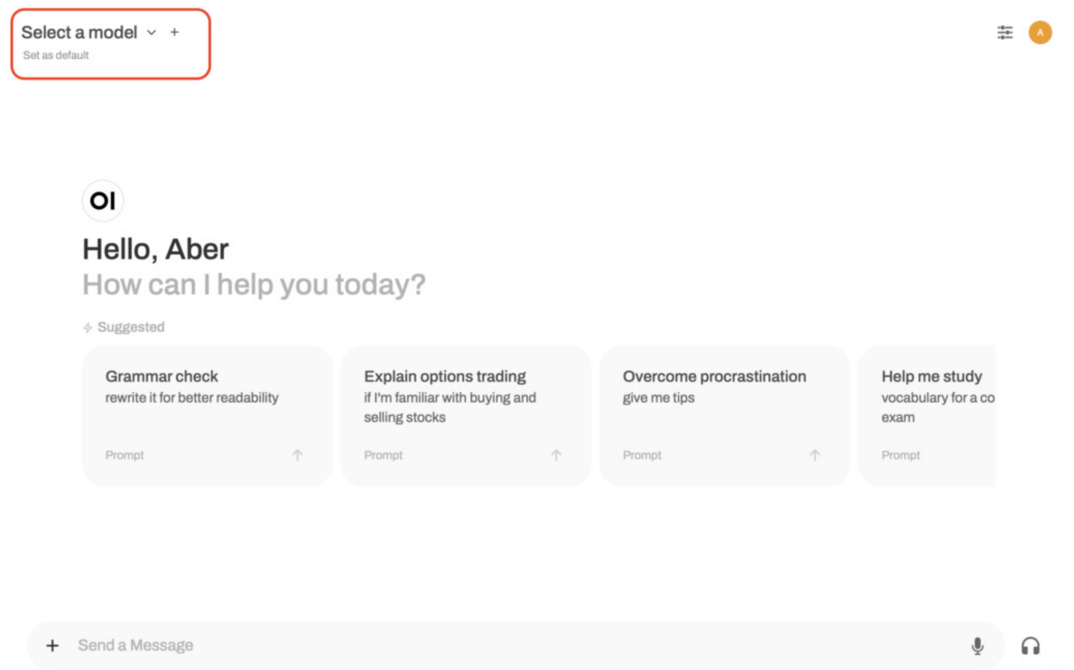
- ghcr.io/open-webui/open-webui:main

- Docker Image Name für das Open WebUI

Danach bekommt man Zugriff auf das UI direkt im Browser via http://localhost:3000

Beim ersten Start ist eine Registrierung mit Name, E-Mail und Passwort erforderlich. Aber das ist nur für die lokale Installation. Dummy Werte genügen völlig für eine erfolgreiche Installation.

Danach wird auf die Hauptseite weitergeleitet. Oben links unter "Select a model" lässt sich eines der lokal über Ollama installierten LLMs auswählen.



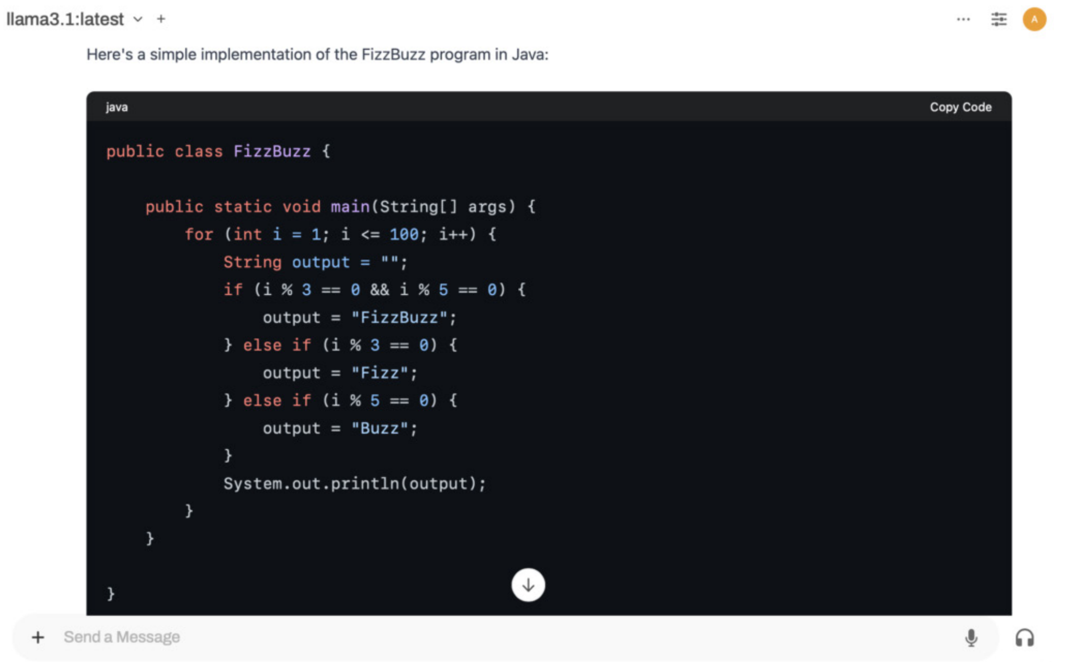
Zeit für die erste Anfrage an die lokale LLM: Das Modell „llama“ soll Java-Code für FizzBuzz generieren. FizzBuzz ist folgendermaßen definiert (und in der Welt der Software Entwickler ziemlich bekannt).

Schreibe ein Programm, das die Zahlen von 1 bis 100 ausgibt. Aber für Vielfache von 3 gebe „Fizz“ anstelle der Zahl aus und für Vielfache von 5 gebe „Buzz“ aus. Für Zahlen, die ein Vielfaches von 3 und 5 sind, gebe „FizzBuzz“ aus.

Fragen wir doch mal “llama”. Die Eingabe von “fizzbuzz in Java” sollte dazu genügen:

- llama kennt FizzBuzz, generiert Java Code dafür und liefert noch zusätzliche Infos.
- Das Fenster mit dem Code hat einen “Copy Code” Link der den Code direkt ins Clipboard kopiert.
- Und das UI kommt in einem ChatGPT ähnlichen Design daher.

Ziemlich cool, lokal und gratis !!!!



Integration von ChatGPT

Es gibt auch die Möglichkeit ChatGPT (OpenAI) ins Open WebUI zu integrieren. OpenAI ist eigentlich schon integriert in Open WebUI, muss aber noch konfiguriert werden.

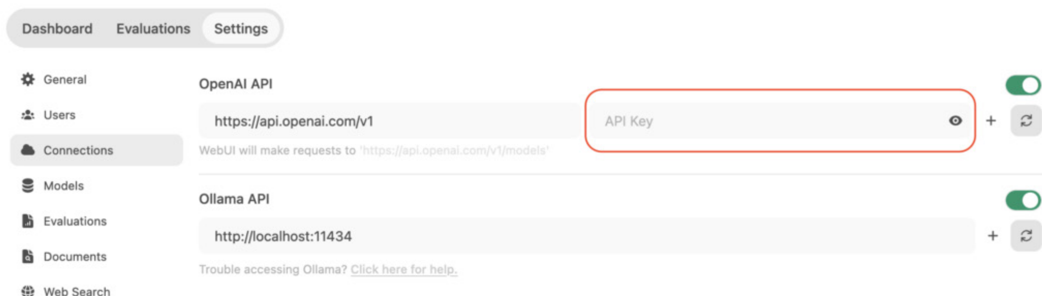
Als ersten Schritt wird ein OPEN_API_KEY benötigt. Der API-Schlüssel wird nach der Registrierung unter https://platform.openai.com und der Hinterlegung einer Kreditkarte bereitgestellt.

Anfragen an ChatGPT sind kostenpflichtig. Bezahlt wird aber per Anfrage. Die Kosten sind für einen einzelnen Benutzer kleiner als ein normales monatliches Abonnement bei ChatGPT.

Für die Integration gibt es 2 Möglichkeiten:

Konfiguration in Open WebUI

Es gibt die Möglichkeit, Docker wie bisher zu starten und in Open WebUI den OPEN_API_KEY zu konfigurieren. Unter "Admin Panel > Settings > Connections > OpenAI API" lässt sich der kopierte OPEN_API_KEY einfügen. Unter "Select a model" stehen dann verschiedene ChatGPT LLMs zur Verfügung.



Konfiguration in Docker

Es gibt aber auch die Möglichkeit, OpenAI direkt beim Docker Aufruf zu konfigurieren.

```
>> docker run -d
-p 3000:8080
-e OPENAI_API_KEY=your_secret_key
-v open-webui:/app/backend/data
--name open-webui
--restart always
ghcr.io/open-webui/open-webui:main
```

Es gibt einen zusätzlichen Parameter -e
- e OPENAI_API_KEY=your_secret_key

- Einfach your_secret_key ersetzen durch den eigenen OPEN_API_KEY.

Einschub Security: Aus Security Gründen wäre es empfehlenswert, den OPEN_API_KEY nicht direkt beim Docker Aufruf mitzugeben, sondern in einer Environment Variable abzulegen und diese dann zu benutzen. Auf einem Mac (unter OSX) kann ein Variable für den OPEN_API_KEY im File „.zprofile“ definiert und exportiert werden:

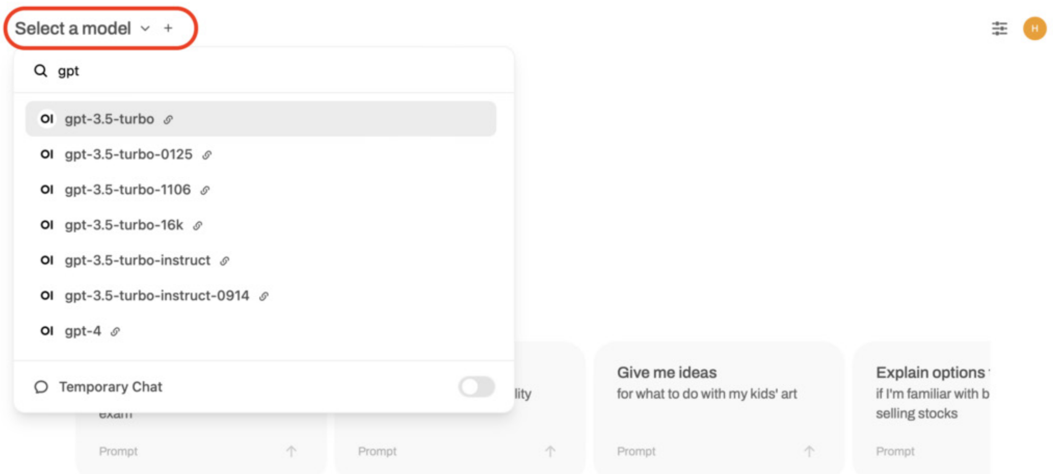
```
export OPENAI_API_KEY=lorem_ipsum_dolor_sit_amet
```

und diese dann beim Aufruf verwendet werden

```
>> docker run -d
-p 3000:8080
-e OPENAI_API_KEY=$OPENAI_API_KEY
-v open-webui:/app/backend/data
--name open-webui
--restart always
ghcr.io/open-webui/open-webui:main
```

Wie dies auf anderen Betriebssystemen funktioniert, lässt sich am besten über eine Suchmaschine oder mithilfe einer lokal installierten LLM herausfinden.

Im Anschluß stehen unter “Select a model” verschiedene ChatGPT LLMs zur Verfügung.
Aber Vorsicht: ChatGPT ist ins Open WebUI integriert, aber ChatGPT läuft nicht lokal. Die Request werden an einen externen Server weitergeleitet.



Performance:

Wie sieht es eigentlich mit der Performance aus? Das kommt auf die Grösse der ausgewählten LLM und der eigenen Hardware an. Die Hardware-Anforderungen für Ollama habe ich direkt beim Anbieter erfragt. Die Antwort: Intel Core i9 mit 16 GB RAM als Minimum und GPUs wären schön (diese werden von Ollama direkt unterstützt). Ja nach ausgewählter LLM können sich die Anforderungen erhöhen.

Zur Anschauung habe ich die gleiche Abfrage auf meinen aktuellen und meinem „legacy“ Laptop gestartet und verglichen.

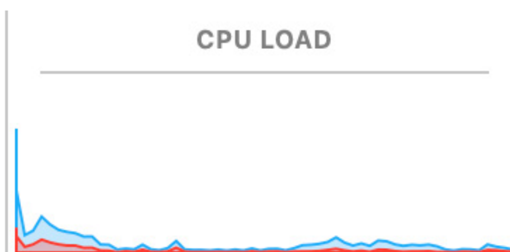
- Als LLM verwende ich llama.3.2 in der grösseren 3B Version (3B heisst, das Modell verfügt über 3 Milliarden Parameter). Zusätzlich gibt es noch eine kleiner 1B Variante.
- Wir starten folgende Anfrage direkt in Ollama (ohne den Umweg über Open WebUI):

```
>> ollama run llama3.2 how can I reverse a collection in java
```

Hardware „Legacy“ Laptop

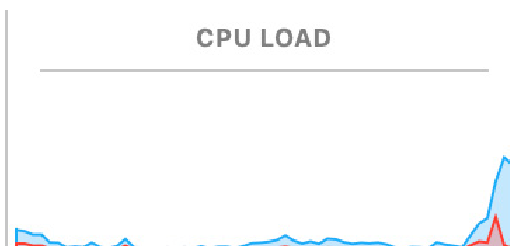
- 5+ Jahre Jahre alt
- Hardware: Dual-Core Intel Core i7 mit 16GB RAM
- Vor der Anfrage hat die CPU ein Auslastung von unter 2%

System:	1.45%
User:	1.36%
Idle:	97.18%



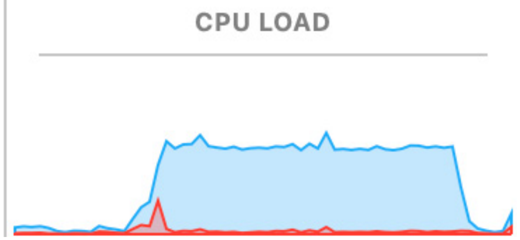
- Beim Start der Anfrage steigt die CPU Auslastung auf ca. 50%

System:	2.10%
User:	53.03%
Idle:	44.87%

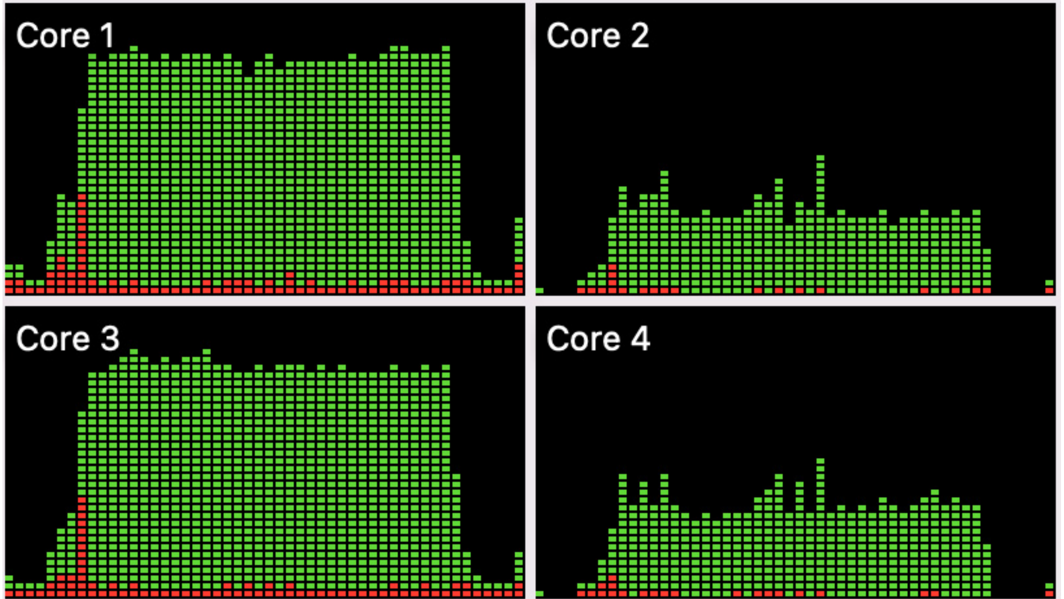


- Nach der vollständigen Verarbeitung der Anfrage sinkt die CPU Auslastung wieder auf den ursprünglichen Wert zurück.

System:	5.54%
User:	8.43%
Idle:	86.02%



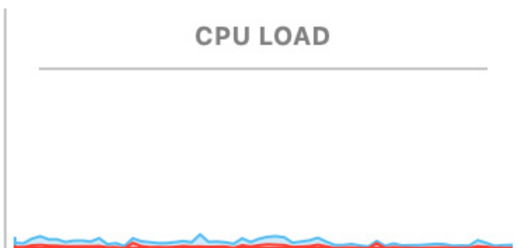
- Die CPU History zeigt, dass 2 Cores für die Bearbeitung der Anfrage ausgelastet sind. Die Anfrage dauert (basierend auf mehreren Versuchen) zwischen 40s und 60s.



Hardware „aktueller“ Laptop

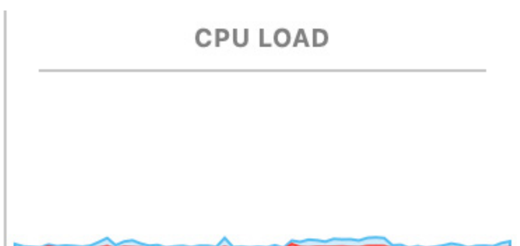
- 1+ Jahr alt
- Hardware: M Prozessor mit 32+GB RAM
- Vor der Anfrage hat die CPU ein Auslastung von unter 2%

System:	1.00%
User:	1.99%
Idle:	97.02%

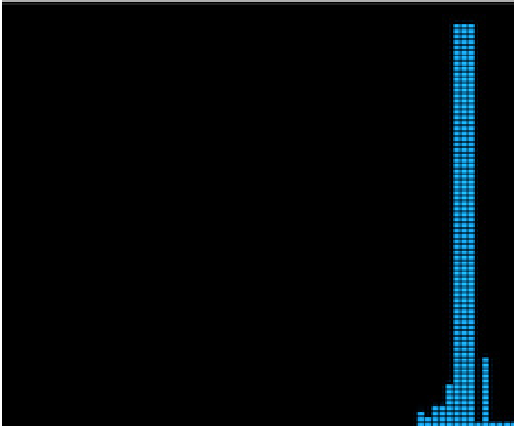


- Beim Start der Anfrage bleibt die CPU Auslastung fast gleich.

System:	2.21%
User:	3.94%
Idle:	93.85%



- Gleichzeitig geht die GPUs Nutzung kurzzeitig nach oben. D.h. die ganze Arbeit wird von den GPUs gemacht und die CPUs merken davon praktisch nichts. Die Zuweisung erfolgt dabei automatisch durch Ollama. Nach der vollständigen Verarbeitung der Anfrage geht die GPU Auslastung wieder auf den ursprünglichen Wert zurück.



Die Anfrage dauert um die 6 Sekunden.

So viel steht fest: Spass macht es nur auf einem Laptop mit aktueller Hardware. Kleinere LLMs könnte man eventuell auch auf einem älteren Laptop laufen lassen, aber das muss man einfach selber ausprobieren.

Fazit:

AI und LLMs sind ein spannendes Thema. Für Softwareentwickler ist jetzt der ideale Moment, in diese Welt einzutauchen. Wer mit lokalen LLMs experimentieren möchte, findet in Ollama einen leicht zugänglichen und günstigen Einstiegspunkt. Man gewöhnt sich schnell daran. Der Umgang damit geht schnell in Fleisch und Blut über: In meinem eigenen Arbeitsalltag nutze ich es inzwischen regelmäßig – oft als produktive Alternative zur klassischen Google-Suche.

> Zurück zum Inhaltsverzeichnis

ENTERPRISE

MicroStream

OPEN-SOURCE

ECLIPSE FOUNDATION

ECLIPSE DATA GRID

ENTERPRISE

Next Generation Caching & In-Memory Searching.

1000x Faster than a Database.
Runs Between Your Application & Database.

- Caching
- In-Memory Searching
- Indexing
- Native Java Object Model
- Any Logic in Java
- ACID Persistence
- Distributed
- Open-Source
- SaaS

github.com/eclipse-datagrid

Eclipse Data Grid is an in-memory data processing layer to boost any sluggish database applications and relieve the database. Eclipse Data Grid is much more than a common cache. It enables ultra-fast in-memory data processing: caching, indexing, searching, and any complex data operations - up to 1000x faster than databases. Unlike traditional caching solutions, Eclipse Data Grid is a native Java layer using the native Java object model. This allows you to work with native Java objects, complex Java object graphs, and any Java types, and to integrate any Java libraries and implement any complex logic in your in-memory data layer with Java. Eclipse Data Grid is distributed, highly horizontally scalable and completely Open-Source. It's built for everyone who needs more than just a cache. Move your complex and performance-critical data and data operations to Eclipse Data Grid to dramatically reduce database workloads and save costs, boost your application and your business.

Contact Us:
www.microstream.one • hello@microstream.one
LinkedIn: linkedin.com/company/microstream

Book a Call:
www.microstream-intro.youcanbook.me

MicroStream

#JAVAPRO #PERFORMANCE

Hitchhiker´s Guide to Java Performance

Autor:

Ingo Düppe ist IT-Berater und Gründer von CROWDCODE. Er berät im Bereich Architekturmanagement und ist verantwortlich für die Konzeption und Umsetzung von Web-, Mobile- und Cloud-Anwendungen. Die Herausforderung besteht darin, die passende Kombination aus Technologien, Methoden und Werkzeugen für den jeweiligen Kontext zu finden. Als Coach unterstützt er Teams beim Einsatz von Java-EE- und Spring-Technologien sowie deren effizienter Integration in den individuellen Softwareentwicklungsprozess.

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

Vergangenheit, Gegenwart und Zukunft

In den letzten 30 Jahren hat sich Java von einer exotischen „write once, run anywhere"-Sprache zu einer der weltweit dominierenden Plattformen für die Softwareentwicklung entwickelt. In den Anfangsjahren galt Java im Vergleich zu Sprachen wie C/C++ zu Recht als langsam, was vor allem auf den anfänglichen Interpreter-Ansatz zurückzuführen war. Die letzten 30 Jahre haben bewiesen, dass das VM-Konzept mit der adaptiven Optimierung der HotSpot-Engine langfristig die effizientere Lösung ist.

Frühere JVM-Versionen führten Bytecode rein interpretiert aus, was Java-Programme 10- bis 20-mal langsamer machte als entsprechenden C-Code. Die Performance war also von Anfang an eine Herausforderung, aber kontinuierliche Optimierungen haben die Ausführungsgeschwindigkeit seitdem drastisch verbessert.

Deshalb werfen wir in diesem Artikel einen detaillierten Blick auf die Evolution der Java-Performance. Wir blicken zurück in die Vergangenheit - von den ersten JVMs der 90er Jahre bis zur Einführung des JIT-Compilers und der ersten Garbage-Collector-Strategien. Anschließend schauen wir auf die Gegenwart moderner JVMs: die HotSpot-Engine, aktuelle Tiered Compiler (C1, C2 und GraalVM), fortschrittliche GCs wie G1, Shenandoah und ZGC, Verbesserungen beim Threading (z. B. Project Loom) und Speicheroptimierungen.

Ein Blick in die Zukunft zeigt, was Entwickler mit Loom, CRaC, nativer Kompilierung und neuen Projekten erwarten können. Zum Schluss noch ein paar praktische Tipps, um sich im Java-Performance-Universum nicht zu verirren.

Vergangenheit: Die Anfänge der Java-Performance

Die allerersten Java-Versionen (JDK 1.0 und 1.1 in den Jahren 1996–1997) setzten ausschließlich auf Interpreter. Der Java-Bytecode wurde also zur Laufzeit Befehl für Befehl emuliert, anstatt in nativen Maschinencode übersetzt zu werden. Dieser Ansatz gewährleistete die Portabilität, führte aber zu einem erheblichen Overhead. Durchschnittliche Java-Anwendungen liefen anfangs 10- bis 20-mal langsamer als in C geschriebene Programme. Entwickler spotteten daher in den 90er Jahren oft über Java als „zu langsam für ernsthafte Anwendungen“.

Ein Just-in-Time-Compiler (JIT) wurde erstmals 1997 mit Java 1.1 eingeführt. Der JIT-Compiler übersetzt häufig ausgeführte Bytecode-Sequenzen dynamisch in nativen Code, um die Ausführung erheblich zu beschleunigen. Allerdings war die frühe JIT-Kompilierung selbst rechenintensiv, so dass sie zunächst nur begrenzt, genutzt werden konnte. Der eigentliche Wendepunkt kam vom HotSpot-Team: Ihre Technologie wurde 1999 zunächst als Option für Java 1.2 zur Verfügung gestellt und wurde ab Java 1.3 (2000) als HotSpot JVM zum Standard. HotSpot

führte die adaptive Optimierung ein: Die JVM beobachtet den Code zur Laufzeit, identifiziert „Hot Spots" (häufig verwendete Codepfade) und kompiliert diese selektiv mit dem JIT, während weniger kritischer Code in interpretierter Form verbleiben kann. Dieser selektive Ansatz reduzierte den Kompilierungsaufwand erheblich und führte zu einer bis zu zehnfachen Leistungssteigerung gegenüber rein interpretiertem Code, wie Benchmarks zeigen.

Ebenfalls erwähnenswert aus dieser Ära ist das Threading-Modell. Die Plattformunabhängigkeit führte zunächst zu Einschränkungen bei der Verwendung von Betriebssystem-Threads. Java 1.1 verwendete auf einigen Plattformen Green Threads - d. h. Threads, die vom JVM-Laufzeitsystem im User Space verwaltet wurden, statt native Betriebssystem-Threads. Obwohl dies Threading auch auf Systemen ohne eigene Thread-Unterstützung ermöglichte und geringe Kosten für Thread-Switches verursachte, war es auf Multiprozessorsystemen nicht skalierbar. Alle Green Threads eines Prozesses teilten sich einen Kernel-Thread, so dass eine blockierende Operation (z. B. I/O) die gesamte VM zum Stillstand bringen konnte.

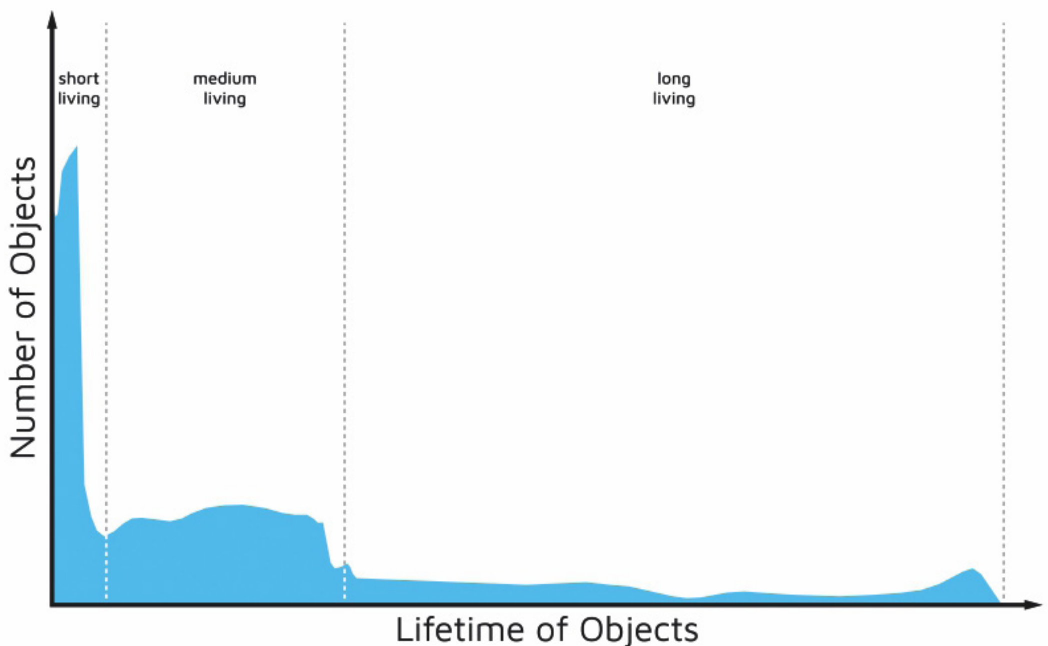
Mit Java 1.3 vollzog die JVM den Wechsel zu nativen Threads, wodurch die Parallelität auf Multi-Core-Systemen erheblich verbessert wurde – allerdings auf Kosten leicht erhöhter Aufwände für die Thread-Erstellung und den Kontextwechsel. Interessanterweise ist Java kürzlich mit Project Loom zum Konzept der ultraleichten, laufzeitverwalteten Threads zurückgekehrt - mehr dazu später.

Memory Management und Garbage Collection (GC) in den 90ern

Ein weiteres wichtiges Leistungsproblem war in den ersten Tagen die Memory Management. Java befreite die Entwickler von der manuellen Speicherzuweisung und -freigabe und reduzierte so die Programmierfehler, übertrug aber die Verantwortung für die Speicherbereinigung an die JVM. Die allerersten JVMs (1.0, 1.1) verwendeten eine einfache Mark-and-Sweep-GC, die den Heap regelmäßig überprüfte, unbenutzte Objekte als Garbage markierte und ihren Speicher freigab. Dieses Verfahren führte häufig zu einer Fragmentierung des Heaps. Darüber hinaus wurde die Sammlung mit einem Stop-the-World-Szenario durchgeführt - mit

anderen Worten, die gesamte Anwendung wurde angehalten, während der GC lief, was zu spürbaren Verzögerungen führte. Darüber hinaus skalierte der GC nicht mit zunehmendem Speicher.

Mit Java 1.2 (1998) wurde die generationale GC eingeführt. Die Weak Generational Hypothesis besagt, dass die meisten Objekte sehr kurzlebig sind und nur wenige Objekte sehr langlebig. Die JVM teilte den Heap entsprechend in eine junge und eine alte Generation ein und bereinigte die junge Generation (mit dem Großteil des kurzlebigen Mülls) sehr häufig und die alte Generation entsprechend seltener. Zusätzlich konnte die Fragmentierung durch das Kopieren von Kollektoransätzen reduziert werden. Dieses generationale GC-Design erwies sich als deutlich leistungsfähiger. Es sorgte zum Beispiel dafür, dass große Heaps besser verdichtet und Speicherlecks vermieden wurden. Trotz dieser Fortschritte blieben GC-Pausen in großen Java-Anwendungen der späten 1990er Jahre ein Problem - die Nebenläufigkeit in der GC selbst war noch rudimentär, was dazu führte, dass die Anwendung während der Speicherbereinigung erheblich stockte.



Weak Generational Hypothesis – Die meisten Objekte sterben jung

Die Herausforderung der Plattformunabhängigkeit

Alles in allem waren die späten 1990er Jahre eine Zeit, in der die grundlegenden Leistungstechniken für Java gerade ausgereift waren. Viele Herausforderungen ergaben sich direkt aus der Plattformunabhängigkeit: Jeder Vorteil musste zur Laufzeit und ohne tiefe Integration in das Betriebssystem erreicht werden. Die frühen Java-

Versionen hatten mit einer trägen GUI-Leistung zu kämpfen, da AWT und Swing auf plattformneutralen Code setzten, anstatt native Widgets zu nutzen. File- und Network-I/O waren aufgrund von abstrakten Streams und Sicherheitsüberprüfungen deutlich langsamer als bei C-APIs. Der JNI-Overhead machte native Bibliotheken ineffizient, weshalb leistungsrelevanter Code oft in C/C++ geschrieben wurde.

Unterschiede im Thread Scheduling erschwerten eine optimale Abstimmung, da die frühen JVMs Java-Threads nicht 1:1 auf Betriebssystem-Threads abbildeten. Dennoch wurde in diesen Jahren der Grundstein für spätere Optimierungen gelegt: Mit JIT, Generational GC und nativen Threads wurde Java um das Jahr 2000 deutlich schneller und skalierbarer, ein Trend, der die Plattform bis heute prägt.

Gegenwart: Optimierte Leistung in modernen JVMs

Die heutigen Java-Versionen (Java 17 bis 23+) verfügen über hoch optimierte virtuelle Maschinen, die das Ergebnis jahrzehntelanger Forschung und Entwicklung sind.

HotSpot-Engine und adaptive Optimierung

Seit über 20 Jahren ist die HotSpot JVM die Grundlage der Java SE-Plattform. Wie ihr Name andeutet, erkennt sie Hotspots im Code und optimiert diese gezielt zur Laufzeit. Sie kombiniert einen Bytecode Interpreter mit zwei JIT-Compilern: C1, der mit begrenzten Optimierungen schnell kompiliert, und C2, der hoch optimierten Maschinencode für langlaufende Serveranwendungen erzeugt.

Bei der abgestuften Kompilierung werden beide Compiler stufenweise eingesetzt: Methoden werden zunächst interpretiert, C1 übernimmt nach einigen Aufrufen, und häufig verwendeter Code wird schließlich von C2 optimiert. Dieser Ansatz ermöglicht eine schnelle Aufwärmphase und maximale Leistung. Während der Ausführung optimiert HotSpot dynamisch mit Techniken wie Inlining-, Loop Collapse- und Peephole-Optimierungen und passt sich dabei an Laufzeitprofile an. Spekulative Optimierungen machen Annahmen über den Code, die automatisch zurückgenommen werden, wenn sie sich als ungültig erweisen.

Ein Meilenstein war die Einführung der Escape-Analyse in Java 6/7. Diese Technik identifiziert Objekte, die einem bestimmten Scope nicht entkommen, und optimiert deren Speicherverwaltung: Objekte können auf dem Stack statt auf dem Heap alloziert oder durch skalare Ersetzung komplett eliminiert werden, wodurch die GC-Last reduziert wird.

Dank dieser Optimierungen erreicht HotSpot Java in vielen Szenarien eine nahezu native Leistung. Bereits 2008 haben Benchmarks gezeigt, dass Java dank fortschrittlicher JIT-Techniken nur 10 bis 30 % hinter C++ zurückbleibt und aufgrund von Laufzeitoptimierungen, die statischen Compilern nicht zur Verfügung stehen, manchmal mit C++ gleichzieht oder es sogar übertrifft.

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

Die Garbage Collector der JVM haben in den letzten Jahren bedeutende Fortschritte gemacht. Neben den klassischen Serial- und Parallel-GCs führte CMS erstmals nebenläufige GC-Mechanismen ein, hatte jedoch Schwächen wie Heap-Fragmentierung. Mit Java 7/8 wurde G1 GC als moderner Ersatz eingeführt und in Java 9 zum Standard, während CMS in Java 14 wieder entfernt wurde.

G1 GC arbeitet auf regionaler Basis und wählt die speicherintensivsten Regionen für die Bereinigung aus, um die konfigurierbaren Pausenzeiten (standardmäßig etwa 200 ms) in Grenzen zu halten. Durch gleichzeitige Markierung und selektive Räumung schafft G1 ein gutes Gleichgewicht zwischen Durchsatz, moderaten Pausen und minimalem Abstimmungsaufwand, wodurch es sich besonders für mittlere bis große Heaps eignet.

Für noch niedrigere Latenzen wurden Shenandoah und ZGC entwickelt. Shenandoah reduziert die Stop-the-World-Pausen, indem es eine vollständig gleichzeitige Heap-Komprimierung unter Verwendung von Brooks-Zeigern durchführt und die Pausen selbst für 200-GB-Heaps im Sub-10-ms-Bereich hält - wenn auch in den frühen Versionen ohne Generationsverhalten. ZGC verfolgt einen ähnlichen Ansatz mit farbigen Zeigern, die Zustandsinformationen direkt in Zeiger einbetten, um Objektbewegungen effizient zu verwalten. Es garantiert Pausen unter

10 Millisekunden, unabhängig von der Heap-Größe. Mit Java 21 wurde Generational ZGC eingeführt, um kurzlebige Objekte effizienter zu sammeln und den Durchsatz weiter zu steigern.

Heutzutage stellt die JVM eine vielseitige Auswahl an Garbage-Collector-Algorithmen bereit, die gezielt auf die individuellen Anforderungen einer Anwendung abgestimmt werden können. Während Oracle G1 als Standard verwendet und für die Zukunft auf generationale ZGC setzt, bevorzugt Red Hat häufig Shenandoah. Dank dieser Fortschritte kann Java selbst mit Terabyte-großen Heaps effizient arbeiten, ohne dass lange Stop-the-World-Pausen zum Problem werden.

Threading und Concurrency: Von Fork/Join zu Loom

Seit Java 5 (2004) und dem java.util.concurrent-Paket hat sich in Sachen Parallelisierung viel getan: Thread-Pools, Sperren, atomare Variablen und nebenläufige Collections ermöglichen den modernen Multicore-Einsatz. Mit Java 7 kam das fork/join-Framework (JSR 166y) zur rekursiven Parallelisierung hinzu, und ab Java 8 wurde dieser Ansatz durch parallele Streams und Lambdas weiter vereinfacht. Gleichzeitig wurden Verbesserungen wie Lock-Striping und effizientere Datenstrukturen (wie die ConcurrentHashMap mit blockweisem Locking) entwickelt. In Java 11 folgten varHandle und Spin-Wait-Hinweise über Thread.onSpinWait(), die eine High-Performance-Concurrency ermöglichen. Ein typisches Beispiel sieht wie folgt aus:

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

Darüber hinaus ist dank der Lock-Elimination über die Escape-Analyse keine Synchronisierung mehr erforderlich, wenn Objekte nur thread-local verwendet werden.

Dennoch bleibt eine Hürde: Jeder Java-Thread entspricht einem Betriebssystem-Thread, was zu hohem Speicherbedarf (meist 1 MB pro Stack/Thread) und teuren Kontextwechseln bei sehr großen Mengen (ca. über 100k) führt. Das Scheduling skaliert dann nicht mehr linear, und die Grenzen des Betriebssystems werden erreicht.

Dies war eine der treibenden Kräfte hinter Project Loom, das wohl die bedeutendste Umwälzung im Java-Concurrency-Modell seit Jahrzehnten darstellt. Mit Project Loom wurden virtuelle Threads (auch bekannt als Fibers) eingeführt, die seit Java 19 als Vorschau verfügbar waren und in Java 21 fertiggestellt wurden.

Virtuelle Threads sind ultraleichte, vom JDK verwaltete Threads, die nicht permanent an Kernel-Threads gebunden sind. Sie können als modernes Äquivalent zu Green Threads betrachtet werden - allerdings ohne deren Nachteile.

Die Idee ist, dass ein Java-Programm Hunderttausende von gleichzeitigen Threads erstellen kann, ohne das Betriebssystem zu überlasten. Virtuelle Threads werden von der JVM geplant und auf eine kleinere Anzahl von Kernel-Threads (sogenannte Carrier-Threads) abgebildet. Wenn ein virtueller Thread blockiert (z. B. während I/O oder Thread.sleep), wird nicht der gesamte Träger-Thread blockiert - die JVM kann den Träger entkoppeln und einem anderen virtuellen Thread zuweisen, während der blockierte Thread im Hintergrund wartet. Dadurch werden blockierende Aufrufe praktisch automatisch asynchron, ohne dass der Programmierer komplexe Callback- oder Future-Logik schreiben muss.

In der Praxis ermöglicht Loom eine hohe Skalierung des bekannten Thread-per-Request-Modells. Bisher musste man bei der Verarbeitung von Tausenden von gleichzeitigen Verbindungen (z. B. in einem Webserver) auf asynchrone I/O oder reaktive Programmierung (Netty, Vert.x usw.) zurückgreifen, um zu vermeiden, dass eine gleiche Anzahl von Betriebssystem-Threads blockiert wird. Mit virtuellen Threads kann nun jede Anfrage in einem eigenen (virtuellen) Thread bearbeitet werden - mit einfachem, synchronem Code - während die JVM dafür sorgt, dass die Hardware optimal ausgelastet wird. Tests zeigen, dass Millionen von schlafenden virtuellen Threads praktisch keine Probleme verursachen. Auch IO-intensive Anwendungen profitieren enorm: In einem Experiment mit 1 Million paralleler HTTP-Anfragen konnten die neuen virtuellen Threads die Last mit sehr wenig Overhead bewältigen, während 1 Million herkömmlicher Threads das System unbrauchbar gemacht hätten.

Ein Beispiel veranschaulicht, wie virtuelle Threads in Java 21 gehandhabt werden:

```
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 diesem Code wird ein ExecutorService erstellt, der für jede eingereichte Aufgabe einen neuen virtuellen Thread startet. Wir übergeben 100.000 Aufgaben, die jeweils eine Sekunde schlafen und anschließend eine Nachricht ausgeben. Mit traditionellen Threads wäre dies nicht praktikabel – allein das Starten von 100.000 Betriebssystem-Threads würde enorme Ressourcen erfordern. Mit virtuellen Threads hingegen ist dies problemlos möglich, da die JVM sie entsprechend der verfügbaren CPU-Kerne auf einige Dutzend echte Threads multiplexen kann.

Für Java-Entwickler bedeutet Loom eine erhebliche Reduzierung der Komplexität bei der parallelen Programmierung: Viele Probleme, die bisher mit asynchronen Callbacks oder reaktiven Streams gelöst wurden (um Threads zu sparen), lassen sich nun als einfache, sequenzielle Logik schreiben – und dennoch hochskalieren. Es ist jedoch wichtig zu beachten, dass virtuelle Threads keine Allzwecklösung sind: Sie beschleunigen keine CPU-intensiven Workloads – hier sind weiterhin mehr Kerne oder effizientere Parallelalgorithmen erforderlich. Für IO-lastige Anwendungen und Systeme mit hohen Nebenläufigkeitsanforderungen hingegen wird die Entwicklung deutlich einfacher und wartungsfreundlicher.

Java-Objekte und Speicherverbrauch

Ein wichtiger Aspekt der Java-Leistung ist der Speicherbedarf von Objekten und die Speicherorganisation (Heap vs. Stack).

Objekte und Heap: Java-Objekte haben einen Overhead (Klassenzugehörigkeit, Synchronisationsstatus usw.). Auf 64-Bit-JVMs beträgt der Header typischerweise 16 Byte für jedes Objekt. Es kann auch Auffüllungen geben, da Objekte aus Gründen der Ausrichtung normalerweise auf Vielfache von 8 Byte ausgerichtet sind. Mit Java 6 wurden komprimierte OOPs (Ordinary Object Pointers) eingeführt: Wenn der Heap kleiner als ~ 32 GB ist, kann die JVM 32-Bit-Zeiger anstelle von 64-Bit-Zeigern verwenden, was 4 Byte pro Objektfeld spart. Wird mehr Speicher benötigt, kann z.B. auf 16-Byte-Alignment (-XX:ObjectAlignmentInBytes) umgestellt werden, so dass auch Heaps bis zu 64 GB noch komprimierte Zeiger verwenden. In der Regel erfordert die Umstellung von 32-Bit- auf 64-Bit-Zeiger das 1,7-fache an Speicher.

String: Seit Java 9 verwendet die JVM intern kompakte Strings. Das heißt, wenn ein String nur aus reinen ASCII-Zeichen besteht, speichert sie die Zeichen als byte[] (8-Bit pro Zeichen) statt wie bisher als char[] (16-Bit pro Zeichen). Dies spart fast die Hälfte des Speicherplatzes für die vielen Strings, die typischerweise aus ASCII bestehen (z. B. JSON-Texte, Protokollnachrichten). In einzelnen Anwendungen wurde allein durch die Umstellung von Java 8 auf 9 ein Leistungsgewinn von ~40% beobachtet.

- **Metaspace:** Mit Java 8 wurde der „PermGen" durch Metaspace ersetzt, der im nativen Speicher (off-heap) verwaltet wird. Metaspace wächst dynamisch und beseitigt das frühere Problem eines festgelegten permanenten Heaps, das zuvor zu OutOfMemoryError führen konnte. Aus Leistungssicht brachte diese Änderung zwar keine direkte Beschleunigung, erhöhte jedoch die Stabilität und erleichterte das Tuning der JVM.
- **Heap vs. Off-Heap Memory:** Neben dem regulären Java-Heap (für Objekte) nutzt die JVM in verschiedenen Situationen auch nativen Speicher. Dazu gehören beispielsweise Metaspace (siehe oben), DirectByteBuffer-Allokationen für NIO, um Byte-Puffer außerhalb des

Heaps zu halten und teure Kopieroperationen in den Kernel-Space zu vermeiden, sowie C-Heaps für JNI-Bibliotheken. Aus Performance-Sicht kann der gezielte Einsatz von Off-Heap-Speicher vorteilhaft sein, da er die Garbage Collection entlastet. Allerdings geht dies auf Kosten einer komplexeren Speicherverwaltung, da die automatische Speicherbereinigung der JVM hier nicht greift.

- **Stack Allocation:** Mithilfe der Escape-Analyse (siehe oben) versucht die JVM, Objekte gar nicht erst auf dem Heap abzulegen, wenn dies nicht erforderlich ist. Dadurch entsteht praktisch eine Art von Wertetypen für kurzlebige Objekte: Diese existieren ausschließlich auf dem Stack und werden automatisch mit dem Methoden-Frame verworfen – völlig ohne Einfluss auf die Garbage Collection.

Zusammenfassend lässt sich sagen, dass im Speicherlayout viel optimiert wurde, um den Footprint von Java-Anwendungen zu reduzieren. Dabei geht es nicht nur um die Einsparung von Speicher: Weniger belegter Speicher und weniger Fragmentierung bedeuten auch weniger Arbeit für die GC und eine effizientere Nutzung der CPU-Caches. Kompaktere Objekte können beispielsweise die Cache-Lokalität verbessern und dadurch insbesondere bei großen Datenstrukturen spürbare Leistungssteigerungen erzielen.

Entwickler sollten dennoch bewusste Entscheidungen treffen: Primitive Arrays vs. Objektlisten, sparsame Nutzung von Strings sowie ein Verständnis der Kosten von Autoboxing. Die JVM optimiert vieles automatisch, aber ein grundlegendes Wissen über Speicherverwaltung hilft, Performance-Engpässe zu vermeiden.

JVM-Implementierungen: HotSpot, OpenJ9, Azul Prime, GraalVM

HotSpot (OpenJDK/Oracle JDK) ist die am weitesten verbreitete JVM, aber es gibt leistungsfähige Alternativen mit spezifischen Vorteilen:

- **Eclipse OpenJ9, IBMs Open-Source-JVM**, ist auf schnellen Start und geringen Speicherverbrauch optimiert. Dank IBMs Testarossa JIT und einem eigenen Garbage Collector eignet sie sich besonders für Cloud-Umgebungen mit begrenztem RAM. Allerdings liegt OpenJ9 beim maximalen Durchsatz leicht hinter HotSpot zurück.

- **Azul Platform Prime** (ehemals Zing) wurde für Latenz-kritische Anwendungen entwickelt. Sein Pauseless C4-GC minimiert Stop-the-World-Pausen selbst bei großen Heaps, während der LLVM-basierte Falcon JIT oft effizienteren Code generiert als HotSpot C2. Azul setzt diese Technologien seit über einem Jahrzehnt in Produktionsumgebungen ein und ist eine bewährte Lösung für Systeme mit hohen Latenzanforderungen.
- **GraalVM** ist eine erweiterte JDK-Distribution von Oracle Labs mit Polyglot-Unterstützung für JavaScript, Python, R, Ruby und WebAssembly. Die Native Image-Funktion ermöglicht eine Ahead-of-Time-Kompilierung (AOT), wodurch Java-Anwendungen sofort starten und weniger RAM verbrauchen. Während dies ideal für serverlose und Function-as-a-Service (FaaS)-Workloads ist, liefert eine gut optimierte HotSpot-VM unter konstanter Last oft eine bessere Latenz.

Diese Alternativen zeigen, dass Java nicht länger ein monolithisches System ist, sondern ein Ökosystem verschiedener VM-Technologien. GraalVM Native Image hat in der Cloud-Ära an Zugkraft gewonnen und macht Java für schnell startende CLIs und FaaS attraktiver. Azul Prime bleibt für Hochfrequenz-Finanzsysteme relevant, während OpenJ9 eine starke Option für speicherempfindliche Arbeitslasten ist. Die Vielfalt der JVMs fördert die Innovation und treibt die Leistungsverbesserung kontinuierlich voran.

Meilensteine der Java-Versionen (Performance-relevant)

Abschließend in diesem Gegenwarts-Block ein kompakter Überblick einiger wichtiger Versionen und deren Performance-Neuerungen

Version	Jahr	Wichtigste Leistungsmerkmale
Java 5 (Tiger)	2004	java.util.concurrent (Thread-Pools, Futures), 64-Bit JVM, verbesserte JIT, CMS GC (GC mit niedriger Pause als Option).
Java 6 (Mustang)	2006	Schnellere JIT-Kompilierung, Escape-Analyse (experimentell), komprimierte OOPs, parallele alte GC, Verbesserungen bei der Synchronisierung.

Java 7 (Dolphin)	2011	Fork/Join-Framework, G1 GC (experimentell, offiziell ab 7u4), invokedynamic, NIO 2 (asynchrone Kanäle), Tiered Compilation (C1+C2).
Java 8 (Spider)	2014	Lambdas & Streams (einfachere Parallelisierung), Metaspace anstelle von PermGen, Compact Strings, Tiered Compilation standardmäßig aktiviert.
Java 9 (Jigsaw)	2017	G1 GC als Standard, Flight Recorder (OracleJDK kommerziell, ab OpenJDK 11 kostenlos), AOT-Kompilierung (experimentell), Spin-Wait-Hinweis, 8-Byte-aligned Heap.
Java 11 (LTS)	2018	ZGC (experimentell), Epsilon GC (No-Op), Flight Recorder + Mission Control in OpenJDK, neuer HTTP-Client (bessere Leistung als HttpUrlConnection).
Java 15	2020	Shenandoah GC in OpenJDK integriert (JEP 379); ZGC jetzt produktionsreif (kein Experiment-Flag mehr, JEP 377); Textblöcke (Syntax, ohne Leistungseinfluss); versteckte Klassen (für Frameworks wie Bytecode-Generatoren, etwas effizienter).
Java 17 (LTS)	2021	Stärkere Kapselung im JDK (Sicherheit vs. Leistung), neue Plattform-Intrinsics (z.B. CRC32, GHASH), Shenandoah im Oracle JDK, konsolidierte Updates von 11-16.
Java 19	2022	Virtuelle Threads (Vorschau) für hohe IO-Parallelität, strukturierte Concurrency (Inkubator), Foreign Function & Memory API (Vorschau) für schnelleren Zugriff auf Native/Off-Heap.
Java 21 (LTS)	2023	Virtual Threads GA (JEP 444), Generational ZGC (Vorschau), insgesamt großer Schritt durch Loom und verbesserte GC.
Java 22	2024	Region Pinning für G1 (JEP 423): Stabilisiert/ verbessert GC durch Fixierung bestimmter Heap-Regionen. Foreign Function & Memory API (JEP 454): Effizientere native Interop, geringerer Overhead. Vektor-API (JEP 460): Nutzt SIMD-Befehle für datenparallele Beschleunigungen. Stream Gatherer (JEP 461): Reduziert den I/O-Overhead durch gebündelte Datenerfassung.
Java 23	2024	Vector API (JEP 469, 8. Inkubator): Beschleunigt datenparallele Aufgaben über SIMD Stream Gatherer (JEP 473, Zweite Vorschau): Verbessert den Durchsatz durch Stapelverarbeitung mehrerer I/O-Operationen. ZGC: Standardmäßiger Generierungsmodus (JEP 474): Effizientere GC, bessere Handhabung von kurzlebigen Objekten.

Java 24	2025	Es beschleunigt mit generational Shenandoah und Compact Object Headers (beide experimentell), zusammen mit Verbesserungen an G1, die den GC-Overhead reduzieren. Es bietet außerdem schnelleres AOT-Laden, verbessertes Streaming und eine erweiterte Vektor-API für SIMD (9. Inkubator), neben anderen Verbesserungen.

Tabelle: Auswahl von Java-Versionen und ihr Einfluss auf Leistung und Optimierung.

Ihr seht, dass die Leistung von Java in fast jeder Version verbessert wurde, entweder direkt (schnellere JVM) oder indirekt durch neue Sprach-/Bibliotheksfunktionen, die effizienteren Code ermöglichen. Insbesondere die LTS-Versionen (8, 11, 17, 21) enthielten oft die experimentellen Funktionen der Zwischenversionen in stabiler Form. Es lohnt sich immer, die neueste Java-Version zu verwenden, um von allen Optimierungen zu profitieren - vor allem die Leistungssteigerung wirkt sich direkt aus, ohne dass neuer Byte-Code kompiliert werden muss.

Die Zukunft: Wie geht es mit Java weiter?

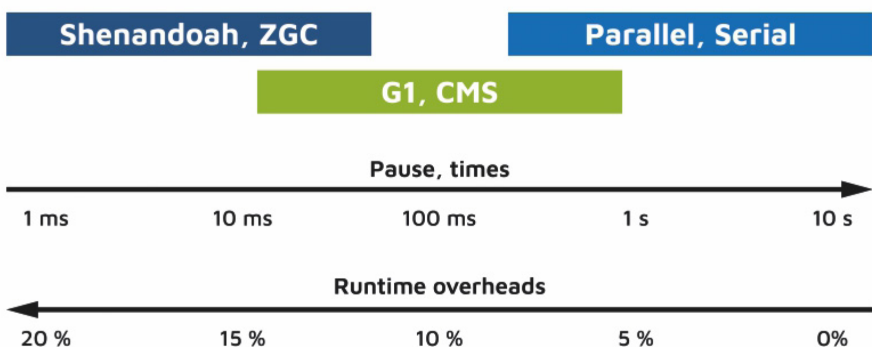
Die Java-Plattform entwickelt sich weiter, wobei mehrere Schlüsselprojekte ihre zukünftige Leistung und Skalierbarkeit bestimmen:

- **Project Loom & Virtual Threads:** Loom ist jetzt Teil von Java 21 und es wird einige Zeit dauern, bis es vollständig übernommen wird, da Frameworks wie Tomcat, Jetty, Spring und Quarkus es integrieren. Virtuelle Threads könnten schließlich reaktive Programmiermodelle übertreffen, indem sie die Concurrency vereinfachen und eine effiziente Skalierung von Thread-per-Request-Architekturen ermöglichen. Structured Concurrency wird Lesbarkeit von nebenläufigem Code weiter verbessern. Während sie für I/O-gebundene Aufgaben ideal ist, profitieren CPU-lastige parallele Streams weniger, da sie durch das Betriebssystem begrenzt bleiben.
- **CRaC (Coordinated Restore at Checkpoint):** Das CRaC-Projekt von OpenJDK zielt darauf ab, die Startzeit drastisch zu reduzieren, indem ein Snapshot eines „aufgewärmten" JVM-Zustands erstellt wird, der innerhalb von Millisekunden wiederhergestellt werden kann. Dies ist von großer Bedeutung für Cloud-Umgebungen, in denen eine schnelle Skalierung erforderlich ist. Anwendungen müssen sich anpassen und die Ressourcen vor einem Snapshot bereinigen.

Derzeit ist CRaC nur für Linux verfügbar und befindet sich noch in der Entwicklung, aber es stellt eine Alternative zu GraalVM Native Image dar, indem es Kaltstarts beschleunigt, ohne JIT-Optimierungen zu verlieren.

- **Native Kompilierung & Project Leyden:** Inspiriert von GraalVM Native Image versucht Project Leyden, die AOT-Kompilierung (AOT = ahead of time) für Java zu standardisieren. Ziel ist es, langsame Startzeiten, insbesondere für Cloud-basierte Anwendungen, zu verringern und gleichzeitig aktuelle Einschränkungen wie eingeschränkte Reflektion und dynamisches Laden zu beheben. Künftige Java-Versionen könnten getrennte JIT- und AOT-Modi einführen, so dass Java-Binärdateien in Umgebungen, in denen ein schneller Start wichtig ist, häufiger verwendet werden.
- **Project Valhalla (Value Types):** Zukünftige Java-Versionen werden wahrscheinlich Wertetypen einführen, die es ermöglichen, Objekte flach in Arrays oder Containern zu speichern, was die Cache-Effizienz verbessert und den Zeiger-Overhead reduziert. Obwohl sich Valhalla noch in der Entwicklung befindet, wird es den Speicherbedarf und die Leistung von Java weiter optimieren.

Die Zukunft von Java konzentriert sich auf schnellere Starts, bessere Concurrency und geringeren Speicher-Overhead. Virtuelle Threads werden die Art und Weise, wie Java mit Skalierbarkeit umgeht, neu gestalten, CRaC befasst sich mit Startverzögerungen, und Valhalla modernisiert die Objektbehandlung. Die letzten beiden sind noch Zukunftsmusik, aber diese Innovationen würden Java flexibler machen und eine Feinabstimmung der Anwendungen auf die Einsatzanforderungen ermöglichen – sei es durch native Binärdateien oder neue Concurrency-modelle.



GC-Selection: Latency vs. Throughput

Praktische Performance-Tipps für Hitchhiker's

Um die Leistung von Java-Anwendungen effektiv zu optimieren, solltet ihr die folgenden Schlüsselstrategien berücksichtigen:

- **Auswahl des richtigen Garbage Collectors:** Die Wahl des GC wirkt sich auf Latenz und Durchsatz aus. G1 ist seit Java 11 der Standard für die meisten Serveranwendungen. Für extrem niedrige Pausenzeiten (Echtzeit, UI) ist ZGC (stabil seit Java 17) oder Shenandoah (bevorzugt in Red Hat/SAP JDKs) ideal. Batch-Workloads profitieren von Parallel GC, während kleine Tools mit kurzen Laufzeiten am besten mit Serial GC arbeiten, um Parallelisierungs-Overhead zu vermeiden.
- **Profiling mit den richtigen Tools:** Identifiziert Leistungsengpässe, bevor Ihr optimiert. Java Flight Recorder (JFR) und Async Profiler bieten tiefe Einblicke. Eine frühzeitige Profilerstellung hilft bei der Erstellung von Baselines und zeigt die größten CPU-Verbraucher und Zuweisungs-Hotspots auf. Für tiefergehende Analysen (z. B. Lock Contention, native Engpässe) solltet ihr spezialisierte Profiler oder Tools verwenden. Ihr beginnt mit einem breiten Spektrum (hochauflösendes CPU-Profiling) und verfeinern den Fokus dann schrittweise. Vermeidet Vermutungen - Engpässe liegen oft dort, wo man sie am wenigsten erwartet (z. B. I/O statt CPU, übermäßige GC statt langsamer Schleifen). Am schnellsten lassen sich Engpässe mit JProfiler [10] finden - seit Jahrzehnten mein treuer Reisebegleiter.
- **Effiziente Datenstrukturen und Speicherverwaltung:** Wählt die richtigen Strukturen, um den Overhead zu reduzieren. Verwendet Primitive anstelle von Wrapper-Typen (z. B. int statt Integer), um unnötiges Boxing zu vermeiden. Zieht für große Collections spezialisierte Bibliotheken wie Trove oder FastUtil für primitiv gestützte Listen in Betracht. Vermeidet übermäßige String-Verkettung in Schleifen - verwendet stattdessen StringBuilder. Minimiert unnötige Objektzuweisungen, aber vermeidet veraltete Objektpools, da moderne JVMs kurzlebige Objekte effizient handhaben.
- **Concurrency und Parallelität:** Verwaltet Threads effizient. Bei CPU-gebundenen Aufgaben entspricht die optimale Thread-Anzahl in etwa der Anzahl der CPU-Kerne. Bei I/O-gebundenen Arbeitslasten ermöglichen virtuelle Threads (Java 21) eine Skalierung auf Tausende

von Aufgaben ohne übermäßigen Overhead. Minimiert die Synchronisierung (Sperren) und bevorzugt sperrenfreie Alternativen wie Atomics, StampedLock oder LMAX Disruptor. Verwendet bei der Arbeit mit Sammlungen im Multithreading konkurrierende Datenstrukturen oder unveränderliche Snapshots anstelle von globalen Sperren. Messt immer die Skalierbarkeit - das Hinzufügen von Threads führt meist nicht zu einer linearen Leistungssteigerung. Verwendet JMH-Benchmarks [11], um den optimalen Grad der Parallelität zu ermitteln.

- **Bleibt mit Euren JDK-Versionen auf dem neuesten Stand:** Neue JDK-Versionen bieten erhebliche Leistungsverbesserungen. Ein Upgrade von Java 8 auf Java 11 kann die GC-Zeit aufgrund von G1-Optimierungen um 20 % reduzieren, während Java 17 die String-Verarbeitung und die GC-Effizienz weiter verbessert. Benchmarks zeigen, dass Java 17 den Durchsatz im Vergleich zu Java 11 um ~15 % steigert, und das, bei geringerer Latenz, selbst ohne Codeänderungen. Regelmäßige Updates sorgen für kostenlose Leistungssteigerungen ohne zusätzlichen Aufwand.

Keep It Simple - Vermeidet verfrühte Optimierungen – konzentriert euch zuerst auf Architektur und Algorithmen. Die JVM ist hochgradig optimiert, aber wenn eine Feinabstimmung erforderlich ist, solltet ihr Profiling-Tools und moderne Funktionen nutzen. Mit dem richtigen Ansatz kann Java eine außergewöhnliche Leistung erzielen.

Fazit

Java hat in den letzten 30 Jahren einen langen Weg zurückgelegt. Dank hoch entwickelter JIT-Compiler, moderner Garbage-Collectors und innovativer Concurrency-Konzepte ist die Plattform, die einst den Ruf hatte, langsam zu sein, heute eine der leistungsfähigsten Umgebungen für skalierbare Serveranwendungen. Die kontinuierlichen Verbesserungen - ob in der HotSpot-Engine, in alternativen JVMs wie GraalVM oder in zukunftsweisenden Projekten wie Loom und CRaC - zeigen, dass Performance in Java ein lebendiges und dynamisches Thema bleibt.

Entwickler sollten die vielen Tools zur Analyse und Optimierung (JFR, JMC, Async Profiler, JMH) konsequent nutzen und sich sowohl an

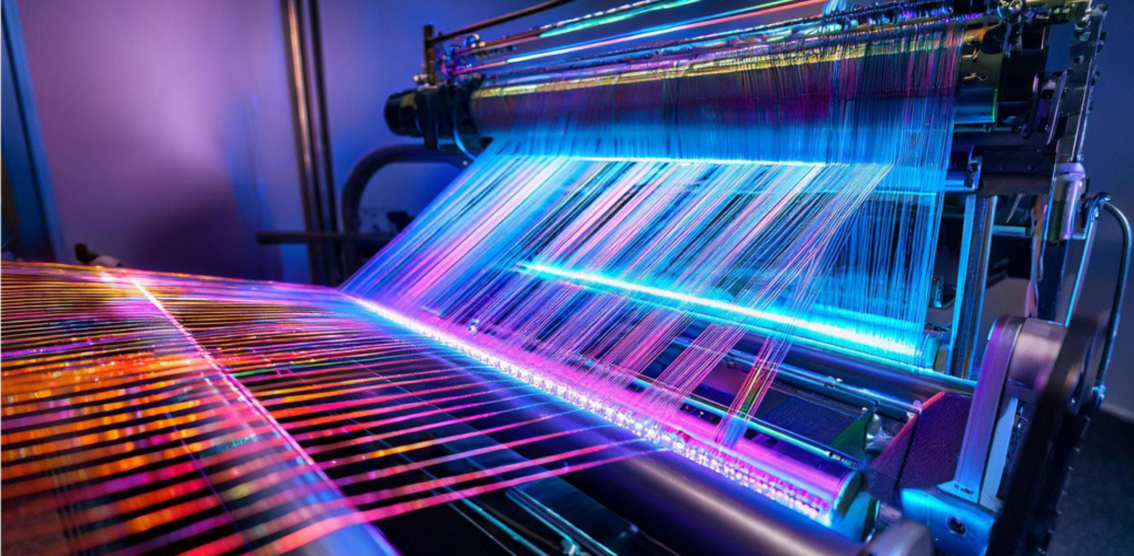
bewährten als auch an neuen Technologien orientieren. So könnt ihr die bestmögliche Performance aus euren Anwendungen herausholen - egal, ob es sich um rechenintensive Prozesse, latenzkritische Systeme oder moderne Cloud-Umgebungen handelt.

Die Zukunft der Java-Plattform verspricht, durch native Kompilierung, optimierte Concurrency und energieeffiziente Ansätze noch leistungsfähiger zu werden. Wer heute auf die richtigen Optimierungstechniken setzt, profitiert nicht nur von höherer Geschwindigkeit, sondern auch von besserer Skalierbarkeit und Wartbarkeit der Software. Java wird also auch in Zukunft eine zentrale Technologie bleiben - bereit, die Herausforderungen moderner Anwendungen zu meistern.

Literatur

[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-shenandoah.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=93b48eef8acfd110755b44ff1a346b65422a2bf4#
[09] https://youtu.be/HCcq6VLuXe0
[10] https://www.ej-technologies.com/jprofiler
[11] https://github.com/openjdk/jmh

> Zurück zum Inhaltsverzeichnis



#JAVAPRO #PERFORMANCE

Von Reactive Streams zu Virtual Threads

Autor:

Adam Warski ist einer der Mitbegründer von SoftwareMill, wo er hauptsächlich mit Java, Scala und anderen spannenden Technologien entwickelt. Er engagiert sich aktiv in Open-Source-Projekten wie Ox, Tapir, sttp, Quicklens, ElasticMQ und weiteren. Zudem war er als Sprecher auf führenden Konferenzen wie JavaOne, Devoxx, GeeCON und ScalaDays vertreten.

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

Virtual Threads bieten eine schnelle und ressourcenschonende Threading-Lösung für die JVM – sowohl im Hinblick auf Speicherverbrauch als auch auf Umschaltgeschwindigkeit. Sie versprechen die Rückkehr zu einem direkten, synchronen Programmiermodell. Doch reicht das aus, um den Status quo im Bereich Daten-Streaming infrage zu stellen? Können wir das Beste aus beiden Welten haben: die Einfachheit, die Virtual Threads versprechen, und gleichzeitig die Robustheit und Sicherheit von Reactive Streams? Finden wir es heraus!

Virtual Threads: Der bisherige Weg und ein Blick in die Zukunft

Zunächst ein kurzer Rückblick: Virtual Threads wurden im September 2023 mit Java 21 veröffentlicht – nach sechs Jahren Entwicklungszeit. Einige verbleibende Einschränkungen dieser Implementierung werden mit dem Release von Java 24 im März 2025 vollständig aufgehoben.

Virtual Threads sind Teil einer größeren Initiative namens Project Loom. Diese Initiative umfasst auch verwandte Funktionen, insbesondere die APIs für Structured Concurrency und Scoped Values, die sich derzeit in der Vorschauphase befinden.

Virtual Threads sind in erster Linie ein Feature der JVM (Java Virtual Machine). Dennoch gibt es auch einige Änderungen an der Standardbibliothek. Dadurch können alle JVM-basierten Sprachen dieses neue, leichtgewichtige Nebenläufigkeitsmodell nutzen – angefangen bei Java, aber auch Kotlin, Scala, Clojure und andere.

Warum wurden Virtual Threads überhaupt eingeführt?

Um die Beziehung zwischen Virtual Threads und Reactive Streams besser zu verstehen, lohnt sich ein Blick auf die Motivation hinter ihrer Einführung.

Früher folgte Java einem synchronen Programmiermodell. Zum Beispiel erhielt bei der Implementierung eines HTTP-Servers jede eingehende Anfrage ihren eigenen Thread, der die Daten las, verarbeitete und anschließend die Antwort zurückschickte.

In älteren Java-Versionen entsprach jedes Objekt vom Typ java.lang.Thread einem Betriebssystem-Thread. Diese sogenannten Plattform-Threads (in der Terminologie der Virtual Threads) sind jedoch schwergewichtige Ressourcen. Zum einen benötigen sie eine nicht unerhebliche Menge an Speicher, zum anderen ist sowohl das Erzeugen eines Threads als auch das Umschalten des CPU-Kontexts zeitintensiv. Daher konnten nur eine begrenzte Anzahl solcher Threads erstellt und genutzt werden – meist nur einige Tausend.

Mit dem Aufstieg des Webs, zunehmendem Datenverkehr und riesigen Datenmengen stieß dieses Modell schnell an seine Skalierungsgrenzen. Threads wurden zu einer knappen und teuren Ressource – und es ergab schlicht keinen Sinn mehr, sie untätig auf Antworten von Datenbanken oder Webdiensten warten zu lassen.

Aus diesem Grund entstanden Thread-Pools, ExecutionContexts und ähnliche Mechanismen, um Threads effizienter zu nutzen. Statt in direktem Stil zu programmieren, wechselte man zu verschiedenen Varianten von Futures. Zum Beispiel könnte eine blockierende, synchrone Implementierung einer Auto-Verkauf-Logik folgendermaßen aussehen:

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

Die gleiche Logik, jedoch effizienter mit Threads durch den Einsatz von CompletableFutures, sieht deutlich anders aus:

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

Ganz offensichtlich haben technische Überlegungen – insbesondere die effiziente Nutzung von Threads – Vorrang vor der Lesbarkeit des Codes erhalten. Project Loom verfolgt das Ziel, die durch den Einsatz von Futures erzielte bessere Thread-Ausnutzung beizubehalten und gleichzeitig die Lesbarkeit und Einfachheit eines direkten Programmierstils zurückzubringen.

Konkret adressiert Project Loom drei Probleme, die mit dem Übergang zur asynchronen Programmierung entstanden sind:

- Syntax: Java soll wieder bevorzugt im direkt lesbaren, synchronen Stil genutzt werden – inklusive der eingebauten Kontrollflusskonstrukte statt zusätzlicher Bibliotheken.
- Viralität: Wird eine Methode aufgerufen, die ein Future zurückliefert, muss oft auch die eigene Methode ein Future zurückgeben (denn blockierendes Warten würde den Vorteil des Futures zunichtemachen); dieses Problem ist auch als Function Coloring bekannt.
- Verlorener Kontext: Bei Fehlern im Zusammenspiel mit Futures enthält der Stacktrace häufig nur den letzten Teil der Kette, was das Debugging erschwert oder gar unmöglich macht.

Und genau diese Probleme konnte Project Loom erfolgreich lösen!

Futures und Virtual Threads: ein Blick unter die Haube

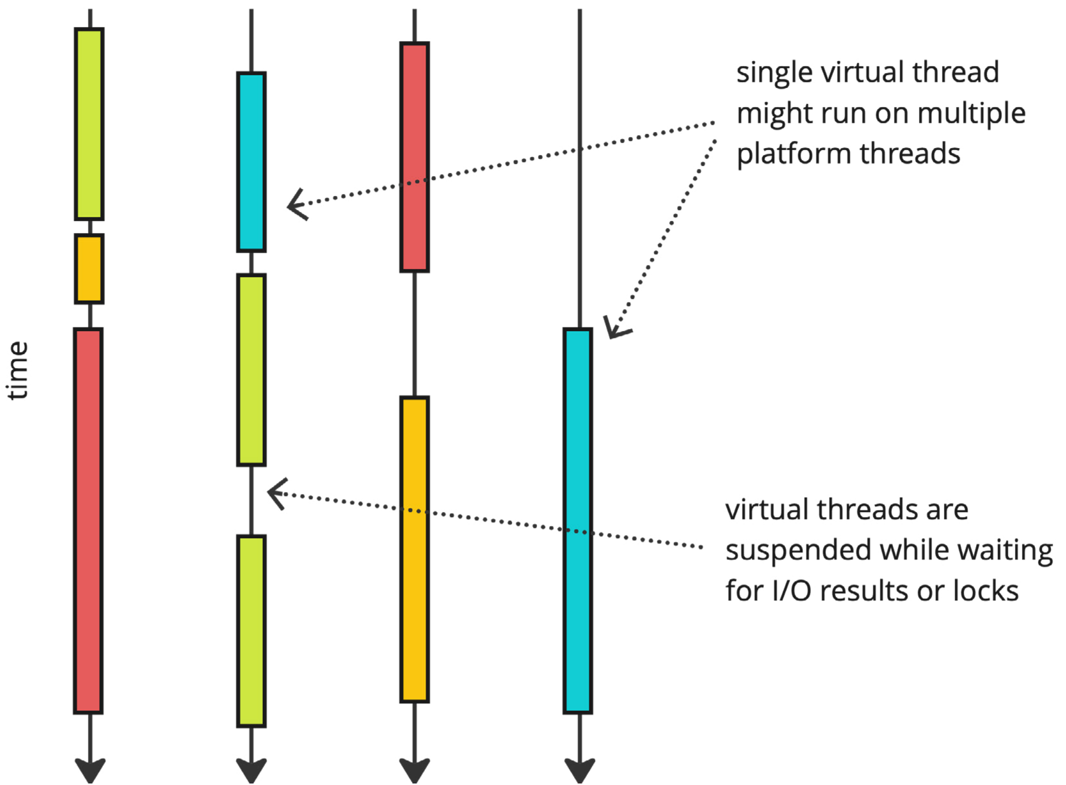
Es lohnt sich auch, einen genaueren Blick darauf zu werfen, wie sowohl Futures als auch Virtual Threads eine bessere Ausnutzung von Threads ermöglichen.

Hinter jedem Executor verbirgt sich ein Thread-Pool, dessen Threads vorab erstellt werden. Diese Threads holen sich Aufgaben aus einer Warteschlange (Task Queue) und führen sie aus. Wenn Code an einen Executor übergeben wird, erhält man ein Future-Objekt zurück – dieses wird abgeschlossen, sobald die übergebene Aufgabe ausgeführt wurde.

Der nächste freie Thread übernimmt also eine Aufgabe aus der Warteschlange. Beim Ausführen dieses Codes können weitere Aufgaben entstehen, neue Futures erzeugt und miteinander verkettet werden.

Genau so funktionieren Virtual Threads – allerdings mit einem wichtigen Unterschied: Während klassische Executor-Services auf Bibliotheksebene implementiert sind, wird das Scheduling der Virtual Threads auf VM-Ebene realisiert – durch Project Loom.

Die JVM verwaltet also im Hintergrund automatisch einen Pool von Plattform-Threads, die die Aufgaben ausführen. Gleichzeitig bleibt die Thread-API erhalten: Eine Instanz kann entweder einen Plattform-Thread oder einen Virtual Thread darstellen.



Allerdings ist der Scheduler, der Virtual Threads auf Plattform-Threads abbildet, nicht der einzige Beitrag von Project Loom zur JVM. Ein weiterer entscheidender Bestandteil ist die nachträgliche Anpassung aller blockierenden APIs, damit sie Virtual-Thread-fähig sind.

Immer wenn Sie eine blockierende Methode aufrufen – etwa beim Erwerb eines Semaphors oder Locks, beim Senden von Daten über einen Socket oder beim Lesen eines InputStreams – wird nicht der zugrunde liegende Plattform-Thread blockiert.

Stattdessen wird nur der Virtual Thread blockiert und solange „beiseitegelegt", bis z. B. ein Permit oder Lock verfügbar ist oder das Ergebnis der Operation bereitsteht. Der Plattform-Thread kann währenddessen andere Virtual Threads weiter ausführen, die bereit zur Ausführung sind.

Einige Ausnahmen von dieser Anpassung (insbesondere synchronized-Methoden) werden mit dem Release von Java 24 aufgehoben.

Wie kamen wir zu Reactive Streams?

Wenden wir uns nun den Reactive Streams zu. Die Initiative zur Entwicklung eines Standards für asynchrone, nicht-blockierende Stream-Verarbeitung begann Ende 2013. Die finale Version wurde 2015 veröffentlicht und 2017 mit Java 9 offiziell in die Sprache aufgenommen.

Reactive Streams verfolgen das Ziel, den Austausch von Datenströmen über asynchrone Grenzen hinweg zu regeln. Dieser Austausch – oder vielmehr eine Reihe von Exchanges – soll dabei stets innerhalb begrenzter Speicherressourcen erfolgen.

Die JVM-Interfaces, die Teil des Reactive-Streams-Standards sind (Publisher, Subscriber und Subscription), befinden sich auf sehr niedriger Abstraktionsebene und sind nicht für die direkte Nutzung durch Endanwender vorgesehen. Stattdessen sollten Bibliotheken verwendet werden, die den Standard implementieren.

Solche Bibliotheken bieten hochgradig abstrahierte Schnittstellen zur Stream-Verarbeitung sowie Werkzeuge für deklaratives Concurrency-Management, die Anbindung an I/O-Operationen und eine sichere Fehlerbehandlung.

Beispiele für Reactive-Streams-Bibliotheken sind Akka Streams, Vert.x, Helidon, RxJava, Reactor (used in Spring) und andere.

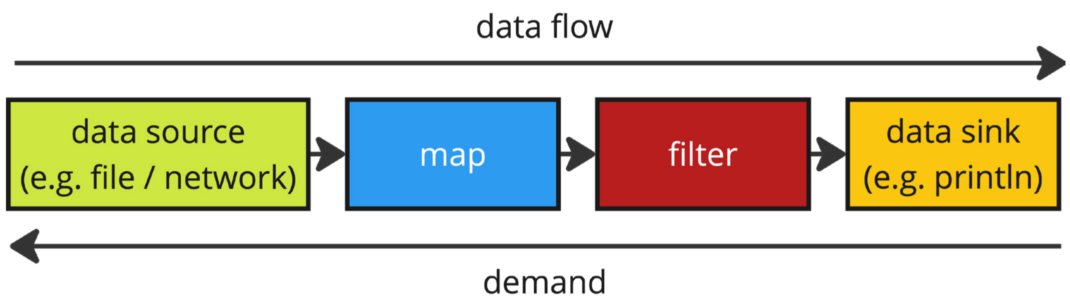
Welche Probleme lösen Reactive Streams?

Wie bereits erwähnt, ermöglichen Reactive Streams die Verarbeitung großer Datenmengen im Streaming-Modus. Bibliotheken, die den Standard umsetzen, basieren häufig auf Futures oder ähnlichen Konzepten und nutzen Thread-Pooling sowie Betriebssystemressourcen besonders effizient – wie zuvor beschrieben.

Grundsätzlich gibt es zwei Problemfelder, die als besonders komplex gelten – und genau hier bieten Reactive Streams robuste und gleichzeitig lesbare Lösungen:

1. Nebenläufigkeit:
   Es gilt als allgemein bekannt, dass das Schreiben nebenläufiger Programme mit Locks schwierig und fehleranfällig ist. Der deklarative Ansatz von Reactive Streams hilft dabei, da Entwickler*innen lediglich beschreiben müssen, was passieren soll – wie es geschieht, übernimmt die Runtime. Die Details der Ausführung (einschließlich Thread-Management) werden vollständig abstrahiert.

2. Fehlerbehandlung:
   Fehler können an den unerwartetsten Stellen auftreten – und eine korrekte Ressourcenverwaltung im Ausnahmefall ist oft das Herzstück komplexer Systeme (oder die Quelle zahlreicher Bugs). Reactive Streams bieten gezielte Mechanismen zur Beschreibung des Fehlerverhaltens, zur sicheren Verwaltung von Ressourcenlebenszyklen und – wo möglich – zur Wiederherstellung nach einem Fehler.



Ein weiteres zentrales Problem, das Reactive Streams adressieren, ist die Verarbeitung von Daten innerhalb begrenzter Speicherressourcen. Dies wird durch das Konzept des Backpressure umgesetzt. Eine Upstream-Komponente darf Daten nur dann zur Weiterverarbeitung senden, wenn sie von der Downstream-Komponente eine entsprechende Anforderung (Demand) erhält. Dieser bidirektionale Datenfluss – Daten von Upstream nach Downstream und Demand von Downstream nach Upstream – stellt sicher, dass stets nur eine begrenzte Anzahl an Elementen gleichzeitig verarbeitet wird.

Einfache Streams implementieren

Trotz ihrer leistungsstarken Streaming-Fähigkeiten bringen Reactive-Streams-Implementierungen auch die zuvor genannten Herausforderungen mit sich: die Viralität von Futures, der verlorene Kontext bei Fehlern und ein hoher Syntax-Overhead durch die Komposition asynchroner Operationen statt einfacher Anweisungen.

Doch können wir die Vorteile von Project Loom nutzen, ohne die Stärken von Reactive Streams aufzugeben? Die Antwort lautet: Ja, das ist möglich! Genau das war unser Ziel bei der Entwicklung einer passenden Implementierung – inspiriert von Go und Kotlin.

Der Quellcode befindet sich im Projekt Jox, das unter der Apache2 Open-Source-Lizenz veröffentlicht wurde. Im Folgenden geben wir einen kurzen Überblick über die Architektur. Wer jedoch herausfinden möchte, wie sich "reaktives" Streaming im Direct-Style anfühlt, sollte die Bibliothek unbedingt selbst ausprobieren!

Ziel war es, eine Datenverarbeitungsbibliothek im Geiste von Project Loom zu entwickeln, die möglichst viele Java-eigene Konstrukte verwendet – etwa if, for, while sowie try-catch-finally zur Fehlerbehandlung.

Auch die Beschreibung der Logik innerhalb der einzelnen Verarbeitungsstufen soll in direkter Syntax erfolgen. Die Operationen werden einfach mit ; verknüpft, ohne dass sie in zusätzliche Container eingeschlossen werden müssen.

Eine einzelne Verarbeitungsstufe innerhalb einer Datenpipeline wird durch die Klasse FlowStage dargestellt. Eine solche Stage kann mit einem Ziel (Sink), an das Daten übergeben werden, ausgeführt werden:

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

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

Zusätzlich führen wir eine Flow-Wrapper-Klasse ein, die die zuletzt definierte Verarbeitungsstufe sowie eine Reihe von Hilfsmethoden enthält, mit denen wir unsere Datenverarbeitungspipeline aufbauen

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

Es zeigt sich, dass genau diese einfachen Bausteine ausreichen, um selbst komplexe Datenflüsse zu beschreiben! Hier zum Beispiel eine Factory-Methode, die einen unendlichen Werte-Stream erzeugt:

```
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);
      }
    });
  }
}
```

Die durch diese Methode erzeugte Flow-Instanz beschreibt lediglich, was geschehen soll – beim Aufruf wird noch keine Verarbeitung ausgeführt. Um die Verarbeitung zu starten, muss eine FlowEmit-Instanz an die letzte Stage übergeben werden – je nach Anwendungsfall und gewünschter Art der Konsumierung.

Bevor wir zur konkreten Nutzung kommen, fällt bereits auf, dass wir einige zentrale Prinzipien von Project Loom umsetzen: wir verwenden Java's eingebaute Kontrollflussstruktur while sowie eine direkte, lesbare Syntax zur Beschreibung der Logik.

Sobald dies steht, kann eine Methode wie die folgende verwendet werden, um den Stream auszuführen und alle verarbeiteten Elemente in einer Liste zu sammeln:

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

Sieht ziemlich einfach aus, oder? Natürlich benötigen wir auch Methoden zur Transformation von Streams, eine der beliebtesten ist map:

```
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)));
    });
  }
}
```

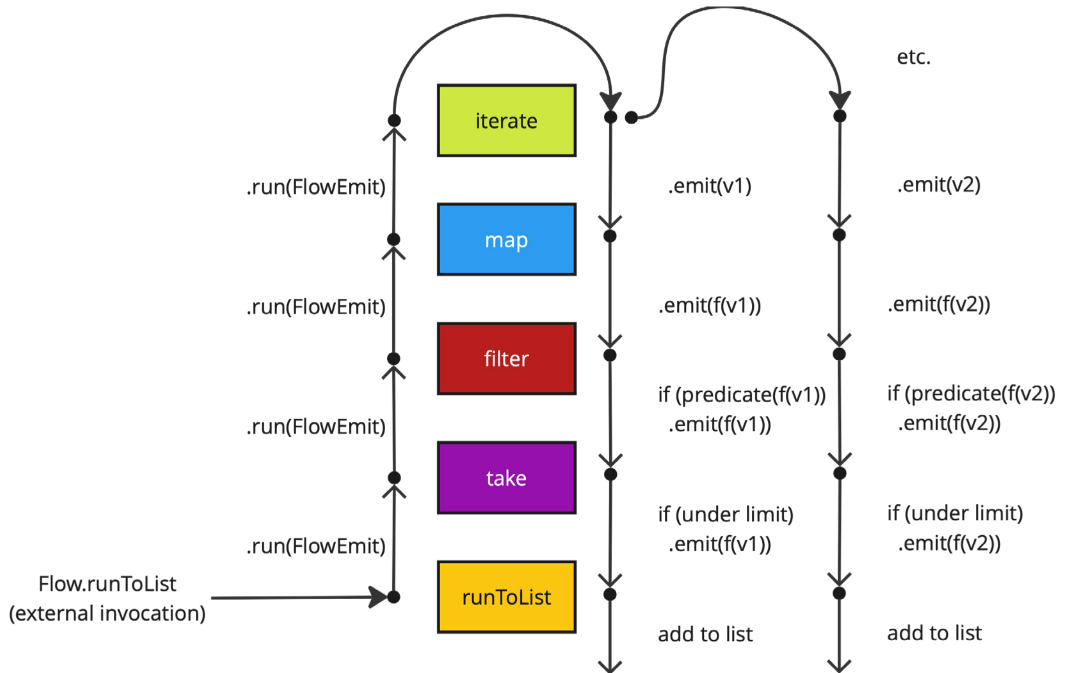
Hier wird einfach das Ergebnis der Mapping-Funktion ausgegeben – basierend auf den Werten, die von der vorherigen (nun „letzten“) Verarbeitungsstufe geliefert wurden. Auf ähnliche Weise lassen sich auch Methoden wie filter, take, mapStateful, sliding, runFold, runDrain usw. implementieren. Mit diesen Bausteinen können wir nun synchrone, single-threaded Datenverarbeitungspipelines aufbauen:

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

Wie funktioniert das? Wie bereits erwähnt, erstellen Aufrufe von Methoden wie iterate, take und map lediglich eine Beschreibung des Datenflusses. Flow ist eine lazy-evaluierte Beschreibung, wie ein Stream verarbeitet werden soll.

Jede neu erzeugte Flow-Instanz enthält eine Referenz auf eine FlowStage, die wiederum auf die vorherige Stufe verweist – so entsteht eine Kette.

Erst beim Aufruf von .runToList() wird die tatsächliche Verarbeitung gestartet. Diese Methode erstellt ein Ziel (Sink) – eine FlowEmit-Instanz – und übergibt sie an die vorherige Stufe. In diesem Fall fügt das Sink die Elemente einfach einer Liste hinzu. Die vorherige Stufe übergibt wiederum ein angepasstes Sink an ihre eigene run-Methode – und so weiter, bis die gesamte Pipeline durchlaufen ist.



Letztlich ruft die produzierende Stufe FlowEmit.emit auf. Jeder einzelne emit-Aufruf schleust die Daten durch die gesamte Pipeline. Das bedeutet: Ein emit-Aufruf wird erst dann abgeschlossen, wenn das jeweilige Element vollständig verarbeitet (oder verworfen) wurde. Dadurch ist garantiert, dass immer nur ein Element gleichzeitig verarbeitet wird – zumindest vorerst – und die Anforderung der Speicherbegrenzung (memory-boundedness) trivial erfüllt ist.

Wir verfügen damit über ein elegantes, streambasiertes API – doch bis zu diesem Punkt hätten wir ein ähnliches Ergebnis auch mit den Stream-Gatherern von Java erzielen können:

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

Spannend wird es aber erst dann, wenn wir Nebenläufigkeit, mehrere parallele Flows oder I/O-Verarbeitung ins Spiel bringen.

Asynchrone Streams implementieren

Wie bereits erwähnt, ist Nebenläufigkeit von Natur aus komplex. Der Einsatz von Locks führt nur allzu leicht zu Deadlocks, und Versuche, Abläufe parallel auszuführen, enden oft in schwer reproduzierbaren Race Conditions. Aus diesem Grund ist die beste Art der Nebenläufigkeit die, bei der sich jemand anderes darum kümmert und dann müssen Sie sich nicht mehr damit beschäftigen.

Reactive Streams haben sich als äußerst nützlich erwiesen, wenn es darum geht, explizite Nebenläufigkeit zu vermeiden – unter anderem durch eine Vielzahl deklarativer Operatoren. Können wir etwas Ähnliches auch selbst umsetzen?

Die Antwort lautet: Ja, das ist möglich! Zunächst benötigen wir jedoch Low-Level-Primitiven, um Daten zwischen gleichzeitig laufenden Virtual Threads zu übertragen.

Man könnte versucht sein, Java's blockierende Queues zu verwenden, stößt dabei aber schnell auf gewisse Einschränkungen. Zum Glück gibt es bereits bewährte Konzepte, die genau die Werkzeuge liefern, die wir brauchen. Besonders die Programmiersprache Go hat gezeigt, dass Channels ein äußerst praktischer und leistungsfähiger Weg sind, um zwischen Goroutines (die Virtual Threads in vielerlei Hinsicht ähneln!) zu kommunizieren und Nebenläufigkeit elegant zu lösen.

Auch Kotlin hat erfolgreich performante, Go-ähnliche Channels auf der JVM implementiert – basierend auf Coroutines. Glücklicherweise haben sie den Algorithmus in einem Fachartikel veröffentlicht – das ebnete den Weg für eine Virtual-Thread-basierte Implementierung, die nun Teil von Jox ist.

Im Vergleich zu blockierenden Queues bieten Channels zwei entscheidende Vorteile:

- Ein Channel kann als done markiert werden – also als abgeschlossen, sodass keine weiteren Elemente mehr gesendet werden. Alternativ kann er in einen Fehlerzustand versetzt werden, wodurch alle gepufferten Elemente verworfen und laufende Sende-/Empfangsoperationen abgebrochen werden.
- Eine select-Operation erlaubt es, genau einen Wert aus einer Liste von Channels zu empfangen (oder zu senden) – eine Funktionalität, die sich in komplexen parallelen Workflows als äußerst hilfreich erweist.

Der zweite grundlegende Baustein – besonders wichtig im Zusammenhang mit Fehlerbehandlung – ist das Konzept der Structured Concurrency. Eine vollständige Einführung würde den Rahmen dieses Artikels sprengen, daher hier nur ein kurzer Überblick.

Es gibt mehrere Möglichkeiten, strukturierte Nebenläufigkeit umzusetzen – einige davon werden in JEP 453 diskutiert. Dieses Java-Feature befindet sich derzeit in der Vorschauphase und wurde zwischen Java 22 und Java 23 grundlegend überarbeitet. Die Community hofft auf eine finale Version in Java 25 – wir dürfen gespannt sein!

Jox bietet eine alternative API für strukturierte Nebenläufigkeit (die intern allerdings auf der JEP-Implementierung basiert), mit besonderem Fokus auf Sicherheit und einfache Anwendbarkeit. Die zentrale Einheit ist ein Scope, innerhalb dessen parallele Tasks – also Virtual Threads – gestartet werden können:

```
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();
});
```

Die zentrale Eigenschaft eines Structured-Concurrency-Scopes ist die Garantie, dass bei Abschluss des supervised-Scopes alle gestarteten Forks beendet sind – entweder erfolgreich, mit einer Ausnahme oder durch Unterbrechung. Aus Sicht der Außenwelt werden Threads und Nebenläufigkeit so zu einem Implementation Detail.

Die Tatsache, dass kein Thread den Scope überleben kann, ist insbesondere im Fehlerfall entscheidend. Tritt in einem der Forks ein Fehler auf, wird der gesamte Scope standardmäßig heruntergefahren, verbleibende Forks werden unterbrochen, und der Scope wartet, bis alle beendet sind.

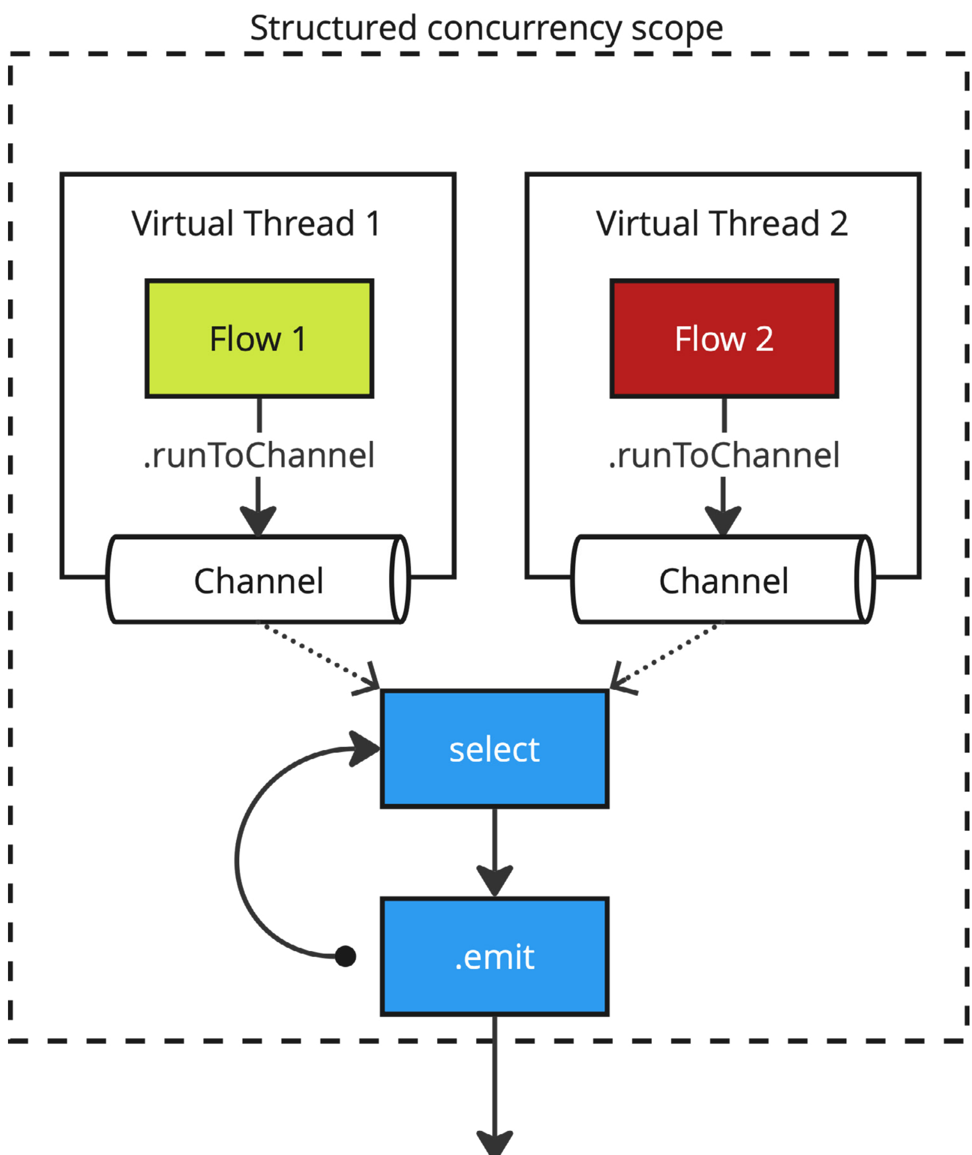
Mit diesen beiden Werkzeugen – Channels und Structured Concurrency – können wir jetzt auch konkurrente Streaming-Operatoren implementieren. Beginnen wir mit dem Mergen zweier Streams. Genau wie zuvor erstellen wir eine lazy evaluierte Flow-Beschreibung. Beim Ausführen des kombinierten Flows wird ein Structured-Concurrency-Scope erzeugt, in dem zwei Forks starten, die jeweils die untergeordneten Flows ausführen. Jeder Flow gibt seine Daten an einen Channel weiter.

Über die select-Funktion können wir dann jeweils den Wert aus dem Channel empfangen, der gerade Daten bereitstellt (die Behandlung von abgeschlossenen Streams wurde hier vereinfacht dargestellt):

```
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;
      });
    });
  }
}
```

Beachten Sie, dass wir auch hier erneut auf die in Java integrierten Kontrollfluss-Konstrukte zurückgreifen: das while-Konstrukt zur Wiederholung, switch mit Pattern Matching zur Auswahl von Channel-Antworten und die eingebaute Fehlerbehandlung mittels throw. So entsteht ein leistungsstarkes, aber dennoch natürlich lesbares API – ganz im Sinne von Project Loom.



In ähnlicher Weise – auch wenn mit etwas mehr Code – lassen sich weitere Streaming-Operatoren implementieren, etwa mapPar, zip, buffer, flatMap, grouped und viele mehr. Diese Operatoren orientieren sich an den bekannten Funktionalitäten aus Reactive-Streams-Bibliotheken wie Akka Streams oder Reactor.

Darüber hinaus verfügen wir nun über die Werkzeuge, um mit der Außenwelt zu interagieren: Wir können beispielsweise Bytestreams aus Dateien erzeugen, diese zeilenweise parsen oder ein Ergebnis direkt als InputStream ausgeben lassen.

Zum Abschluss folgt ein etwas umfangreicheres Beispiel, das einige der oben beschriebenen Funktionalitäten kombiniert:

```
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

Das klingt alles vielversprechend – doch wie sieht es mit der Resilienz unserer Implementierung aus? Können wir, wie bei Reactive Streams, garantieren, dass der Speicherverbrauch begrenzt bleibt? Schließlich haben wir in den bisherigen Beispielen keine explizite Backpressure-Weitergabe gesehen.

Die Wahrheit ist: Backpressure ist vorhanden, allerdings implizit – es ist nicht nötig, eigene Codepfade für deren Management zu definieren.

Backpressure entsteht durch begrenzte Puffer (Channels, Queues) und Blockierung von Virtual Threads. Wenn eine Verarbeitungsstufe z. B. in einem Fork im Hintergrund läuft (also asynchron), die nachgelagerte

Stufe jedoch nicht schnell genug Daten verarbeiten kann, füllt sich der Puffer des Channels irgendwann. Der .send-Aufruf blockiert dann den entsprechenden Thread – und pausiert so ganz natürlich die Produktion in der vorgelagerten Stufe. Backpressure ergibt sich hier also durch den Einsatz von thread-blockierenden Operationen – ein eleganter, natürlicher Mechanismus.

Fehlerbehandlung

Der letzte Aspekt, den wir im Zusammenhang mit Virtual Threads betrachten müssen, ist die Fehlerbehandlung.

Im Fall eines einfachen, synchronen Streams ist die Fehlerbehandlung relativ unkompliziert: Jede Ausnahme, die in einer Verarbeitungsstufe auftritt, wird über die .emit-Kette weitergereicht und schließlich in der jeweiligen Konsummethode (z. B. .runForeach oder .runDiscard) erneut ausgelöst.

Java’s eigene Fehlerbehandlungsmechanismen reichen in diesem Fall völlig aus. Komfortmethoden wie Flow.onError oder Flow.onComplete erstellen lediglich zusätzliche Stages, die mit try-catch bzw. try-finally arbeiten.

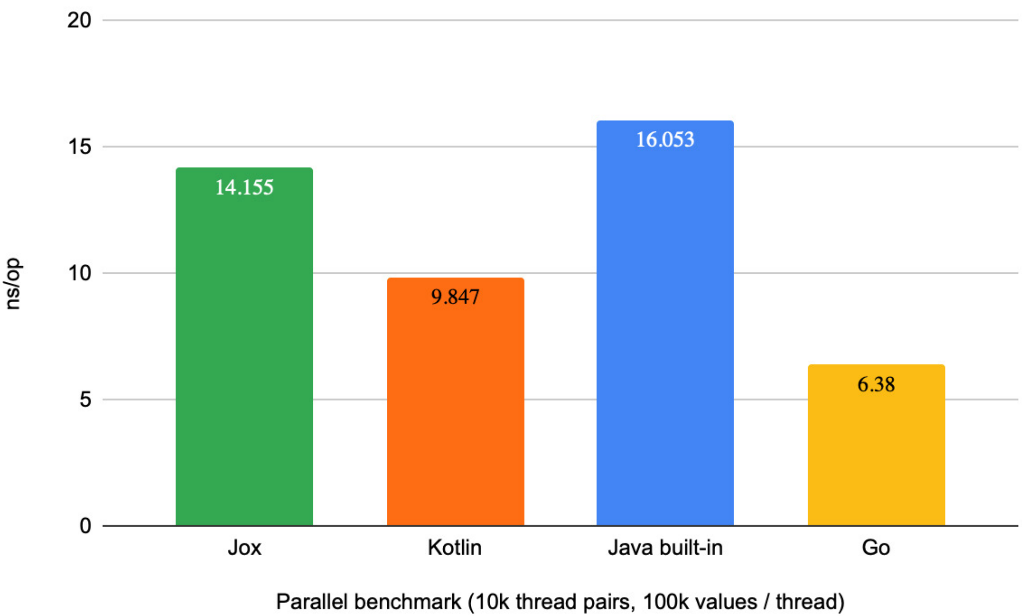
Bei asynchronen Streams – also solchen, die mehrere Forks (Threads) starten und Flows im Hintergrund ausführen – ist die Situation nur leicht komplexer. Und das nur dank strukturierter Nebenläufigkeit, wie wir sie zuvor besprochen haben: Wenn ein Fork eine Ausnahme wirft, wird der gesamte Scope heruntergefahren. Die Ausnahme (gegebenenfalls verpackt) wird jedoch erst dann erneut ausgelöst, wenn alle anderen Forks beendet bzw. unterbrochen wurden.

Daher macht es aus Sicht anderer Verarbeitungsstufen keinen Unterschied, ob eine bestimmte Stage asynchron ausgeführt wird oder nicht. Threading wird zu einem Implementierungsdetail – wenn auch zu einem wichtigen.

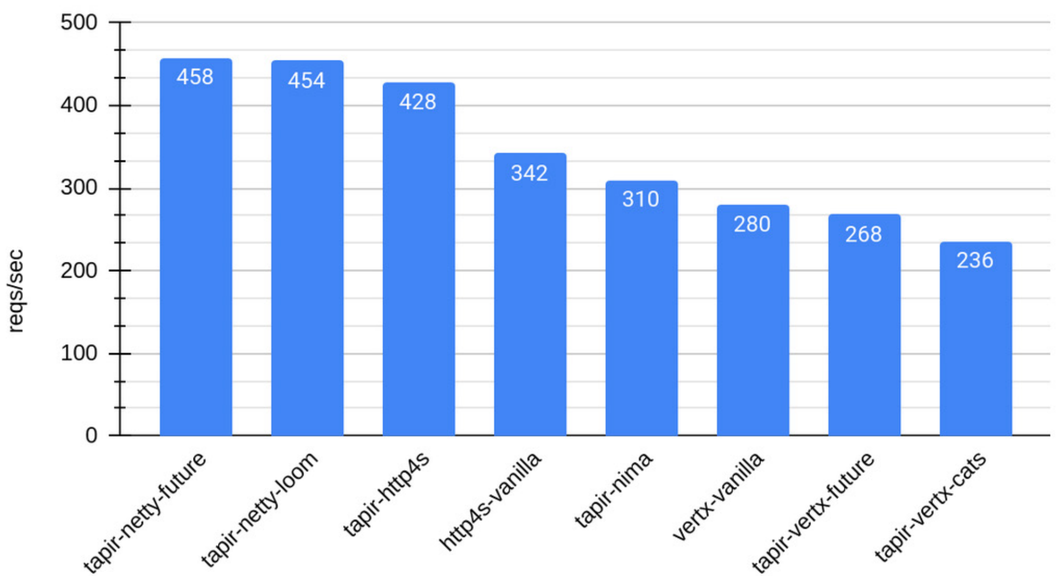
Performance

Was die Funktionalität betrifft, so lässt sich mit unserer Lösung ein Großteil der Features von Reactive-Streams-Bibliotheken nachbilden. Aber wie sieht es mit der Performance aus?

Ein vollständiges Benchmarking beider Streaming-Ansätze steht noch aus, daher verfügen wir bislang nur über Teildaten. Immerhin haben wir den Channel, den wir zur Kommunikation zwischen asynchron laufenden Stages verwenden, bereits benchmarkt, profiliert und optimiert. Das Ergebnis: Die Performance ist vergleichbar mit den Concurrent-Datenstrukturen von Java sowie mit Kotlin's Channel-Implementierung – und liegt nur leicht hinter Go zurück.



Zweitens haben wir die Performance von HTTP-Servern, die entweder auf Loom oder auf Reactive-Bibliotheken basieren, miteinander verglichen (Benchmark hier). In unseren Tests konnten wir keine signifikanten Unterschiede feststellen.



Das sollte kaum überraschen: Die Grundidee – nämlich ein Pool von Betriebssystem-Threads, auf dem viele „virtuelle" Threads (Coroutines, Futures etc.) ausgeführt werden – ist sowohl bei „reactiven" als auch bei „Loom"-Ansätzen dieselbe.

Fazit

Kann man eine Virtual-Thread-basierte Bibliothek mit Funktionsumfang vergleichbar zu bestehenden Reactive-Streams-Implementierungen entwickeln? Ja. Das Projekt Jox ist der Beweis.
Werden Virtual Threads Reactive Streams vollständig ersetzen? Teilweise. Die Reactive-Streams-Spezifikation bleibt weiterhin ein nützlicher Standard zur Interoperabilität. Sie wird vermutlich auch in Zukunft eine wichtige Rolle bei der Umsetzung von performanter Low-Level-Integrationslogik spielen. So implementiert auch Jox das Publisher-Interface von Reactive Streams, um die Integration mit Drittanbieter-Bibliotheken zu ermöglichen, die diesen Standard unterstützen.

Wenn es jedoch um das Schreiben von Datenverarbeitungspipelines geht, wird die Popularität klassischer Reactive-Streams-Bibliotheken wahrscheinlich zurückgehen. Das Modell der Virtual Threads ist einfacher, sowohl beim Schreiben als auch beim Lesen von Code. Es ermöglicht die Nutzung der nativen Java-Kontrollstrukturen und des Exception-Handling-Modells – ganz ohne Function Coloring (Viralität von Futures). Außerdem ist keine zusätzliche Runtime auf Bibliotheksebene notwendig, da die Thread-Verwaltung direkt von der JVM übernommen wird.

Bibliotheken wie Akka Streams oder Reactor haben aktuell noch die Nase vorn, was Reife, API-Vielfalt und Integrationen betrifft. Doch das Virtual-Threads-Ökosystem wird diese Lücke schließen – und wenn meine Einschätzung richtig ist, es langfristig sogar ablösen.

Virtual Threads bieten eine robuste und performante Plattform, auf deren Basis sich Systeme entwickeln lassen, die von den Prinzipien reaktiver Architekturen profitieren – und sie gleichzeitig weiterentwickeln. Sie eröffnen einen deutlich entwicklerfreundlicheren Weg, resiliente Systeme zu bauen.

In gewisser Weise ist das Aufkommen der Virtual Threads die ultimative Bestätigung der Grundideen hinter Reactive Streams, deren Wert so groß war, dass sie schließlich direkt in die JVM aufgenommen wurden.

> Zurück zum Inhaltsverzeichnis

#JAVAPRO #PERFORMANCE

So beschleunigen Sie existierende Deployments mit den richtigen JVM-Features

Autor:

Dmitry Chuyko ist OpenJDK-Committer, Artikelautor und Sprecher auf internationalen Konferenzen. Seine früheren Erfahrungen bei Unternehmen wie Oracle und der Deutschen Bank haben gezeigt, dass die interessantesten Anwendungsprobleme in Verbindung mit der zugrunde liegenden Plattform gelöst werden können. Dmitrys Schwerpunkt bei BellSoft liegt auf der Optimierung der JVM für x86 und ARM sowie der Entwicklung kleiner, schneller und sicherer JDK-Container.

https://www.linkedin.com/in/dchuyko/

In der heutigen Tech-Landschaft sehen sich Java-Anwendungen mit einer entscheidenden Herausforderung konfrontiert. Unternehmen müssen steigenden Performance-Anforderungen gerecht werden und gleichzeitig die Kosten für die Infrastruktur unter Kontrolle halten – und das ohne kostspielige Neuentwicklungen.

Diese Herausforderung hat sich mit der Einführung von Cloud Computing noch weiter verschärft. Jede Millisekunde Latenz und jedes Megabyte an

Speicher wirken sich nun direkt auf die laufenden Kosten aus. Die Zahlen sind beachtlich: Basierend auf meinen Erfahrungen können schlecht konfigurierte JVMs die Cloud-Kosten um 30–50 % erhöhen. Für große Unternehmen entspricht das Millionen an vermeidbaren Ausgaben.

Häufige Optimierungsfehler

Trotz der offensichtlichen finanziellen Konsequenzen gehen die meisten Unternehmen die JVM-Optimierung nicht effektiv an. Diese vier Fehler beobachten wir immer wieder:

1. **Verwendung der Standardeinstellungen:** Viele Teams nutzen einfach die vorkonfigurierten JVM-Einstellungen. Diese Standardkonfigurationen legen den Fokus eher auf Kompatibilität als auf Performance, was zu ineffizienter Heap-Nutzung, suboptimalen Garbage-Collection-Muster und ressourcenintensive Start- und Warmup-Phasen führt.

2. **Willkürliches Kopieren von Konfigurationen:** Bei Optimierungsversuchen greifen Teams oft auf Einstellungen aus Blogs oder Foren zurück. Diese Konfigurationen sind allerdings zumeist für andere Workloads oder JVM-Versionen entwickelt und verschlechtern oft eher die Performance, als sie zu optimieren.

3. **Lösung von Problemen mit Ressourcen:** Statt die zugrunde liegenden Probleme zu beheben, steigern viele Unternehmen schlicht CPU, Speicher und Maschinen. So geraten sie in einen Teufelskreis wachsender Kosten, ohne das Problem an der Wurzel anzugehen.

4. **Fokus auf die falschen Optimierungen:** Selbst gezielte Performance-Optimierungen fokussieren sich oft auf die falschen Bereiche. Teams können beispielsweise Wochen mit der Feinjustierung der Garbage Collection zubringen, während sie einfache Konfigurationsänderungen, die einen viel größeren Einfluss hätten, unbeachtet lassen.

Diese Muster halten sich, weil viele Teams die modernen Performance-Merkmale der JVM und die leicht zugänglichen Optimierungsmöglichkeiten nicht vollständig erfassen.

Die neue Realität verstehen

Bevor wir die uns zur Verfügung stehenden Quick Wins erläutern, müssen wir die grundlegenden Veränderungen verstehen, welche die heutige JVM-Landschaft geprägt haben:

1. **Containerisierung:** Die meisten Java-Anwendungen laufen heutzutage in Containern. Dadurch ergeben sich neue Herausforderungen wie beispielsweise die Interaktion der JVM mit Ressourcengrenzen, eröffnet allerdings auch neues Optimierungspotenzial.

2. **Fortschrittliche JVM-Features:** Moderne JVMs bieten leistungsstarke Performance-Features, die in den meisten Deployments nicht genutzt werden. Von ausgeklügelten Garbage Collectors bis hin zur Startbeschleunigung bieten diese Features immense Vorteile bei minimalem Implementierungsaufwand.

3. **Cloud-Ökonomie:** Durch die Verlagerung von Anwendungen in die Cloud sind Performance und Betriebskosten nun eng miteinander verknüpft. Bei Optimierungen geht es nicht länger nur um die Benutzererfahrung – sie ist eine Notwendigkeit zur Kostenkontrolle.

Diese Veränderungen erfordern eine neuen Ansatz in Sachen JVM-Performance – und zwar einen, der die modernen Möglichkeiten nutzt, um die spezifischen Herausforderungen containerisierter, Cloud-nativer Deployments zu bewältigen.

In diesem Artikel werden wir schrittweise immer fortschrittlichere Optimierungstechniken erläutern, mit denen sich die Performance bei nur minimalem Aufwand drastisch steigern lässt.

Beginnen wir unsere Reise in die Welt der JMV-Performance-Optimierung mit den Grundlagen: die richtige Container-Konfiguration. Hier sind keine Änderungen der Anwendung erforderlich – nur eine einsichtige Dockerfile-Konstruktion und JVM-Einstellungen, die sofortige Performance-Steigerungen ermöglichen. Diese Quick Wins sind bei Java-Anwendungen Ihr erster Schritt, um Performance-Einbußen mit minimalem Aufwand und Risiko einzudämmen.

Einfache Performancesteigerung mit Container-Optimierung

Mit Containern hat sich die Art und Weise revolutioniert, wie wir Java-Anwendungen verpacken und bereitstellen. Dennoch erkennen viele Unternehmen nicht die entscheidenden Möglichkeiten der Performance-Optimierung, welche die Konfiguration von Containern bieten. Bevor fortschrittliche JVM-Justierungen angegangen werden, können bereits die Art und Weise, wie wir unsere Container-Images erstellen und ihre Laufzeitumgebungen konfigurieren, erheblich zur Performance-Steigerung beitragen.

Tipp 1: Das Basis-Image ist entscheidend

Der Ausgangspunkt jeder containerisierten Java-Anwendung ist die Wahl des richtigen Basis-Images. Diese scheinbar simple Entscheidung hat weitreichende Auswirkungen auf alle Bereiche von der Startzeit bis hin zum Speicherverbrauch.

Viele Entwickler greifen standardmäßig auf allgemeine Linux-Distributionen als Basis-Image zurück. Diese Distributionen sind in der Regel rund 300 MB groß und enthalten hunderte von Utilities, Bibliotheken und Services, die für die Ausführung von Java-Anwendungen irrelevant sind. Einige Anbieter bieten schlankere Versionen ihres Betriebssystems an, wie zum Beispiel Red Hats UBI Minimal (80–100 MB) oder Debian Slim. Ein Schritt in die richtige Richtung, der aber noch weitergehen kann.

Die musl-basierten Linux-Distributionen sind deutlich kleiner als solche mit glibc. Zu dieser Erkenntnis gelangte man, als BellSoft 2019 den Alpine Linux Port (JEP 386) zum OpenJDK beitrug und damit die richtige Unterstützung für musl libc einführte. Diese Innovation führte zu dramatischen Verringerungen der Image-Größe bei Java-Anwendungen.

Noch weiter geht die Optimierung mit der speziell für Java-Workloads optimierten Linux-Distribution Alpaquita. Sie bietet eine geringe Image-Größe, lässt Nutzern die Wahl zwischen musl- und glibc-Varianten und bietet darüber hinaus Laufzeitoptimierungen, die speziell auf JVM-Workloads abgestimmt sind, auf die wir in diesem Artikel noch näher eingehen werden.

Alternativ können Sie auf einsatzbereite Java-Images wie den Liberica Runtime Container zurückgreifen. Er ist in Alpaquita Linux und Liberica JDK Lite enthalten, einer minimierten Version des OpenJDK-Builds. Durch diese Kombination können Sie bis zu 30 % RAM und Festplattenspeicher bei Cloud-Deployments einsparen – und das ohne manuelle Feinjustierung.

Ein weiterer einfacher Optimierungsansatz besteht in der Verwendung von JRE anstelle des vollständigen JDK bei Produktion-Containern. Diese Änderung macht sich besonders bei Microservice-Architekturen mit Dutzenden oder Hunderten von Instanzen besonders bemerkbar, da der Speicherbedarf ohne Performance-Einbußen um 15–25 % verringert wird.

Diese Optimierungen dienen allerdings nicht nur der Verringerung der Image-Größe – obwohl allein dadurch schon eine Verringerung der Deployment-Zeit und des Netzwerk-Traffics erzielt wird. Viel wichtiger ist noch, dass speziell für Java entwickelte Basis-Images häufig Performance-orientierte Konfigurationen und Unterstützung für Performance-Features bieten, die bei Universal-Images fehlen.

Tipp 2: Trennung von Anwendungsschichten

Die Schichtung von Container-Images mutet auf den ersten Blick wie ein technisches Detail an, stellt aber eine weitere Möglichkeit der Performance-Optimierung dar.

Statt eine einzige Schicht zu verwenden, sollten Sie die Komponenten basierend auf ihrer Änderungsfrequenz trennen:

```
# Ineffiziente Vorgehensweise - eine Schicht für alle Abhängigkeiten
COPY ./target/application.jar app.jar

# Optimierte Vorgehensweise - Trennung von Schichten basierend auf
Änderungsfrequenz
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
```

Diese Vorgehensweise nutzt Dockers Caching-Mechanismen, sodass es nur bei Änderungen ein Rebuild und Transfer durchgeführt werden. Bei der Aktualisierung eines Microservices könnten so beispielsweise lediglich 10 KB des geänderten Codes statt des gesamten Images mit 200 MB oder mehr übertragen werden. Bei Hunderten von Containern und häufigen Deployments kann Ihr Netzwerk-Traffic so um ein Vielfaches reduziert werden.

Tipp 3: JVM-Speicherkonfiguration

Die wichtigste Optimierungsmaßnahme für Container ist die richtige JVM-Speicherkonfiguration. Ältere JVM-Versionen erkennen standardmäßig keine Container-Speichergrenzen, was zu Out-of-Memory-Fehlern oder einer nicht optimal genutzten Ressourcen führen kann.

Eine einfache Lösung bietet die Speicherkonfiguration:

MaxRAMPercentage=80: Verwendet 80 % des Container-Speichers (je nach Anwendungstyp anzupassen)

```
# Häufiger Fehler - keine Speichereinstellungen
ENTRYPOINT [„java“, „-jar“, „app.jar“]



# Optimale Vorgehensweise - prozentuale Einstellungen
ENTRYPOINT [„java“, \
„-XX:MaxRAMPercentage=80“, \
„-jar“, „app.jar“]
```

Diese einfachen Änderungen passen die JVM-Ressourcennutzung optimal an die Container-Grenzen an und verbessern den Durchsatz oftmals um 20–40 %, ohne dass Codeänderungen erforderlich sind.

Diese Container-Optimierungen stellen die erste Reihe an Quick Wins zur Performance-Steigerung von Java-Anwendungen dar. Von der Auswahl des Basis-Images bis hin zur Speicherkonfiguration verspricht jeder Schritt signifikante Verbesserungen bei minimalem Aufwand, was die Optimierungen für Teams aller Erfahrungsstufen anwendbar macht.

Als nächstes widmen wir uns einer weiteren wichtigen Performance-Herausforderung: der Startzeitoptimierung. Wer die ressourcenintensive Initialisierungsphase von Java-Anwendungen versteht und gezielt anzupassen weiß, kann weitere signifikante Verbesserungen in Sachen Deployment-Effizienz und Ressourcennutzung erzielen.

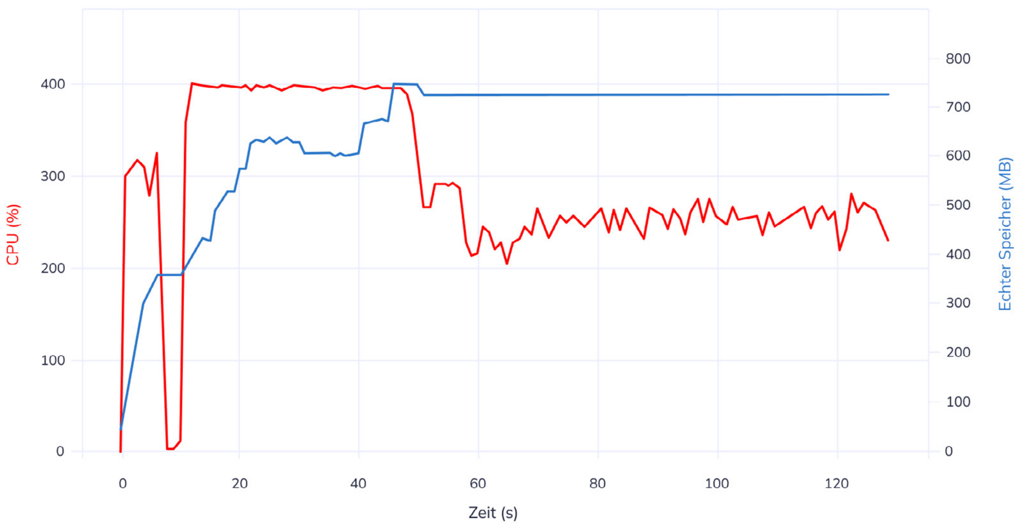
Der Weg zur Near-Zero-Startzeit

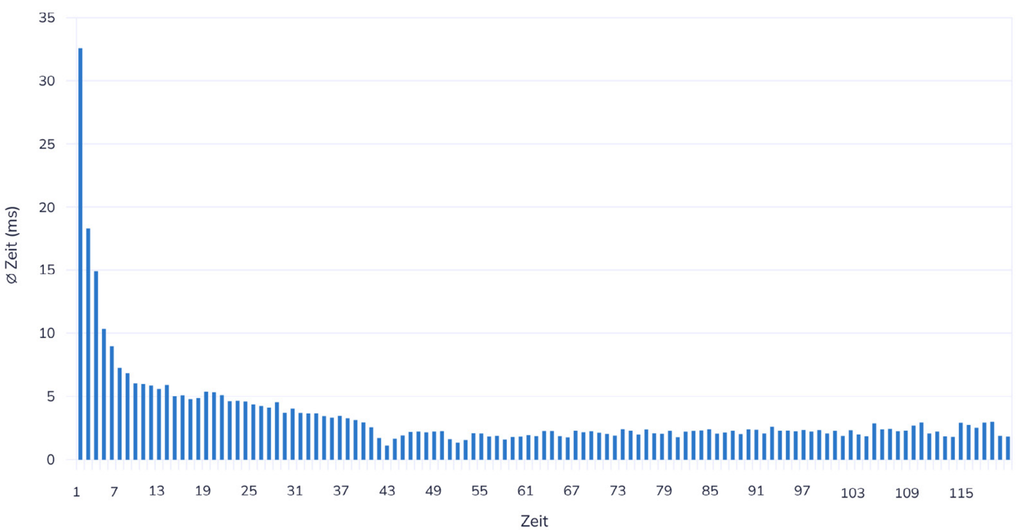
Java-Anwendungen haben ein Ressourcennutzungsmuster, das erhebliche Skalierungsherausforderungen mit sich bringt. Statt einen gleichmäßigen Ressourcenverbrauch über den gesamten Zyklus hinweg aufrechtzuerhalten, folgen sie einem charakteristischen „Peak-then-Plateau"-Muster, bei dem der Verbrauch zu Beginn der Startphase extrem hoch ist und subsequent im Normalbetrieb stark sinkt.

Dieses Ressourcennutzungsprofil bringt eine grundlegende Skalierungsineffizienz mit sich. Besonders die horizontale Skalierung gestaltet sich problematisch, da jede neue Java-Instanz eine solche intensive Initialisierungsphase erfordert. So kann das Hinzufügen weiterer Instanzen sogar die Gesamt-Performance des Systems verringern, da Ressourcen für den Startprozess statt für tatsächliche Anfragen verbraucht werden.

Werfen wir einen Blick auf echte Performance-Daten aus unserem Test mit einer typischen Spring-Boot-Anwendung:

- **CPU-Nutzung:** Peak bei 400 % (4 Kerne) zu Beginn der Startphase, pendelt sich im Normalbetrieb bei ~ 100 % ein.
- **Speichernutzung:** Hoch zu Beginn der Heap-Initialisierung, stabilisiert sich nach mehreren Garbage-Collection-Zyklen.
- **Antwortzeit:** Die ersten Anfragen benötigen 5–10-mal länger als Anfragen im Normalbetrieb.
- **Startzeitlänge:** 41 Sekunden, um (unter Testbedingungen) bei 1000 Anfragen/Sekunde voll betriebsbereit zu sein.





In Production-Umgebungen mit komplexeren Initialisierungs-anforderungen kann die Startzeit auf 15–20 Minuten steigen. Während dieses Zeitraums können die Antwortzeiten 5–10-mal langsamer als standardmäßig vorgesehen sein, wobei auch die Fehlerquote häufig ansteigt, während Verbindungspools und Caches initialisiert werden. Dies führt zu einem kritischen Skalierungsproblem: Die Ressourcen können nicht verdoppelt werden, um die Performance zu verdoppeln. Nicht optimierte Java-Anwendungen bringen unweigerlich eine Unterauslastung der Ressourcen und erhöhte Cloud-Kosten mit sich und liefern gleichzeitig eine inkonsistente Benutzererfahrung bei Skalierung.

4 Möglichkeiten zur Verringerung der Startzeit

Das Java-Ökosystem bietet mehrere starke Quick Wins, um Herausforderungen bei Startzeit-Engpässen zu meistern, wobei jeder Schritt raffinierter als der vorangegangene ist.

Level 0: Client VM – Der einfache Switch

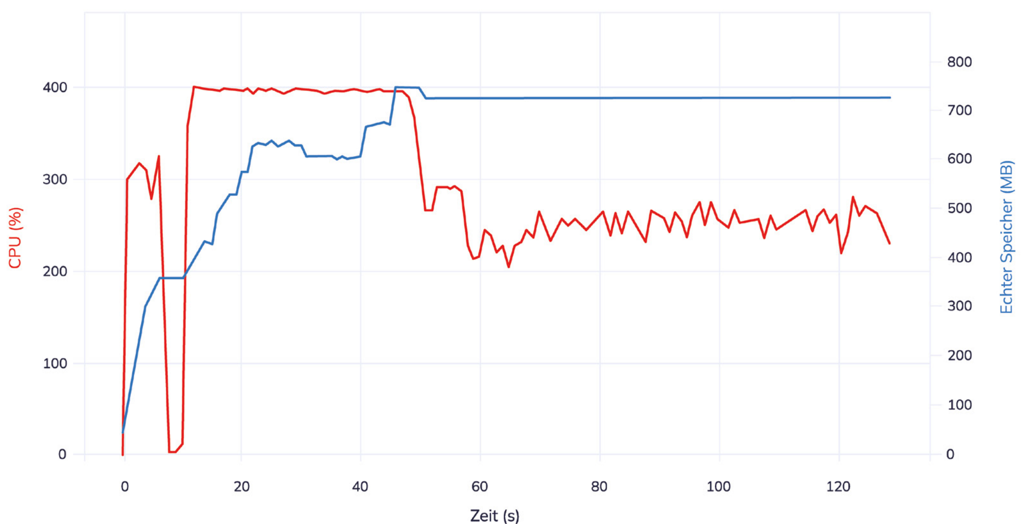
Die schlichteste Strategie setzt auf die oft übersehene Client VM, die gezielt einen schnellen Start optimiert, statt den langfristigen Durchsatz zu maximieren:

```
ENTRYPOINT [„java“, „-client“, „-jar“, “app.jar“]
```

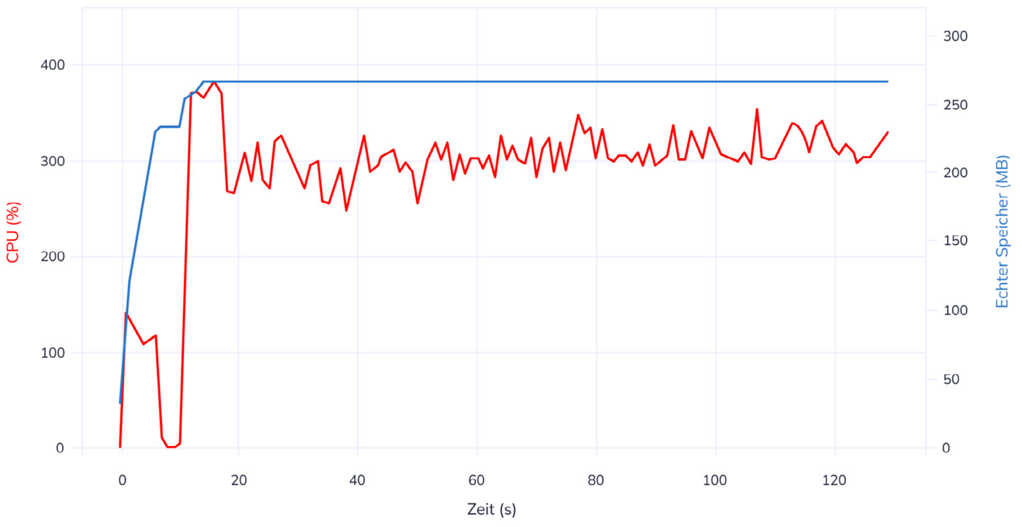
Hinweis: Hierfür benötigen Sie ein JDK-Bundle, das eine Client VM enthält.

Die Tests mit einer standardmäßigen Spring-Boot-Anwendung ergeben die folgenden Werte:

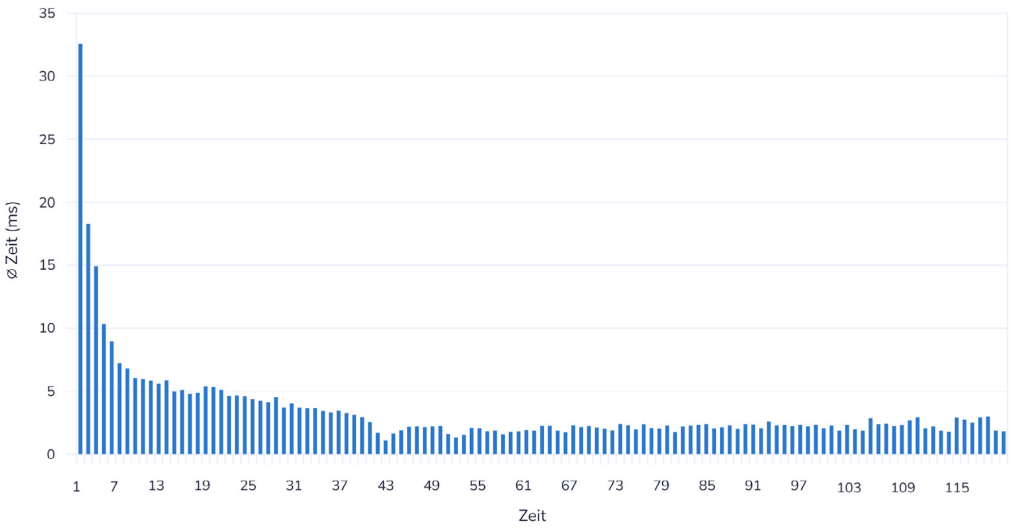
- **Startzeit:** um 15–25 % verringert.
- **CPU-Verbrauch:** Niedrigerer anfänglicher Peak, Initialisierung erfordert normalerweise 1–2 Kerne statt 3–4.
- **Speicherverbrauch:** rund 10 MB weniger.
- **Trade-off:** Möglicherweise niedrigerer Peak-Durchsatz bei anhaltend hoher Last.



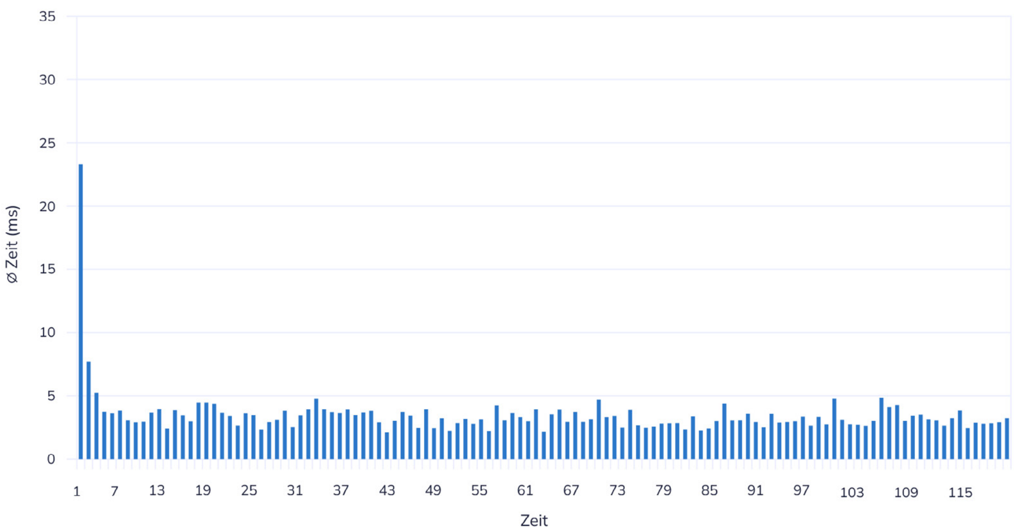
Client VM



Server VM



Client VM



Diese Optimierung erweist sich insbesondere bei serverlosen Deployments, kurzlebigen Prozessen und Anwendungen mit moderatem Durchsatz als vorteilhaft. AWS Lambda empfiehlt beispielsweise die Option -XX:TieredStopAtLevel=1. Damit verhält sich die Server VM beim Kompilieren ähnlich wie die Client VM,wodurch Kaltstartzeiten erheblich verkürzt werden, während die Performance für serverlose Funktionen auf einem akzeptablen Niveau bleibt.

Level 1: Class Data Sharing – Wiederverwendung geladener Klassen

Das Class Data Sharing (CDS) optimiert einen der ressourcenintensivsten Aspekte des Java-Startvorgangs: das Laden von Klassen. Durch die Erstellung im Speicher abgelegter Archive bereits geladener Klassen verkürzt CDS die Startzeit erheblich und reduziert zugleich den Speicherverbrauch:

```
# Erste Phase: Erstellen eines geteilten Archivs
java -XX:ArchiveClassesAtExit=./application.jsa -jar target/app.jar

# Starten der Anwendung und Nutzung des geteilten Archivs:
java -XX:SharedArchiveFile=application.jsa -jar target/app.jar
```

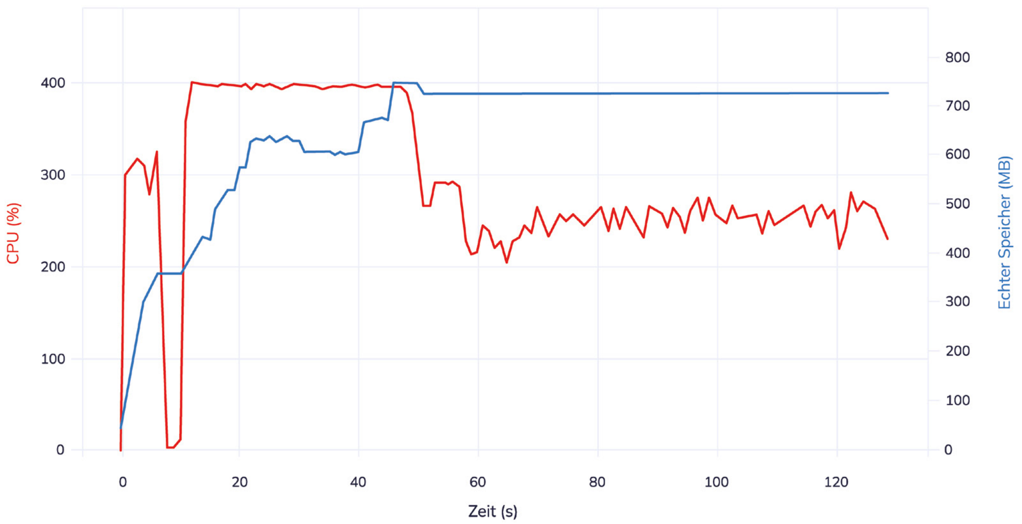
Bei Spring-Boot-Anwendungen gestaltet sich diese Optimierung noch einfacher, da CDS bereits nativ unterstützt wird:

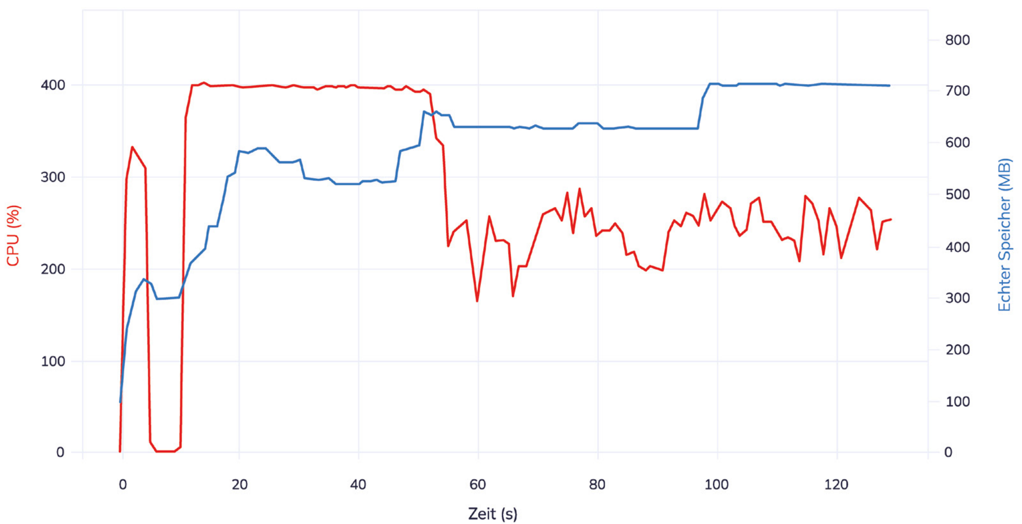
```
# Erstellen eines ausführbaren JAR
FROM bellsoft/liberica-runtime-container:jdk-23-stream-musl as
builder

WORKDIR /home
ADD app /home/app
RUN cd app && ./mvnw clean package

# Erstellen eines geschichteten JAR
FROM bellsoft/liberica-runtime-container:jdk-23-cds-slim-musl as
optimizer

WORKDIR /app
```

```
COPY --from=builder /home/app/target/*.jar app.jar
RUN java -Djarmode=tools -jar app.jar extract --layers --launcher

# Kopieren der Anwendungsschichten in das neue Basis-Image und
Erstellung des Archivs
FROM bellsoft/liberica-runtime-container:jdk-23-cds-slim-musl

COPY --from=optimizer /app/app/dependencies/ ./
COPY --from=optimizer /app/app/spring-boot-loader/ ./
COPY --from=optimizer /app/app/snapshot-dependencies/ ./
COPY --from=optimizer /app/app/application/ ./

# Ausführen der Anwendung im Container zur Erstellung des Archivs.
Die Anwendung wird automatisch geschlossen. Aktivierung von Spring
AOT.
RUN java -Dspring.aot.enabled=true \
  -XX:ArchiveClassesAtExit=./application.jsa \
  -Dspring.context.exit=onRefresh \
  org.springframework.boot.loader.launch.JarLauncher

# Ausführen der Anwendung mit aktivierter Spring AOT und unter
Verwendung des geteilten Archivs
ENTRYPOINT [„java“, \
  „-Dspring.aot.enabled=true“, \
  „-XX:SharedArchiveFile=application.jsa“, \
  „org.springframework.boot.loader.launch.JarLauncher“]
```

Tests zeigen, dass CDS in der Regel folgende Vorteile bietet:

- **Startzeit:** Verringerung um 30–40 %
- **Speichereinsparungen:** 5–15 % weniger Speichernutzung dank geteiltem Metaspace
- **CPU-Verringerung:** weniger ausgeprägter initialer CPU-Spitzenlast beim Start
- **Kompatibilität:** Funktioniert mit nahezu allen Java-Anwendungen, ohne dass Code-Änderungen erforderlich sind.





Level 2: Native Image von Grund auf kompiliert

GraalVM Native Image verfolgt einen radikaleren Ansatz, indem Java-Anwendungen im Voraus zu eigenständigen nativen Executables kompiliert:

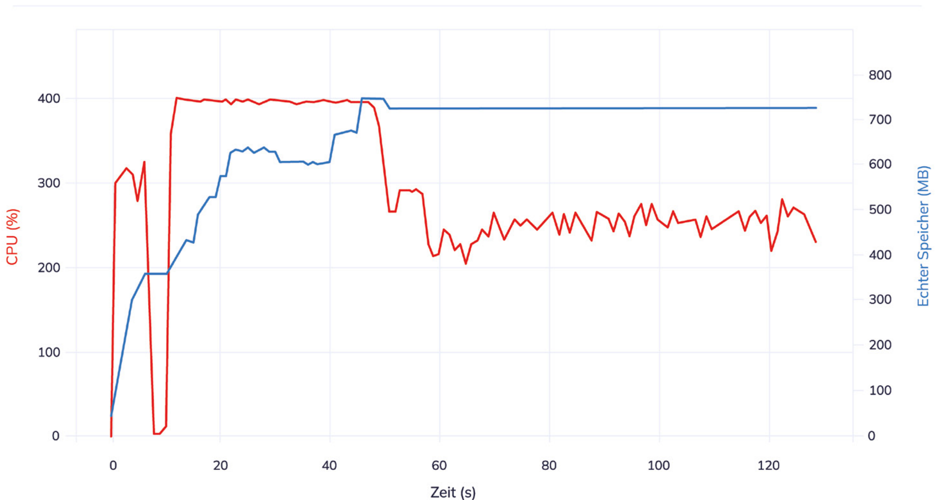
```
# Build-Phase mit GraalVM CE-basiertem Liberica Native Image Kit
FROM bellsoft/liberica-native-image-kit-container:jdk-21-nik-23.
1-stream-musl
WORKDIR /home/myapp
COPY Demo.java /home/myapp/
RUN javac Demo.java
RUN native-image Demo
```

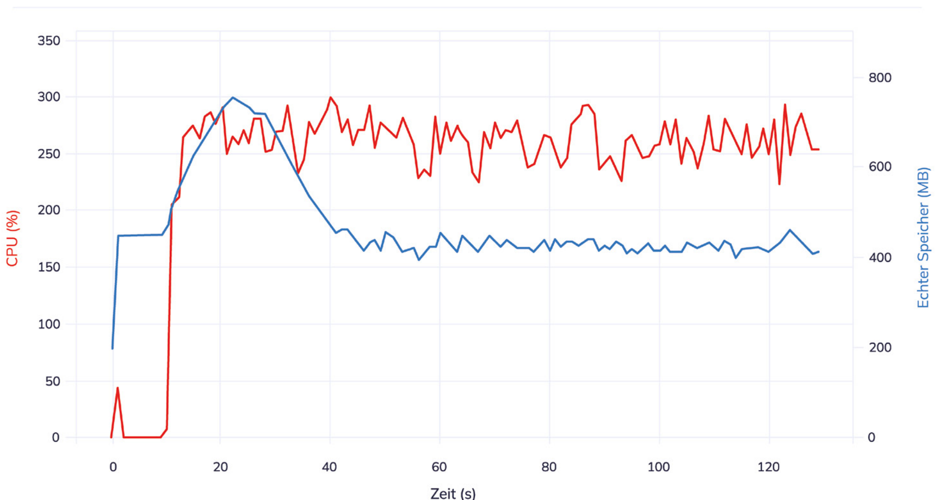
```
# Run-Phase
FROM bellsoft/alpaquita-linux-base:stream-musl
WORKDIR /home/myapp
COPY --from=0 /home/myapp/demo.
CMD [“./demo”]r
```

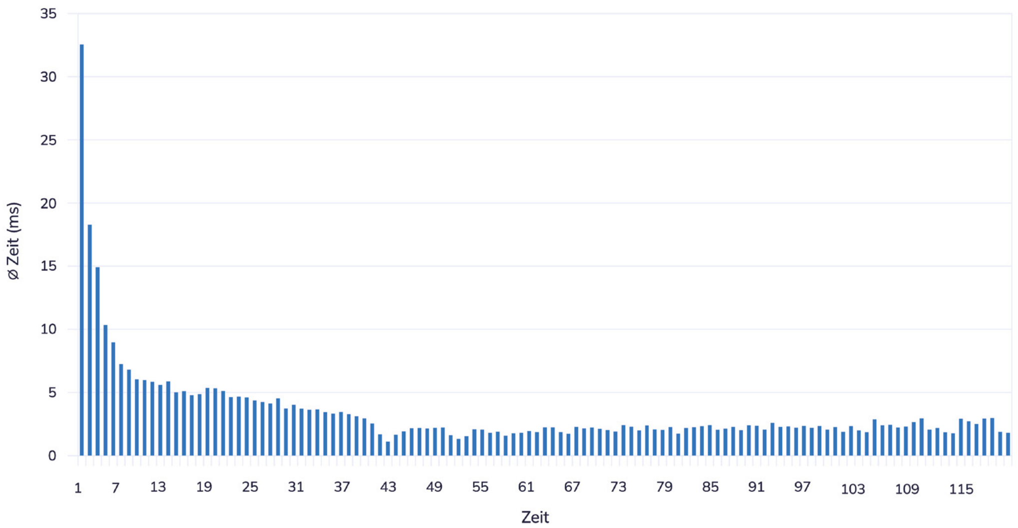
Für Spring-Boot-Anwendungen wird der Native-Image-Prozess durch die eingebaute Unterstützung weiter vereinfacht: ./mvnw -Pnative spring-boot:build-image.

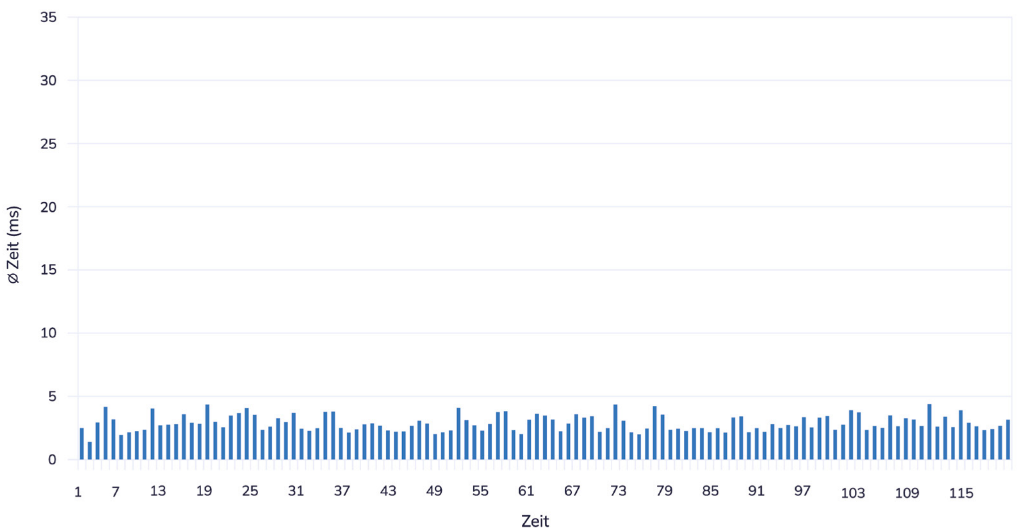
Native Image liefert drastische Verbesserungen beim Start:

- **Startzeit:** 10–100-mal schneller (Millisekunden statt Sekunden)
- **Speicherverbrauch:** Häufig 50 % geringer
- **Trade-off:** Einschränkungen bei der Reflexion, möglicherweise geringerer Spitzendurchsatz









Level 3: Checkpoint/Restore (CRaC) – Einfrieren einer laufenden Anwendung

Der fortschrittlichste Ansatz nutzt das OpenJDK-Projekt Coordinated Restore at Checkpoint (CRaC). CRaC funktioniert wie das Erstellen eines Snapshots Ihrer vollständig initialisierten und laufenden Java-Anwendung, die dann „eingefroren" wird. Sie können diesen Snapshot sofort wieder „auftauen" und wiederherstellen, wodurch die gesamte Start- und Warmup-Phase umgangen wird:

```
# Erstellen des Container-Images, Starten und Beenden der Anwendung
in einem Container zu Erstellung einer gespeicherten Datei der
Anwendung
FROM bellsoft/liberica-runtime-container:jdk-21-crac-musl as builder
```

```
WORKDIR /home
ADD app /home/app
RUN cd app && ./mvnw clean package

FROM bellsoft/liberica-runtime-container:jre-21-crac-slim-musl as
optimizer

WORKDIR /app
COPY --from=builder /home/app/target/app.jar /app/app.jar

RUN java -Djarmode=tools -jar app.jar extract --layers --launcher

FROM bellsoft/liberica-runtime-container:jre-21-crac-slim-musl

# Bleiben Sie als Root im Container, um CRaC zu nutzen.
VOLUME /tmp
EXPOSE 8080

COPY --from=optimizer /app/app/dependencies/ ./
COPY --from=optimizer /app/app/spring-boot-loader/ ./
COPY --from=optimizer /app/app/snapshot-dependencies/ ./
COPY --from=optimizer /app/app/application/ ./

ENTRYPOINT [„java“, „-Dspring.context.checkpoint=onRefresh“,
„-XX:CRaCCheckpointTo=/checkpoint“, „-XX:MaxRAMPercentage=80.0“,
„org.springframework.boot.loader.launch.JarLauncher“]
```

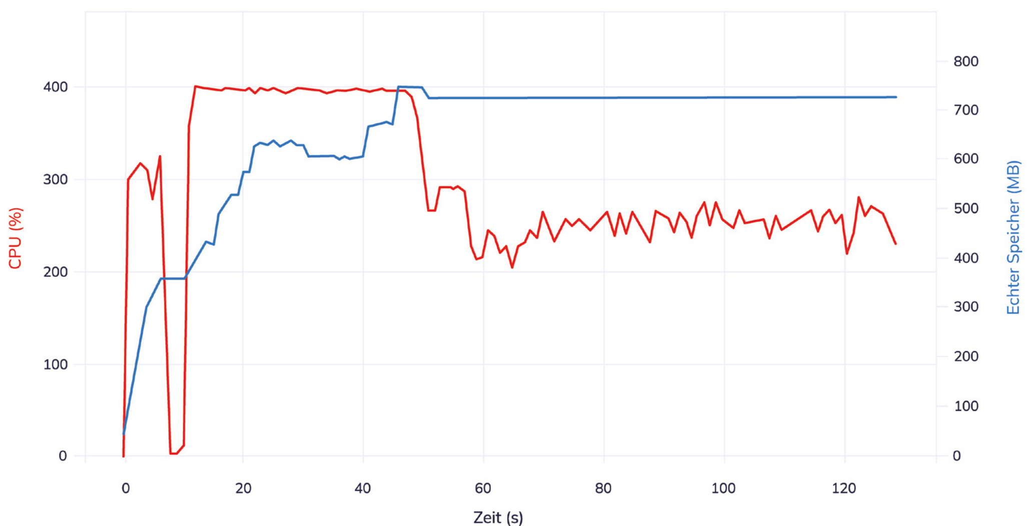
Hier ist der vollständige Workflow zur Implementierung von CRaC in Ihrer Deployment-Pipeline vom Container-Start über das Erstellen des Checkpoints bis hin zum Starten der Anwendung aus dem gespeicherten Zustand:

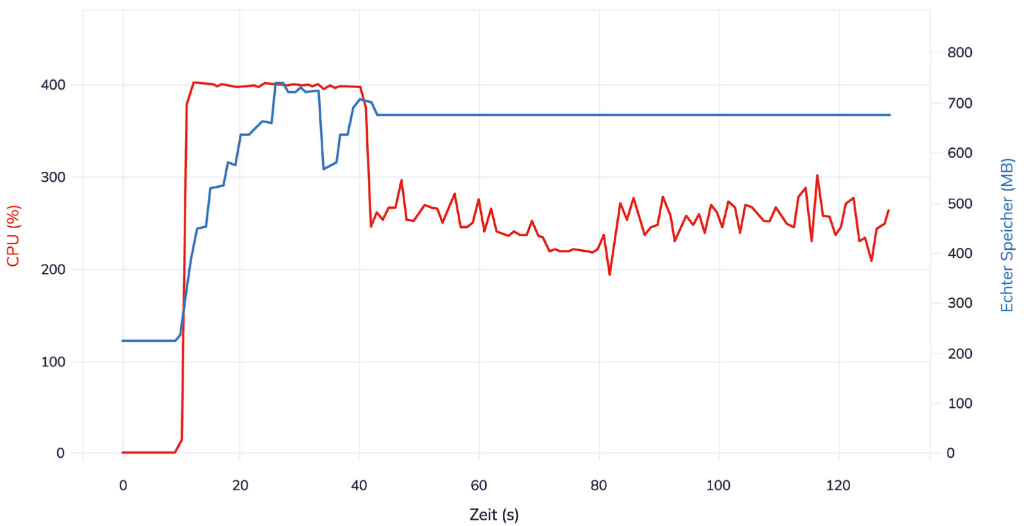
```
# Erstellen des Container-Images
docker build . -t app-crac-checkpoint -f Dockerfile-crac

# Starten des Container-Images
docker run --cap-add CHECKPOINT_RESTORE --cap-add SYS_PTRACE --
name app-crac app-crac-checkpoint
```

```
# Übertragen des Image-Inhalts in ein neues Image und Ändern des
Entrypoints, um die Anwendung aus der Datei neu zu starten:
docker commit --change='ENTRYPOINT [„java“, „-XX:CRaCRestoreFrom=/
checkpoint“]' app-crac app-crac-restore

# Entfernen des ursprünglichen Containers:
docker rm -f app-crac

# Ausführen des zweiten Images mit der gespeicherten Anwendung:
docker run -it --rm -p 8080:8080 --cap-add CHECKPOINT_RESTORE --cap-
add SYS_PTRACE --name app-crac app-crac-restore
```

CRaC bietet die ultimative Startzeit-Optimierung:

- **Startzeit:** nahezu sofort (Millisekunden)
- **Speichereinsparungen:** vorinitialisierter Heap ohne Störungen durch Garbage Collection
- **CPU-Verringerung:** Fängt nahezu alle CPU-Spitzenlasten beim Start ab
- **Kompatibilität:** Erfordert eine CRaC-kompatible JVM (z. B. Liberica JDK CRaC) sowie Anwendungsänderungen für eine sichere und saubere Checkpoint/Restore-Funktion.





Die Zukunft: Project Leyden

Project Leyden stellt die natürliche Weiterentwicklung der bereits erwähnten Optimierungstechniken dar. Doch statt eine Technologie der fernen Zukunft zu sein, wird es bereits aktiv entwickelt, um die AppCDS-Funktionen innerhalb von OpenJDK noch zu übertreffen. Aufbauen auf AppCDS zielt Leyden darauf ab, einen einheitlichen Cache zu schaffen, der nicht nur geladene Klassen speichert, sondern auch kompilierten Code, Methodenprofile und verknüpfte Klassen. Leyden wird auf allen unterstützten Java-Plattformen arbeiten und nach einem initialen Trainingslauf einen zustandslosen Betrieb ermöglichen. Dieser umfassende Ansatz verspricht, die Startvorteile der AOT-Kompilierung mit der Laufzeit-Performance einer JVM im Normalbetrieb zu kombinieren, und das ohne die Trade-offs aktueller Ansätze.

Die richtige Startoptimierung wählen

Jeder Lösungsansatz zur Startzeitoptimierung erfordert andere Kompromisse. Für die meisten Anwendungen ist ein schrittweises Implementierungsmodell am besten geeignet:

- Beginnen Sie mit Client VM und/oder CDS, um sofortige Verbesserungen bei minimalem Risiko zu erzielen.
- Ziehen Sie für Anwendungen mit besonders hohen Startzeitanforderungen Native Image oder CRaC in Betracht.

Durch die Wahl des richtigen Optimierungsansatzes können Unternehmen ihren Ressourcenverbrauch drastisch verringern, was eine effizientere Skalierung und deutlich reduzierte Cloud-Kosten mit sich bringt.

Garbage Collection: Die Wahl des richtigen Collectors ist entscheidend

Garbage Collection (GC) ist ein automatisierter Prozess, durch den ungenutzter Speicher in Java-Anwendungen freigegeben wird. So entfällt die Notwendigkeit für ein manuelles Speichermanagement. Allerdings kann eine falsche GC-Konfiguration zu häufigen und/oder langen Pausen führen, welche die Reaktionsfähigkeit und den Durchsatz der Anwendung beeinträchtigen.

Garbage Collection galt lange als der komplexeste Aspekt beim JVM-Tuning. Teams haben Wochen damit zugebracht, GC-Parameter zu optimieren, was oft nur in marginalen Verbesserungen resultierte. Moderne JVMs bieten jedoch einen einfacheren und effektiveren Ansatz: die Wahl des richtigen Garbage Collectors auf Grundlage Ihrer spezifischen Anwendungsanforderungen.

Anwendungstyp	Collector-Wechsel	Typische Verbesserung
Latenzempfindliche API	G1 → ZGC	90–99 % Verringerung der max. Pausenzeiten
Batch-Verarbeitung	G1 → Parallel	10–15 % Durchsatzverbesserung
Begrenzter Speicher	G1 → Serial	5–10 % Verringerung des Speicherverbrauchs

Die durch die richtige Wahl des Collectors erzielten Performance-Steigerungen können erheblich sein. Noch beeindruckender ist, dass diese Verbesserungen nur eine einzige Konfigurationsänderung erfordern – im Gegensatz zum traditionellen GC-Tuning, das oft Dutzende von Parametern umfasst. Daher besteht der einfachste und effektivste Quick Win bei der GC-Optimierung schlicht darin, den richtigen Garbage Collector für die eigenen Performance-Ziele auszuwählen.

- **Für Low-Pause-Anwendungen:** ZGC und Shenandoah. Diese beiden GCs liefern unabhängig von der Heap-Größe (selbst bei über 100 GB) Pausen von unter einer Millisekunde bei vorhersehbarer Performance unter Speicherdruck, wenn auch auf Kosten eines geringen Durchsatzverlustes.

- **Für maximalen Durchsatz:** Parallel GC. Parallel GC bietet 10–15 % höheren Durchsatz für Batch-Verarbeitungs- und CPU-intensive Anwendungen, wobei höhere Pausenzeiten in Kauf genommen werden müssen.

- **Für Ausgewogenheit:** G1 GC. G1 bietet eine gute Balance für allgemeine Anwendungen mit konfigurierbaren Pausenzielen und angemessenem Durchsatz.

- **Für Umgebungen mit begrenzten Ressourcen:** Serial GC. Serial GC verringert den Speicher-Overhead und die CPU-Konkurrenz in kleinen Containern, wodurch er in Umgebungen mit begrenzten Ressourcen besonders effizient ist.

- **Die nächste Generation:** Generational ZGC und Generational Shendandoah. Diese beiden neuen GC-Versionen behalten die ultra-niedrigen Pausenzeiten bei, reduzieren den Speicherverbrauch um 10–25 % und verbessern die Handhabung kurzlebiger Objekte – eine Lösung ohne Kompromisse.

Die Zukunft: Project Lilliput

Projekt Lilliput stellt jenseits der Collector-Wahl einen weiteren kommenden Quick Win für die Speichereffizienz dar und verringert die Größe der Java-Objekt-Header:

```
# Aktivieren von Lilliput für eine verringerte Objekt-Header-Größe
(JDK 24+ experimentell)

ENTRYPOINT [„java“, „-XX:+UnlockExperimentalVMOptions“,
„-XX:+UseCompactObjectHeaders“, „-jar“, „app.jar“]
```

Lilliput kann den Heap-Speicherbedarf in objektintensiven Anwendungen ohne Code-Änderungen um 10–20 % verringern. Das Projekt zielt darauf ab, ein seit langem bestehendes Speicherproblem von Java zu beheben, bei dem der Objekt-Header unverhältnismäßig viel Heap-Speicher verbraucht.

GC-Tuning in der Praxis

Der moderne Ansatz zur GC-Optimierung folgt einem schrittweisen Vorgehen von einfachen zu fortgeschrittenen Techniken:

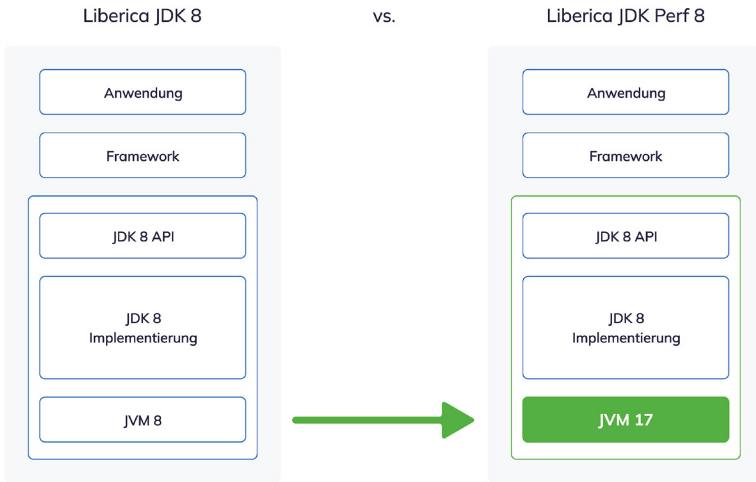
1. Beginnen Sie mit der Auswahl eines Collectors, basierend auf Ihrem primären Performance-Ziel.
2. Fügen Sie eine geeignete Speicherkonfiguration hinzu, die auf Ihre Container-Umgebung abgestimmt ist.
3. Ziehen Sie zur weiteren Optimierung experimentelle Features wie Lilliput in Betracht.
4. Erwägen Sie Parameter-Feinjustierung nur, wenn es absolut notwendig ist.

Dieser Ansatz stellt eine drastische Vereinfachung gegenüber dem traditionellen CG-Tuning dar und bietet deutlich höhere Renditen. Die Zeiten, in denen ein GC-Experte notwendig war, um eine exzellente Java-Performance zu erzielen, sind praktisch vorbei.

Legacy-Anwendungen: Es gibt noch Hoffnung

Die bisher vorgestellten Optimierungstechniken entfalten ihr volles Potenzial auf modernen JVMs – also hauptsächlich ab Java 17. In der Realität der meisten Unternehmen stellt dies allerdings eine Herausforderung dar, da ihre Produktion-Workloads immer noch auf älteren Versionen laufen.

Laut verschiedenen Quellen (z. B. „Status quo des Java-Ökosystems 2024“ von New Relic oder „The State of Developer Ecosystem 2023“ von JetBrains) verwenden 29–50 % der Befragten noch Java 8, während 32–38 % auf Java 11 setzen. In der Java Developer Survey von BellSoft gaben zwei Drittel der Befragten an, ihre Anwendungen noch auf Java 11 oder früheren Versionen auszuführen.



Diese Statistiken spiegeln eine gängige Realität in Unternehmen wider: Die Migration von Legacy-Anwendungen auf neuere Java-Versionen stellt häufig erhebliche technische und geschäftliche Herausforderungen dar. Das Resultat besteht aus Performance- und Effizienzeinbußen, da Unternehmen, die ältere Java-Versionen verwenden, wichtige Optimierungsmöglichkeiten verpassen, die in neueren JDKs zur Verfügung gestellt werden.

Fused JDKs: Moderne Performance für Legacy-Anwendungen

Glücklicherweise gibt es für Java-Legacy-Anwendungen einen leistungsstarken Quick Win: Fused JDKs. Diese spezialisierten Distributionen kombinieren die API-Kompatibilität älterer Java-Versionen mit den Performance-Verbesserungen moderner oder maßgeschneiderter JVM-Implementierungen.

Liberica JDK Performance Edition stellt ein Open-Source-Beispiel für diesen Ansatz dar. Die Edition bewahrt eine klare Trennung zwischen der Java-API (mit der Ihr Anwendungsquellcode interagiert) und der JVM-Implementierung (der Laufzeitumgebung, die Ihren Code ausführt):

- **API-Schicht:** Bleibt 100 % kompatibel mit der Ziel-Java-Version (8 oder 11).

- **JVM-Schicht:** Integriert Optimierungen aus neueren JVM-Versionen.

Diese Architektur ermöglicht es Anwendungen, von modernen JVM-Optimierungen zu profitieren, ohne dass Code-Änderungen erforderlich sind oder Kompatibilitätsrisiken bestehen. Ihre Anwendung nutzt weiterhin die APIs von Java 8 oder Java 11, für die sie entwickelt wurde, nutzt dabei aber die modernen Performance-Verbesserungen moderner JVMs:

1. **Moderne Garbage Collection:** Legacy-Anwendungen können auf ZGC und andere Low-Pause-Collector zurückgreifen, die eine Verringerung der Pausenzeiten um bis zu 90 % ermöglichen – ganz ohne Änderungen an Ihrer Anwendung.
2. **Laufzeitoptimierung:** Anwendung auf Grundlage von Java 8 oder 11 profitieren von JIT-Compiler-Verbesserungen, optimierter String-Verarbeitung und erweiterten Intrinsics, was zu 10–30 % höherem Durchsatz bei vielen Workloads führt.

3. **Speichereffizienz:** Speicheroptimierungen aus neuen JVMs verringern den Speicherbedarf und verbessern die Effizienz der Garbage Collection bei Legacy-Anwendungen.
4. **Container-Bewusstsein:** Legacy-Anwendungen profitieren von einem container-bewussten Ressourcenmanagement, das in den Java-Versionen 8 und 11 noch nicht verfügbar war.

Performance-Tests mit Java-Standardbenchmarks zeigen klare Verbesserungen:

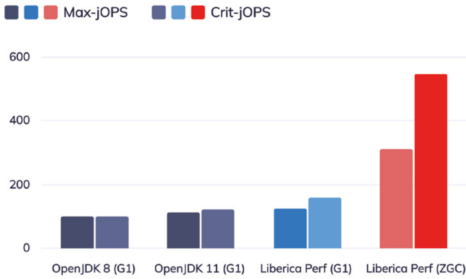
- Spring Boot 2.7.x-Anwendungen auf Java 8: 15–20 % Durchsatzverbesserung
- Java 8-Webanwendungen: bis zu 70 % Verringerung der GC-Pausenzeiten
- Java 11 Batch-Verarbeitung: 10–15 % schnellere Ausführungszeit

Benchmark-Ergebnisse

im Vergleich zu OpenJDK 8

- G1 Peak-Durchsatz **+25%**
- G1 Durchschnittsdurchsatz **+50%**
- Z Peak-Durchsatz bis zu **+200%**

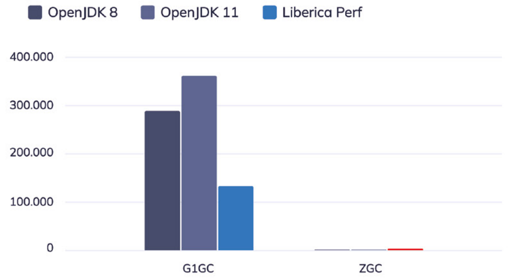
Durchsatz – SPECJBB (%), mehr = besser



im Vergleich zu OpenJDK 8

- G1-Latenz **54 %** besser
- ZGC-Latenz **>1000 %** besser

BigRamTester, GC-Latenz*, ms, weniger = besser



* Hardware: Intel Xeon 3GHz 24 Core, Ubuntu 22.04, BigRamTester, 1 Std. Laufzeit mit 50 GB.

Noch wichtiger: Diese Verbesserungen erfordern keine Änderungen am Anwendungsquellcode, keine komplexen Migrationsprojekte und nur minimale Betriebsänderungen – der einfache Wechsel zu einer Fused-JDK-Distribution liefert also sofortige Vorteile.

Bei der Auswahl eines Anbieters für ein Fused JDK ist es wichtig, die Distribution mit Ihrer eigenen Anwendung zu testen. Verschiedene Builds können unterschiedliche Performance-Verbesserungen bei unterschiedlichen Workloads bieten. Ein weiterer wichtiger Aspekt ist der Vendor Lock-in. Viele Fused JDKs verwenden maßgeschneiderte JVMs. Dies kann zu Hindernissen führen, wenn Sie zum betreffenden JDK wechseln oder später auf neuere Versionen des JDK upgraden möchten. Berücksichtigen Sie dies bei Ihrer Entscheidung – wir empfehlen, sich an Open-Source-Lösungen zu halten, die Ihnen die Flexibilität gewähren, Ihren Stack in Zukunft anzupassen.

Alles in allem können Unternehmen durch den Einsatz von Fused JDKs die Lebensdauer von Java-Legacy-Anwendungen verlängern und gleichzeitig Performance und Ressourceneffizienz signifikant steigern – ein „legales Aufputschmittel“ für einen bedeutenden Teil der Java-Unternehmenswelt, der weiterhin auf älteren JDK-Versionen läuft.

Buildpacks: Die ultimative Alternative zu Dockerfiles

Das manuelle Optimieren von Dockerfiles und JVM-Einstellung bringt schon an und für sich erhebliche Vorteile. Buildpacks bieten allerdings

noch weitaus mehr – und das bei deutlich geringerem Aufwand. Diese Technologie eliminiert die Notwendigkeit, vollständig zu schreiben, indem sie Ihren Anwendungsquellcode direkt in optimierte Container-Images umwandelt.

Buildpacks vereinfachen den Prozess des Erstellens containerisierter Anwendungen und integrieren parallel fortschrittliche JVM-Anpassungen und -Optimierungen. Sie bieten sofort einsatzbereite Unterstützung für fortgeschrittene Features wie AppCDS und Native Image, die andernfalls umfangreiche Expertise erfordern würden.

Statt komplexe Dockerfiles zu schreiben und zu warten, können Sie das Build-System Ihres Projekts (Maven oder Gradle) verwenden und mit einem einzigen Befehl ein Container-Image erstellen:

Ein einziger Befehl ersetzt Ihr gesamtes Dockerfile

```
mvn spring-boot:build-image
```

oder

```
gradle bootBuildImage
```

Dieser einfache Befehl löst einen fortschrittlichen Workflow aus:

1. **Erkennungsphase:** Das Buildpack analysiert zur Identifizierung des Anwendungstyps Ihre Codebase. Es ermittelt die Anforderungen und stellt alle Ressourcen bereit, die für den Betrieb der Anwendung erforderlich sind.

2. **Build-Phase:** Das Buildpack erfüllt seine Aufgabe, indem es ein optimiertes Container-Image erstellt.

Performance-Vorteile ganz ohne Konfiguration

In den meisten Fällen funktionieren Buildpacks ganz ohne Konfiguration. Sie verwenden standardmäßig die neueste Version optimierter Basis-Images und bringen automatische Performance-Verbesserungen mit sich. Für Spring-Boot-Anwendungen erstellen die Buildpacks von Paketo automatisch geschichtete JARs mit optimierten Strukturen, die – wie bereits in diesem Artikel erläutert – einen schnelleren Start und einen

geringeren Speicherverbrauch ermöglichen.

Buildpacks bieten mehrere Quick Fixes, mit denen Sie problemlos sofortige Performance-Verbesserungen erzielen können:

1. **Automatisierte Ressourcenberechnung:** Eine der größten Vorteile von Buildpacks besteht in der Fähigkeit, Ressourcen basierend auf den tatsächlichen Anwendungsanforderungen korrekt zu berechnen.
2. **Schnelles Patchen von OS-Sicherheitslücken:** Buildpacks integrieren sich nahtlos in moderne CI/CD-Systeme. Somit ist ein schneller Rebuild Ihrer Anwendungen möglich, ohne die Build-Tools umfassend anzupassen. Mit Buildpacks in Ihrem CI können Sie die Betriebssystemschicht Ihrer Anwendungs-Images ohne Rebuild Ihres Quellcodes patchen.
3. **Standardisierte Optimierungsansätze:** Buildpacks erzwingen eine konsistente Optimierung über Ihr gesamtes Anwendungsportfolio hinweg. Alle Anwendungen erhalten das gleiche Maß an Optimierung, sodass sich Entwickler auf den Code statt auf Container-Konfigurationen konzentrieren können. Neue JVM-Optimierungen werden von Build-Paketen automatisch integriert, sobald sie verfügbar sind.

Fortgeschrittenes Performance-Tuning leicht gemacht

Die wahre Stärke von Buildpacks zeigt sich, wenn Sie Ihre Performance auf das nächste Level heben möchten. Statt sich mit der Erlernung Dutzender JVM-Optimierungs-Flags abzumühen, können Sie die Funktionen von Buildpacks durch einfache Konfiguration nutzen. Hier sind einige Beispiele:

1/Optimierung für einen schnellen Start:

```
# AppCDS für schnellere Starts aktivieren
BP_JVM_CDS_ENABLED=true
```

2/Umstieg auf Native Image für eine Near-Zero-Startzeit:

```
# Für Spring Boot 3.x-Anwendungen
./mvnw -Pnative spring-boot:build-image
```

Während Dockerfiles nach wie vor den gängigsten Containerisierungsansatz darstellen, bietet sich mit Buildpacks eine fortschrittliche und Java-bewusste Lösung. Sie vereinen jahrelange JVM-Optimierungsexpertise in einem benutzerfreundlichen Tool, das konstant bessere Ergebnisse als manuell erstellte Container liefert.

Für die meisten Java-Anwendungen bringt der Wechsel von Dockerfiles zu Buildpacks sofortige Performance-Verbesserungen bei geringem Aufwand. Somit handelt es sich um den vielleicht einfachsten, aber effektivsten Quick Win für Teams.

Aufbau einer nachhaltigen Optimierungsstrategie

In diesem Artikel haben wir leistungsstarke Quick Wins zur Verbesserung der Java-Anwendungs-Performance ohne Code-Änderungen untersucht. Diese Optimierungen schaffen einen positiven Kreislauf: Eine verbesserte Performance führt zu Ressourceneinsparungen, die wiederum weitere Optimierungen und letztlich eine Modernisierung ermöglichen können.

Die Wahl des perfekten Optimierungsansatzes hängt von Ihrer aktuellen Java-Version ab:

JDK-Version	Erste Ebene	Zweite Ebene	Dritte Ebene	Migrationspfad
JDK 8	Fused JDKs (Liberica JDK Performance Edition) oder Liberica JDK Lite	Container-Optimierung		Test mit Java 11
JDK 11	Fused JDKs (Liberica JDK Performance Edition) oder Liberica JDK Lite	Container-Optimierung	AppCDS	Validierung mit Java 17
JDK 17+	Container-Optimierung, Generational ZGC / Shenandoah	CRaC, GraalVM oder AppCDS	Project Lilliput, Buildpacks, Virtuelle Threads (JDK 21+)	Informieren Sie sich über Updates.

Optimierung und ihre Alternativen

Bei der Bewältigung von Performance-Problemen ziehen Unternehmen in der Regel drei Ansätze in Betracht:

- **Einfach mehr Ressourcen:** Schneller Ansatz, der allerdings zu nicht nachhaltigem Kostenzuwachs führt und das Problem nicht an der Wurzel angeht.
- **Komplettes Neuschreiben:** Beseitigt technische Defizite, ist jedoch mit hohen Kosten, Risiken und Zeitaufwand verbunden.
- **Strategische Optimierung:** Liefert sofortige Vorteile bei geringem Aufwand und Risiko; schafft Raum für geplante Modernisierungen.

Der nachhaltige Weg nach vorn

Die effektivste und nachhaltigste Java-Optimierungsstrategie folgt den folgenden Prinzipien:

1. **Bleiben Sie soweit möglich auf dem neuesten Stand.** Neue JDK-Versionen bringen standardmäßig Performance-Verbesserungen mit sich.
2. **Erst optimieren, dann skalieren.** Beheben Sie Effizienzprobleme, bevor Sie mehr Ressourcen hinzufügen.
3. **Optimierung automatisieren.** Nutzen Sie Buildpacks, um Performance-Expertise zu demokratisieren.
4. **Alles wird gemessen.** Treffen Sie Ihre Entscheidung auf Grundlage der Datenlage, nicht auf Grundlage von Annahmen.
5. **Progressive Verbesserungen.** Betrachten Sie die Optimierung als einen kontinuierlichen Prozess.

Durch die Anwendung dieser Quick Wins können selbst Java-Legacy-Anwendungen dramatische Performance-Verbesserungen erzielen – und das ganz ohne Risiko oder die Kosten einer Neuschreibung. Dieser nachhaltige Ansatz wägt die sofortigen Performance-Anforderungen gegen eine langfristig funktionale Architektur ab und transformiert ihre Anwendung durch Evolution statt Revolution.

> Zurück zum Inhaltsverzeichnis

Code the Future at JCON USA 2025 @ IBM TechXchange!

Join the ultimate Java event, October 6-9 in Orlando, Florida!

- Learn from top experts
- Dive into hands-on workshops
- Network with developers from around the world

Mark Stoodley
Chief Architect for IBM Java

Edward Burns
Principal Architect for Java on Azure

Register Now and Save 30%

#JAVAPRO #ARCHITECTURE #MICROSERVICES

Überwinde 20 Jahre mit Migration Engineering

Autor:

Merlin Bögershausen ist Softwareentwickler, Architekt und Oracle ACE mit über 10 Jahren Erfahrung in verschiedenen Bereichen und Programmiersprachen. Sein Hauptfokus liegt auf Java-Enterprise-Anwendungen mit modernem und Next-Generation-Java. Als Migrationsingenieur unterstützt er Teams und Einzelpersonen dabei, neue Funktionen zu nutzen und sie bei der Migration zu begleiten. Neben der Entwicklung, dem Sprechen auf Konferenzen und seiner Elternrolle versucht er, an Community-Events teilzunehmen, Menschen das Gleitschirmfliegen beizubringen (ja, ich bin Fluglehrer!) und Volleyball zu spielen.

https://www.linkedin.com/in/merlin-boegershausen/

Seit drei Jahrzehnten heißt es im Java Umfeld Write once, run Everywhere. Am Kern dieses Slogans hat sich bis heute nichts geändert. 30 Jahre alte Java-Programme sind noch heute lauffähig. Die Art und Weise Programme zu schreiben, hat sich in den letzten 30 Jahren extrem verändert. Die Sprache hat sich weiterentwickelt, die verwendeten Frameworks haben sich gewandelt und der Stil zu programmieren, hat sich grundlegend verändert. All diese Veränderungen führen

dazu, dass ein Update von 20 Jahre altem Source Code auf modernes Java äußerst komplex geworden ist – so komplex, dass viele Unternehmen trotz offensichtlicher Nachteile jahrelang zögern und lieber einen massiven Berg technischer Schulden in Kauf nehmen.

Diese technischen Schulden verursachen nicht nur erhebliche Wartungskosten, sondern stellen auch ein Risiko für den ökonomischen Erfolg eines Unternehmens dar. Die größte Gefahr ist die Software nicht zu aktualisieren und auf das Schließen von Sicherheitslücken und die Performancegewinne durch neue Versionen zu verzichten.

In diesem Artikel möchte ich Sie ermutigen und Ihnen die Angst vor Migration nehmen. Ich zeige Ihnen das Werkzeug Migration Engineering, mit dem es möglich ist, 20 Jahre gealterten Sourcecode in einen technisch neuen Zustand zu bringen. Wir werden uns dafür anschauen, was Migration Engineering ist und wie wir die Werkzeuge des Migration-Engineerings nutzen können, um den Source Code zu analysieren und anschließend zu migrieren. Ihre Projekte werden besondere Anforderungen haben und sie werden sehen, wie die vorhandenen Rezepte um die passenden Änderungen erweitert werden. Zum Abschluss werfen wir einen Blick darauf, wie Sie der Migrationswelle stets einen Schritt voraus bleiben – mit kontinuierlich aktuellem Sourcecode und modernen Frameworks. So fördern Sie nicht nur die Freude Ihrer Entwickler:innen an der Arbeit, sondern steigern auch nachhaltig die Produktivität Ihrer Softwareentwicklung.

Was ist Migration Engineering und brauche ich das?

Migration Engineering beschäftigt sich mit einer zentralen Frage: *Wie gelangt man von einer Softwareversion zur nächsten?*
Vergleicht man diesen Ansatz mit klassischen Ingenieursdisziplinen wie Elektrotechnik, Maschinenbau oder – meinem persönlichen Favoriten – der Luft- und Raumfahrt, wird schnell klar: Migration Engineering ist im Kern nichts anderes als klassisches Engineering – nämlich die gezielte Weiterentwicklung eines Produkts. In der Praxis bedeutet das häufig, bestimmte Komponenten auszutauschen, um das System in einen besseren Zustand zu überführen.

Doch zieht man den Vergleich mit traditionellen Ingenieursdisziplinen weiter, zeigt sich schnell: Der Vergleich hinkt ein wenig. Denn der große Unterschied zwischen einem Luftfahrzeug und unserem Source Code ist: Source Code ist veränderbar. Ein Luftfahrzeug ist nur in geringem Maße veränderbar. Wenn wir eine Migration von einem Testframework in unserem Testcode betrachten, dann ändert sich in der Differenzbetrachtung die komplette Source Code-Datei (siehe Screenshot), welche diesen Test definiert.

```
@Test(expected = NoSuchBeanDefinitionException.class)
@Test
public void testLocalSchedulerEnabled() {
        assertFalse(context.getEnvironment().containsProperty("kubernetes_service_host"));
        assertFalse(CloudPlatform.CLOUD_FOUNDRY.isActive(context.getEnvironment()));
        context.getBean(Scheduler.class);
        assertThrows(NoSuchBeanDefinitionException.class, () -> {
                assertFalse(context.getEnvironment().containsProperty("kubernetes_service_host"));
                assertFalse(CloudPlatform.CLOUD_FOUNDRY.isActive(context.getEnvironment()));
                context.getBean(Scheduler.class);
        });
}
```

Eine Softwareingenieur:in behauptet, dies ist weiterhin derselbe Test. Das ist ungefähr so als ob man das Triebwerk eines Jumbos von der Mitte des Flügels an die Enden des Flügels verlegt und behauptet, es ist derselbe Jumbo. Auf diesem Level von Wahnsinn und Komplexität befinden sich die Softwareentwicklungen tagtäglich. Wir verändern integrale, fundamentale Bestandteile und behaupten, es ist die exakt gleiche Software wie vorher. Diese Komplexität zu beherrschen, damit beschäftigt sich Migration Engineering. Migration Engineering funktioniert nur mit einer hohen Testabdeckung, denn ansonsten kann nicht sichergestellt werden, dass die Software sich nach der Migration auch wirklich noch so verhält wie vorher.

Die Arbeit eines Migration-Engineers möchte ich kurz anhand meiner ersten Wochen bei Moderne Inc. beschreiben. Ein Tag begann mit einer Nachricht in „Spring Boot 3.4 ist verfügbar, lasst uns auf aktualisieren!". Die Migrationen von Spring Boot Versionen sind in der Regel durch den veröffentlichten Migration Guide einfach nachvollziehbar. Doch der Migration Guide besteht aus groben Schritten. Verwende folgende neue Klasse und Abhängigkeit. Dieser grobe Schritt muss von einem Migration-Engineer erst einmal weiter zerlegt werden. Bis die atomaren Codeänderungen bekannt sind, welche ein Softwareentwickler:inn manuell durchführt.

- Füge eine Dependency in die Maven Pom hinzu
- Füge einen neuen Import hinzu
- Verwende die neue Klasse anstelle der alten
- Tausche die Reihenfolge der Aufrufparameter
- Lösche den alten Import
- Lösche die alte Dependency

Nachdem diese einzelnen Schritte vorhanden sind, werden diese Schritte aggregiert und in die korrekte Reihenfolge gebracht, bis sie die Zielmigration ergeben. Nun kann sie auf alle vorhandenen Projekte angewendet werden. Das hieß bei uns ein Pull Request für jeden Service, den wir betreiben. Insgesamt bedeutete das einen erheblichen Aufwand – bedingt durch die Vielzahl mittelgroßer Pull Requests, die parallel bearbeitet werden mussten. Und natürlich gab es Rückfragen und Änderungswünsche. Einzelne Services mussten erneut migriert werden. Damit musste eine Migration der Migration stattfinden. Es mussten neue Rezepte erstellt und auf die bereits vorhandenen Änderungen angewandt werden.

Die Komponenten sind inzwischen aktualisiert, und ich möchte eine grobe Einschätzung dazu geben, welcher Aufwand das Ganze war. Ich habe Anfang des Jahres 2025 beim Moderne Inc. begonnen, habe grob eine Woche die Migration implementiert. Ungefähr 10 Minuten lang die Migration angewendet und dann mit verschiedenen Personen grob eine Woche daran gearbeitet, die Merge Requests in den Source Code zu übernehmen. Wäre diese Migration von verschiedenen Teams per Hand gemacht worden, wären divergente Anwendungen der neuen Komponenten und eine längere Umsetzungszeit die Konsequenz gewesen.

Offen bleibt die Frage: Braucht jetzt jedes Projekt einen eigenen Migration Engineer, der sich darüber Gedanken macht, wie die Softwarekomponenten von einer Version zur nächsten überführt werden können? – Ich denke nicht, halte es aber für wichtig, dass jede Softwareentwickler:in sich mit der Thematik beschäftigt und Migrationspfade für ihre Komponenten aufzeigt. Gehört ihr Team zu einem Plattformteam und beeinflussen Entscheidungen andere Teams oder wird eine Bibliothek zur weiteren

Verwendung bereitgestellt? Dann ist es auf jeden Fall die Aufgabe des Teams, nicht nur zu zeigen, was die Änderungen sind, sondern auch, wie diese Änderungen im Einzelnen durchgeführt werden können. Nur so können Ihre Teams dafür sorgen, dass die verwendeten Stile und Versionen kohärent sind. OpenRewrite stellt hierfür die Technologie dar, um Änderungen zu formulieren und skalierbar anzuwenden.

OpenRewrite zum Migrieren von Source Code

Die Firma Moderne Inc. kümmert sich um die Weiterentwicklung und die Community von OpenRewrite. In dieser Zusammenarbeit sind viele grundlegende Bausteine entstanden, welche zu wertstiftenden Migrationen kombiniert werden. Darunter Migrationen von Majorversionen für Spring Boot, Quarkus, Hibernate, Maven und Gradle, aber auch kleine Helfer wie Anwenden von Best Practices oder Beheben der gängigsten Issues von statischer Codeanalyse.

Alle anwendbaren Rezepte sind in dem Recipe Catalog in der OpenRewrite Dokumentation zu finden. Um das passende Rezept für den eigenen Anwendungsfall zu finden, kann die Suche oben links (Kastenkürzel: Strg/CMD+K) verwendet werden. Diese Suche indiziert nicht nur die Namen, sondern auch Teile der Dokumentation einer Migration, um besser zugeschnittene Ergebnisse zu ermöglichen. Noch bessere Ergebnisse liefert die Suche in der Moderne SaaS unter https://app.moderne.io, diese liefert auch passende Ergebnisse bei ungenauen Anfragen. Alternativ sind die Migrationen auch entsprechend ihren Themengebieten getagt und anhand dieser Tags gruppiert. Diese Gruppen ermöglichen eine explorative Suche, die besonders bei den ersten Anwendungen hilfreich sind.

Ein Team könnte so auf die Migration Migrate to Java 21 aufmerksam werden. Die zugehörige Dokumentationsseite ist automatisch erzeugt und für alle Migrationen gleich aufgebaut, siehe auch Abbildung oben. Unter der Überschrift ist der Fully Qualified Name angegeben, dieser ist eindeutig und wird für die weitere Verwendung benötigt. Weiterhin sind die Tags, ein Link zu den Ressourcen sowie die Lizenzinformationen gegeben. Bei den Lizenzen gibt es hauptsächlich drei verschiedene Arten:

- **Apache License 2.0** das Core-Framework ist Apache-lizenziert, damit Framework-Autoren Migrationen für ihre Kunden anbieten können. Einige tun dies, wie Micronaut und Quarkus. In den Fällen, in denen die Framework-Autoren solche Migrationen nicht bereitstellen, gibt es einen Marktplatz für Migrationen von Drittanbietern, darunter auch die von Moderne.
- **Moderne Source Available License** sind Rezepte, die frei verfügbar sind, aber nicht als Teil eines anderen Service bereitgestellt werden dürfen. In der Vergangenheit haben große Unternehmen OpenRewrite ausgenutzt, um ihre kommerziellen AI-Migration-Lösungen zuverlässiger zu gestalten. Um Produkte wie Amazon Q, GitHub Copilot und IBM Watson zu einer Kooperation zu bewegen, veröffentlicht Moderne seine Rezepte nun unter der MSAL. So ist es Projekten weiterhin möglich, die Rezepte in vollem Umfang auf sich selbst anzuwenden, jedoch verboten, sie kommerziell weiter zu verwerten.
- **Moderne Proprietary** sind Rezepte, die nur mit einer Lizenz von Moderne verwendet werden dürfen. Es handelt sich hier um besonders wertschöpfende Rezepte, die mit großem Investment durch Moderne in Verbindung stehen. Auf Open-Source Projekte sind auch diese Rezepte anwendbar.

Die unterschiedlichen Lizenzen dienen dazu, Dritten zu untersagen, die Rezepte in ihren kommerziellen Services zu verwenden und weiter zu vermarkten, ohne dem Projekt etwas zurückzugeben. Eine genaue und aktuelle Aufstellung findet sich in der Moderne Dokumentation. Zum Zeitpunkt der Veröffentlichung suchen Moderne Inc. und die OpenRewrite Community noch nach einer endgültigen Lösung. Ziel ist es weiterhin, Open-Source-Projekten Zugriff auf die Modernisierungsfähigkeiten von OpenRewrite und den modernen Produkten zu ermöglichen.

Nach den rechtlichen Anmerkungen führt die Dokumentation die durchzuführenden Migrationsschritte auf. Jeder Schritt ist ebenfalls eine Migration mit einer gleich aufgebauten Dokumentationsseite. In einem späteren Absatz wird behandelt, wie solche Migrationen für die eigenen Bedürfnisse erstellt und über YAML-Dateien konfiguriert werden können. In der Dokumentation folgt eine Aufzählung der Möglichkeiten, wie diese Migration angewendet werden kann. Migrationen sammeln neben Codeanpassungen auch Daten zur späteren Analyse, die DataTables. Gängige Informationen sind zu Laufzeiten, geänderte Dateien oder Analyseinformationen zur Verwendung von Konstrukten. Nun soll die Migration angewandt werden. Nach der Entscheidung für eine Migration kann der oben erwähnte Absatz Usage verwendet werden, um die Anwendung eines Rezeptes zu studieren. Grundsätzlich gibt es drei Möglichkeiten, eine Migration zur Ausführung zu bringen.
Maven, über die CLI oder das Rewrite Maven Plugin

- **Maven,** über die CLI oder das Rewrite Maven Plugin
- **Gradle,** Rewrite Plugin oder ein Init Script
- **Moderne CLI**, das standalone Power-User-Tool
- **Moderne SaaS**, auf OpenRewrite Webanwendung

Über die Maven CLI kann eine Liste von OpenRewrite Rezepten angegeben werden, welche mit dem Rewrite Maven Plugin ausgeführt werden. Dieser Modus eignet sich in erster Linie für die Verwendung von einmal Migrationen wie Framework Updates, dasselbe gilt für das Gradle Init Script. Es empfiehlt sich bei dieser Art der Anwendung die Beispiele aus der Dokumentation zu kopieren. Für unsere JUnit Migrate to Java 21 Migration ist der Maven CLI Aufruf im Listing unten gegeben. Es wird das Rewrite Maven Plugin Goal run aufgerufen und mit rewrite.activeRecipes die Migration UpgradeToJava21 angegeben. Da dieses nicht in OpenRewrite integriert ist, müssen die Artefaktkoordinaten für rewrite-migrate-java unter rewrite.recipeArtifactCoordinates angegeben werden.

```
mvn -U org.openrewrite.maven:rewrite-maven-plugin:run
  -Drewrite.recipeArtifactCoordinates=org.openrewrite.recipe:rewrite-
   migrate-java:RELEASE
  -Drewrite.activeRecipes=org.openrewrite.java.migrate.UpgradeTo
   Java21
```

Im Fall der Migrate to Java 21 Migration ergibt es Sinn, das Rezept gelegentlich erneut auszuführen. Denn durch Unachtsamkeit könnte eine Entwickler:inn alte Java API verwenden, welche erneut migriert werden müssen. Hierzu bietet es sich an, das Rezept als optionales Plugin im Build einzusetzen. Dazu wird wie in Listing unten das rewrite-mave-plugin eingebunden, das Rezept UpgradeToJava21 aktiviert und das Migrationsartefakt als Dependency hinzugefügt. Durch ein einfaches mvn rewrite:run kommt das Plugin zur Ausführung, führt die konfigurierten Migrationen aus und wendet die notwendigen Änderungen auf den Code an. Bei jedem Durchlauf der Rewrite Plugins meldet das Plugin, welche Migration in welcher Datei eine Änderung vorgenommen hat. Zusätzlich erstellt es eine grobe Schätzung, wie viel Aufwand eine manuelle Bearbeitung bedeutet hätte. Von hier aus muss nur noch committed, gepusht und gereviewt werden.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.openrewrite.maven</groupId>
        <artifactId>rewrite-maven-plugin</artifactId>
        <version>6.3.1</version>
        <configuration>
          <exportDatatables>true</exportDatatables>
          <activeRecipes>
            <recipe>org.openrewrite.java.migrate.UpgradeToJava21
            </recipe>
          </activeRecipes>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>org.openrewrite.recipe</groupId>
            <artifactId>rewrite-migrate-java</artifactId>
            <version>3.4.0</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
```

```
</project>
```

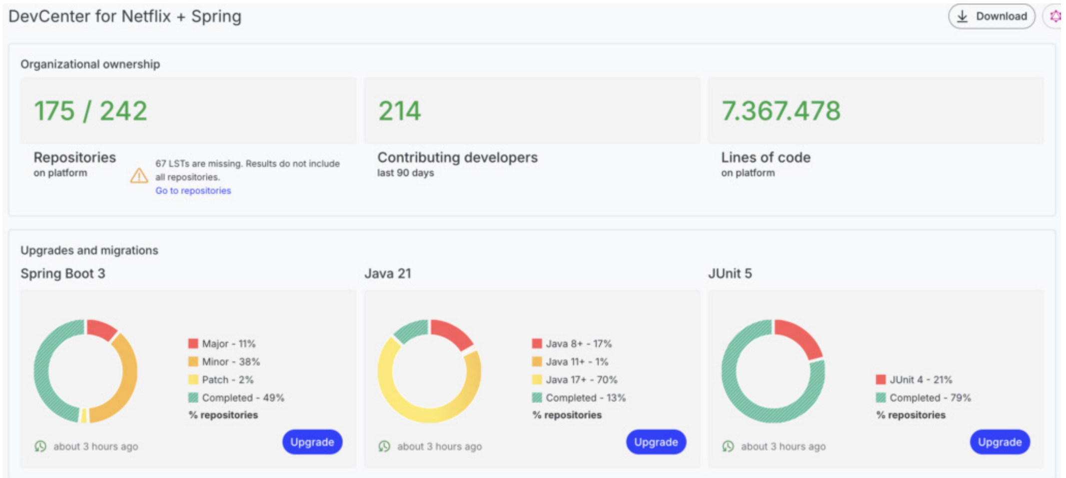
Als letzte Option bleibt noch die Moderne CLI als Powertool basierend auf OpenRewrite. Die Moderne CLI ist kostenlos einsetzbar. Für Projekte in öffentlichen Repositorys, für Closed-Source Projekte ist eine Lizenzierung notwendig. Im Gegensatz zu den Rewrite Plugins kann die Moderne CLI auf mehrere Projekte gleichzeitig und parallel angewendet werden. Weiterhin kann die Moderne CLI die erzeugten Metadaten persistieren und weiterverwenden. Dadurch werden nicht bei jedem Durchlauf die kompletten Metadaten erzeugt und die Laufzeit verbessert sich drastisch. Die Moderne CLI wird als native Applikation installiert, für alle gängigen Betriebssysteme existieren Executables und über Maven Central wird zusätzlich eine reine JAR Variante bereitgestellt. Nach der Installation ist die CLI im Terminal verfügbar.

Vor der Ausführung einer Migration müssen die Metadaten erzeugt werden. Hierzu wird das Kommando mod build . genutzt. Dieses sucht im aktuellen Verzeichnis nach Projekten und erzeugt die Metadaten. Zusätzlich muss mit mod sync die Liste der verfügbaren Migrationen heruntergeladen werden. Um ein Rezept auszuführen, wird mit dem mod run. Kommando und dem parameter —recipe die auszuführende Migration gestartet. Nach der Migration könnten die Änderungen angewandt oder studiert werden.

Grundsätzlich wird der Prozess verständlich durch die CLI geleitet. Hierzu meldet sie nach jedem Durchlauf, welche nächsten Schritte sinnvoll wären. Eine Schritt-für-Schritt-Anleitung findet sich in der Dokumentation. Jede Ausführung eines Rezepts liefert neben Codeanpassungen auch Datatables. Die Datatables sind die Basis für alle Analysemöglichkeiten mit der OpenRewrite Technologie. Die Maven und Gradle Integration produziert Datatables für ein Projekt. Im Gegensatz liefern die CLI und die Moderne SaaS Daten für mehrere Projekte gleichzeitig.

In der Moderne SaaS sind einige Standardvisualisierungen vorhanden, die es Migration Engineers ermöglichen, die Verwendung von unterschiedlichen Framework-Versionen oder der Verteilung von Änderungen zu analysieren. In dem Screenshot unten ist das DevCenter zu sehen, in welchem für eine beispielhafte Organisation Informationen über die enthaltenen Repositorys enthalten sind. Zusätzlich sind in

überschaubarer Art die Erreichung von strategischen Unternehmenszielen wie Spring Boot 3 oder Java 21 Migrationen gezeigt.



Nicht selten haben Projekte in Unternehmen besondere Anforderungen an die Verwendung von bestimmten Patterns oder internen Bibliotheken. In diesem Fall müssen die vorhandenen OpenRewrite Rezepte erweitert werden.

Angepasste OpenRewrite Rezepte

In der OpenRewrite Technologie werden Migrationen als Rezepte definiert. Ein Rezept instruiert das OpenRewrite Werkzeug, wie der vorhandene Code angepasst wird, um ein Ziel zu erreichen. Rezepte können auf 3 verschiedenen Wegen mit steigender Komplexität und Funktionsumfang definiert werden:

- Deklarative YML Rezepte
- Refaster Templates in Java-Rezepten
- Imperative Java Rezepte

In der Tabelle unten sind die einzelnen Eigenschaften der Rezepte aufgeführt. Den einfachsten Einstieg bieten die deklarativen Rezepte, diese folgen einem YAML-Schema und definieren neben den notwendigen Eigenschaften die Liste der auszuführenden Rezepte mit ihren Konfigurationen. Diese Rezepte werden verwendet, um andere Rezepte zu aggregieren. Refaster Rezepte verwenden Refaster Templates, wie sie das Google Error Prone Projekt definiert. Diese Templates werden für typsicheres Suchen und Ersetzen von Methodenaufrufen verwendet. Werden komplexere Manipulationen benötigt, welche nicht durch die Basisbausteine abgebildet werden können, ist es notwendig,

ein imperatives Rezept zu formulieren. Hier ist der komplette Umfang von Java als Programmiersprache verwendbar und viele Utilitys von OpenRewrite verfügbar, um die Migrationslogik umzusetzen.

Deklaratives Rezept	Refaster Rezept	Imperatives Rezept
YAML	Refaster Templates in Java	Pure Java
Aggregiert Rezepte	Suchen & Ersetzen	Hochflexibel
Konfiguriert Rezepte	Typsicher	Visitor-Pattern
Einfach	Begrenzte Einsetzbarkeit	Viele Utilities

Arten von Rezepten und ihre Eigenschaften

Übersicht über die unterschiedlichen Arten von Rezepten

Es ist empfehlenswert, mit einem deklarativen Rezept zu starten und zu versuchen, einen möglichst großen Umfang der Migration durch die Konfiguration von vorhandenen Rezepten abzubilden. Doch bevor ein Rezept erstellt werden kann, muss ein entsprechendes Projekt aufgesetzt und die Akzeptanztests definiert werden.

Deklarative Rezepte werden verwendet, um vorhandene Rezepte zu konfigurieren und gemeinsam auszuführen. Im ersten Schritt wird im Open Rewrite Rezeptkatalog das benötigte Rezept identifiziert und der vollqualifizierte Name identifiziert. Der vollqualifizierte Name ist direkt unter der Überschrift der Dokumentation angegeben. Sollen Rezepte aus dem eigenen Bestand verwendet werden, so wird auch in diesem Fall der qualifizierende Name verwendet. Um etwa eine Klassenreferenz auszutauschen, bietet sich das Rezept org.openrewrite.java.ChangeType an. Im moderneinc/rewrite-recipe-starter GitHub Projekt wird dieses Rezept in [resources/META-INF/rewrite/stringutils.yml](), siehe auch Listing unten, konfiguriert, um die Korrekten StringUtils Klasse zu verwenden.

```yaml
type: specs.openrewrite.org/v1beta/recipe
name: com.yourorg.UseApacheStringUtils
displayName: Use Apache `StringUtils`
description: Replace Spring string utilities with Apache string
utilities.
recipeList:
  - org.openrewrite.java.ChangeType:
    oldFullyQualifiedTypeName: org.springframework.util.StringUtils
    newFullyQualifiedTypeName: org.apache.commons.lang3.StringUtils
```

Durch den type wird es als Rezept für OpenRewrite Migrationen ausgewiesen und ist über den qualifizierenden Namen hinter name ansprechbar. Der displayName wird vor allem bei dem automatisiert erstellten Rezeptkatalog verwendet. In der recipeList werden die auszuführenden Rezepte angegeben. Die möglichen Konfigurationen sind auf den entsprechenden Rezeptseiten angegeben. Reichen die vorhandenen Basisrezepte im Rezeptkatalog nicht aus, müssen neue Rezepte geschrieben werden.

Ein mögliches Beispiel wäre das Vereinfachen eines ternären Operators zu einem konstanten Ausdruck. Hier bietet sich die Anwendung von Refaster Templates aus dem Error Prone Projekt an. Mit Error Prone Refaster können typischer musterbasierte Veränderungen durchgeführt werden. OpenRewrite unterstützt die Templates von Refaster und bietet somit eine passende Abstraktion für die Anpassung von Methodenaufrufen. Ein Refaster Template Rezept wird durch die @RecipeDesciptor als ein Rezept markiert, mit einem Namen für die Dokumentation und einer Beschreibung versehen. Die auszutauschende Codepassage wird in der @BeforeTemplate markierten Methode angegeben. Hierbei werden in der Methode jeweils die Expressionen typsicher verwendet. Die mit @AfterTemplate markierte Methode definiert den Zielzustand und verwendet die Typsicherheit, um den neuen Ausdruck zu erstellen. Refaster Templates können nur Änderungen innerhalb eines Codeblocks durchführen.

```
@RecipeDescriptor(
  name = „Replace `booleanExpression ? true : false` with
` booleanExpression`“,
  description = „Replace ternary expressions like `
  booleanExpression ? true :  false` with `booleanExpression`.“)
public static class SimplifyTernaryTrueFalse {
    @BeforeTemplate
    boolean before(boolean expr) {
        return expr ? true : false;
    }
    @AfterTemplate
    boolean after(boolean expr) {
        return expr;
    }
}
```

Sind Anpassungen darüber hinaus notwendig und diese nicht durch die Kombination aus vorhandenen Rezepten möglich, kann ein imperatives Rezept erstellt werden. Diese imperativen Rezepte erweitern die abstrakte Klasse org.openrewrite.Recipe, siehe Listing unten.

```
public class TestRecipe extends org.openrewrite.Recipe {
  @Override
  public String getDisplayName() {
    return „An example Recipe“;
  }
  @Override
  public String getDescription() {
    return „This Recipe is an example.“;
  }
  @Override
  public TreeVisitor<?, ExecutionContext> getVisitor() {
    return new JavaIsoVisitor<ExecutionContext>() {
      @Override
      public J.MethodDeclaration visitMethodDeclaration
        (J.MethodDeclaration m, ExecutionContext ctx) {
        J.MethodDeclaration m2 = super.visitMethodDeclaration
      (m, ctx);
        return m2.getName().getSimpleName().equals(„foo“) ?
```

```
                m2.withName(m2.getName().withName(„bar“)) :
                m2;
        }
      };
    }
  }
```

Die Methode getDisplayName liefert den Anzeigenamen und getDescription die Beschreibung für die automatisiert erstellte Dokumentation. Die getVisitor Methode liefert eine Instanz eines TreeVisitors. Ein TreeVisitor traversiert den Lossless Semantic Tree (LST) und manipuliert einzelne Knoten in diesem Baum. Der LST ist die OpenRewrite Repräsentation des kompletten, Sprachen und Technologie überspannenden Source-Codes innerhalb des Projekts. Der LST wird beim Start von Open Rewrite erzeugt und wird nach der Durchführung aller Rezepte verwendet, um die Änderungen wieder in Source Code zu überführen. Für jede unterstützte Sprache gibt es eine eigene LST-Implementierung und angepasste Visitoren.

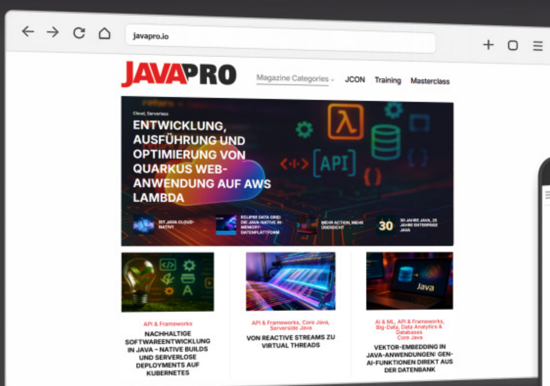
Um Java-Source-Code zu manipulieren, wird beispielsweise der org.openrewrite.java.JavaIsoVisitor verwendet. Dieser enthält eine Visit-Methode für jede Art von Elementen im LST. Diese liefern das geänderte Element zurück. In diesem Beispiel werden alle Methoden Deklarationen besucht und die J.MethodDeclaration gesucht, welche den Namen „foo“ hat, um sie nach „bar“ umzubenennen. Wenn die aktuell betrachtete Methode beibehalten werden soll, wird sie unterwandert zurückgeliefert. Wird null zurückgeliefert, so wird die Methodendefinition gelöscht. Weitere vollständige Implementierung von Rezepten findet sich im moderneinc/rewrite-recipe-starter. Unteranderem in der Klasse NoGuavaListsNewArrayList, dieses Rezept migriert die Verwendung von Guave zu JDK enthaltenen Methoden zum Umgang mit ArrayLists. Ein ausführlicher Schritt-für-Schritt-Workshop für das Homeoffice ist auf Moderne.io vorhanden und behandelt neben den hier angeschnittenen Techniken weitere wichtige Werkzeuge.

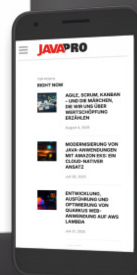
Vor den Migrationen bleiben

Migrationen werden in den kommenden Monaten und Jahren weiterhin auf uns Entwickler:innen einprasseln. Mit der OpenRewrite Technologie und dem hier beschriebenen Ansatz des Migration-Engineerings können Organisationen sich selbst in die Lage versetzen, wieder planvoll zu agieren, anstatt weiterhin nur zu reagieren. Einen guten Startpunkt bieten hierbei die OpenRewrite Step-by-Step-Guides rund um Aktualisierung von Java 5 bis Java 21 oder die Migration von Java EE zu Jakarta EE 10.0. Durch die Anwendung dieser beiden Guides können Teams lernen, wie viel Zeit bei der Migration von 20 Jahren alten Anwendungen einsparbar ist. Das gelernte Wissen kann dann für interne Migrationen und Bibliotheken angewandt werden, um die Organisation für weitere 30 Jahre Java fit zu halten.

> Zurück zum Inhaltsverzeichnis

ENTDECKE WEITERE SPANNENDE ARTIKEL ONLINE:
JAVAPRO





www.javapro.io
www.javapro.io

#JAVAPRO #ARCHITECTURE #MICROSERVICES

Nachhaltige Softwareentwicklung in Java – Native Builds und serverlose Deployments auf Kubernetes

Autor:

Marius Stein ist freiberuflicher IT-Berater und Cloud Engineer. Er verfügt über langjährige Erfahrung in der Beratung von Unternehmen beim Aufbau von Cloud-Plattformen und der Entwicklung cloud-nativer Systeme. Außerdem ist er Mitglied des Programmkomitees des Cloudland Festivals und Mitorganisator der Cloud Native Computing Rheinland Meetup-Gruppe.

https://www.linkedin.com/in/marius-stein-it/

Vishal Shanbhag
Unternehmer | Startup-Mentor | Blogger | Technologieberater | Tech Lead und Programmierer | Trekking-Enthusiast
Über 20 Jahre Erfahrung mit nachweislicher Erfolgsbilanz als Senior Technical Consultant, Softwarearchitekt, Lead Developer und Pre-Sales Consultant.

https://www.linkedin.com/in/vishal-shanbhag-70b679a/

Rechenzentrums-Migration - Im IT-Umfeld, insbesondere in großen Unternehmen, ist der Begriff Rechenzentrums-Migration (Data Center Migration) weit verbreitet. In vielen Fällen liegen bereits praktische Erfahrungen mit der Migration in die Cloud vor. Solche Projekte wurden schon lange realisiert, noch bevor der Begriff Cloud-Rechenzentrum überhaupt geprägt wurde.

Kosteneinsparungen sind für die meisten Entscheidungsträger das Hauptmotiv. Doch es gibt noch einen weiteren entscheidenden Faktor -Nachhaltigkeit. Rechenzentren verbrauchen enorme Mengen an Strom. Im Jahr 2022 wurde ihr weltweiter Energieverbrauch auf 415 TWh geschätzt -nahezu so viel wie der gesamte Stromverbrauch Frankreichs (426 TWh). Und das war noch vor dem durch KI-Tools wie ChatGPT ausgelösten Boom. Der Verbrauch wird in Zukunft voraussichtlich weiter steigen.

Daher sind alle technologischen Ansätze, die den Energieverbrauch von Anwendungen reduzieren können, von großem Interesse. Gleichzeitig tragen solche Maßnahmen zur Senkung der Betriebskosten von Anwendungen bei.

Dieser Artikel untersucht, wie sich drei grundlegende Technologien -Java, Virtualisierung und Containerisierung -weiterentwickelt haben. Wir beginnen mit dem frühen Java und der Virtualisierung und nähern uns modernen Ansätzen wie GraalVM-Native-Builds und serverlosen Anwendungen auf Kubernetes. Wir zeigen, wie eine unternehmensreife Java-Anwendung von einem einfachen Kubernetes-Deployment zu einem nativen ausführbaren Programm als serverloser Container mit Knative überführt werden kann -optimiert für Kaltstarts unter einer Sekunde -und wie dieser Wandel zur Nachhaltigkeit beiträgt.

Geschichte – Die 2000er

Anfang der 2000er wurde Java als Sprache gefeiert, mit der sich portable Anwendungen erstellen ließen. Man konnte einmal Code schreiben und ihn in eine Java-Bytecode-Repräsentation kompilieren. Dieser Bytecode (auch bekannt als .class-Dateien) konnte in ein komprimiertes Dateiformat (.jar) gepackt und überall ausgeführt werden, sofern auf dem Zielsystem eine Java-Laufzeitumgebung (JRE) verfügbar war.

Dies bedeutete, dass Anwendungen unter Windows oder macOS entwickelt und gebaut werden konnten – Betriebssysteme, die bei Endnutzern größere Verbreitung fanden – während dasselbe Paket dennoch auf Linux- oder Unix-Systemen ausgeführt werden konnte. Voraussetzung dafür war lediglich, dass das Zielsystem über eine kompatible JRE verfügte.

Zusammen mit anderen Faktoren wie der Verfügbarkeit zahlreicher Bibliotheken sowie Sprachfunktionen im Bereich objektorientierter Programmierung und Speicherverwaltung konnte Java schnell weite Verbreitung finden. Entwickler konnten nun Datenmodelle in einer an die reale Welt angelehnten Terminologie beschreiben – dank objektorientierter Features. Komplexe Programme ließen sich schreiben, ohne dass man sich um jedes einzelne Byte an Speicher kümmern musste. Solange man sich an bewährte Speicherverwaltungspraktiken hielt, kümmerte sich das System um die Zuweisung und Freigabe von Speicher.

Obwohl Java Plattformunabhängigkeit bot, wurden Unternehmensanwendungen damals häufig auf physischen Servern nach dem „eine Anwendung pro Maschine"-Prinzip betrieben. Die Vorteile dieses Ansatzes liegen in der starken Isolation zwischen Anwendungen auf verschiedenen Servern sowie in der Möglichkeit, unterschiedliche Betriebssystem-Abhängigkeiten in separaten Anwendungen zu verwenden. Die offensichtlichen Nachteile waren mangelnde Skalierbarkeit und Ressourcenverschwendung. Wenn nur eine Anwendung auf einem physischen Server lief, konnten alle verfügbaren Ressourcen ausschließlich von dieser Anwendung genutzt werden. Ressourcen-Sharing war in diesem Szenario nicht einfach möglich.

Virtuelle Maschinen – JRE und Hypervisoren

Aber wie erreichte Java seine Plattformunabhängigkeit? Einfach gesagt: Eine Java-Anwendung interagiert nicht direkt mit dem Betriebssystem. Jede Interaktion mit dem OS wird durch die Java Virtual Machine (JVM) abstrahiert. Für Entwickler fühlt sich die JVM an wie eine echte Maschine. Sie bietet Zugriff auf Systemressourcen wie Dateisystem, CPU und Speicher über die Java-Bibliotheken. Betriebssystemspezifische Kenntnisse sind

nicht nötig, solange man mit den Java-Systembibliotheken umgehen kann.

Java-Programme werden in eine plattformunabhängige, niedrigstufige Repräsentation – den Bytecode – kompiliert. Dieser Bytecode kann nicht direkt auf einem System ausgeführt werden. Stattdessen startet die JRE zur Laufzeit eine JVM, die weiß, wie sie die Bytecode-Instruktionen interpretieren muss.

An dieser Stelle kommt Just-In-Time-Compilation (JIT) ins Spiel. Nicht alle für eine Anwendung benötigten Bibliotheken sind im Bytecode enthalten. Eine Bibliothek wird erst beim ersten Aufruf dynamisch geladen und in den Speicher eingebunden.

Das bedeutet, dass ein Java-Paket eine lockere Zusammenstellung von Bytecode-Komponenten ist, die zur Laufzeit vom JIT-Compiler zusammengeführt und optimiert werden.

In Zeiten, in denen die Beschaffung von Hardware mehrere Monate dauern konnte, war diese Portabilität von Java hilfreich – Anwendungen konnten entwickelt werden, ohne sich auf eine bestimmte Hardware festzulegen. Solange das Rechenzentrum über freie Ressourcen verfügte, konnte die Anwendung dort bereitgestellt werden. Dadurch erlebte Java zwischen 2000 und 2010 eine rasante Verbreitung.

Eine weitere Technologie, die in diesem Zeitraum große Verbreitung fand, waren Hypervisoren. Hypervisoren basieren – genau wie Java – auf dem Prinzip der Virtualisierung, wenden es jedoch nicht auf Anwendungsebene, sondern auf Betriebssystem- und Kernel-Ebene an. Durch Virtualisierung konnte ein physischer Server in mehrere virtuelle Maschinen aufgeteilt werden, die sich CPU- und Speicherressourcen teilen – und so eine effizientere Ressourcennutzung ermöglichen.

Die Virtualisierung funktioniert, indem die Hardware-Schnittstelle eines physischen Servers durch eine Softwarekomponente – den Hypervisor – simuliert wird. Dieser stellt mehreren virtuellen Maschinen eine virtualisierte Hardwareumgebung bereit. Jede virtuelle Maschine besitzt ihren eigenen Kernel zur Interaktion mit dem virtualisierten Betriebssystem. Dadurch werden Prozesse auf verschiedenen virtuellen

Maschinen streng voneinander getrennt, bei gleichzeitig effizienter Ressourcennutzung – mit einem Sicherheits- und Isolationsniveau, das dem von physischen Servern nahekommt. Dennoch bringt Virtualisierung einen gewissen Overhead mit sich, da jede VM ihr eigenes Betriebssystem und ihren eigenen Kernel betreibt, was Ressourcen kostet.

Auch auf Anwendungsebene verursacht Virtualisierung Overhead. Alles, was innerhalb einer JVM läuft, hat:

- einen höheren Ressourcenverbrauch als ein nativ-kompiliertes Programm, da die JVM selbst Speicher benötigt.
- langsamere Startzeiten, da auch die JVM erst initialisiert werden muss.
- bei der ersten Ausführung einer Funktion meist schlechtere Performance als bei der zweiten, wegen der dynamischen JIT-Kompilierung.

Um diese JIT-bedingten Probleme zu umgehen, wurden Bibliotheken wie Enterprise Java Beans oder Spring eingeführt. Sie ermöglichten das Vorladen benötigter Bibliotheken über sogenannte „Beans". Jede häufig genutzte Bibliothek wurde beim Start einmal in den Speicher geladen und war dann jederzeit verfügbar.

Das verbesserte die Ausführungszeiten nach dem Start, hatte aber auch Nachteile. Hauptsächlich führte es zu längeren Startzeiten, da zusätzlich zur JVM nun auch eine „Aufwärmphase" nötig war, in der die Beans geladen wurden. Die Laufzeitperformance war jedoch überlegen, da kein zusätzlicher JIT-Overhead mehr anfiel. Alle zur Laufzeit benötigten Klassen waren bereits geladen.

Container, Serverless und GraalVM Native Images

Im Jahr 2013 popularisierte Docker eine Technologie namens Containerisierung, die seither die Art und Weise, wie Anwendungen entwickelt, ausgeliefert und bereitgestellt werden, revolutioniert hat. Anders als virtuelle Maschinen simulieren Container keine vollständige Hardwareumgebung. Stattdessen teilen sie sich den Kernel des Host-Betriebssystems, was eine leichtgewichtige und effiziente Isolation von Prozessen ermöglicht. Jeder Container enthält die Anwendung samt

all ihrer Abhängigkeiten und Bibliotheken, was sicherstellt, dass sie in verschiedenen Umgebungen – von der Entwicklung bis zur Produktion – konsistent läuft.

Dieser Ansatz macht ein vollständiges Betriebssystem innerhalb jeder Instanz überflüssig, was zu schnelleren Startzeiten und geringerem Ressourcenverbrauch im Vergleich zu klassischen virtuellen Maschinen führt.

Container ermöglichen zudem eine höhere Skalierbarkeit und Flexibilität. Mit Orchestrierungstools wie Kubernetes können Unternehmen Tausende von Containern über Cluster hinweg verwalten und je nach Nachfrage dynamisch Ressourcen skalieren. Container-Technologie ist somit ideal für Microservice-Architekturen oder zellbasierte Architekturen, bei denen Anwendungen in kleinere, unabhängig deploybare Komponenten aufgeteilt werden. Darüber hinaus vereinfacht die Portabilität von Containern Entwicklungs-Workflows und ermöglicht es Entwicklern, sich auf das Schreiben von Code zu konzentrieren, ohne sich um Kompatibilitätsprobleme in unterschiedlichen Umgebungen sorgen zu müssen.

Eine weitere moderne Entwicklung in der Softwaretechnik ist das Aufkommen von Serverless Computing, einem Paradigma, das die Infrastrukturverwaltung abstrahiert und es Entwicklern erlaubt, sich ausschließlich auf das Schreiben und Bereitstellen von Code zu konzentrieren. Obwohl der Begriff „Serverless“ je nach Kontext unterschiedlich verwendet wird, definieren wir ihn hier als Bereitstellungsmodell, das folgende Kriterien erfüllt:

- **Dynamische Skalierung:** Eine Serverless-Anwendung skaliert automatisch entsprechend der Nachfrage und sorgt so für optimale Performance ohne manuelles Eingreifen.
- **Skalierung auf Null:** Bei fehlender Nachfrage fährt sich die Serverless-Anwendung auf null herunter, wodurch keine Kosten für ungenutzte Ressourcen entstehen.
- **Abstraktion der Serververwaltung:** Entwickler müssen sich nicht mehr um die zugrunde liegende Infrastruktur kümmern – also etwa um Provisionierung, Patch-Management oder das horizontale/vertikale Skalieren.

Durch dynamische Skalierung und das Herunterfahren auf Null lässt sich die Ressourcenauslastung signifikant verbessern. Anwendungen mit wenig oder keinem Traffic geben ihre zugewiesenen CPU- und Speicherressourcen frei, die dann für andere Workloads im Cluster zur Verfügung stehen. Bei steigender Last sorgt das Serverless-Modell durch horizontales Hochskalieren für Ausfallsicherheit – ohne dass Ressourcen überdimensioniert bereitgestellt werden müssen.

Während das Serverless-Paradigma ursprünglich durch Public-Cloud-Anbieter wie AWS Lambda, Azure Functions oder Google Cloud Functions geprägt wurde, hat es mittlerweile auch Einzug in On-Premises- und Hybrid-Umgebungen gehalten – durch Open-Source-Lösungen wie Knative oder OpenFaaS, die Serverless-Bereitstellungen auf Kubernetes-Clustern ermöglichen. Das erlaubt es Unternehmen, die bereits stark in Container-Orchestrierung investiert haben, Serverless-Prinzipien auch intern zu nutzen.

Allerdings bedeutet der Wechsel zum Serverless-Modell für Anwendungen, die im JIT-Paradigma entwickelt wurden, dass die Vorteile von JIT nicht mehr greifen – ja sogar kontraproduktiv werden können.

Zum einen erlaubt es die Containerisierung (z. B. mit Docker) dem Entwickler, die gesamte Anwendungsumgebung inklusive Betriebssystem zu kontrollieren – ohne sich um die zugrundeliegende Hardware kümmern zu müssen. Dadurch ist der JVM-Portabilitätsvorteil weitgehend irrelevant geworden, da man gezielt ein spezifisches Zielbetriebssystem innerhalb des Containers auswählen kann.

Zum anderen arbeitet Java immer noch innerhalb einer JVM – und somit bleibt die Startzeit der JVM ein realer Flaschenhals. Wenn man dann noch umfangreiche Bibliotheken wie Spring verwendet, die zur Laufzeit eine große Anzahl an Beans laden, werden Anwendungen zu schwergewichtig, um sie nach Bedarf starten und stoppen zu können. Startzeiten können je nach Komplexität mehrere Sekunden bis hin zu Minuten betragen.

Die Umstellung solcher monolithischer Anwendungen auf mehrere kleinere Microservices ist zwar längst Realität und wird von vielen Java-Entwickler:innen praktiziert – dennoch bleiben Startzeiten im Sub-

Sekunden-Bereich eine Herausforderung.

Ganz so düster ist die Lage jedoch nicht – die Java-Community stellt sich der Herausforderung. Eine der Lösungen ist bereits historisch bewährt: In klassischen kompilierten Sprachen wie C oder C++ wird der Code direkt in native ausführbare Binärdateien für die Zielplattform übersetzt.

Im Gegensatz zur Java-Bytecode-Kompilierung erzeugen diese Sprachen direkt plattformabhängige Maschinencode-Binaries. Diese enthalten nicht nur den Programmcode selbst, sondern auch alle zugehörigen Bibliotheken, Systemfunktionen und Abhängigkeiten – alles in einer einzigen, optimierten Binärdatei verlinkt. Dieser Prozess wird als Ahead-of-Time (AOT) Compilation bezeichnet.

Bei AOT-Kompilierung sind alle für die Ausführung benötigten Bibliotheken bereits zur Kompilierzeit bekannt. Dadurch entfällt zur Laufzeit jegliche dynamische Klassennachladung oder Bytecode-Interpretation – was zu schnellerem Start und geringerem Ressourcenverbrauch führt.

Die Lösung für das Problem langer Start- und Aufwärmzeiten in Java-Anwendungen liegt also in der Rückbesinnung auf bewährte Prinzipien – kombiniert mit einer modernen Java-gerechten Umsetzung: und genau hier kommt GraalVM ins Spiel.

Ohne zu sehr in die technischen Details der AOT-Kompilierung einzutauchen, lässt sich sagen: GraalVM ist eine Technologieplattform, die genau das ermöglicht. GraalVM analysiert vorhandenen Java-Code statisch, erkennt alle genutzten Abhängigkeiten und generiert daraus eine native ausführbare Datei oder Bibliothek. Dabei werden alle Klassen aus dem Classpath und dem JVM-Laufzeitsystem aufgenommen, die tatsächlich benötigt werden. Alles andere wird weggelassen, was die resultierende Binärdatei deutlich schlanker und effizienter macht.

Eine so generierte native Binärdatei ist in vielen Fällen deutlich schneller startbereit als eine reguläre Java-Anwendung, die innerhalb der JVM ausgeführt wird. Denn es gibt keine JVM, die gestartet werden muss, keine Klassen, die dynamisch geladen werden, keine Just-in-Time-Kompilierung – alle Verknüpfungen sind bereits zur Compilezeit erfolgt, sodass die Anwendung beim Start direkt loslegen kann.

Natürlich hat auch dieser Ansatz gewisse Einschränkungen. Da GraalVM eine statische Analyse durchführt, kann es bei Programmen, die dynamische Funktionen wie Reflection, JNI (Java Native Interface) oder Serialisierung einsetzen, vorkommen, dass nicht alle Abhängigkeiten automatisch erkannt werden. In solchen Fällen müssen Entwickler konfigurationsseitig nachhelfen, um GraalVM mitzuteilen, welche weiteren Klassen oder Ressourcen zur Laufzeit verwendet werden sollen.

Die Dokumentation von GraalVM bietet hierfür detaillierte Werkzeuge und Hilfsmittel – diese technischen Feinheiten gehen jedoch über den Rahmen dieses Artikels hinaus.

Festzuhalten bleibt: Mit der Verfügbarkeit von Ahead-of-Time-Kompilierung durch GraalVM lassen sich Java-Anwendungen deutlich besser für die Cloud adaptieren – sei es zur Ausführung in Kubernetes-Umgebungen, in einem Knative-Setup oder sogar innerhalb von AWS Lambda.

Aspekt	JVM (z. B. HotSpot)	GraalVM Native Image
Startzeit	Langsamer (durch Klassenladen, JIT-Warm-up)	Sehr schnell (Ahead-of-Time kompiliert, nahezu sofortiger Start)
Speicherverbrauch	Höher (JIT-Overhead, Laufzeit-Metadaten)	Geringer (minimale Laufzeit, aggressive Optimierungen)
Leistung (Spitzenwert)	Höher (JIT-Optimierungen passen sich zur Laufzeit an)	Gut, aber im Allgemeinen geringer als JVM bei lang laufenden Prozessen
Build-Zeit	Schnell (Standard-Kompilierung)	Langsam (Erzeugung des Native Image ist komplex und zeitaufwändig)
Anwendungsgröße	Kleiner (nur Bytecode und Abhängigkeiten)	Größer (enthält statisch verlinkten nativen Code)

Kompabilität	Breit (unterstützt die meisten Java-Bibliotheken und -Funktionen)	Eingeschränkt (dynamische Features und Reflection benötigen Konfiguration)
Warm-up-Verhalten	Erfordert Warm-up, um Spitzenleistung zu erreichen	Kein Warm-up erforderlich
Eignung für Deployment	Ideal für lang laufende Dienste	Ideal für kurzlebige/ serverless oder „scale-to-zero"-Anwendungen

Im weiteren Verlauf dieses Artikels zeigen wir, wie eine unternehmensreife Spring Boot Java-Anwendung auf Kubernetes in eine serverlose Anwendung mit Knative überführt werden kann. Wir erklären, in welchen Szenarien eine solche Migration sinnvoll ist und welche Schritte notwendig sind. Zudem gehen wir auf häufige Stolpersteine ein und wie man sie umgeht. Die verwendete Demo-Anwendung ist auf GitHub verfügbar: https://github.com/stein-solutions/java-knative-demo

Die Demo-Anwendung

Unsere Demo-Anwendung ist eine Java Spring Boot-Anwendung mit einer einfachen REST-API, mit der Produkt- und Kategorie-Entitäten verwaltet werden können. Obwohl die Anwendung zustandslos ist (sie speichert keinen Zustand auf einem physischen Laufwerk), initialisiert sie eine H2-In-Memory-Datenbank mit Produkt- und Kategorietabellen über JPA. Dies simuliert realitätsnahe Startzeiten. In realen Anwendungen würde H2 durch persistente Datenbanken wie MySQL oder PostgreSQL ersetzt werden.

Darüber hinaus bietet die Anwendung Metriken im Prometheus-Format zur Integration in ein Cloud-natives Monitoring. Diese Metriken können regelmäßig durch Prometheus abgefragt werden. Die nötige Prometheus-Konfiguration wird mithilfe des Kubernetes Prometheus Operators und der ServiceMonitor Custom Resource automatisch angewendet.Die Anwendung wird als Deployment auf Kubernetes ausgerollt und ist über einen HTTP-Endpunkt erreichbar, der über eine Kubernetes Ingress-Resource konfiguriert wird. Eine automatische Skalierung ist für die Demo-Anwendung nicht aktiviert.

Erstellung eines nativen Builds

Nach Installation der erforderlichen GraalVM-Abhängigkeiten kann ein nativer Build einer Spring Boot-Anwendung einfach erstellt werden. Aktuelle Spring Boot-Versionen bringen bereits ein natives Maven-Profil und das Plugin native-maven-plugin mit. Mit folgendem Befehl erstellen wir ein GraalVM-Native-Binary unserer Anwendung:

```
mvn -Pnative native:compile
```

Das Ergebnis ist ein Ahead-of-Time (AOT) kompiliertes Binärprogramm, das direkt auf dem Host System lauffähig ist. Bibliotheken, die Reflections oder JNI nutzen, müssen diese Aufrufe registrieren. Falls das nicht geschieht (z. B. bei H2), kann das Community-Projekt graalvm-reachability-metadata (https://github.com/oracle/graalvm-reachability-metadata) helfen. Wo dies nicht verfügbar ist, müssen Entwickler selbst sogenannte Runtime-Hints angeben, etwa bei dynamischem Code wie bei Project Lombok.

Dockerisierte Builds und Ausführungsumgebungen

Als Nächstes dockerisieren wir das native Executable. Wir empfehlen die Verwendung eines Multistage-Builds, bei dem wir in einem ersten Schritt das native Executable erstellen und im zweiten Schritt das endgültige Container-Image haben, das nur das erstellte Executable und möglicherweise seine Abhängigkeiten enthält. Für den Build-Schritt empfehlen wir, ein von GraalVM bereitgestelltes Basis-Image zu verwenden. Dieses Basis-Image enthält bereits alle erforderlichen Abhängigkeiten, um erfolgreich ein natives Executable unserer Anwendung zu erstellen. Wir müssen den Container-Bauprozess nur anweisen, das native Executable zu erstellen, indem wir das oben erwähnte Maven-Ziel aufrufen. Der vollständige Build-Schritt sieht wie folgt aus:

```
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
```

Der zweite Schritt unseres Container-Bauprozesses bereitet die Laufzeitumgebung für unser natives Executable vor. Wenn wir ein Basis-Image verwenden, das dieselben OS-Level-Abhängigkeiten wie das Basis-Image des Build-Schritts bietet, müssen wir nur das erstellte Executable kopieren und einen Entrypoint-Befehl definieren, der dem Container-Runtime mitteilt, was beim Starten des Containers ausgeführt werden soll. In diesem Beispiel verwenden wir „ubuntu:jammi" als Basis-Image. Es ist außerdem eine gute Praxis, hier die Ports zu definieren, auf denen unsere Anwendung hört. Dies beeinflusst jedoch nicht direkt, wie der Container ausgeführt wird und dient hier nur beschreibenden Zwecken. Das vollständige Dockerfile sieht wie folgt aus:

```
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"]
```

Das resultierende Container-Image ist etwa 220 MB groß.

Optimierung durch statische Verlinkung

Das native Executable enthält noch viele nicht benötigte Betriebssystem-Bibliotheken durch die Verwendung von ubuntu:jammy als Basis-Image. Wir können die Dateigröße weiter reduzieren, indem wir ein vollständig statisch verlinktes Executable unserer Anwendung erstellen. GraalVM unterstützt dies direkt, wir müssen nur die Parameter --static und --libc=musl zum Build-Befehl hinzufügen, und das resultierende Executable wird mit musl als Standard-C-Compiler-Bibliothek verlinkt.

Es empfiehlt sich, ein separates Maven-Profil für statisch verlinkte Builds zu erstellen. Der folgende Code zeigt, wie das geht:

```
    <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>
```

Dieser Code erweitert die Konfiguration des native-maven-plugin, um die erforderlichen Parameter hinzuzufügen. Um den Build auszuführen, führen wir mvn -Pnative,nativelinked native:compile aus. Es ist wichtig zu beachten, dass der obige Befehl auf ARM-Macs fehlschlagen wird. Das resultierende Executable enthält alle erforderlichen Abhängigkeiten, um auf einer bestimmten CPU-Architektur ausgeführt zu werden.

Mit allen OS-Level-Abhängigkeiten, die in das statische Executable eingebunden sind, können wir das Basis-Image für den Anwendungs-Container entfernen und stattdessen das Container-Image von Grund auf neu erstellen. Das resultierende Docker-Image enthält nur unser Executable und keine anderen Dateien oder Ordner. Damit der eingebettete Tomcat innerhalb von Spring Boot funktioniert, müssen wir allerdings noch ein /tmp-Verzeichnis erstellen. Da unser Scratch-Basis-Image jedoch keine Tools zum Erstellen von Verzeichnissen hat, kopieren

wir einfach ein leeres Verzeichnis aus dem Build-Schritt.

```
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“]
```

Deployment auf Kubernetes mit Knative

Knative ist eine Open-Source-Lösung für Kubernetes, die die Bereitstellung von serverlosen Anwendungen ermöglicht, die auf Kubernetes auf Null skalieren können. Um eine Anwendung auf einem Knative-aktivierten Kubernetes-Cluster bereitzustellen, kann die folgende YAML-Konfiguration verwendet werden:

```
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
```

Dies erstellt einen Knative-Dienst, der dynamisch basierend auf der Anzahl der HTTP-Anfragen skaliert, die an die Anwendung gesendet werden. Wenn keine HTTP-Anfragen an unsere Anwendung gerichtet sind, wird Knative die Anwendung auf Null Replikate skalieren, wodurch alle CPU- und Speicherkapazitäten freigegeben werden. In unserer Konfiguration ist der Zeitraum, nach dem der Knative skaliert, wenn keine Anfragen empfangen werden, auf 6 Sekunden festgelegt (der minimale mögliche Zeitraum). In Produktionsumgebungen ist es wahrscheinlich sinnvoll, einen längeren Zeitraum zu wählen, um Kaltstarts zu vermeiden.

Kaltstarts - das Übel der serverlosen Architektur und wie man es löst

Für serverlose Anwendungen sind akzeptable Kaltstartzeiten eine Notwendigkeit. Wenn man benutzerorientierte APIs betreibt, sind Antwortzeiten von mehreren zehn Sekunden inakzeptabel. Hier kommen native Builds ins Spiel.

„Normale" (also nicht-native) Spring Boot-Anwendungen haben relativ langsame Startzeiten. Während wir unsere Demo-Anwendung auf einem M2 Pro MacBook ausführen, dauert der Start der Anwendung etwa 1,6 Sekunden, jedoch überschreitet die Startzeit in ressourcenbeschränkten Umgebungen leicht 10 Sekunden und mehr. Wenn wir die verfügbaren CPUs für unsere containerisierte Spring Boot-Anwendung auf 200 Milicores (20% eines CPU-Kerns) begrenzen, benötigt die Anwendung rund 30 Sekunden, bis sie mit Knative vollständig gestartet ist. Auch bei einer vollen CPU dauert es noch etwa 10 Sekunden, und wenn keine CPU-Nutzung eingeschränkt wird (auf einem Azure Standard_D2s_v3 Worker Node - 2 verfügbare CPUs), erreichen wir Startzeiten von rund 5 Sekunden.

Das Problem bei der Zuweisung einer vollständigen CPU oder mehr an unsere Demo-Anwendung ist, dass diese Ressourcen für den gesamten Lebenszyklus der Anwendung blockiert werden und nicht nur für den Start der Anwendung, wenn sie benötigt werden. Nach dem Start benötigt die Anwendung nur etwa 200 Milicores oder weniger, um HTTP-Anfragen zu beantworten, und während der Leerlaufzeit muss praktisch keine Ressource für unsere Anwendung reserviert werden. Allerdings blockiert unsere Anwendung weiterhin 100% des angeforderten CPU-Kerns.

Um das Problem der Ressourcenüberversorgung zu lösen, bietet Kubernetes die Möglichkeit, Ressourcenerfordernisse und -limits für Container zu konfigurieren. Ressourcenerfordernisse definieren die minimal garantierten CPU- und Speicherressourcen, auf die ein Container zugreifen kann, um Stabilität und Leistung zu gewährleisten. Limits hingegen setzen ein Maximum an Ressourcen, das ein Container verwenden kann, um zu verhindern, dass er mehr als seinen zugewiesenen Anteil verbraucht und andere Workloads beeinträchtigt.

Durch die Festlegung eines niedrigeren CPU-Anforderungswerts (z.B. 200 Millicores) und eines höheren CPU-Limits (z.B. 1 voller Kern) für unsere Spring Boot-Anwendung können wir ein Gleichgewicht finden. Diese Methode stellt sicher, dass die Anwendung während des Starts ausreichend Ressourcen erhält, während der Leerlaufressourcenverbrauch verringert wird. Die Anwendung kann bei Bedarf vorübergehend zusätzliche Ressourcen nutzen, ohne diese dauerhaft für andere Workloads zu

blockieren. Ein wesentlicher Nachteil dieser Methode besteht darin, dass der Anwendung kein vollständiger CPU-Kern garantiert zur Verfügung steht, wenn nicht genügend CPU-Kapazität vorhanden ist; sie erhält in diesem Fall lediglich den zugesicherten Minimalwert von 200 Millicores. Zudem bleiben diese 200 Millicores auch während inaktiver Phasen für die Anwendung reserviert und stehen somit anderen Anwendungen nicht zur Verfügung.

Durch die Optimierung der Ressourcenzuweisung mit Kubernetes und die dynamische Anpassung von CPU-Anforderungs- und -Limitwerten können wir die Notwendigkeit der Überversorgung verringern, wodurch die Anzahl der in unserem Cluster benötigten CPUs begrenzt wird. Diese Methode unterstützt eine nachhaltigere IT-Infrastruktur, indem sie den Umwelteinfluss durch die Aufrechterhaltung überschüssiger Hardware minimiert.

Wir haben einige Tests mit unserer Demo-Anwendung durchgeführt, die über Knative auf einem Azure Kubernetes-Cluster bereitgestellt wurde. Als Worker-Nodes verwendeten wir Azure Standard_D2s_v3-Instanzen mit 2 Kernen und 6 GB RAM. Die folgende Tabelle zeigt Metriken zu der Request-Duration und App-Startzeiten bei Kaltstarts. Für jede Konfiguration, wie sie durch CPU und Speicherzuweisung definiert ist, haben wir 80 Kaltstarts gemessen. Alle Messungen hatten das Container-Image unserer Anwendung bereits auf dem jeweiligen Worker-Node heruntergeladen. Da nicht nur die Anwendung gestartet werden muss, sondern auch Knative die Anfrage zum gestarteten Pod weiterleiten muss, dauert die gesamte Anfrage länger als nur der einfache App-Start.

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

Wenn der Anwendung ein vollständiger CPU-Kern zugewiesen wird, dauert ein Kaltstart im Durchschnitt 990 ms. Allerdings benötigt unsere Anwendung nur etwa 530 ms, um zu starten und bereit zu sein, Anfragen zu verarbeiten. Die verbleibende Zeit wird von der Knative-Activator-Komponente beansprucht, bevor die Anfrage an den tatsächlichen Anwendungspod weitergeleitet wird.

Eine bemerkenswerte Entdeckung war, dass wir die CPU- und Speicherzuweisung auf etwa 125 Millicores (1/8 eines CPU-Kerns) und 256 MB RAM reduzieren können, bevor sich die Antwortzeit der Anwendung signifikant verschlechtert. Mit dieser Konfiguration erreichen wir immer noch eine durchschnittliche Anfragedauer von 1,177 Sekunden – nur etwa 180 ms langsamer als mit einem vollen CPU-Kern. Allerdings ist die Startzeit der Anwendung in dieser Konfiguration deutlich höher. Die App benötigt hier im Schnitt 1,021 Sekunden zum Starten – fast 500 ms mehr als in der ersten Konfiguration. Dies verschafft uns als Anwendungsentwickler einen gewissen Spielraum: Ein App-Start von etwa einer Sekunde führt zu ähnlichen Antwortzeiten wie ein App-Start von etwa 500 ms.

Bei Konfigurationen mit 75 Millicores (1/16 eines CPU-Kerns) nehmen die Anfragen deutlich mehr Zeit in Anspruch. Dieser Anstieg in der Anfragedauer ist hauptsächlich auf die längere Startzeit der Anwendung zurückzuführen – der Overhead durch das Knative-Routing beträgt lediglich rund 100 ms.

Einschränkungen

Die Architektur hinter Knative erlaubt lediglich Autoscaling auf Basis von HTTP-Anfragen, nicht jedoch auf Basis von TCP-Verbindungen. Aus diesem Grund können WebSocket-Verbindungen zwar initialisiert werden, werden aber bei Skalierungsentscheidungen von Knative nicht berücksichtigt.

Zudem ist es für akzeptable Kaltstartzeiten notwendig, dass die Anwendung zustandslos (stateless) ist. Obwohl es technisch möglich ist, persistente Volumes einzubinden, wird davon abgeraten. Der Mounting-Prozess solcher Volumes dauert üblicherweise mehrere Sekunden, was zu inakzeptablen Kaltstartzeiten führt. Selbst das Mounten des Kubernetes-Service-Account-Tokens zur Anwendung erhöht die durchschnittliche Anfragedauer eines Kaltstarts um etwa 200 ms.

Aus Nachhaltigkeitsperspektive ist es notwendig, mehrere Anwendungen auf unserer Infrastruktur zu betreiben, bevor sich eine spürbare Verbesserung ergibt. Laut offizieller Dokumentation benötigt Knative selbst mindestens 6 CPUs und 6 GiB RAM, um auf einem Kubernetes-Cluster betrieben zu werden. Damit unser Setup nachhaltig ist, müssen also eine kritische Anzahl von Knative-Services im Cluster laufen.

Darüber hinaus sollte das Lastmuster, das von unserer Anwendung verarbeitet wird, volatil sein, sodass längere Zeiträume ohne Anfragen tatsächlich vorkommen. Knative spielt seine Stärken bei selten angefragten Anwendungen aus – je konstanter die Last ist, desto weniger Ressourcen können durch dieses Modell eingespart werden.

Ausblick

In diesem Artikel haben wir gezeigt, dass es möglich ist, zustandslose, unternehmensfähige Java-Anwendungen als serverlose Container auf Kubernetes zu betreiben. Die Kaltstartzeiten dieser Container sind mit denen anderer serverloser Technologien wie AWS Lambda vergleichbar. Dieses Vorgehen kann in bestimmten Lastszenarien zu nachhaltigeren IT-Systemen führen. Anwendungen, die nur selten aufgerufen werden, aber dennoch schnelle Reaktionszeiten erfordern, würden von dieser Architektur profitieren.

Wie bei allen IT-Architekturen gilt jedoch: Diese Lösung ist kein Allheilmittel für alle Anwendungen und Anforderungen.

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/

> Zurück zum Inhaltsverzeichnis

#JAVAPRO #ARCHITECTURE #MICROSERVICES

Modularer Monolith mit SpringBoot & Maven

Autor:

Max Beckers

Solutions Architect bei PAYONE GmbH / einem Unternehmen der Worldline-Gruppe,
FinTech, Zahlungsverkehr, Speaker, Blogautor, Open-Source-Enthusiast, Ehemann & Vater / Familie, Kochen

https://www.linkedin.com/in/maximilianbeckers/

Monolithen sind ein bekanntes Architekturmuster, bei dem die gesamte Software als ein Deployable bereitgestellt wird. Sie sind dafür bekannt, dass ihre Wartbarkeit und Erweiterbarkeit im Laufe der Zeit zu einer Herausforderung werden können. Microservices sollen genau diesen technischen Schulden entgegenwirken. Dabei handelt es sich um einen Architekturansatz, bei dem die Fachlogik in einzelne Services aufgeteilt und separat bereitgestellt wird, was zu einer hohen Flexibilität führt. Diese Flexibilität hat jedoch auch ihren Preis: erhöhte Infrastukturkomplexität, zusätzlicher Netzwerktraffic, erschwertes Debugging von Fehlern und Bugs sowie kompliziertere Lösungen für Transaktionsprozesse, beispielsweise beim Schreiben in Datenbanken.

Es gibt jedoch noch eine weitere Architektur, die sich zwischen Monolithen und Microservices einordnen lässt: den modularen Monolithen. Vielleicht stellen sich nun Fragen wie „Was ist das?“, „Welche Vorteile bietet er?“ oder „Wie sieht das konkret aus?“. Genau diesen Fragen möchte ich in diesem Beitrag nachgehen und Antworten liefern.

Die Idee eines modularen Monolithen

Die Grundidee eines modularen Monolithen - auch Modulith genannt - besteht darin, weiterhin ein einziges Deployable wie bei einem klassischen Monolithen zu erstellen. Gleichzeitig wird jedoch durch eine klare Definition von Modulen darauf geachtet, die Logik sauber zu trennen und zu kapseln, ähnlich wie bei Microservices. Dadurch wird die Wartbarkeit der Software verbessert, ohne dass eine aufwändige Netzwerkkommunikation erforderlich ist. Jedes Modul definiert eine API, die von anderen Modulen genutzt werden kann. Diese API kann Folgendes beinhalten:

1. **DTOs (Data Transfer Objects):** Ähnlich den Objekten in einer OpenAPI-Definition. Sie beschreiben die Daten, die an der Schnittstelle ausgetauscht werden. Notwendige Enums zählen in diesem Fall zu den DTOs.

2. **Interfaces:** Vergleichbar mit den Endpunkten einer API-Definition, also den Funktionen, die das Modul bereitstellt.

3. **Exceptions:** Vergleichbar mit den Error-Responses und -Codes in einer RESTful API.

4. **Messages:** Definitionen von Datenstrukturen für asynchrone Kommunikation (z.B. über Message Queues wie Kafka).

Durch diese Struktur entstehen ähnlich wie in einer Microservice-Architektur sauber abgegrenzte Module. Für den Aufrufer bleibt das Innere eines Moduls eine Blackbox, und es darf nicht direkt aufgerufen werden.

Es gibt viele Möglichkeiten, die Trennung zwischen der API eines Moduls und dessen Fachlogik umzusetzen. Ich bevorzuge es, die API in unterschiedliche Namespaces wie DTO, Interface, Exception und Message direkt auf oberster Ebene im Modul zu platzieren, während die gesamte

interne Logik in einem „Internal"-Namespace bleibt. Dies ermöglicht es, schnell zu erkennen und auch automatisiert sicherzustellen, dass keine interne Logik eines anderen Moduls direkt aufgerufen wird. Alternativ kann man auch die API und die Logik in zwei separate Module aufteilen.

Ein Aspekt, den ich bei der Modularisierung noch nicht angesprochen habe, betrifft die eingehende HTTP-Kommunikation. Wo sollten die Controller platziert werden? Sollte jedes Modul seine eigenen Controller haben, um direkt aufgerufen werden zu können, oder gibt es ein zentrales Modul mit allen Controllern, das als eine Art API-Gateway fungiert und die Anfragen an die entsprechenden Module weiterleitet? Wie so oft ist dies eine Architekturentscheidung, die getroffen werden muss, und sie lässt sich nicht pauschal beantworten. Persönlich ziehe ich es vor, die Anfragen in einem „API-Gateway" zu bündeln und von dort aus an die entsprechenden Module zu verteilen. Dies ist insbesondere dann sinnvoll, wenn eine Anfrage an verschiedene Module gerichtet werden soll. Letztlich hängt die Entscheidung stark von den Anforderungen ab. Es ist ebenso möglich, dass jedes Modul einen eigenen Controller-Namespace erhält, in dem die Controller für dieses Modul verwaltet werden und von dort aus die modulinterne Logik aufrufen.

Mit einem Monolith anfangen

Ein bekanntes Zitat, wenn es um Microservices geht, stammt von Martin Fowler:

> **„You shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile."** - *Martin Fowler, MonolithFirst*

Diese Aussage impliziert indirekt das Konzept eines Modulithen. Zu Beginn eines Projekts sollte man, um die Software einfach handhaben zu können, mit einem Monolithen und seinen Vorteilen arbeiten. Dabei ist es wichtig, bereits von Anfang an Module zu definieren, die später als eigenständige Deployables (Microservices) ausgegliedert werden können. Der große Vorteil liegt darin, dass Anpassungen an Modulgrenzen und Schnittstellen schnell und unkompliziert vorgenommen werden können. Sobald sich ein stabiles Fundament etabliert hat, kann man beginnen, einzelne Teile der Anwendung in separate Deployables zu überführen.

Auch bei einem gewachsenen Monolithen ohne klar definierte Module ist es sinnvoll, diesen zunächst in einen Modulithen zu transformieren, bevor man möglicherweise einzelne Microservices herauszieht. Ein wichtiger Schritt bei einem gewachsenen Monolithen ist, den Code sinnvoll aufzubrechen. Ein mindestens ebenso wichtiger und oft schwierigerer Schritt ist es, die Daten zu trennen. Jedes Modul sollte seine eigene Datenhaltung haben, wobei die Daten von anderen Modulen getrennt sind und ein direkter Zugriff mittels Berechtigungen ausgeschlossen wird. Es gibt verschiedene Ansätze zur Datentrennung: von einem Datenbankschema mit unterschiedlichen Tabellen-Präfixen über verschiedene Schemas bis hin zu unterschiedlichen Datenbanken (je nach Anwendungsfall eventuell auch verschiedene Datenbanktypen wie Postgres, MySQL, MongoDB). Wenn möglich, empfehle ich, Tabellen innerhalb eines Schemas zu vermeiden, um eine klare Trennung zu gewährleisten.

Eine wichtige Frage in diesem Zusammenhang ist: Brauche ich wirklich die Flexibilität von Microservices? Oder geht es in den meisten Systemen nicht vielmehr darum, eine gute Wartbarkeit und Änderbarkeit sicherzustellen? Meiner Erfahrung nach ist ein Modulith oft eine gute Lösung. In den meisten Fällen muss bei erhöhter Last das gesamte System skaliert werden, und nicht nur einzelne Teile der Anwendung.

Ein weiterer Grund, sich für Microservices zu entscheiden, kann sein, dass verschiedene Module von unterschiedlichen Teams verwaltet werden. In einer solchen modulithischen Anwendung kann die Koordination von Releases mit mehreren Entwicklungsteams ein guter Grund für die Entkopplung in unabhängige Deployables sein. Grundsätzlich ermöglicht die Trennung der fachlichen Module jedoch auch die Entwicklung durch mehrere Teams innerhalb derselben Software.

Modulith mit SpringBoot und Maven

Wie sieht ein Modulith konkret aus? In diesem Abschnitt möchte ich das genauer beleuchten. Da wir über verschiedene Module sprechen, beginnen wir mit einem Multi-Maven-Projekt. Hierbei fungiert das Hauptprojekt als Wrapper und die fachlich definierten Module als Sub-Module.

Nehmen wir zur besseren Veranschaulichung folgendes Beispiel: Wir haben vier Module - Ordering (Bestellmanagement), Inventory (Inventarmanagement), Shipping (Versandmanagement) und Payment (Bezahlmanagement).

Die Projektstruktur sieht dann folgendermaßen aus:

- pom.xml (main pom)
  - api-gateway
    - pom.xml
    - main (including the Controllers)
    - test
  - ordering
    - pom.xml
    - main
    - test
  - inventory
    - pom.xml
    - main
    - test
  - shipping
    - pom.xml
    - main
    - test
  - payment
    - pom.xml
    - main
    - test

Die Main-POM (pom.xml) sieht so aus:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.4.2</version>
    <relativePath/>
</parent>

<groupId>com.example</groupId>
```

```
<artifactId>modulith-example</artifactId>
<version>1.0.0</version>
<packaging>pom</packaging>
<name>Modulith Example</name>
<modules>
    <module>api-gateway</module>
    <module>ordering</module>
    <module>inventory</module>
    <module>shipping</module>
    <module>payment</module>
</modules>
...
```

In den POM-Dateien der Sub-Module wird der Main-POM als Parent definiert. Um die Struktur eines Moduls verständlicher zu machen, schauen wir uns das Modul „Ordering" genauer an:

- com.example.ordering
  - DTO
    - Address.java
    - Customer.java
    - Order.java
    - ShoppingCart.java
    - ...
  - Exception
    - InvalidOrderException.java
  - Interface
    - OrderInitiator.java
  - **Internal**
    - Consumer
      - ChargebackConsumer.java (handlesChargeback Message.java)
    - Entity
    - Repository
    - Service
      - OrderService.java (implements OrderInitiator.java)
  - Message
    - ChargebackMessage.java

Diese Übersicht halte ich absichtlich schlank, um die grundsätzliche Idee des Modulaufbaus zu zeigen. Es gibt DTOs für den Datentransfer, Exceptions für Fehlerfälle, Interfaces, die dem Client die Logik bereitstellen, sowie Messages für die asynchrone Kommunikation. In diesem Beispiel wird ein synchroner Ablauf zur Erstellung neuer Bestellungen über den OrderService dargestellt. Diese Funktionalität wird dem Client über das OrderInitiator-Interface bereitgestellt. Zudem verarbeitet der ChargebackConsumer Nachrichten, die asynchron über Chargebacks in das System gelangen.

Dieser Ansatz lässt sich auf jedes System übertragen. Selbstverständlich bieten nicht alle Module sowohl synchrone als auch asynchrone Schnittstellen an.

Eine weitere Möglichkeit zur Implementierung eines Modulithen ist das Projekt [Spring Modulith](). Dieses Projekt verfolgt einen etwas anderen Ansatz: Es verwendet nicht mehrere Module, sondern definiert Module über Namespaces innerhalb eines einzigen Moduls. Der Aufbau innerhalb der Module ist jedoch identisch oder ähnlich zu dem zuvor gezeigten.

Fazit

Der modulare Monolith ist kein Allheilmittel gegen alle Herausforderungen, die auf Software zukommen. Dennoch bietet er eine wertvolle Alternative zu Microservices, indem er die Komplexität an einigen Stellen reduziert. Bei der Entwicklung neuer Software sollte ein Modulith als Ausgangspunkt in Betracht gezogen werden. Dies umfasst sauber getrennte Module mit eigener Datenhaltung. Einzelne Module können dann bei Bedarf als Microservices aus der Code-Basis herausgelöst werden. Dank der klar definierten APIs ist dies mit geringem Aufwand möglich.

Bei der Umsetzung eines Modulithen gibt es viele Freiheiten, wie der Ansatz im Detail gestaltet wird. Wichtig ist jedoch sicherzustellen, dass die Kommunikation nur über die öffentlichen APIs der Module erfolgt und dass die Daten sauber getrennt sind, ohne direkte Integrationen auf Datenbankebene.

> Zurück zum Inhaltsverzeichnis

#JAVAPRO #TESTING #QUALITY

Die Kunst der statischen Codeanalyse

Autor:

Martin Toshev ist Solution Architect und IT-Berater, der professionelle Schulungen für sowohl Einsteiger als auch erfahrene Entwickler durchführt. Er ist Java-Enthusiast und einer der Leiter der Bulgarian Java User Group (BG JUG), wo er zu den Organisatoren der jPrime-Konferenz gehört.

https://www.linkedin.com/in/martin-toshev-07046127/

Die Notwendigkeit der statischen Analyse von Quellcode

Die meisten Java-Entwickler (und nicht nur) haben zumindest eine Art statisches Analysetool verwendet, um eine Aufgabe wie (um nur einige zu nennen) auszuführen:

- Ableiten von Quellcodemetriken wie Codezeilen oder zyklomatische Komplexität;
- Entdecken von Fehlern, Schwachstellen oder Code-Smells wie ungenutzten Variablen (was beliebte IDEs typischerweise tun);
- Durchführen eines automatisierten Refactorings oder einer Code-Vervollständigung;

- Durchsetzung von Code- und Qualitätsstandards.

Um eine statische Codeanalyse durchzuführen, benötigen wir normalerweise eine geeignete Darstellung des Quellcodes, die für die Analyse geeignet ist. Eine Programmiersprache kann durch eine formale Grammatik beschrieben werden. Darüber hinaus kann ein Parser erstellt oder generiert werden, indem man den Regeln einer formalen Grammatik folgt, um aus dem Quellcode eine ordnungsgemäße Darstellung (normalerweise einen Analysebaum) zu erstellen. Abhängig von der Art der Sprache, die wir darstellen möchten, können wir unterschiedliche Arten formaler Grammatiken verwenden:

- reguläre Grammatiken (d. h. reguläre Ausdrücke): Sie sind in den meisten Programmiersprachen verfügbar, werden aber normalerweise für grundlegendere Analyseaufgaben verwendet. In vielen Fällen sind sie nicht zum Parsen einer modernen Programmiersprache geeignet, da sie langsam und schwer zu warten sind;

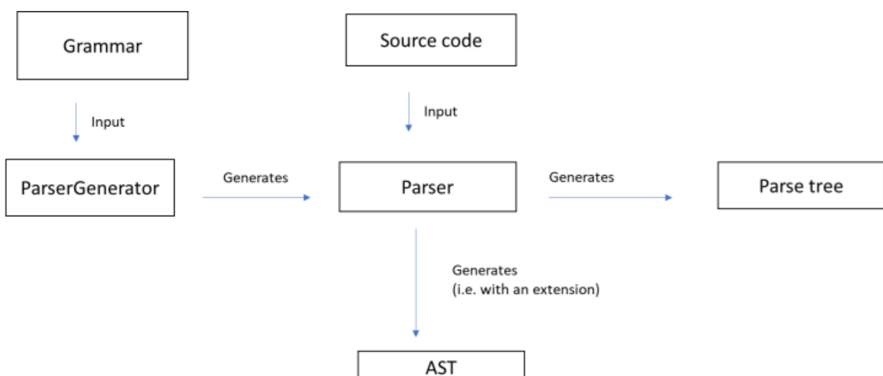
- kontextfreie Grammatiken (z. B. BNF oder eBNF): Eines der bekanntesten Formate ist BNF (und seine Varianten), ein Parser kann auch durch die Grammatikregeln generiert werden;

- andere formale Grammatiken (z. B. PEG).

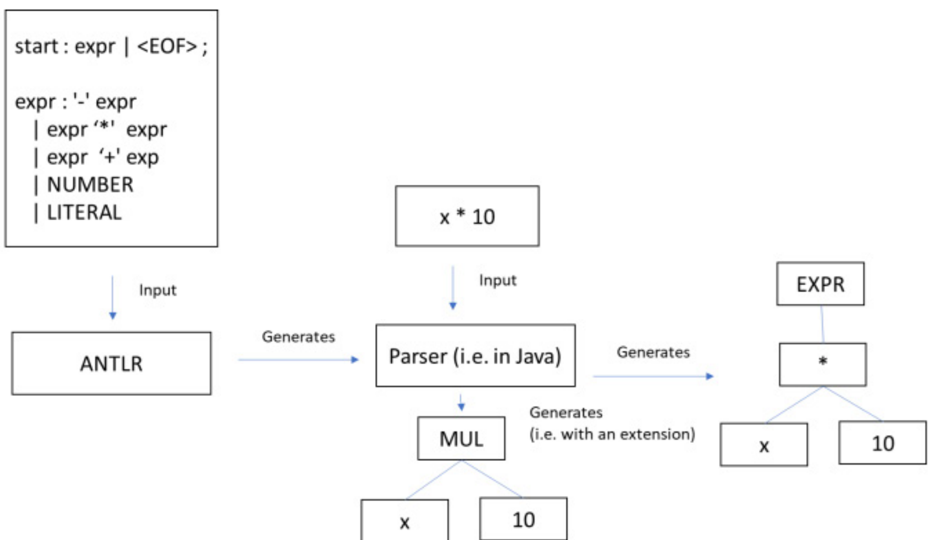
Es kommt häufig vor, dass in der Anfangszeit für verschiedene Tools zur statischen Codeanalyse das manuelle Schreiben eines Parsers erforderlich war, was keine triviale Aufgabe ist ...

Parser-Generatoren zur Rettung ...

Es können Tools erstellt werden, um Parser auf Grundlage kontextfreier Grammatikregeln zu generieren. Dies ist beispielsweise bei Tools wie LEX und YACC der Fall, die in C geschrieben sind und Code in C generieren. Auf hoher Ebene wird die Parsergenerierung durch das folgende Diagramm veranschaulicht:



In den Anfangstagen von Java hat Sun Microsystems einen Parser-Generator namens Jack entwickelt, der später in JavaCC (was für Java Compiler-Compiler steht) umbenannt wurde. Eine weitere beliebte Alternative zum Generieren eines Parsers für eine Java-Grammatik ist ANTLR (ANother Tool for Language Recognition). Beide dieser Parsergeneratoren werden gut unterstützt und sind in Java geschrieben. JavaCC (ähnlich wie YACC für C) kann Grammatikregeln mit Java-Code kombinieren, der in den generierten Parser aufgenommen wird. Allerdings bietet JavaCC nur Codegenerierung für Java, während ANTLR ein Allzweckprogramm ist, über eine große Anzahl von Grammatiken für eine Reihe von Programmiersprachen verfügt und die Möglichkeit bietet, Parser in verschiedenen Sprachen zu generieren. Beide Tools arbeiten mit formalen Grammatiken in eBNF. Betrachtet man beispielsweise das obige allgemeine Diagramm, sieht der Prozess der Parsergenerierung in Antlr anhand eines einfachen Beispiels eines Ausdrucksparsergenerators folgendermaßen aus:



Der vom Parser selbst generierte Analysebaum erfordert mehr Aufwand hinsichtlich der Codeanalyse. Aus diesem Grund bieten Parsergeneratoren normalerweise die Möglichkeit, eine prägnantere Darstellung zu generieren, die zusätzliche Symbole eliminiert und zusätzliche Funktionen zur Symbolauflösung bietet: den AST (abstrakter Syntaxbaum). Der Prozess der Verwendung von ANTLR oder JavaCC in einem Standard-Maven/Gradle-Projekt ist sehr ähnlich. Zum Beispiel für ANTLR:

- Antlr4-Abhängigkeit und Plugin in Maven/Gradle-Build-Datei hinzufügen;

```
<dependency>
   <groupId>org.antlr</groupId>
   <artifactId>antlr4-runtime</artifactId>
   <version>4.7.1</version>
</dependency>
 …
<plugin>
   <groupId>org.antlr</groupId>
   <artifactId>antlr4-maven-plugin</artifactId>
   <version>4.7.1</version>
   <executions>
      <execution>
         <goals>
         <goal>antlr4</goal>
      </goals>
      </execution>
   </execution>
</plugin>
...
```

- Erstellen Sie Grammatikdateien unter src/main/antlr4 (im G4-Format, Java 20-Grammatikdateien verfügbar unter https://github.com/antlr/grammars-v4/tree/master/java/java20);
- Generieren Sie Lexer und Parser mithilfe des Maven/Gradle-Builds.

Sobald dies vorhanden ist, kann der generierte Parser verwendet werden, um einen Analysebaum zu generieren, an den beispielsweise ein Listener für bestimmte Ausführungen während des Analysevorgangs angehängt werden kann. Beispiel mit einem von ANTLR generierten Parser:

```
String content = „public class Example { public void func(int x)
{ return x + 10; } }“;

Java20Lexer lexer = new Java20Lexer(CharStreams.fromString(content));
CommonTokenStream tokens = new CommonTokenStream(lexer);
Java20Parser parser = new Java20Parser(tokens);
ParseTree tree = parser.compilationUnit();
ParseTreeWalker walker = new ParseTreeWalker();
```

```
ExprListener listener = new ExprListener();
walker.walk(listener, tree);
```

Eine alternative Möglichkeit zum Erstellen eines Parsers ist eine Parsing Expression Grammar (PEG). Eine Bibliothek, die diesen Ansatz implementiert (die Grammatikregeln werden direkt als Teil der Anwendung in Java-Code geschrieben), ist Parboiled.

Java-Bibliotheken zur Rettung ...

Parsergeneratoren und PEG-Parser sind ziemlich allgemein. Möglicherweise sind sie auch nicht auf dem neuesten Stand der gewünschten Java-Version. Alternativ kann eine spezialisierte Parsing-Bibliothek wie JavaParser oder Eclipse JDT verwendet werden.

javaparser

Es basiert auf JavaCC, wird gut gepflegt und bietet Unterstützung für JDK 21. Es bietet eine verbesserte Symbolauflösung und generiert einen AST aus dem Quellcode. Darüber hinaus bietet es die Möglichkeit, den AST über eine von der Bibliothek bereitgestellte DSL abzufragen, Code aus dem AST zu generieren oder ihn zu ändern. Der Einstieg in die Verwendung der Bibliothek ist ganz einfach. Das folgende Beispiel zählt die Anzahl der Methoden in einer Klasse:

```
public static int countMethods(File file) throws FileNotFoundException
{
   CompilationUnit cu = StaticJavaParser.parse(file);
   int count = 0;
   for (Node node : cu.findAll(MethodDeclaration.class)) {
      count++;
   }
   return count;
}
```

Um mit der Verwendung von JavaParser zu beginnen, reicht es aus, die folgende Abhängigkeit einzubinden:

```
<dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-symbol-solver-core</artifactId>
    <version>3.26.3</version>
</dependency>
```

Eclipse JDT

Eclipse JDT (Java Developer Tools) ist der Hauptantrieb des Java-Editors in der Eclipse IDE, der erweiterte Funktionen wie partielle Kompilierung, Code-Vervollständigung usw. bietet. In früheren Zeiten von Eclipse war es nicht so einfach, JDT außerhalb der Eclipse IDE zu verwenden, vor allem, weil hierfür auch eine Reihe zusätzlicher Abhängigkeiten mitgezogen werden mussten. Jetzt ist Eclipse JDT als eigenständige Bibliothek über die folgende Abhängigkeit verfügbar:

```
<dependency>
   <groupId>org.eclipse.jdt</groupId>
   <artifactId>org.eclipse.jdt.core</artifactId>
   <version>3.36.0</version>
</dependency>
```

Das folgende Beispiel implementiert eine Methode zum Zählen der Anzahl der Methoden in einer Java-Klasse:

```
public static int countMethods(File file)
    throws IOException, MalformedTreeException,
    BadLocationException {

    String source = FileUtils.readFileToString(file,
    Charset.defaultCharset());
    Document document = new Document(source);
    ASTParser parser = ASTParser.newParser(AST.JLS21);
    parser.setSource(document.get().toCharArray());
    CompilationUnit unit = (CompilationUnit) parser.createAST(null);

    int count = 0;
    List<AbstractTypeDeclaration> types = unit.types();
    for (AbstractTypeDeclaration type : types) {
```

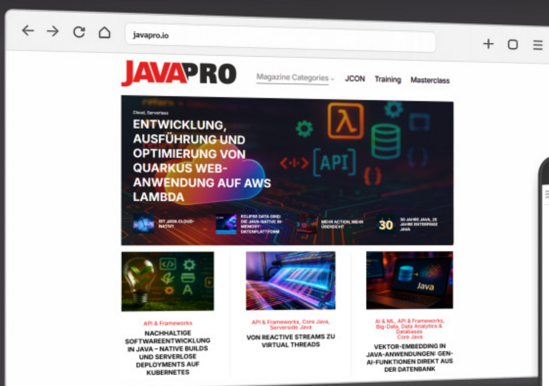
```
            if (type.getNodeType() == ASTNode.TYPE_DECLARATION) {
            List<BodyDeclaration> bodies = type.bodyDeclarations();
            for (BodyDeclaration body : bodies) {
                if (body.getNodeType() == ASTNode.METHOD_DECLARATION) {
                    count++;
                }
            }
        }
}
return count;
}
```

Wie Sie sehen, stehen Ihnen mehrere Optionen zur Auswahl, um mit dem Schreiben Ihres eigenen Tools zur statischen Analyse von Java-Code zu beginnen.

> Zurück zum Inhaltsverzeichnis

ENTDECKE WEITERE SPANNENDE ARTIKEL ONLINE:

JAVAPRO



www.javapro.io

www.javapro.io

Für Ihre Software Probleme brauchen Sie keinen Besserwisser.

Sie brauchen einen {BUDDY}

Richard Fichter
CEO @ XDEV

Veraltete Software? Steigende Wartungskosten? Sicherheitsrisiken? Neben modernen Tools bieten wir die passende Unterstützung für Ihre **Java-Modernisierung**. Wir helfen Ihnen, Ihre Java-Anwendungen fit für die Zukunft zu machen – mit klarem Konzept und auf Augenhöhe.

- **Kein Besserwisser – ein Buddy:** Wir begleiten Ihr Team mit Erfahrung, ohne den Klugscheißer zu spielen.
- **Modernisierung mit Strategie:** Agile Methoden & Proof of Concept für ein sicheres Update.
- **Robuste Lösungen:** Wir setzen auf bewährte Technologien und praxisnahe Konzepte – ohne unnötige Komplexität.

Lassen Sie uns Ihr Java-Projekt gemeinsam voranbringen!

BEST {BUDDYS} IN CODE

trusted by

Vereinbaren Sie hier einen unverbindlichen Termin!

XDEV