Special Edition 3 / 2025 1 | 2
#JAVAPRO

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

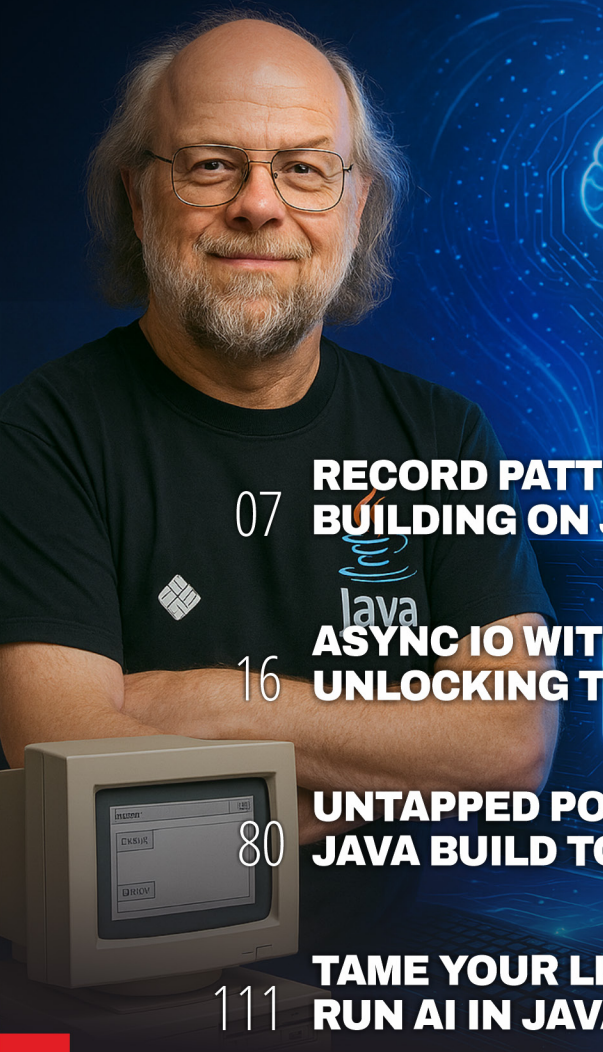
JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

30 YEARS OF JAVA

SPECIAL EDITION

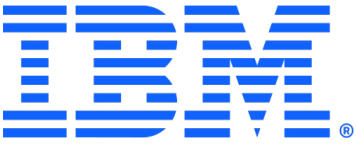
PART 3 1 | 2



With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors



Gold Sponsors

Silver Sponsors

Bronze Sponsors

JAVAPRO

Publisher:
JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
Germany

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

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

Editorial:
info@javapro.io

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

Javapro is an independent magazine and is not sponsored by Oracle Corporation.

Articles marked with a name do not necessarily reflect the opinion of the editors.

The images featured in this issue are sourced from royalty-free platforms such as Pixabay and Unsplash, contributed by our authors, or creatively generated with the help of artificial intelligence."

#JAVAPRO #EDITORIAL

30 Years of Java – Part 3 of an Ongoing Success Story

Three decades of Java prove that lasting success in technology comes from solid evolution, not fancy features. From its origins as an object-oriented language to its role in enterprise platforms and cloud-native development, Java has grown without losing its core promise: stability and portability. This third installment of our anniversary series highlights new directions in design, tooling, architecture, and use-cases practice that are shaping how Java is used today and tomorrow.

Patterns Rediscovered

Familiar concepts are reinterpreted in a modern context. Patterns gain new expression through records and sealed classes, while domain-driven design offers clarity for complex systems. Migration remains a hallmark - carrying applications across decades while adapting them to today's runtime environments and deployment models.

Breaking New Ground in Performance

Efficiency has become as important as scalability. Virtual threads simplify concurrency, while asynchronous I/O and serverless deployments extend Java into new domains. What once seemed limits are now opportunities: proof that the platform is not bound by its past but open to continuous transformation.

Tools and Teams Evolving Together

The developer experience is also shifting. Build tools are being reimagined for speed and usability, while modern testing strategies balance proven frameworks with smarter automation. AI increasingly supports reviews and quality checks, yet these advances also raise cultural questions: how teams collaborate in the future, how trust is built, and how agile practices can help or sometimes even hinder progress.

A Future Built on Renewal

The lesson after thirty years is clear: Java's vitality lies in its ability to renew itself. This third and final instalment in our anniversary series shows a platform that connects tradition with transformation, combining proven stability with the courage to embrace change. Java remains not only a language of the past, but one of the most relevant platforms for the future.

Welcome to more exciting episodes of the Java story!

Markus Kett
Editor in Chief
JAVAPRO

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

#JAVAPRO #COREJAVA

Record Patterns — Building on JAVA Records

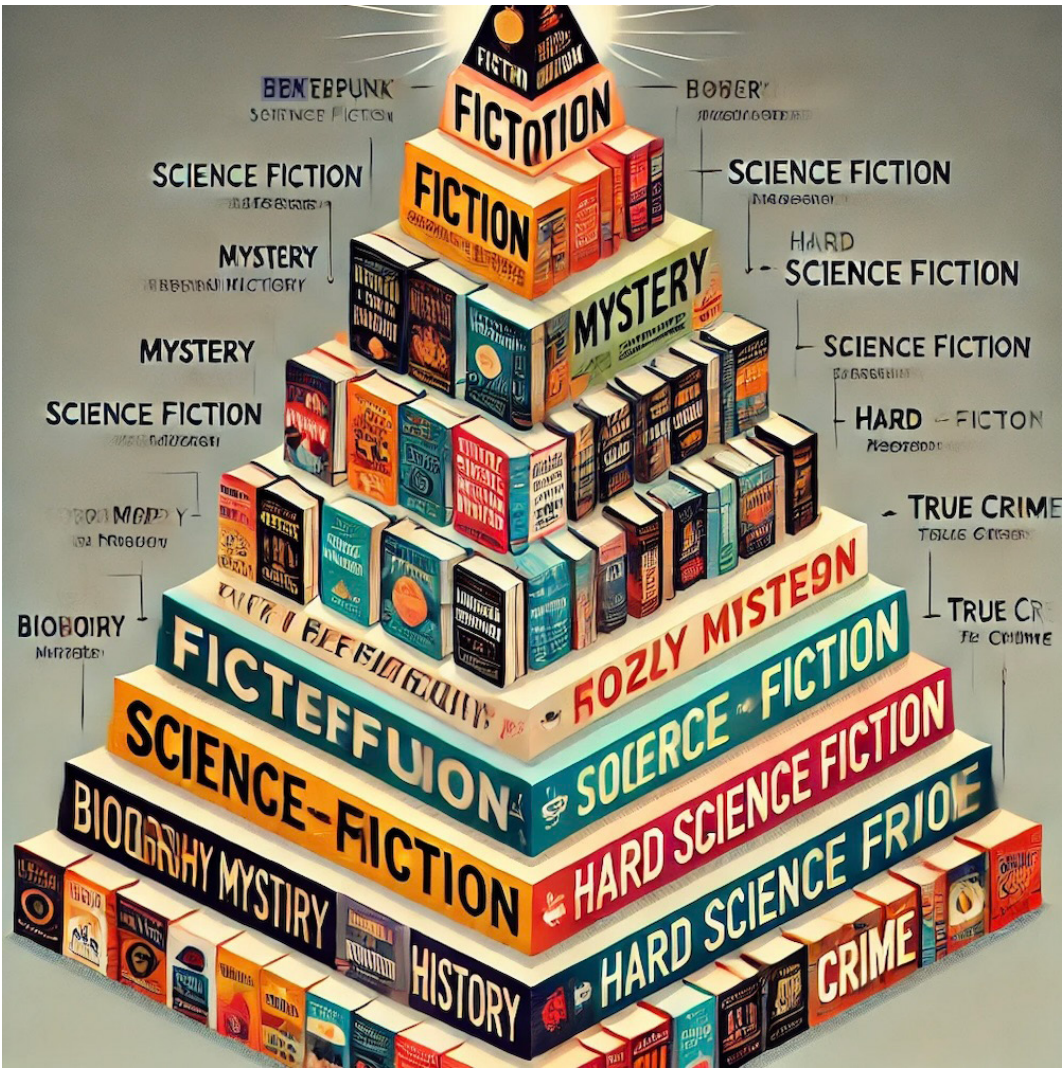
Author:

Manoj Nalledathu Palat is an Open Source Lead/ Committer working at IBM leading the Java Compiler in Eclipse (https://projects.eclipse.org/projects/eclipse.jdt/PL/mpalat) . In parallel, Represented Eclipse Foundation in the Experts Group for Platform JSRs for multiple Java SE [eg: https://openjdk.java.net/projects/jdk/21/spec/] and IBM in Eclipse IDE WG Steering Committee. Passionate about training (Worked as Professor of Practice from January 2024-Dec 2024 at NIT, Calicut), Provide training internally at IBM regularly, blogs sometimes (https://medium.com/@manojnp,), mentors people. Holds a Masters degree in Computer Science from Indian Institute of Science(IISc), Bangalore and B-Tech in Computer Science from National Institute of Technology (Formerly REC) Calicut, India

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

Records are becoming quite popular nowadays. Put it simply, they are just constant classes. So what's the big deal about them. Well, the actual power of records comes with record patterns. Let's delve into this interesting construct.

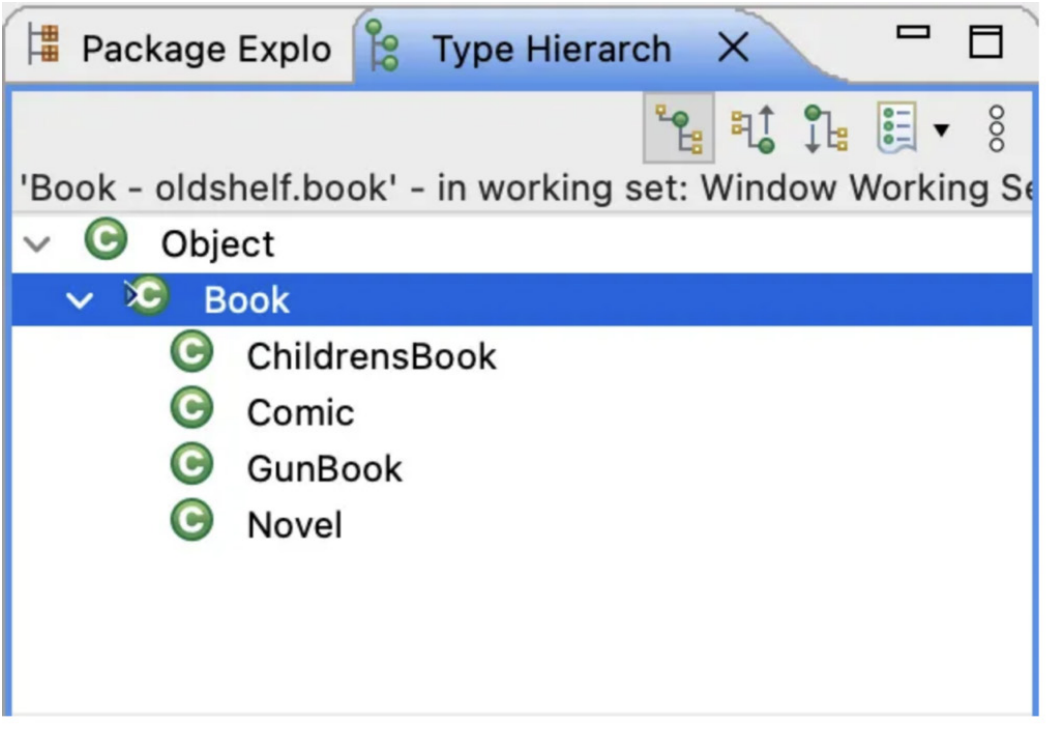
Let us say that we have a stack of books:



And we have been confronted with a task — Given a Book, select the book or do not select depending on the type and some conditions. In the old way, let us say the Book is described as a Class:

```
package oldshelf.book;


import oldshelf.author.Author;


public class Book {
 public Author author;
 public String title;


 Book(Author author, String title) {
  this.author = author;
  this.title = title;
 }
}
```

And let us say we have different kinds of Books - For simplicity's sake, let us assume that we have only the following categories of books — Moving to the technical representation here is a snapshot of the Type Hierarchy:



Different kinds of books

For the completeness' sake, let us look at the definitions of each:

```
package oldshelf.book;

import oldshelf.author.Author;

public class ChildrensBook extends Book {

 ChildrensBook(Author author, String title) {
  super(author, title);
  // TODO Auto-generated constructor stub
 }

 public int uptoAge;
}
```

```
package oldshelf.book;

import oldshelf.author.Author;

public class Comic extends Book {

 String nameOfMainCharacter;
 public int fromAge;

 Comic(Author author, String title) {
  super(author, title);
  // TODO Auto-generated constructor stub
 }
}
```

```
package oldshelf.book;

import oldshelf.author.Author;

public class GunBook extends Book {

 GunBook(Author author, String title) {
  super(author, title);
  // TODO Auto-generated constructor stub
 }
```

```
}
```

```
package oldshelf.book;

import oldshelf.author.Author;

public class Novel extends Book {

 Novel(Author author, String title) {
  super(author, title);
  // TODO Auto-generated constructor stub
 }

 String yearOfPublication;

}
```

Now, I must have lost you already. Anyway, for those who remain, let us see our selection criteria:

```
package oldshelf;

import oldshelf.book.*;

public class MySelection {

 public boolean isBookSuitableForAge(Object o, int age) {
  if (!(o instanceof Book))
    return false;

  if (o instanceof Comic) {
   Comic comic = (Comic) o;
   return age >= comic.fromAge;
  }

  if (o instanceof ChildrensBook) {
   ChildrensBook cb = (ChildrensBook) o;
   return age <= cb.uptoAge;
  }
```

```
  if (o instanceof GunBook) {
   return false;
  }
  return age > 18;
 }
}
```

No, I don't want you to go through each and every line of code, but just want to highlight the following points which we do for a Book:

- if its not a book, its of course false
- if its a comic book, cast, then we look whether the age is greater than the field fromAge
- if its a childrensBook, cast, then check for less than..
- if its GunBook, no
- default is to check for age>18.

Essentially, the point is that the condition is different for each type but the operations have similarities in different ways. And definitely there is a priority in terms of the control flow.

Disclaimer here: Let us see what we can do with Records. The design is suboptimal and just pedagogic (academic), a more acceptable design would be sealed and permits — but I don't want to use those restricted identifiers and the concepts here.

So we create a Book in the new shelf as :

```
package newshelf.book;
import newshelf.author.*;

record BookInfo(Author author, String title) {}

record Novel(BookInfo info, String genre) {}
record GunBook(BookInfo info) {}
record ChildrensBook(BookInfo info, int uptoAge) {}
record Comic(BookInfo info, int fromAge) {}
```

```
class Book {

 @SuppressWarnings(„preview“)
 public boolean isBookSuitableForAge(Object o, int age) {

  return switch (o) {
    case Comic(var info, var fromAge) -> age >= fromAge;
    case ChildrensBook(BookInfo info, int uptoAge) -> age <= uptoAge;
    case GunBook(var inf) -> false;
    default -> age > 18;
  };
 }

}
```

There are just records here. We can see that across the different types of Books, the first component is a BookInfo which itself is another record. Interesting part is this switch:

```
return switch (o) {
    case Comic(var info, var fromAge) -> age >= fromAge;
    case ChildrensBook(BookInfo info, int uptoAge) -> age <= uptoAge;
    case GunBook(var inf) -> false;
    default -> age > 18;
  };
```

This is the Switch Pattern. Switch patterns are available rom Java 21 onwards) take a type as an expression to switch over and can have case statements that take a type in the Case arm or the case LHS. Let us look at the Case arm in detail, say the first case statement:

```
case Comic(var info, var fromAge) -> age >= fromAge;
```

Are you familiar with this syntax? If not its, ok. Here, if we think logically, it would mean if the Object o is a type of Comic, then, this arm should be taken.. but wait a second, why should we specify parameters in the Type?

Well, this is a record, and a record is identified by the Name and Type — its kind of a cross between a type and a method.. we can have a different record with the same name with different parameter types.

ok. so we understand that.. but the RHS doesn't make much sense.. We've just given some variable name on the LHS and we are just using that on RHS. How's that possible?

Under the hood, the compiler will extract these variables, or let's say would deconstruct this variable fromAge and it will check whether age is less than fromAge of that record and will return accordingly. This is a deconstruction pattern.

Ok, but a question lingers? what happens if its not a Comic? Since its "->" this is essentially equal to a break after the Case? How does this go to the next line to check the next case?

Well, that's done by the compiler for you. There is an implicit loop which checks one by one, whether which type you are switching on and whether the type matches? Why one by one? Because a subtype can come earlier than a supertype and the match should be as close as possible. Well, that stuff is for Switch Patterns, let us get back to record patterns.

One more quick question — I just define var fromAge, and then the type is found out by the compiler, Is that correct? yes, the compiler does that work for you.

So now, with the use of records in Case, we are able to extract the extract variable or component of the record. and off you go..

Great! so we are going to have support for Selection? What about if we have lot of Books? Is there any record support in looping? Yes and no — in 20, we had a preview where the foreach construct was supporting Records, but now its gone in 21 since a few issues need to be ironed out — It will come at a later stage.

Before we conclude, lets look at the usage of Record patterns in another construct, a simpler one this time — The "instanceof".

```
public String getAuthor(Object o) {
  if (o instanceof BookInfo(
              Author(Name(var f, var lastName), var dob, String Na
               tionality), String title)) {
     return lastName;
  }
  return null;
}
```

We can see that, we can use the Record patterns to multiple nesting level and still get the lastname?

How does the compiler do it?

Internally, the compiler unwinds each of them.. ie generates code to unwind each of these and then creates the variables and populates the values using the record accessors. All behind the scenes!

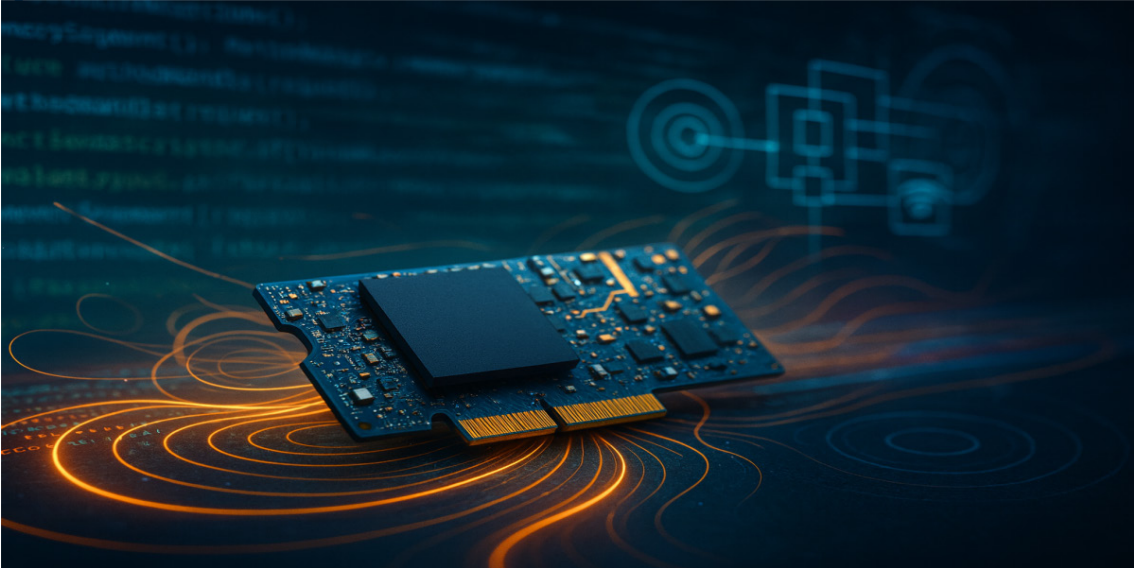
Great! So now we can inspect one Book. Is there anything coming for multiple books? Loops?

Yes, this is also on the cards — We had a glimpse of Record Patterns for ForEach in Java 20 as preview but that got shelved and we can expect a more, technically sound version to show its head soon.

And before winding up, one last point — Here I was interested only in lastName, but I have to give all the components — That's not minimalistic. Well, the underscore is available from Java 22 onwards — just use _ and you don't have to bother about variable you don't care..

Happy Pattern Creation with Record Patterns!

> Back to Table of Content



#JAVAPRO #COREJAVA

Async IO with Java and Panama: Unlocking the Power of IO_uring

Author:

David Vlijmincx is a senior Java developer with 8+ years of experience, Oracle Ace, author, blogger, and conference speaker with a passion for Java development.

https://www.linkedin.com/in/david-vlijmincx

When I first started exploring Virtual Threads in Java, I wanted to understand everything about them—their yielding behavior, performance characteristics, and limitations. This journey led me to an interesting challenge: file I/O operations cause Virtual Threads to become „pinned" to platform threads, limiting their effectiveness for I/O-heavy applications.

This pinning issue has been widely acknowledged across the Java community. It's mentioned in the Virtual Threads JEP, discussed on Reddit threads, debated on mailing lists, and highlighted in the „State of Loom". The consistent message: File I/O and Virtual Threads don't play nicely together.

The consensus is that this shouldn't be a problem because disk access

is „fast enough." But is it? When accessing remote filesystems or handling thousands of concurrent I/O operations, those milliseconds add up quickly. Java's inability to differentiate between local and remote filesystems creates an opportunity for improvement.

I'll share my expedition into solving this challenge by combining two powerful technologies: Project Panama for native interoperability and Linux's io_uring for high-performance asynchronous I/O. I'll walk through how I built a bridge between Java's Virtual Threads and io_uring, transforming blocking I/O operations into yielding ones.

If you're building high-throughput web servers, data processing pipelines, or any application with intensive I/O requirements, this approach might be the missing piece in fully realizing the potential of Virtual Threads.

What is Panama

Project Panama is Java's modern approach to native interoperability, released as a standard feature in Java 22. It offers an alternative to JNI with a more elegant and safe API for interacting with native code and memory. Panama's API provides the developers with the performance benefits of native code while maintaining Java's safety and usability characteristics.

At its core, Panama solves a fundamental problem that Java developers have faced for decades: how to efficiently interact with code and data outside the Java runtime while maintaining Java's safety guarantees. Before Panama, developers relied on JNI, which required writing C/C++ code, compiling shared libraries, and writing verbose Java code to load and call these libraries. JNI development was error-prone and difficult to debug, creating a significant cognitive disconnect from Java's usual programming model.

For applications that need to leverage native libraries or system capabilities, Panama transforms what was a dreaded necessity into a natural extension of Java development. This transformation is especially relevant for domains like high-performance compting, machine learning, and the focus of our exploration of high-performance I/O operations. Let's break down Panama's capabilities into its core components: memory

management and function calling, to understand how they enable our io_uring integration.

Managing Memory

Memory management represents one of the most significant challenges when interacting with native code. Panama addresses this through an approach centered around the Arenas.

For Java applications, you rarely have to think about memory management thanks to the garbage collector. When working with native code, however, memory must be explicitly allocated and freed. This manual management creates numerous opportunities for errors like memory leaks or accessing freed memory.

Panama's Arena API provides deterministic resource management for off-heap memory. When an Arena closes, all memory segments allocated within it are automatically deallocated. This approach aligns with Java's resource management patterns, making it intuitive for Java developers. All with a simple try-with-resource statement.

Different Arena implementations accommodate various usage patterns. The confined Arena ensures single-thread access. The shared Arena allows multiple threads to access memory segments. The global Arena addresses cases where memory segments need to outlive their creating threads. The Auto Arena uses the Garbage collector to deallocate segments. Creating an Arena can be done as follows. The Arena is active inside the try-with-resource statement. This means the MemorySegments created inside of it are only available inside that scope and will be freed when the arena is closed.

```
try(Arena arena = Arena.ofConfined()){
    MemorySegment memorySegment = arena.allocate(1024);
}
```

MemorySegment serves as Panama's primary abstraction for working with memory, representing a contiguous region with well-defined boundaries and lifecycle. Unlike direct ByteBuffers, MemorySegments are explicitly bound to an Arena, creating a clear ownership model.

For structured data, Panama provides the MemoryLayout API to describe complex memory organizations including structs, arrays, and unions. This enables safe manipulation of complex data structures without resorting to error-prone pointer arithmetic. Like is done in the following example. Here a StructLayout is created with different kinds of values. The values can be accessed using a VarHandle.

```
AddressLayout C_POINTER = ValueLayout.ADDRESS
        .withTargetLayout(MemoryLayout.sequenceLayout(Long.MAX_VALUE,
        JAVA_BYTE));

StructLayout requestLayout = MemoryLayout.structLayout(
                ValueLayout.JAVA_LONG.withName(„id“),
                C_POINTER.withName(„buffer“),
                ValueLayout.JAVA_INT.withName(„fd“),
                ValueLayout.JAVA_BOOLEAN.withName(„read“))
        .withName(„request“);

VarHandle idHandle = requestLayout.varHandle
(MemoryLayout.PathElement.groupElement(„id“));
```

The integration of these concepts creates an approach to memory management that maintains Java's safety guarantees while providing the flexibility needed for native interoperability.

Making Calls Up and Down

Panama's approach to function calls builds upon Java's existing metaprogramming features like Method handles to create a bridge between Java methods and native functions. The API distinguishes between downcalls (Java calling native functions) and upcalls (native code calling back into Java methods).

For downcalls, Panama's Linker API creates method handles that invoke native functions. This process begins with obtaining a SymbolLookup to locate native symbols within libraries. Once found, the Linker transforms these symbols into method handles using FunctionDescriptors that

specify parameter and return types.

```
MethodHandle malloc = linker.downcallHandle(
        linker.defaultLookup().find(„malloc“).orElseThrow(),
        FunctionDescriptor.of(ADDRESS, JAVA_LONG)
);

malloc.invokeExact(size);;
```

This description-based approach eliminates the verbose boilerplate associated with JNI. Instead of writing C code to bridge between Java and native functions, developers describe the function signature using Java code. This not only reduces error potential but also enables better tooling support. You will get an exception if you call the method with incorrect typing.

Upcalls follow a similar pattern but in reverse, with Java methods wrapped as function pointers that native code can invoke. This capability is particularly valuable when working with callback-based APIs, where native code needs to notify Java about events or completion status. In the following example, a down call and an up call are made for the native signal method.

```
public static void main(String[] args) throws Throwable {

    final var linker = Linker.nativeLinker();

    // up call
    // describes the Java method
    MethodHandle handleSignal = MethodHandles.lookup()
            .findStatic(UpCallExample.class,
                    „handleSignal“,
                    MethodType.methodType(void.class, int.class));
    // Signature of the java method
    FunctionDescriptor signalDescriptor = FunctionDescriptor.ofVo-
id(ValueLayout.JAVA_INT);

    // The stub to be passed to the native code
    MemorySegment handlerFunc = linker.upcallStub(handleSignal,
```

```
            signalDescriptor,
            Arena.ofAuto());

    // down call
    MethodHandle signal = linker.downcallHandle(
            linker.defaultLookup().find(„signal“).orElseThrow(),
            FunctionDescriptor.ofVoid(JAVA_INT, ADDRESS)
    );

    signal.invoke(2, handlerFunc);
}

static void handleSignal(int signal) {
    System.out.println(„Received signal: „ + signal);
}
```

Both upcalls and downcalls benefit from Panama's unified type mapping system, which automatically handles conversion between Java and native types. When passing complex data structures to native functions, developers can use MemoryLayout to describe the structure and MemorySegment to allocate appropriate memory. What is great about this approach is that it is a complete shift from JNI. Rather than treating native code as a separate environment with its own rules, Panama extends Java's programming model to more natural interaction with native code. Now that we understand how Panama enables safe and efficient native interoperability, let's examine how it can be used with io_uring.

What is IO_uring

IO_uring is a Linux kernel interface that improves how applications perform input/output operations. At its core, it establishes a shared memory communication channel between applications and the kernel using two ring buffers: one for submitting requests and another for receiving completions.

Applications submit multiple I/O operations (in batches) to the submission ring without making individual system calls for each operation. Applications can continue executing other tasks while I/O operations complete asynchronously, with results delivered through the

completion queue.

It addresses the performance limitations of traditional I/O methods by eliminating per-operation system calls and context switches, enabling true asynchronous operation for all types of I/O, and dramatically reducing overhead under high concurrency. This makes it particularly valuable for applications that need to handle thousands of concurrent I/O operations efficiently.

The relationship between io_uring and Virtual Threads is particularly interesting. Virtual Threads excel at managing concurrent tasks but can become pinned during blocking I/O operations. IO_uring's asynchronous nature potentially addresses this limitation, allowing Virtual Threads to yield during I/O operations rather than remaining pinned to carrier threads. To make these concepts concrete, let's implement a basic file read operation using io_uring through Panama. This example will demonstrate the essential pattern that supports more complex I/O operations.

Single Read with Java and Uring

Implementing a basic file read operation with io_uring through Panama illustrates how these technologies work together to create an alternative for Java's I/O capabilities. The following application creates a bridge between Java and io_uring native library.

The implementation involves initializing an io_uring instance, preparing a read request, submitting it to the kernel, and checking for completion. Throughout this process, Panama provides memory management for buffers, function calling, and type mapping between Java and native structures.

The following pattern demonstrates how we bridge Java and io_uring through Panama to perform a read operation. First, we initialize the io_uring instance and allocate memory for our buffer within a confined Arena to ensure proper resource management. We then prepare a Submission Queue Entry (SQE) that describes our read operation, including the file descriptor, buffer location, size, and offset. After submitting this request to io_uring with io_uring_submit, we wait for the operation to complete

using io_uring_wait_cqe. Once completed, we can extract the result from the Completion Queue Entry (CQE) and process the data that was read into our buffer. Finally, we mark the completion as seen and let the Arena's automatic cleanup handle resource deallocation. This basic pattern forms the foundation for more sophisticated I/O operations while maintaining both Java's safety guarantees and io_uring's performance benefits.

```
try (Arena arena = Arena.ofConfined()) {
    // Create and initialize the ring
    MemorySegment ring = arena.allocate(ring_layout);
    int ret = (int) io_uring_queue_init.invokeExact(10, ring, 0);

    // Prepare file descriptor and buffer
    int fd = (int) open.invokeExact(MemorySegment.ofArray(„red_file“.
    getBytes()), 0, 0);
    MemorySegment buffer = arena.allocate(1024);

    // Prepare read operation
    MemorySegment sqe = (MemorySegment) io_uring_get_sqe.
    invokeExact(ring);
    io_uring_prep_read.invokeExact(sqe, fd, buffer, buffer.
    byteSize(), 0L);

    // Submit to uring
    ret = (int) io_uring_submit.invokeExact(ring);

    // Wait for and process completion
    MemorySegment cqePtr = arena.allocate(ValueLayout.ADDRESS);
    ret = (int) io_uring_wait_cqe.invokeExact(ring, cqePtr);

    // print the result
    System.out.println(buffer.getString(0));

    // Process data and cleanup
    io_uring_cqe_seen.invokeExact(ring, cqePtr);
}
```

For applications that perform numerous read operations, such as web servers, databases, or file-processing utilities, this integration delivers significant performance improvements by reducing system call overhead and enabling true asynchronous I/O. Another benefit is that io_uring supports sockets, streamlining the process of reading data and sending it to clients without unnecessary data copying. While this basic implementation works, we can further optimize it by examining some of the performance characteristics and challenges when bridging Java and native code.

Performance Improvements

The combination of Panama, io_uring, and Virtual Threads creates new opportunities for Java applications with I/O-intensive workloads. Using a native library on its own can be beneficial and provide performance improvements. To get more speed out of the bindings we need to take a closer look at three patterns for memory allocation. The first speed-up is for applications that do a lot of allocations.

When an Arena allocates memory using the allocateSegment function you get a continuous region of memory that is filled with zeroes. This is a safe segment of memory the application can use. However, the actual creation of a segment is quite an expensive operation compared to using malloc or calloc. These two native methods are a lot quicker when you need a continuous region of memory. In the following output, you can see how much faster these native methods are when you need to allocate some memory.

```
A higher score is better
Benchmark                (memory size)   Mode      Score   Units
MemoryBenchmark.arena               64  thrpt  23759.714  ops/ms
MemoryBenchmark.arena              128  thrpt  24430.636  ops/ms
MemoryBenchmark.arena              512  thrpt  18927.759  ops/ms
MemoryBenchmark.arena             1024  thrpt  15068.157  ops/ms
MemoryBenchmark.arena             2048  thrpt   6657.966  ops/ms

MemoryBenchmark.calloc              64  thrpt  44291.364  ops/ms
MemoryBenchmark.calloc             128  thrpt  25280.973  ops/ms
MemoryBenchmark.calloc             512  thrpt  20667.819  ops/ms
```

```
MemoryBenchmark.calloc          1024  thrpt  22050.309  ops/ms
MemoryBenchmark.calloc          2048  thrpt  19291.683  ops/ms

MemoryBenchmark.malloc            64  thrpt  82383.964  ops/ms
MemoryBenchmark.malloc           128  thrpt  82172.938  ops/ms
MemoryBenchmark.malloc           512  thrpt  86222.730  ops/ms
MemoryBenchmark.malloc          1024  thrpt  85490.634  ops/ms
MemoryBenchmark.malloc          2048  thrpt  29076.375  ops/ms
```

As you can see from the output generated by JMH benchmarks using native methods can allocate memory faster than an Arena can. If your use case requires the application to allocate lots of memory segments it can be beneficial to switch to native methods.

There are two ways to use this native memory. We can tie its lifetime to an arena, or you can manage the lifetime of these segments yourself and free them when necessary. The first option gives you more safety while the latter gives you more control over resources. Calling free() is easy enough, so let's explore how we can use malloc and calloc in a safe way using the Arenas that project Panama provides us.

```
public class AllocArena implements Arena {

    private static final MethodHandle malloc;
    private static final MethodHandle calloc;
    private static final MethodHandle free;

    static {
        Linker linker = Linker.nativeLinker();

        malloc = linker.downcallHandle(linker.defaultLookup().
                find(„malloc“).orElseThrow(),
               FunctionDescriptor.of(ADDRESS, JAVA_LONG));
        free = linker.downcallHandle(linker.defaultLookup().
                find(„free“).orElseThrow(),
                 FunctionDescriptor.ofVoid(ADDRESS));
    }
```

```
    final Arena arena = Arena.ofConfined();

    @Override
    public MemorySegment allocate(long byteSize, long byteAlignment)
    {
        return ((MemorySegment) malloc.invokeExact(size)).
        reinterpret(size, arena, m -> free(m));
    }

    @Override
    public MemorySegment.Scope scope() {
        return arena.scope();
    }

    @Override
    public void close() {
        arena.close();
    }
}
```

The arena overrides the allocation method. Doing so we can use our own code that uses malloc or calloc to allocate a memory segment. Calloc and malloc return a pointer to a contiguous region of memory. A MemorySegment that is a pointer has a length of zero, so we can't use it directly. To make it usable we have to call reinterpret which takes as a parameter the new size, and optionally an arena and cleaning method. From the client's perspective, this arena doesn't look or behave differently, just the allocation happens using malloc.

Using Native memory is not the only way to pass values to native code. There is also the option to pass the address of heap-backed memory to native code. The benefit of doing so is that it saves you from doing a memory allocation but also having to manage that segment's lifetime. All of this responsibility is handed to Java. To be able to pass heap-backed memory you need to mark a method as critical using the Linker. This option is only meant for calls that have a very short lifetime. It's also not meant to be used to create callbacks. What happens behind the scenes is that Java will create a temporary address for that value and pass it to the native code. In the following section, you see how to mark

methods as critical and use them.

```
open = linker.downcallHandle(
        linker.defaultLookup().find(„open“).orElseThrow(),
        FunctionDescriptor.of(JAVA_INT, ADDRESS, JAVA_INT, JAVA_INT),
        Linker.Option.critical(true));

open.invokeExact(MemorySegment.ofArray((filePath).getBytes()), flags,
mode);
```

Using Linker.Option.critical(true) enables the application to use heap memory. The invokeExact method creates a heap-backed memorySegment of a String. This should really only be used for methods that perform worse by making a copy of the data first.

There are several ways to speed up memory segment allocation but the best way is to reuse memory that you have already allocated. The risk is accessing previous values, if you are not careful. The benefit is the almost free performance improvement. Allocating memory will always have a cost and so will be turning that memory into a memorySegment. This can all be prevented by reusing segments. The easiest/safest way is to reuse memory that holds pointers, this is an easy way to start and learn how to do it safely. In the end, it looks almost the same as Java. Beyond memory optimization, our primary goal was addressing the Virtual Thread pinning issue. Let's explore how we can transform blocking operations into yielding operations to preserve how Virtual Thread work.

Turning Pinning Into Yielding

Virtual Threads face a significant limitation with file I/O operations: pinning. When a Virtual Thread performs blocking operations, including file I/O, it becomes „pinned" to its carrier thread, preventing that carrier thread from executing other Virtual Threads. This pinning effectively negates one of the primary benefits of the Virtual Thread model. More carrier threads get created if this happens, so prevent a degradation in performance. Making native calls is not great either as those will pin the virtual thread as well. So let's see how we can prevent pinning in the first place.

The trick to prevent pinning is to yield a virtual thread before making a pinning call. The mechanism needed looks familiar to what happens inside Java. Ask yourself the question: how does a virtual thread know when to wake up? The answer is another thread wakes it up. To get the same behavior we can recreate it with a polling thread and futures. The future will block the thread till the polling threads wake it up with a result. The next example shows you the essence of what is needed for this behavior.

```
public final class BlockingReadResult {

    private final CompletableFuture<Void> future = new
    CompletableFuture<>();
    private MemorySegment buffer;

    BlockingReadResult(long id) {
        super(id);
    }

    public MemorySegment getBuffer() {
        try {
            future.get();
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }
        return buffer;
    }

    // code continues
```

The trick is to have it wait till a result is available. One way of doing so is to use CompletableFuture.get(). The virtual
thread will yield till a result is available. The polling thread that checks if values are available looks like the following example. It's a while loop that will set the result of the CompletableFuture allowing the virtual thread to continue.

```
while (running) {
    final List<Result> results = jUring.peekForBatchResult(100);
```

```
        results.forEach(result -> {
            BlockingResult request = requests.remove(result.getId());
            request.setResult(result);
        });
    }
```

The approach is pretty straightforward and allows you to yield virtua threads for operations that do not yield them. The cost of doing this is that you will need a polling thread. If this is beneficial to you application depends on the workload.

Bringing It All Together

We explored how Panama, io_uring, and Virtual Threads can fundamentally improve Java's approach to I/O operations. By integrating io_uring using Panama, Java applications can leverage Linux's I/O capabilities while maintaining Java's safety and productivity benefits.

This implementation exercise also reveals two distinct approaches to native integration. You can create direct bindings where Java methods map directly to native calls, resulting in a less Java-like but more straightforward API. Alternatively, you can build a facade around the native library, creating a more idiomatic Java API at the cost of additional abstraction.

For most production applications, I recommend the facade approach. It gives you more control over the API and creates a more familiar experience for Java developers. The slight performance overhead is usually worth the improved maintainability and safety.

While this solution focuses on file I/O, the same approach can be applied to other blocking operations that cause Virtual thread to pin. Network I/O, database access, and inter-process communication are all candidates for similar optimization.

As Panama and Virtual Threads continue to mature in future Java releases, we can expect even tighter integration between these technologies, potentially making solutions like this part of the standard library. Until then, this approach offersa powerful way to unlock the full potential of Virtual Threads for I/O-intensive applications.

#JAVAPRO #COREJAVA

A Visual Chronicle of the JDK's Journey

Author:

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

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

The Java Development Kit (JDK) has undergone significant transformations since its inception. From it's „write-once, run everywhere" beginnings, over the applet wars, through the first release of the OpenJDK to the new release cadence, that has kept us occupied for the last eight years already. In 30 years a lot has happened.

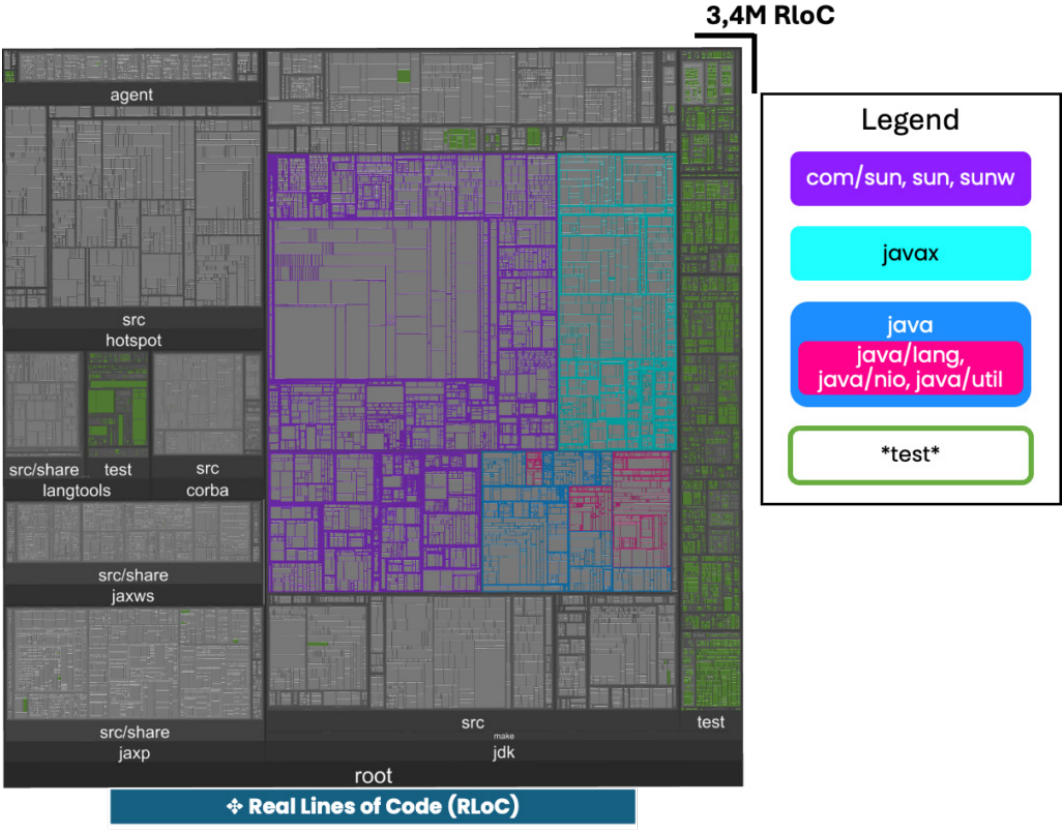
Most of that journey is chronicled only in, arguably, dry textual representation. The text tells us how many features were added, changed or, very seldomly, removed. It does not tell us what impact that had on the code. What were the key architectural changes, what were the refinements and how were the complexities of such a large code base handled?

We can visualise this journey by mapping the JDK. There are multiple tools that can do so. For this article we are using CodeCharta because it is free, open-source, and maintained since 2017. I am also very familiar with it, because I helped develop it.

The code base we'll analyse is the OpenJDK. It is the reference implementation of Java SE since version 10 (2018) and is available on GitHub since version 16 (2021). The first OpenJDK was version 7 (2011) but we can go back quite a bit further, since the first commit was December 1st 2007. We cannot however go back to JDK 1 (1996). In addition I am not a member or affiliated with the developers of the JDK. Most insights are based on dry textual representation, educated guesses and a nice visualisation.

JDK 7

With CodeCharta we can visualise the JDK as a 3D city map. Each file turns into building. The size of the building (✣) represents the real lines of code (RloC), that is the lines that are not comments or whitespace and that we actually have to read to understand the code. In the following map packages that belong to sun are marked in purple, java standard extensions (such as Swing) are marked in cyan while standard java is marked in blue. The red taint is given to commonly used java.lang, java.nio and java.util packages for size comparison. Additionally we are only colouring files whose names include the keyword *test*.



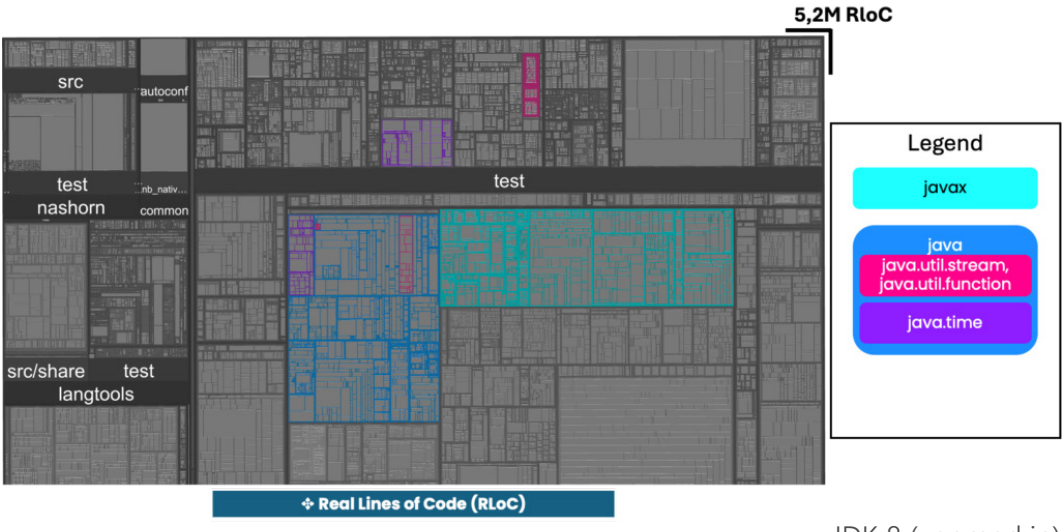
JDK 7 (filtered for *test*)

We can already see that even in 2011 Java is massive with 3,4 million lines. At the same time it is probably also not very well covered with tests, as the jdk/test package contains only 283K lines, or 12% of the total code. Later JDK versions will show us a much higher percentage. Additionally the sun-specific code is quite large and its presence is creating problems for the JDK people to this day. Finally we can also see a folder labelled hotspot. This is the java virtual machine (JVM) that executes the java code and fulfils the promise „write once, run anywhere". It is written mostly in C++ and has been the default VM since version 1.3 in 2000.

JDK 7 was supposed to contain two massive changes: anonymous functions, aka lambdas, as well as a modularised structure. But in 2010 a plan B was formed to release a smaller JDK 7 first. A JDK that is rather light on features. Which is why the map only focuses on the rough structure.

JDK 8

JDK 8, released in 2014, has jumped significantly in size to 5,2 million lines. We have zoomed in a bit to take a look at two new added features.



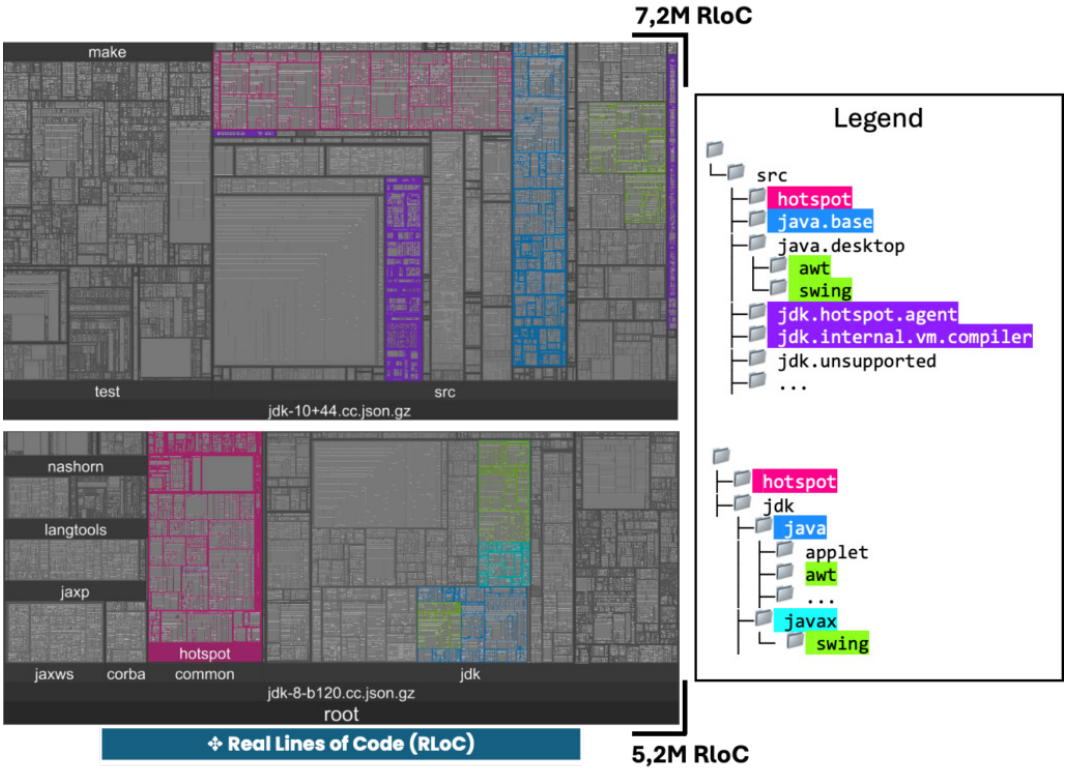
JDK 8 (zoomed in)

Streams (10k lines, red tint), the functional interfaces necessary for anonymous functions (373 lines, tiny red tint to the right of the purple block) and java.time (19k lines). Interestingly the tests for streams have a good 1:1 relation (11k lines) but java.time requires a lot of tests (45k lines). To me this just shows how hard dealing with time actually is and how many cases the tests have to cover.

These features also show the limits of our analysis capabilities. Adding anonymous functions to the JDK involved much more than writing 373 lines of functional interfaces but the required changes are much more widespread and not discernible without first-hand knowledge.

JDK 9+10

JDK 9, released September 2017, grew again and reached 7,3M lines. It's most ambitious feature was that the JDK now brought JPMS (Java Platform Module System) and the JDK was modularised with it. This modularisation changed the structure completely. Some restructuring, particularly those concerning the hotspot VM, were not completed with version 9 but left to the next version. Perhaps appropriate as 10 was also the version that had a particular enhancement: „consolidate the JDK forest into a single repository". Since the modularisation structure is stable with JDK 10 (March 2018) we'll target it with our next map.



JDK 8+10 with respective folder structure

At 7,2 million lines, version 10 is actually a bit smaller than 9 but still 2 million lines larger than 8. Most of that growth is supported with tests though. They now amount to 29% of the code base. We can see this relation clearly because tests have become a top-level concern.

The src folder is even more interesting than the test growth however. What was once a large monolith has been split into 20 java modules and 37 jdk modules. These modules have received a purpose-based grouping. UI-frameworks like AWT and Swing have been grouped into a java.desktop module. A java.base module exist that contains just the code that all other modules rely upon. It prominently contains collections, math, time and I/O (Input/Output).

Even jdk-internal concerns have received their own modules. The Nashorn JavaScript engine (yes this JDK can execute ECMAScript 5.1 code) is one such module. The Java code that hotspot uses was also given modules such as jdk.hotspot or jdk.internal.vm.compiler. Previously those were located in the hotspot package. The remaining 738k lines in the hotspot package are written in C++.

But that is just how the code was structured. The utilised module system provided another, much larger benefit. The internals of all

the new modules were no longer accessible from the outside. All the module developers had to guarantee was that the exposed API of each module remained stable, leaving them free to change all the internal implementation details without worrying it might break someones code.

Some code was not hidden to the public though, even though the JDK maintainers would have liked it. One file in particular enables writing performance-critical but unsafe code. The appropriately named class sun.misc.Unsafe is to this day part of the JDK. Replacing it with viable but safe alternatives has been a challenge that is still not complete. 79 out of 87 methods are finally deprecated with JDK 23 (September 2023) though. Finally removing Unsafe is part of an ongoing effort the developers are calling „Integrity by default". The idea is that „developers expect that their code and data is protected against use that is unwanted or unwise."

JDK 11

JDK 11 was released September 2018. It and its two predecessors are part of a new release cadence where a new JDK is released every six months. Instead of tying JDKs to a specific feature set, and having endless meetings about key feature progress or lack thereof, the maintainers are now following a release-train model. Every six months the train leaves. The features that are ready get released. If your feature is not ready then do not stress about it. You have another train six months later.

With the release of a new JDK, the previous version won't receive any support any more. But upgrading the JDK every six months is not something large enterprises want. They want their stable feature set that only gets security or bug patches, not risky new features. This is where so-called long-term support (LTS) releases come in. Every couple of years, the JDK releases a new JDK that it promises to support for years with fixes but without adding features. The JDK maintainers call this the tip & tail model and they suggest library maintainers should follow it as well.

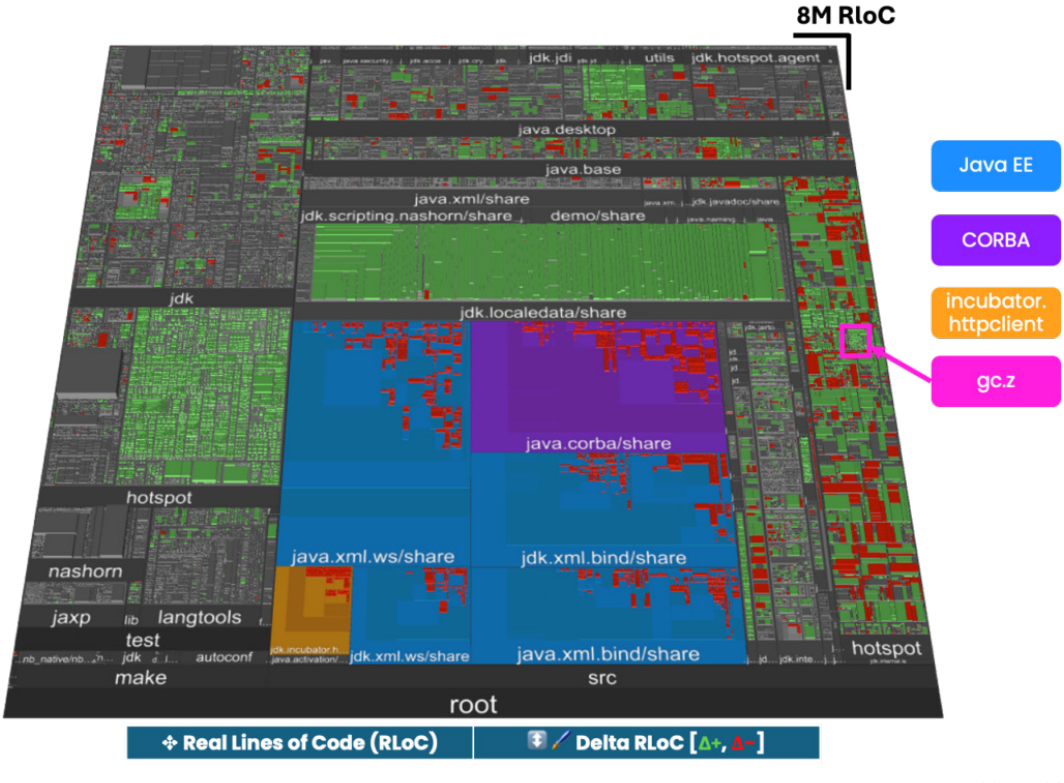
JDK 11 is such a LTS release. The free support from Oracle ended September 2023, five years after it was released (other vendors such as Adoptium still provide free support though). This gives enterprises plenty of time to use the stable LTS-version before having to upgrade to

the next version.

From a feature perspective, LTS releases are not treated differently to the short-term releases. Both are governed by the JDK Enhancement-Proposal (JEP). Such enhancement do not have to be final, they can also be:

- Experimental: Test-bed mechanism used to gather feedback on nontrivial HotSpot enhancements. Experimental JEPs needto be enabled individually via JVM options -XX:+ UnlockExperimentalVMOptions
- Incubating: Non-final API or tool, intended to gather early feedback. Incubating JEPs need to be added individually via JVM options --enable-preview --add-modules jdk.incubator.xyz
- Preview: API or tool intended to be final, still want to gather feedback. Preview JEPs are not enabled individually but all or nothing via JVM option --enable-preview.

Since LTS releases are not treated differently from non-LTS releases, they can also contain non-final enhancements. Such is the case for version 11, which contains an experimental garbage collector called ZGC (Z for short). The following map shows the new garbage collector but also some code removals. A JEP does not have to be a new API or tool, it can also be the removal of one.



JDK 11 vs 10

This map compares JDK 10 with JDK 11. Green buildings show files that have more lines than in version 10, red buildings have less lines. At 8 million lines version 11 has increased again. The tests now amount to 36% of the code base. Because we zoomed in a bit and gained more floor labels, we can now also see that the tests are not just one big ball of tests. Like the src folder, the tests are split into folders depending on if they target the jdk, hotspot, nashorn etc. The hotspot tests have gotten a huge boost with a lot of files added (the vmTestbase folder).

The jdk.localedata has also seen a huge boost. It contains the xml files of the Unicode Common Locale Data Repository (CLDR). The CLDR „provides key building blocks for software to support the world's languages, with the largest and most extensive standard repository of locale data available". The JDK uses it „to format dates, times, currencies, languages, countries, and time zones in the standard Java APIs". The jdk.localedata takes up quite a lot of space with its 1,1 million lines. But since those lines are mostly xml-files, we'll generally exclude it in future maps to focus on code changes.

In the map we can also see that some modules have not gotten a boost but were instead removed. JEP 320 delivered the removal of Java EE and CORBA modules. They were removed because maintaining them was no longer worth the effort compared to how little the features were used. In total this means the developers no longer have to maintain 357k lines.

One thing that was only seemingly removed was the incubator.httpclient. In actuality it was moved, modified and standardised into the module java.net.http. This process is special for incubating modules. They always start out separate from standardised modules. Depending on the progress they can later turn into preview features or immediately become stable like the http client. They might also be removed during incubating phase, if they do not provide enough value. This process is different for experimental hotspot JEPs. These do not start out as separate modules. The new scalable garbage collector „Z" is located directly in the hotspot code, even though it is not stable. The fact that hotspot is C++ code and cannot be controlled by the JPMS could have something to do with this.

JDK 12 to 17

Following the release of version 11 we see 5 more releases before September 2021 gives us the next „big" release in the form of JDK 17. Big here only means that 17 has LTS status. Which means enterprise developers can finally use long-standardised language features like switch expressions, records, sealed classes, sealed interfaces and pattern matching for instance of. The maintainers are clearly making continuous changes to the JDK. This becomes even more obvious when we map the hotspot and java.base areas of the JDK. Every release these areas increase (Δ+) or decrease (Δ-) by several thousand lines of code. In fact almost every file changes as can be seen in the following collage.

Hotspot&java.base Collage: JDK 12 to 17

That java.base can change so much, is already amazing. java.base is the module we all depend on and it is 586k lines big. That almost all of hotspot is changed with every release is even more impressive. These are the 806k lines that run the world. Well, a big part of the world at least. And no one has noticed all the internal changes.

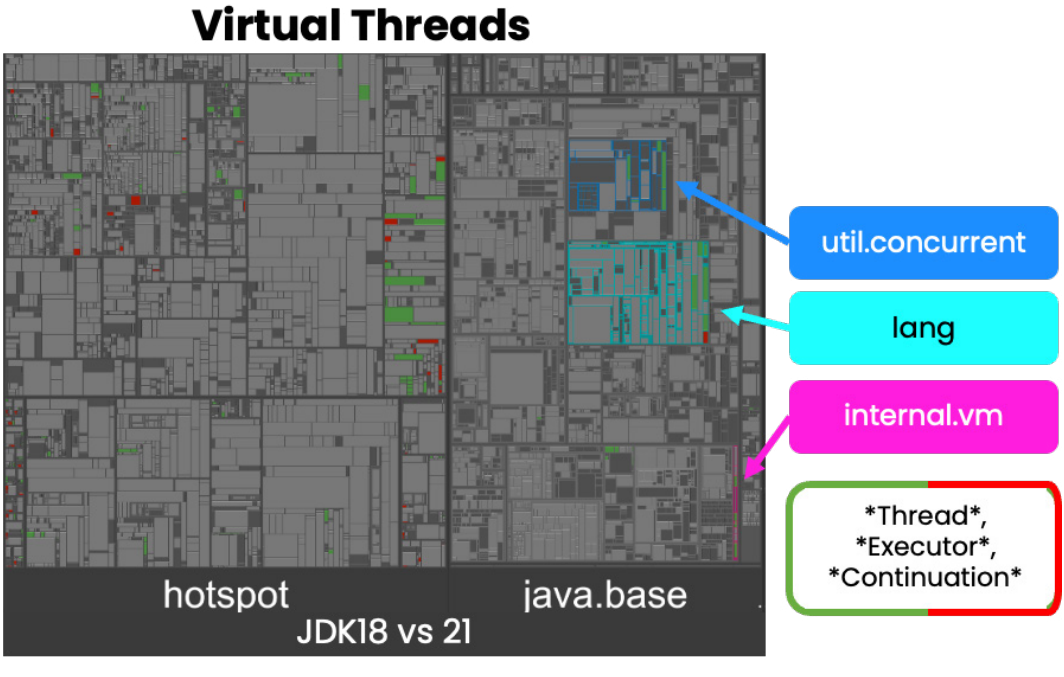
But why change the JVM at all? Two reasons immediately come to mind. First, there are significant performance improvements happening in hotspot all the time. Depending on your use case, version 17 can be up to 20% faster than version 11. Second, hotspot has to change to accommodate new enhancements. In particular new language features such as records.

At this level of detail it is however hard to locate these new language features, at least without prior knowledge. It is very likely they had an effect on the hotspot module. The 104k lines of the jdk.compiler were also changed with every release. How much they had to change just for records is hard to say though. No obvious class called CompileRecord.java exists in there. Records might also be used in java.base and lead to code reductions. In short language changes are harder to pinpoint and visualize, as they do not create obvious structure changes. So although I'd like to talk more about them, I currently do not have the knowledge to investigate further.

JDK 18 to 21

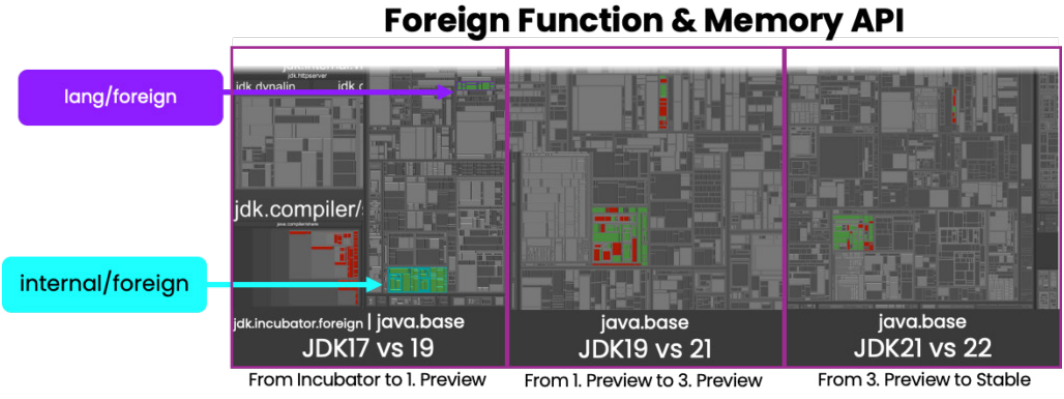
After version 17 the time until the next LTS was shortened to every 2 years, down from every 3 years. Which is why September 2023 marks the LTS release of JDK 21. By this point the maintainers have really adopted the idea to gather feedback in production before stabilising enhancements. Both version 19 and 20 had no user-visible JEP. Instead the enhancement were all experimental, incubating or preview. After multiple rounds of feedback however, many of these enhancements were stabilised with JDK 21. Language-wise we got stable record patterns and pattern matching for switch. Two features that paved the way for what the maintainers call data-oriented programming. It is an amazing new way to program in Java.

What we also got after two previews was virtual Threads. Virtual threads are an alternative to the classic platform threads. They are very useful to write high-throughput concurrent applications when the tasks to be done are I/O-bound (e.g. they interact a lot with the network). They are also a drop-in replacement of platform threads as they actually use the same API. A great thing for sure but that makes finding the necessary modifications harder again. We can imagine that supporting virtual threads required quite heavy modification to hotspot. But there is no package called virtual or virtualthread. We can search for *Thread*, *Executor*, *Continuation* and indeed we see a few modifications. But quite few compared to what was most likely changed. In lang the class Thread.java received 377 additional lines, VirtualThread.java with 778 lines was added as well as ThreadBuilders.java with 357 lines. In util.concurrent the class Executors.java received another 20 new lines. In internal.vm a new file called Continuation.java was added. And of course hotspot received a huge update, but at this level we cannot isolate what change was motivated by virtual threads.



Virtual Threads: JDK 18 vs 21

An enhancement where we can isolate the changes (or at least I think we can) is the new foreign function & memory API. It allows java programmers to safely invoke foreign functions (code outside the JVM) and access foreign memory (memory not managed by the JVM) and supersedes the JNI (Java Native Interface).



Foreign Function & Memory API: JDK 17 to 22

The FF&M API started out as an incubator and was made a preview feature in 19. We can see that the incubator code was moved, changed and made available as preview. Interestingly the bulk of its logic is internal to the JDK (11k lines in internal/foreign). Only a select few of the classes are visible to the outside world (1k lines in lang/foreign). It is quite hard to see this tiny stretch of exposed code even though we zoomed in on java.base. This structure stayed the same over the following previews. The code was optimised with each release until it was stabilised with JDK 22, a non-LTS release. It is obvious that the JDK maintainers take their time to get it right. They don't rush to hit a release train, even a long-term support train. In turn I don't split the FF&M analysis into this chapter and the one where it was actually stabilised but place it where it fits best.

JDK 22 to 25

JDK 25, with an expected release date of September 2025, is not just the latest LTS release. It is also the first time where release version and release year match. This has only whimsical importance but since we are celebrating 30 years of java why not add another fun fact to the list.

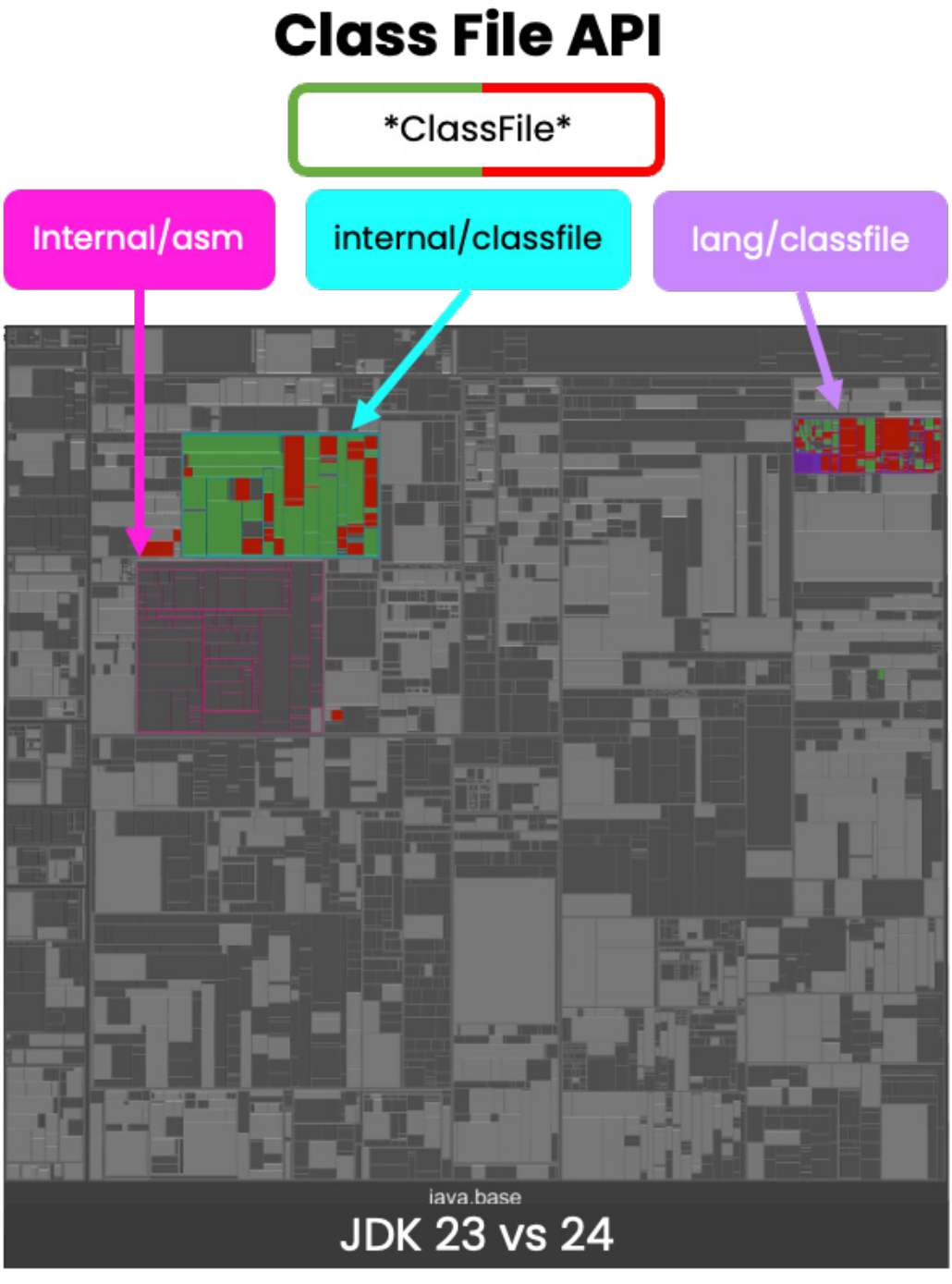
As of time of writing it is unclear what enhancements will be stable in JDK 25 but we can guess. Virtual threads will most likely be improved further. Version 24 already provided synchronize virtual threads without pinning. Perhaps we'll see scoped values in 25, the easier to reason about replacements for thread-local variables. We won't see structured concurrency being stabilised, as a fifth preview is planned for JDK 25.

We will probably see some new stable language features. Such as Primitive Types in Patterns, instanceof, and switch, Flexible Constructor Bodies, Module Import Declarations and Simple Source Files and Instance Main Methods. The latter JEP would however be great to have in 25, since it is „paving the on-ramp".

The idea of „paving the on-ramp" is to make it easier for people to learn java. Currently writing a simple HelloWorld requires learning oh so many keywords and concepts just to print two words to the screen. It would be great if we could reduce our class HelloWorld { public static void main ... to just the essentials: `void main() { println(„Hello World"); }`. This is exactly what the JEP does. The other concepts are not gone but they can be introduced gradually. Whether or not that is finalised with 25 is speculation though. The maintainers do not let arbitrary LTS deadlines tie them down.

What is not speculation is that JDK 24 delivered one enhancement that has been needed for at least 10 years, the Class File API. Up to its release, the JDK provided no official way to process the byte code it generated. It could of course always execute the byte code, but there was no official way to parse, generate and transform the class files that contain the byte code. This is something that frameworks often do to add functionality. It is also something that JDK does, for example to support lambda expressions at run time.

In the past the JDK has bundled it's own version of the popular open-source bytecode processor ASM to process their own bytecode. This creates multiple problems. As described by the JEP for the Class File API, one of them is a vicious circle: „The ASM version for JDK N cannot finalize until after JDK N finalizes, so tools in JDK N cannot handle class-file features that are new in JDK N, which means javac cannot safely emit class-file features which are new in JDK N until JDK N+1." When you pair this problem with a 6-month release cadence, it becomes very complicated for the maintainers. Arguably even more so for framework and the ASM developers to keep up. All the more reason to have not only an API for the JDK internally but also for users of the JDK.

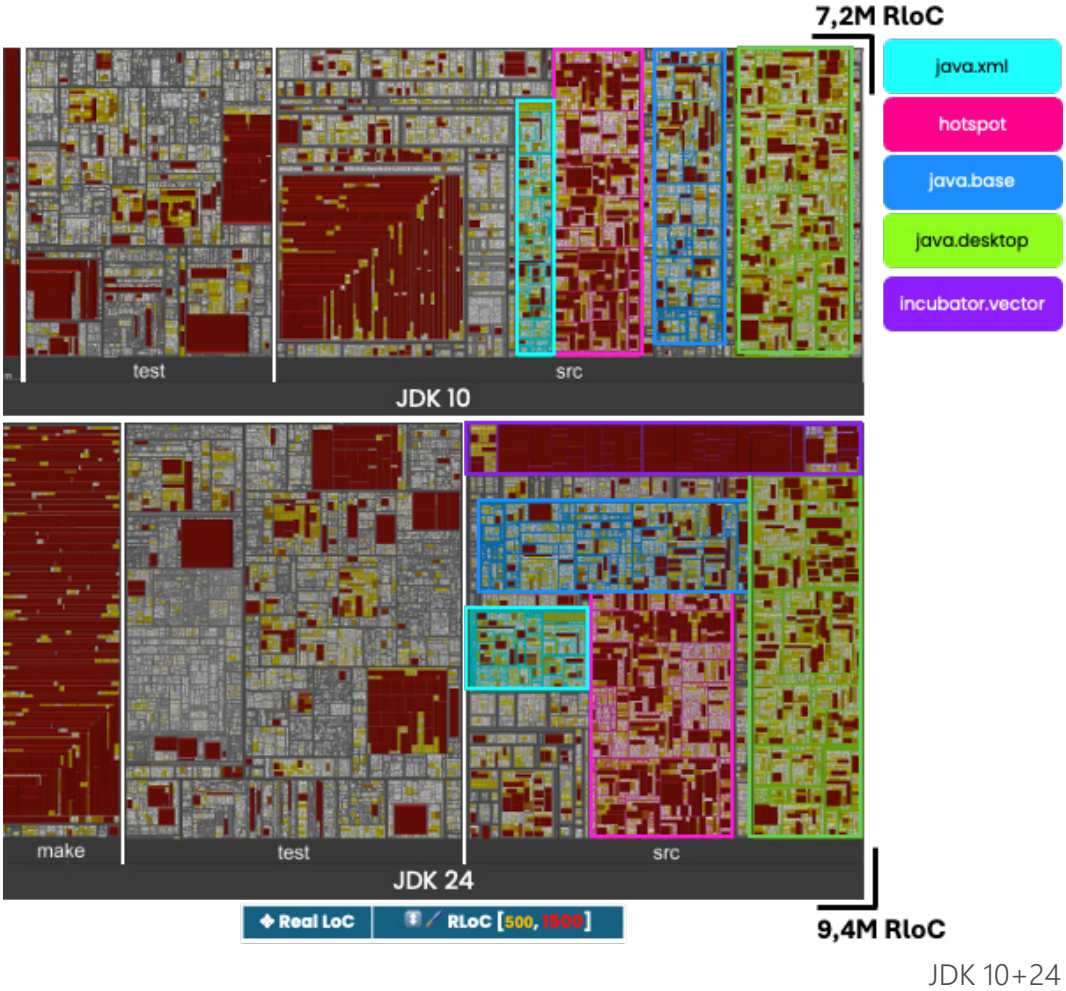


Class File API: JDK 23 vs 24

Like the FF&M API, the largest chunk of code is internal and thus hidden. The Class File API also wants to keep the exposed API surface as small as possible. At 6k lines the visible API is still quite large. At same time it is an achievement because without the module system it would have been 4 times as large. The other 21k lines are located in internal/classfile but they do not add to the API surface. The visible API and the internal logic together are roughly the size of ASM. Just based on size, we can surmise that the new API has roughly the same features as the bundled ASM. And indeed, the plan is to eventually remove this copy from the JDK.

The Journey So Far

The JDK is now almost 30 years old. We started this visual journey with JDK 7 in 2011 at the half-way point. Back then the JDK was 3,4 million lines heavy and only 12% of those were tests. With JDK 9 (September 2017) and 10 (March 2018) the JDK was changed to the new structure that has been kept until today. Version 10 contained 7,2 million lines and 29% of them were tests. We are now at JDK 24 with a release date of March 2025. The code today amounts to 9,4 million lines. 40% of them are tests. Without coverage data there is of course no way to say how that translates to path coverage. In my experience the src/test code split is healthy at roughly 50:50 though.



JDK 10+24

When we zoom in on the 4,3 million lines of src code we can see the familiar modules. java.desktop amounts to 25% of src. Today we mostly rely on that code to render IntelliJ, Eclipse and other IDEs. Another 22% go to hotspot, 15% is java.base and 6% are taken up by java.xml. All modules that were already around in JDK 10. The big newcomer is the incubating module for vector operations that takes up 12% of all src code.

In short, the JDK is massive. A fact that is also reflected in the size of files which we coloured in the map. Files above 500 lines are large and yellow, anything above 1500 lines is very large and red (note that we are counting the real lines of code, empty lines or JavaDoc comments are excluded). The maintainers do not share the same size sentiment. I think a big reason for that is that the JDK has to have a lot of convenience code to make the usage nicer. The List.java interface has some 50 methods for all the various interactions you might want to do. Arrays.java has above 100 methods which brings in 2k lines.

Even by these numbers the big blocks of red code in the center of incubator.vector are special. These are assembly files for linux and windows with 3k to 20k lines each. 402 thousand lines in total. I have no idea if they are hand-written or generated from some other source. It would be strange to have generated code committed to version control. I do hope they are all generated though because writing them by hand and keeping them in sync would be… challenging.

What the sum of these files do is clear. They support the new vector operations. Vector operations are in itself nothing special. Multiplying vectors and matrixes are common math operations after all. It is so common that CPU architectures often provide extensions that can make vector operations significantly faster. Hotspot even has a feature to auto-vectorize appropriate code but this is happening without programmer intent.

The Vector API now makes these operations accessible to performance critical code and it has been incubated nine times already. The first incubator was delivered with JDK 16 in March 2021. In the last couple of releases the assembly code has not changed but the java API receives additions to almost every file in almost every release. The maintainers are still working on the API to make it as clear and concise as possible. They will continue to do so until „necessary features of Project Valhalla become available as preview features“. But what is Valhalla and when will it be available?

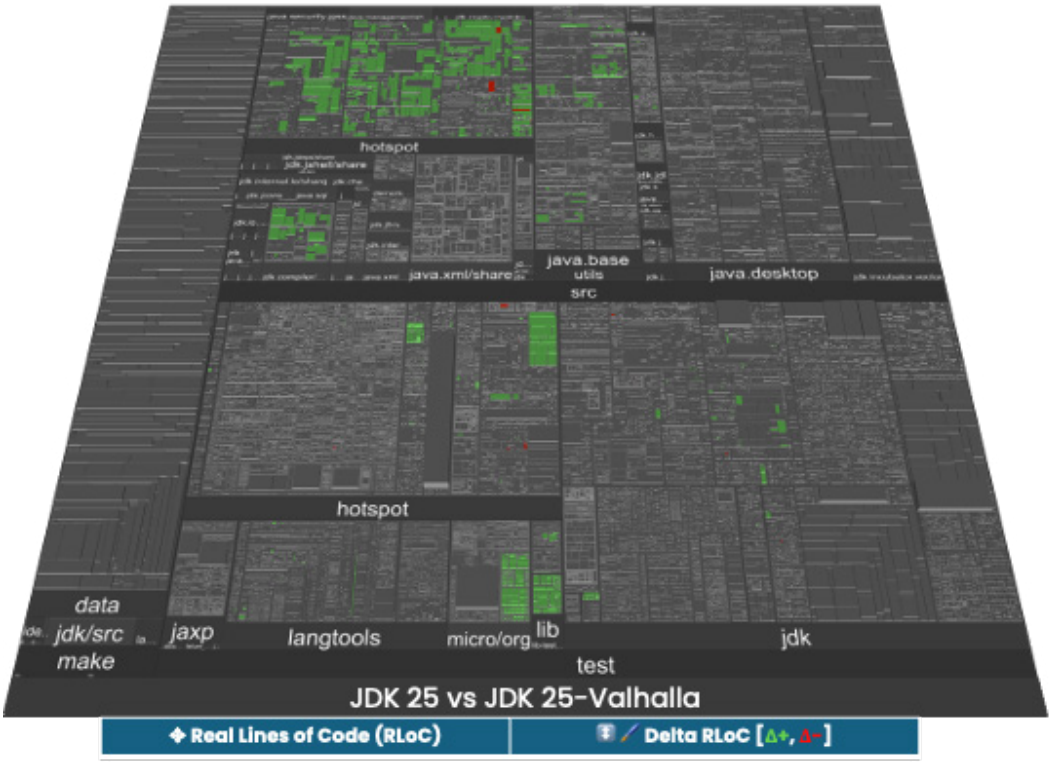
The Road Ahead

Project Valhalla's goal is to introduce value objects into Java that „code like a class, work like an int". This would allow objects with the performance characteristics of primitives (i.e. fast) but having the full modelling possibilities of classes. The performance is also the reason why the Vector API wants to build on the work of Valhalla.

If these value objects existed in the JDK, then there would be no difference between a primitive and a value object. We could unify the type system. For instance boxing/autoboxing from `int` to `Integer` and back would be a thing of the past. `List<Integer>` would be the same as `List<int>`. You would also be able to forbid `null` being put where only primitives should be allowed. Currently an `int[]` array forbids null but a `List<Integer>` allows it. The type system is not unified.

With Valhalla we would be able to chose to declare a `value class`, defined only by the values it contains, or the familiar `class`, defined by it's identity in the heap. Defining the latter is actually done by writing `identity class` but `identity` is the default keyword for classes, so we can omit it. Both class types can be written in essentially the same way but `value classes` have more constraints than `identity classes` have. The constraints are what allows hotspot to optimise the performance at runtime. One of these constraints, that us developers will be able to define, is whether or not the object is nullable („JEP draft: Null-Restricted Value Class Types")

Clearly having `value classes` would be great but getting them into the JDK has been a 11 year effort already that was started around the release of JDK 8. Some improvements have already been merged back to the JDK but most of the work is still being done in a separate openjdk/valhalla repository. The mainline jdk is being merged into the repository at certain points in time, allowing us to see what Valhalla will change.



JDK 25 vs Valhalla

Valhalla currently adds 124k lines of code. 17% in hotspot and 81% in tests. The test consist of 39% more hotspot tests and 31% more micro/org. The latter are probably the performance benchmarks. If that is the full extend of Valhalla remains to be seen.

At its release, Valhalla will certainly be the most significant transformation of the JDK yet. Which is quite an achievement when you consider what journey the JDK has already been on. It has kept the maintainers occupied for 10 years already and it is one of the puzzle pieces to get Java ready for the next 30 years. And Valhalla will not be the only enhancement to the JDK. A lot will happen in the following years.

Reproducing the Maps

If this article was interesting to you but you want even more details, then I encourage you to reproduce the results.

1. Clone the OpenJDK
2. Switch to the tag of the JDK you want to analyse. I picked the version of the JDK releases that was generally available (GA). The correct build number is listed on the page of the reference implementation. The jdk-xx-ga tags seem to have exactly the same purpose as the explicit ones below.

1. For JDK 21 it is jdk-21+35
2. For JDK 17 it is jdk-17+35
3. For JDK 11 it is jdk-11+28

3. Download CodeCharta

4. Generate a map of the OpenJDK with CodeCharta. Merge Tokei metrics with git metrics as shown in the docs. There is also a script for automated simple analysis.

5. Visualise the map in the Web Studio.

And the Final Step:

Ask your colleagues for help analysing the code. In my case I had enormous help from Stephan Schneider and Hans Spielvogel. I could not have written this article without their analysis. Thank you

> Back to Table of Content

#JAVAPRO #COREJAVA

Brewing Patterns in Java - An Informal Primer

Author:

Manoj Nalledathu Palat is an Open Source Lead/ Committer working at IBM leading the Java Compiler in Eclipse (https://projects.eclipse.org/projects/eclipse.jdt/PL/mpalat) . In parallel, Represented Eclipse Foundation in the Experts Group for Platform JSRs for multiple Java SE [eg: https://openjdk.java.net/projects/jdk/21/spec/] and IBM in Eclipse IDE WG Steering Committee. Passionate about training (Worked as Professor of Practice from January 2024-Dec 2024 at NIT, Calicut), Provide training internally at IBM regularly, blogs sometimes (https://medium.com/@manojnp,), mentors people. Holds a Masters degree in Computer Science from Indian Institute of Science(IISc), Bangalore and B-Tech in Computer Science from National Institute of Technology (Formerly REC) Calicut, India

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

Prologue

Not so long ago, Java lovers were engulfed by a mammoth wave of change — yes, we all know, right ? — the Lambda Expressions in Java 1.8! Eclipsed by lambda, minor in its avatar for the oblivious, was another message - ***" _ should not be used as an identifier, since it is a reserved keyword from source level 1.8 on"*** - An underscore was being removed as a legal identifier, quietly - And it was being promoted (sssshhh..underscore does not know yet) to a more complex role - It was just another sign of times to come - Time to Match Patterns in Java!

```
public class X {
   public int _; // warning:
}
```

ABCs of Pattern Matching

So what is pattern matching? All those unix fans out there would suddenly remember the good old awk; wikipedia says *"**AWK** .. used as a data extraction...tool;... A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed."* — A Given **Pattern is matched, Data is extracted** and **Some action** is taken - with or without the data. If this sounds too involved, we will be surprised to know that we have been using this all-along — The Find and Replace options all of us are very familiar with.

Let us take the example of a text "A journey of Thousand miles" where in we want to find and replace "miles" with "kilometres". So the pattern we are looking for is "miles", the data we are extracting is "miles" itself in this case — in these cases the pattern and the data are the same — and then we take an action on the data, ie replacing data- "miles" with "kilometres".

These first principles remain the same in Java pattern matching as well albeit with the addition of bells and whistles of supporting syntax and semantics.

The First Step

If we look closely to the above, we can clearly see a pattern emerging — a Match-Extract-Action sequence. Not surprisingly, pattern matching forayed into Java by a seemingly easy improvement — the Pattern Instanceof as shown below:

```
public void foo(Object obj) {
    if(obj instanceof Integer) { // Match
        Integer myInteger = (Integer) obj; // Extract
        System.out.println(myInteger); // Action
    }
}
```

All of us would have used the instanceof check and in most cases what we would do is to typecast this into the type for which we were checking; ie

```
if (obj instanceof Integer myInteger) { // Match and Extract.
    System.out.println(myInteger); // Action
}
```

The match and extract operations were combined into one by the pattern instanceof to provide a more compact code for the programmer. This has become a standard feature in Java 16.

Less Is More

Armed with the initial success of the pattern instanceof in the "if statement", the natural progression was to figure out how to get this naturalised in a switch statement. The term "naturalised" was used intentionally since the same construct used in switch would look ugly, to put it bluntly, and hence in Java 17, a preview feature is tested called the "Pattern Switch". And with Java 21, we have this as a standard feature and let's see how it looks:

```
switch(obj) {
 case Integer myInteger -> System.out.println(myInteger);
 case String myString -> System.out.println(myString);
 default -> System.out.println(„Object“);
}
```

Traditionally switch always had constant case labels — Now with Pattern Switches, the case labels contains types rather than constants. Can we bring the meaning of “constant-ness” in some way to the types? hmm.. good thought… Looks like we have something brewing already for this — **Enter Records** and **Sealed Types**.. These are quite involved topics themselves crying for their own blog-spaces, which will be for another article, but neverthless let us briefly glide over their surfaces for now.

Records — A Case of Constant Class

The concept of Records was introduced in Java 16; In fact, it was a preview feature for the previous two releases and became standard in 16. Suffice to say for now that Records are, practically, constant classes with a specific syntax — A syntax akin to a constructor — the definition as shown **below:record R(String name, Integer age) {}**

For now, just keep in mind that this compiles into a “constant” class, or just a class R with two private final fields “name” and “age”, the values of both of which need to be given at the time of instantiating the class. We would also have two accessors “built-in” by the compiler — namely “name()” and “age()” which return the values name and age respectively. Thus, with the advent of records, we are achieving the Constant-ness in “Data”.

Seal The Types

There is one more dimension to constant-ness in types. You can continue to derive subtypes from a type. From a compiler view point, especially from a pattern switch view point, this would mean that there should always be a “default” case arm to do the “catch-all” slippage of the pattern. If we really want to achieve “constant-ness” in that, all sub-types be known at the compile time, so that we can enumerate all the types in

the code, then we should be able to somehow “seal” the hierarchy and “permit” only those subtypes which we “permit”. And from 17 onwards, this can be achieved via the “sealed” and “permits” combination as shown below:

```
sealed interface I permits Y, Z, R {}
final class Y implements I {}
final class Z implements I {}
record R(String name, Integer age) implements I {}
```

“sealed” and “permits” are restricted identifiers where they have special meaning at class or interface definitions — we are sealing the interface I for the hierarchy at compile-time and permitting only classes Y and Z and the record R to implement the interface — thus giving an extra power for compile-time check for exhaustive analysis. Notice that the “final” is missing from R, since a record is final by definition, and hence the modifier final is optional.

Putting It All Together

Now, let us use this new-gained knowledge in our pattern switches to see how it looks, if we were switching on the interface I:

```
public void foo(I myInterface) {
 switch(myInterface) {
  case Y y -> System.out .println(y);
  case Z z -> System.out.println(z);
  case R r -> System.out.println(r.name());
    }
}
```

A careful reader would observe that the default arm is missing since we are enumerating all the cases — similar to the case where we cover all enum values — the essence is the same.

We Want More...Ooops.. Less!

Are we satisfied? Nice Try — we want more , Nay! We want less — less at extraction; So what can we do? Let us look at the code above — we

see that in the case of record, we are not using the record R as it is, but we are extracting the name — can we do something better? we can, if we are able to put the "name and age" in the case label itself as shown below:

```
case R(name, age) -> System.out.println(name);
```

Now, the extraction is simpler; its just using the "name" from the record type with all the „unrolling" being done behind the hood by the compiler.

Oh UnderScore! Wherefore art Thou?

We have come to the end of this article, but as mentioned earlier, each of these features deserves separate article(s) to justify their nuances. And more in-depth articles will be here soon... Before you go into deep slumber, let us check about our forgotten hero, the underscore (_). Where does this little fellow fit in the whole story?

Anyone who has tried Python will know that the underscores can be used to signal that we are not interested in that variable — now, if you don't know Python, its fine — this is just an unabashed attempt of Yours Truly to showcase that he knows more than what he does; given that he is still learning Java itself for the almost a decade making progress in such a pace giving the laziest snail a competition! So, cutting the blah, blah..what is the possible use of _ here? In the code in the previous section, we know that we use only "name"; "age" was added just to make the syntax complete.

```
case R(name, _) -> System.out.println(name);
```

What if, we could just put an underscore instead, for those variables which we don't use or care — like the code above? Well,with JEP 456: Unnamed Variables & Pattern this has become a reality -only point is we need to use Java 22 to use this feature. And to give _ it's rightful place, suffice to end saying that, when you see an underscore, please remember that there is this whole story behind it — Never underestimate the power of the "_".

Thanks for being with me so far!

> Back to Table of Content

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 #FRAMEWORKS #API

Transforming POJOs and Java Records with Froporec: Deep Immutability and Beyond

Author:

Mohamed Ashraf Bayor is a Software Engineer with over 15 years of experience in Java development. Throughout his career, he has navigated diverse high-technology industries, including FinTech, Aerospace Manufacturing, Telecommunications and Healthcare. Passionate about open source contributions, Mohamed launched two impactful projects in November 2021: Froporec (https://froporec.org/) and Jisel (https://jisel.org/). In October 2023, Mohamed also introduced JAPO (Java Annotations Repo - https://japo.dev/), which marks the world's first search engine dedicated to searching and exploring Java annotations.

https://www.linkedin.com/in/mohamed-ashraf-bayor

Java has been evolving, with major improvements over the years, especially in the area of data handling. One of the latest advancements, introduced in Java 14, is the Record class, a special type of class aimed at reducing boilerplate code for immutable data carriers. While records

offer many advantages, they come with certain limitations, especially regarding immutability and extensibility. **Froporec**, a Java annotation processor, offers a robust solution to these issues by simplifying the migration from POJOs (Plain Old Java Objects) to records while also providing deep immutability and the ability to extend records in ways Java doesn't natively support.

What Is Froporec?

Froporec (https://froporec.org) is an open-source Java annotation processor that **simplifies the migration from POJOs to records** and **enhances the functionality of records**. Designed to work with Java 17 or higher, Froporec provides several key annotations to facilitate various use cases:

- **Migration from POJOs to Records:** Convert existing POJOs into records while retaining their data structure.
- **Deep Immutability for Records:** Address the limitation of Java records, where mutable objects like collections or POJOs can break the immutability of records. Froporec ensures deep immutability, making Records fully immutable and secure.
- **Extending Records:** Java records cannot be extended due to their final nature. Froporec introduces a way to extend records by merging them with fields from other POJOs or records.

Additionally, Froporec provides features for **factory methods** and **constants for field names**, but we will not cover them in this article.

For installation instructions and setup details, visit https://froporec.org/#installation.

Migration From POJOs to Records

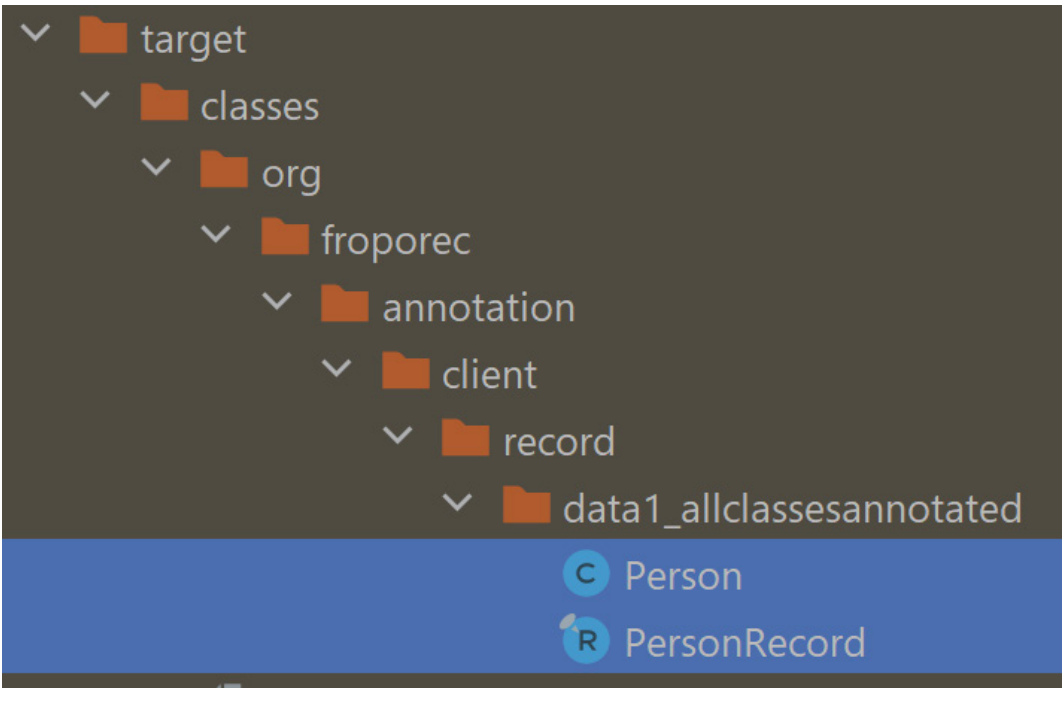
POJO (Plain Old Java Object) classes, the traditional Java classes for data transfer, are mutable by default, making them less secure in some cases. Froporec makes it easy to migrate from these mutable POJOs to immutable records using **simple annotations**. By annotating a POJO with `@Record`, Froporec generates a corresponding record class, ensuring that your codebase remains consistent and modern.

```
package org.froporec.annotation.client.record.data1_
allclassesannotated;

import org.froporec.annotations.Record;

@Record
public class Person {

    private String lastname;
    private int age;

    public Person(String lastname, int age) {
        this.lastname = lastname;
        this.age = age;
    }

    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Note: We include the all-args constructor only to simplify instance creation, but Froporec does not require it to function.

Upon compilation, the code above will generate a PersonRecord class with the same fields, but all properties will be immutable. If you're

building a Maven project, the generated `PersonRecord` class will appear in the `target` folder, within the same package as the original `Person` POJO class. The `PersonRecord` class will be located alongside the `Person` POJO class and will be accessible from anywhere in your code in the same way as the `Person` POJO class.



Here is the content of the generated PersonRecord class (the `@Generated` annotation section, constant fields, and factory methods are omitted for better readability):

```
package org.froporec.annotation.client.record.data1_
allclassesannotated;

/* REMOVED @Generated annotation section */
public record PersonRecord(java.lang.String lastname, int age)  {

/* REMOVED Constant fields section */

public PersonRecord(org.froporec.annotation.client.record.data1_
allclassesannotated.Person person) {
          this(person.getLastname(), person.getAge());
    }

/* REMOVED Factory methods section */
}
```

As seen in the generated `PersonRecord` class above, two major changes stand out:

- **PersonRecord is a record class, not a POJO** – The declaration is now record `PersonRecord(java.lang.String lastname, int age)`
- **A special constructor is generated** – This constructor accepts a Person POJO instance as a single parameter and passes its data to the record's canonical constructor. This ensures that the generated record maintains the same data as the original POJO class.

Below is an example of how to access and use the generated `PersonRecord` class by passing an instance of the original POJO class:

```
package org.froporec.annotation.client.sampleuse;

import org.froporec.annotation.client.record.data1_
allclassesannotated.Person;
import org.froporec.annotation.client.record.data1_
allclassesannotated.PersonRecord;

public class PersonRecordUse {

    public static void main(String[] args) {

        Person johnPojo = new Person(“Doe“, 50);

        System.out.printf(“““
                          %n
                          --- POJO DATA ---
                          LastName: %s, Age: %d
                         ““““,
                johnPojo.getLastname(),
                johnPojo.getAge());

        PersonRecord johnRecord = new PersonRecord(johnPojo);

        System.out.printf(“““
                          %n
```

```
                        --- RECORD DATA ---
                        LastName: %s, Age: %d
                       """,
                johnRecord.lastname(),
                johnRecord.age());
    }
}
```

Here is the execution result:

```
 --- POJO DATA ---
 LastName: Doe, Age: 50

 --- RECORD DATA ---
 LastName: Doe, Age: 50
```

Guaranteed Deep Immutability

Let's now see what happens if the POJO class has a mutable collection member.

```
package org.froporec.annotation.client.record.data1_
allclassesannotated;

import org.froporec.annotations.Record;
import java.util.List;

@Record
public class Person {

    private String lastname;
    private int age;
    private List<String> addresses; /* Mutable collection */

    // Getters, setters,...
}
```

Here is the content of the generated `PersonRecord` class:

```
package org.froporec.annotation.client.record.data1_
allclassesannotated;

/* REMOVED @Generated annotation section */
public record PersonRecord(java.lang.String lastname,
                           int age,
                           java.util.List<java.lang.String>
                           addresses)  {

    /* REMOVED Constant fields section */

public PersonRecord(org.froporec.annotation.client.record.data1_
    allclassesannotated.Person person) {
        this(
                person.getLastname(),
                person.getAge(),
                java.util.Optional.ofNullable(person.getAddresses()).
                 isEmpty()
                        ? java.util.List.of()
                        : person.getAddresses().stream().map(object
                        ->
(object)).collect(java.util.stream.Collectors.toUnmodifiableList())
        );
    }
    /* REMOVED Factory methods section */
}
```

In the generated `PersonRecord` constructor's body, the third argument ensures that the addresses field is always an unmodifiable list. It first checks if `person.getAddresses()` is null or empty, returning an empty immutable list `(List.of())` if true. Otherwise, it processes the list with a stream, mapping each element unchanged and collecting the result as an unmodifiable list using `Collectors.toUnmodifiableList()`, **ensuring immutability and preventing accidental modifications**.

Alternative Uses of `@Record`

The `@Record` annotation offers flexibility beyond standard class-level declarations. Below are different ways to use it effectively.

Annotating a Class Field Type

You can place `@Record` next to a field declaration to ensure that a record class is generated for the enclosed object type.

```
@Record
class Person {
    private @Record Address address;
}
```

In this example:

- A corresponding `AddressRecord` class is generated for the `Address` type.
- The generated `PersonRecord` class will contain an `AddressRecord` instance instead of an `Address` instance, ensuring consistency in immutability.

Annotating a Method Parameter

You can also annotate a method parameter with `@Record`, triggering the generation of a record class during compilation.

```
void doSomething(@Record Person person) {
    PersonRecord personRecord = new PersonRecord(person);
}
```

This ensures that the `PersonRecord` class is available within the method. Note that the record class is generated and accessible only after at least one successful compilation.

Using the `alsoConvert` Attribute

Instead of annotating multiple classes separately, you can use the `alsoConvert` attribute to generate multiple record classes in one go

```
@Record(alsoConvert = { Address.class, Email.class, Job.class })
class Person {

    private String lastname;
    private int age;
    private Address address;
    private Email email;
    private Job job;

    // Getters, setters,...
}
```

This approach generates record classes for `Person, Address, Email,` and `Job,` streamlining the conversion of multiple POJO classes into their record equivalents.

Note: The `alsoConvert` attribute may contain a mix of both existing POJO and record `.class` values, allowing seamless conversion regardless of the original class type.

Deep Immutability for Record Classes

In Java, record classes offer a convenient way to define immutable data carriers, with **all their fields implicitly marked as final**. However, this immutability is ***shallow*** by default, meaning that while the reference to the field itself is immutable, the contents of mutable objects within those fields (e.g. collections) are still mutable. This can lead to unintended side effects, especially when working with mutable collections.

Consider the following example:

```
package org.froporec.annotation.client.immutable.data4_
collectionsannotated;
```

```
import java.util.List;

public record Person(
        String lastname,
        int age,
        List<String> addresses /* Mutable collection */) {
}
```

In this case, the `addresses` field is a mutable `List<String>`. Even though the record itself ensures that the reference to `addresses` is final, **the content of the list remains mutable**. Here's how you can still modify the contents of the collection:

```
void addAddress() {
    Person person = new Person(„Doe“, 50, new ArrayList<>()); // an
    empty List is passed as third argument
    person.addresses().add(„office address“); // we can add a new item
    to the addresses List
}
```

To resolve this issue, Froporec provides the `@Immutable` annotation, which guarantees that mutable collections within a record are wrapped in unmodifiable views, ensuring that the content is immutable.

```
package org.froporec.annotation.client.immutable.data4_
collectionsannotated;

import org.froporec.annotations.Immutable;
import java.util.List;

@Immutable
public record Person(
        String lastname,
        int age,
        List<String> addresses) {
}
```

By annotating the `Person` record with `@Immutable`, the `addresses` field is transformed into an unmodifiable list. This is achieved by processing the collection as a stream, mapping each element unchanged, and then collecting the result as an unmodifiable list using `Collectors.toUnmodifiableList()`. Here is the content of the generated `ImmutablePerson` class:

```
package org.froporec.annotation.client.immutable.data4_
collectionsannotated;

/* REMOVED @Generated annotation section */
public record ImmutablePerson(java.lang.String lastname,
                              int age,
                              java.util.List<java.lang.String>
                              addresses) {

    /* REMOVED Constant fields section */

public ImmutablePerson(org.froporec.annotation.client.immutable.
    data4_collectionsannotated.Person person) {
        this(
                person.lastname(),
                person.age(),
                java.util.Optional.ofNullable(person.addresses()).
                 isEmpty()
                        ? java.util.List.of()
                        : person.addresses().stream().map(object ->
(object)).collect(java.util.stream.Collectors.toUnmodifiableList())
        );
    }

    /* REMOVED Factory methods section */
}
```

Notes:

- As you may have noticed in the code above, the generated record class is named **`ImmutablePerson`** instead of `PersonRecord`, which is the default naming convention when using `@Record`.

- The alternative uses of the `@Record` annotation (annotating a class field type, a method parameter, and using `the alsoConvert` attribute), as discussed in the previous section, also apply to the `@Immutable` annotation. This means you can ensure deep immutability of objects while benefiting from the same conversion and record class generation mechanisms.

Extending Records

Java records are **implicitly final**, meaning we cannot extend them. This design choice is intentional and aligns with the primary purpose of records: to be **transparent, immutable data carriers** (JEP 395).

To bypass this limitation, Froporec offers the **`@SuperRecord`** annotation, which allows you to combine fields from multiple POJOs and records into a new record class, without using traditional Java inheritance. The `@SuperRecord` annotation can be used on top of either a POJO or a record class.

```
@SuperRecord(mergeWith = {School.class, Student.class})
public class Person {

    private String lastname;
    private int age;

    // Getters, setters,...
}

record School(String name) {
}

class Student {

    private int mark;
    private String grade;

    // Getters, setters,...
}
```

Upon compilation, the code above generates a new record class named `PersonSuperRecord`, which combines all fields from the `Person`, `School`, and `Student` classes. The fields from the classes specified in the `mergeWith` attribute are automatically added to the Person class.

```
public record PersonSuperRecord(java.lang.String lastnamePerson,
                                 int agePerson,
                                 java.lang.String nameSchool,
                                 int markStudent,
                                 java.lang.String gradeStudent) {
    public PersonSuperRecord(org. froporec.annotation.client.
    superrecord.Person person,
    org.froporec.annotation.client.superrecord.School school,
    org.froporec.annotation.client.superrecord.Student student) {
        this(
                person.getLastname(),
                person.getAge(),
                school.name(),
                student.getMark(),
                student.getGrade()
        );
    }
}
```

The generated `PersonSuperRecord` record provides a constructor that allows you to create a new instance by passing existing instances of the POJOs or records listed in the `mergeWith` attribute.

Integration With Lombok

In previous examples, our POJO classes explicitly defined **getters and setters**, a common characteristic of traditional Java POJOs. As of version 1.4, Froporec relies solely on getter methods to access fields in a POJO when generating the corresponding record class. This means that when using the `@Record` annotation, **getters must be present in the POJO** for Froporec to function correctly.

Manually writing getters for every field can be repetitive and time-consuming. This is where **Lombok**, a widely used open-source Java annotation processor, comes into play. Lombok simplifies Java development by reducing boilerplate code, particularly for common methods like getters, setters, constructors, and more. With Lombok's `@Getter` annotation, developers can automatically generate getters at compile time without explicitly writing them in the class.

The following configurations are for projects built with Maven.

Maven Dependency Configuration

To integrate Froporec and Lombok in a Maven project, add the following dependencies to your pom.xml:

```
<dependencies>
    <dependency>
        <groupId>org.froporec</groupId>
        <artifactId>froporec</artifactId>
        <version>${froporec.version}</version> <!-- Use latest -->
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version> <!-- Use latest -->
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Ensure Correct Annotation Processing Order

Since both Lombok and Froporec are annotation processors, they must be explicitly added to the Maven compiler plugin configuration to ensure they run during compilation. However, **the order in which they are declared in the `annotationProcessorPaths` section is critical**:

- **Lombok must execute first** to generate the necessary getter methods.
- **Froporec must execute after Lombok** to process the POJO with the generated getters.

If the annotation processing order is incorrect, Froporec may fail to detect the required getters.

Maven Compiler Plugin Configuration

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven-compiler-plugin.version}</version>
            <configuration>
                <release>21</release>
                <compilerArgs>-Xlint:unchecked</compilerArgs>
                <annotationProcessorPaths>
                        <!--
                            Annotation processors execute in the
                           order listed below.
                            Lombok runs first to generate boilerplate
                            code (e.g., getters/setters),
                            ensuring Froporec processes the updated
                            classes correctly.
                        -->
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${lombok.version}</version>
                    </path>
                    <path>
                        <groupId>org.froporec</groupId>
                        <artifactId>froporec</artifactId>
                        <version>${froporec.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
</build>
```

Using Lombok and Froporec Annotations Together

With this setup, you can now use both Lombok and Froporec in your POJO class:

```
@Record
@Getter
public class Person {

    private String lastname;
    private int age;
    private List<String> addresses;
}
```

Generated PersonRecord Class

Upon compilation, `Froporec` generates the following record class:

```
/* REMOVED @Generated annotation section */
public record PersonRecord(java.lang.String lastname,
                           int age,
                           java.util.List<java.lang.String>
                           addresses)  {

    /* REMOVED Constant fields section */

    public PersonRecord(org.froporec.annotation.client.record.data1_
    allclassesannotated.Person person) {
        this(
                person.getLastname(),
                person.getAge(),
                java.util.Optional.ofNullable(person.getAddresses()).
                isEmpty()
                ? java.util.List.of()
                : person.getAddresses().stream().map(object ->
                 (object)).collect(java.util.stream.Collectors.
                 toUnmodifiableList())
        );
```

```
    }
    /* REMOVED Factory methods section */
}
```

Summary

Java records have introduced a more efficient way to handle immutable data structures, but they come with limitations, particularly around deep immutability and extensibility. Froporec addresses these issues by offering a streamlined way to migrate POJOs to records while ensuring true immutability

Library Name	FROPORECC
Library Type	Annotation Processor (Supports `@Record, @Immutable`, and `@SuperRecord` annotations as of v1.4)
Minimum Java Version	Java 17
Specific Requirements	The annotated class must have Getter methods (unless it is a `Record` class)
Execution	- Runs at compile-time - Generates a record class alongside the annotated class within the same package
Compatibility	Works well with libraries like Lombok and similar tools (A correct order of annotation processors in the build configuration is required for seamless integration)

Froporec Java Library

Key takeaways from this article:

- **Seamless POJO-to-Record conversion** with minimal effort using the `@Record` annotation.
- **Guaranteed deep immutability,** ensuring that even mutable objects like collections remain immutable within records. Additionally, the `@Immutable` annotation can be applied to existing record classes to enforce immutability on fields that might otherwise be mutable.
- **Flexible annotation usage**, including class-level, field-level, and method parameter-level conversions.

- **Batch conversion** via `alsoConvert`, enabling multiple POJOs and/or Record classes to be transformed in a single annotation.
- **Record Extensibility** with `@SuperRecord`, combining fields from multiple POJOs and records into a new record class without relying on traditional Java inheritance, enhancing flexibility and reusability.

To explore Froporec further, including additional features, installation instructions, and code samples, visit the official webpage or GitHub repository. There, you can also contribute to the project's growth by reporting issues or suggesting improvements.

Whether you're modernizing legacy Java applications or starting a new project, Froporec provides a clean and efficient approach to handle immutable data. It's a must-have tool for teams transitioning to modern Java practices. By simplifying immutable data management, Froporec helps build safer, more maintainable applications and ensures a smoother development experience.

> Back to Table of Content

#JAVAPRO #FRAMEWORKS #API

Crafting Your Own Railway Display with Java!

Author:

Rijo Sam works as a Java Chapter Lead based in the Netherlands. He possesses extensive experience in developing applications within the payments and credits sector of the banking domain. Rijo is originally from India and is now settled in the Netherlands.

https://www.linkedin.com/in/rijosam19/

Have you fancied to have your own railway display at home? If you love traveling by public transport and always jump on the train just before the door closes like me, it's really cool and highly efficient to have your own personalized display.

Background

Without live data, this project could not work. Luckily, all the data that was needed, was publicly and freely available with Nederlandse Spoorwegen (NS) APIs. To access them, you only need to create an account in the NS API Portal and subscribe to the APIs you want to use.

For the application's backend, Java, Jakarta EE, and SpringBoot were chosen. For the front end, Vaadin was selected because it is closely related to Java and is perfect for a simple screen. Raspberry Pi 4 and an existing monitor were used as the hardware for the project.

Implementation

1. Setting Up the Backend

Jakarta WebTarget was used to connect to the Departures API from NS, which requires the query parameter 'uicCode', an international unique identifier for railway stations. Stations in Netherlands have a UIC code that starts with 84 (e.g., 8400058 for Amsterdam Centraal).

Sample code for connecting to the NS API using WebTarget

```
private Response getTrainsInfo(String stationUicCode) {
  return webTargetProvider.getWebTarget(getUri(stationUicCode))
    .request().accept(MediaType.APPLICATION_JSON)
    .header(HttpHeaders.CACHE_CONTROL, „no-cache“)
    .header(„Ocp-Apim-Subscription-Key“, subscriptionKey)
    .buildGet().invoke();
}

private URI getUri(String stationUicCode) {
  return UriBuilder
    .fromUri(baseUrl).path(uriPath)
    .queryParam(„uicCode“, stationUicCode)
    .build();
}
```

The required data from the API response was captured as TrainInfo with the following fields.

```
public record TrainInfo(String direction,
  String plannedDepartureTime,
  String actualDepartureTime,
  String actualTrack,
  String trainCategory,
  String routeStations,
  String departureStatus,
  boolean isCancelled)
}
```

2. Building the View

The Grid component was used from Vaadin flow. It is a simple component for displaying tabular data with different rendering options. Renderers like Component Renderer or Lit Renderer can then customize the content displayed in specific columns

```
public MainView(TrainDepartureService trainDepartureService) {
  grid = new Grid<>();
  var trainDepartures = trainDepartureService.getDepartureInfo();
  grid.setItems(trainDepartures);
  grid.addThemeVariants(GridVariant.LUMO_ROW_STRIPES);
  grid.addColumn(createPlatformRenderer()).setHeader(„PLATFORM“);
  grid.addColumn(createStatusRenderer()).setHeader(„STATUS“);
  add(grid);
}
```

```
private static Renderer<TrainDeparture> createPlatformRenderer() {
  return LitRenderer.<TrainDeparture>of(
    „<vaadin-horizontal-layout
    style=\“align-items: center;\“theme=\“spacing\“>“ +
    „<span part=\“platformStyle\“> ${item.actualTrack} </span>“ +
    „</vaadin-horizontal-layout>“)
    .withProperty(„actualTrack“, TrainDeparture::actualTrack);
}
```

Sample code for Component Renderer

```
private static ComponentRenderer<Span, TrainDeparture>
createStatusRenderer() {
  return new ComponentRenderer<>(Span::new,(span, trainDeparture) ->
  {
    span.setText(trainDeparture.status().name()); });
}
```

3. Styling the View

The Grid component was used from Vaadin flow. It is a simple component Finding the right color and font was the trickiest part. Luckily, there are websites that can help you find the exact RGB code by uploading your image and zooming over it.
There are also a few sites that can extract the different fonts used in your image, but be aware that the fonts suggested might not always be readily available or compatible.

4. Scheduled Refresh

To show the updated information on screen with an interval of one minute, the combination of scheduler from Spring Boot and Push function from Vaadin were used.

```
// executed at the start of every minute.
@Scheduled(cron = „0 * * * * ?“)
public void updateGrid() {
  var trainDepartures = trainDepartureService.getDepartureInfo();
  getUI().ifPresent(ui -> {
  if (ui.isAttached()) ui.access(() -> grid.
  setItems(trainDepartures));
  });
}
```

Vaadin Server push is based on a client-server connection established by the client. The server can then use the connection to send updates to the client. The @Push annotation was used on the application class to enable server push.

```
@Push
@EnableScheduling
@SpringBootApplication
public class Application implements AppShellConfigurator {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);}
}
```

5. Setting up the Raspberry Pi 4

Raspberry Pi Imager was used to install the Raspberry Pi OS and SDKMAN to install Java 17. To run the NS application on the Raspberry Pi, a fat jar file was created and the application was started by executing the JAR file. For additional debugging within the Raspberry Pi, the already shipped VSCode was used.

Conclusion

In this project, Java, SpringBoot and Vaadin were explored to create a UI application seamlessly. Vaadin wrapped many complexities in annotations like @Push or the Component Grid frameworks. It was truly amazing to develop and run this application on Raspberry Pi 4. Finally, it's not so much about technology as about the basic human need to solve

problems and the curious mind that gets creative with every challenge. If you would like to know more about the project, please check out on GitHub.

References

- NS API Portal: https://apiportal.ns.nl/
- SDKMAN: https://foojay.io/today/installing-java-with-sdkman-on-raspberry-pi/
- Vaadin Docs: https://vaadin.com/docs/latest/overview
- RBG Color Picker: https://imagecolorpicker.com/
- Font Finder: https://www.myfonts.com/pages/whatthefont

> Back to Table of Content

#JAVAPRO #FRAMEWORKS #API

Untapped Potential in the Java Build Tool Experience

Author:

Li Haoyi graduated from MIT with a degree in Computer Science and Engineering, since then has built core infrastructure for high-growth companies like Dropbox and Databricks, and has been a major contributor to the open source community. His projects have over 10,000 stars on Github, and are downloaded over 20,000,000 times a month. Haoyi has deep experience in the JVM and has professionally built distributed backend systems, programming languages, high-performance web applications, and much more.

The Java language is known to be fast, safe, and easy, but Java build tools like Maven or Gradle don't always live up to that reputation. This article will explore what „could be": where current Java build tools fall short in performance, extensibility, and IDE experience, and the reasons to believe that we can do better. We will end with a demonstration of an experimental build tool „Mill" that makes use of these ideas, proving out the idea that Java build tooling has the potential to be much faster and easier to use than it is today.

Can Java Build Tools Be Better?

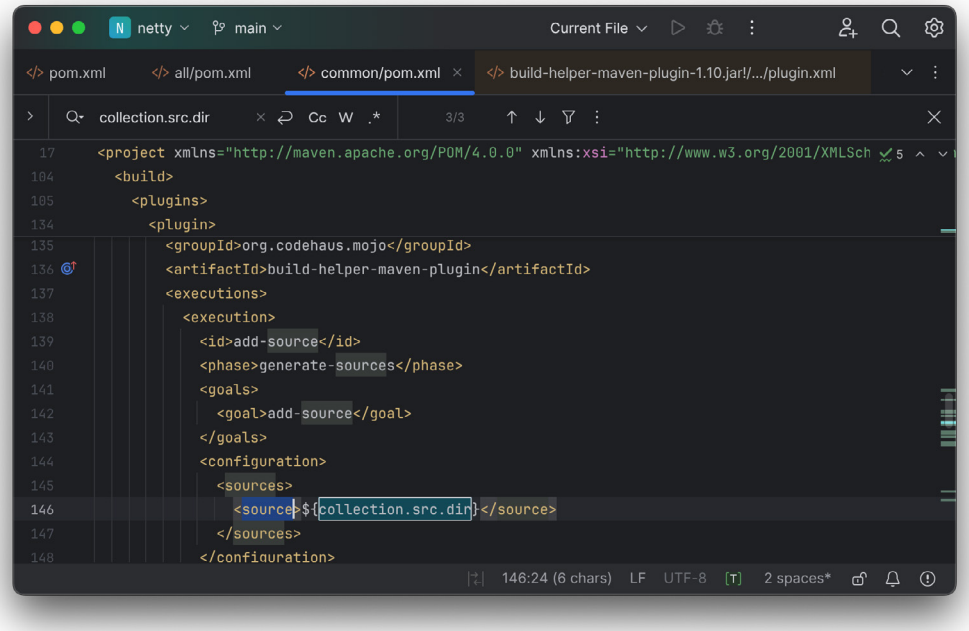
Build tools like Maven and Gradle have been staples of the JVM ecosystem for decades now. Countless developers have used these tools to build projects large and small, so they clearly work for the tasks they are used for. However, just because something works doesn't mean that there isn't room to improve! There are three main areas that these Java build tools typically fall short:

1. IDE Experience
2. Extensibility
3. Performance

We will discuss each one in turn.

IDE Experience

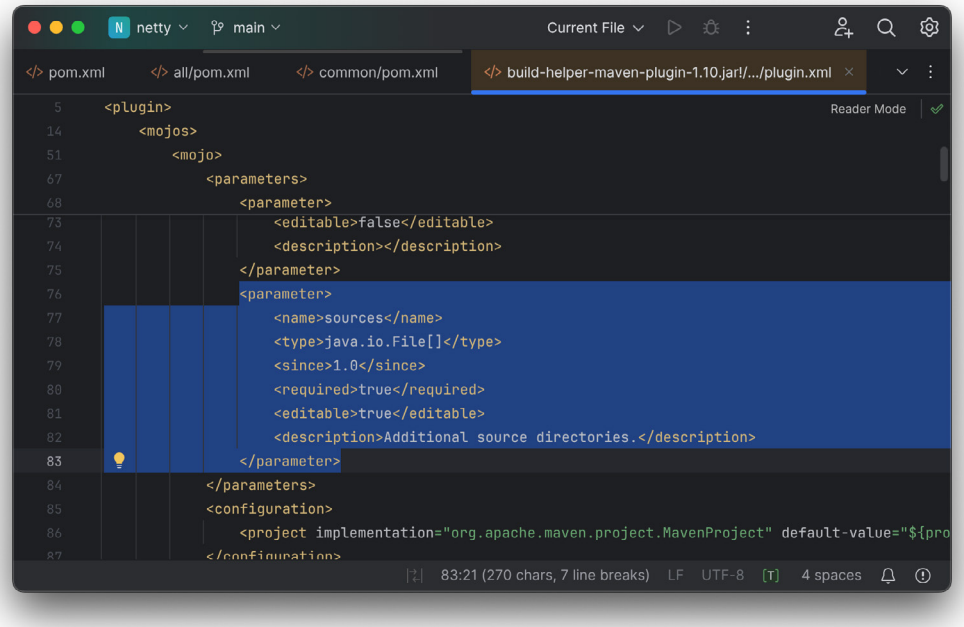
Although build tools like Maven or Gradle have support in all modern IDEs, that support can be surprisingly thin and unhelpful. For example, consider the following snippet in a Maven `pom.xml` file:



If you already familiar with the various plugins involved and have already memorized how they are configured, this snippet makes perfect sense. However, in the real world things are not always quite so neat, and a

developer may not be familiar enough with every single tool or plugin to be able to instantly recall how they are configured and used. Sometimes mistakes are made and the developer has to dig deep to investigate a bug or misbehavior. That is where IDEs come in: they assist the developer by instantly pulling up the documentation or implementation of APIs they may be unfamiliar with, so they can quickly learn what something does and how it can be used to accomplish their goals.

However, if you try to use the Intellij IDE's jump to definition on this Maven `pom.xml` snippet, it brings you to this:

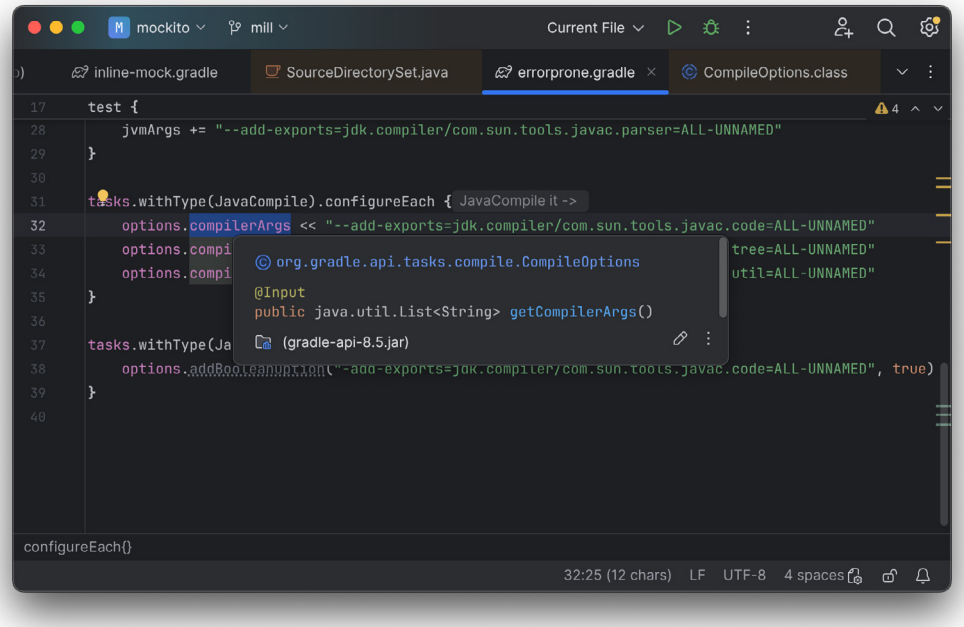


This is the signature of the sources configuration value: we now know that sources has the `typejava.io.File[]`, it is required, and it is documented as `Additional source directories`. If that is all you need then great, but sometimes it isn't. If I as a developer need to debug an issue in the local build or the upstream plugin, I may need to ask follow ups: how is sources used by the plugin? How do changes in `sources` end up influencing the behavior of the system, in this case the build tool?

In application code, you can almost always use your IDE's jump to the definition, find usages, or other code navigation tools to explore and understand the underlying logic, but these tools are conspicuously absent when working with build tool config files. As a result you often end up re-reading the documentation for the Nth time, digging through Stackoverflow, or copy-pasting from other local examples. While this can help, it is much more time-consuming and error-prone than being able

to explore the system in your IDE.

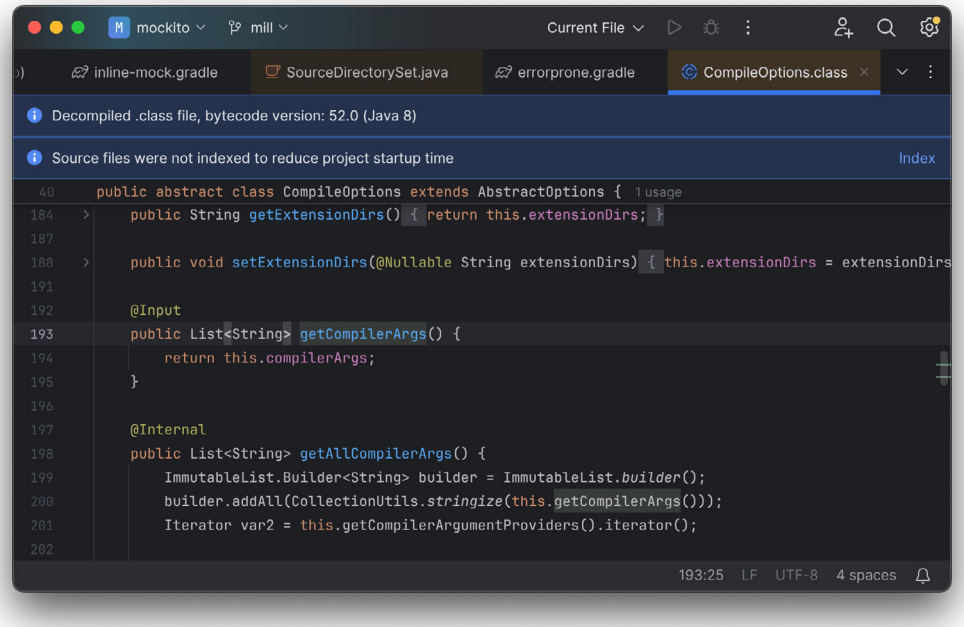
This problem is not unique to Maven and it's `pom.xml`; Gradle suffers the same problem. For example, let's say you looked at the following code and weren't 100% sure what `compilerArgs` does:



Again, you could probably guess what this does: it configures the compiler. But it is still reasonable to want to learn more:

- *Which* compiler? Gradle supports Java, Kotlin, Scala, Swift, and other languages!
- What is the default value of this field?
- What was the value of this field before we appended the `--add-exports` flag to it?
- Does it apply to test code compilation? Or `javadoc` compilation?

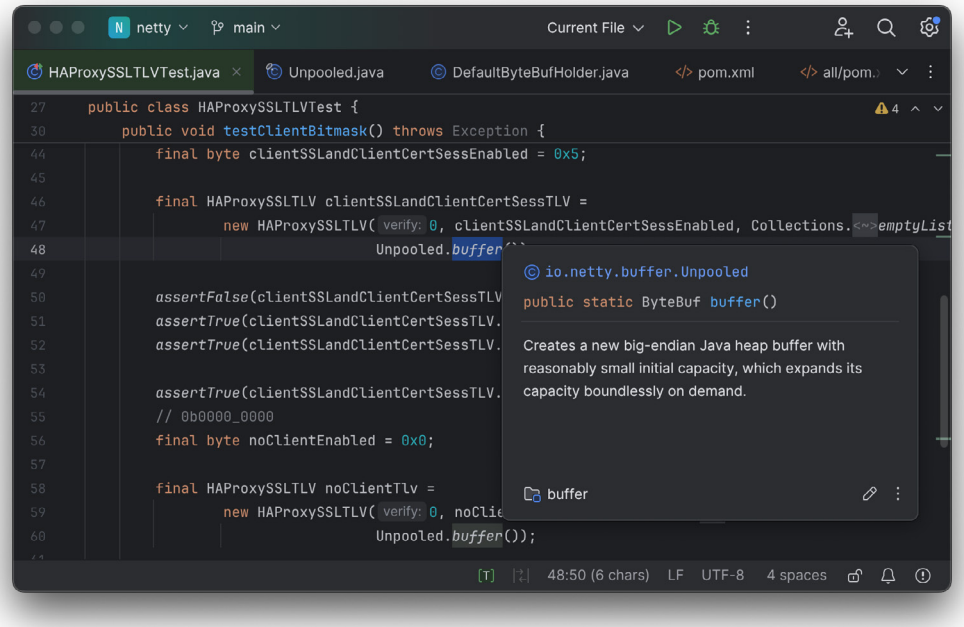
But if you asked your IDE to jump-to-definition to try and learn more about this flag, you will receive a screenful of de-compiled bytecode



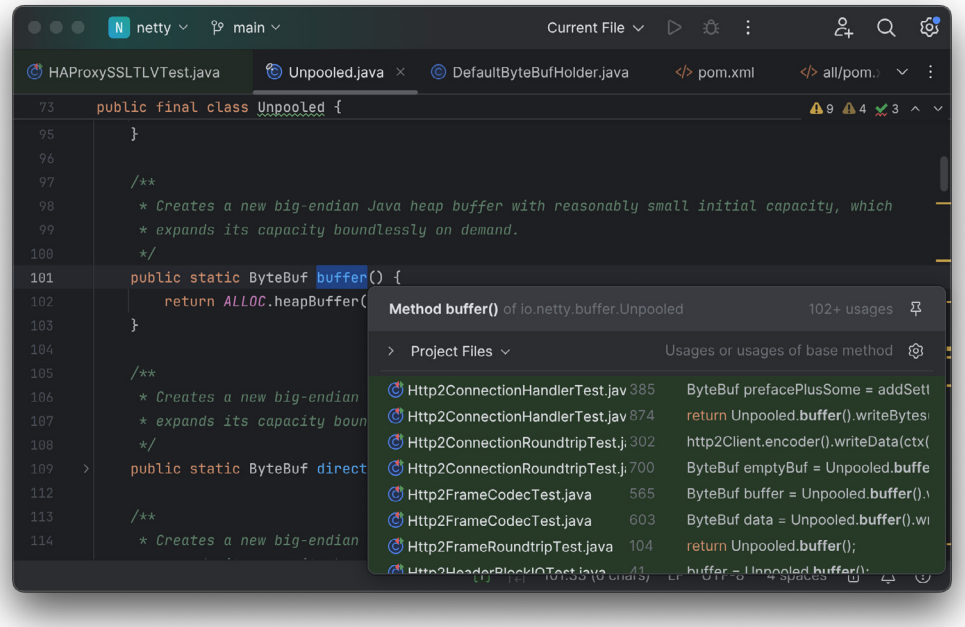
Not only is the de-compiled code missing all javadoc and comments and other niceties, but there is a more fundamental problem: the code is a simple *getter* and *setter*! All you know is that *someone* is *setting* this mutable variable, and *someone* is *getting* it, but these could be happening anywhere in the huge Gradle codebase, any of the third-party plugins you use, or anywhere in your own code. Although your IDE nominally lets you *jump to definition* in your Gradle build files, in practice it often isn't helpful in figuring out where the values actually come from.

While the Gradle example above is in Groovy, the experience with Gradle Kotlin is similar, despite Kotlin being a language with excellent IDE support in most other scenarios.

The reason there is untapped potential here is that this is *not* the IDE experience that someone expects from a JVM project! If I open a random Java file in any Java project, I expect to be able to pull up the documentation and signature of *any* identifier in that file with a single keypress:



And with another keypress, I expect to be able to *jump to definition* to see where the identifier comes from, and also *find usages* to see where the identifier is used throughout the program:



In a typical Java setup, this seamless *jump-to-definition* and *find usages* works throughout: your own code, third-party libraries, standard libraries, and so on. You can use this to get a deep understanding of any codebase quickly and conveniently in your IDE. But somehow that experience does not translate to build tools, and it is common to find yourself Google-ing for answers, reading and re-reading online docs that are always somehow not quite enough, and digging through plugin source code on Github. Can build tools do better, to really match the seamless IDE experience that Java developers are used to?

Extensibility

All build tools are extensible to some degree: even if 99% of the time you are doing standard things like compiling Java sources into classfiles and packaging them into jars, 1% of the time you actually do need to do something custom and unusual. Let's consider a simple requirement:

- *„Count the number of lines of code in the project and save it in a line-count.txt resource file"*

While this requirement may be contrived and arbitrary, it is representative of the many arbitrary things that any real-world project needs. Custom linters, custom deployment artifacts, custom BOM metadata, etc. are all things that real world build systems need to support. „counting the lines and saving it" is just the „hello world" version of these common real-world build customizations

Both Maven and Gradle can be extended to let you do this, but it is non-trivial. For example, in Maven you may come up with:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.1.0</version>
    <executions>
        <execution>
            <id>generate-line-count</id>
            <phase>generate-resources</phase>
            <goals
                <goal>exec</goal>
            </goals>
            <configuration>
                <executable>sh</executable>
                <arguments>
                    <argument>-c</argument>
                    <argument>
                        mkdir -p target/generated-resources
                        &amp;&amp;
```

```
                        find src -name ,*.java‘ | xargs wc -l >
                        target/generated-resources/line-count.txt
                    </argument>
                </arguments>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This snippet uses the `exec-maven-plugin` to count the lines of code using shell command find `src -name ,*.java‘ | xargs wc -l > line-count.txt`. This works, but there’s a lot of subtlety and trickiness around it:

1. You have to make sure to use `&&` rather than `;`, or `set -e`, otherwise errors may be silently ignored
2. You have to escape the `&&` as `&&` in XML in order to make it parse correctly
3. find only works on Mac and Linux, so this config as-written won’t work on Windows
4. What happens if some source files live outside of `src/`, e.g. Generated source files in `target/`?

You can also implement this line-count logic in Gradle, and it looks something like the following:

```
import java.io.File

tasks.register(„generateLineCount“) {
    val sourceDirs = listOf(„src/main/java“)
    val outputDir = layout.buildDirectory.dir(„generated-resources“)
    val outputFile = outputDir.get().file(„line-count.txt“)

    inputs.files(fileTree(„src/main“))
    outputs.file(outputFile)

    doLast {
```

```
        var totalLines = 0

        sourceDirs.map(::file).filter { it.exists() }.forEach { srcDir
          ->
        srcDir.walkTopDown()
                .filter { it.isFile && it.extension in listOf(„java“)}
                .forEach { file ->
                    totalLines += file.readLines().size
                }
        }

        outputFile.asFile.writeText(totalLines.toString())
        println(„Generated line-count.txt with $totalLines lines“)
    }
}

tasks.named(„processResources“) {
    dependsOn(„generateLineCount“)
    from(layout.buildDirectory.dir(„generated-resources“))
}
```

This is also tricky, but for different reasons than the Maven XML version. It runs as Kotlin on the JVM, so you don't need to worry about cross-platform support. But there are other issues:

1. You need to remember to keep `sourceDirs` in sync with `inputs.files`. In the example above, the two refer to different folders, which would result in unnecessary re-running of this tasks if e.g. `src/main/resources/` changes
2. You need to remember to add the `dependsOn` clause in `processResources`, otherwise you will find `generateLineCount` not re-running when it should resulting in a stale `line-count.txt`.
3. Again, `src/main/java` is not the only place where sources live! What if it's actually `src/main/kotlin`? What about generated sources in `target/`

The above code has a bug that can cause the `line-count.txt` to be spuriously re-computed if some files not in `src/main/java` are modified.

Can you spot it?

In general, the bugs in the Gradle case won't come from the line count logic: The `sourceDirs.map(::file)`, `walkTopDown`, `totalLines += file.readLines().size` section is verbose but straightforward. Rather, bugs would come from how this piece of code integrates with the rest of Gradle: the registration of tasks, registration of task dependencies, registration of input files and output files, all of which are done manually and be easily fat-fingered or fall out of sync as the code changes over time.

What is notable about „count lines and save it to a file" requirement is that it is *entirely trivial*. Any first-year programming student should be able to write it, and any professional programmer should be able to bang it out in their language of choice in about 30 seconds and have it work flawlessly. But the build tools like Maven or Gradle make this trivial task decidedly non-trivial. That's not to say it's impossible, but it's a lot harder than such a simple task should be!

Performance

The last area to discuss here is performance. Java is a very performant language, and the Java compiler written in Java is also very fast. But no Java build tool seems to be able to surface that speed to the developer.

For example, consider the time taken to clean compile a single module, `common`, in the 500kLOC Netty codebase

```
> ./mvnw clean; time ./mvnw -pl common -Pfast compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.204 s
[INFO] Finished at: 2025-03-25T13:53:43+01:00
[INFO] ------------------------------------------------------------------------
./mvnw -pl common -Pfast compile  17.53s user 1.05s system 301% cpu
6.153 total
```

The exact time taken will depend on hardware, OS, and Java versions. But when I ran it above it took about ~6s to compile this common module, which contains ~30kLOC

```
> find common/src/main/java | grep \\.java | xargs wc -l
...
   29712 total
```

6s to compile 30kLOC works out to be about ~5k lines per second. If we do another benchmark and try to compile the entire 500kLOC codebase, it takes about ~100s to compile on a single core:

```
> ./mvnw clean; time ./mvnw -Pfast  -Dcheckstyle.skip -Denforcer.
skip=true -DskipTests install
222.12s user 19.51s system 245% cpu 1:38.53 total
```

Again, we are looking at Maven compile ~5k lines of Java code per second. These numbers are with all dependencies already downloaded, with the ~/.m2 cache already populated, and with all linters and tests disabled via -DskipTests and -Pfast. We just want to look at compilation speeds.

In isolation that number seems fine: ~6s to compile a single moderately-sized module, ~100s to compile the entire project (this drops to ~50s if we parallelize it with -T10). But it's worth asking, how fast should Java code compile?

It turns out, Java compiles a lot faster than 5kLOC per second! If we go back to the common module we looked at earlier, and try compiling it directly with javac rather than going through

```
> time javac -d out/test \
    -cp /Users/lihaoyi/Github/netty/common/target/classes:/
    Users/lihaoyi/.m2/repository/org/graalvm/nativeimage/svm/19.3.6/
    svm-19.3.6.jar:/Users/lihaoyi/.m2/repository/org/graalvm/sdk/
    graal- sdk/19.3.6/graal-sdk-19.3.6.jar:/Users/lihaoyi/.m2/
    repository/org/graalvm/nativeimage/objectfile/19.3.6/objectfile-
    19.3.6.jar:/Users/lihaoyi/.m2/repository/org/graalvm/nativeimage/
    pointsto/19.3.6/pointsto-19.3.6.jar:/Users/lihaoyi/.m2/
```

```
    repository/org/graalvm/truffle/truffle-nfi/19.3.6/truffle-nfi-
    19.3.6.jar:/Users/lihaoyi/.m2/repository/org/graalvm/truffle/
    truffle-api/19.3.6/truffle-api-19.3.6.jar:/Users/lihaoyi/.m2/
    repository/org/graalvm/compiler/compiler/19.3.6/compiler-
    19.3.6.jar:/Users/lihaoyi/.m2/repository/org/jctools/jctools-
    core/4.0.5/jctools-core-4.0.5.jar:/Users/lihaoyi/.m2/repository/
    org/jetbrains/annotations-java5/23.0.0/annotations-java5-
    23.0.0.jar:/Users/lihaoyi/.m2/repository/org/slf4j/slf4j-
    api/1.7.30/slf4j-api-1.7.30.jar:/Users/lihaoyi/.m2/repository/
    commons-logging/commons-logging/1.2/commons-logging-1.2.jar:/
    Users/lihaoyi/.m2/repository/org/apache/logging/log4j/log4j-1.2-
    api/2.17.2/log4j-1.2-api-2.17.2.jar:/Users/lihaoyi/.m2/
    repository/org/apache/logging/log4j/log4j-api/2.17.2/log4j-api-
    2.17.2.jar:/Users/lihaoyi/.m2/repository/io/projectreactor/tools/
    blockhound/1.0.6.RELEASE/blockhound-1.0.6.RELEASE.jar \
    common/src/main/java/**/*.java
5.75s user 0.25s system 406% cpu 1.476 total
```

Above we are calling `javac` and passing in the (rather long) classpath along with a source file glob to the `javac` command line tool. This does effectively the same thing that Maven does internally, but it does it without Maven, and we are seeing a ~4x speedup as a result. This suggests that 3/4 of the time Maven spends on a `./mvnw compile` is actually just build tool overhead unrelated to the compilation itself.

But that's not all: `javac` is a Java program, and any experienced Java developer knows that running a Java program „cold" from the command line gets you the worst possible performance out of it. Java programs should be kept warm in-memory in a long-lived process so the JVM has time to JIT-compile and optimize the bytecode. If `javac` run cold from the commandline can compile Java at ~20kLOC/s, we would expect `javac` kept warm in a long-lived process to be much faster.

To test this, we can run javac in-memory using its `javax.tools.*` API that is available with most recent versions of the JDK. The `Bench.java` file below simply runs this over and over in a `while` loop and prints out the time taken:

```
// Bench.java
import javax.tools.*;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.nio.file.*;
import java.util.List;
import java.util.stream.Collectors;

public class Bench {
    public static void main(String[] args) throws Exception {
        String classpath = args[0];
        Path sourceFolder = Paths.get(args[1]);
        long lineCount = Files.walk(sourceFolder)
                .filter(p -> p.toString().endsWith(„.java“))
                .map(p -> {
                    try { return Files.readAllLines(p).size(); }
                    catch(Exception e){ throw new
                    RuntimeException(e); }
                })
                .reduce(0, (x, y) -> x + y);

        while (true) {
            List<JavaFileObject> files = Files.walk(sourceFolder)
                    .filter(p -> p.toString().endsWith(„.java“))
                    .map(p ->
                            new SimpleJavaFileObject(p.toUri(),
                            JavaFileObject.Kind.SOURCE) {
                                public CharSequence
                                getCharContent(boolean
                                ignoreEncodingErrors) throws
                                IOException {
                                    return Files.readString(p);
                                }
                            }
                    )
                    .collect(Collectors.toList());

            long now = System.currentTimeMillis();
```

```
            JavaCompiler compiler = ToolProvider.
            getSystemJavaCompiler();

            StandardJavaFileManager fileManager = compiler
                    .getStandardFileManager(null, null, null);

            // Run the compiler
            JavaCompiler.CompilationTask task = compiler.getTask(
                    new OutputStreamWriter(System.out),
                    fileManager,
                    null,
                    List.of(„-classpath“, classpath, „-d“, „target/
                    bench“),
                    null,
                    files
            );

            System.out.println(„Compile Result: „ + task.call());
            long end = System.currentTimeMillis();

            System.out.println(„Lines: „ + lineCount);
            System.out.println(„Duration: „ + (end - now));
            System.out.println(„Lines/second: „ + (int)(lineCount /
            ((end - now) / 1000.0)));
        }
    }
}
```

This can be run as follows:

```
> java Bench.java \
  /Users/lihaoyi/Github/netty/common/target/classes:/Users/lihaoyi/.
  m2/repository/org/graalvm/nativeimage/svm/19.3.6/svm-19.3.6.jar:/
  Users/lihaoyi/.m2/repository/org/graalvm/sdk/graal-sdk/19.3.6/
  graal-sdk-19.3.6.jar:/Users/lihaoyi/.m2/repository/org/graalvm/
  nativeimage/objectfile/19.3.6/objectfile-19.3.6.jar:/Users/lihaoyi/.
  m2/repository/org/graalvm/nativeimage/pointsto/19.3.6/pointsto-
  19.3.6.jar:/Users/lihaoyi/.m2/repository/org/graalvm/truffle/truffle-
  nfi/19.3.6/truffle-nfi-19.3.6.jar:/Users/lihaoyi/.m2/repository/org/
```

```
graalvm/truffle/truffle-api/19.3.6/truffle-api-19.3.6.jar:/Users/
lihaoyi/.m2/repository/org/graalvm/compiler/compiler/19.3.6/
compiler-19.3.6.jar:/Users/lihaoyi/.m2/repository/org/jctools/
jctools-core/4.0.5/jctools-core-4.0.5.jar:/Users/lihaoyi/.m2/
repository/org/jetbrains/annotations-java5/23.0.0/annotations-
java5-23.0.0.jar:/Users/lihaoyi/.m2/repository/org/slf4j/slf4j-
api/1.7.30/slf4j-api-1.7.30.jar:/Users/lihaoyi/.m2/repository/
commons-logging/commons-logging/1.2/commons-logging-1.2.jar:/Users/
lihaoyi/.m2/repository/org/apache/logging/log4j/log4j-1.2-
api/2.17.2/log4j-1.2-api-2.17.2.jar:/Users/lihaoyi/.m2/repository/
org/apache/logging/log4j/log4j-api/2.17.2/log4j-api-2.17.2.jar:/
Users/lihaoyi/.m2/repository/io/projectreactor/tools/
blockhound/1.0.6.RELEASE/blockhound-1.0.6.RELEASE.jar \
common/src/main/java
```

If you do this, you will find it start of slow but gradually speed up over time. Running this for about ~30s on my laptop results in the following output near the end:

```
Lines: 29712
Duration: 251
Lines/second: 118374

Lines: 29712
Duration: 253
Lines/second: 117438
```

So while Maven compiles Java code at ~5kLOC/s, and `javac` run from the command line compiles at ~20kLOC/s, `javac` run in-memory and allowed to get hot compiles at almost ~120kLOC/s!

If we repeat this exercise with Gradle, using the Mockito codebase as an example, we get similar results, which I tabulated below

Mockito Core	Time	Compiler lines/s	Slow-down	Netty Common	Time	Compiler lines/s	Slow-down
Javac Hot	0.36s	115,600	1.0x	Javac Hot	0.25s	117,500	1.0x
Javac Cold	1.29s	32,220	4.4x	Javac Cold	1.48s	20,100	5.1s
Gradle	4.41s	9,400	15.2x	Maven	6.15s	4,800	24.6x

So Maven and Gradle actually compile Java code 15-20x slower than `javac` itself is able to do so! While Netty's ~500kLOC compiles in ~100s with Maven, it *should* compile in 4-5s if compiled using Javac directly. Java compiles *should* be basically instant even for large codebases like Netty, but build tools like Maven or Gradle add enough overhead that you can really feel the slowness.

The Mill Build Tool

Mill is a fast, scalable, multi-language build tool that supports Java, Scala, Kotlin. At its core, it does many of the same things as Maven or Gradle, but tries hard to improve upon the IDE experience, extensibility, and performance issues described above. You define a `build.mill` file as below:

```
// build.mill
package build
import mill._, javalib._

object foo extends JavaModule {
  def ivyDeps = Agg(
    ivy“net.sourceforge.argparse4j:argparse4j:0.9.0“,
    ivy“org.thymeleaf:thymeleaf:3.1.1.RELEASE“
  )

  object test extends JavaTests with TestModule.Junit4
}
```

And you can use this build file to compile, test, run, and generate assemblies of your project:

```
> ./mill foo.compile
compiling 1 Java source...

> ./mill foo.run --text hello
<h1>hello</h1>

> ./mill foo.test
Test foo.FooTest.testEscaping finished, ...
Test foo.FooTest.testSimple finished, ...
0 failed, 0 ignored, 2 total, ...

> ./mill show foo.assembly
„.../out/foo/assembly.dest/out.jar“

> ./out/foo/assembly.dest/out.jar --text hello
<h1>hello</h1>
```

Mill can build any Java application if configured with the appropriate dependencies. For example, the following `build.mill` configures a Spring-Boot TodoMVC application with all bells and whistles: `data-jpa, thymeleaf, validation, jaxb-api, webjars` for the CSS and Javascriptc, as well as unit in-memory tests using H2 and integration tests using Dockerized Postgres in TestContainers:

```
// build.mill
package build
import mill._, javalib._

object `package` extends RootModule with JavaModule {
  def ivyDeps = Agg(
    ivy“org.springframework.boot:spring-boot-starter-data-jpa:2.5.4“,
    ivy“org.springframework.boot:spring-boot-starter-
    thymeleaf:2.5.4“,
    ivy“org.springframework.boot:spring-boot-starter-
    validation:2.5.4“,
    ivy“org.springframework.boot:spring-boot-starter-web:2.5.4“,
    ivy“javax.xml.bind:jaxb-api:2.3.1“,
    ivy“org.webjars:webjars-locator:0.41“,
    ivy“org.webjars.npm:todomvc-common:1.0.5“,
```

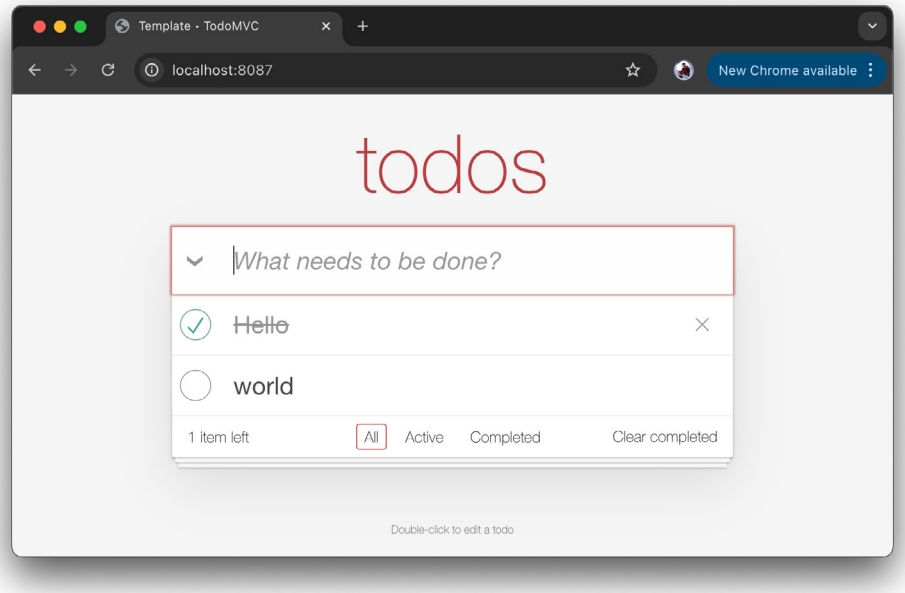
```
    ivy“org.webjars.npm:todomvc-app-css:2.4.1“
  )

  trait CommonTestModule extends JavaTests with TestModule.Junit5 {
    def mainClass = Some(„com.example.TodomvcApplication“)
    def ivyDeps = super.ivyDeps() ++ Agg(
      ivy“org.springframework.boot:spring-boot-starter-test:2.5.6“
    )
  }
  object test extends CommonTestModule {
    def ivyDeps = super.ivyDeps() ++ Agg(
      ivy“com.h2database:h2:2.3.230“
    )
  }
  object integration extends CommonTestModule {
    def ivyDeps = super.ivyDeps() ++ Agg(
      ivy“org.testcontainers:testcontainers:1.18.0“,
      ivy“org.testcontainers:junit-jupiter:1.18.0“,
      ivy“org.testcontainers:postgresql:1.18.0“,
      ivy“org.postgresql:postgresql:42.6.0“
    )
  }
}
```

This can be run locally from the command line:

```
> mill test
...com.example.TodomvcTests#homePageLoads() finished...
...com.example.TodomvcTests#addNewTodoItem() finished...

> mill integration
...com.example.TodomvcIntegrationTests#homePageLoads() finished...
...com.example.TodomvcIntegrationTests#addNewTodoItem() finished...

> mill test.runBackground

> curl http://localhost:8087
...<h1>todos</h1>...
```

```
> mill clean runBackground
```

Or interacted with in the browser:



So far, nothing we have seen here is unusual: these are just things that any Maven or Gradle project can do. So what value does Mill provide as a build tool?

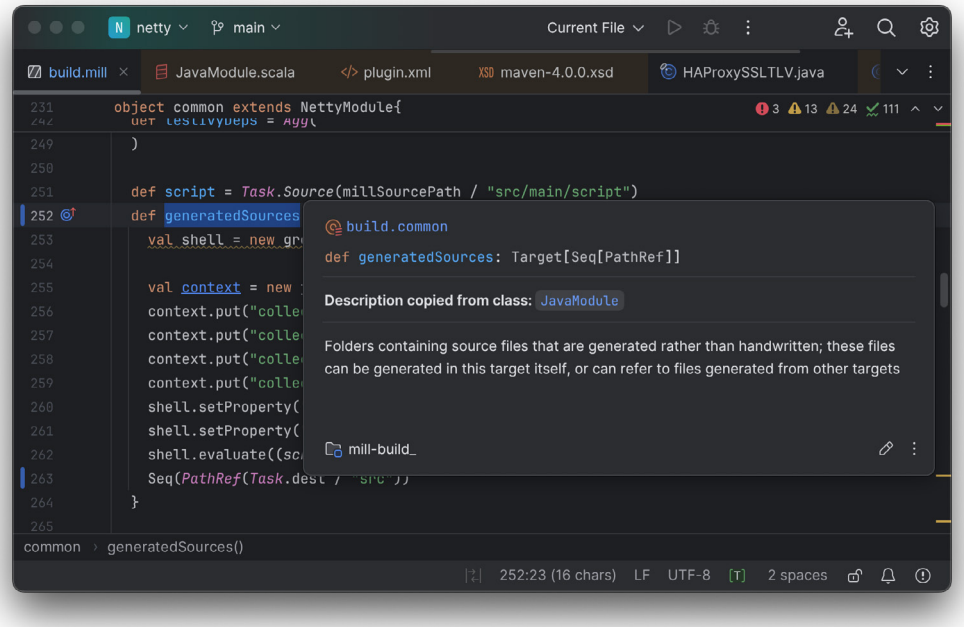
How Mill Improves Upon the Java Build Tool Experience

To look at how Mill improves upon things, we will consider the same three areas we looked at earlier: IDE experience, extensibility, and performance:

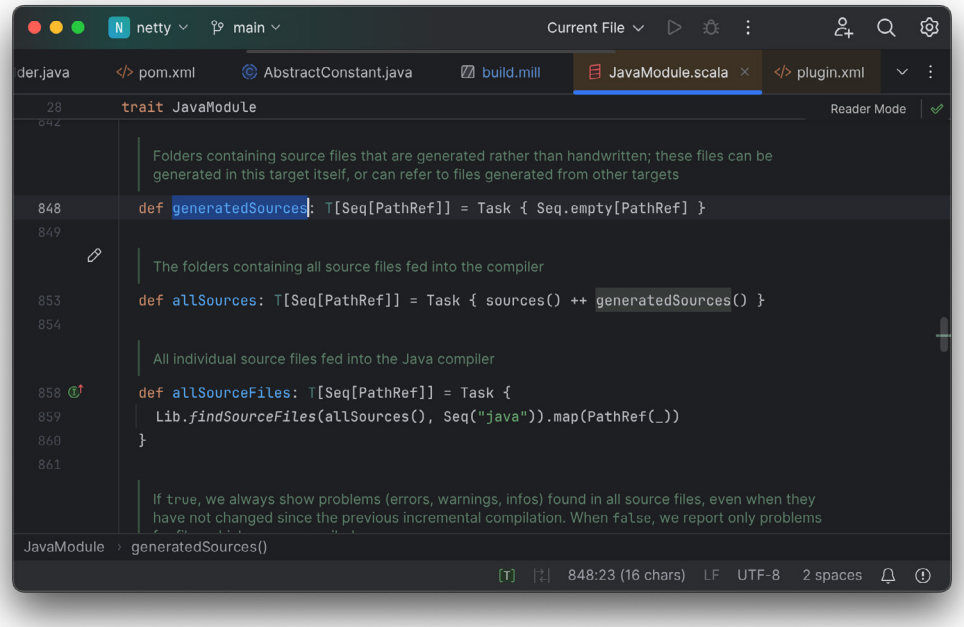
IDE Experience

In terms of IDE experience, Mill has support in IDEs like IntelliJ or VSCode like Maven or Gradle do, but that support turns out to be much more useful.

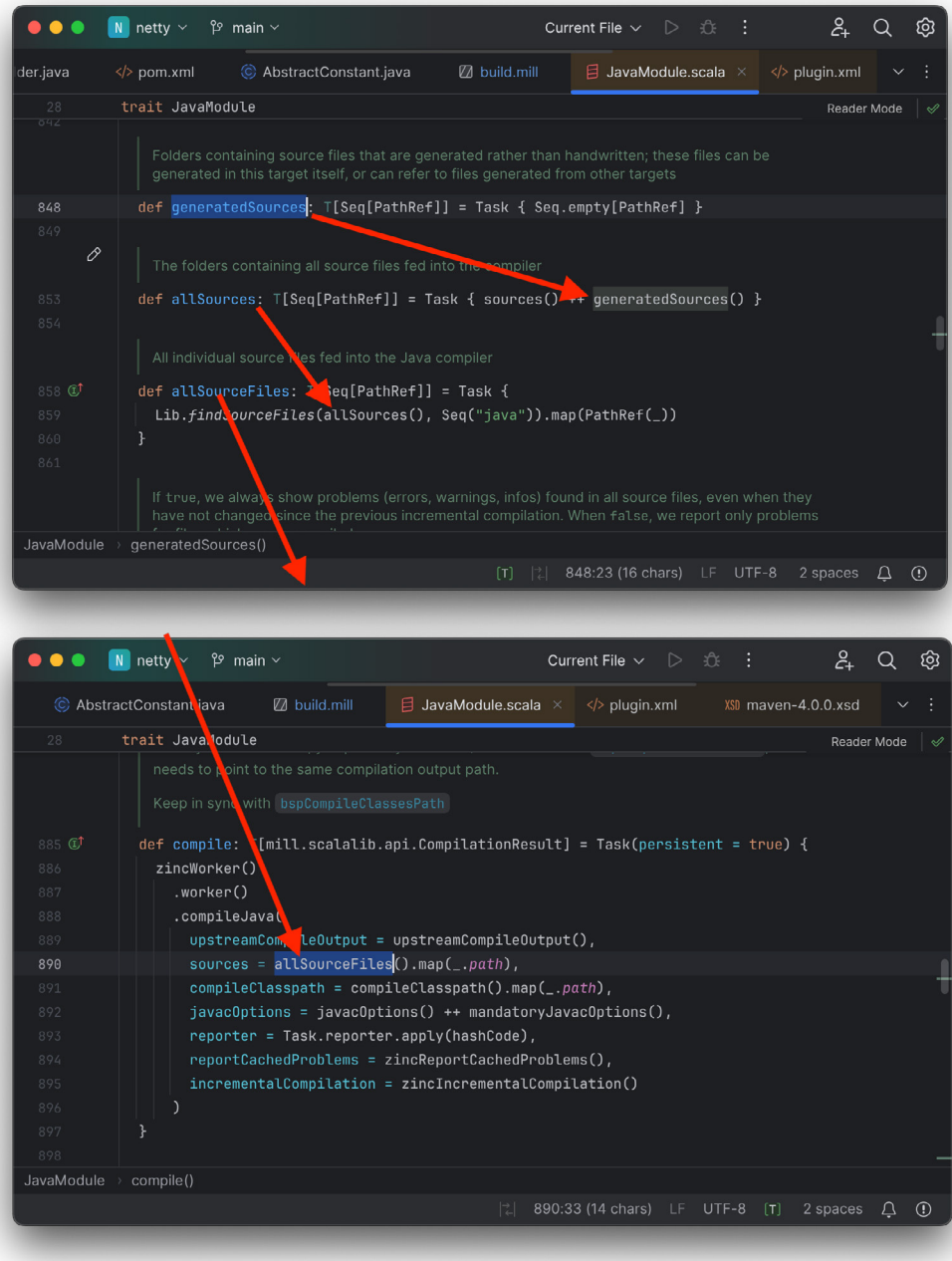
Earlier, we looked a Maven and Gradle build, and saw how the IDE experience of exploring the configuration was shallow: it told us what types things were, but not how these configuration values came from or how they were actually used. Mill's experience is different. For example, consider the following custom task that creates some generated sources:



Right off the bat, you can mouse over the task and see the documentation inherited from the overridden task. If you want to learn more, you can jump to definition to see where the overridden task is defined:



From there, you can see how this `generatedSources` task ends up being used in `allSources`, `allSourceFiles`, and finally in `compile`:



The Mill build logic is no simpler than Maven or Gradle: it still needs to process the input configuration to decide how to correctly build your project. Where Mill differs is that the build logic is navigate-able in your IDE: you can see how these configuration values, files, and tasks combine together to produce the output artifacts you want. While in the example above we stopped after seeing how `def generatedSources` feeds into def compile, we can continue to explore the build logic arbitrarily deeply if we desired: looking at the implementation of `.findSourceFiles`, `.compileJava`, and so on.

This IDE experience is not new: it is what Java developers everywhere are experiencing every day working on their application code! But Mill is able to bring this IDE experience to your build tool in a way that other build tools like Maven or Gradle cannot, which greatly helps anyone who

is trying to debug issues with their build and understand why it is acting the way it does.

Perhaps the key insight here is that Mill tasks and configuration values are *just methods*! The Mill build is just made of methods calling other methods to form a call-graph, which Mill uses as the foundation for tasks calling other tasks forming a build-graph. And IDEs like IntelliJ or VSCode already know how to deal with method defs, method calls with ()s, and even more advanced features like `override` and `super`. As far as your IDE is concerned your Mill build logic looks just like any application codebase made of methods calling each other, and so the IDE can provide the same deep understanding and code navigation that it provides for any Java application codebase.

Extensibility

Most build tools require extensions to be written as plugins: you end up assembling a collection of plugins off of Github of varying quality, maintenance, and fit, and trying to coerce those plugins into doing what you want. Mill is different in that it allows you to just *write code* and *use any Java library from Maven Central* to configure your build and do what you want.

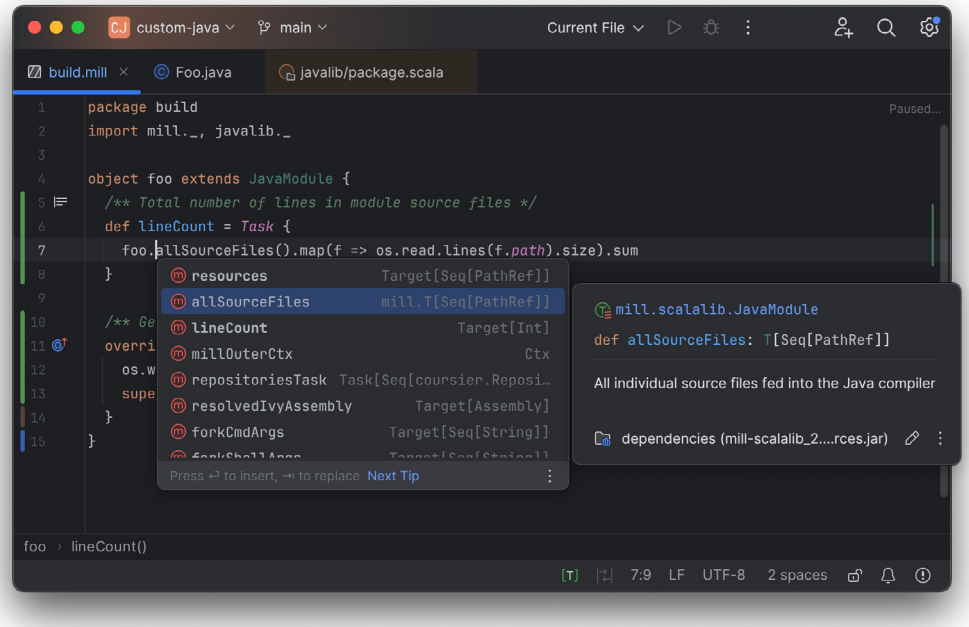
In Mill, customizing a module to add a generated `line-count.txt` resource file looks like this:

```
object foo extends JavaModule {
  /** Total number of lines in module source files */
  def lineCount = Task {
    foo.allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  /** Generate resources using lineCount of sources */
  override def resources = Task {
    os.write(Task.dest / „line-count.txt“, „“ + lineCount())
    super.resources() ++ Seq(PathRef(Task.dest))
  }
}
```

One method `def` to compute the line count, another `override def` to replace resources, and that's it. We see the same the business logic here as in the Gradle/Kotlin example earlier, perhaps slightly simplified (the `allSourceFiles()` method already handles the recursive listing of `.java` files for us so we don't need to repeat it here). We can then use `./mill show foo.lineCount` to see the task's return value directly, or `./mill foo.run` to run the application code that makes use of the `line-count.txt` resource file:

```
> mill show foo.lineCount
17

> mill foo.run
Line Count: 17
```

What's notable here is what we *don't* see here: `doLast`, `dependsOn`, `inputs.files`, `outputs.files`, hard-coded references to paths like `„src/main“` and `„src/main/java“`. With Mill, you just write the business logic of what your build tasks need to do: `def lineCount` reads the lines of each file and sums them up, `override def resources` writes out the `line-count.txt` file and adds a path reference to it to the resource path. All the manual work done in Gradle to register task dependencies, register input-output files, decide when the task will run, etc. is all done automatically for you in Mill. And it is done with full IDE support, so if you're uncertain about what methods are available or what they do, your IDE is happily to provide real-time autocomplete and documentation to assist you:



Build-time HTML Rendering

Most Java applications use more than just the standard library to do their work, and build systems are no different: you often need custom code-generators, IDL parsers, and other third-party libraries in your build. Most build systems require these libraries to be bundled as plugins, which adds extra indirection and complexity (is there a plugin on Github? Does the plugin do everything I need? Is the plugin well maintained? Do I need to fork it or write my own plugin?) on top of the existing complexity of the underlying library

Mill takes a different approach: you can directly use any third-party Java library as part of your Mill build. This removes a layer of indirection, and allows Java developers to directly use the same Java libraries they are familiar with without needing to first wrap them in a plugin or find an existing wrapper.

For example, let's say we had a new requirement that the `line-count.txt` should be rendered as a HTML string. Mill as a build tool does not come with HTML templating libraries built in, but it makes it very easy to import such libraries using the `import $ivy` syntax:

```
package build
import mill._, javalib._
import $ivy.`org.thymeleaf:thymeleaf:3.1.1.RELEASE`
import org.thymeleaf.TemplateEngine
import org.thymeleaf.context.Context
object foo extends JavaModule {
   def lineCount = Task {
     foo.allSourceFiles().map(f => os.read.lines(f.path).size).sum
   }

   def htmlSnippet = Task {
    val context = new Context()
    context.setVariable(„lineCount“, „“ + lineCount())
    new TemplateEngine().process(„<h1 th:text=\“${lineCount}\“></
    h1>“, context)
  }
```

```
    def resources = Task.Sources{
      os.write(Task.dest / „snippet.txt“, htmlSnippet())
      super.resources() ++ Seq(PathRef(Task.dest))
    }
  }
```

In this snippet, we `import $ivy the org thymeleaf:thymeleaf:-3.1.1.RELEASE` library, which immediately makes it available in our build config. We can then directly `import org.thymeleaf.TemplateEngine and org.thymeleaf.context.Context` and use it just as we would use it in any Java application. In this example, we are using it to pre-render a `<h1>{lineCount}</h1>` html snippet in `line-count.txt` for our application to use at runtime. We can then use `./mill show` or `./mill foo.run` to see the value being generated and used at runtime:

```
> mill show foo.htmlSnippet
„<h1>17</h1>“

> mill foo.run
Line Count: <h1>17</h1>
```

What's interesting about this example is that there are no plugins here:

- No `mill-thymeleaf-plugin` to integrate HTML rendering into your build pipeline
- No `mill-linecount-plugin` to count the lines of code
- No `mill-generated-resources` plugin to generate resource files that can be read at runtime.

Instead, Mill lets you directly write code to do exactly what you want, using the common open source Java libraries you already know how to use. This democratizes your build so that any Java developer can configure it, rather than being limited to those that hold the title of „build tool expert“ or „plugin author“.

Declarative vs Imperative Configuration

The last thing worth mentioning on the topic of extensibility is the idea of „declarative" vs „imperative" builds: Maven using XML is usually thought of as „declarative", while Gradle is considered „imperative". But the line between the two is a lot blurrier than people realize. For example, consider again the Maven line-count implementation we saw earlier:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.1.0</version>
    <executions>
        <execution>
            <id>generate-line-count</id>
            <phase>generate-resources</phase>
            <goals>
                <goal>exec</goal>
            </goals>
            <configuration>
                <executable>sh</executable>
                <arguments>
                    <argument>-c</argument>
                    <argument>
                        mkdir -p target/generated-resources
                        &amp;&amp;
                        find src -name ‚*.java‘ | xargs wc -l >
                        target/generated-resources/line-count.txt
                    </argument>
                </arguments>
            </configuration>
        </execution>
    </executions>
</plugin>
```

While nominally a „declarative" XML config, it actually contains Bash code that is as imperative as any code ever written:

```
mkdir -p target/generated-resources &&
find src -name ,*.java‘ | xargs wc -l > target/generated-resources/
line-count.txt
```

This imperative Bash code is functionally equivalent to the Mill configuration we saw earlier:

```
def lineCount = Task {
  allSourceFiles().map(f => os.read.lines(f.path).size).sum
}

override def resources = Task {
  os.write(Task.dest / „line-count.txt“, „“ + lineCount())
  super.resources() ++ Seq(PathRef(Task.dest))
}
```

And is also equivalent to part of the Gradle Kotlin code we saw:

```
sourceDirs.map(::file).filter { it.exists() }.forEach { srcDir ->
    srcDir.walkTopDown()
        .filter { it.isFile && it.extension in listOf(„java“) }
        .forEach { file ->
            totalLines += file.readLines().size
        }
}

outputFile.asFile.writeText(totalLines.toString())
```

The lesson here is that build configuration does have some *intrinsic complexity*. There are things you want your build tool to do, and you need to somehow tell your build tool to do it. Whether you end up writing those instructions in a Maven Bash snippet, in Mill code, or Gradle Kotlin code, that logic needs to exist *somewhere*. Just because you wrap your imperative Bash script in a screenful of declarative XML does not make that Bash script any less imperative, or help mitigate the fragile and non-portable nature of the Bash language!

This also calls into question why Gradle code is confusing. Common wisdom says that Gradle code is confusing because it is imperative, but

if you look at the Gradle snippet above it is no more complex than the Mill snippet or the Maven bash snippet: a student or junior developer should have no problems writing any of these snippets without error. The error-prone part of the Gradle configuration isn't the *business logic* of counting lines of code, but the *plumbing*: manually registering input files, output files, task dependencies, and so on to make it fit into the Gradle framework:

```
import java.io.File

tasks.register(„generateLineCount“) {
    val sourceDirs = listOf(„src/main/java“)
    val outputDir = layout.buildDirectory.dir(„generated-resources“)
    val outputFile = outputDir.get().file(„line-count.txt“)

    inputs.files(fileTree(„src/main“))
    outputs.file(outputFile)

    doLast {
        ...
    }
}

tasks.named(„processResources“) {
    dependsOn(„generateLineCount“)
    from(layout.buildDirectory.dir(„generated-resources“))
}
```

While anyone can write a line-count program without issue, I would expect most developers to have to spend significant amounts of time Googling and poring over documentation to figure out how to write the snippet above, and even more time to convince themselves they have done it correctly!

Arguably the problem with Gradle isn't so much that the configuration is code: it is that the configuration is very low-level code that forces the user to manually perform a lot of steps that are both tedious and easy to get wrong. So while Mill's build config is also code, the fact that it automates away all this tedious boilerplate makes the def lineCount and override

def resources Mill implementation easier to read and maintain than both the Maven and Gradle versions.

Performance

Compilation Performance

Mill can compile the same Java module much faster than Maven or Gradle. For example, we can compile the Netty `common` module we discussed earlier via

```
> ./mill clean common; time ./mill common.compile
0.02s user 0.05s system 6% cpu 1.114 total
```

Or compile the entire 500kLOC Netty codebase on a single thread via

```
> ./mill clean && time ./mill -j1 __.compile
0.25s user 0.79s system 4% cpu 22.782 total
```

Below we extend the table we saw earlier, and repeat the Mill measurements on the same Mockito codebase we used to test Gradle. we can see that although Mill only matches the performance of `Javac Cold`, and still has significant overhead over `Javac Hot`, it is able to compile the same code much faster than Maven or Gradle:

Mockito Core	Time	Compiler lines/s	Slow-down	Netty Common	Time	Compiler lines/s	Slow-down
Javac Hot	0.36s	115,600	1.0x	Javac Hot	0.25s	117,500	1.0x
Javac Cold	1.29s	32,220	4.4x	Javac Cold	1.48s	20,100	5.1s
Gradle	4.41s	9,400	15.2x	Maven	6.15s	4,800	24.6x
Mill	**1.20s**	**34,700**	**4.1s**	**Mill**	**1.11s**	**26,800**	**3.8x**

Mill compiling the entire Netty codebase in ~23s on a single thread still does not live up to the ~4-5s that extrapolating our single-module benchmarks would suggests, but is a significant ~4x improvement over Maven compiling the codebase in ~100s on a single core! Similarly, compiling Netty's `common` module in 1.11s is a big improvement over Maven compiling it in 6.15s, even if not quite as fast as `Javac Hot` compiling it

in 0.29s.

If we benchmark a variety of scenarios, we see that the speedup is pretty consistent regardless of whether you run on a single-thread or in parallel, whether you compile the entire codebase or just a single module, and whether you do clean or incremental compiles. In fact, Mill's performance stands out even more for incremental compiles, where it is able to compile and return in ~0.2s where Gradle takes >1s and Maven several seconds!

Benchmark	Maven	Mill	Speed Up
Sequential Clean Compile All	98.80s	23.41s	4.2x
Parallel Clean Compile All	48.92s	9.29s	5.3x
Clean Compile Single Module	4.89s	0.88s	5.6x
Incremental Compile Single Module	6.82s	0.18s	37.9x
No-Op Compile Single Module	5.25s	0.12s	43.8x

Benchmark	Gradle	Mill	Speed Up
Sequential Clean Compile All	17.6s	5.86s	3.0x
Parallel Clean Compile All	12.3as	3.75s	3.3x
Clean Compile Single Module	4.41s	1.30s	3.4x
Incremental Compile Single Module	1.37s	0.20s	6.9s
No-Op Compile Single Module	0.94s	0.11s	8.5s

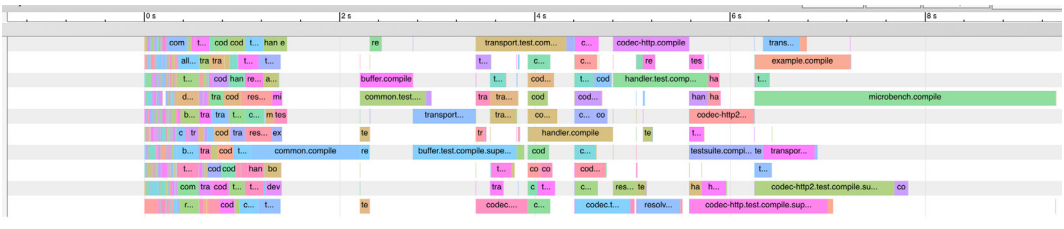
Other Performance Features

Compilation speed is just one aspect of a build tool's performance, and while it may be the easiest to measure, it is by no means the most important. Mill

- **--watch and re-run:** automatically re-run tests or services when code changes, saving time having to manually click the „run" button or tab back to your terminal to repeat a command
- **Parallel Testing:** this lets you use all cores on your machine to speed up testing, which makes an enormous difference in today's multi-

core world

- **Selective Test Execution:** speeds up CI by only running tests affected by code being changed. This can cut hour-long PR validation down to minutes by skipping unrelated tests
- **Incremental Assembly Jar Creation:** cuts down the creation of large assembly jars from 20s to 1s, which really speeds up workflows that use them (e.g. manual testing, or `spark-submit`)
- **Automatic Parallel Profile Generation:** mill automatically generates profile files for every command, that you can load into your `chrome://tracing` profiler included in any chrome browser to visualize where the time in your build is being spent



Conclusion

In this article, we have looked at the ways in which Java build tools like Maven and Gradle fall short of Java's reputation as a performant and easy-to-use language with great IDE support. While the performance, usability, and IDE integrations with these build tools are ok, they are not great, and are far from what you would expect working within the application code of any Java codebase.

We then introduced the Mill build tool and how it is able to improve upon the shortcomings of tools like Maven or Gradle: 3-6x faster compiles, easier extensibility, and an IDE experience that allows you to navigate your build logic and as easily as any Java library. While there is still room to improve on all counts, it nevertheless shows a significant step up from tools like Maven or Gradle.

JVM build tools definitely have a lot of room to improve. While Mill takes some baby steps in that direction, there's a lot more work to be done to give Java developers a build tool experience worthy of the platform it runs on.

> Back to Table of Content

#JAVAPRO #AI #ML

Tame Your Llama: Run AI in Java

Author:

Lutske de Leeuw is a Software Engineer at Craftsmen, specializing in full-stack development with a focus on Java and Angular. Passionate about machine learning, board games, and cats. Committed to sharing knowledge so that everyone can benefit. Loves organizing events like Devoxx4Kids, ApeldoornJUG, JUG Noord, and other knowledge-sharing initiatives.

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

Introduction to AI in Java

What is Llama?

Llama is an advanced open-source AI language model developed by Meta, designed for natural language understanding and generation. Unlike cloud-based AI models, Llama can be run locally, providing a powerful alternative for Java developers who want to integrate AI into their applications without relying on external services. The ability to run Llama models locally means developers can harness AI without exposing data to third-party providers, ensuring privacy and security.

Llama stands out due to its efficiency in processing language tasks while maintaining performance suitable for local execution. With models like Llama 2-7B, developers can leverage AI-driven capabilities in text processing, automation, and decision-making, all within their Java applications. The combination of Java and AI allows for intelligent systems that enhance user interactions and automate complex workflows.

The Local AI Advantage

Running AI models locally presents several advantages. Privacy is a key benefit, no data leaves your machine, making it suitable for applications that require confidentiality. Cost-efficiency is another major factor, as local AI eliminates the need for expensive API calls to cloud-based services. Performance also improves since latency is reduced when accessing the model directly. Finally, independence from third-party providers ensures uninterrupted service, even if cloud APIs change or become deprecated.

With local execution, developers avoid unpredictable pricing models of cloud services. AI-driven applications become more predictable in terms of performance and expenses. Additionally, running Llama locally means applications can function offline, a crucial feature for environments with limited internet access or strict regulatory requirements.

Setting Up Llama Locally

System Requirements

Before running Llama on your local machine, ensure you have the following:

- **A modern CPU** with AVX2 support or an **NVIDIA GPU with CUDA** for faster inference.
- **At least 16GB of RAM** (32GB recommended for larger models).
- **Adequate disk space** (~10GB for models like Llama 2-7B).
- **Java 11+** (preferably Java 17 or later).
- A compatible JDK (e.g., OpenJDK or Amazon Corretto).

- **Ollama4J,** a Java binding for the Ollama framework, which facilitates local execution of Llama models.

Installation Walkthrough

Setting up Llama locally is straightforward. First, download and install Ollama from Ollama Download. Once installed, pull the Llama model using the following command:

ollama pull llama2

Next, add Ollama4J to your Java project. If you're using Maven, include this dependency in your pom.xml:

```
<dependency>
    <groupId>io.github.ollama4j</groupId>
    <artifactId>ollama4j</artifactId>
    <version>1.0.98</version>
</dependency>
```

Verify the installation by running ollama list to ensure the model is available.

Note: Ollama can also be run using Docker, but this article focuses on running it natively on your system.

Integrating Llama into a Java Application

Ollama4J provides an intuitive API for interacting with local Llama models. Once installed, you can start querying the model in just a few lines of Java code.

Using AI in Java applications has historically been complex due to the lack of direct integration with deep learning frameworks. However, Ollama4J simplifies this process by providing Java-friendly APIs that remove the need for dealing with Python or external AI services. This makes it easier to build AI-powered applications while leveraging Java's ecosystem.

Expanding AI Capabilities in Java

Llama can be used for:

- **Text Summarization:** Extracting key points from large text bodies.
- **Code Generation:** Assisting developers by generating code snippets.
- **Chatbots and Assistants:** Creating interactive AI-driven assistants.
- **Content Moderation:** Identifying and filtering inappropriate or harmful content.
- **Personalized Recommendations:** Tailoring responses based on user input.

These capabilities can be seamlessly integrated into Java applications to create smarter, more responsive systems.

Performance Optimization for Llama in Java

While running AI models locally is powerful, performance optimization is necessary to ensure smooth execution. When integrating Llama into Java applications, consider the following:

- **Memory Management:** Allocate sufficient heap space using JVM options (-Xmx16G).
- **Concurrency Handling:** Use multi-threading to parallelize tasks.
- **Caching Responses:** Store common queries to reduce computation time.
- **Lazy Loading:** Load AI models only when needed to conserve resources.

By applying these techniques, developers can ensure efficient AI execution without overwhelming system resources.

Challenges and Best Practices

Handling Large Models in Java

Running large AI models in Java requires careful memory management. Increasing the heap size using JVM options like -Xmx16G helps accommodate larger models. Lazy loading and asynchronous processing also improve performance:

```
CompletableFuture.supplyAsync(() -> ollama.generate(„llama2“,
„Explain Java Streams.“))
    .thenAccept(System.out::println);
```

Additionally, developers should be mindful of garbage collection (GC) behavior when working with large AI models. Tuning GC settings and monitoring memory usage can help prevent performance bottlenecks.

Security and Privacy Concerns

When using local AI, developers must consider data sanitization to remove sensitive information before processing. Implementing access control mechanisms ensures only authorized users can interact with the AI. Additionally, logging and auditing AI interactions help maintain oversight and traceability.

Another security aspect is model integrity. Ensuring that the AI model has not been tampered with before deployment is critical. Hash verification of downloaded models can be implemented to safeguard against unauthorized modifications.

Choosing the Right AI Model

Llama is ideal for privacy-focused, offline AI solutions. However, for cutting-edge advancements with massive datasets, cloud-based models like GPT-4 may be preferable. Developers should assess their specific needs, balancing performance, cost, and data security.

When choosing between different Llama model versions, developers should test their applications with various configurations to find the

best trade-off between accuracy and performance. For lightweight applications, smaller models like Llama 2-7B may suffice, while enterprise solutions may require larger, more capable versions.

Conclusion

Java developers now have an easy way to integrate AI into their applications with Llama. Running models locally with Ollama4J allows for privacy-focused, cost-effective, and highly responsive AI-driven features. Whether you're building chatbots, automating workflows, or enhancing user experiences, Llama provides a robust foundation for AI in Java.

By following the steps outlined in this article, you can start leveraging AI in your projects today. The power of AI is now at your fingertips. Go tame your Llama and bring intelligence to your Java applications!

Llama and Java together create new opportunities for AI-driven development, bridging the gap between traditional enterprise applications and modern AI-powered solutions. The ability to run powerful models locally ensures that developers can build intelligent systems without compromising data privacy or application performance. As AI continues to evolve, Java remains a strong candidate for enterprise-ready AI integration, making Llama an exciting addition to the Java ecosystem.

> Back to Table of Content

#JAVAPRO #ARCHITECTURE #MICROSERVICES

Mastering the Basics of Domain-Driven Design with Java

Author:

Otavio Santana is an award-winning software engineer and architect who is passionate about empowering other engineers with open-source best practices to build highly scalable and efficient software. He is a renowned contributor to the Java and open-source ecosystems and has received numerous awards and accolades for his work. Otavio's interests include history, the economy, travel, and fluency in multiple languages, all seasoned with a great sense of humor.

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

Domain-driven design (DDD) is a critical approach in software development, yet its essence often gets buried under layers of complexity and misunderstanding. At its core, DDD aims to align software design with the needs of the business or stakeholders, ensuring that what we build addresses real problems. Even that sounds like a cliche in software development, and obviously, it is still a challenge to most organizations.

One standout example is *Forbes' 16 Obstacles to a Successful Software Project*, which emphasizes challenges such as hyper-focused planning, unclear expectations, and poor collaboration. These issues often lead to scenarios where software is more about managing complexity than solving problems.

This disconnect can be likened to going to a restaurant, ordering a pizza, waiting for hours, and then receiving a Caesar salad with an expensive bill. You pay for something you didn't want and didn't need. Unfortunately, this is still the reality in many software projects. DDD aims to change that. However, before diving into how it helps, it's essential to clarify some common misconceptions and explain what DDD is not.

DDD is not a framework, programming language, or specific paradigm. It's a methodology that can be applied across languages and frameworks. DDD is not inherently tied to microservices but can help define bounded contexts. Most importantly, DDD is not solely about code—it's about fostering a positive communication process between developers and domain experts to translate business knowledge accurately into software.

The essence of DDD lies in its ability to extract and encapsulate domain knowledge. It focuses on transferring this knowledge, often held by domain experts, into the software design process. In practice, DDD is divided into two primary parts: strategic and tactical. While tactical patterns often steal the spotlight, strategic DDD is the foundation that ensures those patterns make sense in the larger context.

Strategic Domain-Driven Design starts with determining the Domain and its Subdomains. This step also includes creating a common vocabulary, which removes any vagueness by embedding the meaning of words in their context. For example, "Ajax" can mean a soccer team, a cleaning agent, or front-end technology, depending on the context. This enables teams to mutually understand four aspects of bridging business and IT.

After the strategies are formulated, tactical DDD starts. It involves incorporating the patterns known as entities, value objects, repositories, services, aggregates and factories. Each of these patterns has an intended use in the design:

- **Entity:** An object with a distinct identity retained through time. For example, a Vehicle (Car) with a unique VIN Number.
- **Value Object:** A collection of attributes considered to have no existence, such as the name of the Manufacturer and the model number, which are used as the basic building blocks and never change.
- **Repository:** A mechanism for persistent store objects and a type of service where business logic is defined, enabling us to interact with and manipulate entities.
- **Service:** Stateless operations do not belong to or are associated with a particular entity or value object.
- **Aggregate:** A collection of related entities and value objects treated as a single unit regarding data changes within a specified limit.
- **Factory:** Complex obj creation and management.

Let's explore a simple example using Jakarta Data and Eclipse JNoSQL to see how these concepts translate into practice. Imagine we are building a car management system. Our first step is to define the core components: the Car entity and the Manufacturer value object.

```
@Entity
public class Car {
    @Id
    private String vin;
    @Column
    private String transmission;
    @Column
    private Manufacturer manufacturer;
    @Column
    private String color;
}
@Embeddable(Embeddable.EmbeddableType.GROUPING)
public record Manufacturer(@Column String name, @Column String model)
 {
}
```

The car represents an Entity with VIN. We also have the Manufacturer as a value object, which is immutable, represented by a record, and contains the car's maker details.

Next, we define a repository interface using Jakarta Data. This repository captures the domain language, making it intuitive and closely aligned with business terminology:

```
@Repository
public interface Garage {
    @Save
    Car park(Car car);
    @Delete
    void unpark(Car car);
    @Find
    Car checkRegistration(@By(„vin“) String vin);
}
```

The availability of methods like park, unpark, and checkRegistration, which may easily be understood in layman's English as about that particular domain, makes the code sensible to programmers and the business community.

When developed to its fullest potential, Domain-Driven Design aids gains and objectives throughout as it helps teams deliver software that can respond effectively and efficiently to the interests of all those who sponsor the development of the software. It facilitates teamwork, reduces the chances of conflicts, and ensures that all activities, including planning and implementation, are centered on the business. DDD and frameworks such as Jakarta Data are instrumental in producing functional software and can address the correct issues. To explore and see the source code, go to GitHub: soujava/ddd-zero-hero.

> Back to Table of Content

#JAVAPRO #ARCHITECTURE #MICROSERVICES

How Coupled Are Your Microservices?

Author:

Wanderson Xesquevixos is a passionate software engineer with over 20 years of experience and author of Software Architecture with Spring (Packt, 2025). Working with Java since 2001 and Spring since 2006, he holds Java and AWS certifications. He specializes in software architecture, microservices, and cloud-native systems, helping teams build scalable solutions while applying a philosophical lens to technical decision-making, constantly questioning the "why" behind each choice and its trade-offs. Wanderson holds a degree in Computer Science and MBAs from USP in Data Science and Software Engineering. He is currently preparing to pursue a master's in Computer Science focused on Distributed Systems and Cloud Computing

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

In this article, we address a crucial topic in software architecture: coupling and its main types in the context of microservices architecture.

The motivation for this writing comes from direct experience with the negative impact of coupling on system design. In one project, delivered by a software house, we had to fully refactor it before going live due to excessive domain, pass-through, shared data, and content coupling. These interdependencies made the services fragile and undermined their autonomy, scalability, and maintainability.

Understanding the different forms of coupling and their implications is essential for designing robust and scalable microservice systems.

Before we explore the types of coupling, let's understand how components communicate in different architectural styles.

Component Communication: In-Process vs. Inter-Process

In monolithic systems, modules interact within the same process, sharing memory, objects, and direct calls. This in-process communication is fast, reliable, and flexible. Although coupling exists between modules, developers can manage them more easily since everything resides in the same space and they can coordinate changes directly.

In contrast, microservices-based architectures run each service in separate processes, often in distributed environments. Inter-process communication occurs over the network using REST APIs, asynchronous events, and gRPC, among others. This introduces challenges such as latency, network failures, versioning, and observability, requiring well-defined contracts and service decoupling.

Thus, while coupling in monolithic systems is more tolerable, it can directly compromise system autonomy, scalability, and resilience in microservices.

How Communication Affects Coupling In Microservices

The way services or modules communicate determines the type and severity of coupling. In monoliths, content coupling or shared data coupling is common (we'll discuss these later), as modules can access each other's internal structures directly.

In microservices, the ideal is to maintain loose domain coupling, where services depend only on public, stable contracts, such as APIs or events, and never on internal logic or data structures. This separation preserves service autonomy and prevents cascading side effects.

Now that we understand the different communication contexts and their impact on coupling, let's examine the main types of coupling.

Types of Coupling

Coupling refers to the degree of dependency between two modules or services and how much one must know or rely on the other to function properly. The lower the coupling, the greater the service autonomy and flexibility.

Not all coupling is inherently bad; eliminating it completely is unfeasible in real-world systems. The goal is to reduce unnecessary or harmful forms of coupling as much as possible.

Figure 1 presents the four main types of coupling that can occur between microservices, ordered from the most desirable (low coupling) to the least desirable (high coupling).

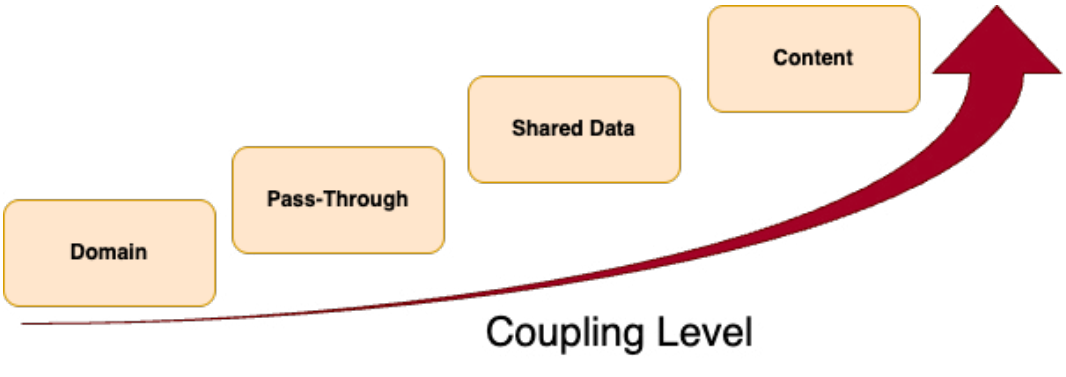


Figure 1: Types of coupling in order of increasing dependency

The most common types of coupling in microservices are domain coupling, pass-through coupling, shared data coupling, and content coupling, which are ordered in increasing dependency. They are organized from the most desirable (low coupling) to the least desirable (high coupling).

Designing effective microservices requires careful attention to these forms of dependency and applying best practices to mitigate them, such

as using asynchronous events, well-defined APIs, and clear separation of responsibilities. Next, we analyze each of these types, starting with domain coupling.

Domain Coupling

Domain coupling occurs when one microservice directly depends on another to perform a business function. It is a common and acceptable type of dependency in microservices since each service handles a specific business domain. Communication between services is natural to fulfill broader system requirements. For example, in a digital library system, a Loan service might need to query a User service to obtain user details such as name, email, address, and the Catalog service to get book details. ***Figure 2*** illustrates domain coupling between the loans, users, and catalog services.

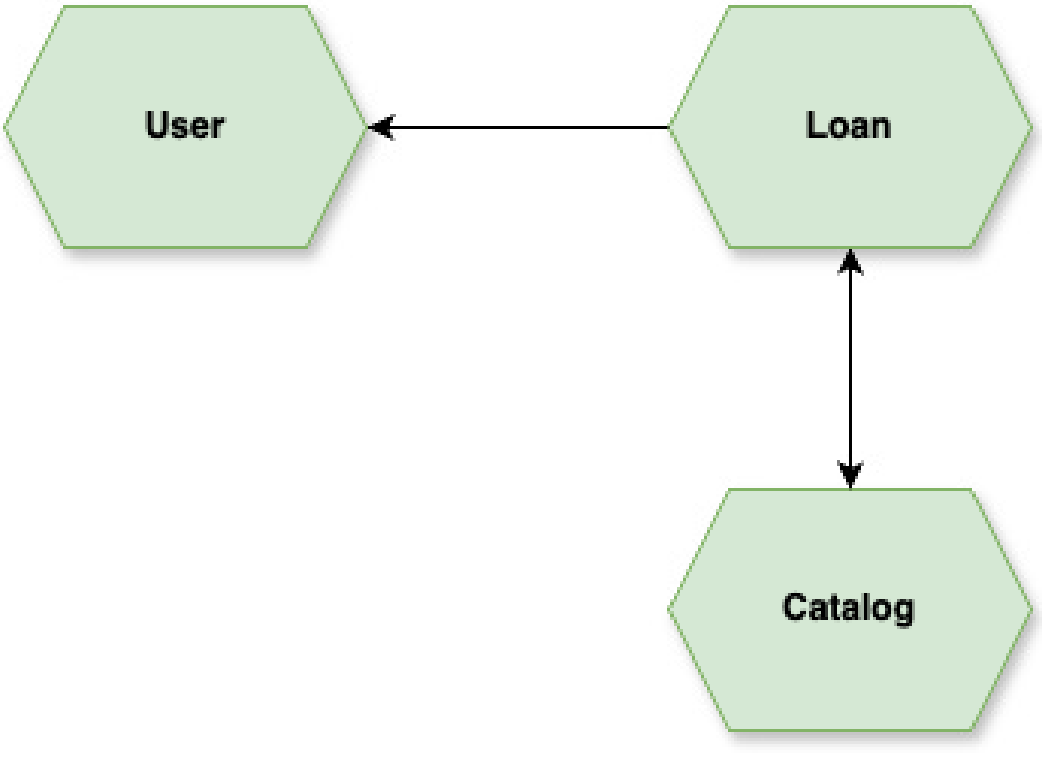


Figure 2: Domain coupling

The Loan service is domain-coupled with both the User and Catalog services. The Catalog service is also domain-coupled with the Loan service, while the User service remains independent and has no domain coupling to other services.

It is worth noting that domain coupling becomes particularly problematic when the boundaries of responsibility between services are not clearly

defined. In many projects, the lack of explicit domain modeling leads to situations where one service depends on rules or decisions belonging to another or, worse, replicates them in parallel. This results in data inconsistencies, duplicated logic, semantic coupling, and difficulty evolving the system.

Without clear boundaries, what would be a natural coupling, such as querying data from a neighboring domain, becomes a fragile and dysfunctional link that requires coordinated deployments, increases the risk of regressions, and undermines service autonomy. In the long run, the system loses the benefits of modularity and tends to become a distributed monolith, where everything depends on everything else.

Therefore, defining business domains, as Domain-Driven Design (DDD) proposed, is fundamental to ensure that domain coupling remains healthy and controlled.

Pass-Through Coupling

Pass-through Coupling occurs when a service passes data to another, not because it needs the data itself, but because a third service will use it. ***Figure 3*** illustrates a pass-through coupling scenario where the delivery of a book needs to notify other users about its availability via email.

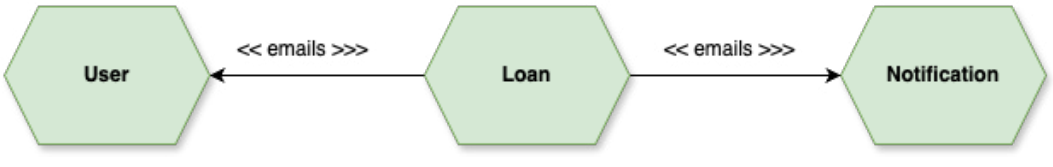


Figure 3: Pass-through coupling

The loan service queries the User service to retrieve user emails and passes them to the Notification service. The Loan service merely forwards the user data to the notification service without using it directly, creating a pass-through coupling.

Drawbacks of Pass-Through Coupling

Pass-through coupling may initially seem harmless, but it introduces several architectural issues that undermine distributed systems' cohesion, maintainability, and evolution. The harms of pass-through coupling are:

- **Increase in unnecessary coupling:** The intermediary service, in this case, the Loan service, does not need the data for itself, yet it becomes dependent on the format, semantics, and contract of data that belongs to the Notification service. As a result, changes in the source (User service) or the destination (Notification service) can break the intermediary service (Loan service).
- **Violation of the single responsibility principle:** The intermediary service starts handling responsibilities outside its domain, merely to satisfy the needs of another service. This leads to a loss of cohesion, as the service becomes too aware of foreign domains.
- **Unnecessary exposure of internal details:** Internal data from one service, such as email structure or addresses, is leaked to another service, even if that service doesn't use it directly. This increases the risk of abstraction leakage and breaks encapsulation.
- **Maintenance fragility:** Changes to how data is represented or used now affect three services instead of two: the original producer (User service), the final consumer (Notification service), and the intermediary (loan service). This increases maintenance costs and the risk of regressions.
- **Unnecessary complexity in the data flow:** The data path becomes longer and harder to trace. Understanding where the data came from and why it is there becomes more challenging, complicating debugging, observability, and auditing.
- **Testing and isolation become harder:** Since the intermediary service now depends on the data and contracts of two other services, its tests must mock or simulate more external behaviors, making them more fragile and time-consuming.

Pass-through coupling violates the principles of modularity and service autonomy. It causes a service to become responsible for data and contracts that do not belong to its domain, making the system more rigid, fragile, and difficult to evolve.

How Can We Avoid Pass-Through Coupling?

A viable alternative would be for the Loan service to forward only the IDs of the users waiting for the book delivery to the Notification service. The Notification service would then be responsible for querying the User service to retrieve the corresponding email addresses. ***Figure 4*** illustrates this alternative.

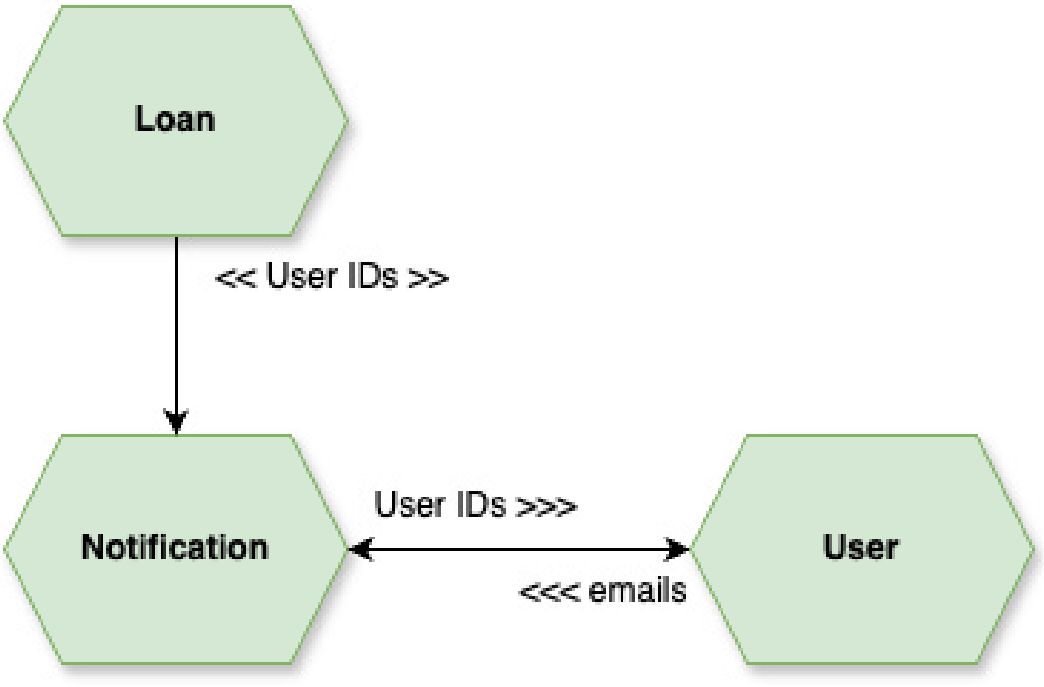


Figure 4: Alternative to avoid pass-through coupling

To avoid temporal coupling, a topic that will be addressed later, we can adopt the same logic of interaction between services, but use asynchronous communication via a message broker instead of direct calls. ***Figure 5*** illustrates this approach.

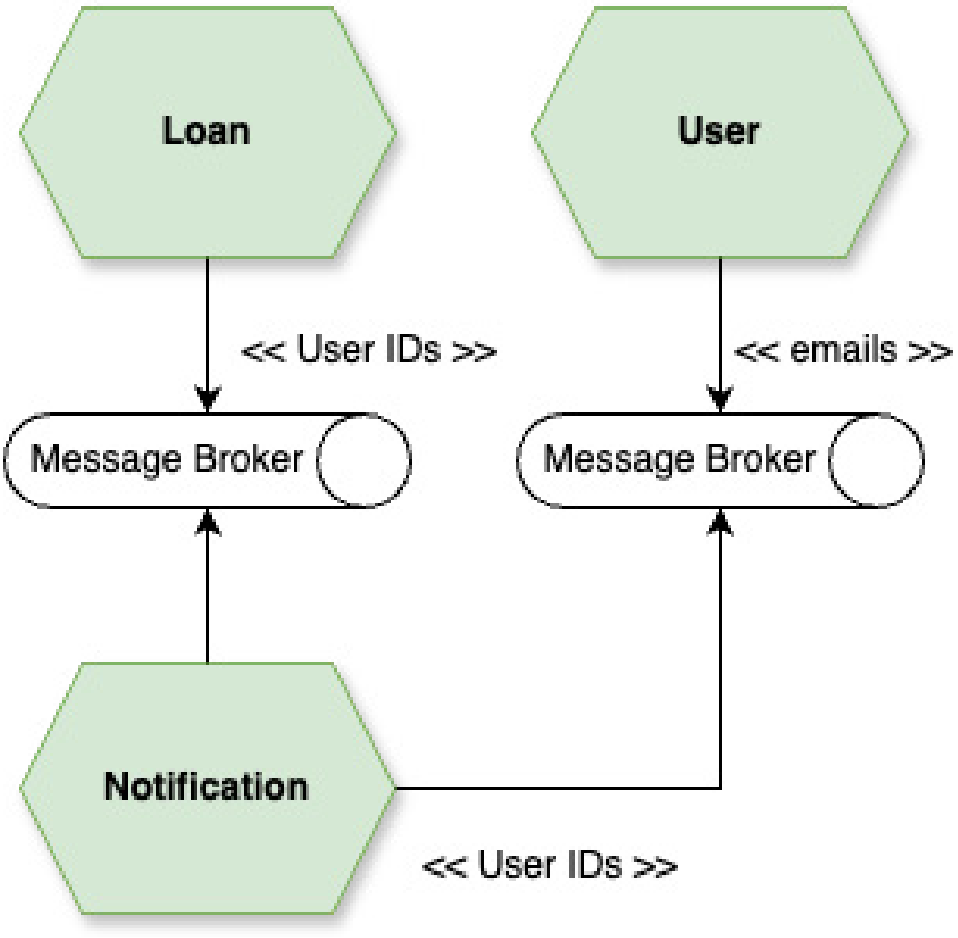


Figure 5: Alternative to avoid pass-through coupling using message brokers

As we have seen, pass-through coupling introduces unnecessary dependencies by making a service intermediate data that does not fall under its responsibility. Now, let's examine shared data coupling.

Shared Data Coupling

At an even more concerning level, shared data coupling occurs when two or more services directly share the same data source, compromising the independence and isolation of functionalities. ***Figure 6*** illustrates a shared data coupling.

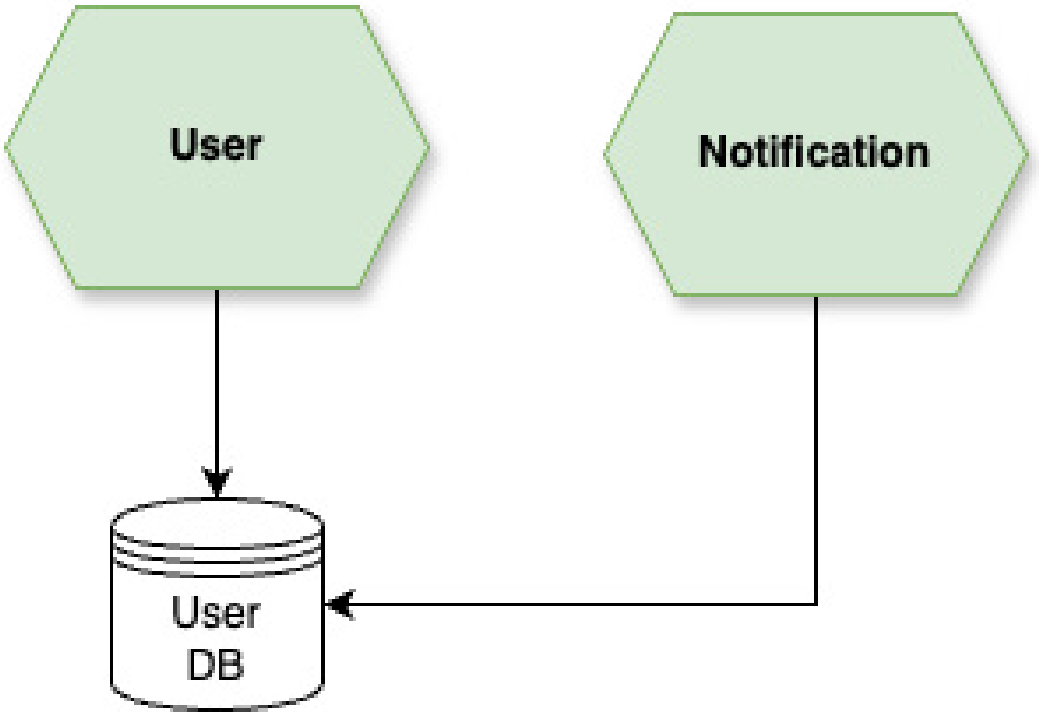


Figure 6: Shared data coupling

Using the previous scenario where users on the waiting list must be notified of a book delivery, shared data coupling would arise if the Notification service accessed the User service's database directly to retrvice's database directly to retrieve email addresses.

Drawbacks of Shared Data Coupling

The main problems associated with shared data coupling are:

- **Loss of service autonomy:** When two or more services access the same data source, be it a database, memory, or shared file system, they are no longer truly autonomous. As a result, any change to the data schema can impact multiple services simultaneously.
- **Fragility in structural changes:** Since services share the physical data structure, changes such as renaming columns, adding indexes, or modifying constraints require team coordination. This makes independent and safe service evolution more difficult.
- **Versioning difficulties:** It becomes nearly impossible to version data independently per service, as all services are tied to the same model and schema. This limits practices such as controlled migrations or safe rollbacks.

- **Concurrency and integrity conflicts:** Different services may attempt to update the same data simultaneously without orchestration or control logic. This can result in inconsistencies, data loss, or transactional corruption.
- **Increased testing and deployment complexity:** Integration testing and deployments become more difficult, as a change in one service may require regression testing in all other services that share the data. This leads to tighter organizational coupling, delays, and increased risk in production environments.
- **Unintentional exposure of internal structures:** Sharing a data source often reveals implementation details such as relationships and normalization. This can result in misuse, duplicated logic, and inconsistent interpretations of the data across services.

When Shared Data Coupling May Be Accepted

Shared data coupling is one of the most problematic forms of coupling, as it violates the principle of service autonomy. Despite the risks, shared data coupling can be tolerated in specific scenarios, such as reading static data, such as lists of usernames, countries, or even cities, provided this information is relatively stable and rarely changes. However, services should never directly access another service's database, as when the Notification service accesses the User service's database.

In situations like the country list example, a more appropriate approach would be to expose this data through a shared cache, such as Redis, thus preserving the independence between services.

Next, we will examine content coupling, which is considered the most critical among all coupling types.

Content Coupling

This is the most undesirable type of coupling. It occurs when an external service directly accesses the internal data of another service, modifying its internal state while completely bypassing the API or business logic responsible for protecting that access. ***Figure 7*** illustrates this scenario.

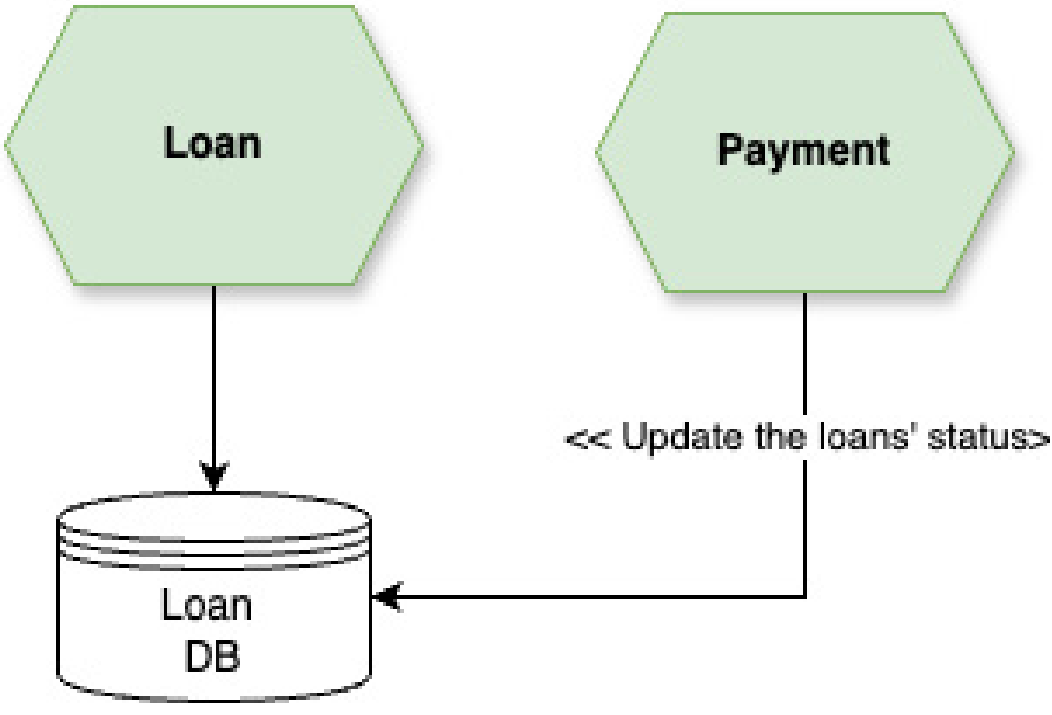


Figure 7: Content coupling

Using our digital library case study, this would be equivalent to the Payment service directly accessing the loans table to update the status field to "Regularized" after a fine is paid without calling the Loan service's API, which should be responsible for validating and applying this change.

Drawbacks of Content Coupling

The main consequences of content coupling are:

- **Violation of encapsulation:** The consuming service bypasses the business logic of the service that owns the data, breaking domain boundaries and allowing changes without validation or control.
- **High risk of regressions:** Any change in the internal structure, such as field names, data types, or business rules, can silently break the consuming service. Local tests may fail to catch these issues.
- **Inability to evolve safely:** The service that owns the data becomes constrained, as it can no longer refactor its internal structure without directly affecting other services. This creates hidden bidirectional coupling.
- **Difficulty in tracing and auditing:** Since changes occur outside the official API, logs, events, and traces become inconsistent. Another service may modify the data without the service owner even realizing it.

Even in emergencies we must avoid the content coupling. Communication between services should always occur through well-defined public interfaces that encapsulate business rules and ensure data consistency. Otherwise, the system becomes fragile, difficult to maintain, and prone to errors that are hard to diagnose.

Temporal Coupling

Temporal coupling occurs when a service depends on another being available at the exact moment of interaction for a functionality to complete successfully. Unlike logical coupling types such as domain, pass-through, content, or shared data coupling, temporal coupling is related to runtime availability.

An example would be the Loan service making a synchronous HTTP call to the User service to verify if the user is active before completing a book checkout. The loan process is blocked or fails if the User service is unavailable or experiencing high latency. Figure 8 illustrates a synchronous call to the User service from the Loan service.

Temporal coupling is not always harmful, but it is essential to recognize its presence. In scenarios where a sequence of microservices communicates synchronously, the challenges of this type of coupling tend to intensify, potentially impacting the system's scalability and resilience. ***Figure 8*** presents a blocking, synchronized network call.

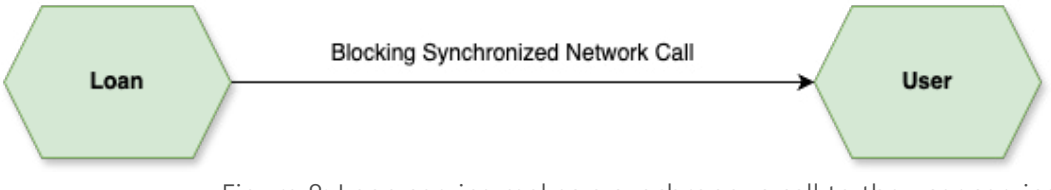


Figure 8: Loan service makes a synchronous call to the user service

One way to reduce temporal coupling is to adopt asynchronous, event-based communication using a broker such as Kafka or RabbitMQ. In addition, it is essential to apply fault-tolerance best practices to mitigate the effects of this type of coupling, such as circuit breakers, retries, timeouts, and degraded responses. The use of local or distributed caching, as well as the separation of critical and non-critical flows, also helps prevent non-essential functionalities from blocking the main operation.

Conclusion

Designing microservices requires awareness of the different types of coupling, how they arise, and how to mitigate them. Understanding coupling types such as domain, pass-through, shared data, content, and temporal is essential to ensure autonomy, scalability, and maintainability in distributed architectures.

While certain coupling levels, such as domain coupling, are inevitable, the goal should always be to minimize unnecessary coupling by favoring well-defined contracts, asynchronous event-driven communication, fault tolerance, and proper encapsulation of each service's responsibilities.

By correctly identifying the type of coupling involved in an interaction and applying the appropriate strategies to reduce it, we can build more resilient, evolvable systems aligned with the principles of service-oriented architecture. Ultimately, good microservice design depends less on the complete absence of coupling and more on its correct identification, management, and isolation.

References

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O'Reilly Media.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley.

> Back to Table of Content

#JAVAPRO #CLOUD

How to Develop, Run and Optimize Quarkus Web Application on AWS Lambda

Author:

Vadym Kazulkin is AWS Serverless Hero and Head of Development at ip.labs GmbH, a 100% subsidiary of the FUJIFILM Group, based in Bonn. ip.labs is the world's leading white label e-commerce software imaging company. Vadym has been involved with the Java ecosystem for over twenty years. His focus and interests currently include the design and implementation of highly scalable and available applications in AWS Cloud with the special passion for Serverless. Vadym is also the co-organizer of the Java User Group Bonn meetup and a frequent speaker at various Meetups and conferences.

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

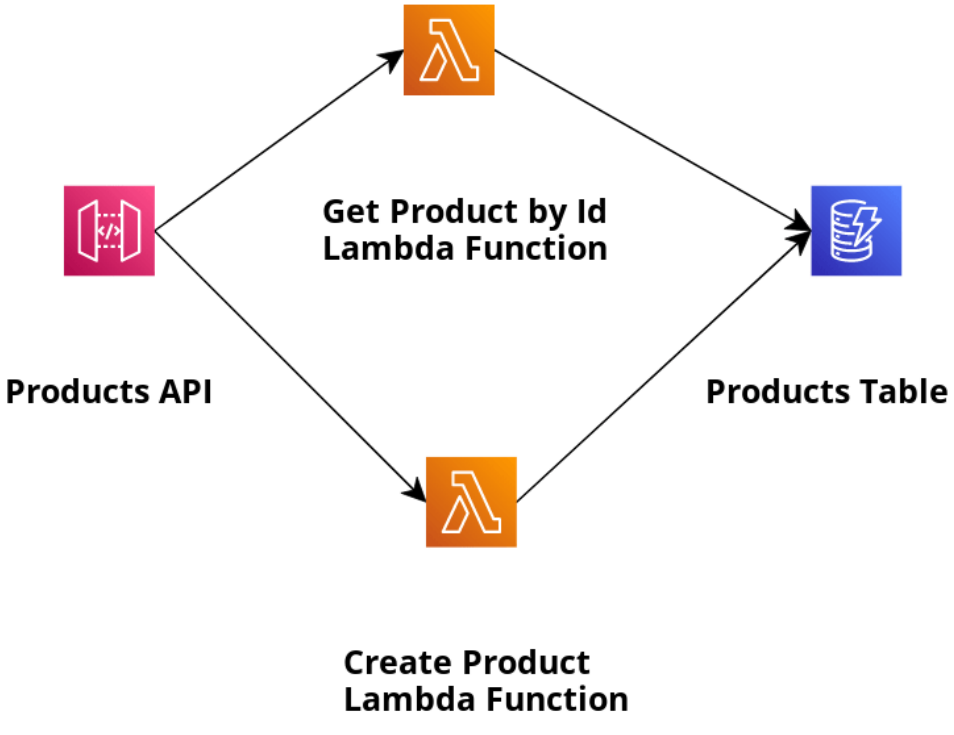
What Will We Explore and Learn in This Article?

In this article, we will explore some ways to develop, deploy and run applications on AWS Lambda using Quarkus framework. Of course, we will measure performance (the cold and warm start times) of the Lambda function. We will also show how we can optimise performance

of Lambda functions using Lambda SnapStart (including various priming techniques) and GraalVM Native Image deployed as AWS Lambda Custom Runtime. You can find code examples for the whole series in my GitHub Account.

Example Application With the Quarkus Framework on AWS Lambda

To explain this, we will use a simple example application, the architecture of which is shown below.



In this application, we will create products and retrieve them by their ID and use Amazon DynamoDB as a NoSQL database for the persistence layer. We use Amazon API Gateway which makes it easy for developers to create, publish, maintain, monitor and secure APIs and AWS Lambda to execute code without the need to provision or manage servers. We also use AWS SAM, which provides a short syntax optimised for defining infrastructure as code (hereafter IaC) for serverless applications. For this article, I assume a basic understanding of the mentioned AWS services, serverless architectures in AWS, Quarkus framework and GraalVM including its Native Image capabilities.

In order to build and deploy the sample application, we need the following local installations: Java 21, Maven, AWS CLI and SAM CLI. For

the GraalVM example, we also need GraalVM and Native Image. For the GraalVM example, additionally GraalVM and Native Image.

Now let's look at relevant source code fragments and start with the sample application that we will run directly on the managed Java 21 runtime of AWS Lambda. AWS Lambda only supports managed Java LTS versions, so version 21 is currently the latest.

First, let's take a look at the source code of the GetProductByIdHandler Lambda function. This Lambda function determines the product based on its ID and returns it.

```
@Named(„getProductById“)
public class GetProductByIdHandler implements RequestHandler {

@Inject
private ObjectMapper objectMapper;

@Inject
private DynamoProductDao productDao;

@Override
public APIGatewayProxyResponseEvent
handleRequest(APIGatewayProxyRequestEvent requestEvent, Context
context) {
    String id = requestEvent.getPathParameters().get(„id“);
    Optional<Product> optionalProduct = productDao.getProduct(id);

    try {
        if (optionalProduct.isEmpty()) {
            context.getLogger().log(„ product with id „ + id + „ not
            found „);
           return new APIGatewayProxyResponseEvent().
           withStatusCode(HttpStatusCode.NOT_FOUND)
                  .withBody(„Product with id = „ + id + „ not
                  found“);
        }
        context.getLogger().log(„ product „ + optionalProduct.get() +
        „ found „);
```

```
            return new APIGatewayProxyResponseEvent().
            withStatusCode(HttpStatusCode.OK)
                    .withBody(objectMapper.
                     writeValueAsString(optionalProduct.get()));
        } catch (Exception je) {
            je.printStackTrace();
            return new APIGatewayProxyResponseEvent().
              withStatusCode(HttpStatusCode.INTERNAL_SERVER_ERROR)
                     .withBody(„Internal Server Error :: „ +
                     je.getMessage());
        }
      }

}
```

We annotate the Lambda function with **@Named(„getProductById“)**, which will be important for mapping, and inject the implementation of DynamoProductDao. The method ***handleRequest*** receives an object of type APIGatewayProxyRequestEvent as input, as APIGatewayRequest invokes the Lambda function, from which we retrieve the product ID by invoking **requestEvent.getPathParameters().get(„id“)** and ask our DynamoProductDao to find the product with this ID in the DynamoDB by calling **productDao.getProduct(id)**. Depending on whether the product exists or not, we wrap the Jackson serialised response in an object of type APIGatewayProxyResponseEvent and send it back to Amazon API Gateway as a response. The source code of the Lambda function CreateProductHandler, which we use to create and persist products, looks similar.

The source code of the Product entity looks very simple:

```
public record Product(String id, String name, BigDecimal price) {}
```

The implementation of the DynamoProductDao persistence layer uses AWS SDK for Java 2.0 to write to or read from the DynamoDB. Here is an example of the source code of the **getProductById** method, which we used in the GetProductByIdHandler Lambda function described above:

```
  public Optional<Product> getProduct(String id) {
    GetItemResponse getItemResponse= dynamoDbClient.
    getItem(GetItemRequest.builder()
      .key(Map.of(„PK“, AttributeValue.builder().s(id).build()))
      .tableName(PRODUCT_TABLE_NAME)
      .build());
    if (getItemResponse.hasItem()) {
      return Optional.of(ProductMapper.
      productFromDynamoDB(getItemResponse.item()));
    } else {
      return Optional.empty();
    }
  }
```

Here we use the instance of DynamoDbClient Client to build GetItemRequest to query DynamoDB table, whose name we get from environment variable (which we will set in AWS SAM template) by invoking **System.getenv(„PRODUCT_TABLE_NAME“)**, for the product based on its ID. If the product is found, we use the custom written ProductMapper to map the DynamoDB item to the attributes of the product entity.

Apart from annotation, we have not yet seen any dependencies on the Quarkus framework. We can see how everything interacts in pom.xml. Apart from dependencies to the Quarkus framework (we are using version 3.18.3, but you are welcome to upgrade to the newer version and most of it should work the same), AWS SDK for Java and other AWS artefacts, we see the following dependency,

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-amazon-lambda</artifactId>
</dependency>
```

which is a bridge between AWS Lambda and Quarkus Framework. Now let's look at the last missing part, namely IaC with AWS SAM, which is defined in template.yaml. There we declare Amazon API Gateway (incl. UsagePlan and API Key), AWS Lambdas and DynamoDB table. We first look at the definition of the lambda function GetProductByIdFunction:

```
GetProductByIdFunction:
   Type: AWS::Serverless::Function
   Properties:
     FunctionName: GetProductByIdWithWithQuarkus318
     AutoPublishAlias: liveVersion
     Policies:
       - DynamoDBReadPolicy:
           TableName: !Ref ProductsTable
     Environment:
       Variables:
         QUARKUS_LAMBDA_HANDLER: getProductById
     Events:
       GetRequestById:
         Type: Api
         Properties:
           RestApiId: !Ref MyApi
           Path: /products/{id}
           Method: get
```

We see that this Lambda function is linked to the HTTP Get call method and the path /products/{id} of the API gateway. But how is GetProductByIdHandler Lambda implementation resolved? We see the environment variable QUARKUS_LAMBDA_HANDLER, whose value getProductById matches the value of the named annotation (@Named(„getProductById")) on the GetProductByIdHandler class. The resolution itself is performed by the generic io.quarkus.amazon.lambda.runtime.QuarkusStreamHandler Lambda handler, which is defined in template.yaml in the Globals section of the Lambda functions as follows:

```
Globals:
  Function:
    Handler: io.quarkus.amazon.lambda.runtime.
    QuarkusStreamHandler::handleRequest
    CodeUri: target/function.zip
    Runtime: java21
    ....
    Environment:
      Variables:
```

```
        ...
        PRODUCT_TABLE_NAME: !Ref ProductsTable
    ....
```

Other parameters are also defined there that are valid for all defined Lambda functions, such as Java runtime environment Java 21 and CodeURI. We have also set DynamoDB table name as environment variable, which is used in the DynamoProductDao class.

Now we have to build the application with **mvn clean package** (function.zip is created and stored in the subdirectory named target) and deploy it with **sam deploy -g**. We will see our customised Amazon API Gateway URL in the return. We can use it to create products and retrieve them by ID. The interface is secured with the API key. We have to send the following as HTTP header: „X-API-Key: a6ZbcDefQW12BN56WEV318", see MyApiKey definition in template.yaml. To create the product with ID=1, we can use the following curl query:

```
curl -m PUT -d ,{ „id“: 1, „name“: „Print 10x13“, „price“: 0.15 }‘
-H „X-API-Key: a6ZbcDefQW12BN56WEV318“ https://{$API_GATEWAY_URL}/
prod/products
```

For example, to query the existing product with ID=1, we can use the following curl query:

```
curl -H „X-API-Key: a6ZbcDefQW12BN56WEV318“ https://{$API_GATEWAY_
URL}/prod/products/1
```

In both cases, we need to replace the {$API_GATEWAY_URL} with the individual Amazon API Gateway URL that is returned by the sam deploy -g command. We can also search for this URL when navigating to our API in the Amazon API Gateway service in the AWS console.

Measurements of Cold and Warm Start Times of Our Application

In the following, we will measure the performance of our GetProductByIdFunction Lambda function, which we will trigger by invoking *curl -H 'X-API-Key: a6ZbcDefQW12BN56WEV318' https://*

{$API_GATEWAY_URL}/prod/products/1. Two aspects are important to us in terms of performance: cold and warm start times. It is known that Java applications in particular have a very high cold start time. The article Understanding the Lambda execution environment lifecycle provides a good overview of this topic. We will also present various performance optimization options so that we can make 5 different measurements:

1. Sample application as we have presented it above
2. Sample application with the activated AWS Lambda SnapStart without using priming
3. Sample application with the activated AWS Lambda SnapStart with application of priming from DynamoDB request
4. Sample application with the activated AWS Lambda SnapStart with priming of the API Gateway request event
5. A rebuilt sample application that we build GraalVM Native Image and deploy it as Lambda Custom Runtime

Now we will go through all 5 performance measurements methods and only at the end will we show how we carried out performance measurements and summarise the results for all 5 methods.

1. Sample Application as Presented Above

To perform the performance measurement, we must ensure that AWS Lambda SnapStart, which we will introduce in a moment, is deactivated with *#*. This is done in template.yaml as follows:

```
Globals:
  Function:
    Handler: io.quarkus.amazon.lambda.runtime.
    QuarkusStreamHandler::handleRequest
    CodeUri: target/function.zip
    Runtime: java21
    #SnapStart:
     #ApplyOn: PublishedVersions
...
```

It is therefore necessary that we perform the 4 measurements on one and the same application and therefore we have to switch some things on and off for each measurement. It is conceivable to solve it more elegantly and e.g. define several AWS SAM templates, with and without SnapStart.

2. Sample Application With the Activated AWS Lambda SnapStart Without Using Priming

As we will see, without any optimizations, the performance measurements for method 1 will show quite high values, especially for the cold start times. We will therefore present various optimization options for methods 2-5. Lambda SnapStart is one of them.

Lambda SnapStart can provide a start time of a lambda function of less than one second. SnapStart simplifies the development of responsive and scalable applications without provisioning resources or implementing complex performance optimizations.

The largest portion of startup latency (often referred to as cold start time) is the time Lambda spends initializing the function, which includes loading the function code, starting the runtime and initialising the function code. With SnapStart, Lambda initializes our function when we publish a function version. Lambda takes a Firecracker microVM snapshot of the memory and disk state of the initialised execution environment, encrypts the snapshot and intelligently caches it to optimize retrieval latency.

To ensure reliability, Lambda manages multiple copies of each snapshot. Lambda automatically patches snapshots and their copies with the latest runtime and security updates. When we call the function version for the first time and as the calls increase, Lambda continues new execution environments from the cached snapshot instead of initialising them from scratch, which improves startup latency. More information can be found in the article Reducing Java cold starts on AWS Lambda functions with SnapStart. I have published the whole series about Lambda SnapStart for Java applications online.

To activate Lambda SnapStart, we have to do the following in template.yaml for the Lambda function:

```
Globals:
  Function:
    Handler: io.quarkus.amazon.lambda.runtime.
    QuarkusStreamHandler::handleRequest
    CodeUri: target/function.zip
    Runtime: java21
    SnapStart:
     ApplyOn: PublishedVersions
...
```

This can be done in the globals section of the Lambda functions, in which case SnapStart applies to all Lambda functions defined in the AWS SAM template, or you can add the 2 lines

```
SnapStart:
 ApplyOn: PublishedVersions
```

to activate SnapStart only for the individual Lambda function. To perform the performance measurement without priming techniques, as we will introduce in methods 3 and 4, please either comment out or remove @Startup annotation in the following 2 Java classes AmazonDynamoDBPrimingResource and AmazonAPIGatewayPrimingResource .

3. Sample Application With the Activated AWS Lambda SnapStart With Application of Priming For DynamoDB Request

We have already learnt about Lambda SnapStart in method 2. Activating Lambda SnapStart is also a prerequisite for this method. As we will see later, cold start times are significantly reduced simply by activating SnapStart without changing our source code. However, there are other techniques to reduce this, which we call priming and which require changes to the source code.

SnapStart and runtime hooks offer you new possibilities to create your Lambda functions for low startup latency. With the pre snapshot hook,

we can prepare our Java application as much as possible for the first call. We load and initialize as much as possible which our Lambda function needs before the snapshot is created. This technique is known as priming.

In this method I will introduce you to the priming of DynamoDB request, which is implemented in the AmazonDynamoDBPrimingResource class.

```
@Startup
@ApplicationScoped
public class AmazonDynamoDBPrimingResource implements Resource {

    @Inject
    private ObjectMapper objectMapper;

    @Inject
    private DynamoProductDao productDao;

    @PostConstruct
    public void init () {
          Core.getGlobalContext().register(this);
    }

    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource>
    context) throws Exception {
           productDao.getProduct(„0“);
    }

    @Override
    public void afterRestore(org.crac.Context<? extends Resource>
    context) throws Exception {
    }

}
```

We use CRaC runtime hooks here. To do this, we need to declare the following dependency in pom.xml:

```
<dependency>
    <groupId>io.github.crac</groupId>
    <artifactId>org-crac</artifactId>
</dependency>
```

AmazonDynamoDBPrimingResource class is annotated with @Startup annotation (so that this class/bean is initialized directly when the application is started) and implements **org.crac.Resource** interface. The class registers itself as a CRaC resource in the ***init*** method annotated with *@PostConstruct* annotation. The priming itself happens in the method where we ask DynamoDB for the product with the ID equal to 0. *beforeCheckpoint* method is CRaC runtime hook that is invoked before creating the microVM snapshot. We are not even interested in the result of the call **productDao.getProduct(„0“)**, but with this call all required classes are instantiated and the expensive one-time initialisation of HTTP Client (default is Apache HTTP Client) and Jackson Marshallers (for the purpose of converting Java objects to JSON and vice versa) is carried out. As this is done during the deployment phase of the Lambda function when SnapStart is activated and before the snapshot is created, the snapshot will already contain all of this. After the fast snapshot restore phase during the Lambda invoke, we will gain a lot in performance in case of cold start by priming this way (see measurements below). We therefore prime the DynamoDB request.

To ensure that only this priming takes effect, please either comment out or remove the @Startup annotation in the following AmazonAPIGatewayPrimingResou class.

4. Sample Application With the Activated AWS Lambda SnapStart With Application of Priming of API Gateway Request Event

Here I'll present you another experimental priming technique that preinitializes the entire web request (API gateway request event). This preinitializes more than method 3, but also requires significantly more code to be written. The idea is nevertheless comparable. Activating Lambda SnapStart is also a prerequisite for this method. Let's take a look at the implementation in the AmazonAPIGatewayPrimingResource class:

```
@Startup
@ApplicationScoped
public class AmazonAPIGatewayPrimingResource implements Resource {

    @PostConstruct
    public void init () {
           Core.getGlobalContext().register(this);
    }

    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource>
    context) throws Exception {
           new QuarkusStreamHandler().handleRequest
            (new
             ByteArrayInputStream(convertAwsProxRequestToJsonBytes()),
                         new ByteArrayOutputStream(), new
                         MockLambdaContext());
    }

    @Override
    public void afterRestore(org.crac.Context<? extends Resource>
    context) throws Exception {
    }

    private static byte[] convertAwsProxRequestToJsonBytes () throws
    JsonProcessingException {
           ObjectWriter ow = new ObjectMapper().writer().
           withDefaultPrettyPrinter();
           return ow.writeValueAsBytes(getAwsProxyRequest());
    }

    private static AwsProxyRequest getAwsProxyRequest () {
           final AwsProxyRequest awsProxyRequest = new AwsProxyRequest
           ();
           awsProxyRequest.setHttpMethod(„GET“);
           awsProxyRequest.setPath(„/products/0“);
           awsProxyRequest.setResource(„/products/{id}“);
           awsProxyRequest.setPathParameters(Map.of(„id“,“0“));
           final AwsProxyRequestContext awsProxyRequestContext = new
```

```
            AwsProxyRequestContext();
            final ApiGatewayRequestIdentity apiGatewayRequestIdentity=
            new ApiGatewayRequestIdentity();
            apiGatewayRequestIdentity.setApiKey(„blabla“);
            awsProxyRequestContext.
            setIdentity(apiGatewayRequestIdentity);
            awsProxyRequest.setRequestContext(awsProxyRequestContext);
            return awsProxyRequest;
    }
}
```

Please make sure that @Startup annotation is present in the AmazonAPIGatewayPrimingResource class so that the priming takes effect. As we can see, in the method getAwsProxyRequest we create object of type AwsProxyRequest, which is sent to the path /products/{id} and sets ID =0. In the CRaC runtime hook ***beforeCheckpoint*** method, AwsProxyRequest is converted into a byte array and processed by calling **QuarkusStreamHandler().handleRequest**. This basically mocks APIGatewayProxyRequestEvent and the call is mapped to the Lambda function GetProductByIdHandler, which handleRequest method is called directly. The priming is performed locally in AWS, so no network trip is required.

The purpose of this priming is to instantiate all required classes and to translate the AWS Lambda programming model (and invocation) into the Quarkus programming model. Through the preinitialized call of the handleRequest method of the GetProductByIdHandler, the priming presented in method number 3 is also carried out automatically by the DynamoDB call.

To ensure that only this priming takes effect, please either comment out or remove the **@Startup** annotation in the following class AmazonDynamoDBPrimingResource.

However, I consider this priming technique to be experimental, as it naturally leads to a lot of extra code, which can be significantly simplified using a few utility methods. Therefore, the decision to use this priming method is left to the reader.

5. A Rebuilt Sample Application That We Build GraalVM Native Image and Deploy It As Lambda Custom Runtime

This article assumes prior knowledge of GraalVM and its native image capabilities. For a concise overview about them and how to get both installed, please refer to the following articles: Introduction to GraalVM, GraalVM Architecture and GraalVM Native Image.

Let's take a look at the differences to our previous example application. As far as the source code of the application is concerned, the ReflectionConfig class has been added. In this class, we use *@RegisterForReflection* annotation to define the classes that are only loaded at runtime.

```
@RegisterForReflection(targets = {
            APIGatewayProxyRequestEvent.class,
            HashSet.class,
            APIGatewayProxyRequestEvent.ProxyRequestContext.class,
            APIGatewayProxyRequestEvent.RequestIdentity.class,
            DateTime.class,
            Product.class,
            Products.class,
})
```

Since GraalVM uses Native Image Ahead-of-Time compilation, we need to provide such classes in advance, otherwise ClassNotFound errors will be thrown at runtime. This includes custom entity classes like Product and Products, some AWS dependencies to APIGateway Proxy Event Request (from the artifact id *aws-lambda-java-events* from pom.xml), DateTime class to convert timestamp from JSON to Java object and some other classes. It sometimes takes several attempts and running the application first to find all such classes.

There are other ways to register classes for Reflection, which are described in this article Tips for writing native applications .

In the pom.xml there are a few more additional declarations necessary. First we need to use Amazon DynamoDB Client Quarkus extension from Quarkiverse (without it we run into the error) as follows:

```
<dependency>
  <groupId>io.quarkiverse.amazonservices</groupId>
  <artifactId>quarkus-amazon-dynamodb</artifactId>
   <version>3.2.0</version>
</dependency>
```

This extension takes effect automatically, we do not have to change the source code of the application. We also need to activate the native profile as follows:

```
<profile>
   <id>native</id>
   <activation>
      <property>
          <name>native</name>
      </property>
   </activation>
   <properties>
      <quarkus.native.additional-build-args>
            --initialize-at-run-time=software.amazonaws.example.
            product.dao
       </quarkus.native.additional-build-args>
       <quarkus.native.enabled>true</quarkus.native.enabled>
   </properties>
</profile>
```

Here we initialize all classes in the **software.amazonaws.example.product.dao** package at runtime. Otherwise we run into the error when creating the GraalVM Native Image.

To build our application as a GraalVM native image, we have to pass the native profile as a parameter. The call then looks like this: **mvn clean package -Dnative**. This will save GraalVM Native Image as a file named boostrap built in function.zip.

The last part of the changes concerns AWS SAM template.yaml. Since there is no managed Lambda GraalVM runtime environment, the question arises how we can deploy our native GraalVM image on AWS Lambda. This is possible if we choose Lambda Custom Runtime as the

runtime environment (this currently only supports Linux) and deploy the built .zip file as a deployment artefact. You can find out more about this in the article Building a custom runtime for AWS Lambda. This is exactly what we define in template.yaml as follows:

```
Globals:
  Function:
    Handler: io.quarkus.amazon.lambda.runtime.Q
    uarkusStreamHandler::handleRequest
    CodeUri: target/function.zip
    Runtime: provided.al2023
...
```

With Runtime *provided.al2023* we define Lambda runtime environment as Amazon Linux 2023 Custom Runtime and with CodeUri target/function.zip we define the path to the deployment artifact compiled with Maven in the previous step. Deployment works the same way with **sam deploy -g**. You can see how to create and query the product in the initial sample application above. Please note that the GraalVM Native Image sample application uses a different API key, namely a6ZbcDefQW12BN56WES318, which is defined in template.yaml. Please use this in your curl calls (see above for examples).

We have now examined all 5 methods and want to measure the performance of the lambda function with all methods.

Here is a summary of the methods. We will later assign the results to the corresponding method number in the table below:

1. Measurement without activating Lambda SnapStart
2. Measurement with activation of Lambda SnapStart, but without using the priming methods
3. Measurement with activation of Lambda SnapStart and with application of priming of DynamoDB Request
4. Measurement with activation of Lambda SnapStart and with application of priming of API Gateway Request Event
5. Measurement with GraalVM Native Image

The results of the experiment are based on reproducing more than 100 cold starts and about 100,000 warm starts with the Lambda function *GetProductByIdFunction* (we ask for the already existing product with ID=1) for the duration of about 1 hour. We give Lambda function 1024 MB memory, which is a good trade-off between performance and cost. We also use (default) x86 Lambda architecture. For the load tests I used the load testing tool hey, which is very similar to Curl. However, you can use any tool you like, e.g. Serverless Artillery or Postman.

Before I present the performance measurements, a few notes on the scope of measurement for methods 1 to 4.

We will measure all 4 measurements with *tiered compilation* (which is default in Java 21, we don't need to set anything separately) and compilation option *XX:+TieredCompilation -XX:TieredStopAtLevel=1*. To use the last option, you have to set it in template.yaml in JAVA_OPTIONS environment variable as follow

```
Globals:
  Function:
    Handler: io.quarkus.amazon.lambda.runtime.
    QuarkusStreamHandler::handleRequest
    ...
    Environment:
      Variables:
        JAVA_TOOL_OPTIONS: „-XX:+TieredCompilation
        -XX:TieredStopAtLevel=1“
```

With these two we achieve the best performance with the different trade-offs. You can also read the article Optimizing AWS Lambda function performance for Java. This compilation option is logically irrelevant for the GraalVM Native Image.

Please also note the effect of AWS SnapStart Snapshot tiered cache. This means that in the case of SnapStart activation, we get the largest cold starts during the first measurements. Due to the tiered cache, the subsequent cold starts will have lower values. For more details about the technical implementation of AWS SnapStart and its tiered cache,

I refer you to the presentation by Mike Danilov: "AWS Lambda Under the Hood". Therefore, I will present the performance measurements for all approx. 100 cold start times (labelled as *all* in the table), but also for the last approx. 70 (labelled as *last 70* in the table), so that the effect of Snapshot Tiered Cache becomes visible to you. Depending on how often the respective Lambda function is updated and thus some layers of the cache are invalidated, a Lambda function can experience thousands or tens of thousands of cold starts during its life cycle, so that the first longer lasting cold starts no longer carry much weight.

Performance Measurement Results

Before we present the measurement results, a few notes on the abbreviations:

- We will assign the results in the table below to the corresponding method number 1 to 5. See the description above
- *C* in the column is the abbreviation for cold start
- *W* in the column is the abbreviation of warm start
- *P* in the column is the abbreviation for percentile
- All measurements

Measurement results for *tiered compilation:*

Methode Nummer	C P50	C P75	C P90	C P99	C P99.9	C Max	W 50	W 75	W 90	W P99	W P99.9	W Max
1	3344	3422	3494	3633	3904	3907	5.92	6.83	8.00	19.46	50.44	1233
2 / all	1643	1703	1953	2007	2084	2084	5.68	6.35	7.39	16.39	49.23	1386
2 / last 70	1604	1664	1728	1798	1798	1798	5.64	6.30	7.33	15.87	47.30	1286
3 / all	666	720	944	1117	1317	1318	5.73	6.45	7.57	16.01	39.07	566
3 / last 70	646	681	774	1043	1043	1043	5.64	6.40	7.51	15.75	37.54	566
4 / all	604	675	948	1181	1197	1198	5.25	6.25	7.33	15.62	41.64	338
4 / last 70	588	599	650	790	790	790	5.46	6.10	7.16	14.90	39.38	241

Measurement results for
XX:+TieredCompilation -XX:TieredStopAtLevel=1 compilation:

Methode Nummer	C P50	C P75	C P90	C P99	C P99.9	C Max	W 50	W 75	W 90	W P99	W P99.9	W Max
1	3357	3456	3554	4039	4060	4060	6.01	6.83	8.13	19.77	53.74	1314
2 / all	1593	1625	1722	1834	1930	1930	5.55	6.21	7.16	16.08	50.44	1401
2 / last 70	1574	1621	1685	1801	1801	1801	5.55	6.20	7.16	15.14	49.23	1401
3 / all	636	701	943	973	1055	1055	5.50	6.20	7.21	14.66	39.07	330
3 / last 70	628	654	692	859	859	859	5.5	6.15	7.04	14.08	37.25	270
4 / all	615	656	917	941	960	961	5.55	6.21	7.39	16.08	41.40	486
4 / last 70	595	619	653	765	765	765	5.47	6.21	7.27	16.08	39.94	486

Measurement results for *GraalVM Native Image:*

Method Number	C P50	C P75	C P90	C P99	C P99.9	C Max	W 50	W 75	W 90	W P99	W P99.9	W Max
5	604	616	629	742	749	749	10.75	15.02	18.04	28.18	52.47	1782

Conclusion

In this article, we have explored some ways to develop, deploy and run applications on AWS Lambda using Quarkus framework and measured performance (the cold and warm start times) of the Lambda function. We also showed how we can optimize Lambda function performance using Lambda SnapStart (including various priming techniques) and GraalVM Native Image deployed as AWS Lambda Custom Runtime.

The warm start times are very acceptable even without activating SnapStart, because Java itself is a very fast programming language. However, the warm start times with GraalVM Native Image are slightly higher than when using the managed Java 21 version in AWS Lambda.

We have seen that enabling Lambda SnapStart alone reduces the cold start time significantly and even further if we implement DynamoDB request priming (method 3), which requires some implementation effort. The experimental API Gateway request priming (method 4) reduces the cold start time even further, but requires writing a lot of code, so it is up to the reader to use it or not. However, the effect of AWS SnapStart Snapshot tiered cache is clearly visible.

However, we achieve the lowest cold start times with GraalVM Native Image.

However, I would recommend the reader to start with Lambda SnapStart, especially if the priming techniques as in method 3 are applicable. SnapStart is completely managed by AWS, so we do not have to worry about the creation, signing, fault-tolerant retention and recovery of a snapshot. In the case of GraalVM, we are responsible for CI/CD ourselves. Note that with both SnapStart (in the deployment phase) and GraalVM Native Image (when creating the native image) additional time is taken (it is several minutes), so I recommend working without activating SnapStart on all environments except *live/production*, for example.

There is much more we can explore for performance optimisation, e.g.

- Assign different RAM to the Lambda function. With more RAM, Lambda becomes more expensive, but has more CPU power and is therefore faster. Since one of the components of Lambda pricing is GB/s, it is worthwhile to experiment with RAM settings to determine the best price/performance ratio. Especially applications on AWS Lambda deployed as GraalVM Native Image remain quite performant with less memory.
- Set different implementations of the HTTP client when creating DynamoDbClient. The default is ApacheClient, but there is also UrlConnection and AWS CRT HTTP client provided by AWS. AWS CRT Client has recently added support for GraalVM Native Image.
- Instead of default x86 Lambda architecture set arm64 (which is 25% cheaper per GB/s as x86, see AWS Lambda pricing).

All these settings result in deviations in the cold and warm start times. However, the experiments are beyond the scope of this article, but I will publish them on my personal blog page over time.

> Back to Table of Content

#JAVAPRO #CLOUD

Modernize Java Applications with Amazon EKS: A Cloud-Native Approach

Author:

Sascha Möllering is a Principal Specialist Solutions Architect for Containers at AWS, where he has been helping customers architect and optimize their cloud-native applications for nearly a decade. With a deep passion for container technologies, serverless computing, and Java development, he works with organizations across the globe to design and implement scalable, resilient solutions on AWS. A regular speaker at industry conferences and technical events, Sascha shares his expertise through hands-on workshops, blog posts, and technical content. His practical experience and technical depth have made him a trusted voice in the cloud-native community. He is particularly known for his work in containerization strategies and Java application modernization on AWS. When not architecting solutions or speaking at events, Sascha actively contributes to the technical community through writing, mentoring, and developing educational content that helps organizations navigate their cloud transform.

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

Co-Author:

Yuriy Bezsonov
https://www.linkedin.com/in/yuriybezsonov/

Cloud-native development has become the cornerstone of modern application development, with containerization leading the charge in deployment strategies. For Java developers and enterprises with significant Java investments, the journey to cloud-native architecture presents both opportunities and challenges. This article explores how Amazon EKS, combined with cutting-edge technologies like Coordinated Restore at Checkpoint (CRaC) and Quarkus, is revolutionizing Java application deployment in the cloud.

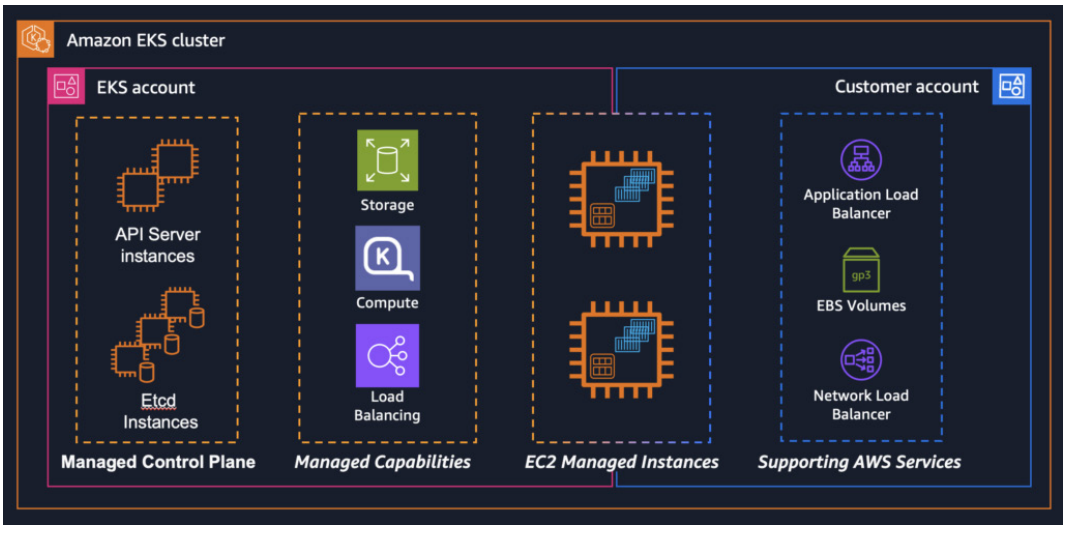
Introduction to Amazon Elastic Kubernetes Service

Amazon Elastic Kubernetes Service (Amazon EKS) is a fully managed Kubernetes service that enables customers to run Kubernetes seamlessly in both AWS Cloud and on-premises data centers. In the cloud, Amazon EKS automates Kubernetes cluster infrastructure management. This is essential for scheduling containers, managing application availability, dynamically scaling resources, optimizing compute, storing cluster data, and performing other critical functions. With Amazon EKS, customers can leverage the robust performance, scalability, reliability, and availability of AWS infrastructure, as well as natively integrate with AWS networking, security, and storage services. To simplify running Kubernetes in on-premises environments, customers can use the same Amazon EKS clusters, features, and tools to run nodes on AWS Outposts or their own infrastructure, or customers can use Amazon EKS Anywhere for self-contained, air-gapped environments.

EKS Auto Mode: The Next Evolution

With the Amazon EKS Auto Mode, customers can automate cluster management without deep Kubernetes expertise, as it selects optimal compute instances, dynamically scales resources, continuously optimizes costs, manages core add-ons, patches operating systems, and integrates with AWS security services. AWS expands its operational responsibility in EKS Auto Mode compared to customer-managed infrastructure in their

EKS clusters. In addition to the EKS control plane, AWS will configure, manage, and secure the AWS infrastructure in EKS clusters that customer's applications need to run.



Amazon EKS cluster architecture with Auto Mode

Customers can now get started quickly, improve performance, and reduce overhead, enabling them to focus on building applications that drive innovation instead of on cluster management tasks. EKS Auto Mode also reduces the work required to acquire and run cost-efficient GPU-accelerated instances so that generative AI workloads have the capacity they need when they need it. It automatically launches EC2 instances based Bottlerocket OS and AWS Elastic Load Balancing (ELB), and provisions Amazon Elastic Block Store (Amazon EBS) volumes inside user AWS accounts and user-provided VPCs when customers deploy their applications. EKS Auto Mode launches and manages the lifecycle of these EC2 instances, scaling and optimizing the data plane as application requirements change during run time, and automatically replacing any unhealthy nodes.

Challenges of Java in Containers

Running Java applications in containers presents a complex web of challenges that demand careful consideration and configuration. Memory management stands as perhaps the most critical concern - prior to Java 10, the JVM couldn't accurately detect container memory limits, leading to potential out-of-memory errors as it based calculations on host resources rather than container constraints. This complexity is compounded by native memory overhead from thread stacks, direct buffers, and JVM internal structures, which must be accounted for

alongside heap allocation.
Off-heap memory management adds another layer of complexity, as tools like Netty or memory-mapped files can bypass heap limits but still count against container memory limits. Startup time poses another significant challenge, particularly in dynamic container environments that demand rapid scaling. The time-consuming process of class loading, especially in large applications with numerous dependencies, can lead to slow container initialization. This is exacerbated by initial heap sizing decisions and JIT compilation overhead, where the tradeoff between startup speed and runtime performance becomes crucial. We will specifically address this challenge in the course of the article.

Resource utilization presents its own set of challenges - CPU throttling in containerized environments can affect Java's ability to optimize code execution, while memory swapping can severely impact performance if not properly managed or disabled. I/O constraints can become particularly problematic for Java applications that rely heavily on file operations or network communication, as container limitations may not be immediately apparent to the application. The traditional „fat JAR" approach commonly used in Java applications results in larger container images, increasing deployment times and resource consumption. Container images are created using layers. Layered JAR files separate the application and its dependencies so that each part can be stored in a dedicated container image layer. This has the advantage that the cached layers can be reused during the build of the application, which significantly speeds up the rebuild of the container image. This can also have an impact on startup time when using technologies like Seekable OCI (SOCI). SOCI is a technology open sourced by AWS that enables containers to launch faster by lazily loading the container image. SOCI works by creating an index (SOCI Index) of the files within an existing container image. This index is a key enabler to launching containers faster, providing the capability to extract an individual file from a container image before downloading the entire archive.

Furthermore, Java's garbage collection behavior can cause unexpected pauses, potentially violating container orchestrators' health checks and leading to unnecessary pod restarts. These challenges become even more pronounced in microservices architectures, where multiple Java containers may compete for resources on the same host, requiring

careful tuning of both JVM parameters and container resource limits to achieve optimal performance and reliability.

Better Performance with CRaC and Warp

Coordinated Restore at Checkpoint (CRaC), an innovative OpenJDK project spearheaded by Azul, represents a significant breakthrough in addressing Java's notorious startup time challenges. By capturing the state of a warmed-up Java application and JVM at any given moment („checkpoint"), CRaC enables applications to restart from this preserved state, effectively bypassing the traditional time-consuming initialization process. This checkpoint capability, which can be integrated as an additional step in container images builds, dramatically improves startup performance by restoring the application to its optimized state immediately. However, this powerful feature comes with important considerations - checkpoint files may contain sensitive data, and stateful elements like file handles and network connections require careful handling during restoration. While Spring Boot provides native support for CRaC, external libraries may need additional implementation work. For instance, applications using the AWS SDK for Java V2 must implement custom logic to rebuild connections after restore. CRaC addresses these challenges through its Resource interface, which provides `beforeCheckpoint()` and `afterRestore()` callbacks, allowing developers to manage state preservation and restoration effectively across their application components.

AWS has a demo application to demonstrate how to use CRaC in combination with the AWS SDK for Java. The application called `UnicornStore` used in this implementation interacts with Amazon EventBridge through the AWS SDK. First, a client is created, then this client is used for performing operations on a EventBridge. Each client maintains its own HTTP connection pool. To capture the checkpoint, the connections in the pool (network connections) needs to be closed — this is achieved by closing the client in `beforeCheckpoint()` method, and re-creating it in `afterRestore()`.
The code snippet below shows how `UnicornPublisher` class is altered to handle CRaC requirements for network connections through `org.crac.Resource`-interface:

```
public class UnicornPublisher implements Resource {
    ...
    @PostConstruct
    public void init() {
        createClient();
        Core.getGlobalContext().register(this);
    }

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context)
    throws Exception {
        logger.info(„Executing beforeCheckpoint...“);
        closeClient();
    }

    @Override
    public void afterRestore(Context<? extends Resource> context)
    throws Exception {
        logger.info(„Executing afterRestore ...“);
        createClient();
    }

    private void createClient() {
        logger.info(„Creating EventBridgeAsyncClient“);

        eventBridgeClient = EventBridgeAsyncClient
                .builder()
                .credentialsProvider(DefaultCredentialsProvider.
                 create())
                .build();
    }

    public void closeClient() {
        logger.info(„Closing EventBridgeAsyncClient“);
        eventBridgeClient.close();
    }
    ...
}
```

Spring has built-in CRaC support since version 6.1 (and Spring Boot since version 3.2), which means, among other things, that CRaC is integrated into the Spring Lifecycle (more information on this can be found here). With this approach, it's possible to solely snapshot the framework code but not the application code. The implication of this approach is of course that it's not necessary to change the application at all if you just use Spring Boot functionalities. With the automatic checkpointing, we don't have a fully warmed up JVM that is going to be snapshotted which means that the startup time is a bit slower compared to the manual snapshotting approach. The following Dockerfile shows a compact multi-stage build-approach to create a container image that is using a CRaC snapshot.

```
FROM azul/zulu-openjdk:21-jdk-crac-latest AS builder
RUN apt-get -qq update && apt-get -qq install -y curl maven
ARG SPRING_DATASOURCE_URL
ENV SPRING_DATASOURCE_URL=$SPRING_DATASOURCE_URL
ARG SPRING_DATASOURCE_PASSWORD
ENV SPRING_DATASOURCE_PASSWORD=$SPRING_DATASOURCE_PASSWORD

COPY ./pom.xml ./pom.xml
COPY src ./src/

# Build the application
RUN mvn clean package -ntp && mv target/store-spring-1.0.0-exec.jar
store-spring.jar

# Run the application and take a checkpoint
RUN <<END_OF_SCRIPT
#!/bin/bash
java -Dspring.context.checkpoint=onRefresh -Djdk.crac.collect-fd-
stacktraces=true \
    -XX:CRaCEngine=warp -XX:CPUFeatures=generic
    -XX:CRaCCheckpointTo=/opt/crac-files -jar /store-spring.jar&PID=$!
    wait $PID || true
    END_OF_SCRIPT

FROM azul/zulu-openjdk:21-jdk-crac-latest AS runner
RUN apt-get -qq update && apt-get -qq install -y adduser
```

```
RUN addgroup --system --gid 1000 spring
RUN adduser --system --disabled-password --gecos „“ --uid 1000 --gid
1000 spring

COPY --from=builder --chown=1000:1000 /opt/crac-files /opt/crac-files
COPY --from=builder --chown=1000:1000 /store-spring.jar /store-
spring.jar

USER 1000:1000
EXPOSE 8080

# Restore the application from the checkpoint
CMD [„java“, „-XX:CRaCEngine=warp“, „-XX:CRaCRestoreFrom=/opt/crac-
files“]
```

The containerization process for CRaC-enabled Java applications follows a multi-stage build approach utilizing Azul's `zulu-openjdk` with CRaC support as the foundation for both builder and runner stages. In the initial builder stage, the process compiles and packages the Unicorn Store application into a JAR file, then executes the application to generate a checkpoint of its warmed-up state. This crucial step captures the optimized runtime state of the JVM and application. The second stage establishes a clean runtime environment, where both the checkpoint files and the application JAR are copied from the builder stage and configured with appropriate permissions for the unprivileged ‚spring' user, ensuring security best practices. Finally, the container is configured to restore the application directly from the checkpoint files at startup, enabling rapid initialization by bypassing the traditional JVM warm-up phase and achieving near-instant application readiness.

The shell script starts the application with the JVM options `-Dspring.context.checkpoint=onRefresh` and `-XX:CRaCCheckpointTo=/opt/crac-files/`. As already indicated, checkpoint is created automatically at startup during the LifecycleProcessor.onRefresh phase.
With the parameter `-XX:CRaCEngine=warp` we've specified a specific engine called Warp. Warp is a new engine available in Azul Zulu builds that can fully replace CRIU (Checkpoint/Restore In Userspace), and does not require any extra privileges. This means, it's not necessary to add

additional permissions for the Kubernetes deployment and a second benefit is that Warp is faster than the CRIU based engine for the demo application. If you want to learn more about Warp, you can read the following third-party blog post .

Kubernetes-Native Java with Quarkus

Quarkus, designed as a Kubernetes-native Java framework, provides a performant foundation for containerized applications on Amazon EKS. Its container-first philosophy results in significantly faster startup times and lower memory footprint compared to traditional Java applications. When combined with Mandrel, Red Hat's downstream distribution of GraalVM, developers can compile their Quarkus applications to native executables that start in milliseconds and consume minimal memory - characteristics particularly valuable in containerized environments. On Amazon EKS, these native executables enable more efficient resource utilization, faster scaling operations, and reduced costs as more containers can be packed onto each node. Mandrel's compatibility with Quarkus ensures a stable and supported path to native compilation while maintaining access to key AWS services through Quarkus extensions. This combination delivers a production-ready stack for modern, cloud-native Java applications that fully leverage the orchestration capabilities of Amazon EKS while minimizing the traditional overhead associated with Java in containers.

The following section demonstrates how to create a Quarkus project with `quarkus-maven-plugin`:

```
mvn io.quarkus.platform:quarkus-maven-plugin:create \
-DprojectGroupId=com.amazon \
-DprojectArtifactId=quarkus-eks \
-DclassName="com.example.GreetingResource" \
-Dextensions="quarkus-container-image-jib,quarkus-kubernetes"
```

The quarkus-kubernetes extension helps generate Kubernetes manifests automatically. In the next step, we can package the application as a JVM-based container image using Jib:

```
mvn package -Dquarkus.container-image.build=true \
-Dquarkus.container-image.name=quarkus-eks \
-Dquarkus.container-image.tag=latest \
-Dquarkus.container-image.registry=<your-ecr-repo> \
-Dquarkus.container-image.push=true \
-Dquarkus.container-image.group=javapro
```

After we've built the image, we can push it to Amazon Elastic Container Registry (ECR), change the image path in the generated Kubernetes YAML file and deploy to EKS by applying Kubernetes YAML files:

```
kubectl apply -f target/kubernetes/kubernetes.yml
kubectl rollout status deployment quarkus-eks
```

By compiling the application to a native executable, we achieve **faster cold starts** and lower **memory usage**, ideal for autoscaling in Kubernetes. First we have to build our application using the native profile defined in `pom.xml`.

```
mvn package -Dnative
```

Now we can build and push the native image to ECR:

```
docker build -f src/main/docker/Dockerfile.native -t <your-container-
image>:latest .
docker push <your-container-image>:latest
```

And finally update and apply the Kubernetes deployment manifest.

Performance Results

At AWS, we've developed a comprehensive "Java on AWS"-Immersion Day, designed to help developers navigate the diverse landscape of Java deployments in cloud environments. This hands-on experience focuses particularly on demonstrating various approaches to running Java workloads on AWS infrastructure, addressing common challenges and modern solutions. One significant component of the program delves

into container optimization techniques. This section explores various strategies to enhance Java applications in containerized environments, helping developers achieve better resource utilization, faster startup times, and more efficient deployments while maintaining application performance and reliability.

Our optimization journey began with an unmodified container image as our baseline. From there, we gradually implemented various optimization techniques to enhance performance. We utilized Jlink and Jdeps to create a custom runtime environment, incorporated Jib for improved container builds, implemented Class Data Sharing (CDS) through archive creation, employed CRaC for container snapshots, and ultimately integrated GraalVM native compilation. Each of these steps was thoroughly tested on Amazon EKS, with measurements focusing on two critical metrics: the resulting image size and application startup time.

Version	Image Size	Start time (p99)
No optimization	351MB	6.459s
Custom JRE	221MB	6.019s
JIB	231MB	5.71s
CDX	633MB	3.194s
GraalVM	460MB	0.581s
CRaC	412MB	0.085s

Performance Comparison

We can see very clearly from the table that different optimizations lead to different results. Custom JRE and Jib shows a significant reduction in the size of the container image. In terms of startup times, GraalVM and CRaC clearly stand out with less than one second, for well-known reasons.

Summary

As containerization becomes increasingly central to modern application development, Java developers face unique challenges in optimizing their applications for cloud environments. This comprehensive guide explores how Amazon EKS, combined with cutting-edge technologies

like CRaC, Quarkus, and GraalVM, is transforming Java deployment in the cloud. To leverage these advances in your own applications, start by evaluating your current containerization strategy against our benchmark results, explore EKS Auto Mode for simplified cluster management, and consider implementing CRaC or Quarkus in combination with GraalVM for applications requiring rapid startup times.

> Back to Table of Content

For your IT projects you don't need a know-it-all.

You need a {BUDDY}

Richard Fichter
CEO @ XDEV

Java Champions

Outdated software? Rising maintenance costs? Security risks? We help you to make Java applications fit for the future - with a clear concept and at eye level. In addition to modern tools, we offer premium support for your **Java modernization.**

- **Not a know-it-all - a buddy:** We support your team with pragmatic methods without playing the wise guy.
- **Modernization with strategy:** agile methods & proof of concept for a secure update.
- **Robust solutions:** We rely on proven technologies and practical concepts - without unnecessary complexity.

Let us move your Java project forward together!

BEST {BUDDYS} IN CODE

trusted by

ATU · EMPORIO ARMANI · CONRAD · Fraport · Red Bull · Allianz

Arrange a free discovery call here!

XDEV