Nummer 1 - 2019

JAVA MAGAZINE

.nl.jug

Onafhankelijk tijdschrift voor de Java-professional

Exclusief interview met James Gosling

Java-koning: James Gosling

Features

Overstappen van Java 8 naar 11

De wederopstanding van Java EE

Debugging Java in de cloud

J·SPRING nlJUG

Save the date:

May 29th, 2019

TivoliVredenburg Utrecht | The Netherlands

https://jspring.nl

MAIN SPONSOR Rabobank

EXHIBITORS LUNATECH · FIRST8 · OVSOFTWARE · Profit4Cloud · sultansofjava · vijfhart IT-OPLEIDINGEN · ABN·AMRO

CALL FOR PAPERS IS NOW OPEN!
SUBMIT YOUR PAPER: NLJUGCFP.NL

TN
TEQNATION
ENABLING THE FUTURE OF IT

CREATE | INNOVATE | CODE

WRITE IT DOWN:
MAY 15TH, 2019
DeFabrique | Utrecht, the Netherlands

MAIN SPONSOR ABN·AMRO
PARTNER Capgemini

Colofon Java Magazine 01-2019

Content manager:
Stijn van de Blankevoort
Eindredactie:
Anne Camphuijsen-Wieleman
Projectcoördinator:
Tedje van Gils
Auteurs:
Edwin Derks, Kevin Donkers, Hanno Embregts, Rosanne Joosten, Hubert Klein Ikkink, Joop Lanting, Paulo Lopez, Ben Ooms, Bas Passon, Bert Jan Schrijver, Robert Scholte, Koen Vervloesem, Lennart ten Wolde, Ivo Woltring
Redactiecommissie:
Stijn van de Blankevoort, Rob Brinkman, Erwin de Gier, Peter Hendriks, Ben Ooms, Bas Passon, Martin Smelt, Hedzer Westra, Ivo Woltring
Vormgeving:
Wonderworks, Haarlem
Uitgever:
Martin Smelt
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
Marc Post
E-mail: mpost@reshift.nl
Telefoon: 023 543 00 08
Abonnementenadministratie:
Tanja Ekel

Lidmaatschap van de NLJUG kost € 44,50 (België € 45,50) 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, Richard Holkade 8, 2033 PZ 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, Richard Holkade 8, 2033 PZ Haarlem of een e-mail naar **members@nljug.org.nl.**

Voorwoord

"Hoi, ik ben Stijn", J-Fall, James Gosling en nog veel meer

Met trillende vingertjes ben ik dit nu aan het typen, want ik heb nog nooit eerder zelf een stukje mogen schrijven voor een magazine. Maar wie ben ik dan? Hier een kort voorstelrondje: ik ben Stijn van de Blankevoort en ben verantwoordelijk voor de community & content management voor de NLJUG. Dat betekent dat ik in de praktijk de websites van de NLJUG up-to-date houd, nieuwsbrieven maak en meehelp met onze awesome evenementen. Daarnaast ben ik ook alweer een jaar verantwoordelijk voor Java Magazine. Wat houdt dat dan in? Eigenlijk betekent het dat ik samen met de redactiecommissie en de schrijvers een zo mooi mogelijk Java Magazine voor jullie wil maken. Als je meer wilt weten, stuur dan gerust even een berichtje via LinkedIn of mail me op **svdblankevoort@reshift.nl**.

Over tot de orde van de dag! We hebben er met z'n allen een geweldige J-Fall en Masters of Java op zitten, dat weer propvol zat met NLJUG'ers. We hebben zelf als eventteam en NLJUG bestuur een hele mooi dag gehad en te horen aan de reacties hadden jullie het ook goed naar jullie zin. Was je er niet bij en ben je benieuwd hoe het eraan toeging of wil je het J-Fall gevoel graag herbeleven? In deze editie vind je diverse terugblikken om aan je trekken te komen.

Maar we zitten niet stil. Ondertussen komt J-Spring er alweer aan! Heb je 'm al in je agenda staan? Op 29 mei 2019 staat de Tivoli Vredenburg te Utrecht in het teken van Java. Houd ons Twitter kanaal en de website (jspring.nl) goed in de gaten voor alle ins- en outs. Daarnaast vond een paar maanden geleden ook de befaamde CodeOne (voorheen JavaOne) plaats en daar was de NLJUG natuurlijk ook bij aanwezig. Lees verderop wat de NLJUG'ers daarvan vonden.

Genoeg over onze evenementen. Wat kan je nog meer allemaal vinden in deze editie van Java Magazine? Je zag het waarschijnlijk al op de cover: we hebben een exclusief interview met James Gosling te pakken! Dé bedenker en maker van Java. In dit interview vind je de mening van James terug op verschillende oude en nieuwe relevante Java onderwerpen.

Veel leesplezier en prettig met jullie kennis te maken ;-).

Stijn van de Blankevoort
Content-/Communitymanager NLJUG
svdblankevoort@reshift.nl

Where Development meets Passion

Rabobank: The IT Company

Rabobank

Growing a better world together.

10 Debugging Java in de cloud

De tijd dat we applicaties puur monolithisch bouwden hebben we alweer enige tijd achter ons gelaten. Tegenwoordig worden applicaties steeds vaker opgebouwd uit een collectie van kleinere services. Best een mooie ontwikkeling, maar het brengt ook een aantal complicaties met zich mee. In dit artikel belichten we één van de complicaties waar iedereen direct mee te maken krijgt: debuggen.

14 Exclusief interview met James Gosling

James Gosling was op 18 oktober in Nederland. De mensen van Blue4IT was het gelukt om de geestelijke vader van Java voor een bezoek naar Nederland te halen. De NLJUG was daar natuurlijk bij! Hier een kort verslag van de bijeenkomst, met daarbij zijn leukste quotes over Java.

20 CodeOne 2018

Eind oktober 2018 was NLJUG aanwezig bij Oracle CodeOne: de grootste Java-conferentie ter wereld. Of moeten we tegenwoordig zeggen: de grootste Code-conferentie ter wereld? JavaOne heeft namelijk een nieuwe naam gekregen: CodeOne. De nieuwe naam duidt aan dat Oracle niet alleen Java, maar ook andere talen, technologieën en communities omarmt.

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.

Java Native Images

Java without a JVM

When SUN Microsystems released Java 1.0 in 1996, it came with the slogan: "*write once, run everywhere*" to illustrate the cross-platform benefits of the Java language. Ideally, this meant that a Java program could be developed on any device and be expected to run on any device equipped with a Java virtual machine (JVM).

The costs of a Virtual Machine and Just-In-Time compilers are extra computing resources and startup time. To understand what happens when a program runs on the JVM several steps need to happen to take a "`.class`" file and convert it into machine code. The code is loaded into memory and is verified for bytecode errors. The interpreter runs the program, and the JIT compiler follows and compiles the java bytecode on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. To run a single "*Hello World*" many things need to happen before those 11 characters get outputted to the terminal. Consider the code in **listing 1**.

If we count the number of instructions needed to run this application using the Linux perf tools, we would observe the output of **listing 2**.

To print the message "**Hello World**" the Operating System (OS) executed 138,839,929 CPU instructions. For long running applications (e.g., a server) this number becomes negligible, for short-lived processes (e.g., command line applications, serverless functions, embedded devices, IoT) this is a considerable performance penalty.
This problem isn't new, there are several attempts (some with success) to solve it, the two most notorious are: "Excelsior JET" which since 1999 works on a proprietary Ahead of Time Java compiler; and GCJ (a defunct GCC subproject) which retargeted "javac" to emit native code instead of bytecode.

More recently (September 2016) Oracle has released to the Open Source OpenJDK project, the tool "jaotc" an Ahead-of-Time (AOT) compiler for Java9 based on GraalVM Open Source project (also sponsored by Oracle).

GraalVM

GraalVM is an extension of the Java virtual machine to support more languages and execution modes. The Graal project includes a new high-performance Java compiler; itself called Graal, which can replace Hotspot's VM default JIT (C2), or in an AOT configuration for the Substrate VM. To understand what Substrate VM (SVM) is about, here is the quote from the project's GitHub readme file:

Paulo Lopzez is a Principal Software Engineer at RedHat and Open Source developer on the Eclipse Vert.x project.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Listing 1

```
$ perf stat -e instructions java Hello
Hello World

 Performance counter stats for 'java Hello':

       138,839,929      instructions:u

       0.045113933 seconds time elapsed

       0.041752000 seconds user
       0.007316000 seconds sys
```

Listing 2

Substrate VM is a framework that allows ahead-of-time compilation of Java applications under closed-world assumption into executable images or shared objects. Starting a Java application requires lots of steps and consumes tremendous amounts of resources (CPU, RAM), it is important to note that SVM chooses the words: "**closed-world assumption**" in the readme. The "closed-world assumption" is what makes SVM extremely fast, but also why not all applications can easily be ported over.

From the current limitations list, *Dynamic Class Loading / Unloading* are unsupported, which means, that most application servers don't compile; *Reflection, Dynamic Proxies* are partly supported, which means that Dependency Injection (DI) frameworks might work or have runtime issues.

Hello World

After downloading the GraalVM SDK and adding the "*bin*" directory to the *PATH*, it is now possible to start exploring. Consider the previous (*Hello World*) example, compile the application to Java bytecode and then generate a native image from the "`.class`" file (**listing 3**).

```
$ javac Hello.java

$ native-image Hello
...
[hello:24355]      compile:   9,989.94 ms
[hello:24355]        image:     897.76 ms
[hello:24355]        write:     180.53 ms
[hello:24355]      [total]:  22,785.96 ms

$ perf stat -e instructions ./hello
Hello World

 Performance counter stats for './hello':

         3,167,965      instructions:u

       0.002779353 seconds time elapsed

       0.000926000 seconds user
       0.001856000 seconds sys
```

Listing 3

The number of instructions, executed by the OS went from **138 million** to just **3 million**; the application time went from **0.042** to **0.00093** seconds. It shows an impressive 45x startup performance improvement.

So how does this work? It all boils down to AOT. Ahead-of-time compilation is the act of compiling the program intermediate representation bytecode, into (system-dependent) machine code so that the resulting binary file can execute natively. To achieve this, SVM needs to know all the classes required to run the application. However, SVM does some extra tricks to get even better startup results. SVM will during compilation run all required class static initializers to create at compile time the initial heap memory image (snapshot) for the process. Preloading the startup heap removes another CPU intensive step from the startup time. To illustrate this feature let's compile the following native image (**listing 4**).

Three things should stand out:

- There is no static initialization log message (the message was printed during native image generation)
- The application time traveled back in time (the static message is hardcoded to the native image generation time)
- The application runs in 0.014 seconds (the artificial sleep step doesn't occur anymore)

Listing 4

```
import java.util.Date;

public class Static {
    private static String message;

    static {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        message = "Hello from " + new Date();
        System.out.println("Initialized!");
    }
    public static void main(String[] args) {
        System.out.println(message);
    }
}
$ javac Static.java
$ native-image Static
Build on Server(pid: 28926, port: 44667)
...
Initialized!
...
[static:28926]      compile:   4,939.63 ms
[static:28926]        image:     557.42 ms
[static:28926]        write:     112.96 ms
[static:28926]      [total]:  14,481.05 ms
```

Why you (most likely) can't use native image your favorite framework application
Until now all examples are simple java applications, but most of the times we write complex applications with frameworks to help us be more productive and reduce boilerplate code. Frameworks can either work or not with SVM; it is a matter of trial and error, let's look at the simple Spring Framework example in **listing 5**.

```
@Controller
public class MyController {

    @Autowired
    private ProductService productService;

    public List<Product> getProducts(){
        return productService.listProducts();
    }
}
```

Listing 5

This piece of code does not compile a native image. When SVM processes this code, there is no knowledge on the runtime environment, so the `@Autowired` service implementation is not known at compile time. SVM requires all classes to be known at native image generation time, which is the opposite of the behavior of the Dependency Injection pattern.
To solve this issue, all runtime injection (and reflection usage) must run at compile time. Choosing a different DI framework might sound tempting; however, DI is just one issue to address, components that require reflection (e.g., JDBC) also need to be modified. A better alternative is to select a Framework that is rather explicit than implicit (e.g., Eclipse Vert.x), this allows full control on your application behavior and the execution path is always known by the compiler making it easy for SVM to build a native image.

Starting with Vert.x native images

The easiest way to start with Vert.x native images is to use the **project generator** web application. The interface should be trivial to understand. Once you open the web application, give a name to your project (e.g., vertx-svm), check the box "native-image" and download the project (**listing 6**).

```
vertx-starter
$ tree .
.
├── Dockerfile
├── mvnw
├── mvnw.cmd
├── pom.xml
├── README.md
└── src
    └── main
        ├── java
        │   ├── com
        │   │   └── example
        │   │       └── vertx_svm
        │   │           └── MainVerticle.java
        │   └── SVMSubstitutions.java
        └── resources
            └── META-INF
                └── native-image
                    └── io.vertx
                        └── vertx-core
                            ├── native-image.properties
                            └── reflection.json

11 directories, 9 files
```

Listing 6

The project structure is a typical Apache Maven project, with your main entry point as "`MainVerticle.java`". There are a couple of other files (e.g., "`SVMSubstitutions.java`" and "`reflection.json`"), these files include hot class code replacements (or substitutions) that modify external library code to conform to the SVM requirements. When there is a piece of code on a dependency library that fails to build, code that isn't compatible with SVM is replaced with a working substitution. The "reflection.json" is a simple way to whitelist code that is accessed by reflection which SVM would not consider being part of the application. The typical use case to add entries to this file is to whitelist classes that cause "ClassNotFound" exceptions at runtime.
The main difference between a "JVM" vertx application and a native image application is that native image vertx applications need a "`public static void main(String[])`" method because SVM needs to know the application entry point to perform the analysis of which code should translate to the final image. With the "**main**" method in place, there are no other differences between vertx applications running on the JVM or SVM.
Run the "**mvn package**" command to build the project locally, in case the host OS isn't (yet) supported (e.g., Windows) the generated project includes a "**Dockerfile**" that can build

the project in a container. Since docker should be on every developer's toolkit, to build this simple application, execute the docker build command (**listing 7**).

At the end of the build, start the application with the "**docker run**" command. The question we need to ask now is: What did SVM give us that we could not get from the JVM? Docker stats helps to answer the question.

```
$ docker stats

CONTAINER ID        NAME
CPU %               MEM USAGE / LIMIT
a71d55ec2699        agitated_hawking
0.02%               16.77MiB / 15.55GiB
```

It is quite impressive that this application has sub-millisecond startup times and uses as little as 16.77MiB of memory. Which vertx modules can already be used with SVM? Currently, many modules work out of the box, to name a few:

- reactive-pg-client (A high-performance modern PostgreSQL client)
- gRPC
- OAuth2
- JWT
- vertx-web (the Swiss Army Knife for building modern, scalable, web apps)

web sockets/event bus bridge

- SSL

Security TLS/SSL support

No modern application runs on plain HTTP anymore. TLS/SSL is a recent addition to SVM (since 1.0.0-RC7). Because of this fact support is limited and documentation is scarce. It is important to know that SVM only accepts SSL certificates in JCE keystores. Even though GraalVM is based on JDK8, SVM only recognizes PCKS12 (JDK9 and higher default format). To secure the example application, generate a certificate and convert it to PCKS12; and enable SSL (**listing 8**).

PCKS12 key stores also work out of the box with on JDK8, the same code runs regardless of the runtime.

Conclusion

Native images are yet another tool to add to your developer toolbox, as with any other tool, they have their pros and cons, and there are perfectly reasonable use cases for them,

```
$ docker build -t vertx-svm .
Sending build context to Docker daemon  120.8kB
Step 1/12 : FROM oracle/graalvm-ce:1.0.0-rc9 AS build-aot
...
Step 12/12 : ENTRYPOINT [ "/vertx_svm" ]
 ---> Running in 9c6be7b40da3
Removing intermediate container 9c6be7b40da3
Successfully built 944b62d317c7
Successfully tagged vertx-svm:latest

$ docker run --rm -it --net=host vertx-svm
Server listening at: http://localhost:8080/
```

Listing 7

```
# convert your existing key store to PCKS12
$ keytool -importkeystore -srckeystore certificates.keystore -dest-
keystore certificates.keystore -deststoretype pkcs12
HttpServerOptions httpOptions = new HttpServerOptions()
    .setSsl(true)
    .setKeyStoreOptions(new JksOptions()
    .setPath("certificates.keystore")
    .setPassword("localhost"));

vertx.createHttpServer(httpOptions).requestHandler(req -> {
    req.response()
    .putHeader("content-type", "text/plain")
    .end("Hello from Vert.x!");
}).listen(8080, res -> {
    if (res.failed()) {
    res.cause().printStackTrace();
    } else {
    System.out.println("Server listening at: https://localhost:8443/");
    }
});
```

Listing 7

e.g., CLI tools, serverless, small apps with a simple installation process, IoT are just some examples. Before I close, it is essential to keep on the back of the mind the fact that, native images are AOT compilation, and this means that there is no JIT anymore.
This small aspect is of great significance for long-running processes (think server-side applications). A long-running process with a JIT will most likely perform better than an AOT application because the JIT constantly re-optimizes the code to the current running conditions. The AOT application, on the other hand, relies on a closed world assumption and can only be tuned with profile guided optimizations (an Enterprise Edition only feature). Nevertheless, native-images allow for the first time, to use Java as a resource efficient and fast startup. The opportunities are endless! ■

Debugging Java in de Cloud

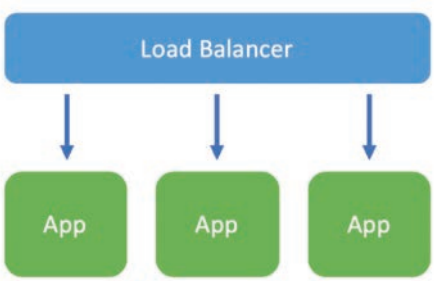
Een complicatie waar iedereen wel eens mee te maken heeft

De tijd dat we applicaties puur monolithisch bouwden hebben we alweer enige tijd achter ons gelaten. Tegenwoordig worden applicaties steeds vaker opgebouwd uit een collectie van kleinere services. Best een mooie ontwikkeling, maar het brengt ook een aantal complicaties met zich mee. In dit artikel belicht ik één van de complicaties waar iedereen direct mee te maken krijgt: debuggen.

De beste vrienden van een ontwikkelaar voor het ontrafelen van problemen met een applicatie zijn logging en breakpoints. Al sinds de begindagen van het programmeren, schrijven ontwikkelaars informatie over het gedrag van de applicatie naar logfiles. Deze informatie kan gebruikt worden voor allerlei doeleinden, maar heeft meestal als hoofdfunctie het kunnen detecteren van problemen en het geven van context om te helpen bij het vinden van de oorzaak van optredende problemen. Als logging niet genoeg informatie geeft om het opgetreden probleem te verhelpen, dan kan het plaatsen van één of meerdere breakpoints in de applicatie helpen. Hiermee kan een applicatie worden stilgezet om de interne status op dat specifieke moment te bekijken. Breakpoints zijn vooral handig tijden het ontwikkelen. Je zal ze niet gauw gebruiken in een productie omgeving om de eenvoudige reden dat je applicatie letterlijk gepauzeerd wordt met als gevolg dat er geen bewerkingen meer plaatsvinden. Wat niet wil zeggen dat het niet kan, maar daarover later meer.

Monolitisch

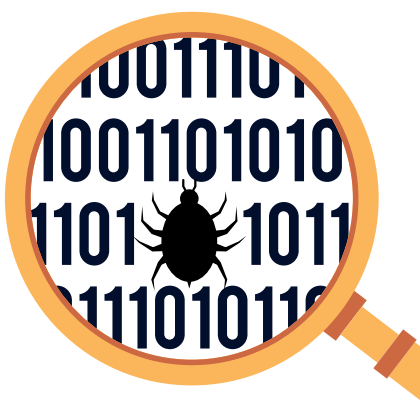
Voor we applicaties zijn gaan opsplitsen, draaiden applicaties als monoliet op een veelvoud aan servers. Als je wilde weten wat er met een request gebeurde, dan moest je uitvinden op welke server de request terecht gekomen was en kon je daar de logfile bekijken. Niet dat dit nou persé heel gemakkelijk was. Soms had je te maken met honderden servers, maar als je de juiste server gevonden had, had je ook meteen alle informatie over die request gevonden.

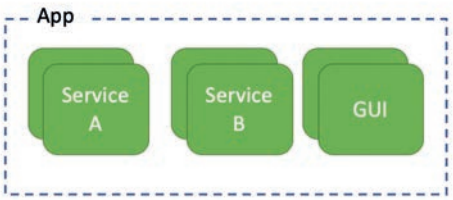


Services

Met het opsplitsen van applicaties in kleinere services die verspreidt draaien over een veelvoud aan servers wordt het een stuk ingewikkelder om de afhandeling van een request te volgen. Waar de afhandeling van een request voorheen door exact één server gedaan werd, wordt nu in veel gevallen de afhandeling door een aantal verschillende services gedaan die zich niet noodzakelijkerwijs op dezelfde server bevinden. Daarmee staat de informatie over die request ook op verschillende servers.

Bas Passon is werkzaam als senior Java developer bij First8. Op dit moment houdt hij zich vooral bezig met Cloud Native Java applicatie ontwikkeling.





Container Orkestratie

Een complicerende factor is dat veel applicaties tegenwoordig in een gedistribueerde containeromgeving draaien waarbij de verschillende services niet permanent op één vooraf gekozen server draaien. Containeromgevingen, gemanaged door containerorkestratie systemen, tegenwoordig bijna altijd Kubernetes, willen services (containers) nog wel eens verplaatsen van de ene server naar de andere. Een eindgebruiker heeft daar geen last van, maar een ontwikkelaar die logging wil inzien wel.

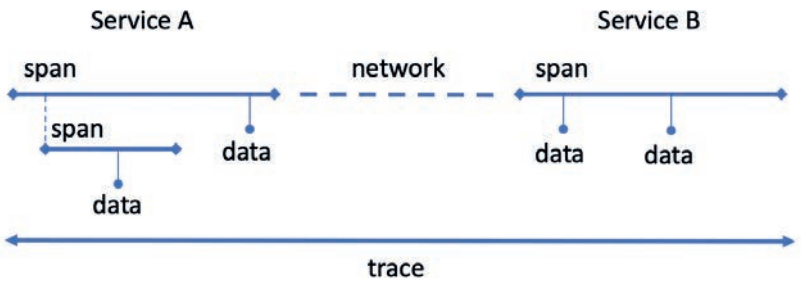
Logging Aggregatie

Met het opsplitsen van applicaties in kleinere services, waarbij deze draaien in gedistribueerde containeromgevingen, is het voor ontwikkelaars een stuk lastiger geworden om snel de benodigde log informatie te vinden. Waar voorheen inloggen op een server en daar de logfile bekijken volstond, is het nu niet altijd evident waar de log informatie te vinden is.

Geen reden tot wanhoop. Nieuwe problemen vragen om nieuwe oplossingen en hier is dat aggregatie van log informatie op een centrale plek. Als ontwikkelaar hoef je dan niet te weten waar services precies draaien, maar kan je alle log informatie op een centrale plek bekijken. Al een hele verbetering ten opzichte van het zoeken naar log informatie op de verschillende servers waar services zich zouden kunnen bevinden. We hebben nu dan wel alle log informatie op een centrale plek beschikbaar, maar het is nog niet heel eenvoudig om de log informatie te relateren aan requests van eindgebruikers. Alles staat door elkaar en het is niet direct duidelijke welke log informatie bij welke request hoort.

Distributed Tracing

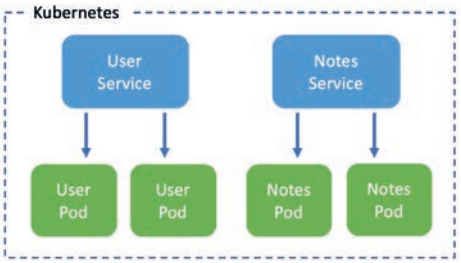
Om log informatie van requests aan elkaar te koppelen over verschillende service aanroepen heen, moeten deze op de één of andere manier aan elkaar gerelateerd kunnen worden. Dit is waar distributed tracing om de hoek komt kijken. Dit is het kunnen volgen en analyseren van een request als deze door het landschap van services beweegt. De basis hiervoor is ooit door Google beschreven in een paper met de naam: 'Dapper, a Large-Scale Distributed Systems Tracing Infrastructure' (1). In de kern komt het erop neer dat bij de start van een request een trace opgestart wordt waar services spans aan kunnen toevoegen met informatie. Als we het model visualiseren, dan krijgen we het volgende:



Het analyseren laten we hier even links liggen. Het gaat ons nu vooral om het kunnen volgen van requests. Dit kunnen we bewerkstelligen door gebruik te maken van de trace en de spans die eraan gelinkt zijn.

Spring Boot

We weten nu hoe we een applicatie opgesplitst in services draaiend in een gedistribueerd container platform kunnen debuggen. Nou ja, theoretisch dan. Laten we eens kijken wat we moeten doen om een applicatie bestaande uit een tweetal Spring Boot gebaseerde services draaiend in Kubernetes, weergegeven in onderstaande figuur, te voorzien van distributed tracing.



De applicatie bestaat uit een simpele use case. We hebben gebruikers die notes hebben opgeslagen. De user service weet alles van de

DE EVOLUTIE VAN MONOLIET NAAR (MICRO) SERVICES EN CLOUD HEEFT EEN HEEL NIEUW SCALA AAN MOGELIJKHEDEN GECREËERD

users en de notes service weet alles van de notes van gebruikers. Een aanroep naar de user service haalt de informatie van de user op inclusief zijn notes, die opgehaald worden bij de notes service. Niet de meest interessante use-case, maar het volstaat voor ons doel: het volgen van requests door het service landschap. Hoe pakken we dat aan? Alles begint met Spring Cloud Sleuth.

Spring Cloud Sleuth

Spring Cloud Sleuth (3) is een distributed tracing implementatie voor het Spring platform. Het maakt gebruik van de ideeën van Google's Dapper paper en bouwt op ZipKin en HTrace, beide distributed tracing frameworks. Sleuth faciliteert tracing door op alle -binnen een Spring Boot applicatie gevonden koppelvlakken- interceptors te plaatsen die zorg te dragen voor het aanmaken van traces en spans en deze te voorzien van analyse informatie. Om Sleuth te gebruiken, moet de dependency van **listing 1** toegevoegd worden aan de project pom.xml.
Met deze dependency op het classpath wordt runtime het logging formaat van je applicatie aangepast, zodat trace en span hierin komen. Dat ziet er dan als **listing 2** uit.

Het verschil met het standaard logging formaat is de toevoeging van **[user-service,1af74bee2eef448c,1af74bee2eef448c,false**] achter het level. Deze informatie wordt door Sleuth ingevoegd en is de basis voor het kunnen volgen van requests. De betekenis van deze waarden is **[service-identifier,trace-id,span-id,exported].** De eerste drie zijn interessant voor het volgen van requests, exported is nuttig als je ook analyses doet met een analyse server, b.v. ZipKin. De betreffende trace is dan geëxporteerd voor analyse doeleinden.

Nu we in de logs waarden hebben die we kunnen gebruiken om log output te koppelen aan een unieke request, moeten we deze nog wel ergens aggregeren. De log output staat immers nu nog op verschillende plekken in het Kubernetes cluster en dat maakt het volgen van de request onhandig. Voor het aggregeren van de log output zijn verschillende opties. Hier kiezen we voor de welbekende Elastic Stack (4) bestaande uit Elasticsearch, Logstash en Kibana.

Elastic Stack

Elasticsearch, Logstash, Filebeat en Kibana zijn vier open source producten die samen een krachtige log analyse oplossing vormen. Elasticsearch is een NoSQL database en uitermate geschikt voor het opslaan van log informatie. Filebeat verzamelt log informatie. Logstash ontvangt en transformeert log informatie en Kibana levert visualisatie en analyse mogelijkheden.

Waar het in de begindagen van cloud computing nog best een uitdaging was om de Elastic Stack aan de praat te krijgen, is dat tegenwoordig een stuk eenvoudiger geworden met de komst van Helm. Dit is een package manager voor Kubernetes. Met behulp van de Helm chart (5) voor Elastic Stack zijn we vrij snel up en running. Na de initiële configuratie van Kibana komen de log regels van de service binnen. Echter, de informatie is nog niet opgesplitst in losse velden waar we op kunnen zoeken en filteren. Om dit voor elkaar te krijgen, moet de configuratie van Logstash aangepast worden. We maken gebruik van de Grok filter plugin.

Grok

De Grok filter plugin kan op basis van reguliere expressies velden extraheren uit de ruwe log informatie. Het opstellen van de juiste reguliere expressie heeft altijd wat voeten in aarde. Gelukkig is voor alles een app. Zo is hier Grok Debug (6) om ons te helpen. Na wat pogingen volstaat de expressie van **listing 3**,

Nu Logstash de verschillende velden uit de log informatie haalt, kunnen we in Kibana gebruik maken van de losse velden. Als we een trace isoleren, ziet dat er als **afbeelding 1** uit.

NIEUWE PROBLEMEN VRAGEN OM NIEUWE OPLOSSINGEN

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Listing 1: dependency

```
2018-12-10 22:48:45.354  INFO [user-service,1af74bee2eef448c,1af74bee
2eef448c,false]
    13522 --- [ctor-http-nio-2] nl.first8.debug.UserController
: Getting user for name joe.
```

Listing 2: trace en span

```
%{TIMESTAMP_ISO8601:timestamp} *%{LOGLEVEL:level} \[%{DATA:applicati
on},%{DATA:trace},%{DATA:span},%{DATA:exported}] %{DATA:pid} --- *\
[%{DATA:thread}] %{JAVACLASS:class} *: %{GREEDYDATA:message}
```

Listing 3: trace en span

Time	level	service	trace	span	message
December 12th 2018, 21:32:40.476	INFO	notes-service	58b1f9deef36c7cb	c27fdeabad03b5eb	Finding notes for user basp
December 12th 2018, 21:32:40.462	INFO	user-service	58b1f9deef36c7cb	58b1f9deef36c7cb	Getting user for name basp.

Afbeelding 1

We kunnen nu de afhandeling van een request volgen zonder dat we hoeven te weten waar de verschillende services zich bevinden. Alle informatie is op basis van trace aan elkaar te koppelen. Het bij elkaar halen van relevante log informatie in het geval van problemen is nu wel heel eenvoudig geworden.

Bonus: Breakspoints in Kubernetes

Zoals al eerder aangaven, is log informatie soms niet genoeg om erachter te komen wat er aan de hand is. Soms wil je een breakpoint kunnen gebruiken om in de service zelf te kijken. Iets wat lokaal heel eenvoudig is, lijkt in een Kubernetes omgeving haast onmogelijk, maar niets is minder waar. De Kubernetes command line utility, kubectl, komt met een standaard port-forward functionaliteit. Deze kunnen we gebruiken om een lokale poort door te lussen naar een port op een container in het cluster. Dat kan natuurlijk ook de debug poort van de JVM zijn. Voegen we de volgende argumenten toe aan de JVM in de container om deze in debug modus te starten:

```
-Xdebug
-Xrunjdwp:server=y,transport=dt_
socket,address=5000,suspend=n
```

De JVM luistert op poort 5000. Hier kan een remote debugger dan op verbinden. Het is goed even extra te vermelden dat je de waarde van suspend echt op n laat om te voorkomen dat de JVM gaat staan wachten tot een remote debugger verbinding maakt. Je wilt uiteindelijk alleen indien nodig verbinden met één van de vele containers. Stel, je wilt een breakpoint zetten in één van de user-service containers uit het eerdere voorbeeld met id user-service-765d459796-258hz, dan kunnen we de de port-forward als volgt opzetten:

```
kubectl port-forward user-ser-
vice-765d459796-258hz 5000:5000
```

Nu kan met een remote debugger verbinding gemaakt worden op localhost:5000 voor het zetten van breakpoints. Zo eenvoudig is het! Let wel, de JVM is dan gepauzeerd en zal geen requests meer afhandelen die er naartoe doorgezet worden, zolang je een andere request aan het debuggen bent.

Conclusie

De evolutie van monoliet naar (micro)services en cloud heeft een heel nieuw scala aan mogelijkheden gecreëerd voor het bouwen van oplossingen en daarmee ook een heel scala aan nieuwe beheer uitdagingen. Logging is er één van en was in de tijd van de monoliet al een uitdaging die er met de komst van (micro)services en cloud niet eenvoudiger op geworden is. In dit artikel heb ik laten zien dat het logging probleem goed op te lossen is met een aantal populaire open source oplossingen. Naast logging hebben we het kort gehad over debuggen met breakpoints. Met wat kunst en vliegwerk is ook dat voor elkaar te krijgen. Het heeft echter wel als nadeel dat eindgebruikers er direct hinder van ondervinden. Zelf zal ik het dan ook alleen in uiterste noodgevallen inzetten op een productie omgeving. ■

ALLES BEGINT MET SPRING CLOUD SLEUTH

REFERENTIES:

1. **https://ai.google/research/pubs/pub36356**
2. **https://spring.io/projects/spring-boot**
3. **https://spring.io/projects/spring-cloud-sleuth**
4. **https://www.elastic.co/**
5. **https://github.com/helm/charts/tree/master/stable/elastic-stack**
6. **https://grokdebug.herokuapp.com/**

Interview James Gosling

James Gosling over het verleden, het heden en de toekomst van Java

James Gosling was op 18 oktober in Nederland. De mensen van Blue4IT was het gelukt om de geestelijke vader van Java voor een bezoek naar Nederland te halen. De NLJUG was daar natuurlijk bij! Hier een kort verslag van de bijeenkomst, met daarbij zijn leukste quotes over Java.

Wat:	Meet-up James Gosling
Waar:	Kantoor Amazon Web Services Amsterdam
Wanneer:	Donderdag 18 oktober 2018
Wie:	Meet-up Host Amazon in samenwerking met Blue4IT

De naam James Gosling behoeft natuurlijk geen verdere toelichting voor de lezers van Java Magazine. Als Java-ontwikkelaar weet je op zijn minst wie de bedenker is van Java. Hij ontwierp de taalstructuur, bouwde de compiler en de virtuele machine. Gosling werkte sinds 1984 bij Sun Micrososystems. In 2010 nam hij ontslag bij Oracle, dat Sun Microsystems had overgenomen.

In 2017 sloot Gosling zich aan bij het ontwikkelteam van Amazon. Op de vraag wat hij daar nou precies doet, was het antwoord kort en duidelijk "Over de meeste interessante zaken waaraan ik werk, kan ik niet praten." Maar op de andere vragen tijdens de Q&A van de bijeenkomst van 18 oktober 2018 kon hij gelukkig uitgebreider antwoorden. Hij besprak van alles over het verleden, het heden en de toekomst van zijn geesteskind, Java.

Het begin van Java: IoT

In het begin was Java gewoon een experiment van enkele ingenieurs bij Sun om pijnpunten aan te pakken in een domein wat we nu IoT zouden noemen, blikt Gosling terug: "We wilden software voor apparaten bouwen die betrouwbaar en veilig is. En dat werkte ook." Maar het geld zat toen niet in IoT, maar in datacenters. En dus werd Java opgepikt in andere domeinen. "Ondertussen zit Java natuurlijk wel in talloze apparaten: niet alleen in Android-telefoons, maar ook in koffiemachines en dergelijke."

Heeft Gosling dan geen spijt over specifieke beslissingen die hij heeft genomen in het ontwerp van de taal? Ja, zo heeft Java geen unsigned int en long, en die zijn volgens hem ook niet nodig. Maar hij gaf wel toe dat hij niet zo gelukkig is met het feit dat byte een signed type is: "Ik heb die beslissing toen genomen uit symmetrie-overwegingen met int en long, maar ik doe zelf veel aan bit fiddling en merk dan dat het gebrek aan een unsigned byte wel lastig is."

Nieuwe taalconstructies

Java evolueert en er komen dan ook regelmatig nieuwe taalconstructies. Voor gebruikers lijkt het soms tergend traag te gaan voordat we van een nieuwe taalconstructie kunnen genieten, maar volgens Gosling is daar een reden voor: "Zo waren lambda's op zich niet zo problematisch om in Java op te nemen, maar het was belangrijk om de performance goed te krijgen. Er was heel wat tijd nodig om alles onder de motorkap in orde te krijgen."

Value types zijn zo'n andere taalconstructie die velen graag in Java zouden zien: een derde soort waarde naast primitieve types en objecten, te vergelijken met structs in C. Volgens Gosling is dat moeilijk om semantisch goed te krijgen. "Ik heb heel wat tijd besteed aan alle randvoorwaarden. Toen Brian Goetz me vroeg hoe moeilijk het was, zei ik: 'Ongeveer een dozijn doctoraatsthesissen'. En Brian antwoordde me: 'Volgens mij zijn het maar twee of drie'. Een jaar geleden kwam hij naar me toe en zei me: 'James, je was nog optimistisch'."

Functioneel programmeren

Iemand stelde de vraag wat Gosling vond van het feit dat Java als objectgeoriënteerde taal

Koen Vervloesem is een freelance journalist met interesse in vele onderwerpen zoals programmeren in Java, C, Ruby, Prolog en XML en zijn toepassingen.

IN HET BEGIN WAS JAVA GEWOON EEN EXPERIMENT VAN ENKELE INGENIEURS BIJ SUN

begon, maar nu meer en meer richting een functionele taal aan het evolueren is. "Beide aspecten van een taal sluiten elkaar niet uit", legt Gosling uit. "Ze doen het heel goed samen. Functioneel programmeren is een stijl. Lambda's, generics en recursie zijn elementen van een functionele programmeerstijl."

"Ik heb persoonlijk altijd in een functionele stijl geprogrammeerd, zelfs al voordat er lambda's waren. Ik ben een voorstander van recursie, ook al vinden veel mensen dat vreemd als ze mijn code lezen. Maar als je echt alles functioneel wilt doen, ziet je code er wat geforceerd uit: daar explodeert je hoofd gewoon van. Ik probeer altijd een evenwicht te vinden tussen object georiënteerd en functioneel programmeren."

Testen

Iets anders waar Gosling veel belang aan hecht, is testen: "Ik ben bijna religieus over testen. Als ik wil weten of een stukje code werkt, begin ik nooit de code te debuggen of een voorbeeldtoepassing te schrijven; ik schrijf een unittest. Elke keer dat ik een fout oplos, voeg ik een unittest toe. Elke keer dat ik een nieuwe functie schrijf, voeg ik ook een unittest toe."

"Daarom hou ik ook van checked exceptions", legt Gosling uit. "Die waarschuwen je op tijd wanneer je het bijna aan het verpesten bent. Veel mensen vinden checked exceptions vervelend. Waarom kan een functie geen null teruggeven wanneer die faalt, zoals in C, zeggen ze dan. Maar niemand controleert op null. Het universum zit vol met code die bestanden opent, die perfect werkt op de laptop van de ontwikkelaar, tot die plots merkt dat het pad op de productiemachine niet bestaat. Checked exceptions zijn mijn vervelende manier om aan ontwikkelaars te zeggen dat ze moeten opletten."

De toekomst van Java

Over de toekomst van Java is Gosling positief: "Oracle heeft de community verplicht om taken op zich te nemen en ik ben redelijk gelukkig over hoe de community dat doet. We hebben OpenJDK en bedrijven, zoals Red Hat en Azul Systems, geven goede Java-ondersteuning."

Ook over de richting waarin de taal zelf aan het evolueren is, is Gosling vrij tevreden, zelfs al verloopt die langzaam: "Het is moeilijk om een evenwicht te vinden tussen de noden van miljoenen ontwikkelaars. Maar de echte problemen liggen volgens mij niet meer in programmeertalen, maar in tools, API's en methodologieën."

Gosling beschouwt tot slot Java als veel meer dan de taal en op dat gebied ziet hij ook mooie ontwikkelingen: "Java is voor mij niet de taal, maar de virtuele machine. Dat er heel wat coole talen op die VM draaien, zoals Kotlin, Clojure en Scala, vind ik geweldig: gebruik gewoon wat het beste werkt voor jou." ■

OVER DE TOEKOMST VAN JAVA IS GOSLING POSITIEF

Van Java 8 naar Java 11

Wij waren early adopters en dit kwamen we tegen

"Je kunt dus kiezen tussen de blauwe en de rode pil. Als je de blauwe pil neemt, dan word je wakker in je bed en blijf je ontwikkelen tegen Long Term Support-releases, met elke drie jaar een grote migratie. Neem je de rode pil, dan ga je wat beleven! Je kunt dan genieten van de nieuwste mogelijkheden en performance-optimalisaties, met elke zes maanden een kleine migratie. De keuze is aan jou; ik bied je enkel de waarheid."

Mark Reinhold is niet alleen fan van 'The Matrix', hij is ook de Chief Architect van de Java Platform Group. En tijdens zijn *keynote* op Devoxx[1] legde hij op deze manier de nieuwe release cadans van Java uit. Wat was ik blij dat we bij Translink nét onze software naar Java 11 hadden gebracht. In de toekomst zouden we nu, net als Neo, de rode pil kunnen kiezen!

Ruimte op de backlog

Zes weken daarvoor schoven we het kaartje "Upgrade naar Java 11" naar de kolom "In Progress". Java 11 was diezelfde dag uitgekomen, dus we waren aan de vroege kant. Maar we konden eigenlijk niet anders. We wisten al dat eind 2018 een groot nieuwbouwproject zou starten en tot die tijd hadden we wat ruimte op de backlog. Bovendien zou in januari 2019 de *support* op Java 8 voor commerciële gebruikers vervallen[2]; upgraden in die maand zou, wegens het geplande nieuwbouwproject, slecht uitkomen.

Vrij kunnen bewegen

Het eerder genoemde nieuwbouwproject heeft alles te maken met de uitdagingen die Translink voorziet in de nabije toekomst. Als verwerker van meer dan 2,5 miljard transacties per jaar hebben we onze sporen verdiend als hart voor het betalingsverkeer in het openbaar vervoer. Maar onze reizigers willen meer betalingsmogelijkheden, zodat ze zich vrij(er) kunnen bewegen. Ze willen niet alleen reizen met een chipkaart, maar ook met een creditcard, smartphone of smartwatch. Translink bereidt deze toekomstige mogelijkheden voor door software te ontwikkelen die deze betaalmiddelen ondersteunt. Zo'n innovatief stuk software verdient beter dan een Java-versie van ruim vijf jaar oud.

Nieuwe mogelijkheden

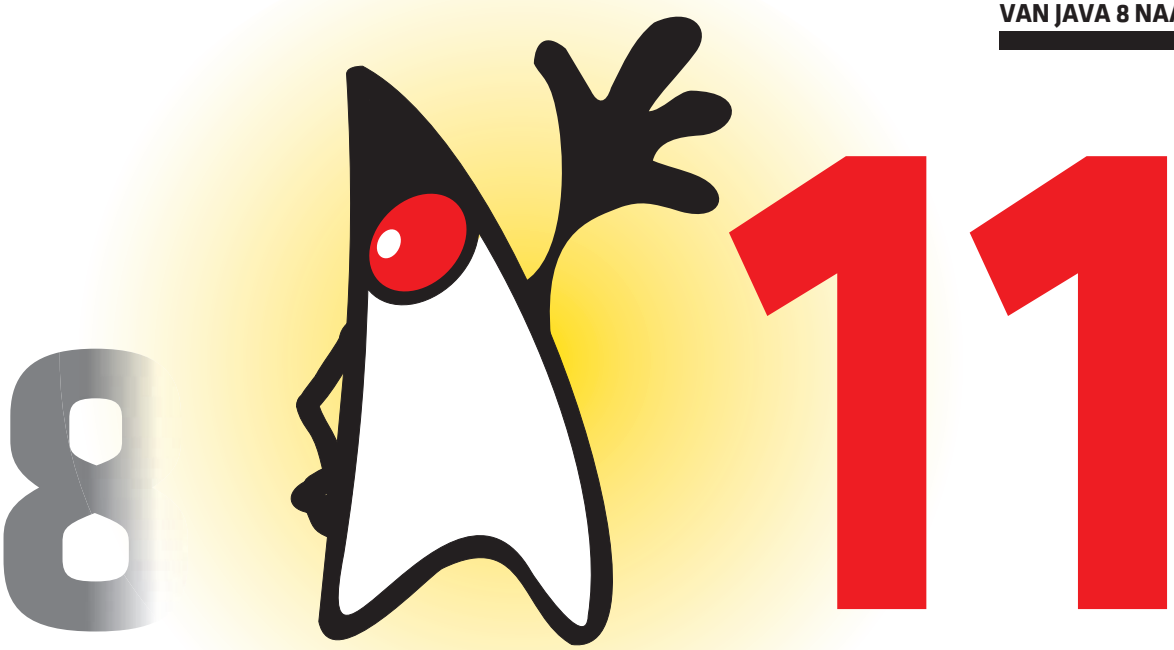
Niet verwonderlijk dus dat de ondersteuning per januari 2019 afloopt. Dat wil zeggen: Oracle brengt in januari 2019 de laatste vrij beschikbare update voor Java 8 uit. Daarna zijn de updates alleen nog beschikbaar voor houders van een commerciële licentie. Met zo'n licentie ben je nog tot maart 2022 onder de pannen[3], maar dat zal wel financiële gevolgen hebben.

Daarnaast mis je natuurlijk allerlei nieuwe mogelijkheden, zolang je op Java 8 blijft. Denk o.a. aan het modulesysteem uit Java 9, `var` als typeaanduiding uit Java 10[4] en het direct uitvoeren van één `.java`-bestand uit Java 11[5]. Verder is Java 11 in de nieuwe release cadans een Long Term Support-versie, met gratis publieke updates t/m 2023[6]. Dat maakt Java 11 een interessante release om naar te upgraden, omdat je vanaf daar beide upgradestrategieën, die eerder in de inleiding genoemd werden, kunt uitvoeren.

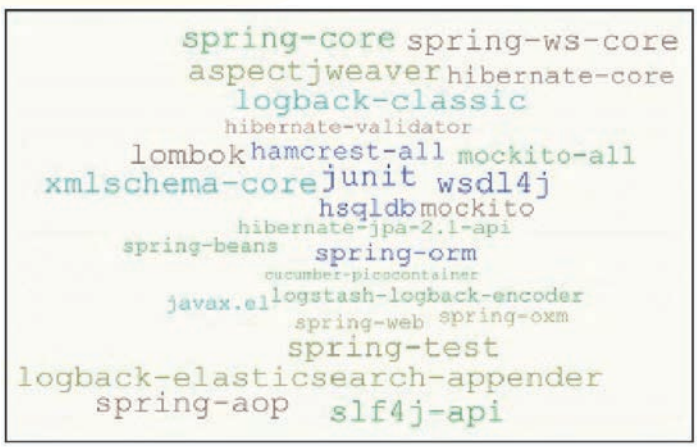
Hoe ziet jullie software er eigenlijk uit?

De componenten die we naar Java 11 hebben gebracht, bevinden zich in de mid-office van Translink. Ze voorzien de OV-chipkaart-website van de benodigde gegevens en logica om bijvoorbeeld reishistorie te bekijken of saldo te bestellen. Ze zijn gebouwd op basis van Spring

Hanno Embregts is als software engineer bij Info Support werkzaam voor Translink. Hij houdt van full-stack software ontwikkelen en dingen upgraden. Daarnaast is hij actief als trainer bij het Kenniscentrum van Info Support en spreekt hij regelmatig op internationale conferenties.



5. Verder maken we voor de *continuous delivery* gebruik van Maven, Jenkins 2, SonarQube en Docker Compose. In **Figuur 1** staat in meer detail opgesomd welke dependencies we in deze componenten voornamelijk gebruiken.



Figuur 1: Groter afgebeelde dependencies komen vaker voor.

Welke stappen hebben jullie uitgevoerd?

We zijn begonnen met het downloaden van JDK 11 en het updaten van onze IDE. Daarna hebben we, mede ingegeven door een uitstekend artikel van Leonardo Zanivan[7], voor elk softwarecomponent de volgende fases doorlopen:

1. Uitvoeren op Java 11;
2. Compileren met Java 11;
3. Modulariseren.

Heb je, net zoals wij, voldoende tijd om de upgrade uit te voeren, dan kun je deze fases gelijktijdig of in ieder geval direct achter elkaar uitvoeren. Is je tijd meer gefragmenteerd, dan is het ook mogelijk om steeds maar één fase te doen, omdat na iedere fase een werkbare situatie blijft bestaan.

Uitvoeren op Java 11

In de meeste gevallen is dit de meest eenvoudige stap: software die met Java 8 gecompileerd is, draait namelijk prima op Java 11, zolang je niets uit Java EE of CORBA gebruikt[8]. In het ergste geval moet je een aantal dependencies toevoegen, zoals `java.activation` of `java.xml.bind`.

Compileren met Java 11

Als je vervolgens je softwareproject met Java 11 gaat compileren, dan zal je de melding tegenkomen die in **Listing 1** staat weergegeven. Al sinds Java 9 verschijnen waarschuwingen als de interne packages `sun.*` en `*.internal.*` worden gebruikt, om de gebruiker te laten weten dat deze packages in de toekomst niet meer te gebruiken zijn.

In dit geval probeert Lombok de genoemde interne packages te gebruiken; dit hebben we opgelost door de laatste versie van Lombok te gebruiken. Als de interne packages vanuit je eigen code worden gebruikt, zal je voor die functionaliteit een alternatief moeten vinden. Overigens kan dit probleem ook optreden als transitieve dependencies interne packages benaderen. In dat geval kun je de transitieve dependency overschrijven, zoals staat weergegeven in **Listing 2**.

```
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ first-component ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 70 source files to C:\dev\translink\first-component\target\classes
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by lombok.javac.apt.LombokProcessor to field com.sun.tools.javac.processing.
JavacProcessingEnvironment.discoveredProcs
WARNING: Please consider reporting this to the maintainers of lombok.javac.apt.LombokProcessor
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

Listing 1: De uitvoer van Maven als je een bestaand project met Java 11 compileert.

Modulariseren

Op dit punt werkt je component al op Java 11. Je gebruikt alleen nog niet het modulesysteem. Dit kun je activeren door in `src/main/java` een `module-info.java` aan te maken. Deze bevat de modulenaam, een lijst van benodigde modules van buitenaf (`requires`) en een lijst van packages die `compile-time` en/of `run-time` beschikbaar zijn voor andere modules (respectievelijk `exports` en `opens`). Je IDE zal je overigens op basis van de `pom.xml` snel kunnen vertellen voor welke modules je `requires` toe moet voegen.

Obstakels

Onderweg kwamen we allerlei obstakels tegen, die we hierna in meer detail behandelen. Wie weet kan onze aanpak jouw project helpen.

‘Split packages’

Sinds Java 9 is het niet meer mogelijk een package te gebruiken die in twee modules bestaat. Java gebruikt hiervoor het *module path*. Directe dependencies komen altijd op het *module path* uit, maar het gedrag van transitieve kan verschillen (afhankelijk van de implementatie). Maven zet transitieve dependencies bijvoorbeeld op het *classpath*, terwijl IntelliJ IDEA ze op het *module path* zet. Zo kan het voorkomen dat je component compileert via Maven, maar dat IntelliJ je om de oren slaat met meldingen over ‘split packages’.

Er bestaan verschillende, tijdelijke oplossingen die je kunt proberen[9], maar de enige definitieve oplossing is het verwijderen van de *package split*. Hoe die oplossing eruit ziet, is afhankelijk van de situatie, maar kan simpeler uitpakken dan je van tevoren misschien denkt. Wij kregen de melding bijvoorbeeld op de package `javax.servlet`, die we hebben opgelost door één referentie naar de `Servlet`-klasse in JavaDoc te vervangen door de tekst “servlet”. Toen was de dependency niet meer nodig en het probleem opgelost.

```
<plugin>
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <version>0.14.0</version>
    <dependencies>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.4.0-b180830.0359</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jaxb</groupId>
            <artifactId>jaxb-runtime</artifactId>
            <version>2.4.0-b180830.0438</version>
        </dependency>
    </dependencies>
</plugin>
```

Listing 2: Overschrijven van transitieve dependencies in Maven.

Docker

Voorafgaand aan de upgrade gebruikten we `openjdk:8-jre-alpine` als basisimage voor onze Docker-containers. Maar Java 11 heeft momenteel helaas geen stabiele *build* voor Alpine Linux[10]; daardoor zijn we voorlopig genoodzaakt om het veel grotere image `openjdk:11-jre-slim` als basis te gebruiken. Mogelijk komt er voor Java 12 wel een stabiele release, dat is nog even afwachten.

SonarQube & Jenkins

Misschien kwam het doordat we zo vroeg waren gestart, maar de ondersteuning van SonarJava[11] en Jenkins[12] voor Java 11 liet even op zich wachten. Sonar kan nu Java 11-bestanden scannen, maar zelf nog niet op Java 11 draaien. Jenkins had de Java 11-ondersteuning pas begin december aangekondigd. Een direct gevolg was dat we een aantal componenten een tijdje op Java 10 hebben gehouden. Nu de ondersteuning er is, kunnen we de upgrade voor die componenten afronden.

Zijn jullie blij met het resultaat?

Voor een paar componenten wachten we dus nog op betere Java 11-ondersteuning. Maar de rest van de upgrade is voltooid en dus zijn

ORACLE BRENGT IN JANUARI 2019 DE LAATSTE VRIJ BESCHIKBARE UPDATE VOOR JAVA 8 UIT

we klaar om nieuwe functionaliteit te gaan opleveren. Wel zijn er een aantal zaken die we een volgende keer anders gaan aanpakken!

Welk component als eerste?

We waren het modulariseren bewust begonnen bij een component van gemiddelde grootte, met een gemiddeld aantal dependencies. De reden daarvoor was dat we onze ervaringen van de upgrade van dit component wilden gebruiken om in de volgende Sprint Planning meer zekerheid te kunnen geven over de doorlooptijd van het geheel. Dat kwam echter niet goed uit de verf, omdat de modularisatie van een component pas voltooid is als de dependencies van dat component óók in modules zijn opgedeeld. Hoe kun je immers in de `module-info.java` refereren naar een module die nog niet bestaat? Zo kwam het dat we eerst zes andere componenten moesten modulariseren voordat bij ons proefcomponent de upgrade voltooid was. En daarmee bleek het positieve effect op de Sprint Planning helaas nihil. Een volgende keer gaan we eerst de dependencies analyseren en starten bij een component dat weinig dependencies op andere projecten heeft, maar zelf wél als dependency gebruikt wordt.

Direct na de release starten

Vanwege de ruimte op de backlog begonnen we de upgrade zelfs op de dag dat Java 11 uitkwam. Ook dat doen we een volgende keer anders. Veel dependencies werkten wegens het gebruik van interne packages in die eerste weken nog niet op Java 11. We konden niet veel anders dan het probleem melden bij de maker en de waarschuwing tijdelijk negeren. Naarmate de tijd verstreek, kwamen steeds meer *libraries* met updates, waardoor de waarschuwingen verdwenen. Mooi natuurlijk, maar tegelijkertijd bekroop ons het gevoel dat het beter was geweest om na de release van Java 11 een week of drie, vier te wachten voordat we aan de upgrade waren begonnen. Voor een volgende upgrade zijn we dus gewaarschuwd.

Wat te doen met mijn project?

Op het moment dat deze editie van Java Magazine verschijnt, is Java 11 al ruim vier maanden beschikbaar. Daardoor kun je op veel meer Java 11-ondersteuning rekenen dan wij dat konden. En ook met weinig ruimte op je backlog kun je aan de upgrade beginnen, door steeds één van de drie stappen apart uit te voeren. Doe dit als het je uitkomt.

Figuur 2: Het mag ook wat minder populair zijn, hoor.

Mocht je overigens elke drie jaar een dergelijke migratie gaan doen (naar de LTS-releases – 'de blauwe pil'), dan kun je het werk wat spreiden door je software tussendoor te draaien met een non-LTS-release en alvast wat aanpassingen te doen. Ga je elke zes maanden migreren – 'de rode pil' – geef dan het regelmatig bijwerken van je *tools* en dependencies prioriteit.

Wat je ook gaat kiezen, vergeet niet te genieten van de nieuwe features die je met de nieuwe Java-versie tot je beschikking krijgt. En… schep er vooral over op. Want (zie **Figuur 2**) dat heeft de Chief Architect van de Java Platform Group ons immers gevraagd. ■

VERGEET NIET TE GENIETEN VAN DE NIEUWE FEATURES DIE JE MET DE NIEUWE JAVA-VERSIE TOT JE BESCHIKKING KRIJGT

VOETNOTEN

[1] "'Java in 2018: Change is the Only Constant' Keynote by Mark Reinhold" - **https://youtu.be/wHoRBvt3U6o**
[2] "Oracle sets date for end of Java 8 updates" / Paul Krill - **https://www.infoworld.com/article/3269332/java/oracle-sets-date-for-end-of-java-8-updates.html**
[3] "Oracle Java SE Support Roadmap" - **https://www.oracle.com/technetwork/java/java-se-support-roadmap.html**
[4] "Java 10: een overzicht van de laatste wijzigingen" / Johannes Boneschanscher – Java Magazine 3, 2018
[5] "JEP 330: Launch Single-File Source-Code Programs" - **https://openjdk.java.net/jeps/330**
[6] De updates van Oracle zijn alleen de eerste zes maanden vrij beschikbaar, daarna heb je een commerciële licentie nodig of moet je een JDK gebruiken die gedurende een langere periode publieke updates uitbrengt, zoals AdoptOpenJDK of Amazon Corretto.
[7] "It's time! Migrating to Java 11" / Leonardo Zanivan - **https://medium.com/cricumadev/its-time-migrating-to-java-11-5eb3868354f9**
[8] "JEP 320: Remove the Java EE and CORBA Modules" - **http://openjdk.java.net/jeps/320**
[9] "Java 9 Migration Guide: The Seven Most Common Challenges" - **https://blog.codefx.org/java/java-9-migration-guide/#Split-Packages**
[10] "Why is the Java 11 base Docker image so large?" - **https://stackoverflow.com/a/53383373**
[11] SonarJava" - **https://www.sonarsource.com/products/codeanalyzers/sonarjava.html#_**
[12] "Java 11 Support Preview is available in Jenkins 2.155+" - **https://jenkins.io/blog/2018/12/14/java11-preview-availability/**

CodeOne 2018

JavaOne in een nieuw jasje

Eind oktober 2018 was NLJUG aanwezig bij Oracle CodeOne: de grootste Java-conferentie ter wereld. Of moeten we tegenwoordig zeggen: de grootste *Code*-conferentie ter wereld? JavaOne heeft namelijk een nieuwe naam gekregen: CodeOne. De nieuwe naam duidt aan dat Oracle niet alleen Java, maar ook andere talen, technologieën en communities omarmt. Oracle organiseert CodeOne tegelijk met Oracle OpenWorld, waar zo'n 60.000 mannen en vrouwen uit zo'n 145 landen op afkomen.

Voor veel Java-ontwikkelaars is het bezoeken van of spreken op CodeOne toch wel een droom. Het is een congres van 4 dagen in San Francisco, met veel feesten en community-activiteiten, die er allemaal op gericht zijn om het voor ontwikkelaars een zo groot mogelijk feest te maken. Het is moeilijk om in zo'n omgeving niet geïnspireerd te raken. Je kunt er lekker eten op Fisherman's Wharf of fietsen door San Francisco. Het is een geweldige plek om community leaders, sprekers en gelijkgezinden te ontmoeten. Alles bij elkaar een evenement waar je toch minimaal één keer geweest moet zijn.

Locatie

CodeOne speelde zich voornamelijk af in het Moscone West gebouw, waar gedurende de vier conferentiedagen veel inspirerende talks en workshops gehouden werden. Opvallend is elke keer weer dat "Java" in de zin van koffie op deze conferentie heel slecht te krijgen was. Tijdens CodeOne en OpenWorld wordt San Francisco overspoeld met bezoekers voor de conferenties. Als gevolg daarvan zijn de hotelprijzen bizar hoog (500 dollar per nacht is niet ongebruikelijk). Een tip: boek met een groep collega's een Airbnb op loopafstand van de conferentie of in de buurt van een metrostation. Een stuk goedkoper en ook nog eens een stuk gezelliger!

Keynotes

Tijdens de technische keynote kreeg Mark Reinhold (Chief Architect van Java) flink wat spreektijd. Eén van de eerste dingen die hij zei was: "*Don't worry, Java's still free.*" Hiermee zette Mark duidelijk de toon. Verder vertelde hij over de toekomst van Java en de halfjaarlijkse release cycles. Java 11 is de LTS-versie (Long Term Support) en het wordt nu onderhand wel tijd om naar Java 11 over te gaan, was de boodschap.

Topics

Ivo: "Ik was erg geïnspireerd door de workshop over GraalVM. Ik had nog niet de tijd gehad om er serieus naar te kijken, en deze workshop (gegeven door Oleg Selajev en Chris Seaton) was goed genoeg om mij een goed beeld te geven van wat het kan en om mij te prikkelen om er zeker verder naar te gaan kijken. Cool stuff, zo te zeggen. Ook het spreken van alle bekenden en *like-minded people* in een relaxte setting is een van de grote pluspunten van CodeOne. Zo was de BBQ van Stephen Chin er eentje om te onthouden. Geweldige crowd en geweldig eten, wat wil je nog meer?"

Bert Jan: "Ik moest een aantal presentaties geven en had daardoor weinig tijd om zelf veel sessies bij te wonen. Op een conferentie als CodeOne probeer ik vooral zoveel mogelijk bij te praten met bekenden uit de Java-wereld. Gelukkig houden de meesten wel van een biertje ;-) Eén van de sessies, die me is bijgebleven, ging over Micrometer: een facade voor instrumentatie van metrics, die aansluit op populaire monitoring systemen. Zie het als een soort SLF4J voor metrics. Micrometer is standaard in Spring Boot 2; meer informatie vind je op **https://micrometer.io.**"

Rosanne Joosten is Software Engineer bij Blue4IT.

Ivo Woltring is Software Architect en Codesmith bij JTech Ordina.

Bert Jan Schrijver is CTO bij OpenValue en bestuurslid bij de NLJUG.

Rosanne: "Bij het uitzoeken van de sessies was er erg veel keuze. Het was soms moeilijk om te kiezen wanneer er meerdere interessante talks tegelijkertijd waren. Wanneer in de titel of in het abstract te vaak de term Oracle genoemd werd, wist je eigenlijk al wel dat je deze sessie kon skippen. Dit zou dan waarschijnlijk een verkooppraatje worden van een Oracle product. Gelukkig had ik dit zelf al wel van te voren bedacht en heb ik tijdens de hele conferentie maar bij 1 verkooppraatje weg hoeven lopen."

CodeOne kleurt oranje

Er was vanuit Nederland weer een flinke groep NLJUG-leden afgereisd naar CodeOne. Gedurende de week trokken we geregeld met zijn allen op: bij de keynotes, bij feestjes en dit jaar zelfs met een tour naar het kantoor van Google. Met een touringcar zijn we met zo'n 50 man naar de Googleplex afgereisd waar we van Kevin Nilson, een bevriende Googler, een tour over het terrein kregen. Je merkt dat we als groep aardig wat bekijks trekken. Ook tijdens de keynotes bleef de NLJUG niet onopgemerkt.

Tot slot

JavaOne / CodeOne is een indrukwekkende ervaring, die je als Java-developer echt een keer moet meemaken. Van de bredere focus van CodeOne ten opzichte van JavaOne was dit jaar nog niet heel veel te merken. Blijft de focus op Java ook de komende jaren behouden? De tijd zal het leren. Zolang dezelfde mensen blijven komen, blijft het in ieder geval een mooi feest! ■

Impressies van een first-timer

Toneelstukjes

Ik was voor het eerst bij Oracle CodeOne en voor het eerst in Amerika. Bij de keynotes merkte ik toch wel iets stereotypisch Amerikaans: de presentaties waren meer toneelstukjes of shows dan dat het een presentatie was. Wat er werd gezegd was inspirerend en grappig (bedoeld). Bij mij zorgden deze toneelstukjes voornamelijk voor kromme tenen.

Feestjes

Er waren genoeg feestjes... en die waren ook nog eens erg leuk. Soms is het moeilijk om jezelf eraan te helpen herinneren dat ik een iets kleiner persoon ben dan de gemiddelde feestganger op Oracle CodeOne feestjes. Gelukkig had ik genoeg beschermheren om me heen. De feestjes waren een uitstekende gelegenheid om te netwerken. Dat is natuurlijk altijd spannend. Maar je kunt de sprekers aanspreken op de toffe onderwerpen waar ze het over hebben gehad. Dat vonden ze vrijwel altijd leuk.

Het "Oracle CloudFest" in het AT&T Park (stadion van de San Francisco Giants) was heel groots opgezet. Ten eerste: het was in een stadion waar, zo is mij verteld, normaal gesproken geen feestjes of concerten worden gegeven. Daarnaast kwamen er een aantal, voor andere mensen, bekende artiesten optreden. Een concert van een goede artiest bijwonen is sowieso tof.

NLJUG op CodeOne

Het was leuk om met een hele club vanuit de NLJUG aanwezig te zijn. Bij elke gelegenheid waar we als groep waren, trokken we met z'n allen massaal ons oranje NLJUG-shirt aan. Toen we tussendoor naar een wedstrijd van Max Verstappen gingen kijken in een Sports Bar wilden er zelfs een aantal mensen foto's van "De Nederlanders" maken. Het groepsgevoel was bij mij zeker aanwezig. Het was leuk om nieuwe mensen te leren kennen met dezelfde interesses in een 'vreemd' land. Dat maakte voor mij de hele ervaring grootser dan hij al was.

Jakarta EE

De wederopstanding van Java EE

Voordat we dieper ingaan op de titel en het onderwerp van dit artikel, wil ik jullie graag even kort meenemen in een fictieve filmtrailer. Deze trailer vat samen wat er gebeurd is met Java EE. Ook vertelt het ons waar we nu staan met Jakarta EE als opvolger van Java EE onder het bewind van de Eclipse Foundation.

Java EE is al enkele decennia een hardwerkend en succesvol software framework. Recentelijk blijft echter zijn manier van werken achter op die van de concurrenten. Zijn klanten verliezen hun interesse, maar ook zijn werkgever: De Corporatie. Uiteindelijk wordt Java EE zelfs ontslagen en belandt hij op straat, met alleen zijn trots en ervaring op zak. Maar voordat hij volledig van het toneel verdwijnt, wordt hij gevonden door leden van 'De Onafhankelijken', die hem in een artificieel coma zetten en vervolgens oplappen. Ze integreren Java EE geheel in de orde van 'De Onafhankelijken'. Dan is het moment daar. De Onafhankelijken ontwaken hem en vragen hem: *"Mr. Jakarta EE, kunt u ons horen?"*. En het legendarische antwoord van de als Jakarta EE herrezen Java EE is: *"I'm back, with a mission!"*.

Hiermee is de trailer voor Jakarta EE's verhaal klaar. Hoe de film eruit gaat zien en wat daarin allemaal gaat gebeuren, moeten we afwachten tot de eerste release. Echter, we kunnen wel al speculeren. Ik zal jullie daarvoor in dit artikel voorzien van de feiten en visies die nu bekend zijn. Dit om een zo concreet mogelijk beeld te schetsen van waar de Eclipse Foundation met Jakarta EE naar toe wil. Pak je laptop, cola en popcorn; en geniet van de 'film'!

De eerste release van Jakarta EE

Het moment van schrijven van dit artikel is december 2018. Dat vermeld ik er expliciet bij, omdat de evolutie van Jakarta EE waarschijnlijk sneller gaat dan het publiceren van dit artikel. Desalniettemin zal ik jullie van de status voorzien die op dit moment waarheid is en vervolgartikelen schrijven in volgende edities met verse updates.

Migratie

De belangrijkste update is dat de migratie van de Java EE codebase vanuit Oracle nog wordt afgerond in 2019, aldus de Jakarta EE werkgroep bij de Eclipse Foundation. Daarmee zijn alle specificaties ondergebracht in een projectstructuur die gehandhaafd wordt bij Eclipse. Wil je dit proces 'live' volgen? Kijk dan op de status pagina die vermeld is in de referenties.

Java EE 8 Compliant

Ondanks dat officieel nog geen versie van Jakarta EE is gereleaset, worden hier al wel talks over gegeven op conferenties, door bijvoorbeeld Adam Bien. Uiteraard kon ik het niet laten om aan te schuiven bij dergelijke talks om Jakarta EE voor het eerst live te zien. Aangezien ik al behoorlijk op de hoogte ben van alles dat speelt, was ik hier wel wat sceptisch over. En terecht, want deze talks bleken uiteindelijk gewoon te gaan over Java EE en niét over Jakarta EE.

Eigenlijk is dit geniaal, want dit bewijst dat de marketingmachine van Jakarta EE wel zijn werk doet. Men komt hier mee weg, omdat de eerste versie van Jakarta EE simpelweg een Java EE 8 compliant versie is, met enkel een splinternieuwe naam en logo.

Jakarta EE Applicatieserver

Concreet betekent dit dus dat er een Java EE 8 compliant applicatieserver moet komen die álle referentie implementaties van de voormalige Java EE 8 specificaties moet bevatten.

Edwin Derks is Software Architect en CodeSmith bij Ordina JTech en heeft naast Java een passie voor cloud-driven development en serverless architecture.

JAKARTA EE

Eclipse GlassFish

Niet onverwacht is de voormalige GlassFish applicatieserver, die nu Eclipse Glassfish heet, hier logischerwijs voor uitverkoren. Zodra deze stabiel bevonden is, zal Eclipse GlassFish 5.1 gereleaset worden, waarmee we daadwerkelijk aan de slag kunnen met Jakarta EE.

Technology Compatibility Kit

Er is nog een opmerkelijke update te noemen: de Technology Compatibility Kits (TCK) zijn ook open source geworden. Hiermee test je of je implementatie van een specificatie voldoet aan alle eisen. Dit is opmerkelijk, omdat deze voorheen diep waren weggestopt in de catacomben van Oracle. Hier kon je alleen gebruik van maken tegen betaling van een fiks bedrag. Op deze manier kon eigenlijk niemand (gratis) inzien wat de tests in de TCK's nu daadwerkelijk inhielden. Dit maakte het aftesten van een referentie implementatie behoorlijk ingewikkeld. Vooral omdat het aanvechten of aanpassen van de tests een lang en bureaucratisch proces was. In mijn opinie een geniale zet van Eclipse om ook deze open source te maken, zodat iedereen deze tests kan inzien en kan verbeteren.

De Jakarta EE roadmap

Na deze feiten op een rijtje te hebben gezet, ben ik gaan onderzoeken in hoeverre de roadmap van Jakarta EE al is uitgestippeld en wat al in gang is gezet voor een volgende release. Toen ik als spreker aanwezig was op de Java2Days conferentie eind november 2018, heb ik van de gelegenheid gebruik gemaakt de aanwezige Jakarta EE werkgroepleden hierover het figuurlijke hemd van het lijf te vragen. Helaas bleek dit niet erg vruchtbaar, omdat zij het simpelweg ook nog niet weten. Met de informatie die ik wél heb verkregen, kan ik in ieder geval voor jullie een concreet overzicht maken.

Jakarata EE versionering

De kans is groot dat het versienummer formaat <<platform release>>.<<feature release>> gaat zijn, zoals dat ook bij Eclipse MicroProfile gehanteerd wordt. Het eerste versienummer van Jakarta EE als Jakarta EE 8 of Jakarta EE 8.0 zal naar verwachting gebruikt worden voor de release van GlassFish 5.1.0.

Nieuw Specificatie Proces

Onder het bewind van Oracle werden nieuwe specificaties behandeld door de JCP. Afgezien van het feit dat dit een tijdrovend proces was, werd hier vaak een specificatie bedacht voor wat men dacht dat een goede aanvulling zou zijn op Java EE. Er werd dan een specificatie uitgewerkt met een referentie implementatie. Daarna werd besloten of deze mee kon in een volgende Java EE release.

IEDEREEN WORDT NADRUKKELIJK OPGEROEPEN OM JEZELF AAN TE MELDEN BIJ DE JAKARTA EE MAILING LIST OF ZELFS ALS MEMBER VAN DE ECLIPSE FOUNDATION

De Eclipse Foundation wil hier vanaf. Zij zien er meer heil in dat een tool is gemaakt door iemand vanuit een behoefte, waar dan een specificatie uit gedestilleerd kan worden. Deze wijziging moet de evolutie van Jakarta EE sneller en beter op de behoefte van de community laten passen. Overigens stappen ze ook af van referentie implementaties. Een specificatie moet één of meer productierijpe implementaties bevatten (bijvoorbeeld het originele project), maar geen officiële referentie implementatie voor een applicatieserver.

Binnen de Eclipse Foundation heeft een project zich kandidaat gesteld om dit nieuwe proces te testen: JNoSQL. Dit is een NoSQL oplossing voor Jakarta EE, want die hebben ze nog niet. Hopelijk kunnen we deze gaan gebruiken in een eerstvolgende Jakarta EE release.

De toekomst van cloud-native Java

De marketing van Jakarta EE predikt de toekomst van cloud-native Java. Maar wat betekent dit nou precies? Een mooie vraag van mij aan de werkgroepleden, dacht ik zelf, maar daar kreeg ik geen concreet antwoord op. Wederom omdat ze dit zelf nog niet weten. Dit concept ligt nu nog te ver open om hier concreet een richting aan te geven. Ik verwacht dat hier meer duidelijkheid in komt nadat versies van Jakarta EE gereleaset zijn, omdat de community dan wat meer ervaring hiermee heeft.

Jakarta EE zelf gebruiken

Als je op het punt staat om toch eens te gaan experimenteren met Jakarta EE, kun je daar nú al mee aan de slag. Zoals ik had vermeld, zijn Java EE 8 en Jakarta EE 8 in eerste instantie compatible en kun je dus nu al beginnen te bouwen op applicatieservers die deze specificaties implementeren.

Maar wil je dat wel? Want we weten allemaal dat applicatieservers log zijn; 'traag' bij opstarten en 'gigantisch' van formaat. Microservice frameworks, zoals Thorntail, die bijvoorbeeld MicroProfile implementeren bouwen UberJAR's. Die starten toch veel sneller op en hebben een veel kleinere memory footprint. Toch?

Nou, volgens mij hebben we inmiddels deze mythe wel doorbroken en weten we dat deze stelling niet per definitie waar is. En het concept van een applicatieserver is ook niet per definitie ouderwets. Naar mijn mening is dit concept altijd al geniaal geweest en kan Jakarta EE daarmee nog prima uit de voeten. Jakarta EE is zelfs per ongeluk al een beetje cloud-native. Voordat je mij direct belt voor uitleg, zou ik dit willen toelichten. Het zit namelijk zo.

Framework-loos programmeren

Een Jakarta EE applicatie die je bouwt als een WAR, is een verlengde van de applicatieservers die de specificaties implementeren. Hierdoor kun je direct gaan ontwikkelen en bouwen op je favoriete applicatieserver. Deze bevat vanwege de lange lijst aan geïmplementeerde specificaties ook ongeveer alles wat je nodig hebt voor je project.

Overigens weet je met een applicatieserver precies waar je aan toe bent qua libraries, modules en configuratie. Daarmee dus ook de versies hiervan, waaraan je kunt ontleden of deze up-to-date genoeg zijn qua security en functionaliteit die je verwacht. Probeer dit maar eens zo makkelijk duidelijk te krijgen met bijvoorbeeld Spring (Boot)... succes!

Containers

Je kunt Docker images verkrijgen voor de populaire applicatieservers zoals Payara, Wildfly en OpenLiberty. Toegegeven, deze zijn een paar honderd MB groot vanwege de applicatieserver die daarop staat. Maar is dat tegenwoordig een probleem? In de context van Docker kan dit zelfs een voordeel zijn. Denk even mee in het volgende scenario.

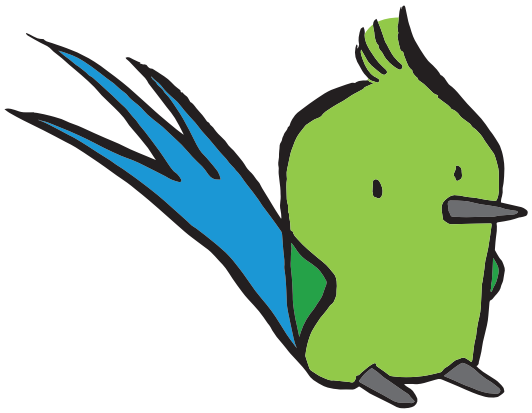
Je bouwt een Jakarta EE app (WAR) op een Docker image die aftakt van een Payara image. Deze WAR is, vanwege het programmeren tegen specificaties aan, maar een paar KB groot en is één enkele laag van je Docker image.

Daarna push je de gebouwde Docker image naar een repo. De eerste keer zal de volledige Docker image overgedragen worden. Een volgende push met een update van je code zal je alleen deze laag van de Docker image doen uploaden en dat is maar een paar KB. Als je dit vaak moet doen, hoef je maar steeds een paar KB te uploaden, dit in tegenstelling tot de vele MB's die je steeds moet uploaden als je dit proces volgt met een UberJAR.

Relatie met Eclipse MicroProfile

Volgens mij is de vraag die veruit het meest is gesteld in de Jakarta EE community; of

DE MIGRATIE VAN DE JAVA EE CODEBASE VANUIT ORACLE WORDT AFGEROND IN 2019



THORNTAIL

MicroProfile samen gevoegd gaat worden met Jakarta EE. MicroProfile is namelijk afgetakt van Jakarta EE, maar gaat waarschijnlijk een heel andere rol als incubatie project krijgen met een eigen release cadans. Als de cadans echter gelijk gaat lopen met Jakarta EE, dan heeft dat uiteindelijk misschien weer geen zin en zullen ze wel mergen. Hoe dan ook, ik laat het jullie graag weten als hierover een besluit is genomen door de leden van de werkgroep.

Zit je intussen met het dilemma of je beter met MicroProfile aan de slag kunt, omdat je microservices wilt bouwen? Dit probleem wordt door eerdergenoemde applicatieservers opgelost. Deze implementeren namelijk niet alleen de Jakarta EE specificaties, maar óók MicroProfile. Dat betekent dat je in de WAR die je bouwt, compleet naadloos gebruik kunt maken van beide specificatie sets. Het enige wat je hoeft te doen in je project, is zowel de Jakarta EE als MicroProfile BOM als provided op te nemen in je Maven pom.xml.

Toekomstvisie

Hoewel de toekomst van Jakarta EE nog wijd open ligt, hebben de leden van de werkgroep wel wensen die zij graag realiteit zien worden. Om te beginnen willen ze echt een open community creëren, waardoor input daadwerkelijk uit de community komt en niet enkel vanuit een vendor. Ze roepen dan ook nadrukkelijk op om jezelf aan te melden bij de Jakarta EE mailing list of zelfs als member van de Eclipse Foundation. Mocht je interesse hebben om het platform waar je mee gaat werken ook mee te bouwen, schroom dan niet en meld je aan.

Ze hopen zelfs dat verschillende specificaties en implementaties niet alleen van binnen, maar ook buiten de Eclipse Foundation worden gerealiseerd om op die manier verschillende platformen te kunnen bouwen, allen gebaseerd op Jakarta EE. Hierdoor heeft zelfs Microsoft zich als contributor aangemeld. Dan kun je ervan uitgaan dat we straks Jakarta EE applicaties kunnen bouwen die perfect te hosten zijn op Azure. Op deze manier verhelp je ook het probleem dat je vendor lockin zou kunnen krijgen.

Verder dromen ze van een release per kwartaal met niet alleen updates, maar ook nieuwe specificaties die een vraag voor het bouwen van cloud-native software oplossen. Denk hierbij bijvoorbeeld aan CQRS, big data en reactive functionaliteit. Als dat ook onafhankelijk kan van een specifieke Java versie, zonder backwards compatibility te breken, zou dat helemaal mooi zijn.

Conclusie

Jakarta EE lijkt klaar voor de toekomst. Maar hoe die eruit gaat zien, daar kunnen we nu nog weinig over zeggen. Daarvoor zullen we de ervaring van één of meerdere releases nodig hebben, naast een meer concrete visie van de werkgroepleden. Men kan in ieder geval met deze herrezen manier van werken met applicatieservers al wel de voordelen van Jakarta EE aanwijzen, maar daar zal de community wel aan moeten wennen. Zoals ik het zie, kun je prima zowel monolieten als microservices bouwen met Jakarta EE, zeker in combinatie met MicroProfile.

Kunnen we hiermee concluderen dat het gebruik van UberJAR's voor het concept van 'rightsizing' je applicatie niet helemaal de potentie waarmaakt? Is Jakarta EE daarmee de start van een nieuw tijdperk voor niet alleen het bouwen, maar ook het deployen van cloud-native Java software, inclusief microservices? We zullen het meemaken, maar het is op zijn minst weer interessante gespreksstof. ■

JAKARTA EE LIJKT KLAAR VOOR DE TOEKOMST. MAAR HOE DIE ERUIT GAAT ZIEN, DAAR KUNNEN WE NU NOG WEINIG OVER ZEGGEN

REFERENTIES

https://jakarta.ee
https://microprofile.io
https://www.eclipse.org/ee4j/status.php
https://jakarta.ee/news/2018/10/16/introducing-the-jakarta-ee-specification-process/
https://thorntail.io/
https://jcp.org/en/home/index
http://www.jnosql.org/


SMART ISN'T SMART ENOUGH FOR YOU?
#WORKYOURTALENTS
werkenbijabnamro.nl
BOSCH
ABN·AMRO

JFall 2018.

De aftrap van de J-Fall 2018 werd dit jaar gedaan door de drummers van 'Hot drum Slagwerk' met een live optreden op het podium. Dit was een goeie start van de dag! Verder was er een dagprogramma vol gevarieerde sessies over de nieuwste technologieën. Het zou een veelbelovende dag worden.

Het volle programma barstte vroeg in de ochtend los. Om 08:00 uur werd er van start gegaan met sessies over het bouwen van apps met 'NativeScript' en 'Reactive Streams'. Dit werd gevolgd door Key note speakers over 'Cloud Foundry', 'front-end development' en 'Quantum Computing'.

Het was dit jaar anders dan anders. Dit jaar was ABN Amro co-sponsor. Er waren verschillende stands van onder andere Malmberg, Capgemini, IBM, Simacan en Vijfhart. Bij de stand van ABN Amro werden "talk nerdy to me" shirts uitgedeeld. Een populair item: zelfs medewerkers van de Rabobank liepen met een shirt naar buiten aan het einde van de dag. Een goede score.

De stands waren druk bezocht en men kwam niet alleen voor de goodies. Het was een plek om te discussiëren over de nieuwste technologieën. Zo kwamen collega's vragen stellen bij de ABN Amro stand over Tikkie: één van de famous projects. Gelukkig waren er twee collega's aanwezig die beiden werken in het scrum team van Tikkie en zij konden alle vragen beantwoorden. Er kwamen collega's met ideeën over verbeteringen voor de app en vragen stellen over de developer portal. Naast de geplande sessies, waren de stands ook een plek om van elkaar te leren.

J-Fall blijkt dus meer te zijn dan alleen kennissessies over programmeertaal. Het is een dag waarbij men bij elkaar komt om te socializen, kennis over te brengen en te leren van anderen. Er waren ontzettend veel Java-enthousiastelingen in één ruimte waar ook de gepassioneerde Java-ontwikkelaars tussen zaten. Het is een evenement met sessies van introducerende tot high-level onderwerpen met praktische voorbeelden die kunnen helpen bij het dagelijkse werk.

Zo was Venkat Subramaniam één van de favorieten. Ook wel de Java-goeroe genoemd onder de key-note speakers. Hij vertelde over Java Functional Programming Idioms met zijn eigen kenmerkende presentatiestijl. Venkat weet namelijk als geen ander op een humoristische wijze veel voorkomende problemen aan te pakken. Op deze manier onthoud je als luisteraar zijn verhaal veel beter en kan je deze problemen herkennen wanneer je ze in je eigen codebase ziet.

Als bezoeker van de J-Fall leer je meer collega's van de bank kennen: een mooie bijkomstigheid van deze dag. Dit zorgt er naast de interessante sessies voor dat collega's allemaal enthousiast zijn over het bezoeken van de volgende J-Fall in 2019. Het komt dus goed uit dat er eerst nog andere mooie evenementen op de planning staan. J-Spring en TEQNation hebben zeker de interesse gewekt bij onze bezoekers. Voor nu kan iedereen J-Fall 2018 in zijn geheel terugkijken via YouTube.

Graag willen we NLJUG bedanken voor de organisatie en de (altijd) spectaculaire opening van het evenement. Laten we koers zetten naar het volgende evenement, TEQNation: we komen eraan! ■

San Francisco – Oracle Code One – 22 t/m 25 oktober 2018

Onlangs heb ik de Oracle CodeOne mogen bijwonen. Omdat een plaatje vaak meer zegt dan duizend woorden heb ik alle sessies die daar werden gehouden gebundeld in een WordCloud. Hierin is goed zichtbaar wat de hoofdthema's waren en wat de toekomst ons brengt. Dus als je je afvraagt waar je je in de toekomst op zou kunnen richten … Vijfhart biedt ook diverse cursussen aan op dit gebied. Naast programmeercursussen zoals Python, Java, en JavaScript kun je ook cursussen vinden op het gebied van Frameworks zoals Spring, Spring Boot en Hibernate, maar ook container- en cloud-infrastructuur en Orchestration, zoals Docker en Kubernetes.

Wil je op de hoogte blijven op het gebied van Java-ontwikkelingen? Schrijf je dan in voor de nieuwsbrief van Vijfhart op **www.Vijfhart.nl/aanmelden-nieuwsbrief**. ■

vijfhart

Oracle Code 4 Kids

Voor deze editie van Oracle Code One zijn JDriven collega's Michel en Jeroen gevraagd om een sessie te verzorgen. En dit keer wel een bijzondere in het kader van Oracle Code 4 Kids. Een hele mooie manier om bij te dragen aan de toekomstige generatie (Java) developers. In Nederland verzorgen we dit soort sessies vaker, dus waarom ook niet op Code One? Met als grootste praktische uitdagingen het vervoeren van de mBots naar San Francisco. Goed om te weten; er passen er exact twaalf in een koffer.

Op Code One hebben we voor 60 kinderen twee sessies verzorgd waarbij ze zelf de mBots mochten programmeren. Ze hebben verschillende opdrachten gedaan waaronder het op afstand besturen, lijnen laten volgen en zelf kijken wat er nog meer mogelijk is (zonder de handleiding te lezen ;)). Vooral dat laatste was erg boeiend en liep in sommige gevallen op een prettige manier uit de hand.

Mocht je meer willen weten over deze sessies en het materiaal dat we hebben gebruikt? Lees dan het volledige verslag op onze website: **https://www.jdriven.com/events/conferentie/oracle-codeone-2018/**

Het was een geweldige en intensieve ervaring om zoveel kinderen enthousiast te maken voor ons fantastische vak! ■

Jeroen Resoort

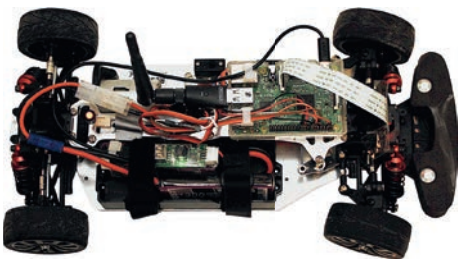
Michel Breevoort

jdriven


Het
Capgemini
Effect
Capgemini
Innovatie brengt organisaties verder als het ook echt werkt.
Hoe ontstaat het Capgemini effect? Samen met onze klanten brengen we in kaart waar innovatie op de juiste manier toe te passen is. We slaan een brug tussen opdrachtgevers en een ecosysteem van start-ups en strategische technologie partners. Zo zorgen we ervoor dat innovatie niet beperkt blijft tot een idee, maar de onderneming echt verder brengt.
Ondek het Capgemini effect
www.capgemini.nl

JPoint wint Duke's Choice Award!

Op 24 oktober heeft JPoint een Duke's Choice Award gewonnen voor de ontwikkeling van een zelfrijdende auto met Java.



De Duke's Choice Awards, de 'Oscars van de Java-wereld', werden uitgereikt tijdens CodeOne in San Francisco, de grootste Java-conferentie ter wereld. De award, die wordt toegekend aan innovatieve toepassingen met Java, werd uitgereikt door Georges Saab, vice president software development van de Java Platform Group.

JPoint mocht deze prestigieuze award in ontvangst nemen voor het resultaat van de (zelf georganiseerde) "RoboRace challenge": een wedstrijd waar 4 teams een kleine RC (remote controlled) auto krijgen en met een beperkt budget en slimme software de auto zelfrijdend moesten maken. Het winnende team heeft haar kennis en ervaringen gedeeld op een aantal internationale conferenties (waaronder Devoxx België, UK en Polen) en daar, naast de techniek, ook onderwerpen als moraliteit en ethiek bij zelfrijdende auto's besproken.

Het uitgangspunt voor de challenge was een "RC car kit": alle teams kregen een basisframe van een radiografisch bestuurbare auto met wielen, een motor, een stuurinrichting en een accu. Vervolgens was het aan de teams om, met een beperkt budget voor sensoren en andere hardware, de auto zo goed mogelijk autonoom te laten rijden. JPoint gebruikte hiervoor Java, Vert.x (een reactive toolkit voor de JVM), OpenCV (een open source library voor computer vision) en een Raspberry Pi met camera.

De Duke's choice award viert extreme innovatie met Java-technologie. Het primaire beoordelingscriterium voor deze prestigieuze award is innovatie. Daarmee worden kleine softwarebedrijven en individuele ontwikkelaars op gelijke voet gesteld met multinationals.
Iedereen kan projecten en personen nomineren, zolang het gerelateerd is aan innovatie rondom Java. Andere winnaars dit jaar waren onder andere Netbeans en MicroProfile. ■

jpoint

Hier maak je echt het verschil

Stel je eens voor...

Rabobank

Growing a better world together.

2018 MASTERS OF JAVA nl.jug

POWERED BY FIRST8










J-Fall 2018

J-Fall 2018 was weer een groot succes!
Op naar J-Spring. Save the date: 29 mei 2019!
www.jspring.nl

Deeplearning4j

Deep Learning op de JVM

Deep Learning en AI zijn al jaren meer dan alleen buzz words. Deep Learning ondersteunt een groot aantal van de services, die we dagelijks gebruiken om het leven makkelijker of leuker te maken. Het is dan ook merkwaardig dat AI en Deep Learning vooral worden toegepast in Python. Is er een toekomst voor Deep Learning op de JVM? En hoe ziet dat er dan uit?

Deeplearning4j is een open source Deep Learning library voor Java en de JVM. Deeplearning4j biedt een moderne API aan om neural networks op te bouwen en te trainen in een gedistribueerde omgeving. Hierbij wordt gebruikt gemaakt van Apache Hadoop en Apache Spark. In 2017 is het open source project gedoneerd aan de Eclipse Foundation en commerciële support wordt geleverd door de startup Skymind.

Deep Learning is een vorm van Machine Learning, die gebruikt wordt voor het werken met neurale netwerken (neural networks). Een neural network is geïnspireerd op de werking van het brein en bestaat uit een aantal lagen aan neuronen (ook wel neurons of nodes). Deze techniek wordt veel gebruikt voor classificatie. Wanneer een neural network ingezet wordt voor de classificatie van data, dan wordt deze data gekoppeld aan een bepaalde groep (ook wel label). De classificatie van de data wordt beïnvloed door patronen, die herkend zijn uit sample data waarmee het netwerk opgezet is. Deep Learning gaat specifiek over diepe neural networks, waarbij altijd 1 input laag, 1 output laag en 1 of meerdere "hidden" lagen zijn. Een neural network kan aan de hand van voorbeelden leren hoe de relaties tussen de neurons/nodes geconfigureerd moet worden om zo dicht mogelijk bij het gewenste antwoord uit te komen.

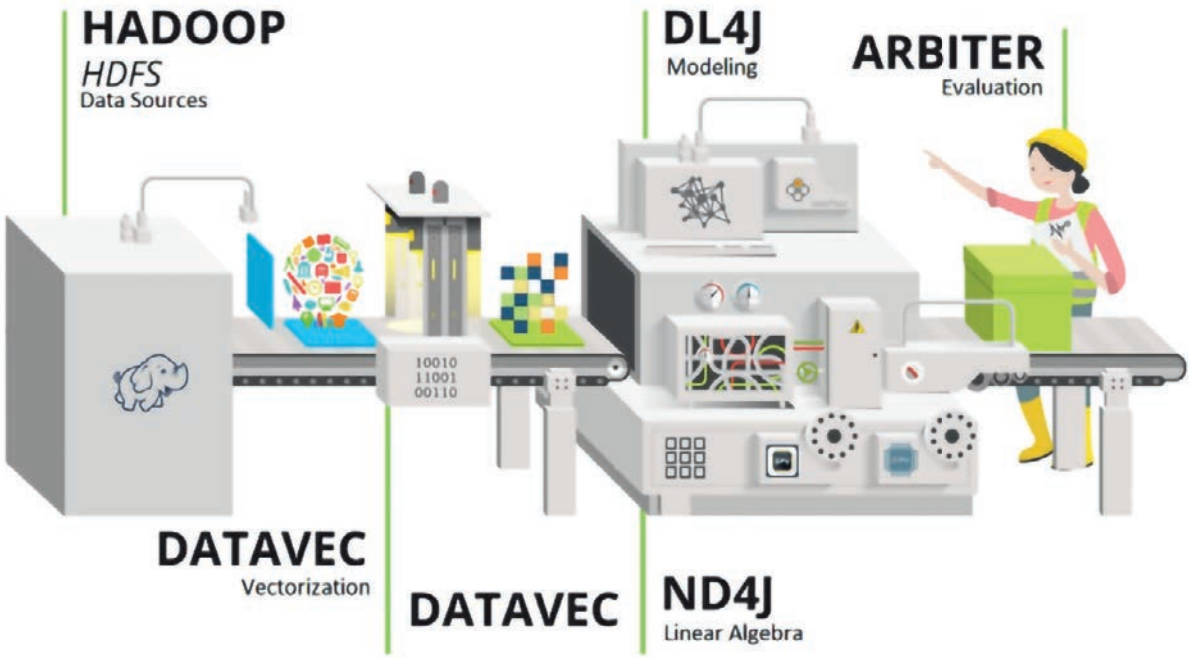
De maintainers van Deeplearning4j zijn zich er van bewust dat data scientists veelal gebruik maken van Python, met behulp van een van de vele Python machine learning frameworks. Dit komt omdat de meest bekende frameworks voor machine learning voor Python ontwikkeld zijn, en de taal relatief makkelijk te begrijpen is in vergelijking met alternatieven. Daar tegenover gebruiken de meeste enterprise Big Data applicaties Java en vaak zijn bedrijven afhankelijk van het Java ecosysteem om consistente instellingen, monitoring en security toe te passen. Om er voor te zorgen dat deze partijen toch met elkaar kunnen samenwerken, is het mogelijk om een model te importeren vanuit Keras, een high-level neural network API voor Python. Keras draait die modellen bovenop TensorFlow, CNTK of Theano. Als je een model uit Keras importeert, is het mogelijk om het te trainen op een Hadoop cluster via Spark. Bij het trainen van een netwerk gebruik je voorbeelden uit een datasource, die al geclassificeerd zijn om het netwerk te kalibreren. Daarna kun je het in productie uitvoeren op bestaande Java technologieën, die veel worden gebruikt in de enterprise wereld.

Met Deeplearning4j is het dus mogelijk om een netwerk zowel in Java als in Python te configureren en te prototypen. Deeplearning4j kan vervolgens toegepast worden om het uit te rollen in productie of gedistribueerd te trainen op een cluster.

Hoe Deeplearning4j werkt

De API van Deeplearning4j is beschikbaar voor Java en Scala. De API van Deeplearning4j bestaat uit een aantal builders voor het samenstellen van neural network configuraties en implementaties voor de veelgebruikte algoritmes en functies in Deep Learning. Verder heeft Deeplearning4j nog een aantal componenten, waaronder een evaluatie API om de nauwkeurigheid van een netwerk te

Lennart ten Wolde is een gepassioneerde software engineer bij Quintor. Door zijn passie gedreven heeft hij een brede kennis opgebouwd van moderne technieken waarbij hij zich richt op toepasbaarheid en werking daarvan.



berekenen, DataVec voor het serializeren van data naar vectors (meer hierover later), Arbiter voor het automatisch optimaliseren van netwerk parameters en Word2Vec voor het verwerken van tekst.

Supported platforms en performance

Het eerste vraagstuk waar je wellicht tegenaan loopt, gaat over de beperkingen van N-dimensionale arrays in Java. Helaas zit er een limiet aan het aantal waardes in een N-dimensionale array. Deze limiet is te laag voor het uitvoeren van matrix-operaties op neurale netwerken met vele en grote lagen. Daarnaast is het uitvoeren van deze berekeningen niet snel. Om dit probleem op te lossen, heeft Deeplearning4j een onderliggende wiskunde library in het leven geroepen, genaamd ND4J.

ND4J is een Java API voor wetenschappelijke berekeningen met N-dimensionale arrays. De syntax emuleert die van Numpy en MATLAB, twee interfaces die veel worden gebruikt voor wiskundige berekeningen. ND4J wordt gebruikt voor het uitvoeren van lineaire algebra en grootschalige matrix berekeningen. Het nd4j-api component heeft een aantal implementaties. Op dit moment is er een C++ en CUDA (NVIDIA GPU) backend en in de toekomst zal er waarschijnlijk ook een backend zijn voor OpenCL en Power8. ND4J is snel en als resultaat biedt Deeplearning4j competitieve snelheid in vergelijking met andere bekende machine learning libraries. Op de site van Deeplearning4j kun je instructies vinden voor het runnen van benchmarks en het tunen van de performance.

DataVec

DataVec is Deeplearning4j's oplossing voor het omgaan met data. Neural networks werken exclusief met vectoren, een genormaliseerde reeks aan decimalen, die de input data representeren. Als je data in CSV-formaat is opgeslagen of een folder met gelabelde plaatjes, dan kan DataVec ervoor zorgen dat deze data wordt omgezet in vectoren waar je neurale netwerk mee kan werken. DataVec kan ook worden gebruikt voor de visualisatie en analyse van datasets. Het is bijvoorbeeld mogelijk om een HTML-analyse of plot te genereren voor een DataVec schema.

Mobile (Android)

Meestal wordt het trainen van een neuraal netwerk uitgevoerd op krachtige computers met veel GPU's. Dit komt omdat bij het trainen vaak grote aantallen aan voorbeelden in batches geëvalueerd worden, waarmee vervolgens het netwerk gekalibreerd wordt. Maar het is ook mogelijk om netwerken te gebruiken of trainen op smartphones. Omdat Deeplearning4j in Java geschreven is, kan het gebruikt worden op een Android device om voorspellingen te doen met een getraind of ongetraind netwerk. Op de site van Deeplearning4j staan voorbeelden voor het ontwikkelen van een afbeeldingsclassificator, waarbij volledig op de achtergrond en zonder inter-

DE API VAN DEEPLEARNING4J IS INTUÏTIEF EN VOELT FIJN AAN ALS JAVA DEVELOPER

netverbinding voorspellingen kunnen worden uitgevoerd met Deeplearning4j.

Deeplearning4j in actie

DeepLearning4j biedt een builder API aan voor het configureren van een neural network setup. Vervolgens kun je dit netwerk trainen en voorspellingen doen voor bepaalde waarden. Hierbij produceert het netwerk een aantal labels en voor elke label een waarschijnlijkheid dat de inputwaarde onder deze label valt. Het aantal labels is dus gelijk aan het aantal output nodes van een netwerk.

Met de NeuralNetConfigurationBuilder, kan je een aantal globale netwerkinstellingen aanpassen en standaardwaardes uitkiezen voor de lagen uit het netwerk. In dit geval is de standaard activation function geconfigureerd als RELU. Wat RELU precies doet is hier niet heel erg relevant. Vervolgens worden de lagen van het netwerk gespecificeerd. Een invoerlaag is altijd present; de grootte hiervan is gelijk aan de nIn waarde van de eerste hidden layer. De eerste hidden layer wordt geïnitialiseerd met een Dense Layer en de laatste laag als Output Layer. Elke laag uit het netwerk heeft zijn eigen builder, met instellingen die afhankelijk zijn van het soort laag. In de output laag wordt de activation function aangepast naar Softmax. Hierbij wordt de standaardwaarde van RELU overschreven. Als laatst wordt met deze configuratie een nieuw netwerk opgebouwd en geïnitialiseerd.

```
MultiLayerConfiguration configuration = new NeuralNetConfiguration.Builder()
    .activation(Activation.RELU) Builder
    .list() ListBuilder
    .layer(new DenseLayer.Builder()
        .nIn(10).nOut(1000)
        .weightInit(WeightInit.XAVIER)
        .build()) ListBuilder
    .layer(new OutputLayer.Builder()
        .nIn(1000).nOut(5)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .build()) ListBuilder
    .build();

MultiLayerNetwork network = new MultiLayerNetwork(configuration);
network.init();
```

Listing 1

```
DataSetIterator trainingData = new RecordReaderDataSetIterator(dataSourceRecordReader,
    batchSize: 50,  labelIndex: 0,  numPossibleLabels: 10);
network.fit(trainingData);
```

Listing 2

```
INDArray inputData = new ImageLoader().asRowVector(new File( pathname: "inputImage.png"));
INDArray prediction = network.output(inputData);
int bestPredictionLabelIndex = prediction.argMax(0).getInt( ...indices: 0);
float bestPredictionConfidence = prediction.max(0).getFloat( i: 0);
```

Listing 3

In **Listing 2** wordt het netwerk getraind met voorbeelden. Dit wordt gedaan door het netwerk te fitten op de data. In de praktijk wordt dit vaak in meerdere iteraties gedaan. Het doel is om een hoge nauwkeurigheid te behalen op de voorbeelden. Uiteraard kun je met Deeplearning4j ook een bestaand netwerk evalueren. Bij het evalueren wordt vaak andere data gebruikt dan bij het trainen om te valideren dat het netwerk ook goed om kan gaan met data, die het nog niet eerder gezien heeft. Het kan namelijk zijn dat een netwerk heel erg goed werkt op de gegeven voorbeeld data, maar niet de juiste patronen herkent om onbekende data te kunnen classificeren. Dit heet "*overfitting*". Omdat je niet veel inzicht hebt op hoe de relaties in een netwerk precies in elkaar zitten, is het erg lastig om de juiste verhouding hierin op te zoeken. Deeplearning4j biedt de mogelijkheid om Early Stopping toe te passen, waarbij het netwerk tussen training iteraties door geëvalueerd wordt om op overfitting te controleren en de training vroegtijdig te stoppen.

In **Listing 3** wordt Deeplearning4j's DataVec ImageLoader gebruikt om een plaatje om te zetten in een vector, die gebruikt kan worden als inputwaarde voor het netwerk. Vervolgens wordt er een voorspelling gedaan voor het plaatje en de uitkomst hiervan is een nieuwe vector met de waarschijnlijkheid dat de input onder de gegeven label valt. Uit deze vector kun je de index van de meest waarschijnlijke label en de bijbehorende waarde ophalen. Deze waardes heten in het voorbeeld bestPredictionLabelIndex en bestPredictionConfidence. De confidence geeft aan hoe zeker het netwerk is dat de input onder het gegeven label valt.

Naast het handmatig opbouwen van een configuratie voor een netwerk is het ook mogelijk om een netwerk te importeren vanuit Keras. Deeplearning4j kan modellen importeren uit Keras HDF5 bestanden met model en training gegevens, of losse tekstbestanden met JSON netwerk configuratie. Importeren vanuit Keras staat uitgebreid uitgelegd op de site van Deeplearning4j.

Deeplearning4j & Spark

Voor complexe netwerken met extreem veel data, kan het handig zijn om met Deeplearning4j gedistribueerd te trainen. Dit kan met Apache Spark. Deeplearning4j ondersteunt neural network training op een cluster aan

VOOR COMPLEXE NETWERKEN MET EXTREEM VEEL DATA, KAN HET HANDIG ZIJN OM MET DEEPLEARNING4J GEDISTRIBUEERD TE TRAINEN

CPU- of GPU-machines met Apache Spark. Spark is een analytics engine voor grootschalige geclusterde verwerking van data. Spark kan bovenop Hadoop, Amazon EC2, YARN, Mesos of Kubernetes draaien en data verwerken uit HDFS, Cassandra, HBase, Hive en vele anderen.

Apache Spark maakt het mogelijk om dataverwerking op zeer grote schaal efficiënt en snel uit te voeren op een cluster. Het wordt door velen gebruikt voor analyse op Big Data Warehouses. Deeplearning4j kan deze technologie gebruiken om aan te sluiten op je bestaande databronnen en hierop trainen in parallel, op een bestaand cluster. Voor bestaande gebruikers van deze softwareoplossingen is dit een hele makkelijke manier om grootschalige deep learning mogelijk te maken. Het is wel belangrijk om af te wegen of het gebruiken van een gedistribueerde training setup nodig is voor een specifiek netwerk. Dit hangt af van de complexiteit van het netwerk en de beschikbare data om mee te trainen.

Deeplearning4j's integratie met Apache Spark maakt gebruik van Resilient Distributed Datasets (RDD's). Een RDD is een fout-tolerante collectie aan elementen, die in parallel verwerkt kunnen worden in een cluster. Het doel van een RDD is om een high-level API beschikbaar te stellen, zodat het makkelijker is voor de programmeur om te focussen op het algoritme in plaats van het managen van gedistribueerde systemen. Deeplearning4j kan data uit RDD's ophalen en in parallel verwerken met een Spark job.

In het bovenstaande minimale voorbeeld wordt een Spark training master gebruikt. De training master bepaalt hoe gedistribueerde training wordt uitgevoerd. Op het moment zijn hier twee implementaties van. De meest simpele variant is de *Parameter Averaging* methode, waarbij de training gedistribueerd wordt en de training master vervolgens de parameters samenvoegt om het netwerk te configureren. De methode die echter tegenwoordig wordt aangeraden is de *Gradient Sharing* methode, gebaseerd op de Strom 2015 neural network paper door Nikko Strom van Amazon (**http://nikkostrom.com/publications/interspeech2015/strom_interspeech2015.pdf**). Een uitgebreide uitleg over deze implementaties is beschikbaar op de site van Deeplearning4j.

```
JavaSparkContext sc = ...;
JavaRDD<DataSet> trainingData = ...;

//Model setup as on a single node. Either a MultiLayerConfiguration or a ComputationGraphConfiguration
MultiLayerConfiguration model = ...;

//Create the TrainingMaster instance
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster = new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
            .(other configuration options)
            .build();

//Create the SparkDl4jMultiLayer instance and fit the network using the training data:
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, model, trainingMaster);

//Execute training:
for (int i = 0; i < numEpochs; i++) {
    sparkNet.fit(trainingData);
}
```

Listing 4

Conclusie

De API van Deeplearning4j is intuïtief en voelt fijn aan als Java developer. Het is immers gebaseerd op moderne patterns, die veel voorkomen in moderne Java interfaces. De integratie met Spark is een mooie toevoeging op deze machine learning library, waardoor Deeplearning4j goed in bestaande Java stacks past. Het wordt je gemakkelijk gemaakt om Deeplearning4j aan te sluiten op bestaande databases en DataVec zorgt ervoor dat je niet na hoeft te denken over hoe veelgebruikte datasoorten omgezet moeten worden naar een genormaliseerde vector. Deeplearning4j maakt het mogelijk om Deep Learning te implementeren in bestaande applicaties, zonder dat je daarbij van de Java taal af hoeft te stappen, en dat zonder een groot verlies in performance en zonder dat je zelf wiskundige functies hoeft te implementeren.

Voor voorbeelden van Deeplearning4j kan je kijken op de Github repository "dl4j-examples" (**https://github.com/deeplearning4j/dl4j-examples**). Met name de MNIST single layer example (**https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/mnist/MLPMnistSingleLayerExample.java**) en andere feedforward voorbeelden (**https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/mnist/MLPMnistSingleLayerExample.java**) zijn een goed startpunt. ■

De afbeelding over Deeplearning4j technieken is afkomstig van Deeplearning4j.org.

Microservices ontwikkelen met

Micronaut

Voor het programmeren van een microservice in Java is Spring Boot tegenwoordig wel de standaard. Maar Spring Boot is niet het enige framework dat we kunnen gebruiken voor het schrijven van een microservice. Aan het eind van 2018 kwam Micronaut uit. Dit belooft een interessant framework te zijn voor het schrijven van Microservices in Java, Kotlin en Groovy. Tijdens JFall 2018 heb ik al het één en ander kunnen laten zien en in dit artikel gaan we nog wat dieper in op Micronaut.

Micronaut wil een goede ontwikkelervaring geven aan ontwikkelaars door een snelle opstarttijd van de applicatie te bieden. Daarnaast is het ook erg geschikt voor cloud deployments, omdat de focus ligt op een zo'n klein mogelijke deployment unit. Dit betekent zowel in de grootte van het JAR bestand met de applicatie als het gebruik van geheugen. Zo klein dat Micronaut ook gebruikt kan worden voor het schrijven van serverless functions. Verder zijn er standaard al vele cloud-native mogelijkheden in Micronaut aanwezig, zoals service discovery en tracing.

Ook al is Micronaut een nieuwe framework, de leercurve is niet erg hoog. De ontwikkelaars weten ook dat Spring Boot veel wordt gebruikt, dus veel annotaties en opzet van een applicatie lijken erg op Spring Boot. Dit maakt de overgang een stuk makkelijker als je al ervaring hebt met Spring Boot. We zullen later ook zien dat zelfs een bestaande Spring Boot applicatie kan gecompileerd worden als Micronaut applicatie.

Installatie

De makkelijkste manier om de Micronaut commandline interface te installeren, is via SDKMAN! (**http://sdkman.io**). We hebben dan het commando mn beschikbaar waarmee we snel een nieuw project voor onze Micronaut applicatie kunnen maken. Eigenlijk is dit alleen nodig voor het maken van een project en is de tool mn niet nodig als we al een bestaand project hebben. Bij het maken van het project kunnen we kiezen of de build tool Maven of Gradle is. Dat is dan ook de tool die we gebruiken om de applicatie te bouwen.

Compile-time dependency injection

Micronaut implementeert dependency injection om afhankelijkheden tussen componenten te kunnen definiëren. We kennen dit ook uit Spring, maar Micronaut doet het net even anders. Bij het compileren van de code worden ook alle dependency injection afhankelijkheden geanalyseerd en worden extra class files gegenereerd met metadata. Bij het opstarten, worden deze class files gebruikt om de dependency injection uit te voeren. Hierdoor is het niet nodig om bij het opstarten van de applicatie de afhankelijkheden te bepalen via reflectie, zoals Spring dat standaard wel doet. Hierdoor neemt de opstarttijd van de applicatie ook niet toe als er meer code in de applicatie komt, iets wat we vaak wel zien bij run-time dependency injection.

Een bijkomend voordeel is dat het geheugengebruik beperkt blijft. Er hoeven geen caches voor metadata of proxy classes aangemaakt te worden die in het geheugen van onze applicatie komen te zitten.

Configuratie

Het configureren van een Micronaut applicatie kan op veel verschillende manieren en is vergelijkbaar wat met Spring Boot kan. We kunnen environment variabelen gebruiken, Java system properties en configuratie bestanden geschreven in bijvoorbeeld YAML. Er is een hierarchy voor het inlezen van configuratie waarden, zodat het altijd mogelijk is om configuratie waarden te overschrijven zonder de applicatie code of deployment unit aan te passen.

Hubert Klein Ikkink is een Java/Groovy consultant en developer bij JDriven. Hij is binnen de Groovy community beter bekend als mrhaki, want dat is de nickname die hij gebruikt om blogposts te schrijven over Groovy, Gradle, Grails en andere zaken die te maken hebben met Groovy technologieën.

Voor het injecteren van configuratie waarden in onze componenten kunnen we gebruik maken van de @Value annotatie. Het is ook mogelijk om een class te maken en die te annoteren met @ConfigurationProperties voor een type-safe configuratie. Hiermee is het mogelijk om ook validatie toe te passen voor de configuratie properties.

HTTP server

Voor het verwerken van HTTP requests gebruikt Micronaut Netty. Netty is een stabiele library voor snelle en efficiënte IO en maakt gebruik van non blocking threads. Hierdoor kunnen we ook gebruik maken van een 'reactive' library om onze code te schrijven. Elke library die de Reactive Streams API implementeert, kan worden gebruikt. Standaard kunnen RxJava2 en Reactor worden gebruikt.

Om een request te verwerken, maken we een controller aan in Micronaut. Dat doen we met de @Controller annotatie die we toepassen op een class. Micronaut defaults gaan uit van JSON input and output, maar dat kunnen we ook wijzigen per controller of method in een controller. Voor JSON support gebruikt Micronaut trouwens de Jackson library. Om een HTTP GET, POST of andere methode te verwerken, plaatsen we annotaties als @Get of @Post bij de methode die de request gaat afhandelen.

Zoals eerder al vermeld, is Micronaut ook reactive als we gebruik maken van bijvoorbeeld RxJava2 of Reactor. Dit zien we ook terugkomen bij de definitie van onze controllers. Wanneer we zeker weten dat een request helemaal reactive afgehandeld kan worden, omdat er geen blocking calls in de chain zitten, dan moeten we de methode argumenten en return types definiëren als reactive types. Micronaut zal dit herkennen en ervoor zorgen dat de request via non blocking threads van Netty afgehandeld worden. Maar gebruiken we non-reactive method argumenten of return types, dan zal een standaard thread pool worden gebruikt, zoals we dat ook kennen als we bijvoorbeeld Tomcat gebruiken als HTTP server.

Naast standaard request/response processing heeft Micronaut ook ondersteuning voor server sent events en web sockets.

```
package mn.sample;

import io.micronaut.http.annotation.*;
import io.reactivex.Single;

@Controller("/greeting")
public class GreetingController {

    @Get("/hello/{name}")
    public Single<Message> hello(String name) {
        return Single.just(new Message("Hello " + name));
    }
}
```

Listing 1

Listing 1 laat zien hoe een controller met een response voor HTTP GET gemaakt wordt:

Management en monitoring

Gebruikers van Spring Boot kennen waarschijnlijk Spring Boot Actuator voor endpoints die het monitoren van onze applicatie mogelijk maken. Micronaut biedt een zelfde functionaliteit via een management feature. Die maakt het mogelijk om onze applicatie te monitoren met bijvoorbeeld health checks en metrics.

Security

Micronaut biedt standaard beveiliging aan voor onze applicatie. Dit is onderdeel van het framework zelf, dus we hebben geen externe dependencies nodig. Via configuratie of annotaties kunnen we aangeven welke endpoints beveiligd moeten worden. We kunnen basic authenticatie gebruiken, session authenticatie, JSON web tokens en LDAP integratie.

HTTP client

Naast een HTTP server gebaseerd op Netty, gebruikt Micronaut ook Netty voor het uitvoeren van HTTP requests naar externe applicaties vanuit onze applicatie. Hierdoor is het mogelijk om een reactive HTTP call te maken naar een andere web service. Micronaut biedt hiervoor een zogenaamde low-level client en de mogelijkheid van een declaratieve HTTP client. We gaan kijken wat dit inhoudt als we een request willen uitvoeren.

De low-level client is een instantie van een class die de HttpClient interface implementeert. We kunnen deze injecteren als component in onze code. We maken gebruik van de speciale @Client annotatie. Via deze annotatie kunnen we bijvoorbeeld ook de externe URL definiëren die we willen benaderen. Ook via externe configuratie waarden kunnen we de client configureren.

OOK AL IS MICRONAUT EEN NIEUWE FRAMEWORK, DE LEERCURVE IS NIET ERG HOOG

De declaratieve client is een andere manier om HTTP requests uit te voeren. In dit geval moeten we @Client annotatie op een interface of abstracte class gebruiken. Micronaut zal voor de methoden in de interface of abstracte class de juiste code genereren bij het compileren. We hoeven dus zelf niet meer de code te schrijven die de daadwerkelijk request gaat uitvoeren, maar alleen de methode argumenten en return types die we verwachten. Net als met de low-level client kunnen we een declaratieve client injecteren in andere componenten.

Een voorbeeld van een declaratieve client zien we in **listing 2**.

```
package mn.sample;

import io.micronaut.http.client.annotation.Client;
import io.reactivex.Single;

@Client("/greeting")
public interface GreetingClient {
    Single<Message> hello(String name);
}
```

Listing 2

Testen

Het is belangrijk dat een framework het testen van code gemakkelijk maakt en daarin stelt Micronaut niet teleur. Doordat een Micronaut applicatie zo snel start, is het ook gemakkelijk om de applicatie te starten in testen. Om bijvoorbeeld controllers te testen, hebben we geen mock of stub componenten nodig en kunnen we de applicatie helemaal starten om alles te testen.

Daarnaast is ook support voor JUnit 5 met een eigen extensie. Het is dan mogelijk configuratie te veranderen voor tests en componenten te vervangen door mock of stub componenten.

Cloud-native features

De microservices die we schrijven in Micronaut zullen vaak draaien in een cloud omgeving. Ook zullen meerdere microservices vaak met elkaar moeten communiceren en elkaar kunnen vinden via bijvoorbeeld service discovery. Micronaut biedt standaard support voor verschillende cloud-native features, zoals gedistribueerde configuratie, service discovery, client-side load balancing, fault tolerance en distributed tracing. Hiervoor zijn geen externe dependencies nodig, want het is allemaal al standaard in Micronaut aanwezig. Er wordt veel gebruik gemaakt van de Micronaut HTTP client in het framework om de support te bieden.

Distributed configuratie

Micronaut kan herkennen in welke omgeving de applicatie draait en maakt dit beschikbaar via metadata, zodat we dat ook kunnen gebruiken in onze applicatie om bijvoorbeeld bepaalde componenten wel of niet te laden. Omgevingen zoals AWS, Kubernetes, Google Compute, Heroku, Azure en nog meer worden al herkend.

Service discovery

Een microservice die we schrijven in Micronaut functioneert vaak niet alleen en heeft afhankelijkheden naar andere microservices. We willen graag dat we deze andere microservices kunnen benaderen, zonder dat we moeten weten welk IP-adres we moeten gebruiken. Hiervoor zijn service discovery servers beschikbaar waar een applicatie zichzelf registreert en met een logische naam is te benaderen door een andere microservice.

Micronaut ondersteunt de service discovery server Consul en Eureka. We moeten via de configuratie aangeven waar de discovery server is te vinden en Micronaut zal de applicatie registreren bij de server.

Clientside load balancing

We kunnen de @Client annotatie gebruiken om een microservice die is geregistreerd bij een service discovery server aan te roepen. De naam die we gebruiken is een logische naam, zodat we niet meer een precieze URL of IP-adres moeten weten. Dit werkt ook voor bijvoorbeeld voor services in Kubernetes.

Fault tolerant

Bij het gebruik van microservices moeten we ervan uitgaan dat dingen fout kunnen gaan. Bijvoorbeeld een service is niet beschikbaar of het netwerk is traag. In onze Micronaut applicatie hebben we verschillende annotaties zoals @Fallback, @Retryable en @CircuitBreaker tot onze beschikking. Op de plekken waar het fout kan gaan, kunnen we dan aangeven of we het een aantal keer opnieuw willen proberen, voordat écht een fout optreedt in

HET CONFIGUREREN VAN EEN MICRONAUT APPLICATIE IS VERGELIJKBAAR MET WAT SPRING BOOT KAN

onze applicatie. Dit maakt onze applicatie een stuk robuuster.

Distributed tracing

Om erachter te komen hoe een gebruikers request door een systeem van microservices gaat, kunnen we distributed tracing gebruiken. Bij de initiële request worden extra properties, zoals een unieke identifier, toegevoegd aan de request. Deze informatie is beschikbaar voor de hele chain die nodig is voor de request en we kunnen dit vastleggen in bijvoorbeeld Jaeger of Zipkin. Met behulp van Jaeger of Zipkin krijgen we via een UI inzicht welke stappen en services nodig waren met daarbij ook de benodigde tijd.

Nog meer...

Naast de genoemde features biedt Micronaut nog veel meer mogelijkheden. We benoemen nog een aantal zaken die Micronaut standaard al biedt voor ons ontwikkelaars.

Serverless

Omdat Micronaut een snelle opstarttijd heeft, weinig geheugen nodig heeft en een kleine deployment unit heeft, is het ook erg geschikt voor serverless functions. Op dit moment wordt AWS Lambda en OpenFaaS ondersteunt.

Het schrijven van een functie is niet moeilijk in Micronaut en we hebben dezelfde framework features als met een standaard microservice.

Native met GraalVM

Micronaut biedt ook ondersteuning voor het schrijven en maken van applicaties voor GraalVM. Deze feature is nog experimenteel, zoals de ontwikkelaars van Micronaut zelf aangeven. Maar het maakt het mogelijk om een native tool te schrijven in Java die ook nog snel is en weinig geheugen gebruikt.

Message-driven microservices

Als we geen HTTP willen gebruiken voor communicatie tussen microservices, dan kunnen we ook gebruik maken van messages. Communicatie gaat dan via een message broker, zoals RabbitMQ of Kafka. Voor Kafka en Kafka Stream heeft Micronaut extra support via verschillende annotaties en configuratie mogelijkheden. Hierdoor is het mogelijk om Kafka te gebruiken voor communicatie tussen microservices geschreven in Micronaut.

Commandline applicaties

We schrijven meestal applicaties die een HTTP server hebben, maar met Micronaut is het ook gemakkelijk om een commandline applicatie te schrijven. Denk bijvoorbeeld aan een tool die een service kan raadplegen via de commandline of batch applicaties. Door het gebruik van de Picoli library, is het heel gemakkelijk commandline argumenten in te lezen en te gebruiken. Verder hebben we de beschikking over het hele Micronaut arsenaal, zoals de HTTP client om code te schrijven.

Data access

Micronaut biedt toegang tot verschillende data stores. Zo kunnen we via JDBC een SQL database benaderen en wordt Hibernate ondersteund. Verder bestaat support voor een nog wat experimentele reactive PostgreSQL library. Hiermee is het mogelijk PostgreSQL te gebruiken een reactive stream van data. Ook is ondersteuning voor reactive NoSQL databases als MongoDB, Neo4J, Redis en Cassandra.

Micronaut for Spring

Naast Micronaut is ook nog Micronaut for Spring beschikbaar. Hiermee is het mogelijk om een Spring Boot applicatie te compileren naar een Micronaut applicatie. De Micronaut applicatie heeft dan alle voordelen van een Micronaut applicatie die we anders helemaal in Micronaut hadden geschreven, maar op source code niveau is het een Spring Boot applicatie.

We hebben al gezien dat veel annotaties in Micronaut lijken op annotaties die we kennen in Spring Boot. Bij het compileren worden de Spring Boot annotaties gemapped op Micronaut annotaties en de uitkomst is een Micronaut applicatie. Dit is natuurlijk erg aantrekkelijk voor bestaande Spring Boot projecten. Zonder aanpassing is de runtime van de applicatie een stuk sneller geworden. Nog niet alle Spring features worden hierin ondersteund, maar het is wel al genoeg om nuttig te zijn.

Conclusie

Micronaut is eind 2018 als version 1.0 op de markt gekomen en het is al een compleet framework voor het maken van microservices. Vooral voor ontwikkelaars die Spring Boot kennen, is veel herkenbaar en dat maakt een overgang vrij makkelijk. De grote voordelen zijn (1) de snelle opstarttijd, (2) het lage geheugengebruik en (3) de reactive support. ■

HET IS BELANGRIJK DAT EEN FRAMEWORK HET TESTEN VAN CODE GEMAKKELIJK MAAKT EN DAARIN STELT MICRONAUT NIET TELEUR

Bamboo Specs

Herbruikbare en beheersbare continuous deployment code

In dit artikel gaan we kijken naar een recente toegevoegde feature in Bamboo. Bamboo is een CD tool van Atlassian. In de zomer van 2018 is de Specs functionaliteit toegevoegd. Dit houdt in dat je (bijna) alles wat je in de GUI kan definiëren ook in code kan definiëren. Dit is in de basis vergelijkbaar met Jenkins pipelines die je in Groovy schrijft.

Wat Bamboo Specs (hierna Specs genoemd) onderscheidt, is dat je de Specs in Java of ieder andere JVM taal kunt schrijven, zoals bijvoorbeeld Kotlin, Groovy of Scala. Daarnaast kun je Specs ook in Yaml definiëren, net zoals in Gitlab CI of Travis CI. De Yaml Specs behandel ik in dit artikel kort, omdat deze variant niet herbruikbaar is, zoals de Java variant. Met herbruikbaar wordt hier bedoeld dat je in staat bent om CD definities te hergebruiken voor verschillende projecten. Zo hoef je in de optimale situatie alleen project specifieke details op te geven wanneer je een nieuw project start. In de praktijk 'copy pasten' we vaak een project om daarna deze te tweaken voor het nieuwe project. Dit hoeft niet als je gebruik maakt van Bamboo Specs. Ook komt beheerbaarheid goed van pas als je project overstijgende aanpassingen moet doen, daar wil je liever ook niet tientallen of honderden CD definities met de hand aanpassen. Om van Bamboo specs gebruik te maken, heb je Bamboo 6.0.0 of hoger nodig.

Na wat Bamboo terminologie gaan we kort de Yaml versie beschrijven en de verschillen met de Java variant. Daarna gaan we een, voor een enterprise scenario, mogelijke aanpak behandelen. Als laatste behandel ik een aantal haken en ogen waar je rekening mee dient te houden als je Bamboo Specs wilt gebruiken.

Bamboo terminologie

Bamboo heeft een CD specifiek domein. Alles start binnen Bamboo met een Project. Een project heeft één of meerdere plans en dient ook als centraal dashboard voor de plans die het bevat.

Plan's bevatten Stages, Jobs en Tasks en maken doorgaans gebruik van een Repository. Daarnaast kan een Plan ook één of meerdere Artifacts opleveren. Een Stage is een container die één of meerdere Jobs bevat. Jobs worden parallel uitgevoerd, desgewenst op meerdere Bamboo agents. Jobs bevatten Tasks. Tasks worden sequentieel uitgevoerd in de volgorde zoals deze in de Job gedefinieerd staan.

Een Repository, zoals de naam doet vermoeden, bevat repository metadata die noodzakelijk is om hiermee te verbinden. Een Artifact is een optionele output definitie waarbij je aan kan geven wat je van de Plan bewaard wilt hebben zodat deze beschikbaar wordt voor een andere Plan. Een simpel voorbeeld zou kunnen zijn dat je de Jar uit de target folder als Artifact definieert, zodat je deze weer gebruikt in een Deployment Project.

Deployment projects maken onder andere gebruik van Artifacts en Environments en definiëren Deployments in de breedste zin. Environments definiëren één of meerdere servers die logisch bij elkaar horen. Naast deze Bamboo domeinobjecten zijn er nog een aantal andere domeinobjecten waarbij je onder andere variabelen, triggers en permissies kunt instellen. Deze kun je in de documentatie terugvinden. In afbeelding 1 zie je een grafische weergave van het Bamboo core domein.

Ben Ooms is een gepassioneerde Java senior sorftware engineer bij Quintor Den Haag die graag kennis deelt.

Yaml

Om te beginnen met Specs gaan we een simpel Maven project definiëren in Yaml waarbij we mvn package willen uitvoeren en de resulterende Jar als Artifact beschikbaar willen stellen. Hiervoor hebben we nodig:

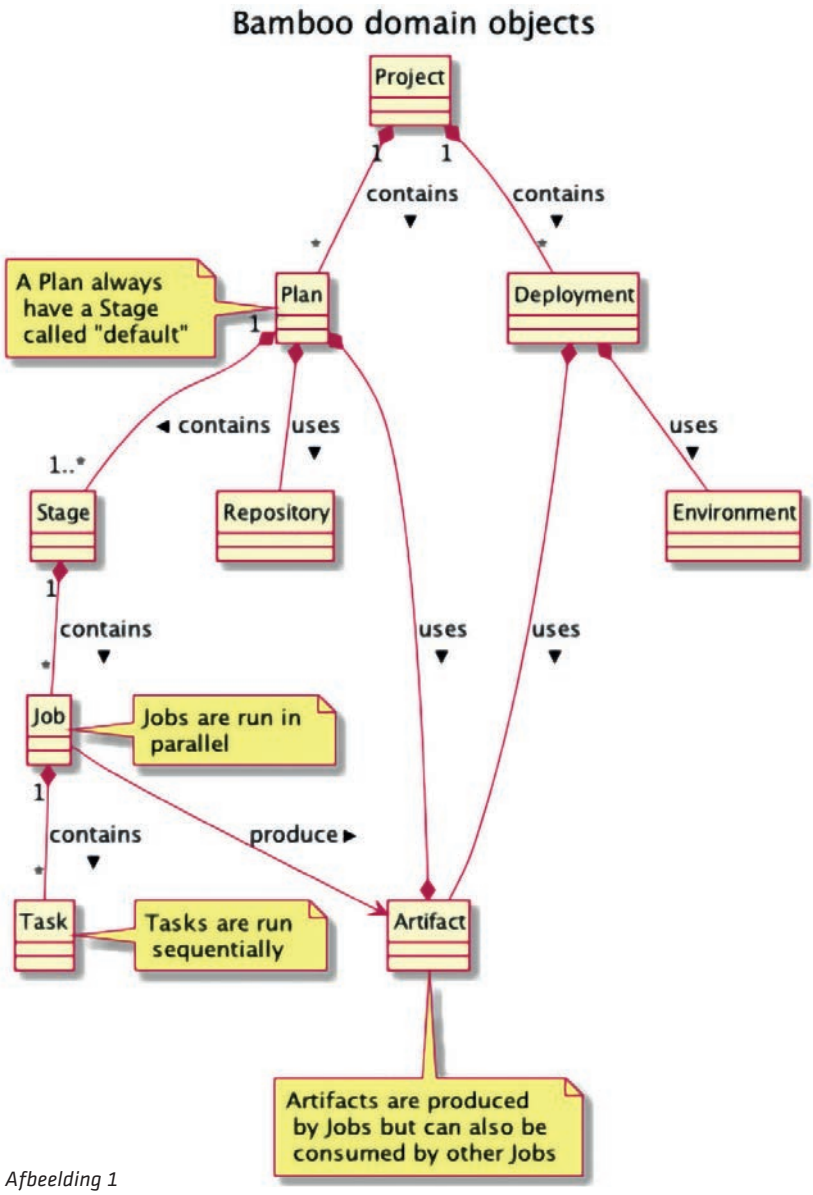
- een Project;
- een Repository;
- een Stage;
- een Job;
- een aantal Tasks;
- een Artifact.

In listing 1 zie je hoe dit is gedefinieerd in Yaml formaat. De Java variant van ditzelfde scenario vind je in listing 2. Zoals je in beide listings kunt zien, is het wanneer je het Bamboo domein beheerst erg eenvoudig om dit project te definiëren, zowel in Yaml als in Java. Helaas zal je opgevallen zijn dat de Tasks altijd van het type 'scripts' zijn in de Yaml variant. Bijna alle voordelen vallen weg als je gebruik maakt van Yaml. Je heb geen type safety, geen overerving, geen mogelijkheid tot testen. Ook is niet het hele Bamboo domein te definiëren in Yaml, maar een kleine subset daarvan. Uit de documentatie en research op onder andere Stackoverflow en de Atlassian support forums zie je dat Atlassian je verwijst naar de Java variant wanneer je iets niet voor elkaar krijgt in de Yaml variant.

Java variant

In de Java variant hebben we wel dit simpele scenario volledig kunnen definiëren en daarnaast kan de definitie hergebruikt worden (uiteraard niet zoals in listing 2, alles zit voor leesbaarheid in de main methode) en kunnen we dit ook testen voordat we de definitie uploaden naar de Bamboo server. We hebben zelfs plan branch management. In Bamboo maakt een plan standaard gebruik van de default branch. Maar je kun ook plan branch management inschakelen. Hiermee koppelt Bamboo je plan aan iedere branch in de repository. Natuurlijk kun je dit ook niet gebruiken of het automatisch aanmaken van de branch plans aan je wensen aanpassen. Het grote voordeel wat we hier zien, is dat een plan universeel te gebruiken is, branch onafhankelijk.

De documentatie van de Yaml en Java Specs zijn redelijk gedocumenteerd. Naast de referentie documentatie heb je voor de Java variant ook nog de Javadocs.



Afbeelding 1

Wat we weggelaten hebben in de Java variant, is de Maven configuratie. Bamboo biedt hier een Archetype die je helpt om snel een CD project op te zetten. Doordat het een Maven project betreft, kan je ook zelf dependencies toevoegen en kun je de definities ook geautomatiseerd testen. Ook kun je libraries ontwikkelen die project overstijgend zijn en hergebruiken in projecten.

Met de Java variant is het dus mogelijk om herbruikbaarheid en de beheersbaarheid te

VOORAL IN HET BEGIN KOST SPECS EVEN TIJD DOOR MIDDEL VAN TRIAL EN ERROR

```
project:
  key: PRJ1
  plan:
    key: PLN1
    name: Build plan
stages:
  - jobs:
      - scripts:
        - git clone https://gitrepo/myfantasticproject && git checkout
master
      - scripts:
          - M2_HOME=${bamboo_capability_system_builder_mvn3_Maven_3}
          - ${bamboo_capability_system_builder_mvn3_Maven_3}/bin/mvn
clean package
        artifacts:
          - name: My jar
            path: target/*.jar
        testParsers:
          - junit
```

Listing 1

promoten. Zo zou je dus met Java alleen al shared libraries kunnen ontwikkelen met project overstijgende defaults. Mogelijke Specs domein objecten zouden kunnen zijn:

- environment;
- environmentPermission;
- plan;
- planpermission;
- variable.

Permisson en Variable

Eerst een korte uitleg over Permission en Variable. Deze hebben we immers nog niet behandeld. De permissions stellen je in staat om toegang en gebruik te beperken tot gebruikers en groepen. Met de Variable kun je waardes delen binnen Bamboo. Variables kunnen hebben een scope, zo kun je variabelen op systeem, Bamboo server en planniveau definiëren. De systeem scope definieer je niet zelf, maar zijn een alias die je automatisch kunt gebruiken. Bijvoorbeeld als je een systeemvariabele genaamd MYVAR hebt, dan kun je deze in Bamboo gebruiken met de alias 'system.MYVAR'.

Met Environment objecten zou je definities kunnen opstellen voor Systeem, Acceptatie en Productie omgevingen. Voor toegang zou je deze kunnen complementeren met Environment permissies.

Plans en Planpermissions

Hetzelfde geldt ook voor Plans en Planpermissions. Naast het Java hergebruik, bieden de Specs ook nog het concept van Parent Plans. Met deze constructie kun je (abstracte) Plans definiëren waar Child plans van overerven. Zo zou je dus omgevingsbrede plans kunnen definiëren waar de omgevingsafhankelijke definities zijn opgenomen waarbij in Child plans de projectdefinities zijn opgenomen. Dit is heel krachtig en levert direct twee grote voordelen op. Enerzijds het sneller opzetten van Plans en anderzijds een betere beheerbaarheid van alle plans, doordat omgevingswijzigingen nu op Parent plans kunnen worden toegepast en beschikbaar zijn voor de Child plans zonder deze aan te passen. Let op, hier zitten een paar haken en ogen aan, dit behandel ik verderop in het artikel.

Optimaal gebruik

Wil je optimaal gebruik maken van overerving en hergebruik, dan zal je initieel tijd moeten investeren in het modelleren van de omgeving, systemen en rollen die gelden binnen de organisatie. Daarna zou je de defaults die in elk project terugkomen, kunnen modelleren in Parent plans. Deze definities zou je in een Maven library kwijt kunnen waar elk project gebruik van zou kunnen maken als basis en daarnaast de parent plans publiceren op de Bamboo server.
Een tip is dat -als je al gebruik maakt van Bamboo- je bestaande definities die je in de GUI gemaakt hebt in Java Specs kan zien. Hiervoor heb je een optie 'View as Specs'. Deze functie genereert Java code voor het huidige project die dan als basis kan dienen. Ik heb deze functie ook gebruikt om ingewikkelde builds eerst in de GUI te definiëren,

DE BESTE COMBINATIE OM SPECS TE LEREN, IS DOOR IEMAND MET VEEL BAMBOO GUI ERVARING SAMEN TE VOEGEN MET EEN JAVA DEVELOPER

zodat ik Java code als basis had die ik dan verder alleen nog hoefde te tweaken.

Zoals je ziet, zijn grote voordelen te behalen om jouw CD omgeving door middel van Specs te modelleren. Ik heb niet alle domein objecten behandeld in dit artikel. Er bestaan nog veel meer! Denk hierbij aan o.a. het definiëren van triggers, afhankelijkheden en Utilities. De utilities zijn Specs helper klassen die hulpmiddelen bieden voor bijvoorbeeld het publiceren van Specs naar een Bamboo server en het omgaan met credentials.

Haken en ogen

Wel is het zo dat naast de voordelen ook een aantal haken en ogen bestaan waar je rekening mee dient te houden (op het moment van schrijven). Het eerste is dat de verantwoordelijken voor CD Java moeten beheersen. Voor ons Javanen natuurlijk geen probleem, maar hier dien je wel organisatorisch naar te kijken. Risico is dat er door manco aan goede basis Java kennis, voordelen wellicht niet behaald worden in termen van herbruikbaarheid en beheerbaarheid.

```
public class SomeHolderClass {

    public static void main(String[] args) {
        Repository repository = new GitRepository()
            .name("my-repo")
            .url("https://gitrepo/myfantasticproject")
            .branch("master");

        Project project = new Project().key("PRJ1");

        MavenTask mavenTask = new MavenTask().goal("clean package");

        Job job = new Job("Job", "JOB")
            .tasks(new VcsCheckoutTask()
                .addCheckoutOfDefaultRepository(), mavenTask)
                .artifacts(new Artifact("My jar")
                    .location("target")
                    .copyPattern("*.jar")
                    .shared(true)));

        Stage stage = new Stage("My Stage").jobs(job);

        Plan plan = new Plan(project, "Build plan", "PLN1")
        .planRepositories(repository)
        .stages(stage)
        .planBranchManagement(new PlanBranchManagement());

    }
}
```

Listing 2

Wanneer je externe dependenties gebruikt, dan worden deze niet gebruikt op de server. De Bamboo server gebruikt zijn eigen pom wat betekent dat je toegevoegde dependencies niet gevonden worden. Hiervoor is een niet al te elegante workaround voor, namelijk de pom op de server aanpassen. Een andere verrassing die je tegen zal komen, is dat wanneer een parent Plan aangepast is de Child plans niet automatisch opnieuw gepubliceerd worden. De wijzigingen worden dus ook niet toegepast op de bestaande Childs! Ook hiervoor is een oplossing in de vorm van een community plug-in die dit wel in de gaten houdt en Child plans bijwerkt na een update van de Parent plan. Hoewel de documentatie wel aanwezig is op Object niveau, mis je documentatie om Specs effectief te gebruiken voor gehele CD flows. Vooral in het begin kost dit even tijd door middel van trial en error. Op dit vlak zou de documentatie veel beter kunnen.

Samenvattend

Samenvattend hebben we een introductie gegeven over Bamboo Specs. Na een uitleg van de core Bamboo objecten hebben een voorbeeld behandeld van een Yaml en Java variant van Specs waarbij de nadelen van de Yaml varianten zichtbaar werden. Het kunnen gebruiken van Java biedt ons het voordeel van testbaarheid, hergebruik en beheerbaarheid. Ook hebben we een generieke aanpak besproken om efficiënt Specs te implementeren. Toen ik enkele maanden geleden in Bamboo Specs dook was het lastigste om het Bamboo domein te leren kennen. Als je voor jouw omgeving Specs zou willen gebruiken, dan lijkt de beste combinatie te bestaan uit iemand met veel Bamboo GUI ervaring gecombineerd met een Java developer. Geen probleem in een DevOps team lijkt mij. Wel is het goed om te kijken naar de complexiteit en grootte van de omgeving(en) en projecten. Als jouw team alleen in een geïsoleerde simpele omgeving bezig is en straight forward projecten bouwt, dan is Bamboo Specs mogelijk overkill.

Tot slot

Ik hoop je inzicht te hebben gegeven over de mogelijkheden van Bamboo Specs, wat het inhoud en welke mogelijkheden het biedt. Als laatste wil ik je nog meegeven dat een publieke Docker image bestaat met een Bamboo server waarbij je met Specs kunt spelen. De image naam is cptactionhank/atlassian-bamboo. Laat je niet afschrikken door het license scherm dat je ziet,. Een trial license van 90 dagen kun je eenvoudig aanvragen met een Atlassian account. De volledige documentatie is te vinden op https://docs.atlassian.com/bamboo-specs-docs/6.7.2/. ■

Mobiele apps met Ionic

Zo gebouwd!

De wereld van mobiele apps is al jaren aan het veranderen. Bij de keuze voor een ontwikkelplatform worden telkens weer de volgende vragen gesteld. Wat is de doelgroep voor de app en voor welk OS dient de app te worden ontwikkeld? Web, hybride of native? Wat zijn de voor- en nadelen van de keuze voor het platform? Kan ik mee met de nieuwste ontwikkelingen en hoe zorg ik dat mijn app up-to-date blijft? In dit artikel ga ik jullie meenemen in de antwoorden, die Ionic hierbij kan bieden.

Op elke vraag zoals hierboven gesteld is per app weer een voor- of tegenargument te bedenken. Dit maakt de keuze om een platform te kiezen erg lastig, want een migratie in een latere fase betekent in veel gevallen dat de app volledig herschreven dient te worden.

Daarnaast zijn er nog meer aspecten belangrijk bij deze keuze: wat is je huidige ontwikkelomgeving? Wat voor kennis is er al aanwezig in de huidige teams? Hoe ziet de community van het platform eruit? Wat is het budget? Deze laatste vraag is in veel gevallen erg belangrijk als er een keuze gemaakt dient te worden tussen native en hybride. De kosten van een hybride app zijn meestal aanzienlijk lager, omdat de core functionaliteit intact blijft en er enkel wijzigingen aan de visuele zijde hoeven te gebeuren, om de app voor elk OS er goed uit te laten zien. Dit verschil is echter verwaarloosbaar indien de app alleen voor iOS of Android hoeft te werken.

Wat is Ionic?

Ionic is een HTML5 deployment framework dat is gericht op het bouwen van hybride apps. In tegenstelling tot een responsive framework (Bootstrap, Foundation, etc), wordt Ionic geleverd met mobiele elementen in de stijl van de native components en layouts, die je zou krijgen met een native SDK op iOS of Android.

Omdat Ionic een HTML5 framework is, heeft het een native wrapper, zoals Cordova of PhoneGap, nodig om als native app te kunnen functioneren. Deze wrapper zorgt ervoor dat de HTML5 app kan draaien in de iOS's UIWebView of de Android's WebView.

Degenen die bekend zijn met web development zullen de structuur van een Ionic app eenvoudig oppakken. In de kern is het gewoon een webpagina, die wordt uitgevoerd in een native app-shell. Dit betekent dat HTML, CSS en Javascript gebruikt kan worden. Het enige verschil is dat we in plaats van een website een applicatie-ervaring bouwen.

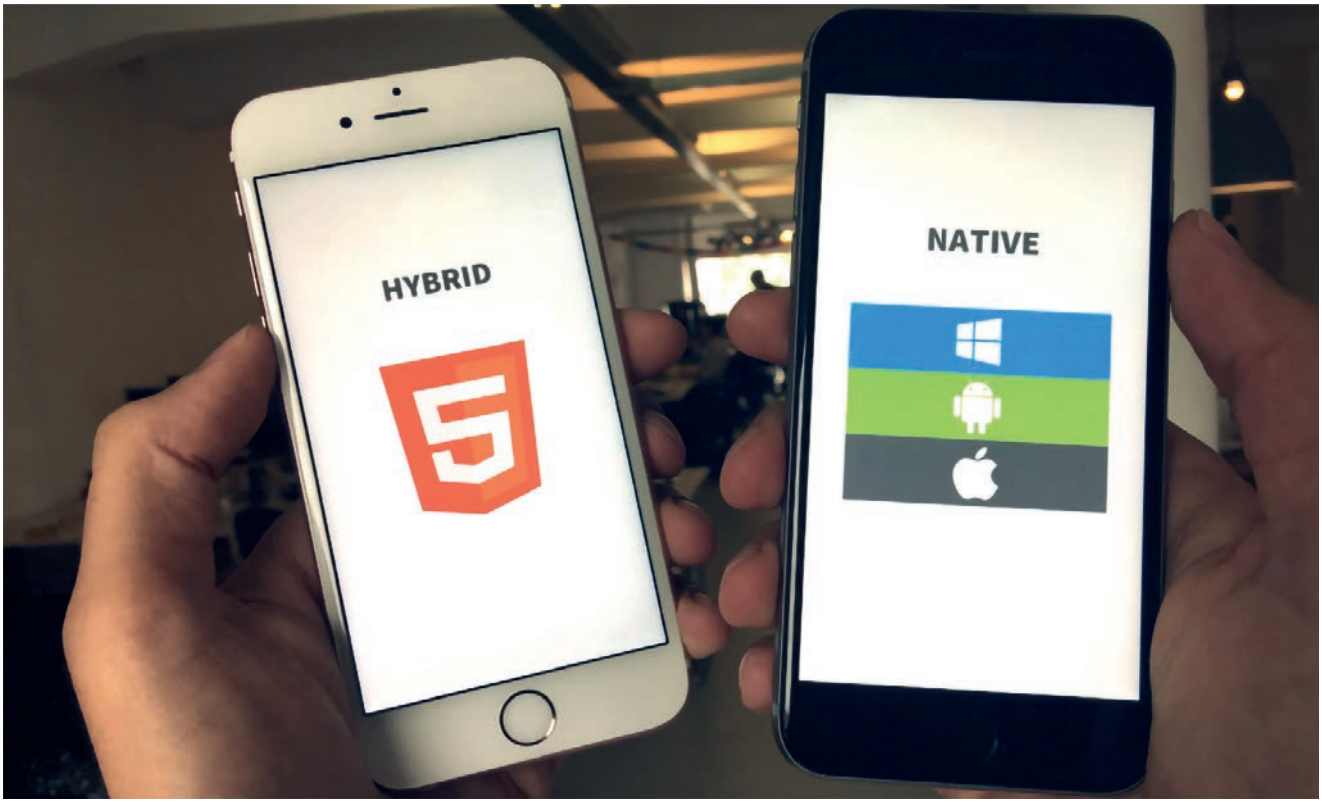
Native vs. Hybrid vs. PWA

Native apps zijn ontworpen voor een specifiek besturingssysteem en dit kan zowel iOS als Android zijn. Deze native apps zijn daarnaast ook gebouwd met de taal die hiervoor ontwikkeld is, zoals Swift, Kotlin, Objective-C of Java. Het gebruik van deze native code zal ervoor zorgen dat deze automatisch beter zal performen en geeft ook de native UI, zoals de guidelines van het desbetreffende OS voorschrijven. Door het gebruik van deze native UI voelen gebruikers zich vaak meer vertrouwd met de applicatie, omdat de rest van de mobiele interface op dezelfde manier zal aanvoelen.

Hybride apps zijn apps, die ontwikkeld zijn met behulp van een enkele codetaal voor meerdere platformen. Je kunt de apparaatspecifieke interacties bereiken met behulp van plug-ins, die zijn gemaakt voor het specifieke besturingssysteem. Het grootste voordeel van Hybride apps is dat hun ontwikkelingsproces veel minder tijd in beslag neemt. De keerzijde is wel dat performance optimalisaties lastiger zijn te verwezenlijken.

Progressive Web Apps (PWA) zijn geschreven in JavaScript / HTML5 en zijn niets anders dan websites, die ontwikkeld zijn om te reageren

Kevin Donkers is Business Unit Manager voor de JavaScript (JSRoots) en Java (JTech) units bij Ordina.



op mobiele gebruikers. Omdat veel apps ook als web-app beschikbaar zijn, is deze optie in de meeste gevallen het minste werk om te ontwikkelen. Dit komt doordat deze optie ook meteen klaar is om gereleased te worden als web-app.

Getting Started

Allereerst moeten we beginnen met de minimumvereisten voor het bouwen van een app met de huidige versie van Ionic. Ionic richt zich op iPhone- en Android-devices. Deze dienen minimaal iOS 7+ en Android 4.1+ te ondersteunen om de app op het platform te releasen. Je kunt Ionic apps ontwikkelen op elk besturingssysteem. De deployment voor iOS kan echter alleen gedaan worden op een OS X systeem.

Eerst gaan we de meest recente versie van Apache Cordova installeren. Hiermee kunnen we vervolgens de wrapping gaan doen naar een native app. Als je Cordova wilt installeren, moet Node.js zijn geïnstalleerd en kan onderstaand commando worden uitgevoerd:

```
npm install –g cordova ionic
```

Vervolgens gaan we een nieuwe app aanmaken door gebruik te maken van het Ionic package, die we zojuist hebben geïnstalleerd:

```
ionic start myfirstapp blank ––type
ionic1
```

Na het aanmaken van de app zal de volgende mappenstructuur worden aangemaakt:

```
├── bower.json     // bower dependencies
├── config.xml     // Cordova configuratie
├── gulpfile.js    // gulp tasks
├── hooks          // aangepaste Cordova hooks om uit te voeren
├── ionic.project  // Ionic configuratie
├── package.json   // node dependencies
├── platforms      // iOS/Android specifieke builds zullen hier verblijven
├── plugins        // waar de Cordova/Ionic plug–ins worden geïnstalleerd
├── scss           // scss code, die zal worden gecompiled naar www/css/
└── www            // applicatie – JS code and libs, CSS, images, etc.
```

OMDAT IONIC EEN HTML5 FRAMEWORK IS, HEEFT HET EEN NATIVE WRAPPER

Het is nu nog belangrijk om aan te geven aan Ionic voor welke platformen de build uiteindelijk gemaakt dienen te worden.

```
ionic cordova platform add ios     // Toevoegen van iOS aan de build
ionic cordova platform add android // Toevoegen van Android aan de build
ionic serve                        // Het lanceren van de applicatie
ionic cordova build ios            // Builden van een iOS .iPA
ionic cordova emulate ios          // Emuleren op een iOS toestelw
```

Je eerste app

Nu de app is opgebouwd en lokaal draait, kunnen we de app gaan voorzien van de nodige lay-out. Aangezien elke Ionic-app in feite een webpagina is, moeten we een index.html-bestand in onze app hebben, dat de eerste pagina definieert, die in de app wordt geladen. Deze is bij het aanmaken van de app reeds aangemaakt, dus laten we deze dan verder gaan uitbreiden. We gaan aan deze pagina een menu toevoegen. Dit kan door onderstaande code toe te voegen tussen de <body>-tag in de bestaande index.html:

```
<ion-side-menus>
  <ion-side-menu-content>
  </ion-side-menu-content>
  <ion-side-menu side="left">
  </ion-side-menu>
</ion-side-menus>
```

In de bovenstaande code hebben we onze `<ion-side-menus>`-controller toegevoegd, die het slepen van het menu zal afhandelen. Binnenin de controller hebben we een `<ion-side-menu-content>`, waar de content van de app zal komen, en een `<ion-side-menu side = "left">` waar de content van het menu zal komen.

Je zult nu zien dat de content van de pagina nu verder leeg is. Deze kunnen we vullen door hier wat componenten aan toe voegen:

```
<!-- Center content -->
 <ion-side-menu-content>
   <ion-header-bar class="bar-dark">
     <h1 class="title">Todo</h1>
   </ion-header-bar>
   <ion-content>
   </ion-content>
 </ion-side-menu-content>

 <!-- Left menu -->
 <ion-side-menu side="left">
   <ion-header-bar class="bar-dark">
     <h1 class="title">Projects</h1>
   </ion-header-bar>
 </ion-side-menu>
```

Gefeliciteerd, je zult nu zien dat je eerste Ionic app een feit is. Deze draait nu in de browser, je kunt hem builden voor Android en/of iOS, en deployed hem meteen door naar de App Store of Play Store, zodra hij klaar is. Een goede oefening is nu om verder te gaan met het ontwikkelen van de app met verschillende routes en navigatiestructuren. Hiervoor kun je de componenten-documentatie van Ionic bekijken met daarbij goede visuele voorbeelden.

Conclusie

De afgelopen jaren zit de mobiele wereld in een sneltrein en worden telkens nieuwe frameworks uitgebracht met daarbij elk zijn voor- en nadelen. Ionic heeft daarbij een ontwikkeling doorgemaakt en heeft zich met de komst van nieuwe platformen staande weten te houden. Het blijft echter altijd een lastige afweging wat de beste keuze is voor jouw app. Het is belangrijk om altijd jouw belangen af te wegen, voordat je begint met de ontwikkeling. Kies je platform of voorzie wat je mogelijke platformen gaan worden. ■

IONIC IS EEN HTML5 DEPLOYMENT FRAMEWORK DAT IS GERICHT OP HET BOUWEN VAN HYBRIDE APPS

REFERENTIES

https://github.com/Ordina-JTech/ionic-workshop/
https://ionicframework.com/docs/v1/

Van het bestuur

2019: Het jaar van...

En toen was het ineens 2019. Wat gaat een jaar toch hard. Er is in het afgelopen jubileumjaar 2018 dan ook genoeg gebeurd. In deze editie van Java Magazine blikken we terug op J-Fall 2018. In het vorige magazine hadden we J-Fall al 'de beste editie tot nu toe' genoemd. Nu we met z'n allen terug kunnen kijken op J-Fall 2018, kunnen we met recht zeggen: 'Het was de beste editie tot nu toe!' Er was een hele goede line-up van sessies en sprekers, vanaf de opening liep alles soepel (die drummers waren best vet), de sfeer onder de mensen was goed en ook de buitenlandse sprekers waren vol lof over de conferentie. Zelfs tijdens Devoxx België werd gememoreerd aan het succes van J-Fall en de NLJUG.

Wat gaan we dan in 2019 doen?

Kunnen we zo'n jaar waarin we als community de nodige topmomenten hebben gehad nog wel verbeteren? Natuurlijk kunnen we dat, want ondanks dat we heel trots zijn op hetgeen we met elkaar hebben neergezet, is er altijd ruimte voor verbetering. We kijken kritisch naar alle evenementen, hebben een kritische blik op de berichten die we uitsturen, voeren gesprekken met businesspartners om betere en andere content te maken en zijn we continu op zoek naar nieuwe 'wegen'. Als bestuur van de NLJUG gaan we binnenkort, tijdens een 'heisessie', alle bovenstaande activiteiten de revue laten passeren en vooral ook vooruitkijken.

Het goede behouden

We gaan natuurlijk door met J-Spring en J-Fall. Wat betreft J-Spring: op woensdag 29 mei gaan de deuren van Tivoli in Utrecht weer open voor een nieuwe editie van J-Spring. De 2018 editie heeft veel lof gekregen en zoals gezegd zullen we de tips die er gegeven zijn verwerken in een betere J-Spring 2019. En ook J-Fall gaat natuurlijk weer terugkomen, inclusief de Pre-conference en de Masters of Java. Deze dag voor J-Fall groeit steeds verder in kwaliteit en ook in kwantiteit. Jullie kunnen nu alvast de datum noteren voor J-Fall 2019: donderdag 31 oktober (en woensdag 30 oktober voor de Pre-conference en de Masters of Java).

Luisteren en verbeteren

Zoals gezegd luisteren we altijd goed naar het commentaar wat wij terugkrijgen via bijvoorbeeld de feedback in de NLJUG-app en evaluatiemails. Maar ook de terugkoppeling die we op andere wijze krijgen, bijvoorbeeld van de Businesspartners, verwerken we in nieuwe concepten. Zoals jullie wellicht weten, verlenen we vanuit de NLJUG ook een bijdrage aan de TEQnation conferentie. In 2018 was deze conferentie verdeeld over twee dagen, maar op basis van de feedback die we hebben gekregen, gaan we dit jaar over naar een 1-daagse conferentie op een nieuwe locatie. Op woensdag 15 mei gaan de deuren open van DeFabrique in Utrecht voor de TEQnation 2019. Als jullie deze column lezen is inmiddels de call-for-papers voor deze conferentie open (tot vrijdag 15 maart kun je papers insturen).

Nieuwe input

En denk je nu: prima deze aanpassingen, maar ik mis nog dingen binnen de NLJUG community, schroom dan niet om contact met ons op te nemen via **board@nljug.org**. Of kom ons op andere wijze versterken; we denken hierbij aan bloggers, vloggers en redacteuren voor Java Magazine.

Laten we er met elkaar een nog mooier NLJUG-jaar van maken!

Bert Breeman is voorzitter van de NLJUG.

Meer met Maven

Lifecycle Phases

In elke editie zal **Robert Scholte** een probleem voorleggen en deze oplossen met behulp van Apache Maven om meer inzicht te geven in Maven zelf en de vele beschikbare plug-ins.

Eén van de belangrijkste kenmerken van Maven is het concept van lifecycles. Afhankelijk van je doel roep je een phase van één van de (tot nu toe) drie lifecycles aan en Maven zal alle plug-ins uitvoeren die nodig zijn om deze phase te bereiken. Bijvoorbeeld: als je wilt testen, dan roep je de test-phase aan. Dit zorgt ervoor dat eerst de main- en testcode gecompileerd wordt, voordat de tests uitgevoerd worden.

Maven biedt de ruimte aan developers om te bepalen tot welke phase plug-ins aangeroepen moeten worden en toch lijkt een groot deel van de ontwikkelaars terug te vallen op 'defaults'. Dit zie je terug op Stack Overflow, op Jira (om een testcase te reproduceren) en zelfs tijdens conferenties. De achterliggende reden is daarbij vaak teleurstellend, want je krijgt antwoorden als "Iedereen doet het zo" of "Zo is het mij aangeleerd". Ik zou er een dagtaak aan hebben als ik iedereen erop zou moeten blijven wijzen waarom '**mvn clean install**' een hele slechte gewoonte is.

IEDEREEN DIE STANDAARD 'MVN CLEAN INSTALL' UITVOERT, LEEFT NOG STEEDS IN HET MAVEN 2-TIJDPERK!

Laten we beginnen met 'clean', een phase van de clean-lifecycle. Deze zorgt ervoor dat de output directory opgeschoond wordt. Dit betekent vaak onnodig veel IO voor zowel het verwijderen van bestanden tijdens deze phase als het genereren van bestanden tijdens de volgende phase, zeker als niks gewijzigd is. Ondertussen zijn de meeste plug-ins geoptimaliseerd en kunnen ze zelf bepalen of ze daadwerkelijk aan de slag moeten. Is er tussen de laatste run en nu niks gewijzigd aan de input-bronnen of -parameters, dan kan deze plug-in overgeslagen worden. Als je een plug-in hebt die dit niet goed doet, dan mag je de plug-in ontwikkelaars daarop aanspreken.

En dan 'install', één van de laatste phases van de build-lifecycle. Het doel van deze phase is om alle deliverables of artifacts te kopiëren naar jouw lokale repository. En daar beginnen ook meteen de potentiele problemen, omdat jouw lokale repository er ineens anders uitziet dan die van jouw collega's. Dit kan ervoor zorgen dat bij de ene een build kapot is, terwijl de ander zal zeggen: "Works on my machine." Een 'install' vervuilt letterlijk je lokale repository en is vaak onnodig en overbodig.

Toegegeven, met een multi-module project en Maven 2 was het nog noodzakelijk om een 'install' uit te voeren. Dit kwam, omdat Maven 2 nog niet in staat was om onderlinge dependencies goed te resolven. Dit probleem is echter al in 2010 opgelost!

Als er een logische default zou zijn voor de uit te voeren phase of goal, dan had Maven die wel gezet, zodat je met slechts '**mvn**' het gewenste resultaat zou krijgen. Voor wie dit fantastisch zou vinden, je kunt dit zelf oplossen in de pom.xml: in de <build>-sectie kun je namelijk een <defaultGoal> opgeven.

Maar als je van Robert geen '**mvn clean install**' uit mag voeren, wat dan wel? Dat hangt volledig van je doel af. Wil je compileren, testen of packagen? Heb je last van een corrupte output directory, dan zal je uiteraard de 'clean' uit moeten voeren. Als je twee projecten hebt die onderling afhankelijk van elkaar zijn en je wilt de wijzigingen van het ene project eerst lokaal testen bij het andere project, dan zal je 'install' uitvoeren, maar dan ben je je er ook van bewust.

Als je geen idee hebt wat je eigenlijk wilt doen, dan is de beste default '**mvn verify**'. Hierbij wordt de code gebundeld tot een artifact (tijdens de package-phase) en als er daarna nog integratietesten zijn, dan worden deze ook nog uitgevoerd. Als deze er niet zijn, dan is het effect gelijk aan '**mvn package**'. Want, iedereen die standaard '**mvn clean install**' uitvoert, leeft nog steeds in het Maven 2-tijdperk!

Mind I/O

Joop Lanting
is vaste columnist voor Java Magazine

Oorspronkelijk was sciencefiction een stijl van verhalen schrijven, meestal gebaseerd op (verkeerd begrepen) actuele wetenschappelijke publicaties. Mooi voorbeeld: Jules Verne. Tot aan de jaren 1970-1980 was SF technisch georiënteerd: het verhaal was soms bijzaak, vooral futuristische mogelijkheden werden geëtaleerd. Na ±1980 kwam het besef dat de meeste voorspellingen wensdromen bleven: 'dankzij' Hendrik Lorentz werd reizen naar de sterren een utopie (niet sneller dan het licht) en computers namen nog steeds de macht niet over.

Daarvoor in de plaats kwam een verschuiving naar de psychologie. Pionier was Isaac Asimov, die reeds in 1939 de denkende machine (de robot) en diens geestelijke problemen beschreef. Er ontstonden diverse stromingen, zoals de invloed van de techniek op de mens of confrontaties met de culturen van aliens. Star Trek, naar het model van de marine van de V.S., probeerde realistisch te blijven (men studeerde serieus op de mogelijkheid van "Beam-me-up, Scotty" en allerlei wapens). Maar daarbij werden onbestaanbare alienvolkeren 'ontdekt'.

Aliens werden intensief beschreven door de schrijver Jack Vance die exotische aardse volkeren model liet staan voor aliens die daardoor menselijke trekken kregen. Fantasy kreeg de boventoon in 'In De Ban Van De Ring' en in 'Star Wars' waar technische zaken werden vervangen door (boze) geesten en magische krachten, zoals de 'Force' en waar de natuurwetten geweld werd aangedaan.

Het zal wellicht zijn opgevallen dat Star Wars zich NIET in onze Melkweg afspeelt (om lastige verbanden met de Aarde te vermijden).

In vele verhalen speelt de macht van de geest een rol: bijvoorbeeld door gedachten te lezen of te schrijven.

LEZEN: hoe zou je gedachten kunnen lezen? Onontkoombaar moet er 'iets' worden uitgezonden, een soort hersenstraling ofzo. De plaats van deze 'uitzendingen' is bepalend voor de mogelijkheden. Als dit fenomeen zou bestaan, dan zou de frontale cortex, de voorste hersenkwab, waarin het bewustzijn zetelt, de aangewezen plek zijn. Ter vergelijking: malware zit in het main-memory van de computer. Disks en peripherals zijn, net als de grote hersenen, passief en kunnen dus niet rechtstreeks worden 'gehackt'. Verder zou de betreffende straling in alle richtingen, maar niet even sterk, worden uitgezonden en op afstand vrij zwak zijn. Meestal zijn verhalen over pure gedachtenlezers problematisch. Ze moeten niet opvallen. Dat is logisch, want niemand wil 'gelezen' worden of zijn privacy kwijtraken en de geheime dienst wil je waarschijnlijk 'misbruiken'. Of: Jack Vance beschrijft iemand die op Aarde terugkeert na 'elders' gedachten te hebben leren lezen. Hij wordt knettergek van 7 miljard gedachten zendende breinen en vlucht weer de ruimte in.

Word je een supermens? Hoewel je kwade bedoelingen kunt detecteren, vijanden kunt ontlopen en verborgen personen kunt opmerken, ben je op afstand kwetsbaar. Een verre sluipschutter of aangebrachte boobytraps worden je fataal. Nog een obstakel: denken we in een spreektaal of in een universele biologische code? In het eerste geval kun je buitenlanders zelden 'lezen'.

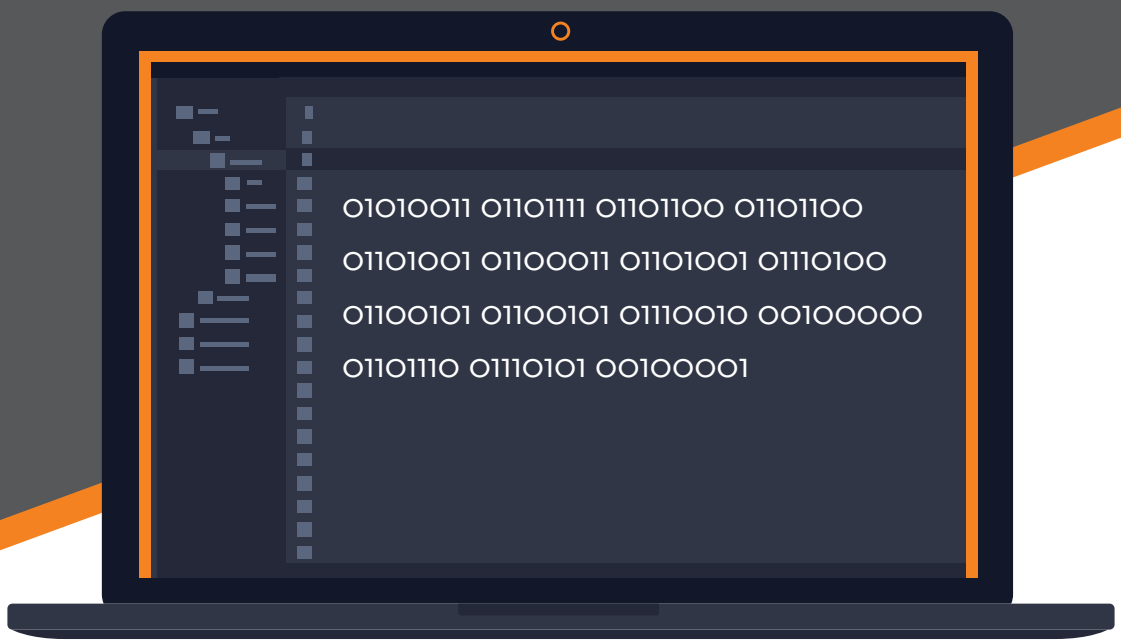
SCHRIJVEN: dit fenomeen is moeilijker te typeren. Waarschijnlijk moet ook hier een soort straling worden gebruikt, maar de zender is niet gemakkelijk te lokaliseren. De ontvanger is vast weer de frontale cortex. Het is de vraag of het slachtoffer zich bewust zal/kan zijn van de actie. Want: zijn we ons bewust van (geslaagde) hypnose? Of zelfs van effectieve intimidatie? Lang niet altijd, maar de ogen spelen een belangrijke rol (e.e.a. gaat OOK met straling gepaard, al is het dan maar gewoon licht!). De pure gedachten-schrijver kan zijn slachtoffer(s) aardig voor de gek houden door zichzelf of de omgeving er anders uit te laten zien, maar ook hier moeten we rekening houden met richting of beperkte afstand. Dus is hier behoefte aan een 'stralingsbundel', anders wordt het moeilijk om doeltreffend te zijn. Je zou dan in een groep mensen alleen maar iedereen tegelijk 'treffen' en dan breekt het verschil in onderlinge afstand, dus intensiteit, je op. Een waarnemer buiten de reikwijdte krijgt je misschien zelfs door!

Het moge duidelijk zijn dat ik de beschreven 'gaven' niet als 'Harry Potter effect' beschouw, maar als een mogelijk fysisch verschijnsel (ik kan mijn achtergrond niet negeren). Want het gegeven dat dit alles van en naar biologische systemen zou moeten 'werken' geeft er een hybride tintje aan. Afgezien van een paar diersoorten die licht kunnen geven en sidderalen die zonder zonnepanelen elektriciteit kunnen opwekken, kennen we eigenlijk geen technische organen bij dieren. Het menselijk ras heeft tot nu toe elke leemte in zijn zintuigen kunnen opvullen met kunstmatige oplossingen, maar ondanks alle wilde verhalen daarover kunnen we de feitelijke gedachten van een persoon niet 'aftappen'. Hersencellen hebben/zijn geen binaire units.

Misschien maar goed ook. Mijn computer en telefoon zijn al lek en van mijn hoofd blijven ze af!

;Joop!

Kun jij lezen wat hier staat?



Dan is Ordina JTech op zoek naar jou!

Laat wat van je horen via jtech@ordina.nl

www.ordina.nl/werkenbij

ORDINA Ahead of change