JAVA { MAGAZINE

.nl.jug

02 > 2021

Onafhankelijk tijdschrift voor de Java-professional

BRAND NEW!

Software & Java Magazine

{ FEATURES

›JAVA 16
NIEUWE FEATURES

›MAVEN 4.0
REACTOR VERBETERINGEN EN MEER

›HTTP 3.0
WAT BETEKENT DIT PROTOCOL VOOR JOU?

AVAVA

TEQJOBS.NL
YOUR NEXT TECH JOB

Senior Full stack developer:

Jij speelt misschien wel dé hoofdrol bij de vertaling van onze strategie naar producten en diensten voor onze klanten. Je werkt mee aan uitdagende functionaliteiten met de nieuwste technologieën. Bovendien speel je een belangrijke rol in onze evolutie naar een DevOps-organisatie. Je leert snel en graag, je ziet waar mogelijkheden tot verbetering liggen en neemt anderen daarin mee. Belangrijke elementen in het Agile werken zoals samenwerken, inspelen op veranderingen en kort cyclisch waarde leveren passen bij jouw manier van werken.

Java Developer bij Caesar Experts

Bij Caesar Experts aan tafel is altijd plek voor ervaren Java developers. Jij bent de consultant die onze klanten ondersteunt in het ontwikkelen, beheren én testen van applicaties. Dit maakt jou de Expert die wij zoeken!

CAESAR EXPERTS

Java developer:

Als Java developer bij de Belastingdienst kun je dan ook voorop lopen binnen jouw vakgebied. Je krijgt bij ons veel kansen en volop de ruimte om jezelf (vakinhoudelijk) te ontwikkelen en door te groeien, bijvoorbeeld door (onder werktijd) deel te nemen aan bijeenkomsten van onze Java vakgroep, conferenties en hackathons bij te wonen, diverse opleidingen te volgen en certificeringen en diploma's te behalen. En dit met inbegrip van goede primaire én secundaire arbeidsvoorwaarden die garant staan voor een goede werk- en privébalans. Bij de Belastingdienst hebben we veel doorgroei mogelijkheden, van junior via medior naar senior. Zo is het mogelijk om na de functie van Senior Java Developer door te groeien naar Java Specialist.

Belastingdienst

Java DevOps Engineer:

Stel je eens voor... dat jij werkt als Java DevOps Engineer in een team dat verantwoordelijk is voor de belangrijkste online betaalmethode van Nederland: iDeal. Bij ons betekent DevOps geen gedefinieerde verantwoordelijkheden of functieomschrijving. Jij bent verantwoordelijk voor de hele levenscyclus. Van idee, tot realisatie en operatie. We volgen dan ook het principe: you build it, you run it! En dat is geen loze belofte. Uit intern onderzoek is naar voren gekomen dat onze IT collega's het werken bij Rabobank IT het meest waarderen vanwege het werken met de nieuwste tech stack, maar ook omdat zij de vrijheid krijgen om nieuwe dingen te proberen én leren. Ervaar het zelf en bekijk deze en andere vacatures op Rabobank.jobs.

Rabobank

IT Engineer:

Bij Grid Mobile & Internet bedenken én bouwen we de digitale winkel waarin we alle digitale producten en diensten van ABN AMRO verkopen. Vanuit de overtuiging dat digitaal bankieren altijd makkelijker, sneller, leuker en persoonlijker kan, ontwikkelen we mooie oplossingen die klanten meer regie over hun bankzaken geven. Daarnaast zijn we voortdurend bezig om ons technisch landschap te moderniseren en hiervoor zijn wij op zoek naar een IT Engineer die weet hoe je samen met je scrum team, sprint op sprint, functionaliteit naar productie brengt.

ABN·AMRO

DevOps Engineer - Rotterdam / Randstad:

Jij kan het verschil maken voor 17 miljoen mensen! Ben jij DevOps engineer en wil je ontdekken hoe jij je skills kunt inzetten om impact te maken in het hart van de Digitale overheid? Dan zijn wij op zoek naar jou!

CGI

Find all these vacancies and more

at teqjobs.nl

{ COLOFON } JAVA MAGAZINE 02-2021

Content Manager & Project Coördinatie:
Stijn van de Blankevoort

Eindredactie:
Tedje van Gils, Jurgen Nijhuis - Argos Media

Auteurs:
Noortje van Berlo, Manon Bosselut, Philippe Charrière, Oussama Chougna, Edwin Derks, Hanno Embregts, Martin Kanters, Bas Knopper, Maarten Koller, Julien Lengrand-Lambert, Maarten Mulders, Bouke Nijhuis, Ben Ooms, Robert Scholte, Jasper Sprengers, Erwin Tinkóczy, Brian Vermeer, Mark van der Walle, Peter Wessels & Ivo Woltring

Redactiecommissie:
Stijn van de Blankevoort, Peter Hendriks, Ben Ooms, Julien Lengrand-Lambert, Maja Reißner, Ted Vinke, Mark van der Walle, Ivo Woltring, Thomas Zeeman

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 € 49,50 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**.

VOORWOORD

Restyle tijd!

De laatste restyle was alweer zo'n 8 jaar geleden dus het was hoog tijd om weer met de tijd mee te gaan en alles wat gelikter te maken. Dus achter de schermen hebben we met de uitgever en de redactiecommissie onze creatieve hoed opgezet en gesleuteld aan het hele magazine. Met dit als resultaat! We horen graag wat je erover denkt op Twitter @JavaMagazineNL .
Maar het is niet alleen gebleven bij een nieuwe vormgevingsstijl, zo hebben we ook een nieuwe rubriek in het leven geroepen: Byte size. Hier vind je hap-snap korte newsflashes, leuke feitjes en meer. Behalve dat dit leuk wegleest, geeft het nieuwe auteurs ook de kans om makkelijk in te stappen met schrijven, zonder dat je meteen pagina's vol moet kalken. Dus was voorheen de stap iets te groot of heb je een klein leuk bericht om te delen met de NLJUG? Stuur een mailtje naar info@nljug.org!
Daarnaast hebben we nóg meer nieuwigheid voor je! Want we zijn weer een Java redactielid rijker geworden! Bij deze wil ik graag Maja Reißner voorstellen! Zoals de ß al doet verklappen, komt ze van origine uit Duitsland maar heeft de Nederlandse taal helemaal meester gemaakt. Verder zet ze zich al goed in voor Java Magazine en zal ons op alle fronten bijstaan, met in het bijzonder security.
En je denkt misschien "Dat was het wel qua nieuwtjes".. Maar nee! Want we mogen ook 3 nieuwe NLJUG bestuursleden verwelkomen. Ze stellen zich voor in het bestuurscolumn, dus lees daar alles over Noortje van Berlo, Brian Vermeer en Bas Knopper!
Verder focussen we in deze editie ons ook op nieuwe software, zoals de nieuwe features van Java 16 en Maven 4.0, een blik op HTTP3 en wat je daar als developer en internetgebruiker mee kan. Daarnaast kan je zien waarvoor je programmeertaal Golo allemaal kan gebruiken en duiken we de dependency hell in met Robert Scholte.
Aangezien we toch op zo'n "nieuw nieuw nieuw"-tour zijn: Wil je als kersverse nieuwe auteur aan de slag? Dat kan! Neem contact met ons op via info@nljug.org en dan kunnen we samen kijken hoe je aan Java Magazine kan bijdragen en uiteindelijk je artikel op oldschool wijze boven je bed kan hangen.
Dan rest mij niet veel meer dan iedereen een gezonde en lekkere zomer(vakantie) toe te wensen!

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

TIKKIE WON'T BE INNOVATIVE FOREVER
#STARTWITHUS
After successfully launching apps like Tikkie and Grip, it would be easy to say "Okay, we made it". But that's not how innovation works. Innovation is about constantly challenging yourself. Daring to acknowledge that something can always be better. Must be better. Because it's this mindset that makes the difference. And to make a difference we need you. Employees that will continue to motivate others, and us.
abnamro.nl/beginbijons
ABN·AMRO

INHOUD

Mastering dependency conflicts

09 We doen het allemaal: zodra we een iets serieuzere applicatie schrijven, maken we gebruik van code die al door iemand anders is geschreven. Deze is meestal beschikbaar gesteld als artifact in een repository zoals Maven Central. Meestal maak je gebruik van meerdere dependencies en deze hebben vaak ook weer eigen dependencies. Zo ontstaat een hele boom aan jars. En hoe groter de boom, des te groter de kans op conflicten.

HTTP/1, 2, ... 3!

25 Zoals de titel doet vermoeden gaat dit artikel over versie drie van het ongeveer dertig jaar oude http-protocol. In dit artikel wil ik je graag op de hoogte brengen van het bestaan van dit protocol. Dat wil zeggen, de draft daarvan. Hoe dat zit, en wat dit 3.0 protocol voor jou kan betekenen in de toekomst als internetgebruiker en developer, lees je in het artikel.

De kunst van het bugfixen

35 Een belangrijk deel van het werk van een ontwikkelaar is het fixen van bugs. Ik zie elke bug als een puzzel. Echter, je hoort en leest er bijna niks over. Met dit artikel wil ik daar verandering in brengen. Ik zal hier uit de doeken doen wat ik de afgelopen jaren heb geleerd over het fixen van bugs.

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.

06 **TOGGLE-SYSTEEM** VAN ERGERNIS TOT FEATURE

09 **MASTERING DEPENDENCY CONFLICTS**

12 **GOLO** THE LITTLE LANGUAGE THAT GIVES YOU SUPERPOWERS

16 **WAT IS ER NIEUW IN MAVEN 4**

20 **SIMPEL MAAR EFFECTIEF**

22 **JAVA 16**

25 **HTTP/1, 2, ... 3!**

27 **TEQNATION 2021 DIGITAL**

32 **AWS** SERVERLESS APPLICATION MODEL

35 **DE KUNST VAN HET** BUGFIXEN

38 **TEAMSFEER EN BETROKKENHEID** EN DAN PAS HET GEREEDSCHAP

40 **PATTERN MATCHING** MAAKT JAVA NÓG KRACHTIGER

43 **TEKTON** CLOUD NATIVE CI/CD

46 **AUDITING OP ENTITEITEN** MET ENVERS

51 **BESTUURSCOLUMN**

52 **COLUMN MAVEN**

53 **COLUMN JOOP**

54 **BYTE SIZE**

TOGGLE-SYSTEEM
Van ergernis tot feature

toggle

Noun
- COMPUTING
a key or command that is operated the same way but with opposite effect on successive occasions.

Verb
- COMPUTING
switch from one effect, feature, or state to another by using a toggle.
“the play/pause button toggles between those functions”

In het werk van een developer wordt vaak gesproken over feature toggles (of flags): een manier om functionaliteit aan of uit te zetten. Volgens de bovenstaande definitie zou een toggle een simpele aan/uitknop moeten zijn. In de praktijk is dat vaak niet zo en zijn er ingewikkelde handelingen nodig om te togglen. Een jaar geleden besloot ik daarom een eigen feature-toggle-systeem te bouwen.
In mijn devOps-team bij de Rabobank gebruiken we ook (feature) toggles. In onze code zit een koppeling met een extern properties-bestand dat op een config-server staat. In de code lezen we een boolean property uit. Om van true naar false te gaan, moeten we het properties-bestand aanpassen en de gewenste waarde intypen. Aangezien dit een verandering van productiecode betreft, wordt er een pull request gemaakt. Nadat een collega het pull request heeft goedgekeurd en gemerged, moet de config-server een refresh krijgen. De nieuwe waarde van de property is daarna beschikbaar voor de applicatie, maar die moet hem nog wel inlezen. Dus nog even een herstart en we zijn eindelijk getoggled, hoera ...

{ HERKENBAAR? }

Volgen we de definitie van een toggle dan zouden we iets moeten hebben dat werkt als een lichtknopje: simpel, makkelijk in gebruik met een direct merkbaar effect dat even zo snel omkeerbaar is. Zelfs een Product Owner zou het moeten kunnen :-).

Maarten Koller *is Software Engineer bij Ordina. Hij verdiept zich graag in teamcultuur, verbeterprojecten en Flutter.*

Ik begon over dit soort zaken na te denken vanwege ergernis over een jaarlijks terugkerende taak. Onze applicatie toont gebruiksvoorwaarden voor klanten, die ieder jaar vernieuwd worden. We laten collega’s zo dicht mogelijk bij 1 januari de juiste bestanden activeren (op 31 december om 23:00 bijvoorbeeld). Waarom konden we dit niet met één klik doen? Of inplannen zodat het automatisch gebeurt? We’re lazy developers damn it!
Zo’n jaarlijkse taak is het natuurlijk niet waard om een eigen feature-toggle-systeem voor te ontwikkelen – behalve dan dat het maken ervan leuk is. Het hebben van zo’n systeem zou er ook voor kunnen zorgen dat we het vaker gebruiken. Hoe langer ik erover nadacht, hoe meer taken met toggle-potentieel ik zag. Hoe ga ik dit aanpakken?

{ KILLER FEATURES }

Het doel is natuurlijk om het werk van mij en mijn collega’s makkelijker en betrouwbaarder te maken. Om hen te overtuigen moest mijn app beter zijn dan de huidige oplossing. Makkelijk van true naar false kunnen togglen is de basis, maar dit is niet voldoende. Daarom bedacht ik de volgende killer features:

> **Inplannen van een toggle**

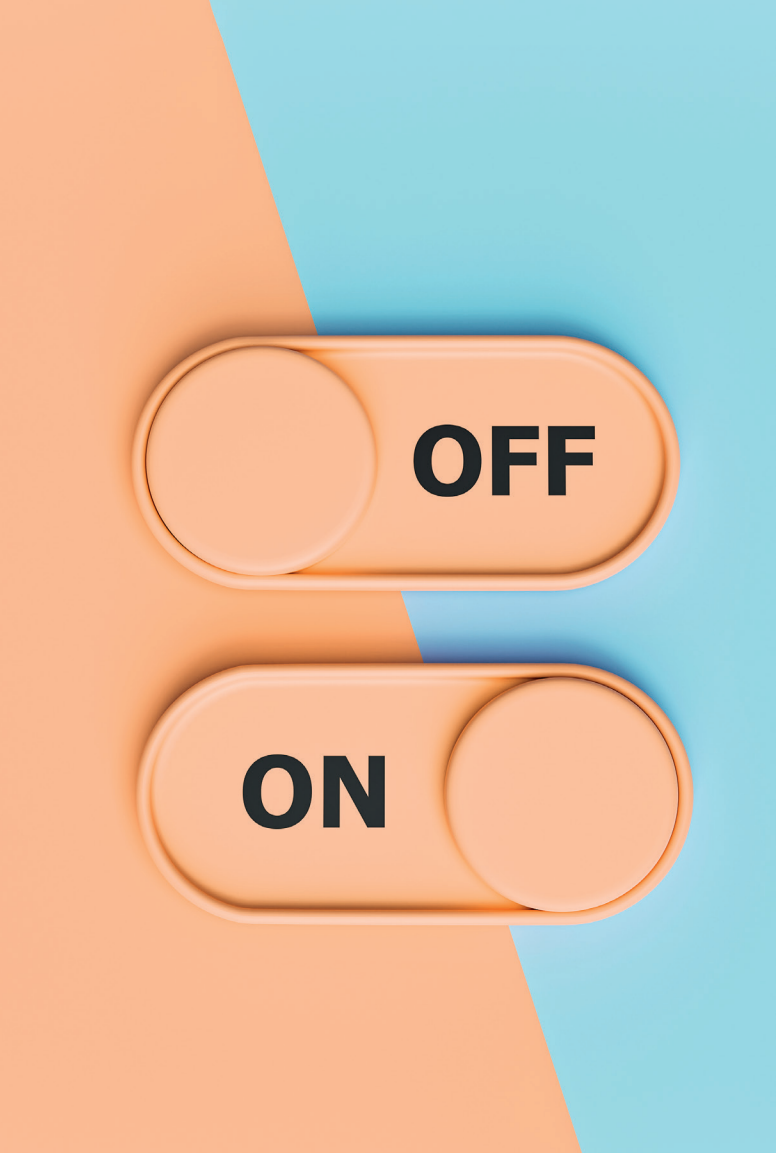
Ik wilde een datum en tijd kunnen selecteren waarop een toggle naar een gekozen waarde verandert.

> **Canary testing: one canary, two canary, n-canaries**

Om canary testing te faciliteren moest een toggle ingesteld kunnen worden om n keer true te geven, om daarna weer terug te keren naar false.

> **Canary testing: percentage**

Ik wilde ook het testen met een variabel percentage mogelijk maken.



› Regenerative Toggles

Het enige moment waarop je je druk zou moeten maken over de naam van je toggle, is tijdens het implementeren ervan. Het dashboard moet een afspiegeling zijn van de implementaties, zonder iets in het dashboard te hoeven doen. Als je een toggle per ongeluk uit het dashboard verwijdert terwijl hij in de code nog bestaat, moet het dashboard zichzelf uiteindelijk herstellen. Ik noem het: Regenerative Toggles ... Bingo!

› Een toggle-dashboard

Via een duidelijke interface moet je de toggles kunnen aflezen, instellen, bedienen en verwijderen.

{ FAIL FAST }

There is nothing so useless as doing efficiently that which should not be done at all. - Peter Drucker

Het fail fast-principe zegt dat we zo snel mogelijk moeten stoppen als er iets fout gaat. Iets doen wat al gedaan is, vind ik erg fout, dus ik begon met de vraag: vind ik het wiel opnieuw uit? Ik vond mijn idee zelf natuurlijk uitermate origineel, maar daar kwam na een rondje langs verschillende collega's en afdelingen snel verandering in. Onze Spring Boot-applicaties draaien in Pivotal Cloud Foundry (PCF) en de lokale PCF-evangelist vertelde me dat PCF geen passende oplossing had, maar vroeg "of ik al aan refresh endpoints had gedacht?". In plaats van een herstart van je applicatie om je properties opnieuw in te laden, roep je slechts een endpoint aan. Het is standaard functionaliteit van Spring, maar wij kenden het nog niet. Bedankt Lars!

Nog lang niet tevreden zocht ik contact met verschillende (IT-support) teams. Daar bleken ze mijn plan een goed idee te vinden. Zo'n goed idee zelfs, dat ze nét een vooronderzoek naar Split.io als succesvol hadden bestempeld. Split is een enterprise-ready feature-toggle-systeem met uitgebreide mogelijkheden & metrics. Om mijn systeem nog enige kans van slagen te geven moest ik nu opeens zorgen dat het eerder in productie beschikbaar zou zijn dan Split.io.

Ik moet helaas bekennen dat ik er pas tijdens het schrijven van dit artikel achterkwam dat er verschillende opensource-toggle-projecten bestaan (bijv. togglz.org). Achteraf gezien had ik die misschien in mijn applicatie moeten gebruiken, maar dan had ik natuurlijk wel minder geleerd.

{ ALLES KWIJT? }

Vol goede moed startte ik met bouwen. De Spring Initializr maakt een Spring Boot-applicatie beginnen lekker makkelijk en ik doopte mijn applicatie AwesomeFeatureToggles. Voor minder dan awesome ging ik niet en het was ook de naam van mijn toenmalige team. Daarnaast bekt AFT ook zo lekker (pun intended!). Al snel maakte ik een endpoint dat false terug gaf. Zo had AFT binnen enkele minuten al de helft van de beoogde functionaliteit! Het begon pas echt ergens op te lijken wanneer ik ook true kon retourneren. Dus nóg een endpoint voor het togglen zelf. Daarnaast nog endpoints voor het verwijderen, inplannen en canary-testen. Ik maakte in Angular een product owner-proof dashboard. Daarvoor had ik nog een endpoint nodig voor het ophalen van een lijst met alle toggles. Al met al had ik een prettig 'lekker bezig!'-gevoel.

Voor een applicatie is de implementatie van AFT simpel: een GET op het AFT-endpoint `(toggles/{toggleId}/value)` retourneert de boolean-waarde.

Vanuit het fail fast-principe en omdat ik graag collega's meeneem in mijn plannen, besprak ik AFT geregeld met hen. "Goh", merkte iemand op, "je houdt de toggles in-memory bij?", "Klopt, in een hashmap, lekker snel!", beaamde ik. "Zeker, maar als AFT herstart ben je dus alles kwijt?".

{ MILLISECONDEN }

Oeps, daar had ik even niet aan gedacht. Om mijn collega's te overtuigen AFT te gebruiken, moest het een herstart overleven. De oplossing is natuurlijk een database.

De database fungeert als backup voor de toggles. Wanneer AFT herstart, wordt de hashmap opnieuw gevuld vanuit de database en bij elke verandering aan een toggle wordt de database geüpdatet. Dit werkte prima in de testomgeving. In de productieomgeving zijn er echter twee van elkaar gescheiden zones. De database is wel gezamenlijk. Maar als ik in de ene zone een

toggle veranderde en daarmee de database, had de hashmap in de andere zone nog de verouderde informatie. Ik moest dus elke verandering in beide zones uitvoeren tenzij ik een manier vond om ze te synchroniseren.
Wat nu als ik de toggle-waarde uit de database zou retourneren in plaats van de hashmap? Om dit te testen, bouwde ik nog een rechtstreeks-naar-de-database-endpoint en daarna gaf Chrome DevTools het antwoord. Het verschil bleek een orde van grootte: de hashmap gaf resultaat in 30 ms. De database in 300ms. Voor de MVP koos ik daarom voor de hashmaps met de database als backup. Synchroniseren kwam later wel.

{ REGENERATIVE TOGGLES }

Zodra AFT een request krijgt voor een toggle waarvan de naam niet in de hashmap voorkomt, wordt de toggle aangemaakt met de standaardwaarde false. Vervolgens is de toggle beschikbaar op het dashboard. Verwijder je via het dashboard (per ongeluk) een toggle terwijl er nog een applicatie is die de waarde ervan opvraagt, dan verschijnt de toggle op zo'n moment dus weer op het dashboard alsof je hem nooit verwijderd had.
Als je een toggle wilt verwijderen moet je dat dus eerst doen op de plek waar het ertoe doet: in de code. Daarna kan je hem via het dashboard verwijderen en blijft hij ook weg.

{ VERANTWOORDELIJKHEID }

In AFT zorgt de eerste klant/gebruiker die door een applicatie gaat waarin een toggle-waarde wordt opgevraagd dus ook voor het aanmaken van de toggle. Je zou dus kunnen zeggen dat de 'verantwoordelijkheid' van het aanmaken daarmee bij de gebruiker wordt gelegd. Een collega vond dat dit niet de bedoeling was. De plek waar deze verantwoordelijkheid zou moeten liggen, bleek voer voor een discussie van bijna een uur. Ergens was ik het wel met hem eens. Tegelijkertijd had ik moeite met het opgeven van mijn idee. Gelukkig had hij veel geduld en zijn we uiteindelijk tot een compromis gekomen: ook via het dashboard zou je voortaan een toggle kunnen aanmaken.
De voornaamste overweging was dat het gebruik in een applicatie met weinig gebruikers zou kunnen leiden tot een lange wachttijd voordat de toggle eindelijk op het dashboard zou verschijnen. Door het handmatig aanmaken hoef je niet te wachten met het instellen of inplannen van de toggle en kan je ook de allereerste gebruiker een true teruggeven.
Bijkomend voordeel van de regenerative toggles is dat het minder secure collega's helpt. Als je in het dashboard een toggle 'foo' aanmaakt terwijl je in de code 'bar' hebt gebruikt dan zal ook de 'bar'-toggle vanzelf op het dashboard verschijnen zodra die voor het eerst wordt opgevraagd. Je hoeft dan dus niet meteen de code in te duiken om de naam aan te passen en opnieuw te deployen.

{ DRAAGVLAK }

Na al deze moeite moest de applicatie natuurlijk wel gebruikt gaan worden door mijn collega's. Want iets wat elke start-up zal beamen: tussen zeggen dat je iets tof vindt en het daadwerkelijk gebruiken zit een wereld van verschil. Hoe krijg ik mijn collega's zover? Vanaf het begin heb ik mijn teamgenoten betrokken bij de problemen waar ik tegenaan liep. Ze kenden AFT dus al enigszins. Ik heb ze daarnaast ook gewoon op de man af gevraagd: "Als ik dit in productie zet, zou je het dan gebruiken?". Als dat niet zo was, vroeg ik wat daarvoor nodig zou zijn en bouwde dat.
Een keuze waarvan ik juist dacht dat het zorgen zou wegnemen, heb ik daarom later teruggedraaid. Zo had ik ervoor gezorgd dat een fout rondom AFT (van een bug in de implementatie van een toggle, tot een bug in AFT zelf) altijd een false zou retourneren. Het grote voordeel is dat je applicatie blijft werken, al dan niet met de gewenste functionaliteit. De fout zou je kunnen loggen en daar een alert op zetten. Dit week nogal af van het fail fast-principe en kwam voort uit mijn onzekerheid: ik wilde nergens per ongeluk een crash veroorzaken. Ergens geloofde ik misschien niet dat het allemaal zou (blijven) werken? Mijn collega's waren het er niet mee eens: "Gebruiken we een extern feature toggle-systeem, dan gaan we er ook voor: faalt er iets, gooi dan gewoon een exceptie". Zo gezegd, zo gebouwd.
Een andere 'techniek' had ik juist beter wat minder kunnen gebruiken. Tijdens de gebruikelijke Agile-rituelen zoals standups en refinements werden alle problemen met onze software besproken. Ik riep vaak dat een feature-toggle de oplossing zou kunnen zijn. Zo vaak, dat ik op een gegeven moment de opmerking kreeg dat ik een nieuwe hamer had en nu overal spijkers zag. Misschien moet ik dus nog wat werken aan mijn growth hacking-skills.

{ VUURDOOP }

Hoe dan ook, het moment suprême naderde: op 31-12-2020 zouden er weer overeenkomsten moeten worden klaargezet; de ontstaansreden van het hele project. Hoe mooi zou het zijn als het nu, een jaar later, daarvoor ingezet kon worden? Ik was dus enorm blij en trots toen ik op 16 december 2020 dit bericht ontving:

En, op 1-1-2021 bleek: hij deed het! Hey Split, I won!

LINKJES

Awesome Feature Toggle service: https://github.com/MaartenKoller/awesome-feature-toggle-service
Awesome Feature Toggles (Dashboard): https://github.com/MaartenKoller/awesome-feature-toggles

MASTERING dependency conflicts

We doen het allemaal: zodra we een iets serieuzere applicatie schrijven, maken we gebruik van code die al door iemand anders is geschreven. Deze is meestal beschikbaar gesteld als artifact in een repository zoals Maven Central. Deze projecten bieden vaak bewezen oplossingen voor problemen waar we mee te maken hebben. En daar is niks mis mee. Meestal maak je gebruik van meerdere dependencies en deze hebben vaak ook weer eigen dependencies. Zo ontstaat een hele boom aan jars. En hoe groter de boom, des te groter de kans op conflicten. Om te begrijpen waardoor conflicten ontstaan is het allereerst goed om uit te leggen wat de classpath, modulepath en classloader is en wat hun rol is bij dependency conflicten.

Robert Scholte *is de voorzitter van het Apache Maven project en schrijft al jaren columns voor Java Magazine.*

{ CLASSPATH }

De classpath is een "ordered list" van jars en/of directories die meegegeven wordt bij het opstarten van de JVM. Voor elke class die nodig is, zal de JVM door de lijst van classpath-entries lopen totdat de class gevonden is. Het is dus heel goed mogelijk, dat een class meerdere keren voorkomt op de classpath, echter zal de eerste altijd winnen.

Het is daarnaast ook mogelijk om een directory met wildcard op te geven, bijvoorbeeld: `/path/to/jars/*`. In dit geval worden alle jars in deze directory aan de classpath toegevoegd. Maar let op: het OS bepaalt dan de volgorde van de jars, en dat is niet per definitie alfabetisch!

Met andere woorden: "Order matters!" Als de volgorde dus belangrijk is, maak dan geen gebruik van wildcards. Maven maakt zelf geen gebruik van wildcards, maar bouwt een classpath op met expliciete referenties naar outputDirectories (target/classes of target/test-classes) en jars.

Als je bovenstaande goed gelezen hebt, zie je wellicht een truc om de classloader te misleiden. Je kunt namelijk (bijna) elke classfile vervangen door in je sources de implementatie van deze class aan te passen. Omdat de sources vóór de dependencies komen, wint de nieuwe implementatie. Dit is enerzijds een hele krachtige, maar tegelijkertijd ook potentieel gevaarlijke optie. Deze truc kan namelijk ook door de dependencies uitgevoerd worden, soms goedschiks maar mogelijk ook kwaadschiks. Waar het bijvoorbeeld zinvol wordt gebruikt is bij elke `<logger>-over-slf4j` implementatie. Hierdoor is het mogelijk om bekende logging frameworks met een aangepaste implementatie via slf4j naar één definitieve logger te routeren.

{ MODULEPATH }

In tegenstelling tot de classpath kan de modulepath beter omschreven worden als een Map. Elke jar op de modulepath kan via een module descriptor aangeven van welke andere modules hij afhankelijk is. Daarnaast bevat elke module descriptor ook een lijst met packages die door anderen gebruikt mogen worden. Door deze administratie kan een class veel sneller gevonden worden. Mocht een jar geen module descriptor bevatten, dan heeft deze toegang tot alle jars op de module path. Dit is gedaan om de overgang van classpath naar modulepath te vereenvoudigen.

Ik heb zelf de modulepath altijd omschreven als een striktere versie van de classpath. Ze werken beide hetzelfde, maar daar waar de classpath alles toestaat, zul je je bij de modulepath aan een aantal regels moeten houden. Bij het opbouwen van de modulepath worden eerst allerlei controles gedaan voordat JVM verder gaat. Wordt er niet voldaan aan alle eisen, dan wordt het proces meteen afgebroken. Dit in tegenstelling tot de classpath, die pas een RuntimeExpection gooit als het probleem zich voordoet. In productie is dat vaak te laat. Een aantal van deze controles zijn:

- Alle vereiste modules moeten aanwezig zijn. Dit is puur op basis van naam. Welke versie van de module gebruikt moet worden wordt overgelaten aan buildtools als Maven of Gradle.
- Er mogen geen split packages zijn. Dit betekent dat als er een jar is met de class nl.jug.conference.Location, er in andere jars geen classes mogen voorkomen met de package nl.jug.conference. Sub- of superpackages mogen wel, oftewel andere jars zouden wel classes mogen bevatten met de package nl, nl.jug, nl.jug.magazine of nl.jug.conference.timetable, zolang het maar niet nl.jug.conference is.
- Elke entry moet een geldige module name hebben, impliciet of expliciet. Als de jar geen expliciete naam heeft, dan wordt geprobeerd om de modulename aan de hand van de bestandsnaam te bepalen.

{ CLASSLOADER }

Alle classes belanden in een zogenoemde classloader. Classloaders zijn hiërarchisch en geïsoleerd van elkaar. De Java runtime bevat 3 built-in classloaders: bootstrap, platform en system classloader. Deze laatste wordt ook wel application classloader genoemd. Wat je meegeeft aan de modulepath of classpath komt in deze laatste classloader terecht. Daarnaast het is ook mogelijk om zelf classloaders aan te maken. Dat is vooral handig voor op plugin gebaseerde architecturen. Denk bijvoorbeeld aan Maven: zonder classloaders zouden maven-plugins allemaal dezelfde jars moeten gebruiken.
Tot zo ver een aantal details over hoe de JVM werkt. Laten we nu kijken naar dependencies en versies.

{ SEMANTIC VERSIONING }

Op de website semver.org staat het volgende:
Given a version number MAJOR.MINOR.PATCH, increment the: MAJOR version when you make incompatible API changes, MINOR version when you add functionality in a backwards compatible manner, and PATCH version when you make backwards compatible bug fixes.

Waarschijnlijk herkent iedereen dit wel. Het lijkt heel eenvoudig en is op bijna elk project wel toe te passen. Daar zit de complexiteit niet in. Maar wat als we gaan libraries gaan combineren met verschillende versies, wat betekent dit dan?

In het schema zie je de dependencies van de Animal Market-applicatie en de bijbehorende services. Je ziet dat Buy Service gecompileerd is met Currency Util 1.6.3, maar dat Sell Service is gecompileerd met Currency Util 2.0.0. Afzonderlijk zullen beide services het prima doen, maar er is een kans dat het gezamenlijk nooit gaat werken. Immers, de classpath zal altijd de eerste class kiezen. Het vervelende is dat je er tijdens het compileren van Animal Market er niet achter komt: deze kijkt alleen naar de sources van Animal Market en doet geen sanity-check van de reeds gecompileerde classes.
Stel dat Euro.class van 1.6.3 als eerste op het classpath staat en SellService gebruikt een methode geïntroduceerd in 2.0.0, dan krijg je pas tijdens runtime een Error.
Omgekeerd, als Euro.class van 2.0.0 als eerste op het classpath staat en BuyService gebruikt een constructor die verwijderd is, ook dan krijg je pas gedurende runtime een Error. Daarmee is ook aangetoond, dat de laatste niet altijd de beste keuze is. Als Sell Service geen gebruikt maakt van de nieuwe of breaking changes in 2.0.0, dan is het beter om gebruik te blijven maken van 1.6.3, totdat de BuyService ook gebruik maakt van Currency Util 2.0.0.

{ MAVENS DEPENDENCY RESOLUTION }

Als we nogmaals kijken naar het schema van de Animal Market, dan zien we Currency Util twee keer gedefinieerd. Echter zal Maven dit optimaliseren. Ervan uitgaande dat beide Currency Utils dezelfde groupId en artifactId gebruiken, dan zal Maven

alleen slechts één op het classpath zetten. Dit gebeurt volgens het principe “Nearest wins”. Met andere woorden: de dependencies die in pom opgesomd staan, de directe dependencies, bepalen welke versie van een dependency gebruikt worden. Daarna wordt laag voor laag door de indirecte dependencies gegaan, ook wel bekend als transitieve dependencies.
Op dit moment wordt de Currency Util 1.6.3 gebruikt, omdat deze als eerste transitieve dependency genoemd staat. Om ervoor te zorgen dat Currency Util 2.0.0 gebruikt wordt, zijn een aantal mogelijkheden:

- Voeg aan de dependencyManagement van AnimalMarket de dependency CurrencyUtil 2.0.0 toe. Voor elke dependency zal Maven controleren of er een “gemanagede” variant bestaat, waar globale configuratie staat zoals de versie.
- Voeg aan de dependencies van AnimalMarket de dependency CurrencyUtil 2.0.0 toe. Daarmee is het een direct dependency geworden en wint deze versie. Dit heeft alleen de voorkeur als de code van AnimalMarket zelf ook gebruik maakt van Currency Util. Stel dat zowel BuyService als SellService geen gebruik meer willen maken van Currency Util, heeft AnimalMarket deze dependency dan nog wel nodig?

{ EXCEPTIONS }

De runtime exceptions veroorzaakt door dependency conflicten zijn allemaal terug te herleiden naar een probleem met een class. Elk situatie heeft zijn eigen strategie om het op te lossen.
Class is niet aanwezig. Een handige manier om de bijbehorende dependency terug te vinden is via Maven Central. Ga naar **https://search.maven.org/** en zoek naar fc:naam.van.de.klasse (fc staat voor Full Classname).
Class is 1 keer aanwezig, maar is niet compatible. De meeste IDEs kunnen je helpen met het vinden van versieconflicten van dependencies. Class is meerdere keren aanwezig, maar de eerste is niet compatible. Ook hierbij kunnen de meeste IDEs helpen met het vinden van de jars waar een class in voorkomt.

Veel developers vallen terug op dependency:tree, maar ik denk dat de IDE veel beter in staat is met het analyseren van dit soort problemen. Daarnaast werkt de -Dverbose vlag, die ook de versieconflicten toont, niet bij alle plugin versies. Versie 2.10 is de laatste versie waarbij deze nog werkte, maar het resultaat kan licht afwijken van de interne dependency resolution van Maven 3. Met de komst van maven-dependency-plugin 3.0.0 zijn de resultaten weer gelijk, maar kon de verbose flag niet eenvoudig ondersteund worden. Met versie 3.2.0 is de verbose optie weer terug, maar deze moet op het moment van schrijven nog gereleased worden.

{ ENFORCER RULES }

Er zijn tal van enforcer rules die kunnen helpen met het tijdig detecteren van mogelijke class incompatibilities. De meesten zijn genoeg te finetunen met expliciete includes/excludes, om het gedrag voor jouw project te optimaliseren.

`dependencyConvergence (Apache Maven)`
Deze rule dwingt af, dat alle gelijke dependencies allemaal exact dezelfde versie hebben. Dit is de ideale situatie, maar alleen mogelijk bij kleine projecten, of waarbij je controle hebt over (bijna) alle dependencies.

`requireUpperBoundDeps (Apache Maven)`
Deze rule is vergelijkbaar met dependencyConvergence, maar dwingt af dat de hoogste versie gebruikt. Deze rule gaat er dus vanuit, dat dependencies altijd backwards compatible zijn.

`banDuplicateClasses (MojoHaus)`
Wat deze regel doet, zal geen verrassing zijn: het zal falen als dezelfde class voorkomt in verschillende jars. Dit komt vaker voor dan je lief is.

`LinkageCheckerRule (Google)`
Deze rule doet iets heel interessants: het probeert te achterhalen of alle classes, methodes, enzovoorts, beschikbaar zijn. Hiermee word je tijdens het bouwen al beschermd tegen LinkageErrors at runtime.

{ OSGI }

Er is nog een andere manier om conflicten te voorkomen, maar dat heeft een behoorlijke impact voor de architectuur van de applicatie. Ik heb al uitgelegd dat het mogelijk is om met classloaders plugins te isoleren. Maar je kunt dit nog verder doortrekken: geef elke jar een eigen classloader. Dat is in de basis het idee achter OSGi. Hierdoor is het mogelijk om meerdere versies te gebruiken terwijl OSGi zorgt voor de juiste hiërarchie tussen de bundles (dat is in OSGi de gebruikte term voor een artifact). Net als een Java module heeft OSGi een descriptor nodig. Daarvoor wordt gebruikgemaakt van de MANIFEST.MF met een eigen set aan headers.

{ CONCLUSIE }

Dependency conflicten zijn van alle tijden, maar met de groei van de applicaties en de eenvoudige toegang tot third party dependencies wordt de kans op conflicten groter en groter. Eén van de kerntaken van buildtools is het oplossen van versieconflicten, maar uiteindelijk is het de developer die bepaalt welke versies gebruikt moeten worden. Met tools als Maven Enforcer Rules is het mogelijk om al tijdens compile time potentiele problemen te detecteren.

LINKS

https://maven.apache.org/enforcer/enforcer-rules/index.html
https://www.mojohaus.org/extra-enforcer-rules/
https://github.com/GoogleCloudPlatform/cloud-opensource-java/wiki/Linkage-Checker-Enforcer-Rule

GOLO the little language that gives you superpowers

Golo is a simple dynamic programming language for the Java Virtual Machine that has been designed since the beginning (Java 7) to leverage the capabilities of the Java invokedynamic instruction and API (JSR 292).

Golo has its language constructs designed with invokedynamic in mind. Golo is an exciting language for rapid prototyping, polyglot application embedding, research (e.g. runtime extensions, language prototyping) and teaching (e.g. programming, dynamic language runtime implementation).

Philippe Charrière *is an old geek and a Twitter addict (please please follow me: @k33g_org). When I finishe my task at my day job (at GitLab), I am a core committer a Golo Hero. And I love* 🍻 , 🍷 *and* 🍔.

{ INSTALLATION AND INTRODUCTION }

It's always difficult to present a programming language with few words. For the author, it's frustrating because you want to talk about everything. To understand a new language quickly the best and shortest way is to provide expressive samples. So let's start with the mandatory "👋 Hello World 🌍".
The simplest way to install Golo is to download the latest release (3.3.0)[5] and:

```
- Unzip golo-3.3.0.zip
```

- And add the lines below to your path:

```
export GOLO_HOME='/path/to/golo-distribution/golo'
export PATH=$PATH:$GOLO_HOME/bin
```

A sophisticated "hello world"

```
module helloWorld

struct Person = { name }

function main = |args| {
  let bob = ImmutablePerson("Bob")
  let message = |person| -> "👋 Hello World 🌍\nI'm
" + bob: name()
  println(message(bob))
}
```

The output will be:
👋 Hello World 🌍
I'm Bob

If you want to test all the examples for real, you can use GitPod by using this url [6], then navigate to the appropriate directory to run the code:

```
cd 01-hello-world
golo golo --files hello-world.golo
```

There are several ways to run a Golo program (you can even compile it to a class or jar file). You can find more details on the golo lang documentation[7].

{ GOLO IS FUNCTIONAL }

Functions are the first-class citizens in Golo. But Golo comes with other functional powers.

No iteration

Suppose you are a "serious functional programmer", you have to try to avoid iteration. It's pretty easy with Golo because lists have the necessary methods like map, filter, reduce, each ... But, let's create my favourite sandwich. In the source code below, I want to "prepare a kebab" from an ingredients list (ingredients). You have 3 closures (noPeppers, noOnions, noFries) that you will use as predicates with the filter method of the ingredients list. Every time you call filter you obtain a new list. Then, you will use a fourth closure (pick) to get a new list from a transformation of every ingredient by calling the map method of the last list. And finally, you will transform the new list of ingredients to a string by calling the reduce method with the fifth closure mixIngredients as parameter. By the way, you can see that you can chain the list methods. In Golo, the ':'-notation is for calling the instance methods while the '.'-notation is for class methods.

```
module kebab

# How to run it:
# golo golo --files kebab.golo
```

```
function main = |args| {

  # these closures return a boolean value
  let noPeppers = |item| -> item isnt "🌶"
  let noOnions  = |item| -> item isnt "□"
  let noFries   = |item| -> item isnt "🍟"

  # these closure return a string value
  let pick = |item| -> "piece of " + item

  let ingredients = list["🍅", "□", "🥦", "🍖",
"🌶", "🍟"]

  # I can compose my own kebab: I don't want fries,
onions and peppers
  let myFavouriteRecipe =
     ingredients              # ["🍅", "□", "🥦", "
🍖", "🌶", "🍟"]
       : filter(noFries)      # ["🍅", "□", "🥦", "
🍖", "🌶"]
       : filter(noOnions)     # ["🍅", "🥦", "🍖",
"🌶"]
       : filter(noPeppers)    # ["🍅", "🥦", "🍖"]
       : map(pick)            # ["piece of 🍅", "piece
of 🥦", "piece of 🍖"]

  println(myFavouriteRecipe)

  # Deliver the kebab
  let mixIngredients = |accItem, nextItem| ->
accItem + nextItem + " "

  let kebab = myFavouriteRecipe: reduce("🥙 with ",
mixIngredients)

  println(kebab)
  # 🥙 with piece of 🍅 piece of 🥦 piece of 🍖 piece
of 🌶
}
```

{ HANDLING ERRORS }

A functional programming language without lovely types dedicated to errors handling is not a functional programming language. Golo provides the appropriate functional types for that, as well as some useful helpers. For example, if you decorate a function with the '@option' decorator, the function will return an Optional:

```
import gololang.Errors

@option
function animal = |name| {
  let animals = map[
    ["panda", "🐼"],
    ["tiger", "🐯"],
    ["bear", "🐻"],
    ["lion", "🦁"]
  ]
  return animals: get(name)
}
```

And you can use the result of the function with a map or an either method:

```
let panda = animal("panda")
              : map(|emoji| -> emoji)
              : orElseGet(-> "😡")

animal("cow"): either(
  default= -> println("😡"),
  mapping= |value| -> println(value)
)
```

Golo provides a similar type: Result. This "keeps" the errors (it encapsulates the type of error in the form of a Throwable instance that can be raised later).

```
@result
function toInt = |value| {
  return java.lang.Integer.parseInt(value)
}
```

And you can use it like this, for example:

```
toInt("42"): either(
  recover= |error| -> println(error),
  mapping= |value| -> println(value)
)
```

Or like that:

```
let error, value = toInt("fourty-two")

if (error oftype NumberFormatException.class) {
  println(error)
} else {
  println(value)
}
```

Read the documentation for more methods and details about error handling [8].

{ GOLO LOVES JAVA }

Another very convenient quality of Golo is its capacity to use and augment existing Java classes. Then, the field of actions becomes very large, allowing you to define your DSL to simplify complex use cases drastically. I'm using this 'superpower ' with my internet

of things experiments. I want to embed an MQTT broker on Raspberry PI Zero, but I need something easily and quickly updatable and improvable. Let see how you can do that:

- For this example, you'll build a jar file containing a Vert.x distribution with the MQTT helpers.
- All the source code is available here [9]

{ CREATE A GOLO MODULE }

You'll first create a Golo module, and embed properties (fields) and method augmentations in this module. Module-level references (mqttClients and mqttSubscriptions) are only visible from their module, although a function may provide accessors to them. You can see this module like a mix of class, interface and traits.

```
module mqttModule

# store connected clients and subscriptions
let mqttClients = map[]
let mqttSubscriptions = map[]
```

Then, you'll add an improved get method to java.util.Map to get an Optional when you look for an element in this map and simplify my nulls management:

```
augment java.util.Map {
  @option
  function getOptional = |this, key| -> this:
get(key)
}
```

Remark: an augmentation function takes the receiver object as its first argument, followed by optional arguments. The main objective is to simplify the management of the clients and subscriptions. For that, you'll add four methods to the MqttServer object:

- updateClients
- updateSubscriptions
- mqttClients (accessor to the mqttClients)
- mqttSubscriptions (accessor to the mqttSubscriptions)

```
augment io.vertx.mqtt.MqttServer {

  function updateClients = |this, endpoint| {
    endpoint: accept(false)
    # update clients connection
    mqttClients: put(endpoint: clientIdentifier(),
endpoint)
    println("🤖 connected client: " + endpoint:
clientIdentifier())
  }

  function updateSubscriptions = |this,
clientIdentifier, subscriptionRequest| {
    # update clients subscriptions
    subscriptionRequest: topicSubscriptions():
each(|subscription| {
      mqttSubscriptions: put(
        clientIdentifier + "-" + subscription:
topicName(),
        subscription
      )
    })
    println("😊 new subscriptions(s): " +
subscriptionRequest: topicSubscriptions(): head())
  }

  function mqttClients = |this| -> mqttClients

  function mqttSubscriptions = |this| ->
mqttSubscriptions
}
```

And finally, since the needs for publishing messages between connected objects are very simple (in the current context), you'll add to the MqttEndpoint a simplePublish method to encompass and call the "real" publish method with default parameters:

```
augment io.vertx.mqtt.MqttEndpoint {
  function simplePublish = |this, topic, payload| {
    this: publish(
      topic,
      Buffer.buffer(payload: toString()),
      MqttQoS.AT_LEAST_ONCE(),
      false,
      false
    )
  }
}
```

{ NOW, LET'S USE YOUR NEW GOLO MODULE }

From now on, you are able to write MQTT brokers very quickly to embed them on a PI Zero to make gateways on demand easily. This is the main code:

```
import mqttModule

function main = |args| {

  let vertx = Vertx.vertx()
  let mqtt_options = MqttServerOptions(): port(1883)
  let mqtt_server = MqttServer.create(vertx, mqtt_
options)

  mqtt_server: endpointHandler(|endpoint| {

    mqtt_server: updateClients(endpoint)
```

```
    endpoint: subscribeHandler(|subscriptionRequest|
{
      mqtt_server: updateSubscriptions(endpoint:
clientIdentifier(), subscriptionRequest)
    })

    endpoint: publishHandler(|message| {
      # at every message
      mqtt_server: mqttClients(): each(|identifier,
client| {
        # check for each client if a subscription
exists
        mqtt_server: mqttSubscriptions():
getOptional(identifier + "-" + message: topicName())
          : either(
              default= -> println("✋ no
subscription for this client"),
              mapping= |subscription| ->
                # send message to the subscribed
client
                client: simplePublish(
                  message: topicName(),
                  message: payload()
                )
            )
      })
    })
  })

  mqtt_server: listen()
}
```

Now, you can adapt the broker for all your incoming use cases 🎉.
Remark: To run the code, use this command: `golo golo --classpath libs/*.jar --files mqtt.module.golo main.golo`

{ MODIFY GOLO }

Another specificity of Golo is that you can, with great ease, develop the language itself and add features and keywords and then recompile it to have an 'improved' version of Golo tailored for you (it's also an excellent way to contribute).
From the start, Julien Ponge wanted Golo to be a playground for his students to teach them how to modify programming languages. The two simplest ways to do this, are the following:

{ THE FIRST, WITH JAVA }

You just need to add a static method to the final class Predefined:

```
public static String panda() {
  return "🐼";
}
```

You can read the source code of the Predefined class here [10]. After the compilation of the Golo project, you'll get a new keyword in Golo:

```
function main = |args| {
  println("this is a panda: " + panda())
}
```

And of course you can develop Golo in Golo.

The second, with Golo

Remember the example of the "functional map":

```
augment java.util.Map {
  @option
  function getOptional = |this, key| -> this:
get(key)
}
```

If you add this augmentation to the 'standard-augmentations.golo' module of the Golo project, the getOptional method of java.util.Map will become a natural feature of the Golo language. You will find the source code of the 'standard-augmentations.golo' module here [11]. You can of course, write a lot of more complicated things and even play with the AST tree.
The possibilities of Golo are numerous, and it is not possible to list them all, but already, this article gives you a small glimpse of why gololang is my favourite language to write small applications.
In case this article made you curious, you can find some tutorials to start with Golo in the Github repository of the language [12]. <

{ REFERENCES }

1 https://twitter.com/jponge
2 https://twitter.com/yannick_loiseau
3 https://twitter.com/k33g_org
4 https://github.com/eclipse/golo-lang
5 https://www.eclipse.org/downloads/download.php?file=/golo/golo-3.3.0.zip
6 https://gitpod.io/#https://gitlab.com/golo-lang/golo-in-action
7 https://golo-lang.org/documentation/3.3.0/index.html#_running_em_hello_world_em
8 https://golo-lang.org/documentation/3.3.0/golodoc/gololang/Errors.html
9 https://gitlab.com/golo-lang/golo-in-action/-/tree/master/04-augment-java
10 https://github.com/eclipse/golo-lang/blob/master/src/main/java/gololang/Predefined.java
11 https://github.com/eclipse/golo-lang/blob/master/src/main/golo/standard-augmentations.golo
12 https://github.com/eclipse/golo-lang/tree/master/tutorials

Wat is er nieuw in MAVEN 4

Kortgeleden heeft de Maven community besloten om toe te gaan werken naar een Maven 4 release. De eerste vraag die bij je opkomt zal waarschijnlijk zijn: wat kunnen we verwachten van deze nieuwe major release van Maven? Helaas kunnen we niet alles behandelen, maar in dit artikel lichten wij een aantal nieuwe features toe waar wij erg enthousiast over zijn.

Martin Kanters *is senior software engineer en Java community lead bij Info Support en actief als committer bij Apache Maven.*

Maarten Mulders *is architect, senior software engineer en trainer bij Info Support; daarnaast spreekt hij op conferenties en is hij als committer actief bij Apache Maven.*

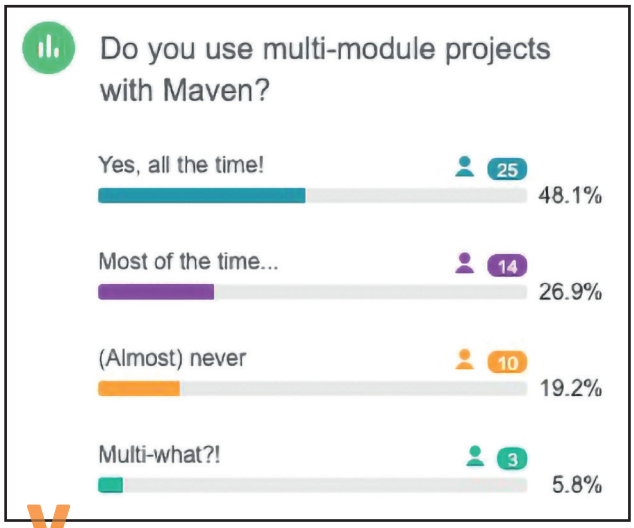
{ VERBETERINGEN AAN DE REACTOR }

De features die waarschijnlijk het meeste effect zullen hebben bij het dagelijks gebruik van Maven zijn enkele verbeteringen aan de Reactor. Niet iedereen zal zich realiseren wat de Reactor is, maar het is een erg belangrijk onderdeel van Maven. Wanneer iemand Maven uitvoert is het de taak van de Reactor om de projectstructuur te verwerken, de modules in de juiste volgorde te zetten en uiteindelijk alles te bouwen. De Reactor maakt het mogelijk om met multimodule-projecten te werken.

Deze verbeteringen zijn dus vooral interessant voor gebruikers van multimodule-projecten. Tijdens onze presentatie op J-Fall 2020 hebben we een poll gehouden om te onderzoeken in hoeverre hiervan gebruik wordt gemaakt. De resultaten daarvan zie je op de volgende pagina. In de volgende paragrafen maken we gebruik van een fictief Maven-project dat uit drie modules bestaat: common, client en app. De afhankelijkheid tussen de modules is hiernaast afgebeeld.

{ GEBRUIK VAN EERDER GEBOUWDE ARTIFACTS BIJ --RESUME-FROM }

Het eerste punt dat we bespreken is een bugfix. Eerder was het zo dat als je Maven-build faalde op de client-module, je een hint te zien kreeg dat je de build verder kon laten gaan vanaf de gefaalde module: `mvn <goal> --resume-from :client`. Dit werkte echter alleen als je van tevoren common in je lokale



Ongeveer 75% van de Maven-gebruikers maakt gebruik van multimodule-projecten.

repository had geïnstalleerd, aangezien client een afhankelijkheid op common heeft. De bugfix zorgt ervoor dat hij de gebouwde class files of artifacts in de target folder kan vinden en gebruiken als afhankelijkheid van client. Precies zoals je verwacht wanneer je de originele build wilt laten verdergaan waar die gebleven is. Meer informatie vind je in het JIRA-ticket: MNG-4660 [1].

{ COMBINATIE VAN --ALSO-MAKE EN --RESUME-FROM }

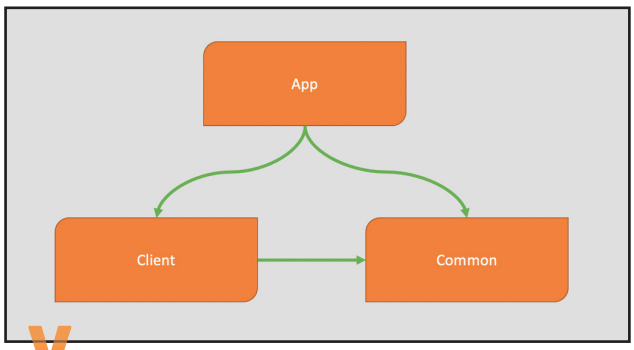
Bij het vorige punt zou je kunnen denken: waarom zou je niet de `--also-make`-vlag toevoegen in combinatie met `--resume-from`? Dit zou automatisch de afhankelijkheden moeten bepalen en opnieuw bouwen. Het antwoord is: omdat het door een andere bug niet werkt in Maven 3. In Maven 4 is deze bug ook gefixt. Meer informatie vind je in het JIRA-ticket: MNG-6863 [2].

{ HERVAT DE BUILD AUTOMATISCH VANAF DE LAATST GEFAALDE MODULE }

Natuurlijk is `--resume-from` een krachtige feature, maar helaas moet je altijd nog zelf de hint lezen onderaan de gefaalde build en vervolgens overnemen in je volgende Maven-aanroep. Dit is meer moeite dan nodig zou moeten zijn. Maven 4 onthoudt zelf welke module als laatste gefaald is, zodat je met de nieuwe `--resume`-vlag met iets meer gemak je build hervat. Meer informatie vind je in het JIRA-ticket: MNG-5760 [3].

{ SUBMODULE BUILDS BEWUST GEMAAKT VAN VOLLEDIGE MULTIMODULE-PROJECT }

Stel je voor dat app een webapplicatie is die je met `mvn jetty-:run` kunt starten. Als je dit uitvoert op de root van het multimodule-project gaat het fout. Maven zal namelijk `jetty:run` ook op client en common uitvoeren. Aangezien deze modules geen webapplicaties zijn, zal het falen.



Fictief multimodule-project met onderlinge afhankelijkheden

Om dit op te lossen kan je Maven ook uitvoeren in de subdirectory van de app-module. Helaas zal Maven 3 zich niet bewust zijn van de andere modules in het multimodule-project. Hij zal de afhankelijkheden dus alleen kunnen vinden en gebruiken wanneer deze in een lokale of remote repository zijn geïnstalleerd. In Maven 4 is dit opgelost door Maven altijd bewust te laten zijn van het volledige multimodule-project, zelfs als je maar één module van het project wilt bouwen. Weer een reden minder om een onnodige mvn install te doen!

Op twee manieren kan `jetty:run` uitgevoerd worden op alleen de app-module:

1. Navigeer naar de directory van de app directory en voer vanaf daar het Maven goal uit: `cd app; mvn jetty:run`.
2. Met de '-f'-vlag kan je refereren naar een POM-bestand dat gebruikt moet worden voor de build. In dit geval vanaf root zou dat er als volgt uitzien: `mvn -f app/pom.xml jetty:run`. Meer informatie vind je in het JIRA-ticket: MNG-6118 [4].

{ STANDAARD RECURSIEVE BUILD BIJ AGGREGATOR-POM'S }

Deze verandering is vooral interessant voor multimodule-projecten die vele levels diep gaan. Neem bijvoorbeeld Quarkus; voor elke extensie zijn er in ieder geval altijd twee submodules, namelijk deployment en runtime.

Eerder was het zo dat als je bijvoorbeeld de gRPC-module inclusief alle vijf de childmodules wilde bouwen, je de lijst van alle modules moest opgeven: `mvn -pl extensions/grpc,extensions/grpc/codegen,extensions/grpc/deployment` et cetera ...

Maven 4 brengt de verandering dat bij de selectie van een aggregator-POM (extensions/grpc in dit geval) Maven standaard de child modules ook zal selecteren. Het bouwen van alleen de aggregator-POM zal meestal niet nuttig zijn. Mocht je dit toch willen, kan je met de vlag `--non-recursive` het oude gedrag terugkrijgen. Meer informatie vind je in het JIRA-ticket: MNG-6981 [5].

{ DETECTEREN VAN EEN WARNING- OF ERROR-VRIJE BUILD }

Er zijn natuurlijk ook wijzigingen gedaan los van de Reactor en multimodule-projecten!

A warning is an error in the making.
(unknown author)

Veel tools geven via warnings aan dat er iets niet helemaal goed gaat en dat dit mogelijk kan leiden tot ongewenst gedrag of errors in de toekomst. Daarom is het een goed idee om bewust te zijn van de warnings en ze te limiteren.
Met de nieuwe vlag `--fail-on-severity <severity>` kun je de build laten falen wanneer er minimaal één WARN- of ERROR- logregel voorkomt in Maven of in één van de plug-ins. Meer informatie vind je in het JIRA-ticket: MNG-6065 [6].

{ CONSISTENTE TIMESTAMP IN MULTIMODULE-PROJECTEN }

Maven genereert tijdens een build een timestamp die uiteindelijk terecht komt in het gebouwde artifact. Er was helaas een bug die ervoor zorgde dat bij elke module de timestamp opnieuw gegenereerd werd, waardoor deze dus per module verschilde.
Dit klopt niet en kan bijvoorbeeld voor in CI-situaties een erg irritante bug zijn.
Nu zal de 'maven.build.timestamp'-property altijd gelijk zijn voor alle modules. Meer informatie vind je in het JIRA-ticket: MNG-6754 [7].

{ EN ER IS MEER! }

Maven 4 brengt nog veel meer moois, maar helaas gaat dat niet passen in dit artikel. In het kort een overzicht van enkele andere features:

- De standaardversie van een aantal plug-ins is geüpdatet. Dit geldt voor de POM, JAR, EJB, WAR, EAR, RAR en Maven Plugin packaging types.
- Wanneer de checksum van een gedownloade file niet correct is zal de build nu breken. Eerder was dit een warning. Snap je nu ons enthousiasme over `--fail-on-severity` ;-)?
- Bepaalde huidige warnings zullen de build nu breken. Opnieuw iets dat `--fail-on-severity` kan voorkomen.
- De POM's voor het bouwen van het project en degene die gedistribueerd zal worden zullen niet meer per sé gelijk zijn. Het nut en de werking hiervan is uitgelegd in de column van Robert Scholte in het Java Magazine (eerste editie 2020) en in een videochat [9].
- De Maven Wrapper (vergelijkbaar met de Gradle Wrapper) was jarenlang apart onderhouden functionaliteit, maar maakt sinds Maven 4 deel uit van Maven zelf.
- Automatische parent/dependency-versionering.

Graag verwijzen we je door naar de release notes [10] voor alle gedetailleerde informatie.

{ BLEEDING EDGE TESTEN }

Naast het doorontwikkelen van Maven is het ook belangrijk om de laatste (onuitgebrachte) versies te testen. Het is immers niet handig als de nieuwste versie vol bugs uitkomt. Dus, als jij de nieuwste versie wilt uitproberen juichen we dat toe! Bijkomend voordeel is dat je meteen kunt profiteren van de nieuwste features en bugfixes. Als je macOS of Linux gebruikt, kun je gebruikmaken van Maartens 'Homebrew tap' [12] om gemakkelijk de nieuwste versie te installeren. Dit werkt als volgt:

```
brew tap mthmulders/maven-snapshot
brew install maven-snapshot
```

Mocht je weer terug willen naar de laatst uitgebrachte Maven-versie, kan dat met de volgende commando's:

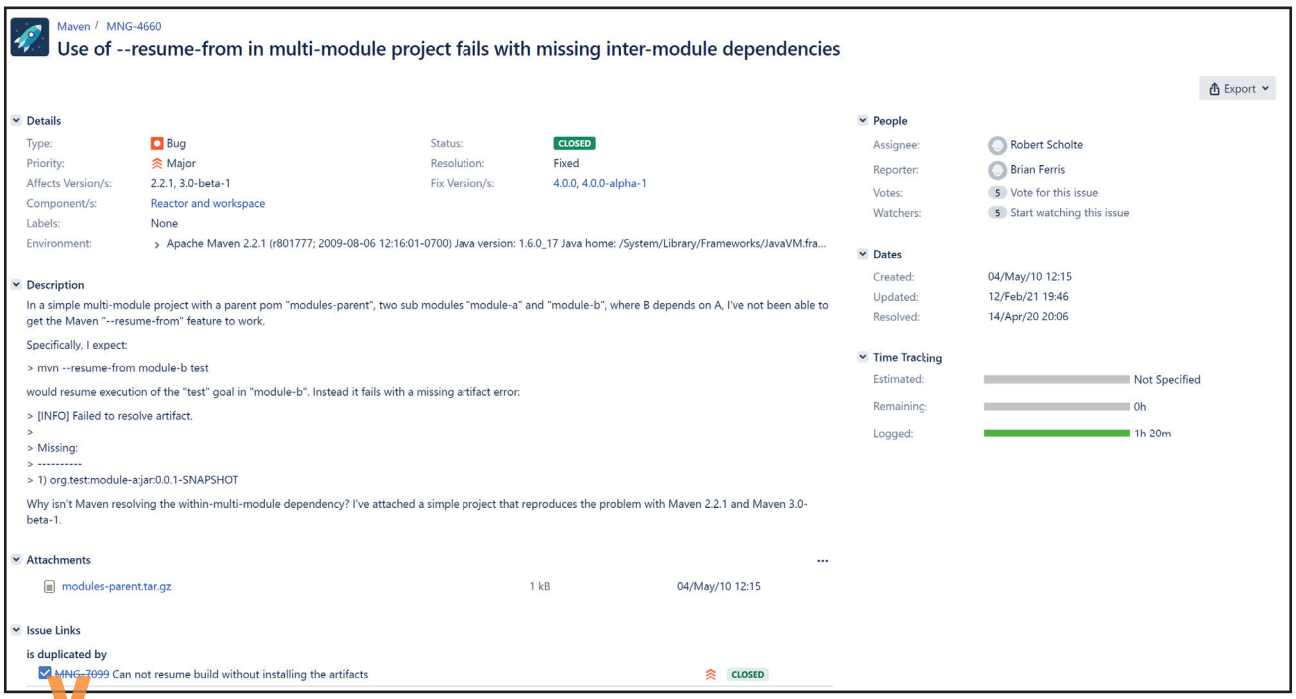
```
brew unlink maven && brew link maven-snapshot
brew unlink maven-snapshot && brew link maven
```

HELP ZELF MEE!

Maven is een enorm belangrijk project voor vrijwel alle Javanen wereldwijd. Gek genoeg zijn er maar enkele tientallen actieve contributors. Mocht je zelf willen meehelpen, kan je dat in ieder geval op de volgende twee manieren doen.

Up-for-grabs oppakken
Zoals meerdere opensource-projecten heeft Maven een grote lijst van issues die opgepakt kunnen worden door ontwikkelaars. Echter, niet alle issues zijn gemakkelijk op te pakken voor ontwikkelaars die (nog) onbekend zijn met de codebase. Speciaal hiervoor is er een selectie gemaakt van issues die dat wel zijn. Deze issues hebben het label 'up-for-grabs' en zijn te vinden op [11].

Automatische builds op GitHub Actions
Of je nu een ervaren Maven-ontwikkelaar bent of een beginner, het eerste wat je wilt weten bij een wijziging is of alles nog naar behoren werkt. Normaliter was het proces dat je als Maven-committer een build moest starten op de Jenkins-servers van de Apache Software Foundation met jouw wijzigingen. Deze build test de wijzingen dan op Windows- en Linux-machines met verschillende JDK-versies. Voordat wij zelf committer werden merkten we dat dat helaas soms lang kan duren.
We hebben dit proces vereenvoudigd met GitHub Actions. Als je nu een pull request opent start automatisch een build op Windows, Linux en macOS tegen verschillende JDKs. Veel snellere feedback dus, juist voor die eerste bijdragen!
Doordat macOS nu ook wordt meegenomen hebben we zelfs een falende test gevonden [8] die alleen op macOS voorkwam. Je kunt de GitHub builds in actie zien op https://github.com/apache/maven/actions.



Een voorbeeld van een Jira ticket.

Op Windows kun je hetzelfde doen door middel van Martins 'Chocolatey'-package. Voer het volgende uit om de maven-snapshot te installeren:

```
choco install maven-snapshot
refreshenv
```

Gebruik daarna het volgende om de origineel geïnstalleerde Maven-versie weer te activeren:

```
choco uninstall maven-snapshot
refreshenv
```

Let wel op: hoewel er getest wordt tegen duizenden integratietesten op verschillende platformen, kunnen er absoluut nog fouten in zitten. We raden het dus niet aan om dit in te zetten in bedrijfskritieke situaties. We moedigen het aan om het in te zetten op je development-machine. Mocht je issues vinden, laat het dan alsjeblieft weten op JIRA [13].

{ SAMENVATTEND }

Wij zijn erg enthousiast over al het moois dat Maven 4 brengt! Wij hebben met veel plezier meegewerkt aan een aantal van deze mooie features in de tijd van onze werkgever Info Support. We zijn erg blij dat we deze tijd krijgen, want we vinden het belangrijk om de tool die zo belangrijk is voor ons werk nóg beter te kunnen maken. Volg de JIRA-links om meer informatie en de relevante GitHub pull requests te vinden.

{ VERWIJZINGEN }

1 MNG-4660: https://issues.apache.org/jira/browse/MNG-4660
2 MNG-6863: https://issues.apache.org/jira/browse/MNG-6863
3 MNG-5760: https://issues.apache.org/jira/browse/MNG-5760
4 MNG-6118: https://issues.apache.org/jira/browse/MNG-6118
5 MNG-6981: https://issues.apache.org/jira/browse/MNG-6981
6 MNG-6065: https://issues.apache.org/jira/browse/MNG-6065
7 MNG-6754: https://issues.apache.org/jira/browse/MNG-6754
8 MNG-6951: https://issues.apache.org/jira/browse/MNG-6951
9 Videochat Build/consumer POM: https://youtu.be/KDAmlNKZJto
10 Maven 4 release notes: https://issues.apache.org/jira/secure/ReleaseNote.jspa?projectId=12316922&version=12346477
11 Up for grab issues: https://s.apache.org/for-the-grabs_maven
12 Maven Snapshot tap voor Homebrew: https://github.com/mthmulders/homebrew-maven-snapshot/
13 Maven issue tracker: https://issues.apache.org/jira/

SIMPEL
maar effectief

De doorsnee backend-applicatie ontkomt er niet aan om validatie toe te passen op business-objecten voordat deze mogen worden gepersisteerd in de database of verder mogen worden verwerkt in de geldende business-logica.

Erwin Tinkóczy *is Java consultant en software architect bij Ordina*

Vaak begint het dan ook dat de applicatie een EJB krijgt die verantwoordelijk is om het object te controleren op eigenschappen en logica. Gevaar hierbij is dat tijdens de ontwikkeling en doorontwikkeling al snel een groeiend woud aan voorwaarden ontstaat naarmate de functionele requirements van de applicatie worden uitgebouwd. Ook in ons huidige project liepen we tegen dit probleem aan, het aantal validatiestappen groeide en groeide, en ook de verschillende typen objecten waarvoor validatie nodig is nam gestaag toe.

{ EEN SIMPELE CONTROLE }

Nu zou je kunnen kiezen voor een rules-framework als bijvoorbeeld Drools, of Easy Rules, maar soms is dat overkill en wil je alleen maar je business-rules op een duidelijke manier onderhoudbaar in je applicatie beschikbaar hebben. Daarom de vraag: "Wat wil je dat er gebeurt bij een validatieregel?"
Het eenvoudige antwoord hierop is: "Een simpele controle dat een bepaalde waarde/eigenschap van mijn onderhanden object voldoet aan de gestelde business-eisen."
En als vervolgvraag: "Hoe kan ik eenvoudig bijhouden welke validatieregels er gelden voor een bepaald business-entiteit?". Dit kan door alle geldende regels te bundelen en dan uit te voeren. En als niet aan een validatieregel wordt voldaan, dan wil ik dat kunnen onderkennen middels een exceptie zodat de overige validatieregels niet meer worden uitgevoerd. Een generieke validatieregel is dus eigenlijk:

```
public interface ValidationRule<O extends Object, E
extends Exception> {

    public void validate(O o) throws E;
}
```

Middels de validate-methode wordt het business-object 'o' gevalideerd. Als deze niet valide is dan wordt exceptie 'E' gegooid.

En een generieke rule-engine kan worden gedefinieerd als verantwoordelijke voor het bijhouden van de geldende regels voor een business-object en voor het uitvoeren van de regels:

```
public interface RuleEngine<O, E extends Exception>
{

    /**
     * Registers a new {@link ValidationRule} in
the RuleEngine to apply for the entity defined in
the Rule
     *
     * @param rule
     */
    void register(ValidationRule<O, E> rule);

    /**
     * Execute all {@link ValidationRule}s that are
registered in the RuleEngine
     *
     * @param object
     * @throws E When a checked exception occurs
while executing the business rules
     */
    void fire(O object) throws E;
}
```

Met de RuleEngine wordt het mogelijk om Validation-rules van een bepaald object type te registeren. Ook kun je een eigen type Exception definiëren waarvan je verwacht dat die zal worden gegooid indien de validatie-regel niet slaagt. Het uitvoeren van de geregistreerde validatieregels wordt getriggered door de fire-methode, met als parameter het business-object dat aan de validatieregels onderworpen moet worden.
Deze rule-engine is ook uit te breiden door het gebruik van een generiek Rule-type in plaats van de ValdationRule-type. De ValidationRuleEngine implementatie zou er vervolgens zo uit kunnen zien:

```
public class ValidationRuleEngine<O, E extends
Exception> implements RuleEngine<O, E> {

    private List<ValidationRule<O, E>>
listOfBusinessRules = new ArrayList<>();

    @Override
    public void register(final ValidationRule<O, E>
rule) {
        listOfBusinessRules.add(rule);
    }

    @Override
    public void fire(final O object) throws E {
        for (ValidationRule<O, E> rule :
listOfBusinessRules) {
            rule.validate(object);
        }
    }
}
```

Dat is niet veel code voor iets dat bestempeld is als 'engine'. En dat hoeft ook niet. De echte uitdaging bij het uitvoeren van de validatie zit hem in de validatieregels zelf. Deze engine zorgt ervoor dat alle validatiecode netjes in kleine rule-classes kan worden geschreven. We kunnen dus bijvoorbeeld de eerste implementatie van een ValidationRule als volgt definiëren, als we willen controleren of een auto die volgens de specificaties vier wielen hoort te hebben die ook daadwerkelijk heeft:

```
public class CarHasFourWheelsRule implements
ValidationRule<Car, InvalidCarException> {

    @Override
    public void validate(final Car car) throws
InvalidCarException {
        if (car.getWheels().stream().count() != 4) {
            throw new InvalidCarException("Your Car
should have 4 wheels according its specifications");
        }
    }
}
```

De RuleEngine die we nodig hebben om een Car-object te valideren is van het type Car en zal een InvalidCarException gooien als de gedefinieerde ValidationRule een fout constateert in ons Car-object.

```
public void validateMyCar(final Car myCar) throws
InvalidCarException {

    RuleEngine<Car, InvalidCarException> ruleEngine
= new ValidationRuleEngine<>();

    ruleEngine.register(new CarHasFourWheelsRule());
    ruleEngine.register(new CarHasEngineRule());
    ruleEngine.register(new CarHasHeadLightsRule());
    ruleEngine.register(new
CarHasBreakLightsRule());
    ruleEngine.register(...);

    ruleEngine.fire(myCar);

}
```

Als je meerdere validaties toevoegt aan je RuleEngine dan blijft het geheel nog erg goed leesbaar. Als er wijzigingen in een bepaalde ValidationRule-implementatie nodig zijn, omdat bijvoorbeeld ook twin-engine Car-objecten als valide gelden dan is dat een eenvoudige exercitie. <

GEEN JSR 303

Het framework dat hier beschreven wordt, is bewust geen implementatie van JSR 303, omdat in de implementatie van de Rules ook gebruik moet kunnen worden gemaakt van andere stateless beans die stukken business-logic uitvoeren. Hierbij worden bijvoorbeeld stamdata opgehaald uit de database en bepaald welke stamdata geldig zijn, waarop deze informatie in de Rule wordt toegepast.

JAVA 16

Oracle heeft sinds Java SE 10 een halfjaarlijks release-schema geïnstalleerd, in tegenstelling tot de globaal driejaarlijkse releases daarvoor. De deadline voor nieuwe features in Java is fixed gemaakt. Dit betekent dat het aantal JEP's (JDK Enhancement Proposals) dat geïmplementeerd kan worden per release variabel is geworden. Ook zal eens in de drie jaar een zogenaamde LTS-versie (Long-Term Support) worden gemaakt.

V

Ivo Woltring *is Software Architect en Codesmith bij Ordina JTech en houdt zich graag bezig met nieuwe ontwikkelingen in de softwarewereld.*

Java SE 11 is zo'n LTS-versie en zal door Oracle nog tot september 2023 ondersteund worden. Dit betekent dat Java 17 ook weer een LTS wordt. Die gaat dit jaar rond september gereleased worden! De versies die tussen LTS-versies uitkomen krijgen alleen support tot een nieuwe versie uitkomt, dus zo ook Java SE 16.
Java SE 16 is in maart 2021 uitgekomen. Dit artikel beschrijft welke nieuwe features in Java SE 16 geïmplementeerd zijn. Tijdens het schrijven van dit artikel was deze laatste releasedatum nog niet bereikt, dus er kunnen dus nog (kleine) scope-wijzigingen plaats hebben gevonden. In Java 16 staan 17 features (JEP's) gepland.
Om met wat features van Java SE 16 te kunnen spelen, zonder de early access echt te hoeven installeren, is alle code in dit artikel uitgeprobeerd binnen een Docker-container met OpenJDK 16 (listing 1).
Dit artikel is in twee hoofdsecties onderverdeeld. Het eerste deel zal gaan over nieuwe standaard-features. Het tweede deel zal ingaan op preview- en incubator-features. In versie 16 worden geen features uitgefaseerd. Bij elke feature zal het JEP-nummer vermeld worden.

{ NIEUWE STANDAARD-FEATURES }

347: Enable C++14 Language Features

C++14 features zullen geactiveerd worden in de JDK-sourcecode-builds. Dit betekent dat de Java-sourcecode gecompileerd zal worden met de C++14 (-std=c++14) opties, geactiveerd voor alle platformen (Windows, Linux, macOS en AIX). De JEP geeft ook richtlijnen over welke van de C++14-features gebruikt mogen/kunnen worden in de HotSpot-code. Tot JDK 15 zijn de features die gebruik maakten van C++-code gelimiteerd geweest tot de C++98/03-standaard. In Java SE 11 is de code geüpdatet zodat hij nieuwe versies aankon, maar er werd nog geen gebruik van gemaakt.

L1
```
$ docker run -it --rm \
      -v $(pwd)/src/main/java:/src \
      openjdk:16 /bin/bash
$ cd /src
```

357: Migrate from Mercurial to Git en 369: Migrate to GitHub

De OpenJDK Community sourcecode-repositories zijn gemigreerd van Mercurial naar Git. De drie hoofdredenen voor het migreren naar Git zijn:
1. De grootte van de versiebeheer metadata
2. Beschikbare tools
3. Beschikbare hosting

Initiële tests hebben aangetoond dat de grootte van de JDK-repository in Git ongeveer 300 MB is en in hg 1,2 GB. Daarnaast wordt Git door bijna alle teksteditors en IDE's geweldig ondersteund. Gekozen is voor het hosten op GitHub: https://github.com/openjdk [3].

376: ZGC: Concurrent Thread-Stack Processing

De Z Garbage Collector is ontworpen om GC-pauzes en schaalbaarheidsproblemen in HotSpot weg te halen. In Java SE 15 is het een standaard product-feature geworden. Java SE 16 worden een paar optimalisaties doorgevoerd om de GC nog sneller te maken.

380: Unix-Domain Socket Channels

De SocketChannel en ServerSocketChannel API's leveren 'blocking'- en 'multiplexed non-blocking'-toegang voor tcp/ip-sockets. In Java SE 16 worden deze classes extended zodat ook Unix domain (AF_UNIX) sockets en server sockets ondersteund worden. Unix-domain sockets wordt gebruikt voor communicatie tussen processen (Inter-process communication - IPC) op dezelfde host.

L2

```
public class JEP390 {
    public static void main(String[] args) {
        Double d = 20.0;
        synchronized (d) { /* do stuff */ }
// javac warning & HotSpot warning
        Object o = d;
        synchronized (o) { /* do stuff */ }
// HotSpot warning
    }
}
$ java JEP390.java
JEP390.java:4: warning: [synchronization] at-
tempt to synchronize on an instance of a va-
lue-based class
        synchronized (d) { /* do stuff */ }
// javac warning & HotSpot warning
        ^
1 warning
```

L3

```
public class JEP394 {
    public void print(Object o) {
        if (o instanceof String bs &&
            bs.length() > 5){
            System.out.println(bs);
        } else {
            System.out.println("Shorter");
        }
    }
  public static void main(String[] args) {
    JEP394 jep394 = new JEP394();
    var site = "IvoNet.nl";
    jep394.print(site);
  }
}
```

L4

```
public class JEP395 {
    static record Point(int x, int y) {};
    public static void main(String[] args) {
        final Point p1 = new Point(1, 2);
        final Point p2 = new Point(1, 2);
        if (p1.equals(p2)) {
            System.out.println(p1);
        }
    }
}
$ java JEP395.java
Point[x=1, y=2]
```

386: Alpine Linux Port

In deze JEP wordt de JDK geport naar Alpine Linux en andere Linux-distributies die musl gebruiken als hun belangrijkste C-Library. Dit gebeurt voor zowel de x64- als de AArch64-architecturen. Musl (uitgesproken in het Engels als "muscle") is een implementatie van de standard-library zoals beschreven is in de standaarden ISO C en POSIX.

388: Windows/AArch64 Port

Ports de JDK naar Windows/AArch64. Met het op de markt komen van nieuwe hardware voor eindgebruikers en AArch64 (ARM64)-servers is de vraag naar deze port erg toegenomen.

387: Elastic Metaspace

De metaspace van de HotSpot, waar class-metadata worden gemanaged, is redelijk berucht om het hoge geheugengebruik. Voor de meeste programma's is dit geen probleem, maar in sommige gevallen kon het niet (tijdig) teruggeven van ongebruikt geheugen leiden tot excessief geheugengebruik. Deze JEP zorgt ervoor dat ongebruikte metaspace tijdig weer teruggegeven wordt aan het Operating Systeem (OS).

390: Warnings for Value-Based Classes

Het Project Valhalla [4] is druk bezig met een grote aanvulling op het Java-programmeermodel in de vorm van primitive classes. Het is de bedoeling dat dit soort classes zich vergelijkbaar gaan gedragen als echte primitives. Deze classes worden ook wel Value-based classes genoemd. Denk hierbij aan classes als 'java.lang.Integer', 'java.lang.Double', etc. Alle classes die in aanmerking komen voor deze aanpassing zijn met `@ValueBased` geannoteerd.

De primitieve wrapper-classes in de JDK die geannoteerd zijn met `@jdk.internal.ValueBased` zullen voortaan een waarschuwing genereren als je er op probeert te synchroniseren. Deze zullen namelijk in de toekomst niet meer werken. Zie listing 2.

392: Packaging Tool

De Packaging Tool is een command-line tool (jpackage) om platformspecifieke packages te maken van Java-applicaties. Dit betekent deb- of rpm-bestanden op Linux, pkg- of dmg-bestanden op macOS en msi- of exe-bestanden op Windows. In het artikel over Java 14 is hier een uitgebreid codevoorbeeld van gegeven [2].

394: Pattern Matching for instanceof

Na twee incubatorfases is deze feature nu onderdeel geworden van de taal. De type-test kan gevolgd worden door een binding-variabele die daarna meteen gebruikt kan worden. In het voorbeeld van listing 3 is dat de bs-variabele. De bs-variabele wordt in de if geïntroduceerd en in hetzelfde statement al gebruikt (`bs.length()`) en in de verdere methode.

L5

```
#include <stdio.h>
void helloworld() {
    printf("Hello Java Magazine readers!\n");
}
$ microdnf install gcc
$ gcc -c -fpic helloworld.c
$ gcc -shared -o helloworld.so helloworld.o
```

395: Records

Records bieden een compacte syntaxis voor het declareren van klassen die houders zijn voor onveranderlijke ('immutable') gegevens. Ook deze feature is nu standaard na twee incubatiefases. Zie listing 4.

396: Strongly Encapsulate JDK Internals by Default

Deze JEP zorgt ervoor dat alle JDK-interne classes nu op de '--illegal-access'-optie standaard de deny (niet toegestaan) in plaats van de permit (toegestaan) krijgen. Alleen kritische API's zoals de 'sun.misc.Unsafe' zullen nog toegankelijk blijven.

{ PREVIEW EN INCUBATOR FEATURES }

338: Vector API (Incubator)

Dit is de eerste iteratie van de 'jdk.incubator.vector'-module. Deze module stelt je in staat om vectorberekeningen uit te drukken die runtime gecompileerd kunnen worden naar optimale vector-hardware-instructies voor ondersteunde CPU-architecturen en daardoor dus de optimale performance voor dit soort berekeningen haalt. Zie referentie [2] voor voorbeeldcode.

389: Foreign Linker API (Incubator)

Deze API levert een volledig Java-manier om native code aan te roepen. Het zal JNI gaan vervangen. Zie listing 5 voor een voorbeeld.

Nu we er een shared library van gemaakt hebben kunnen we het gaan aanroepen vanuit Java. In die code zoeken we de library en dan de method. Deze wordt aan een eigen method handle gebonden door een FunctionDescriptor en een MethodType aan te geven waarna de method aangeroepen kan worden. Omdat het nog een Incubator-feature is moet de module expliciet meegegeven worden en aangezet (`--add-modules jdk.incubator.foreign -Dforeign.restricted=permit`) Zie listing 6.

393: Foreign-Memory Access API (Third Incubator)

Deze API is al in Java SE 14 aangeboden als incubator-module en is in dat artikel uitgebreid behandeld geweest [2]. In Java SE 15 zijn er nog aanpassingen gedaan op basis van bevindingen en ook in Java SE 16 worden verfijningen aangebracht, waardoor een derde Incubatie-fase ingegaan wordt.

L6

```
import jdk.incubator.foreign.CLinker;
import jdk.incubator.foreign.FunctionDescriptor;
import jdk.incubator.foreign.LibraryLookup;
import java.lang.invoke.MethodType;
import java.nio.file.Path;
public class JEP389 {
  public static void main(String[] args)  {
    var lib = LibraryLookup
      .ofPath(Path.of("/src/helloworld.so"));
    var sym = lib.lookup("helloworld").get();
    var fd = FunctionDescriptor.ofVoid();
    var mt = MethodType.methodType(Void.TYPE);
    var mh = CLinker.getInstance()
      .downcallHandle(sym.address(), mt, fd);
    mh.invokeExact();
  }
}
$ java --add-modules jdk.incubator.foreign
-Dforeign.restricted=permit JEP389.java
WARNING: Using incubator modules: jdk.incubator.
foreign
warning: using incubating module(s): jdk.incuba-
tor.foreign
1 warning
Hello Java Magazine readers!
```

397: Sealed Classes (Second Preview)

Deze JEP biedt Java een manier om implementaties en overerving van classes, interfaces en records te beperken tot wat expliciet toegestaan is. In Java SE 15 is deze ter preview aangeboden en in het bijbehorende artikel behandeld [2]. Deze preview bouwt hierop voort.

{ CONCLUSIE }

In deze versie is niks verwijderd, maar wel een heleboel toegevoegd. Voor het dagelijkse werk van de meeste Software Engineers zullen Records en de Pattern matching-JEPS waarschijnlijk het meest geanticipeerd zijn. De volgende versie (17) wordt weer een LTS-versie. De kinderziektes van Java SE 9 zijn lang voorbij en sinds die versie zijn vele mooie dingen toegevoegd aan de taal. De meeste belangrijke frameworks zijn al lang over naar tenminste Java SE 11 LTS en voor bedrijven die nu nog Java SE 1.8 (8) gebruiken is er zo langzamerhand geen fatsoenlijk excuus meer om die versie te handhaven. <

REFERENTIES

1 http://openjdk.java.net/projects/jdk/16/
2 http://ivo2u.nl/oz
3 https://github.com/openjdk
4 https://openjdk.java.net/projects/valhalla/

HTTP/1, 2, … 3!

Zoals de titel doet vermoeden gaat dit artikel over versie drie (referentie 1) van het ongeveer dertig jaar oude http-protocol. In dit artikel wil ik je graag op de hoogte brengen van het bestaan van dit protocol. Dat wil zeggen, de draft daarvan, omdat er op het moment van schrijven van dit artikel nog geen details over een officiële release bekend zijn.

Edwin Derks *is Principal Software Architect en Codesmith bij Ordina en heeft een passie voor alles wat met java en cloud-driven development te maken heeft.*

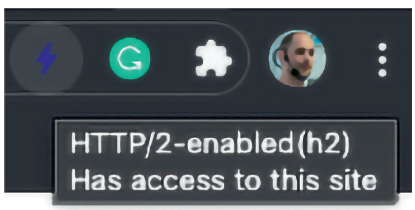
Echter, het daadwerkelijk gebruiken van dit protocol is dichterbij dan je misschien vermoedt. Het kan namelijk zo zijn dat je het op de achtergrond toch al hebt gebruikt in bepaalde toepassingen. Hoe dat zit, en wat dit protocol voor jou kan betekenen in de toekomst als internetgebruiker en developer, lees je hieronder.

{ VAN HTTP/1.0 NAAR HTTP/1.1 }

Het http-protocol is bijna niet weg te denken als communicatiemiddel van browsers om webpagina's mee op te halen. Dergelijke webpagina's zijn vaak samengesteld uit vele resources zoals html, javascript en css-bestanden. Deze kunnen uiteindelijk vele Mb's aan dataverkeer verwezenlijken. Met de internetsnelheden en -bundels van tegenwoordig is dit meestal niet meer merkbaar, en niet direct een probleem. Maar onder de motorkap van een browser is dit niet altijd efficiënt qua geheugen- en throughput-optimalisatie, wat in bepaalde situaties niet wenselijk is.
Dit geldt zeker in het geval wanneer het originele http/1.0 protocol gebruikt wordt. Http definiëert de API die onder andere de GET en POST methoden bevat. Onderliggend wordt het tcp-protocol gebruikt om daadwerkelijk de data over de lijn te transporteren. Op basis van het aantal op te halen resources, dient ook hetzelfde aantal tcp-connecties geopend te worden. De evolutie van http/1.0 naar http/1.1 gooit daar al een efficiëntieslag overheen door middel van de "keep alive"-feature. Hierdoor wordt het aantal te initiëren connecties beperkt wanneer mogelijk. Daarmee waren we er echter nog niet, en dus is men doorgegaan met het nog efficiënter maken van het http-protocol.

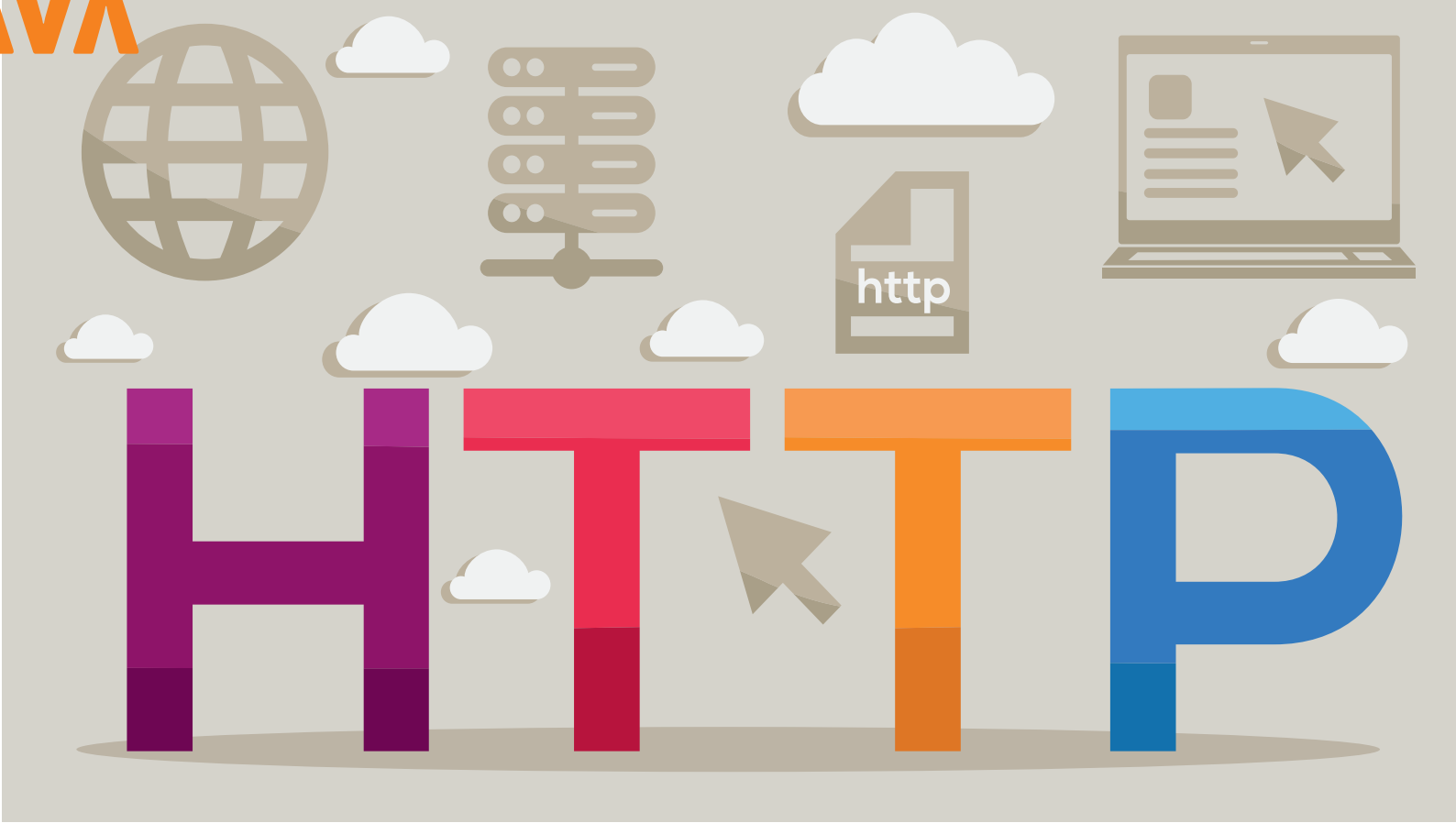
{ VERBETERINGEN MET HTTP/2 }

In 2015 werd het http/2-protocol gereleased, gebaseerd op het experimentele SPDY-protocol van Google (referentie 2). Hoewel de API van het http-protocol niet wijzigde, veranderde er onder de motorkap van het protocol wel heel wat. Dit waren enkele minder significante aanpassingen, zoals efficiëntere headers. Het paradepaardje van http/2 was echter om resources efficiënter op te halen. Door gebruik te maken van multiplexing konden deze voortaan zoveel mogelijk gebundeld worden in één tcp-connectie. Hiermee kunnen deze kostbare connecties efficiënter gebruikt worden, omdat tweerichtingsverkeer van data over een connectie veel minder blokkeert. Hierdoor werden in beperkte mate de blocking-issues opgelost die inherent zijn aan het gebruik van tcp. Zelfs voor de release van http/2 werd er al veel met het SPDY-protocol geëxperimenteerd, in bijvoorbeeld de Chrome-browser. Dit heb je als eindgebruiker waarschijnlijk niet direct gemerkt. Mocht je geïnteresseerd zijn in te zien wanneer Chrome een SPDY- of http/2-connectie gebruikt dan kun je hiervoor een plugin installeren die dit weergeeft (referentie 3). Plaatje 1 geeft op basis daarvan een indruk hoe detectie van een SPDY- of http/2-connectie er uitziet bij het openen van een webpagina.



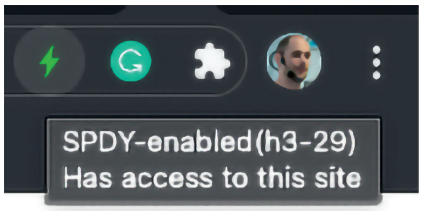
{ HTTP, MAAR DAN "QUIC" }

Zoals gezegd is men bezig met efficiëntieslagen voor het http-protocol, waarbij http/3 voortborduurt op http/2. Ook met de komst van http/3 is de API van het originele http-protocol niet gewijzigd. Wat je misschien in eerste instantie niet verwacht, is dat in http/3 het fundament van het http-protocol compleet vervangen is. In deze versie is namelijk het onderliggende tcp-protocol vervangen door een nieuw protocol dat beter geschikt lijkt te zijn.
De natuur van tcp waarmee in bepaalde gevallen nog steeds efficientieverlies kan optreden, wordt opgelost door toepassing van het



QUIC-protocol (Quick UDP Internet Connections). Dit spreek je dan ook uit als “quick”, wat qua woordspeling een leuke twist aan deze abbreviatie geeft. QUIC past op een slimme manier het gebruik van de multiplexed-connecties van http/2 toe, in combinatie met het onderliggende udp-protocol in plaats van tcp. Dit voorkomt head-of-line-blocking en is efficiënter qua resource-gebruik. Wil je hier alle technische details over weten, neem dan een kijkje op (referentie 4).

Hoewel de http/3-standaard in verschillende browsers al als preview is geïmplementeerd, heb ik op het moment van schrijven helaas nog geen concrete output of bewijs kunnen vinden van calls die hier daadwerkelijk mee uitgevoerd worden.



Wel kun je zien in plaatje 2 dat de eerder genoemde Chrome-plug-in voor het detecteren van SPDY- en http/2-connecties aangeeft dat h3-29 wordt gebruikt. Dit is Googles werkversie van de implementatie van het QUIC-protocol. Opmerkelijk – en ingenieus – wordt QUIC dus momenteel al min of meer ‘on the fly’ gebruikt. Alleen dan in combinatie met http/2, als vervanging van het onderliggende tcp-protocol hiervan. Er is dan ook overwogen om dit concept als http-over-QUIC te definiëren. Deze term werd echter niet als een logische definitie voor de evolutie van het http-protocol gezien, daarom is uiteindelijk gekozen voor de term http/3.

{ CONCLUSIE }

De definitie van http heeft zich intussen wel bewezen als een zeer goed idee, dat dertig jaar na dato nog steeds standhoudt. In de libraries die je gebruikt om te communiceren met andere services over http, zul je in bijna alle gevallen nog connecties opzetten die gebaseerd zijn op het http/1.1-protocol. Daar is helemaal niets mis mee, zeker niet als je simpelweg om een enkele resource vraagt van een andere service. Dit is veelvoorkomend gedrag in bijvoorbeeld een microservicesarchitectuur. Mocht je toch de behoefte aan een http/2-client in java hebben: deze is sinds java 11 beschikbaar in de JDK (referentie 5). Je hebt dus de keuze om de juiste versie van http te gebruiken voor het juiste doel. Tot nu toe lijkt het er op dat er alleen maar voordelen zitten aan de evolutie van http. Mocht iemand toch ooit een nadeel ontdekken, dan ben ik graag de eerste die het hoort. ‹

REFERENTIES

1 ***https://en.wikipedia.org/wiki/http/3***
2 ***https://en.wikipedia.org/wiki/SPDY***
3 ***https://chrome.google.com/webstore/detail/http2-and-spdy-indicator/mpbpobfflnpcgagjijhmgnchggcjblin***
4 ***https://en.wikipedia.org/wiki/QUIC***
5 ***https://openjdk.java.net/groups/net/httpclient/intro.html***

PERSONALISATION AS A SERVICE, A DATA JOURNEY

'Marketing': A dirty word, many would say, and how does that work at a bank? In ABN AMRO's case we use marketing automation to provide our customers with the best experience and the most suitable products and services. We want to optimise your menu based on your most used interactions, deep link you to the self service page in case of need and help you find the best suitable mortgage. But, how do we actually achieve all of that and what is in the pipeline? It all starts with the data at the foundation of it, a smart choice of SaaS solutions and API's to enhance and personalise experiences. In this talk we are not going to talk about customer journeys but take a slightly different perspective and tag you along on the data journey.

Finding data in a big and regulated financial organisation can be difficult, the data sources are innumerable and all of them could be useful to provide better services to our beloved customers. But how can we find the right data and how can we make good use of it?

ABN Amro has designed the perfect big data journey: 1 Marketplace, many Domain Data stores.

The journey starts as in any good recipe, in the right and sole Data Marketplace! A virtual repository of the banking data coming from the thousand's applications used in the daily business. In a subscription fashioned manner is possible to simply reach and use what is needed for a Domain Data Store to provide insights. This is the first topic we will have a closer look at.

There are many Domain Data Stores spread across domains and departments that can then simply reshape, remodel and compute new and meaningful analytics for reaching our customers in the most efficient way! According to the Mash Data Architecture each Domain Data Store is a mix of Big Data Technologies, Center of Design, and an Inner Source Development Center that gives a meaning to petabytes of data.

The other component of the marketing automation chain that we'll take a closer look at is the API layer that provides personalised content to the channels of ABN AMRO like the mobile app, internet banking and call center. These APIs and underlying components process millions of requests a day and are developed using a cloud native approach. This is all powered by the architecture that utilises the serverless stack of AWS: APIGateway, Lambda, Aurora and DynamoDb to name a few. The infrastructure is defined in code (Infrastructure as Code) and generic data structures enable content to be presented regardless of the channel it is presented on. During the development of these APIs many challenges were faced and we'll share the lessons learned of developing an enterprise scale API layer.

{ ANNE FERINGA }
IT Lead Marketing Automation @ ABN AMRO

Anne has been working in IT for almost a decade, in various roles and various companies, such as Capgemini, KLM and the Belastingdienst. In her most recent years she has worked for ABN AMRO as a User Experience Lead, Product Owner and IT Lead for the departments that focus on the customer interaction side of the bank.

{ GIUSEPPE GUIDA }
Big Data Engineer @ ABN AMRO

Giuseppe is a Big Data engineer with experience in 3 different countries: Italy, Denmark and now The Netherlands. His expertise lies in building robust data solutions both in the Big Data and Machine Learning fields and in multiple industries: e-commerce (Coolblue), consumer goods (Unilever), financial sector (Intesa San Paolo, ABN AMRO) and consultancy (D60, Reply).

{ WESLEY DELPEUT }
Software Engineer @ Capgemini

Wesley is a Software Engineer at Capgemini and worked primarily in the Financial Services industry. He came a long way around in the IT field given his background as a television producer. He started as a web developer specialised in design systems and today he works as a DevOps engineer at ABN AMRO.

ABN·AMRO Bank

Capgemini

Session details

GRAPHQL-IFY YOUR APIS

Capgemini

Session description: *GraphQL is query language for APIs, but what are the advantages and how would one implement such in their microservices/APIs? In this session, Soham will go through the basics of GraphQL, different aspects of GraphQL and the architecture of such APIs including a demo/live-coding session. Lots of examples, live coding and helpful comparison on structure, usage and implementations of GraphQL in Springboot & Java world.*

It was designed by Facebook to allow its mobile clients to define exactly what data should be sent back by an API and therefore avoid unnecessary roundtrips and data usage, GraphQL is a JSON based query language for APIs. Since it was open-sourced by Facebook in 2015, it was rapidly adopted and many companies have already switch to the GraphQL way of building APIs.

A few months ago Netflix open-sourced their own GraphQL framework(dgs), giving GraphQL in the microservice world an extra boost. Apart from this framework there are a couple of other frameworks also available to implement GraphQL in Java microservice space. But how do these frameworks work, what are the advantages and usefulness of one framework over another. Not only you would learn the basics of GraphQL, different aspects of GraphQL and architecture of such APIs, but also what would you choose to implement your GraphQL APIs.

Soham Dasgupta

Join Soham's session and get to know how to GraphQL-ify your APIs in 3 different ways, receive a detailed comparison among the available frameworks, supported by demo/live coding and most importantly get a better understanding if/when/where/why GraphQL applies for your organization or specific use case.

.BLD()
Tech Talks

EDGE COMPUTING WITH PLANTS, AZURE AND NUXT

Tuesday 25 May | 19.30

REGISTER NOW

Capgemini

Session details

TRANSACTION MONITORING 2.0 WITH MLOPS

Lars Haringa

Jorg Klein

Session description: Hardly a week goes by without news of money laundering affecting a financial institution. As gatekeepers of the financial system, banks play a crucial role in finding and reporting instances of financial crime. In this talk, we describe the challenges of detecting money laundering and outline why employing machine learning techniques is critically important to identify complex and ever-changing patterns.
We will present an overview of data science work related to transaction monitoring, and deep-dive into some of the challenges that we faced at ABN AMRO as well as the solutions we came up with. This includes how we continuously enhance an existing rule-based system with machine learning, as well as a method to deal with sudden data shifts while running in production. We will also discuss an in-house-developed model explainability technique that helps our transaction monitoring analysts focus their investigation and better understand what caused the alert.
A key capability for our enhanced transaction monitoring setup is how we facilitate both monitoring and retraining machine learning models live in production. MLOps has become the de facto way of working for teams that need to bring ML models to production. But how do you actually move from a few single models created by data scientists on an experimentation environment, to dozens of models that are fully automatically trained, tested, deployed, monitored and retrained – while ensuring traceability and reproducibility? We will illustrate the journey organizations go through as they move towards the next MLOps maturity level.

Day: *Monday May 10, room 1, time 17:10-17:50*

ABN·AMRO Bank

Session details

AI, HOW DO WE DEAL WITH IT?

Lisa noorlander

Ramon Ankersmit

Session description: In the present age of information technology, many data processing machines govern our daily lives: from servers that keep records of our financial transactions to mobile phones that track our voices and locations. Every day these machines become more independent and intelligent. Using artificial intelligence techniques, organisations and governments can lighten their employees' workloads, find errors, and predict results. At the Dutch tax office, several departments already use these techniques to improve their methods and processes.
No matter how great this may sound, we must remember that these AI techniques are used both for people and by people. And people simply make mistakes. So, do AI models actually respect our human rights? Do we have any privacy left? Can we assume AI makes decisions free from biases?
During this session we explain how to make a reliable AI model. We also show what products we are already developing using machine learning at the Dutch tax office and how we include ethical considerations in our development process. Lastly, we will show what happens when we let the free Internet loose on AI without caution...

Day: *Tuesday May 11, Room 2*
Time 17:10-17:50

Belastingdienst

A SNEAK PREVIEW OF TEQNATION 2021

TEQnation 2021 is coming. The biggest multitrack developer conference of the Netherlands will be fully online this year. What can you expect on TEQnation? In this article, we will highlight two very different but both pretty remarkable demos.

Damiaan Zwietering

Edward Ciggaar

{ HOW TO BUILD A CORONARADAR? }

Off course, Covid-19 plays a huge role in TEQnation 2021 – and not only because of the social restrictions and lockdown. IBM Developer Advocate and Datascientist Damiaan Zwietering has developed a coronaradar. This TEQnation you can follow a session in which he demonstrates his model and explains how to build it. All you need is a bit of Python knowledge and a small set of open data.

How do you get from raw reported data to an animated map of projected disease spread? Damiaan build a relatively simple model to interpretate the Covid-spread in a specific country. The model shows if a wave exists out of one gigantic peak or more, almost simultaneously peaking small waves. Damiaan build his model using open source libraries such as NumPy, Pandas, PyEarth and Folium and a bit of high school mathematics. He relied upon some visualization, curve fitting, modeling, mapping and Python to tie it all together.

You don't need a huge, unstructured dataset to build a radar-like model. For his coronaradar Damiaan used a minimal, open dataset with the location, numbers and date of contaminations. Why? Big datasets tend to show a lot of random correlations, which aren't relevant for the questions you are trying to answer. A relatively limited dataset has another benefit: it makes the model widely applicable. In this case, you can easily copy-paste the Dutch model to the situation in every other European country. All you need are the open EU CDC data for the three parameters 'location, 'numbers' and 'date'.

{ CAN APP DEVELOPMENT BECOME ANY MORE AGILE? }

Sure, if you make the switch to cloud native development. In this demo and workshop on TEQnation 2021 IBM Developer Advocate Edward Ciggaar will demonstrate how cloud native application development can make your live in the agile world a lot less complicated.

Are CI/CD, automated testing, static analysis, vulnerability checking, logging and monitoring properly in place? For most developers, these are the first steps in efficiently building a qualitative app. But is it really necessary to implement all these tools and processes? In this workshop Edward will demonstrate how to kickstart your projects in a simple, but enterprise ready way.

The Cloud Native Toolkit is the key to implement all the necessary tools and processes in a much faster and straightforward manner. This toolkit consists of an open-source collection of assets that enable application development and support teams to quickly deliver business value using Red Hat OpenShift or Kubernetes. The tools themselves, like Tekton for continuous integration and Argo-CD for continuous deployment, are well recognized and broadly used in the market.

In this preview we offered you a sneak preview of two sessions at TEQnation 2021. It is no secret that the event will address a broad range of other subjects – from cloud native app development to building a coronaradar and everything in between.

IBM

CURIOUS WHICH OTHER DEMOS AND WORKSHOPS TEQNATION 2021 HAS TO OFFER? VISIT HTTPS://CONFERENCE.TEQNATION.COM

Session details

STABILIZING AND SYNCHRONIZING OUR JAVA AND .NET CODEBASE

Session description: How to achieve continuously releasable code, and to provide functionality in more than one language without too much effort. Our Open Source codebase is available on two platforms, Java & .NET and it wasn't an easy task to keep them always in sync and buildable. In the old days we did this manually, risking broken develop branches in both codebases and this meant getting a .NET release out could take a month (or more).
In this talk we'll share how we overcame these hurdles and transitioned from a manually tested and ported codebase to an automated system, where develop is always green! It's continuously being improved, such as the introduction of build agents in the cloud. This would enable us to run tests on different platforms and configurations almost effortlessly. We'll explain how we achieved this by introducing different tooling. The main part of our talk will be about our Merge Pipeline (trademark pending) based on Jenkins, the backbone of our fully-fledged automated system. We'll share its internal details and explain how it handles the different steps that are needed to get a Java branch merged, and automatically ported into the Java and .NET develop branches. It's designed in such a way that as little as possible time is wasted by running steps in parallel, keeping track of what was, or was not run already. Code that does not pass Sonar will not make it into develop. Functional tests are in a separate repository but go together with the code they are testing.

Demey Michaël

Day: *May 12, room 2, time 17:10-17:50*
Track: *Developer Experience*
Level: *Intermediate*
Language: *English*
Coding Language(s): *Minimal Java and C#*

Session details

HOW TO BECOME A BETTER DEVELOPER WITH A HACKER'S MINDSET

Sébastien Hoekstra

Session description: Hackers look at an application in a very different way than developers do. Where you as a developer see high-end tech stack, they see opportunities to undermine your great piece of work. That is why it's good to move into the hacker's mindset and see an application in the way they do. By doing this you will challenge yourself and deliver more secure tech stack. In this talk we will dive into the world of ethical hacking and red teaming and take you through the context and background I wish I had when I was a developer myself.

Level of talk: medior/senior developers with interest to work/move into cyber security

Day: Monday, May 10
Room: 2
Time: 16:10-16:50

AWS
Serverless Application Model

Je wilt een serverless applicatie bouwen en deployen naar de AWS Cloud. Een serverless applicatie kan uit meerdere functions bestaan, en daarnaast interactie hebben met andere AWS (of externe) services of resources zoals een database of message queue. Maar hoe bouw, test, debug en deploy je een dergelijke applicatie in de AWS Cloud? Kun je de applicatie lokaal ontwikkelen en voor de code gebruikmaken van je favoriete IDE?

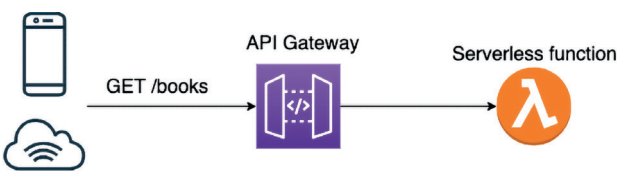
Oussama Chougna *is Software Engineer bij Blue4IT. Hij is enthousiast over Cloud en houdt zich momenteel bezig met API engineering & security.*

AWS biedt je de mogelijkheid om serverless functions te schrijven in de browser, op zich is dat prima om wat kleine serverless functions uit te proberen, maar bij het schrijven van meerdere functions of grotere serverless applicaties is het ontwikkelen in de browser niet handig. Het gaat je een hoop rond geklik opleveren. Daarnaast kun je in de browser ook niet debuggen en moet je het alleen met de logs doen wat neer komt op een hoop trial and error. Ook ondersteunt de code editor van AWS Lambda, wel de Python en NodeJs runtimes maar nog niet die van Java. Voor Java moet je code in een zip of jar uploaden wat je trial and error nog omslachtiger maakt.

AWS Serverless Application Model (AWS SAM) is een framework die developers helpt bij het definiëren, bouwen, testen, debuggen en deployen van serverless applicaties in de AWS Cloud. In dit artikel gaan we hier kennis mee maken door een kleine eenvoudige serverless applicatie op te zetten.

{ DE SERVERLESS APPLICATIE }

In de afbeelding zie je een eenvoudige serverless applicatie. Het is een REST API met een serverless backend. Consumers komen binnen op de API gateway, http requests worden door de gateway ge-route naar een serverless function in AWS Lambda.



{ ONTWIKKELOMGEVING OPZETTEN }

Voor het ontwikkelen van de serverless applicatie heb je de volgende zaken nodig:

- Je favoriete IDE
- Docker [1]
- De SAM Command Line Interface (SAM CLI) [2]

De SAM CLI vormt de basis voor AWS SAM om serverless applicaties te bouwen.

{ BOOTSTRAPPEN VAN DE APPLICATIE }

Een AWS serverless applicatie wordt gedefinieerd middels een template. Dit template is gebaseerd op AWS Cloudformation templates en heeft het JSON of YAML formaat. Met Cloudformation templates definieer je alle AWS resources die deel uitmaken van je applicatie, dit wordt ook wel een stack genoemd. Het is een vorm van infrastructure as code. Het bootstrappen doe je met de SAM CLI met

```
'sam init --name book-store --runtime java11
--dependency-manager maven --app-template hello-
world'
```

Bovenstaand commando genereert een directory met de naam book-store met daarin een compleet maven project voor je serverless applicatie. Tevens vind je in deze directory de template.yml met de definitie van de serverless applicatie. AWS SAM maakt gebruik van Cookiecutter [3] om deze bestandsstructuur te genereren. Dit gebeurt op basis van een Cookiecutter template, via de --app-template parameter hebben we aangegeven gebruik te willen maken van een AWS Quick start template genaamd hello-world. Dit is de snelste manier om te beginnen. We kunnen

L1

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: >
 SAM Template for book-store
Globals:
 Function:
   Timeout: 20 # Standaard (global) timeout voor alle lambda functies
Resources:
 GetBooksFunction: # De definitie van de lambda function GetBooksFunction voor het ophalen van boeken.
   Type: AWS::Serverless::Function # Type resource is een AWS Lambda Function
   Properties:
     CodeUri: GetBooksFunction # Dit verwijst naar de code van de functie. Dit is een maven module met
     deze naam in de huidige directory.
     Handler: nl.java.magazine.GetBooks::handleRequest # De fully qualified class name + functienaam
     die de code bevat
     Runtime: java11 # De functie moet in een java11 runtime draaien
     MemorySize: 512 # 512 MB memory
     Events: # Events die de lambda functie kunnen 'triggeren'.
       BookStore:
         Type: Api # een Api gateway event.
         Properties:
           Path: /books # URI path voor de Api
           Method: get # HTTP methode
```

L2

```
public class GetBooks implements RequestHandler<APIGatewayProxyRequestEvent,
APIGatewayProxyResponseEvent> {

   public APIGatewayProxyResponseEvent handleRequest(final APIGatewayProxyRequestEvent input, final
   Context context) {
       final String output = this.getBooks();
       return new APIGatewayProxyResponseEvent()
               .withHeaders(Map.of("Content-Type", "application/json"))
               .withStatusCode(200)
               .withBody(output);
   }

   private String getBooks() {
       return "{ \"books\" : [{ \"title\": \"JSON for Dummies\", \"author\": \"Jackson\" }, {
       \"title\": \"YAML for Dummies\", \"author\": \"SnakeYML\" }]}";
   }
}
```

nu prima het gegenereerde voorbeeldproject aanpassen aan onze wensen.

{ DE TEMPLATE AANPASSEN }

In afbeelding 1 hebben we gezien dat onze serverless applicatie bestaat uit een API gateway, en een lambda functie. Deze AWS resources definiëren we in een SAM template. Het gegenereerde template.yml bestand uit het vorige hoofdstuk moeten we dus aanpassen. Zie L1 voor het aangepaste bestand met uitleg.

Zoals we eerder in afbeelding 1 hebben gezien, zit onze serverless functie achter een API gateway. Deze AWS resource vind je niet terug in de template.yml onder de Resources property, maar deze staat echter impliciet wel gedefinieerd. Namelijk onder de Events

L3

```
$ sam local start-api
Mounting GetBooksFunction at
http://127.0.0.1:3000/books [GET]
You can now browse to the above endpoints to
invoke your functions. You do not need to
restart/reload SAM CLI while working on your
functions, changes will be reflected instantly/
automatically. You only need to restart SAM CLI
if you update your AWS SAM template
2021-01-26 23:47:23  * Running on
http://127.0.0.1:3000/ (Press CTRL+C to quit)
```

property van de GetBooksFunction. Omdat een Api event de serverless function kan triggeren, zal AWS SAM automatisch een API gateway voor ons provisionen tijdens deployment en de function beschikbaar stellen via een HTTP GET op het pad /books.

{ DE SERVERLESS CODE }

In de vorige paragraaf hebben we in het template.yml bestand een AWS serverless functie gedefinieerd. Een van de properties is de 'CodeUri', dit is een verwijzing naar de maven module die de serverless code bevat. De 'Handler' property geeft aan dat we een functie 'handleRequest' in een klasse 'GetBooks' nodig hebben. Deze klasse en functie vind je terug in L2.

{ BOUWEN }

Als de serverless functiecode klaar is, is het tijd om te gaan bouwen en testen. Voor het aftrappen van de build gebruik je de SAM CLI command `sam build`. AWS SAM zal onder water o.a. via maven je project bouwen en de output van de build plaatsen in een .aws-sam directory.

{ TESTEN }

De functie wordt aangeroepen via een API gateway. We gaan eerst lokaal een API gateway opstarten via het SAM cli command `sam local start-api`, zie L3.

SAM heeft voor ons een API gateway opgestart, maar ook gelijk onze serverless functie daarachter beschikbaar gesteld. Dit komt omdat SAM de template.yml (listing 1) parsed en dus ziet dat de serverless functie via een API gateway aangeroepen wordt. We gaan het endpoint op de API gateway aanroepen, zie L4.

L4

```
$ curl http://127.0.0.1:3000/books
{ "books" : [{ "title": "JSON for Dummies",
"author": "Jackson" }, { "title": "YAML for
Dummies", "author": "SnakeYML" }]}
$
```

De output van het curl command komt overeen met onze implementatie van de function in listing 2. Op de achtergrond zal SAM eerst een docker image ophalen. Dit image bevat de execution environment van AWS Lamba met Java 11, dit garandeert dat je serverless functie in een soortgelijke omgeving draait als in de AWS Cloud. Dit is sowieso een best practice bij het ontwikkelen en testen van code. Op de achtergrond zal dus een docker container worden opgestart waarin je serverless functie zal draaien. Vervolgens wordt het request afgevuurd op de serverless functie GetBooksFunction en het response wordt weer via de API gateway teruggestuurd. Je kunt de serverless functie ook direct aanroepen zonder API gateway, dit doe je met `sam local invoke GetBooksFunction`.

{ DEBUGGEN }

Het `sam local invoke` command kent voor het debuggen de parameter **--debug-port** (of **-d**) met daarachter het poortnummer waarop de debugger een connectie kan maken. Helaas is er op dit moment nog geen ondersteuning voor Java runtimes. Wel biedt de AWS Toolkit andere mogelijkheden om te debuggen, zie hiervoor [4].

{ PACKAGE & DEPLOY }

Voordat je kan deployen dien je de applicatie eerst te packagen, het is hierbij noodzakelijk dat je build output ergens op de AWS Cloud wordt opgeslagen. In dit geval in Amazon S3, de storage service van AWS. Voor het packagen gebruik je de CLI command **sam package**. De build output wordt geüpload naar een S3 bucket, zo'n bucket kun je vergelijken met een folder op een filesystem. Na het packagen kun je met het **sam deploy** command de applicatie daadwerkelijk deployen naar de AWS Cloud. Zowel de API gateway als de lambda functie staan dan 'live'.

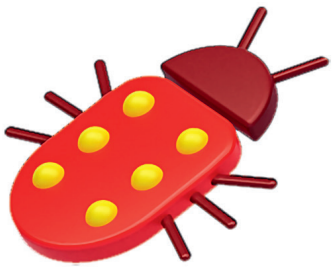
{ CONCLUSIE }

In dit artikel hebben we kennis gemaakt met het AWS SAM framework. Het framework helpt ons een AWS serverless applicatie lokaal te ontwikkelen en testen. Hierbij kun je gebruik maken van je favoriete IDE. De code, de template.yml met je serverless applicatie stack kun je in een versiebeheersysteem zetten zoals GIT. Bovendien kun je eenvoudig de serverless applicatie stack packagen en deployen naar de Cloud. ‹

REFERENTIES

[1] https://docs.docker.com/get-docker/
[2] https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install.html
[3] https://cookiecutter.readthedocs.io/en/1.7.2/#
[4] https://aws.amazon.com/intellij/

De kunst van het BUGFIXEN



Een belangrijk deel van het werk van een ontwikkelaar is het fixen van bugs. Daarnaast is het één van mijn favoriete professionele bezigheden. Ik zie elke bug als een puzzel. Echter, je hoort en leest er bijna niks over. Met dit artikel wil ik daar verandering in brengen. Ik zal hier uit de doeken doen wat ik de afgelopen jaren heb geleerd over het fixen van bugs.

Bouke Nijhuis is managing consultant bij CINQ ICT.

Allereerst een stukje context. Je bent onderdeel van een team dat software schrijft. Uiteraard draait deze software niet in isolatie en communiceert het met andere systemen. Tot slot beoefent het team devops, dus je bent verantwoordelijk voor de ontwikkeling en het beheer van de software.

Het artikel beschrijft in essentie welke fases er zijn in een bug fix traject. Bij elke fase wordt omschreven hoe je deze herkent en wat je vervolgens kan doen. Tot slot is er een kader met hoe je in de toekomst nog sneller bugs kan fixen.

{ DETECTIEFASE }

Het leven van een bug begint op het moment dat de bug ontstaat. Echter, je weet nog niet dat de bug bestaat, want anders had je hem wel direct gefixed of helemaal nooit laten ontstaan. De eerste keer dat je je bewust wordt van een bug is vaak de melding van een incident. Uiteraard kun je zelf ook een bug vinden, maar dit artikel gaat er vanuit dat iemand anders de bug vindt.

Zo start de detectiefase. Je wordt je bewust van het feit dat er iets niet helemaal in de haak is. Er is nog veel onduidelijk. Het is nog niet zeker dat het een bug is, en het is ook onduidelijk of het probleem zich in je eigen software of in één van de aanpalende systemen bevindt.

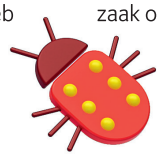
Uit het bestaan van een incident kun je afleiden dat er een gebruiker is met een onverwachte ervaring. Het eerste dat je nu wilt doen, is controleren of het ook echt wel een bug is. Ik heb meerdere keren meegemaakt dat het gemelde gedrag een feature was. Oftewel, in dit geval heeft de gebruiker van de software een verkeerde verwacht van het systeem. Ben je ervan overtuigt dat het een bug is? Ga dan proberen het probleem te reproduceren. Als je geluk hebt dan heeft de gebruiker voldoende informatie toegevoegd om dit te kunnen doen. Als dit niet het geval is, zul je erom moeten vragen. Zonder de mogelijkheid tot reproductie wordt het erg moeilijk om te bewijzen dat er een bug is en dat deze daadwerkelijk is opgelost aan het einde van het traject. In de praktijk worden niet-reproduceerbare bugs niet vaak opgelost.

{ REPRODUCTIEFASE }

Het is nu tijd om zelf de bug na te spelen. De hoofdreden hiervoor is snelheid. In het hele traject zul je de bug nog vaak moeten naspelen. Als je hiervoor elke keer iemand anders nodig hebt, dan zal het traject vele malen langer duren dan wanneer je zelf in staat bent om de bug te reproduceren.

Het einddoel van deze fase is het reproduceren van de bug op je lokale systeem. Soms is het probleem te ingewikkeld om dit in één keer voor elkaar te krijgen. Een goede tussenstap is dan het zelf naspelen op de omgeving waar de bug is gevonden. Soms is het te moeilijk om het lokaal te reproduceren. In dat geval is naspelen op een andere omgeving het best haalbare.

Als je in staat bent om de bug zelf te reproduceren, dan is het zaak om dit zo efficiënt mogelijk te doen. In deze fase is het



waarschijnlijk moeilijk om een unittest te maken die het probleem aantoont. Echter, het is vaak wel mogelijk om een end-to-end test of zelfs een integratietest te maken die het probleem naspeelt. Hoe sneller je het probleem kan naspelen, hoe sneller de volgende fases verlopen. Het is belangrijk om hier niet op te bezuinigen. Een investering om snel te kunnen reproduceren, zal zich later dubbel en dwars terugverdienen.

{ ONDERZOEKSFASE }

Nu je in staat bent om de bug snel te reproduceren, is de vervolgstap onderzoeken wat de bug veroorzaakt. Er is een aantal technieken die je hierbij kan gebruiken. Het eerste dat je probeert, is de locatie van het probleem vinden. Als er een stacktrace is dan is dit relatief simpel. Met een beetje geluk geeft de stacktrace de exacte locatie in code waar het probleem optreedt. Soms heb je alleen de locatie waar de exceptie is afgevangen, maar vaak kan je van daaruit wel beredeneren waar de fout exact vandaankomt.

Als er geen stacktrace is, dan is het lastiger om te achterhalen waar het probleem optreedt, maar het is zeker niet onmogelijk. De volgende plek om te kijken is de logfile. Bekijk welke logging er is rondom het optreden van de bug. Is er op dat moment geen enkele logging? Verhoog dan het log level en probeer het nog eens. De meeste applicaties loggen normaal gesproken op het niveau info. Voor meer informatie zet je het niveau op debug of nog hoger. Nog steeds geen interessante logging? Dan is het tijd om om zelf logregels aan de code toe te voegen. Als je echt geen idee hebt waar het probleem optreedt, dan begin je met het toevoegen van logregels op plekken waar je denkt dat het misgaat.

Vervolgens gebruik je deze nieuwe logregels om het probleem te lokaliseren. Je blijft net zo lang logregels toevoegen, totdat je precies weet waar het misgaat. Pas met het loggen wel op dat je geen gegevens logt die privacygevoelig zijn. Mocht deze toch willen loggen, encrypt de logging dan dusdanig dat alleen jij de logging kan lezen.

Als je eenmaal weet waar het misgaat, dan is de vervolgstap om te kijken waarom het misgaat. Mocht je een stacktrace hebben, is het nu tijd om deze een rustig te lezen. Het is mij meerdere keren overkomen dat ik veel tijd had bespaard als ik rustig de foutmelding had gelezen. Een stacktrace bevat vaak een melding met daarin informatie wat er misgaat of wat er werd verwacht. Een ontwikkelaar heeft de tijd genomen om een extra bericht aan de stacktrace toe te voegen. Er is dus een behoorlijke kans dat deze informatie je gaat helpen om het probleem te begrijpen.

BUG IN EEN LIBRARY

Het komt voor dat de bug niet in je eigen code zit, maar in een gebruikte library. Door gebruik te maken van de debugger weet je precies waar de bug zich bevindt. De eerste stap is dan een issue aanmaken bij het project dat de library beheert. De overtreffende stap is dan een pull request maken die de bug oplost. Dit laatste kan redelijk wat tijd en energie kosten, maar hoe gaaf is het als je kan zeggen dat er code van jou in een bekende library zit?!

Als er een foutmelding is, dan is het altijd handig om deze te googelen. Een grote kans dat je dan op Stack Overflow uitkomt en dat is precies waar je wilt zijn. In de beste gevallen levert dit de oplossing en in mindere gevallen geeft het nieuwe inzichten of richtingen om verder te onderzoeken.

Mocht Google de oplossing niet hebben, dan is de volgende stap debuggen. Op dit moment in het traject weet je precies op welke regel code het probleem zich voordoet. Dus je zet een breakpoint op deze regel en je start de applicatie in debug modus. Als je vervolgens de bug reproduceert, zal de debugger stoppen op de regel met het breakpoint en kun je uitgebreid onderzoeken waarom de bug optreedt. Je kijkt dan voornamelijk of de variabelen gevuld zijn met de verwachte waarden. Het komt voor dat het probleem zich niet bevindt in eigen code, maar in een library. Ook dit is geen probleem, want elke moderne ontwikkelomgeving kan prima overweg met breakpoints in libraries. Je zou nu voldoende informatie verzameld moeten hebben om door te gaan naar de volgende fase.

Echter, als je na bovenstaande tips nog steeds geen idee hebt wat het probleem veroorzaakt, is mijn laatste redmiddel de rubber ducky techniek. Je gaat het probleem uitleggen aan een uh... rubberen badeendje. Het idee hierachter is dat het uitleggen van het probleem je nieuwe inzichten gaat geven. Deze techniek heeft mij meerdere keren geholpen om een probleem beter te begrijpen. In mijn geval moet de rubber ducky een echt persoon zijn, want anders werkt de techniek voor mij niet. Dit heeft als bijkomend voordeel dat de rubber ducky na het luisteren jou kan helpen het probleem te doorgronden.

{ OPLOSSINGSFASE }

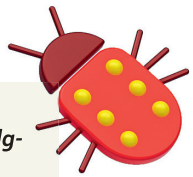
Je weet nu precies wat het probleem veroorzaakt en het is tijd om een oplossing te maken. Als het probleem zich voordoet in een library en het daar ook moet worden opgelost, is daar nog een kader over. Deze paragraaf gaat ervan uit dat het probleem wordt opgelost in eigen code.

De eerste stap is een unittest schrijven die de juiste werking van het systeem test. Deze unittest zal nu nog falen, want de bug is nog niet opgelost. Echter, na het schrijven van de test, ga je de bug fixen en behoort de unittest groen te worden. Er zijn twee belangrijke redenen om eerst een unittest te schrijven. Allereerst kan je deze unittest gebruiken om te bewijzen dat het probleem is opgelost. Daarnaast zal deze voorkomen dat hetzelfde probleem nog een keer optreedt in de toekomst.

Als de unittest slaagt, is het tijd om handmatig te controleren of het probleem echt is opgelost. Het komt voor dat er meerdere problemen onder één incident schuilgaan. Het fixen van één probleem is dan niet voldoende. Het handmatig controleren doe je eerst op je eigen machine. Als het probleem lokaal verdwenen is, breng je een nieuwe release naar productie om daar vervolgens ook nog eens te testen. Dit lijkt overkill, maar het komt voor dat de gekozen oplossing alleen lokaal werkt. Het is dus belangrijk om ook altijd op productie te kijken of het probleem ook echt weg is.

{ AFRONDINGSFASE }

In de laatste fase zijn er nog een paar dingen die je zou kunnen doen. Allereerst het afsluiten van het incident. Dit laat de gebruiker weten dat het probleem is opgelost. Vervolgens is het handig om je teamgenoten op de hoogte te brengen. Zeker in het geval van complexe bugs schrijf je een post-mortem of presenteer je hoe de bug is gevonden en hoe deze is verholpen. Dit zorgt ervoor dat hetzelfde type bug in de toekomst sneller wordt opgelost. Tot slot vier je dat de bug is opgelost! <



SNELLER BUGS FIXEN

Je hebt gelezen hoe je bugs kan fixen. Echter, de vervolgstap is dit in de toekomst nog sneller te kunnen. Er zijn een aantal zaken belangrijk. De meeste zijn hierboven al aan bod gekomen, maar ik herhaal hier de belangrijkste punten.

Zorg dat je logging op orde is. Zonder goede logging is het moeilijk om de locatie van de bug te vinden. Elke keer dat je logregels moet toevoegen, moet je ook releasen. Dit is tegenwoordig een stuk goedkoper dan vroeger, maar het kost je nog steeds tijd. Dus goede logging kost tijd op de korte termijn, maar bespaart veel tijd op de lange termijn.

Zorg dat je handig bent of wordt met een debugger. Verdiep je in de mogelijkheden van dit geweldige hulpmiddel. Een paar losse tips. Wist je bijvoorbeeld dat een debugger overweg kan met elk Java proces dat er open voor staat? Het Java proces hoeft niet eens op je eigen computer te draaien. Dit wordt remote debugging genoemd. Wist je dat je ook conditionele breakpoints aan kan maken? Soms wil je alleen stoppen als een bepaalde conditie waar is. Verder heeft IntelliJ een stream debugger en deze kan (heel verrassend) streams debuggen. Deze zijn van nature lastig te onderzoeken, maar met deze tool wordt het een stuk makkelijker. Ik hoop in de toekomst nog een keer een artikel te schrijven dat specifiek ingaat op dit gereedschap.

De laatste tip om sneller bugs te fixen, is het hebben van een grote variatie in je unit- & integratietests. Dit verhoogt de kans dat je een bestaande test kan kopiëren en aanpassen om het probleem te reproduceren en de gevonden bug af te dekken. Dit is natuurlijk veel sneller dan een compleet nieuwe test schrijven. Je behaalt dan winst in de reproductie- en oplossingsfase!

TEAMSFEER EN BETROKKENHEID
en dan pas het gereedschap

Hier is een leuke oefening als je al wat jaren en werkgevers op je cv als developer hebt staan. Geef al die projecten eens een intuïtief punt. Bij een tien zou je morgen meteen weer willen beginnen en bij een drie nog niet als ze je het dubbele salaris boden. Denk er vooral niet te lang over na. Klaar? Geef nu elk project afzonderlijk punten voor de volgende drie eigenschappen.

Jasper Sprengers *is na 20 jaar in verschillende rollen nog altijd nieuwsgierig, zowel over het development vak zelf als over de technieken erachter.*

De eerste zou je het *nerd appeal* kunnen noemen: gebruikte je innovatieve tools, technologieën en/of werkvormen? De tweede gaat over de sfeer binnen het team en de organisatie: hoe hartelijk, behulpzaam en professioneel was die? Hoe zat het tenslotte met je affiniteit tot de organisatie en het product waar je aan werkte? Kreeg je daar een gevoel van trots van, of liet dat je vrij onverschillig? Misschien werkte je wel aan een product waar je moreel niet achter kon staan, zoals bemiddeling van stoute afspraakjes voor mensen in een vaste relatie.

Als ik terugkijk op twintig jaar in de IT, scoorden de projecten en werkgevers waar ik 't het best naar mijn zin had altijd hoog op teamsfeer en voelde ik affiniteit met het product en verbondenheid met de organisatie, zelfs als inhuurkracht. Technieken en hardware waren daar meestal ondergeschikt aan. Toen ik in 2012 een project louter koos omdat ik met Google Web Toolkit kon werken, liep dat uit op een teleurstelling. Toch lijken met name recruiters te denken dat IT'ers allereerst een werkomgeving zoeken waar ze met de hipste technieken kunnen werken. Of je er ook met inspirerende collega's aan waardevolle producten kunt werken, beschouwen ze kennelijk als vanzelfsprekend, maar dat is het zeker niet.

{ MOET JE ALTIJD EARLY ADAPTER WILLEN ZIJN? }

Ik snap die nadruk op innovatie wel. Het Java-wereldje is immers klein en als je goede developers wil aantrekken, moet je als bedrijf niet bekend staan als die stoffige Java 6 dinosaurus die hun monoliet nog op JBoss 4 draait. Ik ben ook niet happig op zulke projecten. Zoiets ruikt verdacht naar onbeheersbare *technical debt*. En als het team graag met legacy werkt, heb ik er niets te zoeken. Daarentegen leerde mijn eerste kennismaking met Docker in 2013 mij dat je ook niet per se early adopter moet willen zijn, zeker niet als een techniek nog in de kinderschoenen staat. Een JavaEE / Oracle stack is misschien niet heel hip, maar ik heb liever een saaie en voorspelbare productie-omgeving dan een innovatieve die op willekeurige zondagavonden kuren heeft. "Kom bij ons en je mag werken met AWS, Serverless, Kotlin, Firebase, etc.". Allemaal hele coole spullen, maar vertel me eerst maar eens met *wie* ik mag werken. Want in een fijn team ga je fluitend naar je werk, terwijl die coole stack er niet meer toe doet als je een of twee arrogante einzelgängers in je groep hebt.

IN EEN FIJN TEAM GA JE FLUITEND NAAR JE WERK

Een disfunctioneel team is niet acceptabel en een verantwoordelijke organisatie moet tijdig maatregelen nemen als zoiets dreigt. Daar zijn boekenkasten over volgeschreven, maar ook zonder dure coaches kun je zelf al veel doen om de binding in je team te verbeteren. Het begint met de juiste instelling jegens je collega's:

OM MEER BETROKKENHEID TE CREËREN, MOET JE JE MENSEN WEL ACTIEF BETREKKEN. WEES DUS WAAKZAAM DAT ER GEEN EILANDCULTUUR ONTSTAAT

ga ervan uit dat het gemotiveerde vakmensen zijn en behandel elkaar zodanig. Werk zoveel mogelijk samen in wisselende combinaties. Dat is niet eenvoudig; integendeel zelfs. Je moet veel beter nadenken over het ontwerp om een codeerklus met drie mensen aan te pakken, maar het resultaat is navenant beter. Het is misschien aantrekkelijker om jezelf in te graven in een probleem en trots als eerste met de oplossing te zwaaien. Maar met dit sologedrag kweek je geen teamgeest. Laat mensen trouwens de samenwerkingsvorm kiezen waar ze zich prettig bij voelen. Niet iedereen doet bijvoorbeeld graag aan Pair Programming. Wees kritisch op elkaars werk, maar maak het niet persoonlijk. Het leuke aan development is dat het zo breed is dat seniors altijd nog kunnen leren van juniors. En de belangrijkste tip: niemand houdt van betweters.

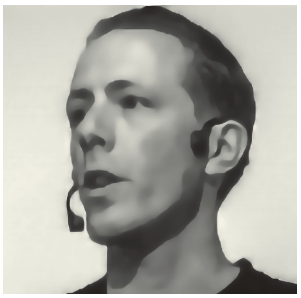
{ EAT YOUR OWN DOG FOOD }

Tenslotte vind ik naast teamgeest je betrokkenheid bij het product en de organisatie een stevige factor van betekenis. Zeg eens eerlijk, hoe vaak heb je jouw eindgebruikers daadwerkelijk het product zien gebruiken? Misschien ben je zelf wel die eindgebruiker. Gefeliciteerd, in dat geval hoor je bij de club van *eat your own dog food*. Ik heb dat zelf niet vaak meegemaakt. Als de klant tevreden was, het team leuk en de techniek boeiend zat ik daar niet zo mee. En toch laat je dan een waardevolle kans liggen. Bij het Havenbedrijf Rotterdam hadden ze dat goed begrepen. Tijdens de introductie bezochten we het coördinatiecentrum op de Kop van Zuid, waar je kilometers ver de schepen kon zien die de verkeersregelaars met onze software in goede banen leidden. Daarna konden we ze tijdens een rit met een patrouillevaartuig ook nog van dichtbij zien. Ondanks dat we niet de eindgebruikers waren van onze eigen software, werden we wel vanaf de eerste dag bij het proces betrokken.

Concluderend zou ik willen stellen dat bedrijven beter moeten beseffen dat developers interesses, normen en waarden hebben die verder reiken dan regels code kloppen.
Misschien nemen sommigen een matige teamsfeer voor lief als ze kunnen werken aan een geweldig innovatief product, zoals een super-efficiënte auto-accu die in vijf minuten laadt. Wie weet. Zelf weet ik in ieder geval zeker dat ik als niet-roker, die zijn grootvader nooit gekend heeft vanwege een vroege tabaksdood, bijvoorbeeld nooit zou willen werken voor Philip Morris of een lobbyist van die industrie. Deze voorbeelden zijn bewust wat gechargeerd, maar ertussenin valt een wereld te winnen. Als developer kun je meer interesse tonen in hoe de eindgebruikers jouw product gebruiken, ook als je niet rechtstreeks aan de gebruikersinterface werkt.
Voor het management is hier nog een grotere taak weggelegd, want om meer betrokkenheid te creëren, moet je je mensen wel actief betrekken. Wees dus waakzaam dat er geen eilandcultuur ontstaat. Betrokken medewerkers zijn enthousiaste medewerkers, en enthousiaste mensen zijn doorgaans productief en trouw aan je bedrijf. Daar kun je zakelijk toch niks op tegen hebben? <

PATTERN MATCHING maakt Java nóg krachtiger

Lambda's en streams hebben zich sinds Java 8 in onze geliefde taal gemanifesteerd en zijn een krachtig middel geworden. In de komende versies van Java worden meer bewezen features uit functionele talen geïntegreerd. Eén daarvan is pattern matching (niet te verwarren met reguliere expressies): een elegante manier om de kenmerken van een object te testen. Wij zochten uit wat er nu al mogelijk is door de introductie van JEP 305 ('Pattern Matching for instanceof') en hoe verdere pattern matching-plannen Java een nóg krachtigere taal kunnen maken.



Hanno Embregts *is software-architect, conferentiespreker en trainer bij Info Support.*

Peter Wessels *is IT-consultant en trainer bij Info Support. Als software engineer bij onder andere ING, Translink en WWplus, zoekt hij altijd een manier om met Java impact te creëren.*

Iedereen maakt het wel eens mee: je wilt bepalen of een object van een bepaald type is om er een methode van dat type op uit te voeren. Zo'n constructie vergt meestal dezelfde handelingen: een `instanceof`-test, het casten van het object en het toekennen van het resultaat aan een variabele. Zulke taalconstructies zijn repetitief, foutgevoelig en bovenal vervelend om te moeten schrijven. Dat moet toch simpeler kunnen!

{ TYPE PATTERNS }

'Pattern Matching for instanceof'[1], sinds Java 14 een preview feature, beschrijft een eerste stap in het toevoegen van pattern matching aan Java. Met deze functionaliteit kunnen we de drie eerder genoemde stappen vervangen door slechts één expressie[2] (zie Listing 1).

In het voorbeeld definiëren we een pattern `Musical m`, dat bestaat uit een type `Musical` en een label `m`. In een `instanceof`-test wordt het type gebruikt om een object van dat type te matchen. Nadat de match heeft plaatsgevonden, wordt het object automatisch gecast en toegekend aan de variabele met het gedefinieerde label. Zo vervang je met een enkel pattern de expliciete cast en het toekennen aan een variabele. In een `equals`-methode is de impact wellicht nog beperkt, maar de kracht wordt pas echt duidelijk wanneer we in de toekomst pattern matching kunnen gaan gebruiken in een `switch`-expressie[3]. De syntax kan nog wijzigen[4], maar een voorbeeld zie je in Listing 2.

Je ziet het; het combineren van de nieuwe `switch`-expressie met pattern matching levert verrassend korte en bondige code op. En dit is nog maar het begin.

{ CONSTANT PATTERNS }

Naast type patterns is een tweede soort pattern jullie al bekend: de constante `case`-labels in een `switch`-statement. De numerieke, String- en `enum`-waardes die daarbij horen noemen we in het vervolg *constant patterns*. Een nieuwe naam voor iets dat al lang bestaat, om zo duidelijker te maken dat de `case`-labels in de toekomst allerlei soorten patterns aankunnen.

{ DECONSTRUCTION PATTERNS }

Deconstruction patterns brengt de ondersteuning voor pattern matching nog een stapje verder door na het matchen op type, ook het object 'uit te pakken'. Zo hoeven we niet meer afzonderlijke getters te gebruiken om interne velden te benaderen, maar kunnen we dat in een enkel statement. Zo'n deconstruction pattern gebruikt een pattern-definitie, een soort omgekeerde van een constructor, om bij matchen van een object de waardes van

L1

```
// Java 13
@Override
public boolean equals(Object o) {
     // As pattern matching is music to our
ears, we defined a domain model of an orchestra.
More details can be found at our Github
repository[2].
    if (!(o instanceof Musical)) {
        return false;
    }

    Musical m = (Musical) o;
    return name.equals(m.name) && isPrincipal ==
m.isPrincipal;
}

// Java 14
@Override
public boolean equals(Object o) {
    return o instanceof Musical m
                && name.equals(m.name) &&
isPrincipal == m.isPrincipal;
}
```

L2

```
String whatDoesTheMusicianSay =
    switch (musical) {
        case Vocal v  -> String.format("The
singer goes %s", v.sing());
          // The singer goes "Aaaa"
        case Guitar g -> String.format("The
guitar goes %s", g.play());
          // The guitar goes "Pling pling"
        case Drums d  -> String.format("The
drums go %s", d.hit());
        // The drums goes "Pats Boem"
        default       -> String.format("Unknown
Musical goes %s" , obj.play());
};
```

L3

```
return switch (musical) {
    case Orchestra(List<Musical> musicians)
-> String.format("Orchestra, consisting of %d
musicians.", musicians.size());
    // Using definition "public pattern
Orchestra(List<Musical> list)"
    case InstrumentFamily(Guitar(true,
guitarName), Trumpet(true, trumpetName)) ->
String.format("Two principals of guitarist %s
and trumpetist %s.", guitarName, trumpetName);
    // Using definitions "public pattern
InstrumentFamily(Musical m1, Musical m2)" and
"public pattern Musical(boolean isPrincipal,
String name)"
};
```

L4

```
case Guitar(var name) -> String.format("The gui-
tar is called %s", name);
```

L5

```
case Orchestra(_, VocalFamily vf),
Orchestra(VocalFamily vf, _) -> "This orchestra
contains a vocalist.";
```

interne velden toe te kennen aan variabelen. In Listing 3 zien we zo'n deconstruction pattern[4].
Zoals je herkent, matchen we in het voorbeeld op het type `InstrumentFamily` en `Orchestra`. Verder zien we dat bij het case-statement van type `Orchestra` de match alleen plaatsvindt als het genoemde veld uitgepakt kan worden als een lijst van `Musical`-objecten. Bij een match is deze variabele direct in scope en kunnen we er methodes op aanroepen. Met deze deconstruction patterns kunnen we objecten dus in een enkel `case`-statement matchen, casten en velden direct benaderen. Krachtiger dan dit wordt het niet, zou je denken. Behalve dan dat we alle beschreven patterns ook kunnen combineren, zoals staat weergegeven in het tweede case-statement in Listing 3. In dit geval combineren we een deconstruction pattern (het uitpakken van objecten bij een match), type patterns (het matchen op type) en een constant pattern (het matchen op een constante waarde). Samengevat matchen we op een `InstrumentFamily` als deze uitgepakt kan worden in een `Guitar` en `Trumpet`, waar voor beiden geldt dat `isPrincipal` gelijk is aan `true`. Zo staat de gehele collectie aan patterns tot je beschikking!

{ VAR & ANY PATTERNS }

Lokale variabelen kunnen sinds Java 10 ook met het keyword `var` worden aangeduid, in plaats van met een expliciet type[5]. Hetzelfde principe ligt ten grondslag aan var patterns: we kunnen in onze patterns var gebruiken in plaats van het expliciete type, en laten de compiler het juiste type pattern afleiden. Er is daarmee geen conditie op het type meer actief; de compiler gebruikt het eerste veld dat een valide type pattern oplevert en bindt de bijbehorende waarde aan de variabele (zie Listing 4).
Een any pattern (aangegeven met een underscore) is als een

L6

```
class Orchestra {
    @Deserializer
    public static Orchestra
deserialize(Musical[] musicals) {
        return Orchestra.ensemble(musicals);
    }

    @Serializer
    public pattern Orchestra(Musical[] musicals)
{
        musicals = this.musicalPeople.toArray();
    }
}
```

L7

```
record Trumpet(boolean isPrincipal, String name)
implements Musical {
    // This code is generated:
    public pattern Trumpet(boolean isPrincipal,
String name) {
        isPrincipal = this.isPrincipal;
        name = this.name;
    }
}
```

L8

```
sealed interface Musical permits Vocal, Guitar,
Drums {}
```

var pattern: het matcht op het eerstgevonden veld, maar bindt daarbij geen waarde aan de variabele. Dat klinkt misschien niet erg nuttig, maar als onderdeel van een genest pattern kan het elegant uitdrukken dat een deel van een component niet-relevant is en genegeerd kan worden (zie Listing 5).

{ BETERE SERIALISATIE }

We zagen eerder al dat een deconstruction pattern een objectstructuur om kan zetten in losse, getypeerde velden. Eigenlijk net zoals een constructor losse, getypeerde velden omzet in een objectstructuur. Ze zijn elkaars tegenovergestelde. En dat inzicht zou wel eens een betere versie van serialisatie kunnen opleveren. Serialisatie is een belangrijke functionaliteit in Java, maar veel mensen hebben een hekel aan de huidige implementatie. Het ondermijnt bijvoorbeeld het toegangsmodel, de serialisatielogica is geen 'leesbare code' en het omzeilt constructors en bijbehorende datavalidatie. Maar het gebruik van patterns zou de situatie drastisch kunnen verbeteren.

RECORDS & SEALED TYPES

Dat pattern matching niet op zichzelf staat, maar deel uitmaakt van een groter plan wordt nog duidelijker als we naar records gaan kijken. Deze compacte manier om immutable data te modelleren werd eerder al geïntroduceerd in Java Magazine[7] en heeft als voordeel dat constructors, getters, `equals()`- en `hashCode()`-implementaties al beschikbaar zijn, zonder ze expliciet te definiëren. Wanneer deconstruction patterns beschikbaar komen in Java, zullen records ook automatisch pattern-definities bevatten[8], waarmee een record direct bruikbaar wordt als deconstruction pattern in een switch expression (zie Listing 7).
Ook een andere, relatief nieuwe feature zal in de toekomst goed samenwerken met pattern matching, namelijk sealed types. Met sealed types - beschikbaar in preview vanaf Java 15 - leg je voor een interface of superklasse uitputtend vast welke implementaties er mogen zijn (zie Listing 8).

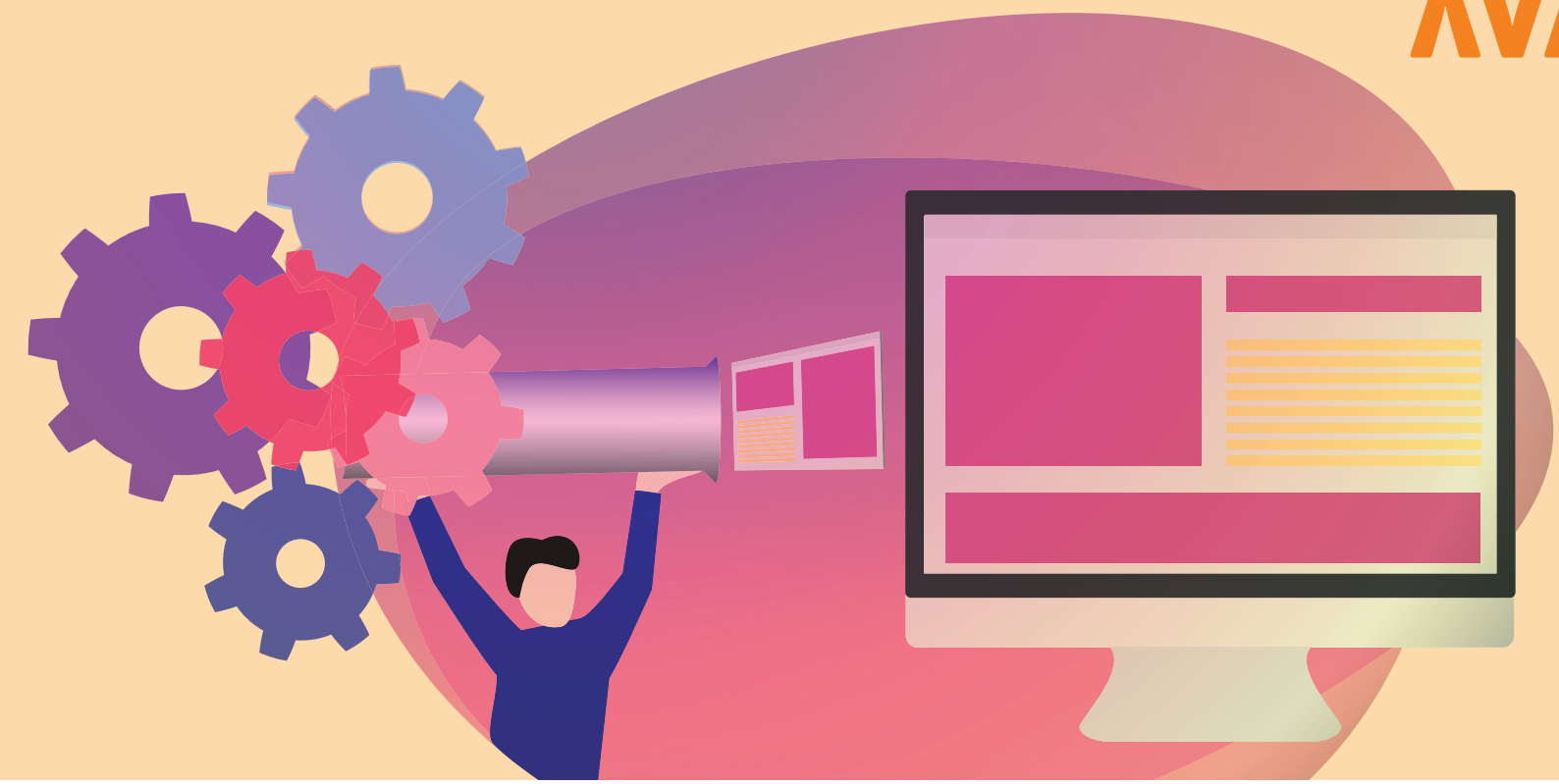
Wanneer klasses in de toekomst pattern-definities kunnen bevatten, zou dat een goede plek zijn om een instantie van die klasse te serialiseren. Deserialiseren gebeurt dan via een (overloaded) constructor of een factory-methode (zie Listing 6). Hiermee zou alle serialisatielogica 'leesbare code' worden en expliciet onderdeel van de klassedefinitie. Door de annotaties is het direct duidelijk waar de code voor dient. Ook zou bestaande datavalidatie eenvoudig aangeroepen kunnen worden.
Het ondersteunen van serialisatie op meerdere versies van een klasse zal een uitdaging blijven, al bestaan er wel plannen om de `@Serializer`- en `@Deserializer`-annotaties met een `version`-property uit te rusten die een klasseversie kan bevatten[6].

{ CONCLUSIE }

Pattern matching gaat een heel stuk verder dan alleen het voorkomen van casts bij een `instanceof`. Het verbetert switch expressions, het kan complexe logica elegant uitdrukken en het deconstrueren van objecten is een 'first-class citizen' geworden, doordat pattern-definities het tegenovergestelde zijn van constructors. Bovendien is bij het ontwerpen van nieuwe features als sealed types en records de ondersteuning van pattern matching alvast voorzien. Dat laat zien dat pattern matching een belangrijke feature in Java gaat zijn én blijven.

REFERENTIES

De voetnoten in dit artikel verwijzen naar referenties die opgenomen zijn in onze GitHub-repository: https://github.com/MrFix93/pattern-matching-orchestra



TEKTON
Cloud Native CI/CD

Jenkins is al jaren een gevestigde optie als het gaat om CI/CD. Binnen het Java ecosysteem is het ongeveer de standaard keuze. Door de beweging van de laatste jaren naar containers, cloud, cloud native en meer zijn ook andere opties interessant geworden. Dit artikel gaat in op een van die opties: Tekton [1]. Omdat velen van ons Jenkins wel zullen kennen gaan we af en toe de vergelijking aan.

Mark van der Walle *is Lead Software Architect bij Sogeti. Daarnaast geeft hij leiding aan het Java Core team van Sogeti en is hij Fellow van SogetiLabs.*

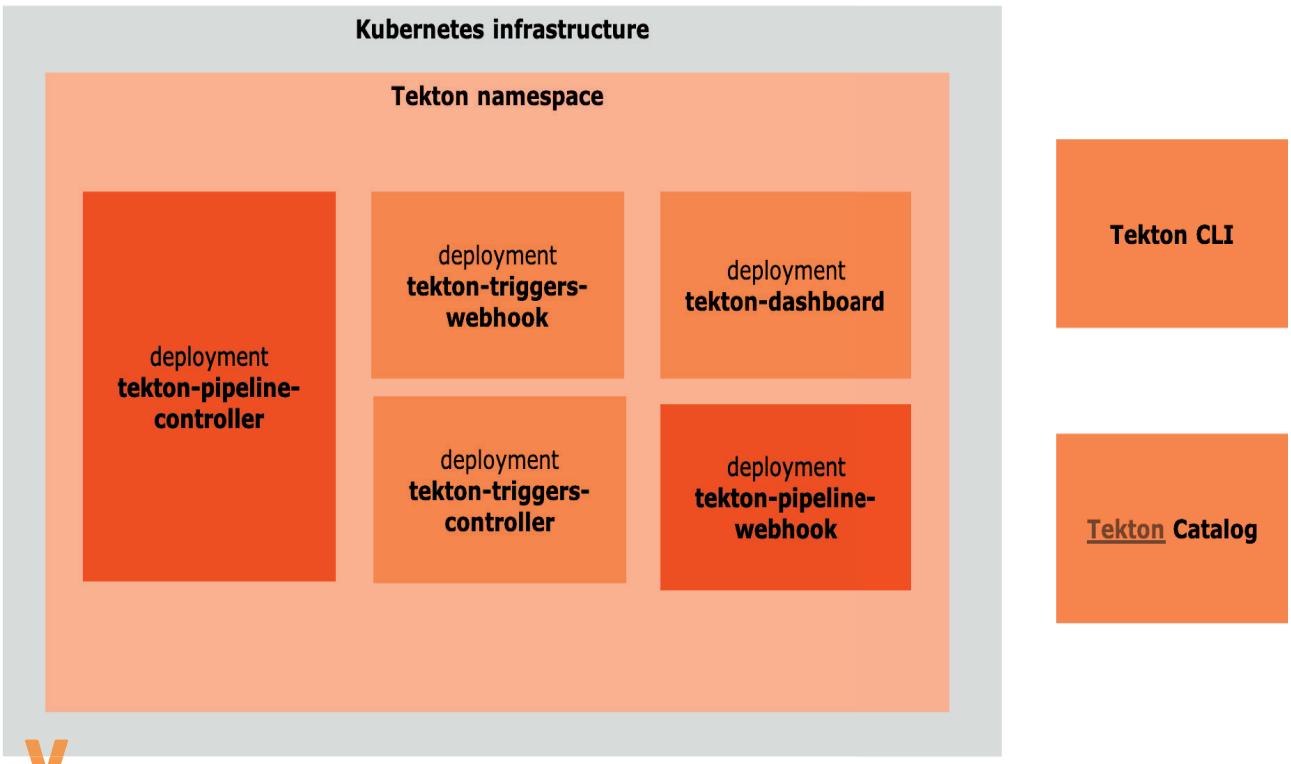
De naam Tekton komt vanuit het Grieks en is zowel een wat algemenere term voor een vakman in de vorm van timmerman of bouwer. Dit is ook precies wat Tekton doet in het kader van CI/CD: het bouwen van in dit geval Cloud Native software.

Tekton draait in een Kubernetes [2] cluster en alles wat het doet draait in een container. Hierdoor maakt Tekton veel gebruik van de mogelijkheden die Kubernetes biedt in het kader van schaalbaarheid en infrastructure as code. Daarom zullen we ook heel beperkt ingaan op een aantal concepten die nodig zijn om Tekton te kunnen begrijpen.

{ KUBERNETES OBJECTEN }

Een kubernetes object wordt meestal beschreven door middel van **YAML** en bestaat uit een viertal verplichte hoofdonderdelen.

- apiVersion - De versie van de objectdefinitie. Bijvoorbeeld **v1**, of **tekton.dev/v1beta1**.
- kind - Het type van de objectdefinitie. Bijvoorbeeld **Pod**, **Deployment**, of **Pipeline**.
- metadata - Metadata kan meerdere velden bevatten. Hierin moet minimaal het veld **name** opgegeven worden. **Name** is uniek binnen een Kubernetes namespace.
- spec



Afbeelding 1 - Een overzicht van de Tekton componenten.

Kubernetes kent een hoop ingebouwde objecten die volgens deze definitie worden gemaakt. Daarnaast is het mogelijk om nieuwe objecten te definiëren in de vorm van **CustomResourceDefinitions**. Een dergelijke definitie volgt dezelfde vorm als reguliere objecten waarbij het schema van de definitie wordt vastgelegd in "spec".

{ TEKTON OBJECTEN }

Tekton objecten zijn, zoals hierboven beschreven, niet veel anders dan Kubernetes objecten met een eigen schema. Er zijn diverse objecten die gebruikt kunnen worden om CI/CD in Kubernetes te doen. In dit artikel worden de belangrijkste beschreven.

Task - Een Tekton task beschrijft een of meerdere steps. Een step is niet veel meer dan een container image met input en output. Een task is te vergelijken met een Jenkins stage.
TaskRun - Een enkele instantie van een Task. Met Tekton is het mogelijk zowel alleen een Task als meerdere Tasks te combineren. Een TaskRun beschrijft parameters zodat de taak die aangeroepen wordt flexibel in te zetten is.
Pipeline - Een pipeline combineert meerdere Tasks waarbij de pipeline de volgorde, condities, parallelliteit en input en output van Tasks beheert. Een pipeline is heel goed te vergelijken met Jenkins pipeline.
PipelineRun - Een enkele instantie van een pipeline. Je kunt dit vergelijken met een Jenkins pipeline waarin bijna alles een parameter heeft. De PipelineRun combineert de parameters en andere componenten die de pipeline nodig heeft. Denk hierbij aan git repository, volumes en service accounts.

L1

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: hello
spec:
  steps:
    - name: hello
      image: ubuntu
      command:
        - echo
      args:
        - "Hello World!"
```

Met deze vier objecten kunnen al complexe pipelines gemaakt worden. Tekton heeft nog meer objecten zoals Triggers en PipelineResources.

Een van sterke punten van Tekton is het hergebruik van taken. Dit is zo'n belangrijk onderdeel dat men een hub [3] heeft waar taken voor allerlei doeleinden gevonden kunnen worden. Er zijn bijvoor-

beeld taken voor maven, maar ook voor de aws cli of kubectl en meer.

{ TEKTON OVERZICHT }

Een plaatje zegt meer dan duizend woorden, daarom is in afbeelding 1 een overzicht te zien van de componenten van Tekton.

Als je Tekton installeert op Kubernetes dan installeer je, naast de **Custom Resource Definitions** een aantal dingen als Kubernetes deployment. Dit is een gebruikelijke manier om Kubernetes uit te breiden. De **tekton-pipeline-controller** en **tekton-pipeline-webhook** zorgen voor het draaien van taken en pipelines als er task runs en pipeline runs worden aangemaakt. Deze twee componenten zijn ook minimaal vereist om Tekton goed te kunnen draaien.

Binnen de namespace zijn er nog een aantal optionele componenten. Om van buiten het cluster pipelines te kunnen triggeren door middel van events zijn de twee trigger deployments nodig. Denk hierbij aan het starten van een pipeline door een github push event.

Een tweede optionele, maar zeer nuttige toevoeging, is het Tekton Dashboard. Het dashboard geeft je een makkelijker overzicht over de beschikbare taken en pipelines, maar ook de status van een draaiende PipelineRun en TaskRun. Dit is vergelijkbaar met hoe in Jenkins van een pipeline de staat en logs ingezien kunnen worden.

Naast de onderdelen die in het cluster staan, zijn er nog twee onderdelen die er buiten staan. Als eerste is er de catalog of hub die al eerder genoemd is. De tweede is de Tekton CLI. De CLI kan gebruikt worden om iets makkelijker met de Tekton resources te praten. Dit is niet heel veel meer dan een schil om kubectl heen.

{ EEN VOORBEELD }

Zoals bijna elk programmeervoorbeeld begint met “Hello World” zullen we dat hier ook gaan doen.

Na het installeren van Tekton op een Kubernetes cluster kunnen we een taak aanmaken. Zie L1 voor het “Hello World”-voorbeeld.

Door dit object aan te maken met behulp van kubectl apply wordt de taak voor iedereen beschikbaar. Om deze taak nu uit te voeren kunnen we hem toevoegen aan een pipeline, maar we kunnen ook een TaskRun maken. Task runs zijn ideaal voor simpele dingen. In L2 is een bijpassende TaskRun beschreven.

Door deze TaskRun aan te maken met kubectl create wordt er elke keer een nieuwe instantie van de taak gedraaid. Het opvragen van het resultaat kan op meerdere manieren, maar het makkelijkste is de Tekton CLI te gebruiken:

L2

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: hello-run-
spec:
  taskRef:
    name: hello
```

```
tkn taskrun logs --last -f
```

Dit zal de logs van de laatste TaskRun volgen.

{ PIPELINES }

Het draaien van een Task is al hele krachtig. Echter komt de kracht van Tekton pas boven bij het combineren van meerdere herbruikbare taken in een pipeline. Hierbij kan de pipeline zorgen dat de output van de ene taak gebruikt kan worden in de volgende taak.

Ook kan de pipeline zorgen voor het koppelen van volumes waarop de taken hun werk kunnen neerzetten. Denk hierbij aan de vergelijking met een Jenkins workspace. Tekton noemt deze ook hetzelfde. Het is mogelijk dat alle taken gebruik maken van de dezelfde workspace of allemaal een verschillende workspace te laten gebruiken.

Doordat workspaces leunen op de persistent volumes van Kubernetes hebben ze ook dezelfde voor en nadelen. Een voorbeeld Pipeline en PipelineRun gaan we in dit artikel niet tonen. De listing hiervan wordt te groot. Op de Tekton website zijn een aantal goede voorbeelden te vinden.

{ SCHAALBAARHEID }

Naast herbruikbaarheid van taken en pipelines is een van de grote krachten van Tekton iets dat het van Kubernetes erft. Tekton schaalt net zover als er ruimte is op je cluster. Daarnaast gebruikt het nauwelijks resources als er geen pipelines draaien.

Het kan bijvoorbeeld zo zijn dat de eerste task op een worker node draait, terwijl de volgende op een andere node draait. Ook taken die in parallel draaien kunnen op verschillende nodes actief zijn.

{ CONCLUSIE }

Tekton is nog behoorlijk nieuwe technologie en het maken van pipelines kost nog wel wat moeite. Ook het debuggen van fouten is nog niet erg eenvoudig. Echter op het moment dat er gebruik wordt gemaakt van de catalogus voor taken en pipelines kunnen er al snel krachtige en schaalbare pipelines draaien. Dit artikel is in ieder geval een tipje van de sluier. Het Tekton eco-systeem bevat nog een aantal interessante onderdelen zoals Events en Metrics.

AUDITING OP ENTITEITEN met Envers

In dit artikel lees je meer over hoe je versiebeheer kunt doen op entiteiten. Het bijhouden van data wijzigen in de breedste zin van het woord. Dit is een van de requirements die een project binnen sluipt met een simpele requirement, zoals de historie moet bewaard worden en later steeds meer uitgebreid moet worden tot aan auditing toe. Als je dit zelf ontwikkelt, kom je erachter dat je onder andere ook iets moet doen met relaties van de entiteit waar je versiebeheer op toepast. Voor je het weet heb je veel code geschreven voor een suboptimale oplossing. Na een introductie over de mogelijkheden en beperkingen ga ik aan de hand van een case de inrichting en gebruik uiteenzetten.

In dit artikel maken we gebruik van een library waar je eenvoudig versiebeheer kan toepassen op een Spring project waar je gebruikmaakt van JPA en Spring Data. Het is belangrijk om te weten dat je enkel gebruikmaakt van JPA op de entiteiten waar je het versiebeheer op wilt doen, want deze library zal geen wijzigingen kunnen bijhouden waarbij andere mechanismen zijn gebruikt om entiteiten te persisteren zoals JDBC. Mocht je dit wel willen, of versiebeheer willen kunnen uitvoeren op NOSQL databases, raad ik je aan te kijken naar een andere library genaamd Javers. Het versiebeheer en auditing wordt alleen toegepast op de entiteiten die je expliciet annoteert.

{ ENVERS }

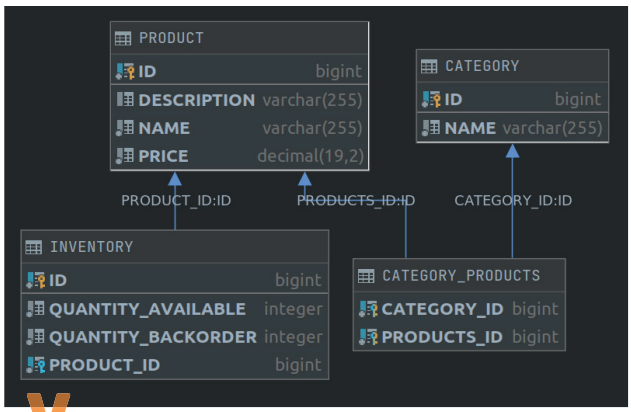
Envers is een library die je met hibernate gebruikt. Envers maakt gebruik van het JPA event systeem en vangt events van wijzigingen op en past dan het versiebeheer toe door historie op te slaan in historie tabellen. Dit kun je instellen op de tabellen waar je versiebeheer op wilt toepassen en kun je zelfs op veldniveau instellen.

Ben Ooms *is een gepassioneerde Java engineer, Spring fan en architect bij de Kamer van koophandel. Ben doet graag kennis op en deelt deze graag door coaching, schrijven en presentaties geven.*

Voor het Auditing deel maken we gebruik van functionaliteit van Spring Data. Deze biedt annotaties en een interface aan die je moet implementeren om de auditing velden te vullen wanneer deze opgeslagen worden.

Om Envers te kunnen gebruiken op een Spring Boot Data project, heb je maar een dependency nodig en dat is `org.springframework.data:spring-data-envers`. Deze implementatie biedt een aantal geïmplementeerde Envers interfaces aan met een Factory bean om Envers specifieke read-only Jpa Repositories aan te maken. Tip: als je geen gebruikmaakt van Spring Data, kun je in de sources van dit project inspiratie op doen om Envers in een andere JPA omgeving te gebruiken.

Na deze dependencies in je project opgenomen te hebben, kun je aan de slag om versiebeheer en auditing toe te passen op entiteiten. Voordat ik dat doe wil ik eerst de case toelichten die als voorbeeld zal dienen. De applicaties gebruikt een aantal Spring Boot starters (web, validation, actuator, data en test) en twee losse dependencies (H2 voor persistentie en Lombok voor gemak ;>). Dit zijn gebruikelijke dependencies voor een Crud gebaseerde Rest service voor een relationeel model obv Spring Boot. De case is een heel simpel domeinmodel dat bestaat uit drie entiteiten bestaande uit Inventory, Product en Category. De code voor de basis is getagged als “Spring-rest-no-versioning”. Op afbeelding 1 zie je een ERD van de database die JPA heeft gegenereerd op basis van deze drie entiteiten.



Afbeelding 1.

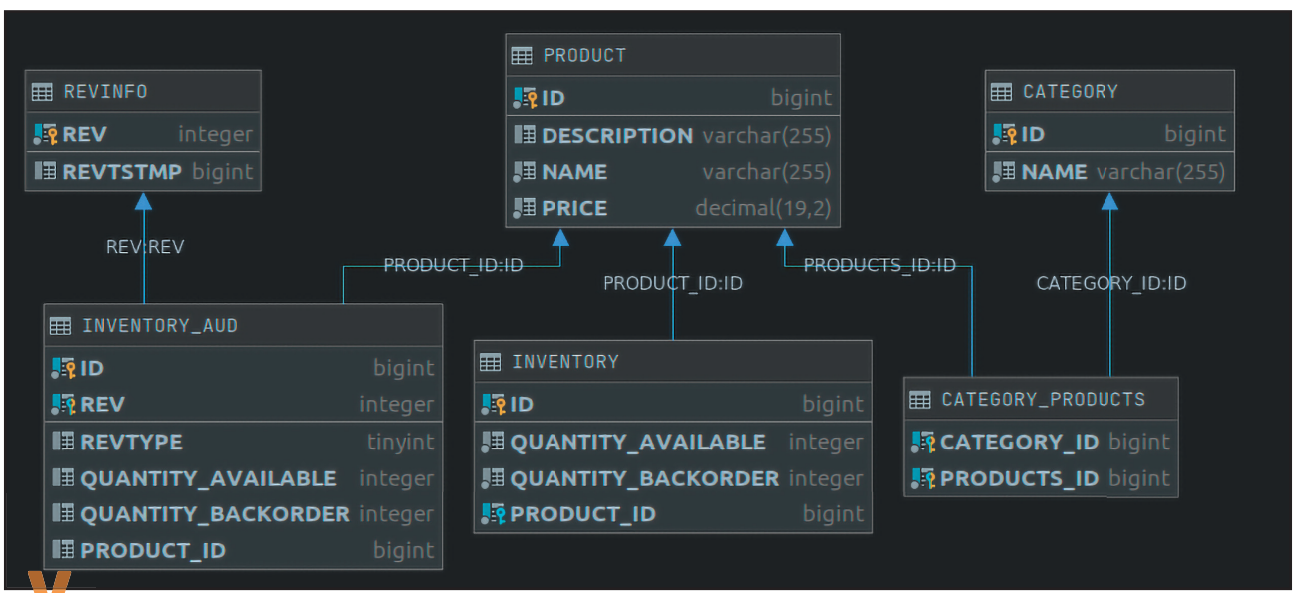
De volgende requirements gaan wij implementeren:
- Volledige versiebeheer op de inventory entiteit op alle velden
- Versiebeheer op de productvelden name en price
- Audit informatie op producten en categories

Daarnaast willen wij weten wie een inventory item heeft toegevoegd en aangepast, inclusief de wijzigingen en de categorie wie de laatste wijziging heeft doorgevoerd.

{ VERSIEBEHEER }

We beginnen met het versiebeheer. Als eerste moeten weeen dependency toevoegen, dit is `org.springframework.data:spring-data-envers:2.4.2`. Met deze dependency halen wij zowel Envers als een aantal Spring klassen binnen voor versiebeheer.
Ik gebruik de schemageneratie van JPA, dit betekent dat tijdens het genereren van het model op basis van de Envers annotaties ook de historie tabellen gegenereerd worden. Ik zal hierbij alle defaults gebruiken waarbij in het laatste deel van het artikel interessante configuratie opties zal bespreken.

Om versiebeheer toe te passen, voeg je de @Audited annotatie toe aan de klasse of aan velden van een klasse. Om dit toe te passen in onze applicatie, voegen wij de annotatie toe op de Inventory entiteit op klasse niveau. Dit betekent dat alle velden van deze klasse onder versiebeheer staan en in de historie opgenomen zullen worden. Als je dit toepast en de Testen uitvoert zal je zien dat deze falen op het feit dat de applicatie context niet gestart kan worden. Dit komt doordat de Inventory entiteit een relatie met product heeft en Product geen versiebeheer heeft. Dit kun je op twee verschillende manieren oplossen afhankelijk van de requirements. Mocht je geen versiebeheer willen op de entiteit waar een relatie mee is, dan kun je dit op veldniveau aangeven door een @Audited annotatie te plaatsen en een parameter targetAuditMode met als waarde NOT_AUDITED toe te voegen. In dit stadium van ontwikkeling zetten wij een @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED) toe. Goed om te weten is dat dit alleen een aanduiding is dat Product niet onder versiebeheer valt, er is nog degelijk wel versiebeheer op het veld productId.
In deze versie van de applicatie is het gegenereerde data model uitgebreid met een nieuwe tabel genaamd REVINFO en INVENTORY_AUD. Op afbeelding 2 zie je een schema met de nieuwe tabellen. De REVINFO tabel wordt door Envers gebruikt om en de INVENTORY_AUD voor het versiebeheer van de Inventory entiteit. Naast de velden van de Inventory entiteit worden ook andere velden toegevoegd, dit zijn standaard velden die Envers toevoegt. De REV geeft de revisienummer terug waarbij de REVTYPE het type mutatie beschrijft, dit kan zijn Insert, Update of Delete. De waarden die gebruikt worden zijn beschreven in de RevisionType enumeratie van de Envers library. Daarnaast vind je in de InventoryHistoryTests ook de verificatie dat er versiebeheer wordt



Afbeelding 2.

toegepast en dat ondanks het Product geannoteerd is met @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED) het wijzigen van een product in de inventory terug is te vinden in de historie. De code tot dit punt kun je vinden door de tag “versioning-done-for-inventory” uit te checken.

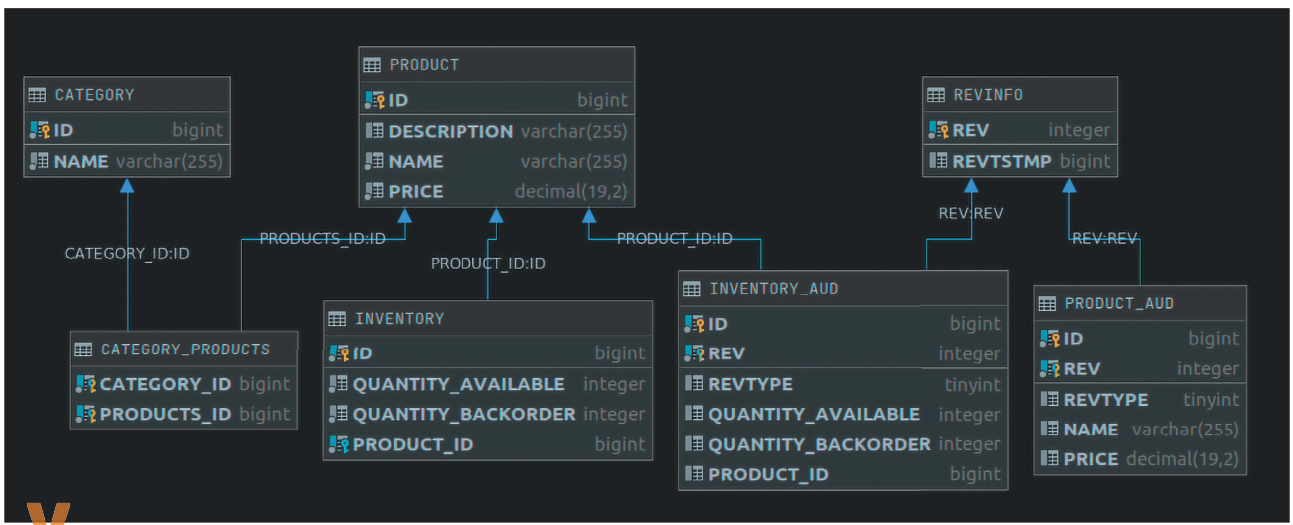
Om onze versiebeheer requirements te voltooien, moet er nog versiebeheer op de Product entiteit worden toegepast. Om dit te doen gaan wij voor de volledigheid gebruik maken van het annoteren op veldniveau. Wij zetten alleen de annotaties op de name en price velden. Daarnaast kunnen wij de annotatie op het veld product van de Inventory entity verwijderen want er is nu versiebeheer op producten, tenminste dat is wat je zou denken. Als je dit doet dan zie je dat de eerder geschreven testen zullen falen ook al laat je de annotatie staan. De reden hiervoor is dat er wel versiebeheer is op Product maar alleen op de velden name en price. Wanneer de revisie informatie opgehaald wordt voor de Inventory entity, wordt ook de bijhorende Product revisie-historie gebruikt. Omdat er geen versiebeheer wordt toegepast op de description veld is deze null en falen testen op de conditie dat het product equals het gebruikte product. Hiervoor moeten de bestaande testen van de Inventory entity aangepast worden. Na dit gedaan te hebben zien wij het uiteindelijke model zoals getoond op afbeelding 3. Daar zie je de nieuwe historie tabel voor producten met alleen de product entiteit velden name en price. Na het toevoegen en aanpassen van een aantal testen zijn de versiebeheer requirements geïmplementeerd. De code tot dit punt is in Git getagged met “versioning-done”. Deze tag zal de basis vormen voor het laatste onderdeel, auditing.

{ CONFIGURATIEMOGELIJKHEDEN VAN ENVERS }

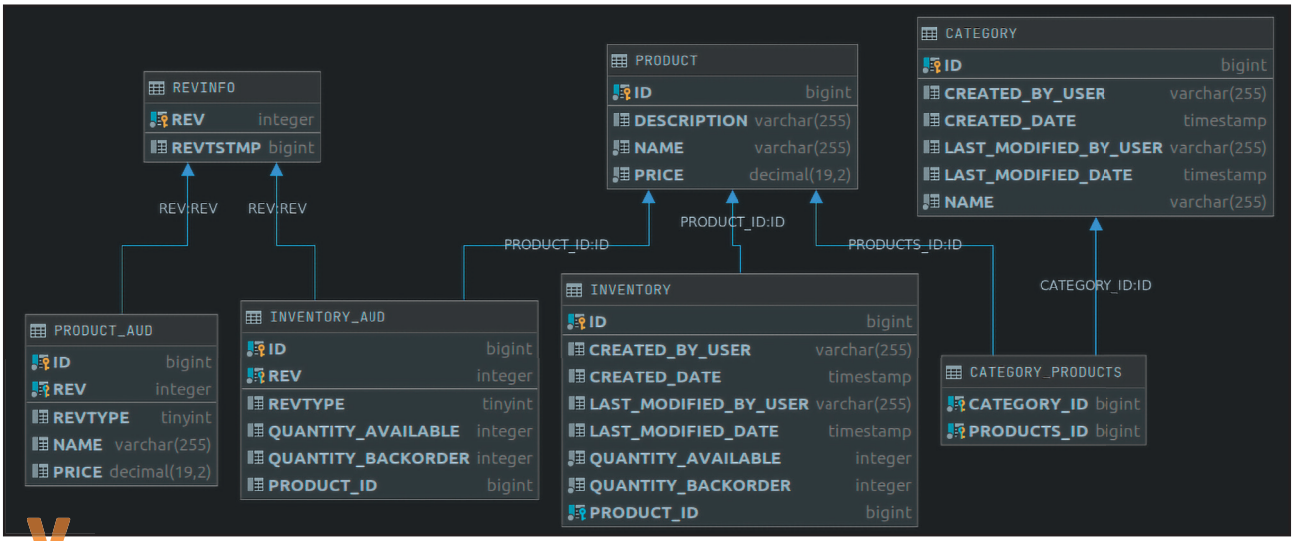
Voordat wij verder gaan met Spring auditing is het goed om stil te staan bij de verschillende configuratiemogelijkheden van Envers. Zonder specifieke configuratie op te geven kun je snel aan de slag door het versiebeheer te activeren met een annotatie op applicatieniveau en grof gezegd te strooien met @Audited annotaties op je entities. Je kunt veel onderdelen van Envers configureren zoals bijvoorbeeld de suffix voor de historie tabellen (default _AUD) en kolomnamen voor revisietype en revisienummer.
Daarnaast zijn er nog twee interessante configuratie opties die je kunt gebruiken. De eerste is de optie om wijzigingen op veldniveau bij te houden. Use case hiervoor is wanneer je ook informatie wilt opslaan over welke velden aangepast zijn. Elk veld in versiebeheer krijgt er met deze optie een boolean kolom bij om aan te geven of een veld aangepast is in de bewuste kolom. Als je deze optie globaal wilt gebruiken kun je in jouw application.properties de property `org.hibernate.envers.global_with_modified_flag` met de waarde true opnemen en op Klasse of veld niveau de property “withModifiedFlag” toe te voegen aan de @Audited annotatie.

Een andere interessante optie is de gebruikte strategie voor het opslaan van de historie.
Met de standaard strategie wordt er bij iedere wijziging een nieuwe record toegevoegd aan de historie. Dit heeft als voordeel dat de update operatie snel is, maar heeft wel als nadeel dat het ophalen van historie subqueries moeten worden uitgevoerd die impact hebben op de performance. Als je de validityAuditStrategie gebruikt wordt tijdens de insert van een nieuwe versie ook alle bestaande historie records bijgewerkt met een referentie naar de laatste records. Dit heeft als voordeel dat het ophalen van de historie geen subqueries behoeft en daardoor sneller is, nadeel is dan dat het updaten van de historie meer tijd neemt. Afhankelijk van de use case kun je voor een strategie kiezen, Haal je vaak de historie op, bijvoorbeeld om standaard historie te tonen, kun je beter de validityAuditStrategie gebruiken. Doe je dit op afroep is de standaard oplossing afdoende.

Goed om te weten is dat je Envers vergaand kunt aanpassen aan jouw wensen. In dit artikel raak ik maar een paar opties aan. Ook is



Afbeelding 3.



Afbeelding 4.

het goed om vooraf goed na te denken over wat je precies wilt met de historie en te kijken naar de volatiliteit van de entities waar je versiebeheer op wilt toepassen want de historie tabellen kunnen zeer groot worden. Daarnaast wil je ook indexen toevoegen op relevante kolommen om de performance goed te houden.

{ AUDIT INFORMATIE TOEVOEGEN }

Wij zijn nu toe aan de laatste requirement, het toevoegen van audit informatie op Producten en Categorieën. Hiervoor moeten wij als eerste auditing activeren, dit doe je door @EnableJpaAuditing op een Configuratie klasse toe te voegen zoals de Bootstrap klasse die geannoteerd is met @SpringBootApplication. Hierna hebben wij een bean nodig die Spring zal aanroepen wanneer de waarden geïnjecteerd worden voor het opslaan. Om deze te implementeren maak je een Bean aan die AuditorAware implementeert. Hiervoor moet je de methode getCurrentAuditor implementeren die de waarde levert onder welke naam de wijziging is aangebracht. Voor de hand liggend gebruik je de user gegevens die je uit de SecurityContext kunt verkrijgen en bij systeem bewerkingen zoals bijvoorbeeld een batch kunt je de gegevens voor auditing gewoon zetten. In onze voorbeeld applicatie geeft deze methode iedere keer de String “Applicatie” terug. Naast deze bean heb je enkel nog annotaties toe te voegen. Spring biedt vier annotaties hiervoor, dit zijn @CreatedBy, @LastModifiedBy, @CreatedDate en @LastModifiedDate. Om onze laatste requirement te implementeren voegen wij deze annotaties toe aan de Product en Category entiteiten. Om het DRY te houden extenden deze klassen nu de AbstractAuditedEntity klasse die de auditing annotaties bevat. Als laatste moeten wij nog een Entity listener opgeven die de velden voor ons gaat vullen, Dit doen wij op klasse niveau door de annotatie `@EntityListeners(AuditingEntityListener.class)` toe te voegen. De meegegeven AuditingEntityListener is een standaard listener uit Spring Data die de auditing velden zal vullen. Op afbeelding 4 zie je het uiteindelijke model waarbij zowel het versiebeheer en de auditing geïmplementeerd is.

Ook aan de auditing kun je veel configureren waarbij wij in de voorbeeld applicatie al een bean hebben geconfigureerd die (vrij nutteloos in de praktijk) altijd “Gebruiker” retourneert. Naast deze provider kun je ook eigen listeners implementeren. Wanneer je zowel de creatie als wijzigingen audit wordt standaard bij creatie ook de wijziging velden gevuld, dit kun je door middel van een property aanpassen.

In dit artikel hebben wij door middel van een voorbeeldapplicatie de werking van versiebeheer met Envers en Auditing met Spring Data uiteen gezet. Met de behandelde onderdelen en opties kun je een goede basis opzetten voor versiebeheer en auditing. Zowel Envers en Spring Data bieden veel aanpasmogelijkheden die niet behandeld zijn in dit artikel. Neem vooral een kijkje in de Github repository, alle behandelde onderwerpen zijn uitgewerkt en er zijn testen die de werking bewijzen. Ook kun je dit project gebruiken om verder te spelen met deze libraries. De broncode is te vinden op **https://github.com/bendh/Auditing-and-versioning-article**

REFERENTIES

https://sunitc.dev/2020/01/21/spring-boot-how-to-add-jpa-hibernate-envers-auditing/
https://hibernate.org/orm/envers/
https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#envers
https://springbootdev.com/2018/03/13/spring-data-jpa-auditing-with-createdby-createddate-lastmodifiedby-and-lastmodifieddate/

FIRST8 | CONCLUSION

MASTERS OF JAVA 2021

het officieuze NK Java Programmeren

De Masters of Java is een roemruchte “funprogging contest” (gebaseerd op Java SE), die toegankelijk is voor iedere Java ontwikkelaar. In deze wedstrijd wordt niet de API-kennis, maar de echte programmeervaardigheid getest. In voorgaande edities heeft Masters of Java al heel wat verhitte strijd opgeleverd!

In de verschillende rondes die de wedstrijd telt, wordt iedere keer een grappige of interessante programmeeropdracht uitgedeeld, die vervolgens binnen een half uur opgelost moet worden. Voor iedere seconde die je overhoudt, scoor je een punt. De afgelopen jaren is de beslissing steeds gevallen tijdens de laatste opdracht. Het belooft dus een zinderende wedstrijd te worden!

Teams bestaan maximaal uit 2 ontwikkelaars en het team dat aan het einde van de dag de meeste punten heeft, mag zich “Master of Java 2021” noemen. Er zal naast eeuwige roem natuurlijk gestreden worden voor mooie en spectaculaire prijzen!

Schrijf je nu in voor de Masters of Java 2021

EEN NLJUG EVENT

.nl. jug

2021 MASTERS OF JAVA nl. jug

Wanneer: woensdag 3 november
Tijd: 13.00u tot uiterlijk 17.30u
Waar: Van der Valk Veenendaal / afhankelijk van de dan geldende Corona maatregelen (nljug.org/masters-of-java-2021)
Kosten: Gratis

Ook dit jaar is First8 hoofdsponsor en maken zij weer zes uitdagende opdrachten voor jullie. Met veel enthousiasme, bevlogenheid en passie werken zij aan deze en andere gave opdrachten. Iets voor jou? We maken graag kennis met je.

www.first8.nl

In verband met de huidige Corona-maatregelen kan het zijn dat Masters of Java 2021 een iets andere opzet krijgt, bijvoorbeeld een online of een hybride variant. Dit is nu nog niet bekend. Uiteraard zal het hele event gefinetuned zijn voor de uiteindelijke vorm. Kortom: je wilt het niet missen. Zoals je van ons gewend bent, zijn de opdrachten uitdagend en kunnen er veel teams meedoen.

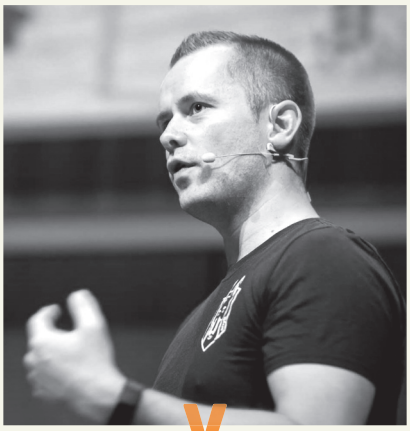
Aanmelden: nljug.org/masters-of-java-2021

VAN HET BESTUUR

We hebben helaas afscheid moeten nemen van Rob Brinkman als NLJUG bestuurslid, maar niet te hard getreurd, want we mogen niet één, niet twee, maar drie nieuwe bestuursleden verwelkomen! Hieronder zullen ze zich even kort voorstellen aan jullie.

{ NOORTJE VAN BERLO }

Noortje is sinds 2021 actief in het bestuur van de NLJUG. Ze draagt de developer community een warm hart toe en wil zich met de NLJUG inzetten om deze community verder te verstevigen. Op kleinere schaal zet(te) ze zich hier dagelijks ook voor in vanuit haar rol als manager van de (Java-) development community bij Alliander. Noortje heeft een passie voor technologie en innovatie en vindt het belangrijk om maatschappelijk betrokken te zijn. Binnen Alliander en de NLJUG kan zij haar ei helemaal kwijt. Vanuit de NLJUG zal zij zich gaan richten op de zichtbaarheid van de community en de businesspartners en op het thema diversiteit. Daarnaast is zij actief in de organisatie van de jaarlijkse Masters of Java.



{ BRIAN VERMEER }

Brian zit sinds 2021 in het bestuur van de NLJUG. Momenteel werkt hij als Developer Advocate voor het security bedrijf Snyk. Verder is hij een Java Champion en erg actief in verschillende communities die betrekking hebben op Java en Security. Binnen de NLJUG zal Brian zich met name bezig houden met de vakinhoudelijke zaken, zoals het programma voor J-Spring en J-Fall, Java Magazine en het verder delen van vakinhoudelijke kennis met Java Developers.

{ BAS KNOPPER}

Bas is vanuit zijn passie voor Java (Development) vanaf begin 2019 betrokken bij de NLJUG als kartrekker van de NLJUG Academy en is sinds 2021 onderdeel van het bestuur. In het dagelijks leven is Bas CEO van Sparkler: een ecosysteem van IT-bedrijven waar vakmanschap & kennisdeling centraal staan en waarvan meerdere bedrijven actief zijn binnen het prachtige Java Ecosysteem.

Binnen het bestuur houdt Bas zich met name bezig met de NLJUG Academy. Een initiatief waarmee de NLJUG het onderwijs (studenten/developers, docenten en onderwijsinstellingen), bedrijven en developers (nog) actiever met elkaar verbindt!

V

COLUMN

Meer met Maven
Apache Maven 3.8.1

*In elke editie zal **Robert Scholte** inzichten geven over Apache Maven en hieraan gerelateerde vraagstukken.*

Sinds de release van Maven 3.6.3 wordt er druk gewerkt aan de nieuwe release. In eerste instantie leek dit versie 3.7.0 te worden, maar gezien de impact van de wijzigingen is op den duur besloten om dit te hernoemen naar Maven 4. Echter werden we eind december gewezen een potentiele vulnerabilities in Maven, waardoor we mogelijk toch nog een Maven 3 versie zouden moeten releasen. Deze kwetsbaarheden waren allemaal gerelateerd aan repositories die in een pom.xml toegevoegd kunnen worden, en vooral als deze gebruik maken van HTTP in plaats van HTTPS. Daarnaast ontstond een discussie over welke repositories allemaal gebruikt worden in welke volgorde ze worden bevraagd voor het downloaden van dependencies. Dit laatste bleek na lang overleg toch geen probleem: Maven werkt conform design, maar de repository context is net iets groter dan je in eerste instantie zou verwachten. Verder bleek dat er in sommige repositories verwezen wordt naar domeinnamen die niet meer in gebruik zijn. Technisch gezien kunnen we hier weinig aan doen, maar om domain-hijacking te voorkomen, heeft Sonatype deze domeinnamen opgekocht.

Wat we wel in anno 2021 als probleem zien, is dat dependencies via een insecure protocol opgehaald kunnen worden, zonder dat je je als gebruiker daar bewust van bent. Om voortaan de gebruiker 'by default' hiertegen te beschermen, hebben we een kleine nieuwe feature toegevoegd. Eén van de onderdelen die je via de settings.xml kunt configureren zijn zogenoemde mirrors. Met mirrors is het mogelijk om repositories in poms door te routeren naar een andere URL. Het meest bekende gebruik van een mirror is waarbij alles doorgerouteerd wordt naar een repository manager binnen een bedrijf of project. Sinds Maven 3.8.1 heeft de mirror een extra veld gekregen: blocked. Hiermee kun je specifieke URLs blokkeren. Daarnaast is de expressie voor het veld mirrorOf uitgebreid. Het is nu mogelijk om met één expressie alle external repositories over HTTP te matchen.

HTTP

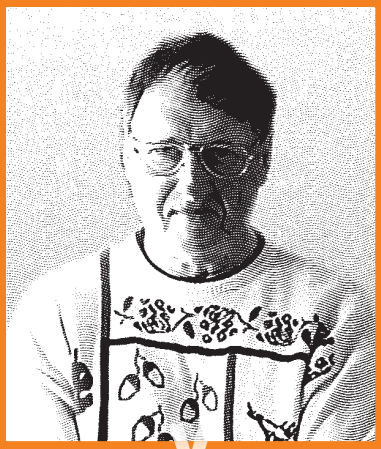
Zoals al eerder genoemd kan het dus zijn dat builds ongemerkt gebruikmaken van een repository over HTTP. Met de nieuwste Maven release zullen deze builds gaan falen en dit is meteen de reden waarom we niet voor Maven versie 3.6.4 zijn gegaan. Uit ons oogpunt zou een bugfix release niet zo'n grote impact mogen hebben.

Dan zou je verwachten dat de versie 3.7.0 genoemd zou moeten worden, maar daar is in het verleden al een aantal keren naar verwezen, inclusief door mijzelf. De features die daarin beloofd werden, zitten echter niet in deze release, dus om maar de mogelijke verwarring te voorkomen hebben we ingezet op 3.8.0.

Echter, tijdens de voting-periode voor 3.8.0 werd ontdekt dat de fix nog niet compleet was (deze credits gaan naar Maarten Mulders!). Om verwarring te voorkomen tussen verschillende 3.8.0 versies, hebben we binnen het Maven team het principe van versies verbranden: we tellen gewoon door. En vandaar dat we uitgekomen zijn op Apache Maven versie 3.8.1.

COLUMN

JOOP
Voorspellingen



Joop Lanting *is vaste columnist voor Java Magazine*

Stel dat we terug konden gaan in de tijd. We begeven ons naar het Vlissingen van 1667 en lopen naar het afgemeerde vlaggenschip van de vloot der Republiek: De Zeven Provinciën: 46 meter lang, 12 meter breed, 1600 ton waterverplaatsing, 2000 m2 zeiloppervlak en 80 kanonnen. We krijgen een onderhoud met de gezagvoerder, Michiel de Ruyter. *"Dit is een prachtig schip, admiraal. In 2002 zal het vlaggenschip van de Nederlandse vloot net zo heten. Wat denkt u, hoe zal het er uit zien?"* 'Bestevaer', zijn bijnaam, is enigszins in verwarring en komt niet verder dan 'iets van 200 stuks geschut'. Onze onthulling bevalt hem helemaal niet: **een ijzeren schip van 144 bij 17 meter en 6000 ton waterverplaatsing.** Maar dan: **één mast, géén zeilen één kanon van 12.7 cm, één robot kanon achterop, torpedo's, en èh … grote vuurpijlen (raketten, dus)**. We worden van boord gejaagd en zoeken een goed heenkomen en onze eigen tijd weer op.#$Tja, in meer dan 300 jaar verandert er nog wel eens wat. Maar Michiel kon zich niet verplaatsen in de situatie van de 21e eeuw en wilde dat ook niet kunnen. Ieder van ons zit net zo goed 'vastgebakken' aan tijd, plaats en omstandigheden. Toch is voorspellen een dagelijks verschijnsel, of het nu het weer, de beurs of de uitslagen betreft van de Eredivisie voetbal. We kunnen het niet, maar we doen het tóch.#$In de 19e eeuw begon de industriële revolutie. Aanvankelijk voorspelde men technische toepassingen. Zodra het brede publiek aan de nieuwe tijd gewend raakte sloeg de fantasie op hol. Het kon niet óp. Jules Verne is een voorbeeld: de reis naar de maan is geplaveid met het negeren van de natuurwetten. Maar de lezers waren over het algemeen laag opgeleid dus e.e.a. viel niet op. Vandaag de dag valt de SF uiteen in twee richtingen: pure fantasie verhalen met prinsessen, zwaarden, draken, vestingen, tovenaars en dan toch nog wat planeten. En anderzijds zijn er de ruimte oorlogen en legio na-de-apocalyps verhalen. De laatste zijn stereotiep en goedkoop te verfilmen. Al wat je nodig hebt voor zo'n film is een hoop puin en een paar herkenbare ruïnes van bijv. de Eiffeltoren, het Witte Huis of een restje piramide van Cheops.#$Je kunt ook een beetje geluk hebben met voorspellen. In 1987 kregen we op het werk PC's. Die waren er al een tijdje, maar peperduur. Voortaan zaten we niet meer op 'domme' terminals aan een centrale machine. In een dolle bui zei ik: "over 20 jaar zijn er geen PC's meer". Dat moest ik uitleggen. "Nou, 20 jaar geleden waren ze er ook niet". Belachelijk! Maar ik kreeg bijna gelijk: plotseling moest alles op tablet en mobiele telefoon. Gelukkig klopte de 'voorspelling' niet, de PC bleef, zij het niet voor iedereen. Daarentegen: de PC waarop ik dit typ zou in 1972 de natte droom zijn geweest voor mensen van het rekencentrum van de Universiteit van Amsterdam met hun collegezaal vullende CDC: mijn apparaat heeft 70000 maal zoveel memory, is 300 maal zo snel, heeft geen 2 maar 4 CPU's en de disk opslag is … 130000(!) maal de hunne. **En helemaal van mij!** Wie had dat kunnen voorspellen? Ik zeker niet.

;JOOP!

THINKING OF LEARNING JAVA? WHY DON'T YOU TRY KOTLIN INSTEAD

To any junior developer with no experience in OOP or Java, I recommend checking out Kotlin, which I find a less intimidating jump into OOP than Java. Where Java can feel a tad verbose and rigid in its syntax, Kotlin's concise and simple grammar makes it less error-prone, while being very human-readable and intuitive. For instance the class declaration, which in Java requires setters, getters and a constructor, is vastly simplified as these are all optional. The language also comes with a plethora of modern convenient features such as extension functions, smart casting and lambda functions.

Kotlin example of a data class declaration:

```
data class Person(val name: String, val age: Int)
```

Java equivalent:

```
public class Person {
    private final String name;
    private final int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Overall Kotlin is a great language for backend development, and is also the preferred language to develop Android applications since 2019 [1]. It is assured to be a technology high in demand in the upcoming years, as many companies have started integrating Kotlin to their Java applications, or switching to it fully.
The downside of it, being such a recent language, can be the lack of online learning resources. Furthermore, a lot of the existing ones are focused on Android app development. The whole Kotlin track is available for free beta on the JetBrains Academy [2] and will let you get familiar with the language through easy to more complex projects. Since the beginning of the year, the Kotlin JetBrains team also started a YouTube channel presenting some of the features and possibilities Kotlin offers.
Seasoned Java developers can get a quick feel for Kotlin and the ways it differs from Java by going through the Kotlin koans on Kotlinlang.org [3].

REFERENCES

1. https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/
2. https://www.jetbrains.com/academy/
3. https://play.kotlinlang.org/koans/overview

Manon Bosselut is a Software engineer working at Ximedes. She started her career in business, and joined the IT industry in 2020.

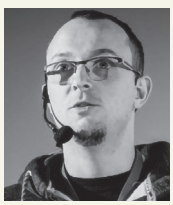
BYTE SIZE

KOTLIN GETS ITS OWN FOUNDATION!

At the end of February, JetBrains and Google jointly announced the creation of the Kotlin Foundation. The point of the foundation is to ensure the future of the language by making sure it is not held completely in the hands of a single company. The foundation will appoint the Lead Designer for the language, and manage the trademarks. You can find similar constructions in the Python foundation, or the more recent the Cloud Native Computing Foundation for example. The current board is composed of folks from Google, JetBrains and the University of Texas. That's a very good sign, because it means that Kotlin is growing, and will be able to keep growing in a healthy way. Go Kotlin!

https://kotlinlang.org/docs/kotlin-foundation.html#scope

Kotlin



Julien Lengrand-Lambert *is developer and Community builder @ing.*

Developing features that millions of people will use
Discover IT at rabobank.jobs
Rabobank
Growing a better world together.

Word jij onze future Codesmith?

Je hebt alles met **software** en het delen van **kennis**. Je leert door het samen te doen. Je wilt werken aan je toekomst. Dat klinkt als een **Future Codesmith**.

- spreken op conferenties & meetups
- schrijven van blogs & artikelen
- ontwikkelen en geven van trainingen & opleiding

Benieuwd hoe het **Codesmith Apprentice programma** future Codesmiths klaarstoomt? Scan dan de QR code.

ORDINA
Ahead of change