JAVA {MAGAZINE

.nl.jug

03 > 2024

Onafhankelijk tijdschrift voor de Java-professional

NLJUG INNOVATION AWARDS 2024

Java developers maken het verschil in innovatieve tech-ontwikkelingen

>KOTLIN 2.0
FEATURES AND CHANGES

>J-SPRING
TERUGBLIK

>SPRING BOOT AND GRAALVM:
A MATCH MADE IN HEAVEN

Become a NLJUG member!

NLJUG Base, Core, and Key memberships giving members privilege for different events each year. Such as Free access to J-Spring, J-Fall and TEQnation. And many more benefits inside the JAVA community.

NLJUG is a Java community dedicated to developing and making the platform accessible for knowledge sharing and networking. With over 4500 members and 80 businesspartners.

BECOME A CORE MEMBER

€ 125,–

› Priority access to the J-Fall conference
Food and drinks included

› Free Java Magazine subscription (4 issues a year)

› NLJUG newsletter containing the most relevant knowledge and meetups

› 50% discounted access to the J-Spring conference

› Free access to NLJUG University sessions

› 50% discounted acces to J-Fall Pre-con Masterclasses

› Additional exclusive discounts on courses and training organized by our partners

› 50% discounted acces to TEQnation conference

BECOME A KEY MEMBER

€ 279,–

› VIP access to the J-Fall conference
Food and drinks included

› Free Java Magazine subscription (4 issues a year)

› NLJUG newsletter containing the most relevant knowledge and meetups

› Free access to J-Spring conference (worth € 169,-)

› Free access to NLJUG University sessions

› Free access to J-Fall Pre-con Masterclasses (worth € 129,-)

› Additional exclusive discounts on courses and training organized by our partners

› Access to J-Spring and J-Fall VIP Lounges incl. reserved free parking

› Exclusive access to the NLJUG Core Slack channel, where you can discuss the future of our community

› NLJUG Key Member Swag (including NLJUG Core Member Hoody, Core Member pin and much more)

› NLJUG Key Member visible on conference badges

› Free acces to TEQnation conference (worth € 239,-)

Advantages of becoming an NLJUG member:

› Free access to J-Fall
› Free access to the NLJUG University sessions
› 4 times a year receive the (printed) Java Magazine
› Discounts on partner events

https://nljug.org/lidworden/

{ COLOFON } JAVA MAGAZINE 03-2024

Project coördinatie:
Lieke Stomps

Communitymanager:
Simon de Groot

Eindredactie:
Suzanne Muller

Auteurs:
Daan van Berkel, Geert Mulders, Alina Yurenko, Julien Lengrand-Lambert ,Giovanni van der Schelde ,Michael Koers ,Hanno Embregts ,Tim te Beek ,Erwin de Gier

Redactiecommissie:
Hanno Embregts, Julien Lengrand-Lambert, Helma Maassen, Elias Nogueira, Maja Reißner, Ivo Woltring, Thomas Zeeman

Vormgeving:
Wonderworks, Haarlem

Traffic & Media order:
Marco Verhoog

Drukkerij:
Senefelder Misset, Doetinchem

Advertenties:
Richelle Bussenius
E-mail: richelle.bussenius@nljug.org
Telefoon: 023 752 39 22
Fax: 023 535 96 27

Abonnementenadministratie:
Tanja Ekel

De prijs van een lidmaatschap van de NLJUG verschilt per membership tier die u afneemt. Een reguliere "base" membership van de NLJUG kost € 68,50 per jaar, waarbij Java Magazine gratis verschijnt. Naast het Java Magazine krijgt u gratis toegang tot de vele NLJUG workshops en het J-Fall congres. Het NLJUG is lid van het wereldwijde netwerk van JAVA user groups. Voor meer informatie of wilt u lid worden, **zie www.nljug.org**. Een nieuw lidmaatschap wordt gestart met de eerst mogelijke editie voor een bepaalde duur. Het lidmaatschap zal na de eerste (betalings)periode stilzwijgend worden omgezet naar lidmaatschap van onbepaalde duur, tenzij u uiterlijk één maand voor afloop van het initiële lidmaatschap schriftelijk (per brief of mail) opzegt. Na de omzetting voor onbepaalde duur kan op ieder moment schriftelijk worden opgezegd per wettelijk voorgeschreven termijn van 3 maanden.
Een lidmaatschap is alleen mogelijk in Nederland en België.
Uw opzegging ontvangen wij bij voorkeur telefonisch. U kunt de Klantenservice bereiken via: 023-5364401. Verder kunt u mailen naar **members@nljug.org** of schrijven naar NLJUG BV, Ledenadministratie, Nijverheidsweg 18, 2031 CP Haarlem.
Verhuisberichten of bezorgklachten kunt u doorgeven via **members@nljug.org** (Klantenservice).

Wij nemen je gegevens, zoals naam, adres en telefoonnummer op in een gegevensbestand. De verwerking van uw gegevens voeren wij uit conform de bepalingen in de Algemene Verordening Gegevensbescherming. De gegevens worden gebruikt voor de uitvoering van afgesloten overeenkomsten, zoals de abonnementenadministratie en, indien je daar toestemming voor hebt gegeven, om je op de hoogte te houden van interessante informatie en/of aanbiedingen. Je kunt uw persoonsgegevens opvragen om inzicht te krijgen in welke gegevens wij van je hebben, deze te corrigeren of, na beëindiging van de abonnee-overeenkomst, te laten verwijderen. Stuur hiertoe een kaartje aan NLJUG BV, afd. klantenservice, Nijverheidsweg 18, 2031 CP Haarlem of een e-mail naar **members@nljug.org**.

VOORWOORD

Zomer

Met trots presenteer ik jullie alweer het derde nummer van Java Magazine in 2024. En ja, nog steeds in print! *The last of the Mohicans*? Wij zijn in ieder geval de enige in de wereld. Geen enkele Java Community wereldwijd heeft een eigen magazine, laat staan in print.

Binnen het NLJUG-bestuur en de redactiecommissie discussiëren we vaak: moet het magazine niet digitaal worden, of beide? Wie leest het nog? Willen we überhaupt nog een magazine? Moeten we onze aandacht niet verleggen naar de maandelijkse NLJUG-nieuwsbrief? Deze discussies eindigen altijd met 'laten we het nog even volhouden' en 'moeten we het niet de community zelf vragen?' Dat is precies wat we gaan doen. Binnenkort zullen we via een online enquête jullie mening vragen.

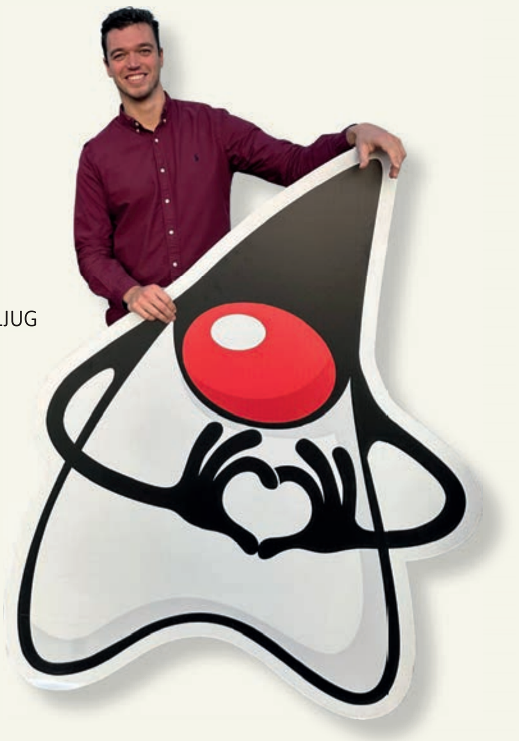
Niet alleen over het magazine, maar ook over ons onlangs geïntroduceerde membershipmodel, en natuurlijk over J-Fall zelf, J-Spring en andere meet-ups. Wij kunnen veel bedenken, maar het gaat erom wat jullie echt willen. Alleen zo houden we de community gezond.

En nu naar de inhoud van deze derde editie van het Java Magazine. De zomer is in volle gang en hoewel je zou denken dat we met z'n allen in zomerslaap zijn, is niets minder waar. We blikken terug op een geslaagde TEQnation en een spetterende J-Spring! Ook nu weer inspirerende kennisdossiers over onder anderen Spring Boot en GraalVM (bijdrage van Alina Yurenko) en AI in Java met LangChain4j (bijdrage van Michael Koers). Laat je inspireren door deze en alle andere artikelen.

Heb jij ook een interessant artikel of een leuk idee om te delen met de community? Voor op de website, in de nieuwsbrief of in het magazine? Aarzel dan niet! Wij zijn dol op inzendingen en kijken graag met je mee. Mail naar **info@nljug.org**. En als je het lastig vindt om je artikel vorm te geven dan biedt ons team van experts begeleiding aan.

Tot op J-Fall 2024!

Met vriendelijke groet,



Simon de Groot
Communitymanager NLJUG

A team of six people created Tikkie. After successfully launching an app like Tikkie, it would be easy to say "Okay, we made it". But that's not how innovation works. Innovation is about constantly challenging yourself and spotting opportunities. Daring to acknowledge that something can always be better. Because it's this mindset that makes the difference. And to make a difference we need more innovators – people like those who developed Tikkie and Groepie. We need you. Employees that will continue to motivate others, and us.

www.workingatabnamro.com

INHOUD

SCHRIJVEN VOOR JAVA MAGAZINE?

Ben je een enthousiast lid van NLJUG en zou je graag willen bijdragen aan Java Magazine? Of ben je werkzaam in de IT en zou je vanuit je functie graag je kennis willen delen met de NLJUG-community? Dat kan! Neem contact op met de redactie, leg uit op welk gebied je expertise ligt en over welk onderwerp je graag zou willen schrijven. Direct artikelen inleveren mag ook. Mail naar info@nljug.org en wij nemen zo spoedig mogelijk contact met je op.

PARSER COMBINATORS A 3-act play

On the interface of systems, data often gets transformed from one form to the other. With the ubiquitous data-format JSON this is often unnoticed. One just includes a library and maybe set some annotations are done.

Geert Mulders is a prolific software developer at Open-Value and currently works for Alliander. At Alliander. He creates software that mitigates prognosed power problems in the electricity network.

Daan van Berkel recently started a new role as Solution Architect at Alliander, where he tries to fascilitate teams to utilize the electricity network more efficiently.

You're not always that lucky. Sometimes it is necessary to handle data in an unconventional, or ad-hoc, format. Without the crutch of a library to rely on developers need to express in code how to transform one kind of data into an in-memory representation of the model the data represents. This is called *parsing* and it is an important tool in a developer's toolbox. However, developers often use sub-par mimicries of actual parsers that barely do the job.

In this play, we will remedy that and introduce the reader to the idea of *parser combinators*. It is best performed in an open office plan to help the germination and spread of the ideas presented here.

{ ACT ONE: A PARSER }

where our protagonists Daan en Geert meet each other behind the keyboard.

D: Geert, what is a parser actually?
G: Glad you asked Daan! A parser is a function that transforms text into objects.
D: So this would be a good signature

```
(String) -> T
```

G: Hmmm, almost
D: What is wrong with it?
G: Well, what would you return when you expect a number and the text is "banana"?
D: Ah, I see. We are missing the possibility to express a failure. Would this be better?

```
(String) -> T?
```

G: Yes, this is somewhat better... But we can do even better!
D: How?
G: What if we could combine functions with this *parser signature* in a way that they become bigger and better parsers?
D: Sounds great! How would we go about that?
G: Say a text that starts with a number, but ends with "+10", we would like the next parser to continue where we left off. So we need a way to return this left-over text.
D: So what about this:

```
(String) -> Pair<T, String>?
```

G: Yes! Maybe we should make this official?
D: Let's do it!

```
typealias Parser<T> = (String) -> Pair<T, String>?
```

D: You said something about combining parsers?
G: I did! But maybe it is better to implement a few simple parsers first. Just to play with the signature and the concept a little bit.

D: Good idea! What about a parser that parses a single 'A' from a String?
G: Yes, good one! What would you suggest?

D: I think the following does implement the Parser<Char> signature:

```
fun parseA(input: String): Pair<Char, String>? =
 if (input.startsWith('A')) {
 'A' to input.drop(1)
 } else {
 null
 }
```

G: It does! Would it not be great if you could make this more generic?
D: Do you want to accept the character to match? Like so

```
fun character(needle: Char): Parser<Char> = { input
->
 if (input.startsWith(needle)) {
 needle to input.drop(1)
 } else {
 null
 }
}
```

D: Wait, I am actually returning a function!
G: That is what functional programming is all about.
D: My head hurts a little, and it's late now, let's get some rest and continue tomorrow?
G: Okay! Goodnight!

{ ACT TWO: COMBINING PARSERS }

After a good night sleep our protagonists meet again

G: Hi Daan, are you well rested, ready to go?
D: I am well rested, thank you. I was dreaming about parsers. You mentioned that it would be nice to combine them. How does that work?
G: Well, let's say you have a parser for an 'A' and a parser for a 'B'. You want to have a parser that parses either 'A' or 'B'.
D: That would be nice. Would this work

```
fun or(p: Parser<Char>, q: Parser<Char>):
Parser<Char> = { input ->
 p(input) ?: q(input)
}
```

G: That is a good first try. But can you improve it?
D: I am not sure.
G: Take a look at the type arguments. Now it can only be used with a `Parser<Char>`.
D: I understand! Let's make it more generic

```
fun <T> or(p: Parser<T>, q: Parser<T>): Parser<T> =
{ input ->
 p(input) ?: q(input)
}
```

G: let's try it out

```
fun main() {
 val p = or(character('A'), character('B'))
 println(p("A"))
 println(p("B"))
 println(p("C"))
}
('A', "")
('B', "")
null
```

D: It works!
G: Whoop whoop!

G: If we can do `OR` we can do `AND`.
D: 🤯
D: Like this?

```
fun <S, T> and(p: Parser<S>, q: Parser<T>):
Parser<Pair<S, T>> = { input ->
 p(input)?.let { ( first, intermediate) ->
 q(intermediate)?.let { (second, rest) ->
 (first to second) to rest
 }
 }
}
```

G: Looking good! But the parser returns a Pair, what if you want something else?
D: Can we transform it?
G: I like how you are thinking! High five!
D: I will get to it.

```
fun <S,T> map(p: Parser<S>, transform: (S) -> T):
Parser<T> = { input ->
 p(input)?.let { (result, rest) -> transform(result)
to rest }
}
```

G: Let's see if it works

```
fun main() {
 val p = map(and(character('A'), character('B'))) {
(left, right) ->
 "$left$right"
 }
 println(p("ABC"))
}
("AB", "C")
```

D: The result is no longer a `Pair<Char`, `Char>`, but a `String`! Just as expected!
G: What other combinators can you think of?
D: Let me see... zero or more occurrences of 'A'?
G: That is a good suggestion.
D: Hmm, let me think of an implementation.

```
fun <T> star(p: Parser<T>): Parser<List<T>> = {
input ->
 val result = p(input)
 if (result != null) {
```

```
 val (match, intermediate) = result
 val (remaining, rest) = star(p)(intermediate)
 (listOf(match) + remaining) to rest
 } else {
 emptyList() to input
 }
}
```

D: Would this work?
G: Lets see

```
fun main() {
 val p = star(character('A'))
 println(p("AAAAAB"))
}
(['A', 'A', 'A', 'A', 'A'], "B")
```

G: It works, but looking at the code it feels a bit … imperative.
D: What do you suggest?
G: We can express the star operation in terms of '`and`', '`or`' and '`map`'
D: That is a novel idea! Let me puzzle on this one:

```
fun <T> star(p: Parser<T>): Parser<List<T>> =
 or(
 map(and(p, star(p))) { (r, rs) -> listOf(r) + rs },
 { input -> emptyList<T>() to input }
 )
```

G: Very nice! You have defined the star combinator recursively in terms of itself! This compiles, but will give you an error on runtime. Shall we see?
D: Yes! Let's go!

```
Exception in thread "main" java.lang.
StackOverflowError
 at kotlin.jvm.internal.Intrinsics.
checkNotNullParameter(Intrinsics.java:130)
 at com.alliander.ptutools.TempJavaMagParserKt.
star(TempJavaMagParser.kt)
 at com.alliander.ptutools.TempJavaMagParserKt.
star(TempJavaMagParser.kt:62)
 at com.alliander.ptutools.TempJavaMagParserKt.
star(TempJavaMagParser.kt:62)
 ...
```

D: Oops! We need a bigger stack!
G: No, we should delay the evaluation of the inner star combinator to when we really need it!
D: I am puzzled, what do you mean?
G: We can make the evaluation lazy.

```
fun <T> lazy(parserProducer: () -> Parser<T>):
Parser<T> = { input ->
 parserProducer()(input)
}
```

```
fun <T> star(p: Parser<T>): Parser<List<T>> =
 or(
 map(and(p, lazy { star(p) } )) { (r, rs) ->
listOf(r) + rs },
 { input -> emptyList<T>() to input }
 )
```

D: I am so excited, and I just can't hide it!

```
fun main() {
 val p = star(character('A'))
 println(p("AAAAAB"))
}
(['A', 'A', 'A', 'A', 'A'], "B")
```

G: Works like a charm! There is one small remark left. The second argument of the `or`, is a function that returns a parser that always succeeds.
D: We can make it more general! How about `succeed`? And then we can also create a fail combinator.

```
fun <T> succeed(value: T): Parser<T> = { input ->
 value to input
}

fun <T> fail(): Parser<T> = { _ ->
 null
}
```

D: Wow, this combinator-stuff is powerful, but also draining. Can we pick this up again tomorrow?
G: Sure thing!

{ ACT THREE: FUNCTIONAL PARSER }

Both our protagonists went for a walk outside to clear their heads, before coming back to the problem at hand.

D: How are you today, Geert?
G: Great! Do you want to continue playing around with parser combinators?
D: Yes, I would like to continue. I want to parse the following kind of binary expressions:

```
110010+1100-10010-110+100
```

D: It should return 'the answer'.
G: We can surely parse such expressions with our library of combinators.
G: What parts do you recognize in your expression
D: Hmmm,… I see binary numbers, like 110 and 110010, as well as operators `+` and `-`.
G: Let's focus on the numbers first.
D: Okay. Let me give it a try.

```
fun number(): Parser<Int> =
 and(leadingDigit(), star(digit()))
```

```
fun leadingDigit() =
 character('1')
```

```
fun digit() =
 or(character('0'), character('1'))
```

G: This will not compile because you did not map the result to an Int.
D: You are so smart!

```
fun number(): Parser<Int> =
 map(and(leadingDigit(), star(digit()))) { (digit,
digits) ->
 (listOf(digit) + digits).fold(0) { acc, d ->
 2 * acc + d
 }
 }

fun leadingDigit(): Parser<Int> =
 map(character('1')) {_ -> 1}

fun digit(): Parser<Int> =
 or(map(character('0')) {_ -> 0},
map(character('1')) {_ -> 1})
```

G: Let's see if it will parse a binary number.
D: I am confident, that it will work!

```
fun main() {
 val p = number()
 println(p("101010"))
}
(42, )
```

D: It works like a charm!
G: Let's move on to the operators.
D: I think I have it

```
fun operator() =
 or(character('+'), character('-'))
```

G: Aren't you forgetting something?
D: Of course! I should map it to something that we need.
G: Exactly.
D: Would this work?

```
fun operator(): Parser<(Int, Int) -> Int> =
 or(
 map(character('+')) {_ -> Int::plus },
 map(character('-')) {_ -> Int::minus },
 )
```

G: Excellent!
G: Now you are ready to write the entire parser, my young Padawan!
D: Am I?

D: I will try

```
fun expression(): Parser<Int> =
 or(
 map(and(
 and(number(), operator()),
 lazy(::expression),
 )) { (first, right) ->
 val (left, op) = first
 op(left, right)},
 number()
 )
```

PARSING IS AN IMPORTANT TOOL IN A DEVELOPER'S TOOLBOX

D: I think this works
G: Let's try it

```
fun main() {
 val p = expression()
 println(p("100000+1000+10"))
}
(42, )
```

{ CLOSING THOUGHTS }

Later the protagonists gathered for a final time.

D: Shall I commit this to the repository?
G: No!
D: Why not?
G: Although there is nothing wrong with the code we have written, it is not complete nor battle-tested.
G: You learned a valuable tool, but we should seek a library that fits our needs
D: 👷
D: You are right, I got carried away in my enthusiasm.

{ ASSIGNMENTS }

The number parser in the functional parser section is not complete. It does not parse the valid binary number 0. Can you fix the number parser to accept 0 as well?

The ergonomics of the combinator leave something to be desired. Can you write an and/or parser that allows any number of arguments?

There is a subtle bug with associativity in the expression parser. Did you catch that? Can you fix it? ‹

SPRING BOOT AND GRAALVM: A MATCH MADE IN HEAVEN

Hello, Java Magazine 👋! I'm very excited about this opportunity to tell you about GraalVM, Spring Boot, how they are even more awesome together, and why you should care. Let's go!

Alina Yurenko is a developer advocate for GraalVM at Oracle Labs, a research & development organization at Oracle. Loves both programming and natural languages, compilers, and open source.

{GRAALVM}

GraalVM is a JDK, like the other JDKs you might know, but with unique features and capabilities.

One such feature is the *Graal JIT compiler*. It's a highly optimizing compiler, which is the outcome of over a decade of research at Oracle Labs. By moving to the GraalVM JIT (by just setting GraalVM as your Java runtime), you can potentially win an extra 10-15% performance improvement. Not every application is performance-critical, but for those that are, every percent of performance improvement — especially with easy migration and no manual tuning required — is a huge win. This improvement is confirmed by both large-scale organizations, such as Oracle NetSuite, using GraalVM JIT in production at scale, and community reports, such the one from Ionut Balosin and Florin Blanaru [1], where the Oracle GraalVM JIT shows a performance improvement of 23% on x86_64 and 17% on arm64 compared to the C2 JIT compiler.

Another feature that GraalVM adds to the JDK is the ability to embed other languages in your Java application. We are lucky to have the incredibly rich Java ecosystem at our fingertips — whenever we have a problem that needs solving, there's always a Java library or tool for that. But sometimes we need to reach out to a specific library from another ecosystem, such as the cool ML libraries in Python, or utilize the scripting capabilities of JavaScript. GraalVM, and more specifically its Truffle component, enables you to do just that: embedding what we call *guest language code* in your Java application. You use what you need to use, and GraalVM will take care of the rest: interoperability, performance, tooling, and security. By using GraalVM, you can combine the rich and powerful Java platform with any other library or tool that you like — how cool is that?

Last, but not least, is GraalVM's Native Image, which enables you to compile your application ahead of time into a small and fast native executable. This is our main topic for today — so let's dive in.

{MEET GRAALVM NATIVE IMAGE}

So what is *Native Image* and how does it work exactly? Native Image is a feature in GraalVM that employs the Graal compiler to compile your application ahead of time into a native executable. The main reason you might want to do this, is to shift the majority of the work that the JVM normally performs at run time (such as loading classes, profiling, and compilation) to build time, removing that overhead when you run your application in the process. In addition to *ahead-of-time* (AOT) compilation, Native Image takes a snapshot of your heap with objects that are safe to initialize at build time, to also reduce the allocation overhead.

Native Image compiles your application into a native executable with the following advantages:

- Fast startup and instant peak performance, as the native executable doesn't need to warm up;
- Low memory footprint, as no memory is used to profile and compile code at runtime;
- Throughput on par with the JVM;
- Compact packaging by including only reachable code;
- Additional security, by eliminating unused code and reducing the attack surface.

{SPRING AOT: SPRING MEETS GRAALVM 🤝}

Now let's talk about Spring Boot, GraalVM, and how they come together. Spring Boot 3.0 introduced the general availability of GraalVM Native Image in Spring projects, and here is how it works.

By default Spring Boot pulls your project's bytecode classes/jars and configuration from all the different sources at runtime, resolves it, and creates an internal representation of your application. Interestingly, GraalVM Native Image does something very similar — it analyzes your code and configuration and creates an internal representation of your application — but it does this at build time. The Spring AOT engine was designed to bridge this gap between two worlds. It transforms your application configuration into native-friendly functional configuration and generates additional files to assist native compilation of Spring projects:

- Java source code (functional configuration);
- Bytecode for things such as dynamic proxies;
- Runtime hints for dynamic Java features when necessary (reflection, resources, and so on).

{ BUILD A NATIVE SPRING APPLICATION }

Let's go to Josh Long's second favorite place [2] and generate our project. The settings I chose are Spring Boot 3.3.1, Java 22, Maven, and my dependencies are Spring Web and GraalVM Native Support. That's all. Let's download and unpack our project, and add a `HelloController.java` so we have something to work with (Listing 1).

L1

```
package com.example.demo;

import org.springframework.web.bind.annotation.
RestController;
import org.springframework.web.bind.annotation.
GetMapping;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello from GraalVM and Spring!💃";
    }
}
```

Guess what – we also need GraalVM. The easiest way to install it on Linux and macOS is with SDKMAN! [3] As I'm writing this article, the latest released version is GraalVM for JDK 22, but we can also be cool and get the early access builds of GraalVM for JDK 23:

```
sdk install java 23.ea.9-graal
```

Now we're all set! Because GraalVM is a normal JDK, you can run your application as you would on any other JDK:

```
mvn spring-boot:run
...
Tomcat started on port 8080 (http) with context path '/'
Started DemoApplication in 1.106 seconds (process
running for 1.363)
```

Navigate to **http://localhost:8080/hello** and you'll see our message.

So far so good, but where's the fun in that? Let's compile it to a native executable with GraalVM Native Image:

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

While this command looks simple on the surface, it invokes the process of analyzing your whole application, pulling its configuration, discovering the reachable code, snapshotting the heap, and then optimizing and compiling. Look at the analysis step – even our fairly trivial app with two user classes (albeit also dependencies on Spring modules and JDK classes) contains quite a few things:

```
[2/8] Performing analysis... [*****] (15.0s @ 1.53GB)
 16,960 reachable types (89.4% of 18,971 total)
 26,038 reachable fields (59.2% of 44,015 total)
 89,842 reachable methods (64.9% of 138,456 total)
```

On my fairly average Linux cloud instance (16 CPU, 32 GB RAM), the build takes 1m 15s by default, and 46.7s with the quick build mode (-Ob).

Now let's run our application:

```
./target/demo
...
Tomcat started on port 8080 (http) with context path '/'
Started DemoApplication in 0.048 seconds (process
running for 0.051)
```

Navigating to **http://localhost:8080/hello** gives us the same message, only now our application is much faster—it started in 48 milliseconds!

We can also quickly assess the runtime characteristics of our application. The size of our application is 62MB, and measuring the runtime memory usage (RSS) while serving incoming requests gives us 69MB. Let's explore performance a bit more, by talking about specific performance optimizations in Native Image.

{ OPTIMIZE PERFORMANCE }

You might say: "OK, I can see how compiling my applications with Native Image is great for startup, memory usage and packaging size, but what about peak performance?" Indeed, we know that at runtime the JVM monitors our application, profiles it, and adapts on the go to optimize the most frequently executed parts. And we said that Native Image compilation takes place before runtime, so how can you optimize for peak performance? Glad you asked! Let's talk about profile-guided optimizations.

{ PROFILE-GUIDED OPTIMIZATIONS }

Profile-guided optimizations (PGO) allow you to build an instrumented version of your application, do a 'training run' applying relevant workloads, and generate a profile file that will be automatically picked up by Native Image to guide optimizations. This way, you get the best of both worlds: the runtime awareness of the JVM and the powerful AOT optimizations of Native Image.

Let's build a PGO-optimized application. To follow along, you can clone this related repo, which comes with profiles and benchmarking scripts: **https://github.com/alina-yur/nljug-native-spring-boot.**

1. Build an instrumented image:
`mvn -Pnative,instrumented native:compile`
2. Run the app and a workload (using the `hey` command-line tool [4]) to simulate the expected production behavior:
```
./target/demo-instrumented
hey -n=1000000 http://localhost:8080/hello
```
3. After you shut down the app, the file *default.iprof* will appear in your working directory.
Build the app with the *optimized* profile (this will pick up the *default.iprof* file):
`mvn -Pnative,optimized native:compile`

This command will generate *demo-optimized* that is aware of application behavior at runtime and has high performance.

Let's look at other performance optimizations available in GraalVM Native Image, and then do performance benchmarking.

{ ML-ENABLED PGO 👩‍🔬 }
The PGO approach described above— where the profiles are collected during a training run and tailored to your app — is the recommended way to do PGO in Native Image.

However, situations may arise when it's not possible to collect profiles, because of your deployment process for example. In that case, it's still possible to get profiling information and optimize your app via machine-learning-enabled PGO. Native Image contains a pre-trained ML model that predicts the probabilities of control flow graph branches, which enables it to additionally optimize the app. This is again available in Oracle GraalVM: it activates automatically in the absence of user-provided profiles.

If you are curious about the impact of this optimization, you can disable it with `-H:-MLProfileInference`. Our measurements indicated a ~6% runtime performance improvement, which is pretty cool for an optimization you get automatically!

{ G1 GC }
There are various *garbage collector* (GC) implementations in Native Image. The default GC in Native Image (*Serial GC*) can be beneficial if you have a short-lived application or want to optimize memory usage. And if you are aiming for the best peak throughput, Oracle GraalVM also supports G1 GC.

In our *optimized* profile it's enabled via `<buildArg>--gc=G1</buildArg>`.

{ OPTIMIZATION LEVELS IN NATIVE IMAGE }
There are several levels of optimizations in Native Image, which can be set at build time for different purposes:

-O0 - No optimizations: Recommended optimization level for debugging native images;
-O1 - Basic optimizations: Basic GraalVM compiler optimizations, still works for debugging.
-**O2** - Advanced optimizations: default optimization level for Native Image.
-O3 - All optimizations for best performance.
-Ob - Optimize for fastest build time: use only for development purposes for faster feedback, remove before compiling for production deployment.
-pgo - Using PGO will automatically trigger -O3 for best performance.

{ MARCH=NATIVE }
If your production CPU architecture matches your development environment (or shares the same CPU features), use `-march=native` for additional performance. This option enables the Graal compiler to use all the CPU features available on the build machine. Note that if you are building your application to distribute to your users or customers (where the exact production environment is unknown), it's better to use the compatibility mode: `-march=compatibility`.

{ PERFORMANCE COMPARISON }
Now that we talked about performance optimizations in Native Image, let's put them to the test. Let's run our Spring Boot application on the JVM and Native Image, and then compare the results. To benchmark the peak throughput, we will send 2 batches of 250000 requests (as a warmup and benchmarking run) using hey. We'll also use psrecord to produce charts showing the memory and CPU consumption.

Run `bench-jit-c2.sh` and `bench-native.sh`. See Image 1 for the results I got on my machine.

We can conclude that:

- Natively compiled applications can perform on par with the JVM! For the best throughput, use PGO, G1 GC and -march=native.
- Applications have instant performance and can also be scaled to zero whenever needed.
- More performance metrics are available than we covered so far.
- An application produced by Native Image uses significantly less

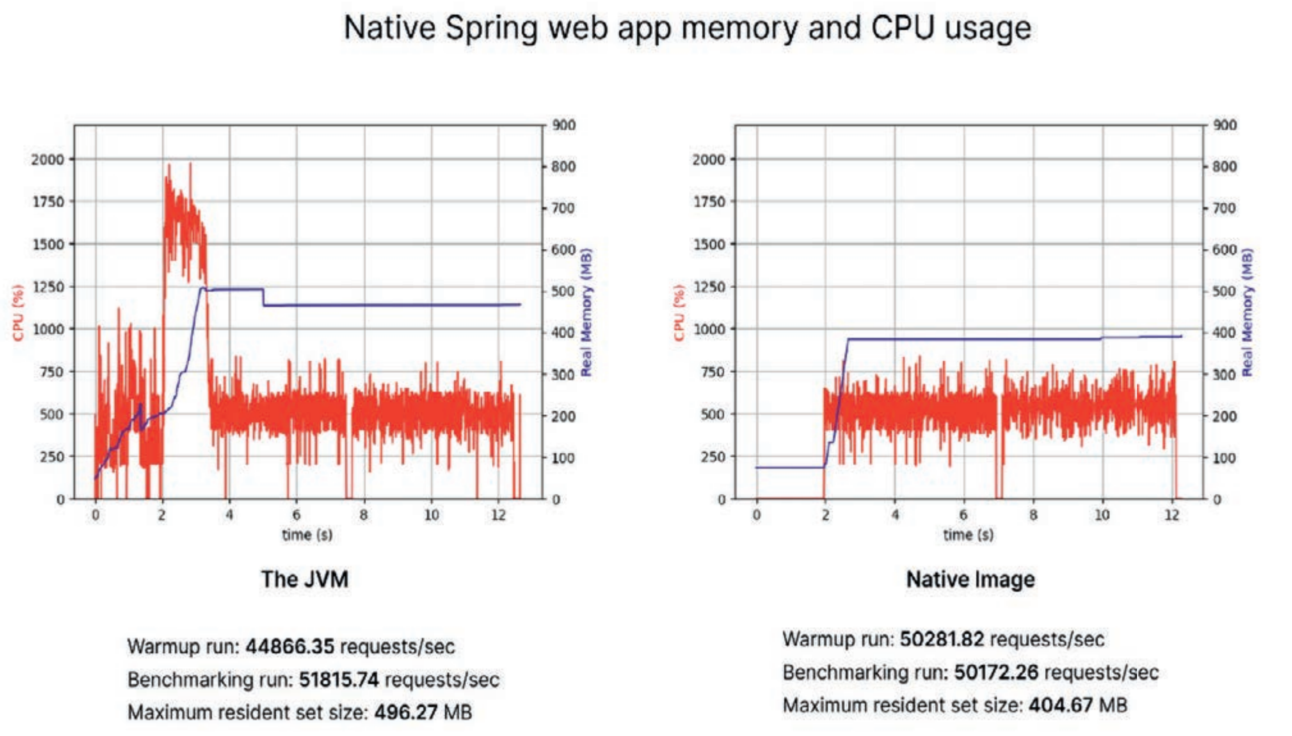


Image 1.

memory than the JVM. You can easily reduce memory by ~2x and keep the same throughput, or deploy two instances of the app for the memory footprint of one.

Our app is a good starting point, but since it doesn't perform any sophisticated logic yet, there is not much the Graal compiler can do. As you expand your project – reading and writing data from a database, adding libraries, and implementing your business logic – the Graal compiler will truly shine. We measured a more complex benchmark example (Spring Petclinic), and AOT outperformed JIT [5] in terms of peak throughput, latency, and memory efficiency.

Now that we have an app that starts fast, uses less memory, and has good peak performance, let's discuss extending our project. The very first step you will probably take (and the number one topic I'm asked about) is using libraries in the native mode.

{ USING DYNAMIC JAVA FEATURES AND THIRD-PARTY LIBRARIES }

Native Image compiles your applications ahead of time. Since we need a complete picture of your application, it compiles under a closed-world assumption: everything there is to know about your app needs to be known at build time. Native Image's static analysis will try to make the best possible predictions about the runtime behavior of your application. When that is not sufficient, you might need to provide it with configuration files to make things such as reflection, resources, JNI, serialization, and proxies 'visible' to Native Image. Note: the word 'configuration' doesn't necessarily mean that this is manual work – let's look at all the many ways this can work:

- A library that is Native-Image-friendly out of the box – this is obviously the ideal scenario [6].
- Existing libraries could need additional configuration to make things such as reflection work. Ideally, this configuration will be included within the library itself, like the way H2 handles this [7]. With this approach no further action is needed from a user – things just work.
- When a library doesn't support GraalVM (yet), the next best option is to configure it in the GraalVM Reachability Metadata Repository [8]: a centralized repository to which both maintainers and users can contribute and then reuse configuration for Native Image. It's integrated into Native Build Tools [9] and now enabled and pulled by default, so for the user, again, things just work. For both of those options, a quick way to assess whether your dependencies work with Native Image is the Ready for Native Image page [10]. This is a list of libraries that are known to be continuously tested with Native Image. There are more compatible libraries out there, but this is a good first assessment step.
- If your library of interest doesn't provide any support for GraalVM, but you are using a framework that does (in our case Spring Boot 3.0), you can use the framework support to produce

custom 'hints' for Native Image (Listing 2) and for resources (Listing 3).
- If neither your library nor your framework support GraalVM, you can use the Tracing Agent that comes with Native Image to produce the necessary configuration automatically [11].
- Finally, you can provide/extend configuration for reflection, JNI, resources, serialization, and predefined classes manually in JSON [12].

Some of those steps look rather scary, but if you're starting a new project, most of the frameworks and libraries will just work. The 'Ready for Native Image'-page mentioned above contains almost 200 libraries and frameworks, including Micronaut, Spring, Quarkus, Helidon, H2, GraphQL, MariaDB, Netty, MySQL, Neo4j, PostgreSQL, Testcontainers, Thymeleaf, and many others. There has never been a better time to be a Spring Boot and GraalVM developer!

{ MONITORING 📈 }
I'll briefly mention *monitoring*, too. Many monitoring tools for Java, such as VisualVM, work out of the box with applications produced by Native Image thanks to its compatibility with many Java monitoring features and protocols. They include Java Flight Recorder, hprof-based heap dumps, and jvmstat. In addition, various monitoring frameworks (such as Micrometer) as well as many cloud vendors also offer additional capabilities for monitoring applications built with GraalVM.

{ WHAT'S NEXT FOR GRAALVM? }
What could be more exciting than the future? Let's talk about two big projects we are currently working on.

Native Image Layers
We want to introduce a brand new way of building and deploying Native Image applications. With *Layers* you'll be able to create a native image that depends on one or more *base images*. This way, every time you recompile your application, you only need to recompile your user code, taking compilation times down to seconds. And the benefits don't end there – once deployed, applications can share the base images, reducing the resources consumption even further. How cool is that!

GraalOS
We are working on a new deployment platform called GraalOS for Java applications, based on GraalVM Native Image. You'll get the same familiar benefits of Native Image – fast startup, low memory usage, high peak performance – and in addition GraalOS will take care of scalability, isolation, leveraging the latest hardware features, and reducing costs. My favorite is 'deploy applications, not containers' – we want to eliminate the overhead and additional efforts associated with containerization. Exciting times ahead!

L2
```
@RegisterReflectionForBinding(Clazz.class)
// will register the annotated element for
reflection
```

L3
```
public void registerHints(RuntimeHints hints,
ClassLoader classLoader) {
    hints.resources().registerResource(new
ClassPathResource("myresource.xml"));
}
// will register a resource
```

{ CONCLUSION }
I hope this article encourages you to try Spring Boot and GraalVM. Together they will give you an easy way to build fast and light-weight applications with access to anything you might need from the rich Spring ecosystem.

To keep learning about GraalVM, check out the resources I linked [13, 14, 15]. And if you have any questions or feedback, just ping me on social media. Now go build something with GraalVM! ‹

{ REFERENCES }

1 https://ionutbalosin.com/2024/02/jvm-performance-comparison-for-jdk-21
2 https://start.spring.io
3 https://sdkman.io
4 https://github.com/rakyll/hey
5 https://medium.com/graalvm/graalvm-for-jdk-21-is-here-ee01177dd12d#0df7
6 twitter.com/YunaMorgenstern/status/1729039787351536084
7 https://github.com/h2database/h2database/blob/master/h2/src/main/META-INF/native-image/reflect-config.json
8 https://github.com/oracle/graalvm-reachability-metadata
9 https://github.com/graalvm/native-build-tools
10 https://www.graalvm.org/native-image/libraries-and-frameworks
11 https://www.graalvm.org/latest/reference-manual/native-image/metadata/AutomaticMetadataCollection
12 https://www.graalvm.org/latest/reference-manual/native-image/metadata/#specifying-metadata-with-json
13 https://www.graalvm.org
14 https://github.com/graalvm/graalvm-demos
15 https://alina-yur.github.io/graalvm-resources

Netcompany

Problem solvers

We use the full technology stack to develop smarter, more connected, and sustainable societies.

Join our talented people and create IT solutions that are fit for the future.

Apply today.

www.netcompany.com

Explore here.

KOTLIN 2.0, HARDER BETTER FASTER STRONGER

It's been long awaited, and the bird has landed! Kotlin 2.0 was officially released last May during KotlinConf. The work has been going on for a while, with the first announcements of the K2 compiler I could find in late 2021 [1]. In this article, I won't focus on all the nitty gritty details of the new version, but bits and pieces that you may not have heard of and I find interesting from a developer perspective. Strap up and let's go! 🚀

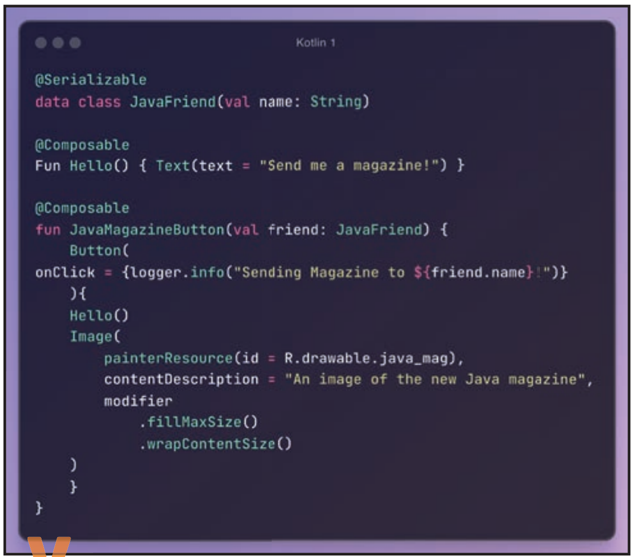
Julien Lengrand-Lambert
is a Lead Developer Advocate and a Kotlin Google Developer Expert.

{ THE NEW K2 COMPILER! }

The K2 compiler is by far the most noticeable, if not to say one of the reasons for the Kotlin 2.0 release. This new compiler brings many different things at once, so let's dive into them one after the other.

{ SPEED }

Compiling projects with K2 should be anywhere between 1.5 to 2 times faster than with the older compiler for all projects.
You can read more yourself on the official website [2], but here are some numbers based on real life projects:
The K2 compiler brings up to 94% compilation speed gains. For example, in the Anki-Android project, clean build times were reduced from 57.7 seconds in Kotlin 1.9.23 to 29.7 seconds in Kotlin 2.0.0. Similarly, for the same projects the initialization phase is up to 488% faster with the K2 compiler. The Kotlin K2 compiler is up to 376% quicker in the analysis phase compared to the previous compiler.



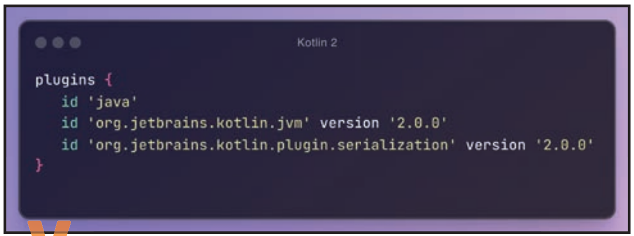
```
Kotlin 1

@Serializable
data class JavaFriend(val name: String)

@Composable
Fun Hello() { Text(text = "Send me a magazine!") }

@Composable
fun JavaMagazineButton(val friend: JavaFriend) {
    Button(
onClick = {logger.info("Sending Magazine to ${friend.name}!")}
    ){
    Hello()
    Image(
        painterResource(id = R.drawable.java_mag),
        contentDescription = "An image of the new Java magazine",
        modifier
            .fillMaxSize()
            .wrapContentSize()
    )
    }
}
```

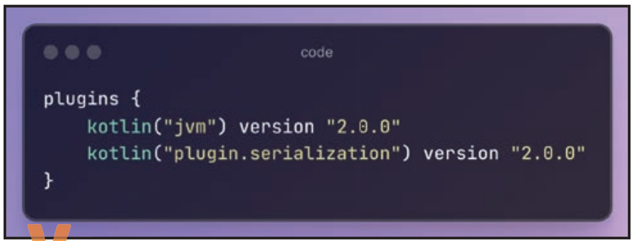
Image 1.



```
Kotlin 2

plugins {
    id 'java'
    id 'org.jetbrains.kotlin.jvm' version '2.0.0'
    id 'org.jetbrains.kotlin.plugin.serialization' version '2.0.0'
}
```

Image 2.



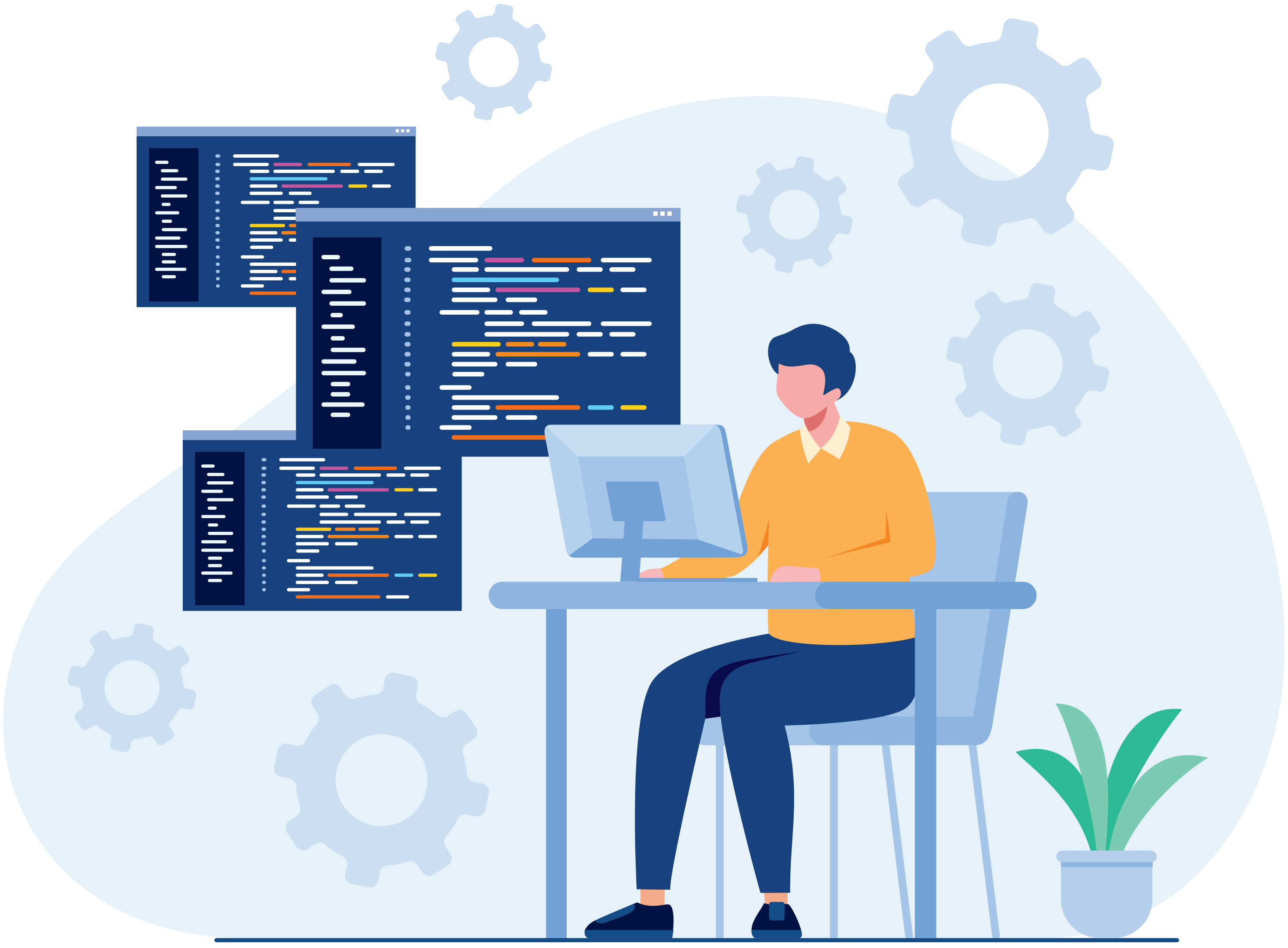
```
code

plugins {
    kotlin("jvm") version "2.0.0"
    kotlin("plugin.serialization") version "2.0.0"
}
```

Image 3.

{ ONE PLATFORM TO RULE THEM ALL }

One of the most awaited features of K2 is the fact that it is built with multi-platform in mind from the ground up.
You should be aware by now that Kotlin Multiplatform (KMP) was announced a few years ago, with the objective of unifying all platform application code (iOS, Android, Desktop, Web, Server,



...) under a single umbrella. This is now becoming reality with Kotlin 2.0 (though with different levels of maturity for the different platforms 🙂).
This also means that as of 2.0, the Jetpack Compose [3] compiler used to create multi-platform applications ships together with Kotlin.

{ A QUICK LOOK AT JETPACK COMPOSE }

We're not going to dive into the details here, but one of the promises with KMP is that part of your UI code can be reused across platforms. If you define a button once, it should look like a native button on Android, iOS, Desktop and Web with minimal changes. At the moment, compose for iOS is in Beta, and Web in Alpha. But some serious projects are already built with KMP, one of the best examples being the Jetbrains Toolbox.
A lot of the UI concepts are similar to what you may already know from React or Vue, focussed on stateless, isolated, composable (It's even in the name, isn't it 😀) components interacting with each other via events.Image 1 shows how a simple custom button can look in Compose.

{ THE K2 COMPILER FOR YOU }

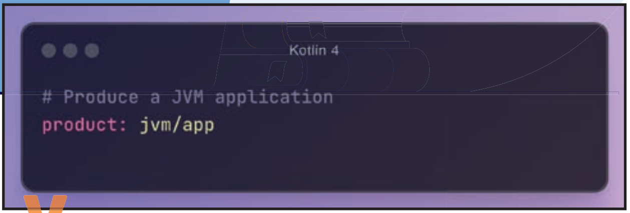
To upgrade the Kotlin version, change it to 2.0.0 in your Gradle and Maven build scripts. To do this, you can change your gradle.build file, see image 2.
This is how it would look for a groovy based Gradle file for a server project, but remember the Kotlin DSL is now the default in Kotlin image 3 shows how it _should_ look 😀.
Note that the Kotlin Gradle plugin makes use of a new *.kotlin* folder to store its data. So you probably want to add it to your *.gitignore* file.
As of Kotlin 2.0, it's important to note that *IntelliJ* itself still disables support by default; meaning that it's giving information based on Kotlin 1.x. This is to reduce any friction for developers as the compiler is rolled out. That being said, you can actively Enable the K2 Kotlin mode in the settings if you want!

{ OTHER (UN)RELATED GOODIES }

Quite a few things were announced in the vicinity of Kotlin 2.0 and are only partly related. Still, I find them interesting especially because most of them are focusing on improving our Developer Experience.

{ PROJECT AMPER }

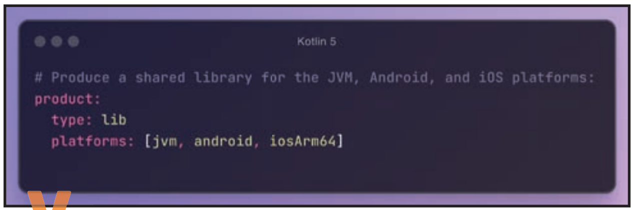
I'm pretty sure I heard most of you sigh when I mentioned Gradle earlier. For good reasons or not, most of us have a strong opinion of Gradle and putting the DSL in Kotlin does not necessarily help. My personal gripe is that with 2 DSL languages now, and a new default it's hard to find the documentation for the flavor you want...
Luckily, JetBrains came out with a young, but promising project: Amper [4]. Amper describes itself as a 'project configuration and build tool'. It's a drop-in replacement to Maven and Gradle, but not only. For example, a fully functional build for a server-side Kotlin project looks like image 4:
A shared library for multiple platforms can look like image 5.
Just like K2, Amper was built with multi-platform in mind. It supports multi-module, multi-platform, Gradle interop and more in mind. It will take a while to see if usage grows for this new tool, but I find it refreshing to have a new alternative coming out.



```
Kotlin 4

# Produce a JVM application
product: jvm/app
```

Image 4.



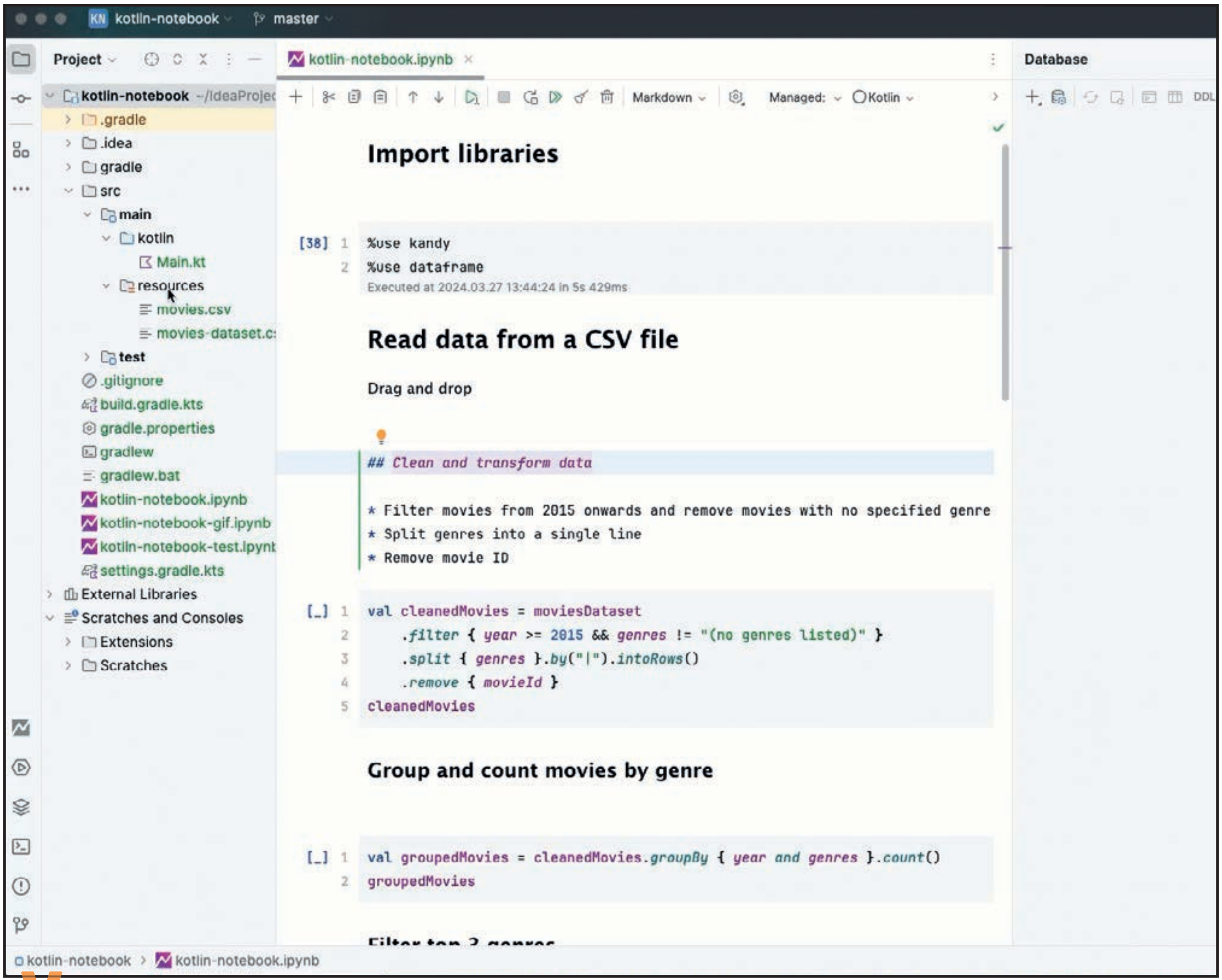
```
Kotlin 5

# Produce a shared library for the JVM, Android, and iOS platforms:
product:
  type: lib
  platforms: [jvm, android, iosArm64]
```

Image 5.

{ KOTLIN NOTEBOOKS }

I'm sure you're familiar with Python Notebooks, those semi-scripts, semi-demo, always enjoyable sheets that unroll as you run the steps. They're super useful in education, to think and manipulate data visually, or to demonstrate a concept. They can easily be shared, documented, and more.
Kotlin Notebooks came out last year in alpha, and are now generally available. They bring all of the nice scripting goodies from the Notebooks, but without the dependance to Python, image 6.
This is nice, especially with the massive push towards AI related applications lately. Python may rule in the data domain, but a lot of



Import libraries

```
[38]
%use kandy
%use dataframe
```

Executed at 2024.03.27 13:44:24 in 5s 429ms

Read data from a CSV file

Drag and drop

```
## Clean and transform data

* Filter movies from 2015 onwards and remove movies with no specified genre
* Split genres into a single line
* Remove movie ID
```

```
[_]
val cleanedMovies = moviesDataset
    .filter { year >= 2015 && genres != "(no genres listed)" }
    .split { genres }.by("|").intoRows()
    .remove { movieId }
cleanedMovies
```

Group and count movies by genre

```
[_]
val groupedMovies = cleanedMovies.groupBy { year and genres }.count()
groupedMovies
```

Filter top 3 genres

Image 6.

applications actually run on Java/JVM and making the bridge easier to cross allows to bring the best of both worlds in one package. You can try out Kotlin Notebooks by installing the IntelliJ plugin [5].

{ KOTLIN FOUNDATION }

One of the things that always made me a little bit nervous about Kotlin was that it was basically only created and supported by JetBrains. Last year, the Kotlin foundation was announced with Google as major sponsor, followed by other companies like Gradle and Shopify. Uber joined the group as well in sync with the release of Kotlin 2.0.

That's super important to ensure the future of the language, but also a healthy and vibrant ecosystem (especially a multi-platform ecosystem). The Kotlin foundation, among other things, offers grants for projects that contribute to improving the ecosystem. Have a look over here [6]!

In this article, I haven't spent much time on the actual new Kotlin features. Instead, I've tried to share what makes me excited about the future of Kotlin: I see a growing ecosystem, with more capabilities, support from a larger array of meaningful companies, and most importantly continuous improvements in the overall Developer Experience. It's really just the beginning though, there's so much more to discover! I hope this gave you a few ideas of things to try out in the near future. <

{ REFERENCES }

1 https://blog.jetbrains.com/kotlin/2021/10/the-road-to-the-k2-compiler/
2 https://kotlinlang.org/docs/k2-compiler-migration-guide.html#performance-improvements
3 https://en.wikipedia.org/wiki/Jetpack_Compose
4 https://github.com/JetBrains/amper
5 https://kotlinlang.org/docs/kotlin-notebook-overview.html
6 https://kotlinfoundation.org/news/

NLJUG INNOVATION AWARD 2024

JAVA DEVELOPERS MAKEN HET VERSCHIL
IN INNOVATIEVE TECHNOLOGISCHE ONTWIKKELINGEN

Is jouw bedrijf een innovator en heb jij een succesvol, innovatief, verantwoord en/of omvangrijk project? Dan is dit jouw kans om in de schijnwerpers te staan! De NLJUG Innovation Award wordt voor de zesde keer uitgereikt tijdens J-Fall en we zijn op zoek naar het meest innovatieve project. Elke NLJUG businesspartner dat een project inschiet, krijgt tevens 2 J-Fall conferentie-tickets! Maak kans op een timeslot binnen het programma van J-Fall en inschrijving door je project voor maandag 16 september 2024 in te dienen!

Winnaars

2018 Yolt

2019 Tennet

2021 Picnic

2022 Omoda

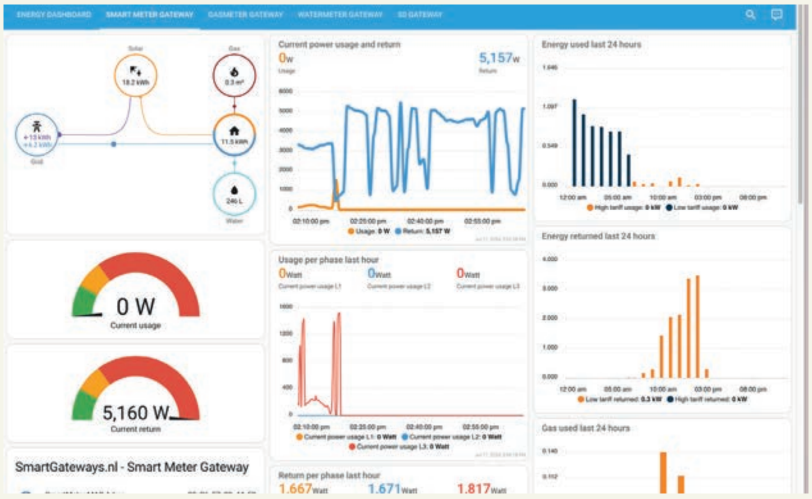
2023 Triopsys

INSCHRIJVING

Vanaf nu tot en met 16 september 2024 kan een project en/of persoon worden voorgedragen. Ken jij een bedrijf, project en/of persoon die dit jaar de NLJUG Innovation Award zou moeten winnen?

Schrijf je in op nljug.org/nljug-innovation-award/

BYTE SIZE



ENERGY DASHBOARD
SMART METER GATEWAY
GASMETER GATEWAY
WATERMETER GATEWAY
Current power usage and return
Energy used last 24 hours
Energy returned last 24 hours
Usage per phase last hour
Return per phase last hour
Gas used last 24 hours
0 W
Current usage
5,160 W
Current return
SmartGateways.nl - Smart Meter Gateway

{ A COMPLETE ENERGY DASHBOARD FOR YOUR HOME IN HOME ASSISTANT }
We all have a smart meter these days, but did you know that with some simple IOT you could have access to your electricity, water and central heating systems? Smart-Gateways specializes in creating those small IOT devices and make them compatible with Home Assistant. A good way to start playing with MQTT at home!

{ DATA-ORIENTED PROGRAMMING IN JAVA }
If you pick the right combination of recently added Java language features, you get something that's much more than the sum of its parts: data-oriented programming. This paradigm was first described for Java by Brian Goetz in June 2022 and after two years, it's time for a revised version 1.1! **https://inside.java/2024/05/23/dop-v1-1-introduction/**

{ MOBILE DEVICE BEST PRACTICES }
Have you tried turning it off and on again? The NSA mentions turning your phone off and on again once a week as an effective mobile device best practice. I'm sure we can find time for that! **https://s3.documentcloud.org/documents/21018353/nsa-mobile-device-best-practices.pdf**

HOW TO WRITE A BYTE SIZE' BYTE SIZE

Do you have a tip to share, an opinion, or simply want to share something short? That's what the byte size format is for! Byte sizes are short bits of information, in a Twitter format: you have roughly 280 characters, and space enough for a link, or a picture! Send us your byte sizes by @JavaMagazineNL on Twitter or by mailing **info@nljug.org**!

> Java Magazine Editorial Committee

See you next year

J·SPRING 2024

POWERED BY

.nl.
jug

MAIN SPONSOR

CHILIT

CO-SPONSORS

ABN·AMRO

Rabobank

PARTNERS

42

EDSN

nedap

PICNIC

profit4cloud

Rijksoverheid

TEQNATION

CODE | INNOVATE | CREATE

Save the date
May 15th 2025

See you next year

Gold sponsors:

ABN·AMRO   nationale nederlanden   Rijksoverheid

Sponsors:

luminis Conversing worlds   ORDINA   Netcompany   PIC NIC   Rabobank   RITD

SEARCH MORE SUSTAINABLY WITH ECOSIA

One of the reasons why I love creating software is because it can make the world a better place. It can solve problems, make people more productive and automate the boring stuff. And I strongly believe that it can contribute to climate action.

Hanno Embregts is an IT Consultant at Info Support with a passion for learning, teaching and making music. He has recently been named a Java Champion.

In fact, I feel we have a responsibility here, because in 2022 our industry as a whole accounted for over 2 percent of global energy demand [1] (and it's rapidly rising [2]). Since software developers are known for their problem-solving skills, let's start solving this problem, too. Which doesn't necessarily have to be hard. Simply switching search engines can be a very good start!

{ SEARCH ENGINE EMISSIONS }

As you may already know, each search query you perform emits CO2. It's hard to get a precise number, but estimates range from 0.2 to 10 grams emitted per search query [2]. That doesn't sound too bad, but given the massive internet user base this quickly adds up, as the data visualization 'CO2GLE' demonstrates in a rather dramatic way [3]. On top of that, an AI-assisted Google search uses 10 times more energy than a regular Google search [4]. So in this day and age of AI, the potential to reduce is large and growing rapidly.

I'm not saying the mainstream search engines are bad, per se. Most of them have promised to become carbon neutral, but timelines on these goals tend to be fluid. On top of that, most tech giants achieve carbon neutrality by buying cheap (and controversial) carbon offsets, instead of making their own processes more sustainable [5]. When it comes to sustainability in search engines, better options are available.

{ THE SEARCH ENGINE THAT PLANTS TREES }

Such as Ecosia! [6] Like any search engine, Ecosia earns money through advertising. But here's the big difference: Ecosia has pledged to spend 100% of their profits to climate action, through tree planting and supporting nature restoration projects [7].
Ecosia produces twice as much renewable energy than they need to power a search, feeding the excess clean energy back into the grid. This crowds out electricity derived from coal and other fossil fuels. On top of that Ecosia plants trees that capture carbon, which means that, all things considered, each Ecosia search actually *removes* 1kg of CO2 from the air, making Ecosia a *carbon-negative* product [8]. So just by simply *using* Ecosia for your daily searches, you can contribute to climate action!

{ SEARCH RESULTS }

Which is great, but are the search results any good? Well, they are delivered by both Bing and Google [9]. In my experience, 95% of what you're searching for can be found quickly. For the remaining 5% it is easy enough to switch to a different search engine like Google by using the 'More' menu or to use the Google shortcut in your search query (#g). More shortcuts exist; the full list is in Ecosia's help documentation [10].

{ PRIVACY }

Ecosia takes user privacy seriously and is committed to protecting your data. They do not sell your information to third parties or track your searches. Ecosia anonymizes all searches within a week, ensuring your privacy is maintained. This means that Ecosia is handling privacy a lot better than Google (though probably not as good as DuckDuckGo).

{ MONEY AT ECOSIA }

Ecosia is transparent about its finances and tree-planting activities: they publish monthly financial reports on their blog to disclose total monthly revenue and what percentage was used to plant trees [11]. For example, in May 2024, the search engine generated over €2.8 million, of which almost 26% was used to finance over 995,000 trees.

{ TREE PLANTING }

Ecosia's tree planting is not about sticking random seeds in the ground and calling it a day. They plant diverse forests (not sterile monocultures), preferring native and endangered species to imported and invasive ones. They plant in biodiversity hotspots that used to be forests, in over 30 countries. And they also monitor the trees with satellite tech and field visits for up to 20 years, while collaborating with local communities who make sure the trees will still be around to benefit their grandchildren.
According to Ecosia, it takes about 45 searches to fund the planting of a single new tree. This number could be reduced dramatically depending on other factors, like whether you click on an advert and how "valuable" the search term is in terms of advertisement payout. So if you want Ecosia to make the most impact, you could consider disabling your ad blocker on ecosia.org, so that you may also discover (and navigate to) sponsored results.

{ CONCLUSION AND OTHER WAYS TO CONTRIBUTE }

Switching to Ecosia is easy enough; most popular browsers already allow you to set it as the default search engine. And if Ecosia isn't your cup of tea, other environmentally-active search engines like OceanHero [12] or OCG [13] might be a better fit for you. Or if you prefer to keep your current search engine, you could try to make an impact somewhere else. For example, the Green Software Foundation [14], which exists to build an ecosystem of people, standards, tools and best practices for building green software, is looking for contributors. Or you could consider switching to a more sustainable cloud provider, like Leafcloud [15]. Because *where* you focus your climate efforts doesn't really matter - as long as you're aware that your actions *matter* and can make a difference. <

{ REFERENCES }

1 https://news.mit.edu/2022/how-can-we-reduce-carbon-footprint-global-computing-0428
2 https://www.theregister.com/2024/07/02/google_datacenter_emissions
3 https://www.janavirgin.com/CO2
4 https://doi.org/10.1016/j.joule.2023.09.004
5 https://www.bloomberg.com/news/articles/2024-07-08/google-is-no-longer-claiming-to-be-carbon-neutral
6 https://ecosia.org
7 https://blog.ecosia.org/manifesto
8 https://blog.ecosia.org/regeneration
9 https://ecosia.helpscoutdocs.com/article/505-new-search-providers
10 https://ecosia.helpscoutdocs.com/article/22-shortcuts
11 https://blog.ecosia.org/ecosia-financial-reports-tree-planting-receipts
12 https://oceanhero.today
13 https://ocg.org
14 https://greensoftware.foundation
15 https://www.leaf.cloud

TEQnation 2024

Last May, NN visited and sponsored the TEQnation conference. During the day, more than 500 visitors gathered at the historic 'de Fabrique' in Utrecht to experience the software development conference. The event was a great success, with a team of enthusiastic NN colleagues representing the company at the NN booth. Romera Hillebrandt and Nikki Doorhof, both Data Scientist at NN's Customer & Digital team, represented NN on stage where they spoke about how they implemented ChatGPT technology on a large scale. A topic that was well received by the audience.

NN's tech talk about ChatGPT
The tech talks at TEQnation covered a wide range of topics, from artificial intelligence and machine learning to cybersecurity and blockchain. Nikki and Romera, gave a tech talk about ChatGPT and the successful implementation of Automated Call Logging (ACL) within the company. NN is one of the leading companies in Europe to have implemented ChatGPT on a large scale. During the talk, they spoke about how they implemented ChatGPT for ACL, the challenges they faced, how ChatGPT increased efficiency and improved customer experience, and, how they got the business on board.

***Nikki**: 'it was really exciting to inspire other engineers with what we've achieved by integrating ChatGPT into our Automated Call Logging project. They were especially impressed by the fact that we have been doing this for a year already, while others are still in the trial phase. I'm really proud that we as NN are one of the early adopters of this technology.'*
***Eddy Vos**, Engineering Manager at NN, says: 'NN surprised the TEQnation visitors with the fact that we use ChatGPT in production on a large scale. After a great talk on stage about this by Nikki and Romera, we had great interaction at our booth about the topic!'*

NN game built by NN's developers
At the NN booth, visitors were able to play a 'word guessing' game, built by NN's developers to showcase the technical skills of our development team. Participants had to guess tech-words in 90 seconds; the five fastest players won a prize at the end of the day. The game was a great way to connect with visitors at the NN booth. Bob Verheij, Full Stack Developer at NN, says: 'it was a lot of fun to see the participants struggle, lost for tech-words, trying to solve the game. During the day, people were piling up to attempt to win.' It was a great way to connect with fellow engineers.

Overall, TEQnation turned out to be a great conference where developers, engineers, and tech enthusiasts found each other, shared ideas, and learned from each other. We are looking forward to next years' conference!

IT jobs at NN
If you're passionate about working with innovative technology solutions like ChatGPT, we invite you to explore the IT career opportunities at NN. We are always looking for talented IT professionals to join our team and help us deliver socially-responsible digital products that our customers love. Visit our **NN career website** today to learn more about the latest job openings.

TERUGBLIK J-SPRING 2024

Wafels, ijs, game AI's en Hot Sauce!

Prikkelt jouw tong nog na van onze Hot Sauce J-Spring 2024? De capsaïcine-resistente tongen van CHILIT in ieder geval niet. Wij kijken vooral terug op een enerverend en geslaagd J-Spring 2024. Even het geheugen prikkelen, wat hebben we ook alweer gedaan?

Keynote: Java-performance issues

Voor het tweede jaar op rij verzorgde CHILIT de Keynote van J-Spring. Dit jaar over de Java-performance issues die niet met een pilletje op te lossen zijn. Vanuit een eigen ervaring boorde Lennart een niet-alledaags probleem aan waarvoor kennis over de achterliggende Java-processen toch best wel handig is.

De tech-talk ging dieper in op het analyseren van mogelijke performance issues. We keken inhoudelijk naar hoe je van tevoren kunt testen hoeveel load je applicatie kan verwerken, en hoe je performanceproblemen vroegtijdig kunt herkennen en voorkomen.

Voedsel voor je brein

Naast een leergierig developer brein, wilden we ook het "tweede brein van het menselijk lichaam", oftewel de maag, genoeg voldoening bieden. Zowel wafels als ijs kon men bij de CHILIT stand vinden. Voor de brutalen was er zelfs de combinatie tussen deze twee te vinden. De echte Java-developers met een ijzeren maag gingen zelfs voor de combinatie tussen de wafel en onze signature Hot Sauce. We zijn heel benieuwd hoe die is bevallen!

Hot Sauce challenge

Onze eigen magen werden ook op de proef gesteld. Keer op keer gingen we weer tot de top in onze eigen Hot Sauce challenge. Keer op keer kregen we weer 135.600 eenheden van de Scoville-scale voor de kiezen. En elke keer met een hele goede reden: Een pittig, maar bovenal goed gesprek met jullie. We ondernemen graag ludieke activiteiten, maar het allerleukste en meest voldoenende onderdeel van J-Spring is een goed gesprek!

Leuke en mooie gesprekken

Onze hoogtepunten zijn dan ook die gesprekken: de ervaringen met performance issues waar jullie tegenaan zijn gelopen en hoe jullie hiermee om zijn gegaan, hoe je schaalbaarheidsproblemen op kunt lossen met een microservice architectuur, of natuurlijk een leuk gesprek over de toepasbaarheid van onze Signature Hot Sauce in de keuken. Al deze gesprekken, dat is het allerleukste onderdeel van J-Spring voor ons.

We willen iedereen ontzettend bedanken voor alle leuke en mooie gesprekken, het samen trotseren van de hete sauzen en aparte wafel-saus combinaties. Wij kijken alweer uit naar J-Fall 2024!

CHILIT

Meer weten over CHILIT? Scan de QR!

Psst, ben je nu al door je flesje Signature Hot Sauce heen? Stuur ons vooral een berichtje over LinkedIn. Wij zullen zorgen dat er een nieuwe jouw kant op komt ;)

Hackathons: where ideas become reality
Make it your job
ing.nl/careers/tech
do your thing

UNLEASHING THE POWER OF MAVEN ARCHETYPES

Creating new projects from scratch is a common task in today's world. Often, this involves manually setting up folders and files or using tools like Spring Initializr, Micronaut Launch, Eclipse Starter for Jakarta EE, Quarkus Starter, or IntelliJ to streamline the initial setup. But what if there was a way to automate and standardise these tasks, taking your project setup to the next level?

Giovanni van der Schelde is an IT consultant and trainer at Info Support who enjoys coding, public speaking and going outdoors.

While these tools are excellent for generating a basic 'Hello World' application, they often fall short when it comes to adding project-specific configurations and components that are standard within your organisation. As a result, you might find yourself spending time in a tedious and error-prone process such as copy-pasting parts of similar applications, renaming files, and replacing variables.

{ INTRODUCTION }

For those unfamiliar with Maven, it is one of the most widely used project management tools in the Java ecosystem. Maven manages a project's build, reporting, and documentation from a central configuration. It operates through plugins that hook into its lifecycle phases, although some plugin goals can be used standalone.

One of those plugins is the Maven Archetype Plugin, which you might have encountered when following the official Getting Started Guide. Officially supported by the Maven Project [1], this plugin allows developers to create and use archetypes. Archetypes are original patterns or models from which other projects of the same kind are made. It is an abstract representation of a kind of project that can be instantiated into a concrete customised project.

Using the Maven Archetype Plugin, developers can quickly and consistently generate new projects. This not only boosts productivity but also ensures that the generated components adhere to best practices, guidelines, and standards within your organisation, while eliminating repetitive and tedious tasks.

{ HOW MAVEN ARCHETYPES WORK }

Before diving into creating an archetype, it's important to understand how they work. Like any other dependency, archetypes are typically hosted in remote or local repositories, using the same configuration as dependencies for resolution. If an archetype is not managed by a repository, you must add an additional repository entry with the ID archetype in the settings.xml file [2].

The Maven Archetype Plugin discovers available archetypes through a catalog file. This catalog stores information about archetypes, including their GAV (GroupId, ArtifactId, and Version), the repository they are hosted in, and their description. If the repository is omitted, the same repository as the catalog is used to locate the archetype.

The plugin comes bundled with several predefined archetypes, which can be displayed using the command: `mvn archetype:-generate -DarchetypeCatalog=internal`.
This command uses the internal catalog. Besides the internal catalog, there is also a local catalog found in the home directory at *~/.m2/archetype-catalog.xml*, and a remote catalog accessible at *https://repo.maven.apache.org/maven2/archetype-catalog.xml*. They can be specifically targeted using remote or local as the value of the **archetypeCatalog** parameter.

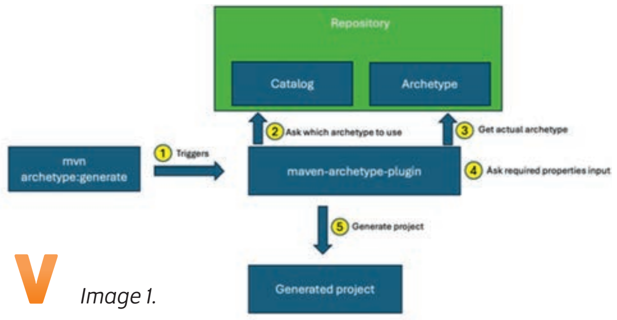


Image 1.

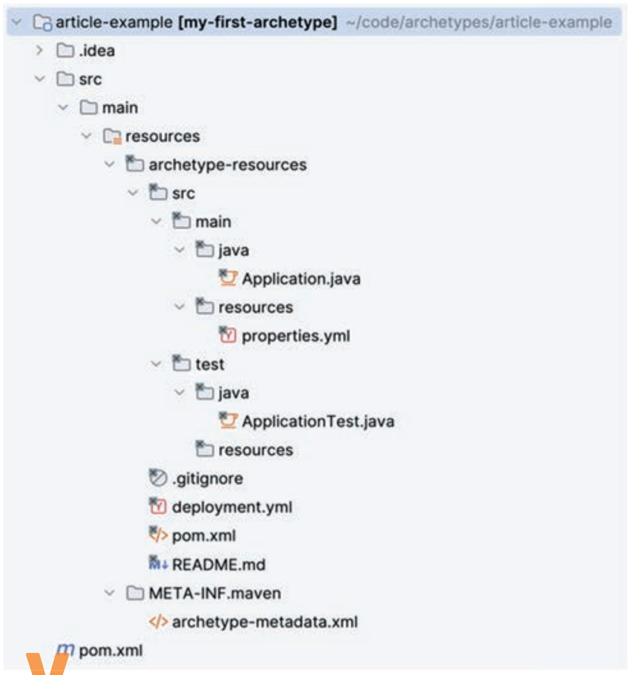
Archetypes are packaged in a JAR file that contains a descriptor. This descriptor includes metadata about the archetype, such as required properties for project generation, possible default values, and the files to include and generate (Image 1).

Generating a project from an archetype can be done interactively or non-interactively using additional command-line flags. The process involves selecting an archetype, entering additional configuration parameters or user input, and generating the project. The templating engine used, Velocity, expands the properties in the files and generates the dynamic content on the fly.

{ CREATING AN ARCHETYPE }

Maven Archetypes can be created from scratch or by using the `archetype:create-from-project` goal, which generates an archetype from an existing project but includes all files. This requires manual filtering of unwanted files afterwards. Alternatively, the `maven-archetype-archetype`, which is an archetype to generate a simple archetype, generates most of the setup but is outdated. In this example, we will build an archetype from scratch, though the same concepts apply to both methods.

Creating an archetype project involves a different setup compared to most Maven projects. Let's assume the current working directory is *~/archetype*. The process begins with creating a new *pom.xml* file and the *src/main/ folders*. Next, place the archetype descriptor in the *src/main/resources/META-INF/archetype-metadata.xml* file. The files that are copied and dynamically filled are placed in the *~/archetype/src/main/resources/archetype-resources* folder. These archetype resources could be build scripts, YAML pipelines, Git ignores, Java classes, and any other standardised files for your organisation. Image 2 shows this archetype setup. Start by creating a new Maven project with a POM similar to the one in listing 1.



```
article-example [my-first-archetype] ~/code/archetypes/article-example
  .idea
  src
    main
      resources
        archetype-resources
          src
            main
              java
                Application.java
              resources
                properties.yml
            test
              java
                ApplicationTest.java
              resources
          .gitignore
          deployment.yml
          pom.xml
          README.md
        META-INF.maven
          archetype-metadata.xml
  pom.xml
```

Image 2.

L1

```
<project xmlns="http://maven.apache.org/
POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.giovds</groupId>
 <artifactId>my-first-archetype</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>maven-archetype</packaging>

 <build>
   <pluginManagement>
     <plugins>
       <plugin>
         <groupId>org.apache.maven.plugins</
groupId>
         <artifactId>maven-archetype-plugin</
artifactId>
         <version>3.2.1</version>
       </plugin>
     </plugins>
   </pluginManagement>

   <extensions>
     <extension>
       <groupId>org.apache.maven.archetype</
groupId>
       <artifactId>archetype-packaging</
artifactId>
       <version>3.2.1</version>
     </extension>
   </extensions>
 </build>
</project>
```

Notice the use of a different packaging type. The archetype-packaging extension defines custom lifecycle bindings [3], allowing Maven to execute the appropriate plugin goals for this new packaging type.

The **archetype-metadata.xml** also known as the descriptor, has two attributes. The name describes the display name when the user selects the archetype. Partial indicates whether the archetype represents an entire project or only a part, such as a specific module or selection of classes. It also contains properties required to generate the archetype. The properties will be passed through the plugin and replaced by Velocity during project generation. The example in listing 2 shows some configuration options..

L2

```
<archetype-descriptor
       xmlns="http://maven.apache.org/
plugins/maven-archetype-plugin/archetype-
descriptor/1.1.0"
       xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.
org/plugins/maven-archetype-plugin/archetype-
descriptor/1.1.0 https://maven.apache.org/xsd/
archetype-descriptor-1.1.0.xsd"
       name="first-archetype"
       partial="false">
   <requiredProperties>
       <requiredProperty key="service-name">
           <defaultValue>my-first-service</
defaultValue>
       </requiredProperty>
<requiredProperty key="team-name">
<validationRegex>^[A-Z].*$</validationRegex>
</requiredProperty>
   </requiredProperties>

   <fileSets>
       <fileSet encoding="UTF-8" filtered="true"
packaged="true">
           <directory>src/main/java</directory>
       </fileSet>
       <fileSet encoding="UTF-8" filtered="true"
packaged="false">
           <directory/>
           <includes>
               <include>**/*.yml</include>
           </includes>
<excludes>
               <exclude>secret.config</exclude>
           </excludes>
       </fileSet>
       <fileSet encoding="UTF-8"
filtered="false" packaged="false">
           <directory/>
           <includes>
               <include>README.md</include>
           </includes>
       </fileSet>
   </fileSets>
</archetype-descriptor>
```

The descriptor also configures which file sets to include and how to include them. The filtered attribute controls whether the files

L3

```
package ${groupId};

import org.springframework.boot.autoconfigure.
SpringBootApplication;
import org.springframework.boot.autoconfigure.
domain.EntityScan;
import org.springframework.boot.
SpringApplication;

@SpringBootApplication
@EntityScan(basePackageClasses = ${service-name}
Application.class)
public class ${service-name}Application {

   public static void main(final String[] args)
{
       SpringApplication.run(${service-name}
Application.class, args);
   }

}
```

within this set contain properties that need to be replaced by Velocity. If set to false, the files are copied as-is. The packaged attribute places the files in a packaged structure, meaning files within this file set will be placed in folders resembling a package-like structure. For example, if a project was generated with a **groupId** of **com.giovds** and the fileset directory to include was **configuration**, the files within would be placed in */my-service/configuration/com/giovds*. This is mainly used for class files. The non-packaged files will be placed in the root folder */my-service*.

To install and use the archetype, the archetype's JAR must be deployed to your local repository using `mvn install`. Then, generate a new project by running `mvn archetype:generate -DarchetypeCatalog=local`. The command line flag tells the plugin we are only interested in locally installed archetypes, to make it easier to find the newly installed one.

{ PROPERTY INTERPOLATION WITH VELOCITY }

Velocity templates in Maven Archetypes allow for dynamic content generation by utilising properties and custom logic.
For basic property replacement, Velocity properties use the same format as Maven properties, denoted as `${property}`. Properties defined in the `requiredProperties` element of the archetype descriptor can be accessed this way. Additionally, `GAV` values can also be used. For example, in a template **POM** file within the **archetype-resources** folder, these properties can automatically fill the groupId, artifactId, and version. However, when replacing properties in filenames, the property name is surrounded by two

underscores on each side. For example, if there is a property service-name and a file needs to include this property in its name, it could be named `__service-name__Application.java`. Within this file, other properties would use the standard `${property}` format, as shown in Listing 3.

In some cases, it's useful to derive values from other properties. Velocity Template Language (VTL) statements allow for defining new variables that can be processed by the engine.
For example, consider a project that uses two different usernames for database access and migrations, following a naming convention like `name_db_user` and `name_migration_user`. If the archetype has a property named database-username filled by the user with a value like `contract_db_user`, a VTL statement can be used to derive the migration username instead of defining a separate property. The statement could split the value by an underscore and select the first and last parts to determine the migration username, listing 4.
This is one of the many examples of how VTL statements can be used to manipulate these properties. More information and advanced usages can be found in the Velocity User Guide [4].

{ KEEPING ARCHETYPES UP TO DATE }

Software evolves over time, and so do the decisions and requirements that shape it. Consequently, archetypes need to be updated to reflect these changes. Manually checking if an archetype still works can be tedious. Fortunately, the plugin provides tools to automate this process.

The integration-test goal of the plugin can be utilised to automate the testing of archetypes. This goal is bound to the Maven integration test phase by default. It generates projects from the current archetype and optionally compares these generated projects to a reference copy.

Each test is represented as a sub-directory in **src/test/resources/projects**. This directory should contain an **archetype.properties** file. This file contains the properties used to generate the project, just like a user would via the command line.
Optionally, it can contain a **goals.txt** to specify one or more goals to run against the generated project. And an optional **reference/** directory containing a copy of the expected project structure and content for comparison.

Since an archetype project is a Maven project itself, and possibly contains other projects that define dependencies, it can benefit from automated dependency management tools to stay up to date. Tools like Renovate and Dependabot can automate the process of updating the archetype project and its dependencies.
In combination with the test suite the archetypes require minimal maintenance and will stay updated over time without unknowingly breaking.

L4

```
#set ($split = $database-username.split("_"))
#set ($splitLastIndex = $usernameSplit.size() -
1)
     username: ${split[0]}_
migration_${split[$splitLastIndex]}
```

{ FINAL THOUGHTS AND MY EXPERIENCE WITH ARCHETYPES }

I have successfully introduced archetypes in two organisations, mostly receiving great feedback from the teams. They are actively used by the developers, enabling them to start development quickly without the hassle of copy-pasting projects, making unnoticed renaming errors, and recreating all the necessary application and deployment configurations.

However, before enthusiastically introducing archetypes to your team, consider the following questions:

- How often do you need to create new services or partial components?
- Do you have the necessary automation in place to test and update the archetypes?
- How often do your project requirements change?

Creating and maintaining archetypes involves a learning curve, especially when working with Velocity templates. If the frequency of creating new components is low, the initial effort and continuous maintenance might not be justified. Without automated pipelines for updates and tests, the maintenance effort might outweigh the benefits. Additionally, frequent changes can impact the usefulness of archetypes since they are not backward compatible and cannot be re-applied to existing projects.

Ready to enhance your development workflow with Maven Archetypes? Start by assessing your team's needs, set up automated pipelines if you haven't already, and dive into the creation of your first archetype. The benefits of a more efficient and error-free setup process are just a few steps away. Don't wait—begin your journey with Maven Archetypes today and empower your teams to focus on what they do best: writing great code. <

{ REFERENCES }

1 https://maven.apache.org/plugins/
2 https://maven.apache.org/archetype/maven-archetype-plugin/archetype-repository.html
3 https://github.com/apache/maven-archetype/blob/master/archetype-packaging/src/main/resources/META-INF/plexus/components.xml
4 https://velocity.apache.org/engine/devel/user-guide.html

AI IN JAVA WITH LANGCHAIN4J

Ever since the launch of OpenAI's ChatGPT at the end of 2022, Artificial Intelligence (AI) has been in a fast moving current, with a new groundbreaking AI tool joining the scene seemingly every week. So far we have seen AI coding assistants, at least one high quality video generator, and multiple audio/voice generators, with probably plenty more AI tools on their way. But what about AI as part of our Java systems? There's a lot of development going on, like Intellij's AI Assistant or Spring AI. In this article I am going to highlight *LangChain4j*, a framework capable of harnessing the power of AI in your Java systems. It will showcase how you can get started with the basics of applying Large Language Models (LLM) in Java.

Michael Koers is a software consultant at Info Support, with most of his knowledge in Java and Azure. He enjoys sharing his knowledge, teaching others and gets excited about home automation.

{ INTRODUCTION }

A lot of companies have started integrating AI into their core business. The result is a vast array of products or services that are now 'powered by AI'. This could potentially lead to groundbreaking results, like greatly improving diagnostic speed and accuracy in the healthcare sector [1]. In a few other cases this has had less successful results, like that one time an AI customer service chatbot 'sold' a brand new car for 1 dollar [2].

How does LangChain4j come into play? LangChain4j is a library that helps integrate AI into your Java system. It brings two major advantages:

- Unified API's: No longer do you need to worry about proprietary API's from different *Large Language Model* (LLM) providers or embedding/vector stores. With LangChain4j, you can easily switch between providers and stores without the need to rewrite your codebase. From their own GitHub: "Think of it as a Hibernate, but for LLMs and embedding stores." [3].
- Comprehensive Toolbox: LangChain4j offers a bunch of different tools for you to use to your heart's desire, from simple prompt-templates to fully fledged Retrieval-Augmented Generation (RAG) [4].

{ GETTING STARTED }

LangChain4j makes it extremely easy to start integrating AI into your Java application. All you need to do are the three following steps:

- Add the LangChain4j and LangChain4j OpenAI dependency to your project.
- Get an OpenAI API key. LangChain4j provides a free key on their GitHub page for testing purposes [3]. You can also get a paid OpenAI key via OpenAI's website, the minimum is 5 dollars, but this 5 dollars will last you a while during testing
- Create an instance of an OpenAI model and you can already start chatting! See Listing 1 to see how easy it is.

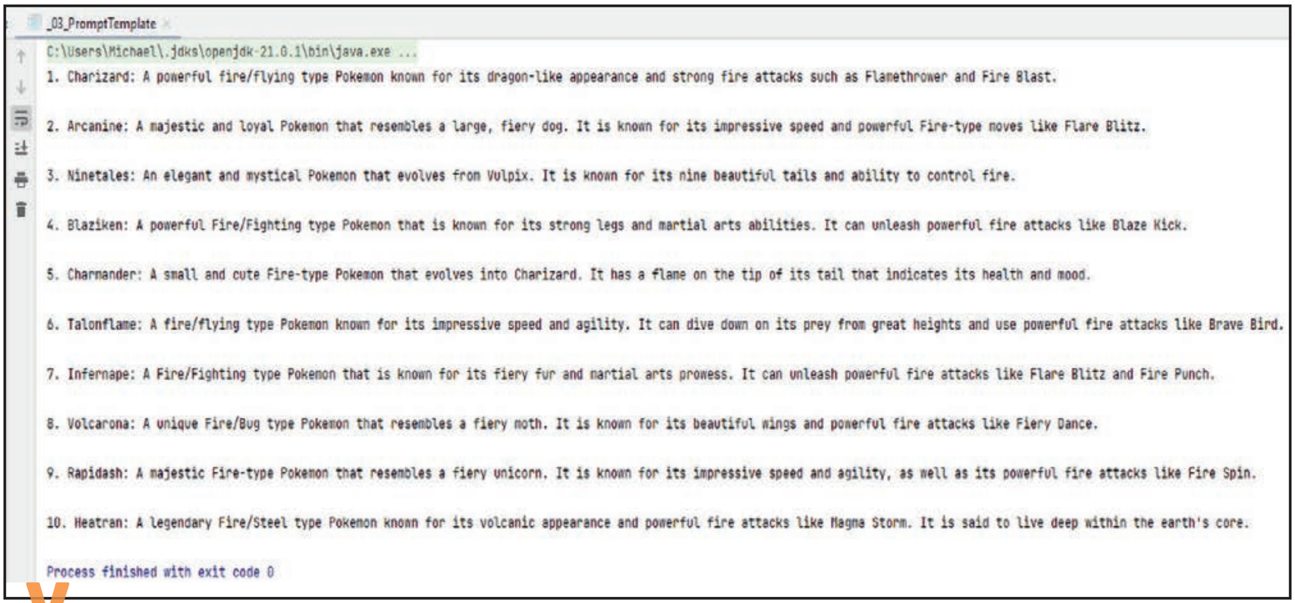
And that's it. You now have a way of interacting with OpenAI's ChatGPT in your Java application. This is just the basics however, there are plenty more options you can change or tweak to get the model just the way you want it. Listing 2 shows the creation of an OpenAI

L1

```
ChatLanguageModel model = OpenAiChatModel.
withApiKey(yourApiKey);

String answer = model.generate("Hello World!
Greetings from NLJug!");

System.out.println(answer); // Hello NLJug! It's
great to connect with you. How are you doing
today?
```



```
_03_PromptTemplate
C:\Users\Michael\.jdks\openjdk-21.0.1\bin\java.exe ...
1. Charizard: A powerful fire/flying type Pokemon known for its dragon-like appearance and strong fire attacks such as Flamethrower and Fire Blast.

2. Arcanine: A majestic and loyal Pokemon that resembles a large, fiery dog. It is known for its impressive speed and powerful Fire-type moves like Flare Blitz.

3. Ninetales: An elegant and mystical Pokemon that evolves from Vulpix. It is known for its nine beautiful tails and ability to control fire.

4. Blaziken: A powerful Fire/Fighting type Pokemon that is known for its strong legs and martial arts abilities. It can unleash powerful fire attacks like Blaze Kick.

5. Charmander: A small and cute Fire-type Pokemon that evolves into Charizard. It has a flame on the tip of its tail that indicates its health and mood.

6. Talonflame: A fire/flying type Pokemon known for its impressive speed and agility. It can dive down on its prey from great heights and use powerful fire attacks like Brave Bird.

7. Infernape: A Fire/Fighting type Pokemon that is known for its fiery fur and martial arts prowess. It can unleash powerful fire attacks like Flare Blitz and Fire Punch.

8. Volcarona: A unique Fire/Bug type Pokemon that resembles a fiery moth. It is known for its beautiful wings and powerful fire attacks like Fiery Dance.

9. Rapidash: A majestic Fire-type Pokemon that resembles a fiery unicorn. It is known for its impressive speed and agility, as well as its powerful fire attacks like Fire Spin.

10. Heatran: A legendary Fire/Steel type Pokemon known for its volcanic appearance and powerful fire attacks like Magma Storm. It is said to live deep within the earth's core.

Process finished with exit code 0
```

Image 1.

instance with the ChatGPT 4 model. Some interesting settings are the maximum number of tokens used by the model, or the temperature which tells the model how much it can 'improvise'.

You also have the option to change the base url used to communicate with the model. Why is this interesting? Because it means you can point your instance to a different LLM provider, for example one self-hosted with Ollama, meaning you can use LangChain4j with a custom or privately hosted model, which is great if privacy is a concern for you or if you need to enrich the model with your own data.

Going forward, it's very important to note that the quality of the Large Language Model used heavily influences the quality of the results. For example, using a model for chatting which was originally trained for programming tasks will perform worse than using a general chat model. So pick a model or LLM-provider accordingly. In this article, I use either OpenAI's ChatGPT 3.5 or 4, as this is a solid general chat bot.

{ TEMPLATING }

Let's start with some simple things: *templating*. LangChain4j offers the capability of templating in two different fashions: PromptTemplates and StructuredPrompts.

PromptTemplates are rather basic: you define a prompt template with some variables in it, and at some other point in your system, you apply values to those variables and send the resulting prompt to your model. This means your system or users only need to fill specific variables, without the need to construct an entire prompt. Listing 3 uses a template to ask AI for a list of 10 fire-type Pokémon. In Image 1 you can see the result of the execution.

L2

```
ChatLanguageModel openAI = OpenAiChatModel.
builder()
        .apiKey(yourApiKey)
        .modelName(OpenAiChatModelName.GPT_4)
        .temperature(0.3d)
        .maxTokens(50)
        .logRequests(true)
        .logResponses(true)
        .maxRetries(3)
        .baseUrl("https://some.other.url.com")
        .build();
```

L3

```
PromptTemplate promptTemplate =
        PromptTemplate.from("Give me an overview
of
{{amount}} pokemon of type {{type}}");

Map<String, Object> values = Map.of(
        "amount", 10,
        "type", "fire"
);

Prompt prompt = promptTemplate.apply(values);

String answer = model.generate(prompt.
toString());

System.out.println(answer);
```

L4

```
@StructuredPrompt({
   """
   Create a {{language}} article about
{{subject}}. Include all of the following
keywords: {{keywords}}
   The article is a max of 100 words.
   Make sure the article exists out of the
following parts:
   - Title
   - Introduction
   - Main body
   - Conclusion
   """
})
class CreateArticlePrompt {
   String language;
   String subject;
   List<String> keywords;

   public CreateArticlePrompt(String language,
String subject, List<String> keywords) {
       this.language = language;
       this.subject = subject;
       this.keywords = keywords;
   }
}
```

StructuredPrompts take templating to a higher level. With the `@StructuredPrompt` annotation you turn a Java class into the prompt and container of data for your prompt, every instance of that class basically becomes a prompt on its own. Listing 4 showcases how to create a structured prompt class for generating articles (I promise this was not used for this article 🙂).

Now you can create as many instances of the CreateArticlePrompt as you want, and all you need to do is specify the programming language, the subject and keywords you want to include. And just like that, you can use AI to generate a whole heap of articles. Listing 5 shows us using the StructuredPrompt to create a Java-article about Artificial Intelligence. From this point, you can tinker with the settings of the model or the prompt to try and achieve the desired result from AI.

{ MEMORY AND PERSISTENCE }

The 'Getting Started' section showed how to create a basic instance of the OpenAI model ChatGPT. It's so basic, it doesn't hold any memory or persist any kind of information. If you ask it a follow up question in the same session, it will have no knowledge of your original prompt or the answer given, the context is gone. Luckily, LangChain4j provides a few different ways in which you can play with memory and/or persistence.

L5

```
CreateArticlePrompt createArticlePromptPromt =
new CreateArticlePrompt("Java"
       , "Artificial Intelligence"
       , List.of("Mind-bending", "future",
"skynet", "booming"));

Prompt prompt = StructuredPromptProcessor.
toPrompt(createArticlePromptPromt);

String answer = model.generate(prompt.
toString());

System.out.println(answer); //Title: The Mind-
Bending Future of Artificial Intelligence: Is
Skynet Booming? ...
```

L6

```
OpenAiChatModel model = OpenAiChatModel.
withApiKey(yourApiKey);

ConversationalChain chain = ConversationalChain.
builder()
       .chatLanguageModel(model)
       .build();

String answer = chain.execute("Hello World! My
name is NLJug!");
System.out.println(answer); // Nice to meet you,
NLJug! How can I assist you today?

answer = chain.execute("Remind me of my name
please");
System.out.println(answer); // Your name is
NLJug.
```

THE FIRST WAY YOU CAN SET UP A MODEL THAT RETAINS MEMORY IS TO USE THE CONVERSATIONAL CHAIN OBJECT. IT WRAPS AROUND A CHAT LANGUAGE MODEL AND A CHAT MEMORY

The first way you can set up a model that retains memory is to use the Conversational Chain object. It wraps around a chat language model and a chat memory. If no memory object is provided, it will default to an in-memory chat memory. Listing 6 shows how you can create and use the Conversational Chain to interact with AI and have it keep its context between prompts. This would not have been possible with the example in Listing 1 as the model alone does not hold any memory or context.

L7

```
List<ChatMessage> chatMemory = new
ArrayList<>();

// Provide LLM with background information
SystemMessage systemMessage = SystemMessage.from(
"""
   We work for a company renowned for its Java
knowledge and experts. Our company only works
with Java and Java-based tools and frameworks,
no other programming languages.
   You are an advisor for the developers of this
company. Answer in max 2 sentences.
   """
);

chatMemory.add(systemMessage);

UserMessage userMessage = UserMessage.from("""
   Which framework should I use for one of our
back-end API's?          ASP.NET Core Web API,
Django, Spring Boot or Rocket?
""");

chatMemory.add(userMessage);

Response<AiMessage> answer = model.
generate(chatMemory);
System.out.println(answer.content().text()); //
I recommend using Spring Boot for developing
your back-end API, as it is a widely used and
well-supported Java-based framework known for
its ease of use and productivity.
```

The second way to retain memory is the `.generate()` method. You've seen it used in examples, but it not only accepts a user prompt, it also accepts a list of chat messages which are sent to the model to process. One thing you can do with this is set a system message. A system message is basically a message sent to the model written by the developers of the system, to instruct the model in one way or another. You can tell the model to keep answers short, to always respond in a certain demeanour or provide it with some information. Listing 7 shows an example using the system message.

Lastly, being able to send a list of chat messages to the model also enables you to do *Few-Shot learning* with the model. Few-Shot learning is the practice of including a few examples in the prompt for the model to learn from. Listing 8 showcases Few-Shot to teach the model it needs to translate the given animal to Dutch. Without giving any instructions aside from the examples, the model knows what to do and successfully translates the word 'Bird' to 'Vogel'.

L8

```
// Set up few-shot, providing example User and
AI message's for the model to learn from
List<ChatMessage> fewShotHistory = new
ArrayList<>();
fewShotHistory.add(UserMessage.from("Cat"));
fewShotHistory.add(AiMessage.from("Kat"));
fewShotHistory.add(UserMessage.from("Dog"));
fewShotHistory.add(AiMessage.from("Hond"));
fewShotHistory.add(UserMessage.from("Horse"));
fewShotHistory.add(AiMessage.from("Paard"));

// And now for an actual question
fewShotHistory.add(UserMessage.from("Bird"));

Response<AiMessage> answer = model.
generate(fewShotHistory);
System.out.println(answer.content().text()); //
"Vogel"
```

{ THIS IS JUST THE START... }

Langchain4j offers much more than showcased in this article. There is also AIServices, (Dynamic) Tools and support for RAG. These tools are much more powerful than sending simple prompts to a model. They enable you to integrate AI and make it a part of your Java system, truly honing in on the 'powered by AI' phrase.

Interested in more? Go watch Lize Raes's Devoxx talk about LangChain4j [5]. It covers everything that was discussed in this article and more, but in greater detail. This is also the talk that first introduced me to LangChain4j, and it is well worth your time if this topic interests you.

To get started yourself you can go to the LangChain4j GitHub page where they provide you with all the information necessary to get started in minutes [3]. All the examples shown in this article can be found on my GitHub [6]. <

{ REFERENCES }

1 https://bepartofresearch.nihr.ac.uk/articles/artificial-intelligence/
2 https://www.upworthy.com/prankster-tricks-a-gm-dealership-chatbot-to-sell-him-a-76000-chevy-tahoe-for-1-rp
3 https://github.com/langchain4j/langchain4j
4 https://research.ibm.com/blog/retrieval-augmented-generation-RAG
5 https://www.youtube.com/watch?v=BD1MSLbs9KE
6 https://github.com/Michael-Koers/Langchain4JDemo

HET SCHAAP MET EEN SUPERHELDENCAPE!

Ben jij een blij schaap dat braaf binnen het werkhekje staat en elke dag zijn kleine taakjes uitvoert? De meeste van ons zijn dit tot op zekere hoogte, en de meeste van ons hebben werk waarbij we het grootste deel van de tijd taken van het bedrijf uitvoeren. Onder de streep is dit prima en voor onszelf én de maatschappij goed dat we als kudde samenwerken en ons daarbij een beetje laten aansturen.

V

Maja Reißner *is Docent cyber security.*

Als ik de vraag verder uitsplits in kleinere vragen, zien mijn antwoorden er wel iets anders uit. Ben ik blij? Doorgaans wel. Vanuit mezelf ben ik wel vaker onnozel blij, over het algemeen vrij tevreden maar ik heb ook een hang naar het negatieve, wat ik zelf vooral als kritisch denken ervaar.

Ben ik een schaap dat als brave werknemer de taken doet die mij worden opgelegd? Ja, echt wel. Ik heb wel veel vrijheden in hoe ik mijn taken invulling geef en natuurlijk heb ik mijn werkgever zelf gekozen maar ik ben wel taken aan het uitvoeren voor de baas en soms voel ik me daarbij ook net een schaap tussen al mijn collega-schapen.

En ben ik daar blij mee? Ja en nee. Ik denk dat het goed, nuttig en rustgevend is om stabiel basiswerk te hebben waarbij je als een schaap een leidinggevende volgt en taken uitvoert. Ik denk alleen ook dat dit niet het enige moet zijn wat je als mens doet. Je mag best meer zijn dan een schaap. Sowieso in je hele leven maar ook specifiek in je werkleven.

Studies laten zien dat een duidelijke zingeving significant bijdraagt aan ons welzijn (zie bijvoorbeeld Frankl 1985 of Debats, van der Lubbe & Wezeman 1993). Ik denk dat er problemen zijn die passen bij jouw waarden en waar je voldoening uit haalt als je eraan werkt. Problemen die aansluiten bij jouw expertise en vakmanschap. Zo'n probleem ga je proberen op te lossen.

En hoe doe je dat precies? Ik volg hiervoor het Feynman problem-solving algoritme:

1) *write down the problem;*
2) *think very hard;*
3) *write down the answer.*

Klinkt zo simpel. Dit algoritme kent dan ook veel kritiek omdat deze drie zinnen op zichzelf staand niet voldoende zijn. Maar het concept klopt. Vind ik ;).

Je begint met het probleem. Als je nog geen goed probleem hebt gedefinieerd ga je nog geen oplossing verzinnen. Dus juist niet alleen werken en taken afvinken maar doortastend naar een probleem zoeken dat nog niet opgelost is. Een probleem waarvan je hebt achterhaald waarom jij het wil oplossen. WAAROM je het wil oplossen én waarom JIJ het wilt/kunt oplossen. Het "waarom" is je motivatie, gekoppeld aan je waarden en aan je zingeving. En voor je eigen gemak wil je natuurlijk een probleem dat enigszins in je domein van competenties en vaardigheden ligt zodat je kans maakt voor dit probleem een oplossing te vinden.

De afgelopen jaren zag je bijvoorbeeld IT'ers die meer en meer in richting *sustainability* gingen onderzoeken. Een onderwerp dat waarschijnlijk dicht bij hun waarden ligt en waar hun expertise in de IT kan helpen om het probleem op te lossen.

Het probleem definiëren is veel werk. Voordat we naar stap twee van Feynmans aanpak gaan, wil ik dan ook nog een keer stilstaan bij "veel werk" en de bewustwording dat veel werk hier niet een teken van je eigen falen is maar dat veel werk gewoon veel werk kan zijn. Veel werk kan betekenen dat je hier maandenlang mee bezig bent. Niet maandenlang 8 uur per dag en ook niet maandenlang uitstellen maar wel een doorlooptijd van maanden en regelmatig bij het probleem terugkomen. Je staat stil bij je waarden, denkt na over mogelijke problemen en doet daar onderzoek naar. Je gaat sparren met vrienden (die waarschijnlijk je waarden delen), met collega's (die je expertise delen). En je maakt notities, valideert en rectificeert. Je splitst stap 1 van Feynmans algoritme eigenlijk uit en past de eerste *diamond* van het *double diamond model* van *design thinking* toe: probleem benoemen, verbreden en ontdekken en weer herdefiniëren.

Ja, wat een bevalling! Even relaxen en heel erg trots op jezelf zijn dat je het probleem scherp hebt. Je bent nu klaar voor stap 2: *think very hard*. Deze stap krijgt doorgaans de meeste kritiek omdat de drie woorden zo vaag zijn dat je ook hier heel goed kunt procrastineren of opgeven. Feynman was een enorme uitzondering in zijn mentale vaardigheden en ik doe het hem zeker niet na. Ik kan met mijn hersenen niet wat hij kon. Maar wat ik hem wel kan nadoen: Ik kan zijn aanpak proberen te begrijpen en de manier van denken na proberen te doen.

Want Feynman kneep niet alleen even zijn ogen hard dicht, dacht hard na en was klaar. Hij omschrijft in zijn boek '*Surely You're Joking, Mr. Feynman!*' hoe hij een mentaal model van een probleem maakt en dit in zijn achterhoofd sluimert als hij met andere dingen bezig is. Hij weet duidelijk waarom hij het probleem wil oplossen, draagt het altijd bij zich en kan dan verschillende situaties daaraan relateren en zo een nieuwe kijk op het probleem en mogelijke oplossingen krijgen.

Hierin ligt denk ik nog een extra verborgen kracht, de kracht van verschillende emoties tijdens het denkproces. En waarom denk ik dat? Van psychologische onderzoeken weten we dat er een relatie tussen positieve emoties en de automatische denkmodus bestaat. Kahneman noemt deze denkmodus '*system 1*', het wat dierachtige brein dat snel, automatisch en intuïtief is. Deze denkmodus is dan ook belangrijk voor creatieve gedachten en cognitieve flexibiliteit.

Daarnaast bestaat '*system 2*', de mens specifieke, analytische denkmodus die traag is. Dit systeem gebruiken we bij wiskunde en het hard nadenken. Positieve emoties kunnen deze denkmodus verstoren zodat we weer terugvallen in de intuïtieve, creatieve, niet doortastende modus 1.

Om een echt probleem goed te definiëren maar ook om het later goed te kunnen oplossen hebben we analytische vaardigheden nodig. We willen focussen en kritisch naar het probleem kunnen kijken. Hiervoor is een neutrale of wat negatievere mentale staat nuttig.

Tegelijkertijd zoeken we naar een oplossing die nog niet bestaat maar waar we verschillende bouwstenen op een nieuwe manier met elkaar kunnen combineren. En dat is juist een creatief, blij proces. De kunst is dan om deze vrolijke, positieve, intuïtieve ingeving niet blij langs je heen te laten gaan maar er meteen weer mee terug te keren naar je analytische manier van werken. Maak bijvoorbeeld direct een notitie van je ingeving. Hoe vaak heb je niet al van het notitieblokje in de douche gehoord?

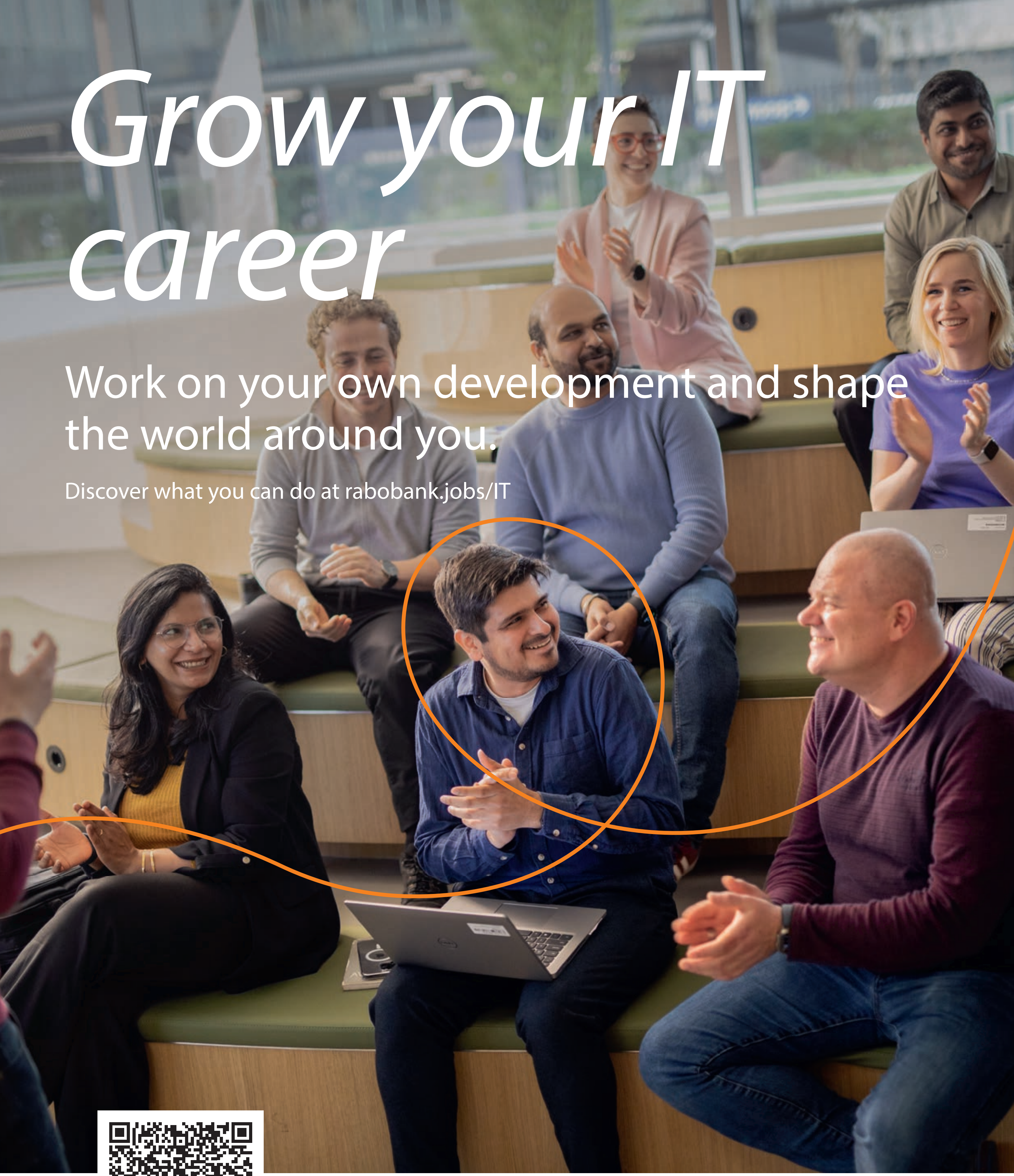
Ik denk dat Feynman onderliggend met '*think very hard*' dus niet alleen bedoelde om heel hard te fronsen en in '*system 2*' analytisch rondjes te draaien, maar juist systematisch af te wisselen tussen de creatieve, losse gedachten (gestimuleerd door activiteiten zoals samen muziek maken en lol hebben) en het analytische brein waardoor nieuwe ideeën op een probleem toegepast en getoetst kunnen worden.

Totdat hij uiteindelijk bij stap 3 was gekomen: De oplossing. En natuurlijk, als je de oplossing eenmaal hebt, hoef je hem alleen op te schrijven of in ons geval misschien eerder uit te programmeren. Dit is natuurlijk te kort door de bocht. In stap 3 weet je waarschijnlijk pas wat voor een project je nu wil gaan bouwen. Als engineers hebben we het geluk dat we ook echt iets kunnen maken en niet alleen een formule verzinnen die iemand anders weer gaat gebruiken.

Naast dat ik denk dat deze aanpak mooi en prettig voor mijn eigen hoofd is, zit er voor mij nog een tweede conclusie in. Ik vind het mooi en belangrijk om te realiseren dat wij neutrale of zelfs beperkt negatieve gevoelens juist kunnen gebruiken om voldoening uit onze analytische vaardigheden te halen als wij ons werk relateren aan onze waarden en onderliggend onze zingeving. Ik ervaar dit als een enorme kracht en wil bij deze al mijn onwijs competente en kritische collega's even extra in het zonnetje zetten! Ik zie helaas vaker dat niet-technici de kritische houding van techneuten afkeuren en in de meeste gevallen ben ik het er niet mee eens. Ik zie juist wat voor bijzondere vaardigheden deze mensen in zich dragen die je moeilijk kunt uitlokken als je maar blij voor je uit blijft staren.

Ik hoop dat je een beetje nieuwsgierig bent geworden naar wat voor een probleem jou zal boeien :). Dat je misschien echt een paar mogelijke problemen uitwerkt, dit met je collega's of vrienden kunt delen en misschien zelfs, alleen of ook samen, een probleem echt aan gaat pakken.

Misschien wordt je dan een superheldenschaap dat een groot gedeelte van zijn tijd wel gewoon schaap is maar daarnaast in kleine stappen de wereld gaat redden. Of het probleem waaraan je werkt is gewoon zo innemend, relevant en belangrijk dat je hiervan je fulltime baan kunt maken. Misschien laat je je schapenvacht achter je en ga je bomen planten zodat de kudde volgende zomer voor de hitte kan schuilen. <


Grow your IT career
Work on your own development and shape the world around you.
Discover what you can do at rabobank.jobs/IT

de coöperatieve Rabobank

SPRING DATA ELASTICSEARCH & ELASTICSEARCH JAVA API CLIENT 8

Wanneer je functionaliteit wil bieden voor het doorzoeken van grote hoeveelheden data en tekst, dan kan het interessant zijn om verder te kijken dan de bekende SQL-database. Een bekende optie is bijvoorbeeld Elasticsearch. Uiteraard wil je deze op een eenvoudige manier integreren in je (Spring Boot) JVM-applicaties.

Erwin de Gier is Software Architect bij Trifork Amsterdam. Daarnaast is hij regelmatig te zien als spreker op diverse softwareconferenties.

E*lasticsearch* is een snelle, gedistribueerde search-engine voor full tekst-search, gebaseerd op Apache Lucene. Spring Data Elasticsearch is onderdeel van het Spring Data-project en biedt een gestandaardiseerd programmeermodel voor integratie met Elasticsearch. Elasticsearch Java API Client 8 is de meest recente Java-library om met Elasticsearch te communiceren vanuit een JVM-applicatie. In dit artikel zal ik beschrijven hoe we deze drie technologieën samen in ons project gebruiken om gebruikers te voorzien van een snelle en veelzijdige zoekoplossing.

We hebben de eis om te kunnen zoeken in tekstvelden, te zoeken en te sorteren op verschillende (geneste) velden en map-achtige velden waar het niet mogelijk is om een simpele *join* naar een andere tabel te maken. We zullen Elasticsearch combineren met een relationele database, die de primaire data-opslag vormt voor lees- en schrijfbewerkingen. We beginnen met het voorbereiden van ons project op het gebruik van Elasticsearch.

{ PROJECT SETUP }

In een bestaand project met de meest recente versie van spring-boot-starter-parent (3.2.x) is het voldoende om spring-boot-starter-data-elasticsearch toe te voegen (listing 1), waarmee de Elasticsearch Java API Client 8 beschikbaar komt. Wij draaien Elasticsearch in docker; Listing 2 bevat de bijbehorende docker-compose-definitie. Standaard start Elasticsearch op zonder authenticatie, maar door de variabele `xpack.security.enabled` op `true` te zetten en de variabele `ELASTIC_PASSWORD` in te stellen kun je authenticatie activeren. Nadat de container is gestart kun je naar **http://localhost:9200** gaan in je browser om te zien of alles correct is opgestart.

De volgende stap is de integratie van Spring met Elasticsearch. Listing 3 laat zien dat we een `ClientConfiguration` nodig hebben met een juist ingestelde hostnaam en poort. Wanneer authenti-

L1
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-
elasticsearch</artifactId>
</dependency>
```

L2
```
elastic:
  image: docker.elastic.co/elasticsearch/
elasticsearch:8.12.0     ports:
    - "9200:9200"
    - "9300:9300"
  environment:
    - xpack.security.enabled=false
    - discovery.type=single-node
  volumes:
    - elastic:/usr/share/elasticsearch/data
```

L3
```
@Configuration
public class ElasticConfig extends
ElasticsearchConfiguration {

  @Override
  public ClientConfiguration
clientConfiguration() {
      return ClientConfiguration.builder()
             .connectedTo("localhost:9200")
         // when Basic authentication is enabled

//.withBasicAuth(username,password)
              .build();
  }
}
```

catie is ingeschakeld, moeten we ook de gebruikersnaam en het wachtwoord opgeven.

Nu we onze infrastructuur gereed hebben, kunnen we gaan kijken hoe we een document kunnen maken en ernaar kunnen zoeken.

{ SPRING DATA REPOSITORY & DOCUMENT }

Spring-Data-Elasticsearch gebruikt de vertrouwde *POJO*-stijl voor het definiëren van *entities*, of in dit geval *documenten*. Je kunt Java-records gebruiken in plaats van classes, wat goed past bij de functie van data holder. Listing 4 laat een voorbeeld zien van een documentdefinitie. Documenten krijgen de annotatie `@Document` en moeten een veld hebben met een @Id-annotatie, welke dient als primary key. Het `indexName`-attribuut kan verwijzen naar een daadwerkelijke index of naar de alias ervan.

Vergelijkbaar met het definiëren van een JPA-entity gebruik je hier annotaties op de velden om het schema te definiëren. Bijvoorbeeld om aan te geven of het veld moet worden geïndexeerd als een keyword of als text. Keyword-velden worden opgeslagen zoals ze zijn (voor exacte matching); text-velden worden opgesplitst en lowercase opgeslagen, voor een wijdere vorm van matching. De fielddata-vlag geeft aan of je het veld wilt gebruiken voor sorteren en aggregaties.

Bij het opstarten van de applicatie zal Spring Data de index en mapping aanmaken op basis van de documentdefinitie. De mapping kan worden beschouwd als een schema voor de index. Het is ook mogelijk om de functie voor het automatisch maken van indexen uit te schakelen en deze handmatig aan te maken (listing 5). Het wordt aangeraden om een index met een alias aan te maken en de aliasnaam in de code te gebruiken, zodat de index opnieuw kan worden aangemaakt onder een andere naam. De alias (bijv. item) verwijst dan naar de nieuwe index (bijv. item-v2) nadat het herbouwen is voltooid.

Nu we een documentmapping hebben, kunnen we een document aan onze index toevoegen (listing 6). We maken hierbij eenvoudig gebruik van de save()-methode op de Spring Data-klasse ItemDocumentRepository. De standaard Spring Data-repository-methoden zijn beschikbaar om de index te doorzoeken, evenals de Query Method-syntax (bijv. `findByName(String name)`), waarmee we onze eigen methoden kunnen maken voor het doorzoeken van specifieke velden. Net als bij Spring Data JPA stelt Spring Data Elastic ons in staat om de `@Query`-annotatie te gebruiken voor het maken van aangepaste query's, die in ons geval een geldige Elasticsearch JSON-query als waarde gebruiken. Placeholders (`?0, ?1`) kunnen gebruikt worden in de query om de methodeparameters te binden.

Voor meer complexe of dynamische query's kunnen we de `NativeQueryBuilder` gebruiken (Listing 7). Dit is een klasse die wordt

L4
```
@Document(indexName = "item")
//@Document(indexName = ITEM_ALIAS_NAME,
createIndex = false) //disable auto //create
index
public record ItemDocument(
    @Id
    @Field(type = FieldType.Keyword)
    String id,

    @Field(type = FieldType.Text, fielddata = true)
    String name,

    @Field(type = FieldType.Text, fielddata = true)
    List<String> groups,

    @Field(type = FieldType.Nested,
includeInParent = true, store = true, name =
"customProperties", fielddata = true)
    List<CustomPropertyDocument>
customProperties,

    @Field(type = FieldType.Date, format =
DateFormat.date_hour_minute_second_millis)
    @JsonFormat(shape = JsonFormat.Shape.STRING,
pattern = DATE_FORMAT)
    LocalDateTime createDateTime
) {
    public static final String DATE_FORMAT =
"yyyy-MM-dd'T'HH:mm:ss.SSS";
}
```

L5
```
    elasticsearchOperations.
indexOps(ItemDocument.class).
putMapping(ItemDocument.class);
```

geleverd met Spring Data Elasticsearch en die Query-instanties als parameters accepteert. De Query-class komt van de Elasticsearch Java API-client. Sinds versie 8 wordt gebruikgemaakt van een meer functionele API voor het opbouwen van de query's; in de volgende paragraaf bekijken we dat verder.

{ ELASTICSEARCH JAVA API CLIENT 8 }

Eén van de belangrijkste redenen waarvoor we Elasticsearch gebruiken is een full text search over alle velden van ons Item-document. We kunnen hiervoor een query string query gebruiken: een flexibele query die het mogelijk maakt in meerdere velden tegelijk te zoeken Daarnaast is het mogelijk gebruik te maken van zowel wildcards (*), als boolean-logica door middel van `AND`- en `OR`-instructies. Listing 8 geeft een voorbeeld van zoeken in alle velden en zoeken in een specifieke lijst van velden met de `Que`

L6

```
public class ItemDocumentRepositoryTest{
  public void testCreateAndFind(){
      itemDocumentRepository.save(new
ItemDocument("id","item document name"));
      itemDocumentRepository.findById("id");
      itemDocumentRepository.findByName("item
document name");
  }
}
public interface ItemDocumentRepository extends
ElasticsearchRepository<ItemDocument, String> {
  Iterable<ItemDocument> findByName(String
name);

  @Query("{\"query_string\": {\"query\":
\"?0\"}}")
  Page<ItemDocument> findCustom(String key,
Pageable pageable);
}
Listing 6

Query query = new NativeQueryBuilder().
withQuery(
      BoolQuery.of(bq -> bq.must(
```

L7

```
    MatchQuery.of(mq -> mq.field("name").
query("item document name"))._toQuery()),
    RangeQuery.of(q -> q.field("editDateTime").
gte(fromDate).lte(toDate))._toQuery()
    )._toQuery()).build();
```

L8

```
//Full text search in alle velden
QueryStringQuery.of(q -> q.query("*" +
itemEntitySearch.criterium() + "*"))._toQuery()
//Zoeken in een lijst van velden
QueryStringQuery.of(qs -> qs.fields(List.of(f.
field)).query( v ))._toQuery()
```

BIJ HET OPSTARTEN VAN DE APPLICATIE ZAL SPRING DATA DE INDEX EN MAPPING AANMAKEN OP BASIS VAN DE DOCUMENTDEFINITIE

L9

```
//Exacte match, bijvoorbeeld boolean velden
TermQuery.of(tq -> tq.field("isTimed").
value(true))._toQuery()

//Tekst search voor een exacte match op
specifieke velden
MatchQuery.of(tq -> tq.field("title").
value("Practice item"))._toQuery()

//Zoeken op datum van 3 dagen geleden tot nu
JsonData from = JsonData.of(LocalDateTime.
now().minusDays(3).format(DateTimeFormatter.
ofPattern(ItemDocument.DATE_FORMAT)));
JsonData to = JsonData.of(LocalDateTime.now().
format(DateTimeFormatter.ofPattern(ItemDocument.
DATE_FORMAT)));
return RangeQuery.of(q ->
q.field("createDateTime").gte(from).lte(to))._
toQuery();
```

Voor het exact matchen van bijvoorbeeld boolean-velden maken we gebruik van de `TermQuery`. Voor het exact matchen van tekst is er de `MatchQuery`. De API van deze query's is vergelijkbaar.

Een interessante query is het zoeken op datum. In Listing 4 hebben we kunnen zien dat het veld `createDateTime` is gedefinieerd als `FieldType.Date` en daarin hebben we door middel van een `Jackson`-annotatie aangegeven welk datumformaat we gebruiken. We kunnen nu gebruikmaken van een RangeQuery. Voor het meegeven van de datum moeten we via een String een JsonData aanmaken. Let hierbij op dat je hetzelfde datumformaat gebruikt als gedefinieerd staat in je documentdefinitie. Vervolgens is het eenvoudig om bij datums 'greater than (equals)' and 'less than (equals)' aan te geven. Al deze voorbeelden staan in Listing 9.

Wanneer we willen kunnen zoeken naar items met bepaalde waarden voor dynamische velden, hebben we *custom properties* nodig. Listing 10 bevat een voorbeeld met twee items: item 1 heeft een custom property `nakijken` en item 2 heeft een custom property `omschrijving`. Beide property's hebben de waarde `handmatig`. Het standaardgedrag van Elasticsearch met bijvoorbeeld een MatchQuery geeft beide items terug als je zoekt op

L10

```
ItemDocument item1 = new
ItemDocument("id1",List.of(new
CustomProperty("nakijken","handmatig")));
ItemDocument item2 = new
ItemDocument("id2",List.of(new
CustomProperty("omschrijving","handmatig")));

//Geeft zowel item1 als item2 terug
BoolQuery.of(bq -> bq.must(
        MatchQuery.of(tq ->
tq.field("customProperties.key").
value("nakijken"))._toQuery(),
        MatchQuery.of(tq ->
tq.field("customProperties.value").
value("handmatig"))._toQuery()
        ));

//Nested query geeft alleen item1 terug
 NestedQuery.of(nq ->
nq.path("customProperties").query(q -> q.bool(bq
-> bq.must(
        MatchQuery.of(mq ->
mq.field("customProperties.key").
query("nakijken"))._toQuery(),
        MatchQuery.of(mq ->
mq.field("customProperties.value").
query("handmatig"))._toQuery()
        ))))._toQuery()
```

"nakijken" en "handmatig". Er wordt namelijk gematched op een customProperty met KEY "nakijken"of met VALUE "handmatig". Als je wilt matchen op zowel KEY "nakijken" en VALUE "handmatig" en dus alleen item 1 wilt vinden, moet je gebruik maken van een `NestedQuery`. In Listing 4 heb je kunnen zien dat we de CustomProperties op ItemDocument hebben gedefinieerd als `FieldType.Nested`. Dit maakt het mogelijk om te zoeken door middel van een NestedQuery.

Naast een *field* en een *query* heeft een NestedQuery ook een *path*; dit is de naam van de collectie van van het nested object in je Document, in dit geval is het "customProperties" in ItemDocument. Verder geven we nog een BooleanQuery mee met als customProperties.key "nakijken" en als customProperties.value "handmatig". Deze NestedQuery geeft alleen resultaten terug met de combinatie van KEY "nakijken" en VALUE "handmatig".

Uiteraard willen we onze resultaten ook kunnen sorteren en gebruik kunnen maken van paginering. Voor het sorteren kun je aan de search-methode van de ElasticsearchClient een `SortOptions`-object meegeven. Voor paginering geef je een vanaf-index

L11

```
//Sorteren en paging
SortOptions sortOptions = SortOptions.of(so ->
so.field(FieldSort.of(f -> f.field(sortBy).
order(SortOrder.Desc))));
elasticsearchClient.search(s -> s.index(ITEM_
ALIAS_NAME)
     .trackTotalHits(TrackHits.of(t ->
t.enabled(true)))
     .query(query)
     .from(page * size).size(size)
     .sort(sortOptions), ItemDocument.class);
```

mee en de grootte van de gevraagde pagina. Door de `trackTotalHits`-parameter op `true` te zetten, krijg je ook het totaal aantal gevonden documenten terug, wat handig kan zijn om terug te geven in een eventuele (REST) API, zie listing 11.

{ CONCLUSIE }

Spring Data Elasticsearch stelt ons in staat om eenvoudig te integreren met Elasticsearch op de bekende Spring Data-manier. Vooral het definiëren van de mapping (het schema) via POJO's maakt dit eenvoudig. Voor het dynamischer opbouwen van zoekquery's is het handig om gebruik te maken van de verschillende classes die de Elasticsearch-client biedt. Deze query's zijn vergelijkbaar met hoe de query's er in JSON uit zouden zien. Dit maakt het mogelijk om via de REST-API van Elasticsearch je query's te testen in JSON-formaat. <

{ REFERENTIES }

1 https://spring.io/projects/spring-data-elasticsearch
2 https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html
3 https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

COLUMN

Schwung, Rucksack, Zettelkasten

***Maja Reißner** is*
Docent cyber security.

If you think 'Fingerspitzengefühl' and 'überhaupt' are some of the most useful German words, wait until you hear about 'Zettelkasten'. No, it's not something you order at a Bavarian beerhall, but it might just be the perfect way to organize information.

The Zettelkasten is a method of note-taking and knowledge management that dates back to the 16th century but only got known with the German sociologist Niklas Luhmann in the 20th century. Luhmann credited his Zettelkasten with helping him publish over 70 books and 400 scholarly articles, so you want to know for sure what it could mean to your productivity.

At its core, a Zettelkasten is a system for capturing, linking, and retrieving knowledge. The key principle of this method is creating atomic notes, where each note captures a single idea or piece of information. These notes are then assigned unique identifiers, making it easy to reference and link to other notes. The beauty of the Zettelkasten lies in its interconnectedness, creating a web of knowledge that mimics the way our brains work. This system transforms your collection of notes into a dynamic, evolving repository of knowledge that can grow and adapt over time.

By organizing information in a clear, interconnected manner, you're more likely to remember it. Additionally, the Zettelkasten stimulates creativity by encouraging the linking of disparate ideas, leading to novel insights and innovative solutions. The system also helps in staying organized, whether you're managing research, planning a project, or keeping track of personal ideas. Be careful though, you can spend an enormous amount of time on your Zettelkasten just duplicating knowledge that you could also easily look up on the internet.

A use-case that I find incredibly powerful is company- and domain-specific knowledge. I use these notes as external and searchable memory. And since I keep this external memory locally, there's no problem with documenting company specifics.

Obsidian is a neat and lightweight tool that many people use for a Zettelkasten. You begin creating atomic notes, capturing individual ideas, concepts or information. Each note should be self-contained and focused on a single topic. The next step is linking notes to create connections between related ideas. For example, a note on the benefits of the *Singleton* pattern in Java could be linked to another note detailing the implementation of the pattern. I personally don't put dev patterns in my notes due to the knowledge duplication but if you learn new patterns you may benefit from putting them into your Zettelkasten as the process of writing down how the pattern works already should improve your understanding of it.

Incorporating a Zettelkasten into your workflow might seem like adopting yet another German word into your vocabulary, but when you're past the initial setup it's just lovely external memory. So, grab those index cards, and start building your personal Zettelkasten today. After all, if it helped Luhmann publish hundreds of works decades ago, imagine what it could do for you now!

VAN HET BESTUUR

Een update vanuit het NLJUG bestuur met dit keer een vooruitblik naar J-Fall en de activiteiten daaromheen.

Het is weer zomer. Op NLJUG-vlak gebeurt in die periode meestal relatief weinig: we zitten tussen J-Spring en J-Fall in, er zijn in de zomer vaak wat minder events en zelfs Java-developers moeten weleens op vakantie ;-).

{ J-FALL CALL FOR PAPERS }

De J-Fall-programmacommissie is op dit moment druk bezig met het beoordelen van de voorstellen die al via de Call for Papers zijn ingediend. Deze sluit op 1 september aan het einde van de dag, dus heb je nog geen voorstel(len) ingediend, doe dat snel!

Via deze weg wil ik de programmacommissie bedanken voor hun waardevolle input en harde werk. Het resultaat daarvan zie je begin november terug in Ede.

{ NLJUG SPEAKER ACADEMY }

Ook dit jaar bieden we vanuit de NLJUG weer kostenloos onze Speaker Academy aan. In 2 avonden word je door ervaren sprekers en leden van de J-Fall-programmacommissie meegenomen in de wereld van spreken op conferenties. We helpen je met het kiezen van onderwerpen, schrijven van proposals voor een Call for Papers, voorbereiden van presentaties en het oefenen van spreken voor een publiek. De eerste Speaker Academy sessie is al geweest, de tweede zal ergens in september plaatsvinden. Wil je nog meedoen? Mail naar **speakers@nljug.org**.

{ J-FALL 2024 }

Ook al zitten we midden in de zomer, de herfst komt eraan! Op 6 en 7 november kun je je weer onderdompelen in de beste presentaties, keynotes en workshops over Java en het Java-ecosysteem. De registratie voor J-Fall gaat binnenkort open. Houd **nljug.org**, je mail en de NLJUG-socials in de gaten voor meer informatie.

{ NLJUG INNOVATION AWARD }

De NLJUG reikt dit jaar voor de zesde keer tijdens J-Fall de NLJUG Innovation Award uit. Deze wordt toegekend aan het meest innovatieve project binnen de NLJUG community.
Een onafhankelijke jury van Java-experts beoordeelt elke voordracht. Het kan daarbij gaan om projecten die succesvol, innovatief, verantwoord en/of omvangrijk zijn. Elk bedrijf dat een project indient krijgt tevens 2 J-Fall-conferentietickets. Meld je project aan via : **https://nljug.org/innovation-award/**.

{ MASTERS OF JAVA }

De Masters of Java staat erom bekend dat het een uitdagende wedstrijd is voor echte Java developers. In deze wedstrijd komt het aan op pure Java skills. Met opdrachten rondom algoritmes, API kennis en bug fixing. Iedereen die meedoet en aanwezig is bij de Masters of Java ontvangt automatisch toegang tot de J-Fall conferentie. Word jij "Master of Java 2024"? Schrijf je in via: **https://nljug.org/masters-of-java-2024/**.

{ TOT SLOT }

We zijn altijd op zoek naar interessante artikelen voor Java Magazine. Heb je een onderwerp waarvan je niet zeker weet of het interessant genoeg is, of kun je hulp gebruiken bij het schrijven of reviewen van het artikel? Geen probleem, de redactiecommissie van Java Magazine helpt je daarbij! Interesse? Neem even contact op via **info@nljug.org**.

groeten,
Bert Jan Schrijver
Bestuurslid NLJUG

FIRST8
CONCLUSION

MASTERS OF JAVA 2024

Funprogging contest voor alle Java Developers

De Masters of Java is een roemruchte "funprogging contest" (gebaseerd op Java SE), die toegankelijk is voor iedere Java ontwikkelaar. In deze wedstrijd wordt niet de API-kennis, maar de echte programmeer-vaardigheid getest.

Er zijn 5 zeer diverse programmeeropdrachten voor de teams. Deze moeten binnen de tijdslimiet worden opgelost. Voor elke seconde die je overhoudt, scoor je een punt. Ook kun je extra punten scoren met bepaalde testen. Het team met de meeste punten wint.

Een team bestaat uit maximaal 2 ontwikkelaars en het team dat aan het einde van de dag de meeste punten heeft, mag zich "Master of Java 2024" noemen. Er zal naast eeuwige roem natuurlijk gestreden worden voor mooie prijzen. De afgelopen jaren is de beslissing steeds gevallen tijdens de laatste opdracht. Het belooft een zinderende wedstrijd te worden!

Schrijf nu in voor Masters of Java

EEN NLJUG EVENT

Wanneer:	**woensdag 6 november**
Tijd:	**13.00u tot 17.30u**
Waar:	**Van der Valk Hotel, Veenendaal**
Kosten:	**Gratis**

Als trotse hoofdsponsor zorgt First8 Conclusion elk jaar voor 5 uitdagende opdrachten en een robuuste game server. Met veel enthousiasme, bevlogenheid en passie werken wij aan deze en andere gave projecten. Iets voor jou? We maken graag kennis met je.

Benieuwd naar eerdere opdrachten?
Check https://github.com/First8/mastersofjava

Aanmelden via:

www.first8.nl

Aanmelden: https://MastersofJava2024.eventbrite.nl

Kom werken voor CHILIT!

Krijg je energie van oplossingen ontwikkelen voor toonaangevende Nederlandse organisaties, uitdaging te zoeken in complexe IT-omgevingen, maar wel bij een club te zitten waar iedereen elkaar kent, de lijntjes kort zijn en iedereen in het bedrijf passie heeft voor programmeren? Scan dan de QR code voor een professionele blind-date.

Gaat een blind-date te snel en wil je eerst even weten wie we zijn? Wij zijn CHILIT, een groep Java developers met hart voor het vak. Wat we doen, doen we goed. We begrijpen elkaar, en we weten dat het belangrijk is om met nieuwe, moderne technieken te werken. Denk aan: cloud-native libraries/frameworks en de nieuwste versies van Java, Spring Boot en Quarkus. Met twee Java developers aan het roer, weten we de juiste opdracht te vinden die bij jouw wensen past. Voor en door developers dus!

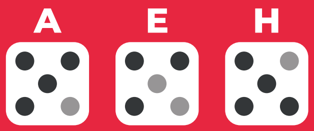
Nu ken je ons wat beter, dus wil je eens verder kennis met ons maken - onder het genot van een kopje koffie of digitaal - dan zien we je graag! Scan de QR en bel ons, mail ons of stuur een postduif (voor instructies zie RFC 1149), dan plannen we wat in.

Kraak onze code en win een Nintendo Switch!

Wij hebben een zin geëncodeerd in onderstaande code. Botvier je programmeerskills door een programma te schrijven waarin je deze code kunt ontcijferen. Onder de inzendingen met een correcte oplossing (zowel code als gedecodeerde zin!) verloten wij een Nintendo Switch.

```
aYGbGPgyhAP80QX8m+hA4Dngapo=
```

We hebben alvast een hint voor je:



Scan de QR om mee te doen!
Zorg dat je inzending binnen is vóór 1 december 2024.
We kijken ernaar uit om je oplossing te ontvangen!

LEGO Pac Man set winnen?

We zijn weer terug met onze AI minigame challenge! (Eigenlijk zijn we ook nooit echt weggeweest.) Nu weer met een nieuwe ronde, dus weer kans om te winnen!

Scan de QR code op deze pagina, bouw je eigen game AI en win een LEGO Pacman set!

HELLO@CHILIT.NL WWW.CHILIT.NL