Nummer 2 - 2018

JAVA MAGAZINE

Onafhankelijk tijdschrift voor de Java-professional

NEDERLANDSE JAVA USER GROUP
15 JAAR JUBILEUM NLJUG!

Microservices
Stap nu over!

J-SPRING

Inclusief vooruitblik J-Spring 2018

TEQNATION

Met plattegrond, timetable en meer!

Features

Development automation met Atomist

Dé checklist voor succes als developer

Go vs. Java: pragmatisch programmeren

NLJUG LID.
50%
KORTINGS-
CODE:
NLJUGLOVESSMART

TEQNATION

THE FUTURE IS SMART

APRIL 26, 2018
JAARBEURS UTRECHT

THE DEVELOPERS CONFERENCE OF TOMORROW

AI & MACHINE LEARNING
CLOUD SERVERLESS
VR & AR
BIG DATA
IOT & EMBEDDED
BLOCKCHAIN

Celebrate the love for Smart Technology and meet like-minded Future Teqies.
At TEQnation The Developers Conference of Tomorrow you can broaden and share your knowledge at our hands-on labs, hackathons, technical sessions and workshops, which are all focused on the technology of the next era.

REGISTER NOW
www.TEQnation.nl

Platinum Partners

ING
Rabobank

Partners

Axxerion
CGI
IBM
kangaroot
SUSE
NCIM GROEP
ORACLE
snow
Quintor
redhat
topicus

Verandering is de enige constante

Ons mooie Java vakgebied is continu in beweging. Trends leggen in volle snelheid de weg af over de hype cycle of verschijnen plotseling op de Technology Radar. Ze komen voorbij als onderwerp bij een goede kop koffie met collega's of in Java Magazine.

Tel je daarbij de impact van thema's, zoals bijvoorbeeld privacy, security of BlockChain op, nou dan weet je zeker dat we nooit stil kunnen blijven staan. Samen continu blijven leren en alles in perspectief plaatsen, dat is wat mij betreft één van de mooiste kanten van ons vak.

Met dit Java Magazine hopen we daar ook weer aan bij te dragen met een diversiteit aan onderwerpen in de kern en aan de randen van ons vakgebied. Denk aan: Full Stack Reactive, Redux, het gebruik van Optional, focus op vakmanschap en aandacht voor de vraag Java of Go?

Behoefte aan meer? Verdiep je dan verder op 31 mei 2018 tijdens J-Spring of pak een stuk verbreding tijdens TEQnation op 25 en 26 april2018 met thema's als AI & Machine Learning, Cloud & Serverless, VR & AR, Big Data, IoT & Embedded en BlockChain.

Zelf bijdragen aan Java Magazine? Laat het mij weten!

Veel leesplezier! ■

Rob Brinkman
rob.brinkman@nljug.org

Colofon Java Magazine 02-2018

Content manager:
Stijn van de Blankevoort
Eindredactie:
Anne Camphuijsen
Projectcoördinator:
Tedje van Gils
Auteurs:
Koen Aerts, Rogier van Apeldoorn, Frans van Buul, David Caron, Peter Eijgermans, Emil van Galen,Erwin de Gier, Johan Janssen, Bart Kerkvliet, Corstijan Kortsmit, Joop Lanting, Robert Scholte, BertJan Schrijver, Michel Schudel, Roy Straub, Brian Vermeer
Redactiecommissie:
Rob Brinkman, Bas Passon, Ben Ooms, Erwin de Gier, Ivo Woltring, Johan Janssen, Nanne Baars, Peter Hendriks
Vormgeving:
Britt Zaal
Uitgever:
Martin Smelt
Traffic & Media order:
Marco Verhoog
Drukkerij:
Senefelder Misset, Doetinchem
Advertenties:
Richelle Bussenius
E-mail: richelle.bussenius@nljug.org
Telefoon: 023 752 39 22
Fax: 023 535 96 27
Marc Post
E-mail: mpost@reshift.nl
Telefoon: 023 543 00 08
Abonnementenadministratie
Tanja Ekel

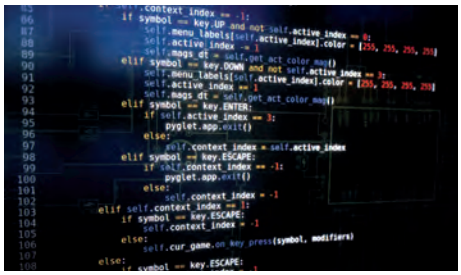
Lidmaatschap van de NLJUG kost € 44,50 (België € 45,50) per jaar, waarbij Java Magazine gratis verschijnt. Naast het Java Magazine krijgt u gratis toegang tot de vele NLJUG workshops en het J-Fall congres. Het NLJUG is lid van het wereldwijde netwerk van JAVA user groups. Voor meer informatie of wilt u lid worden, zie **www.nljug.org**. Een nieuw lidmaatschap wordt gestart met de eerst mogelijke editie voor een bepaalde duur. Het lidmaatschap zal na de eerste (betalings)periode stilzwijgend worden omgezet naar lidmaatschap van onbepaalde duur, tenzij u uiterlijk één maand voor afloop van het initiële lidmaatschap schriftelijk (per brief of mail) opzegt. Na de omzetting voor onbepaalde duur kan op ieder moment schriftelijk worden opgezegd per wettelijk voorgeschreven termijn van 3 maanden. Een lidmaatschap is alleen mogelijk in Nederland en België. Uw opzegging ontvangen wij bij voorkeur telefonisch. U kunt de Klantenservice bereiken via: 023-5364401. Verder kunt u mailen naar **members@nljug.org** of schrijven naar NLJUG BV, Ledenadministratie, Richard Holkade 8, 2033 PZ Haarlem. Verhuisberichten of bezorgklachten kunt u doorgeven via **members@nljug.org** (Klantenservice).

Wij nemen je gegevens, zoals naam, adres en telefoonnummer op in een gegevensbestand. De verwerking van uw gegevens voeren wij uit conform de bepalingen in de Algemene Verordening Gegevensbescherming. De gegevens worden gebruikt voor de uitvoering van afgesloten overeenkomsten, zoals de abonnementenadministratie en, indien je daar toestemming voor hebt gegeven, om je op de hoogte te houden van interessante informatie en/of aanbiedingen. je kunt uw persoonsgegevens opvragen om inzicht te krijgen in welke gegevens wij van je hebben, deze te corrigeren of, na beëindiging van de abonnee-overeenkomst, te laten verwijderen. Stuur hiertoe een kaartje aan NLJUG BV, afd. klantenservice, Richard Holkade 8, 2033 PZ Haarlem of een e-mail naar **members@nljug.org.nl**.


Algemene Inlichtingen- en Veiligheidsdienst
Ministerie van Binnenlandse Zaken en Koninkrijksrelaties
We zeggen het liefst zo weinig mogelijk over ons werk.
Maar kom eens langs voor een kop koffie. Dan vertellen we waarom we op zoek zijn naar jou.
Werken bij de Algemene Inlichtingen- en Veiligheidsdienst betekent werken aan een veilig Nederland.
We komen graag in contact met ervaren Java-ontwikkelaars en -programmeurs.
Wil je nu al meer weten over de AIVD? Kijk dan op **www.aivd.nl**.

14 Development automation met Atomist

Als ontwikkelaars willen we vooral bezig zijn met het schrijven van toffe code. Helaas zijn we een flink deel van onze tijd kwijt aan allerlei saaie randzaken. In dit artikel nemen we daarom het Atomist platform onder de loep. Atomist neemt een groot deel van het saaie werk voor zijn rekening, zodat jij je weer kan concentreren op wat je leuk vindt: gave code schrijven!



21 Terugblik op de eerste editie van JVMCON

Op 30 januari 2018 was het dan eindelijk zover! Na anderhalf jaar voorbereiding vond de eerste editie van JVMCON in Cinemec Ede plaats. Alweer Cinemec Ede? Jazeker, door de bioscoopopstelling heb je daar goed zicht op het scherm en de spreker. Ook ligt Ede lekker centraal en is het goed betaalbaar.

26 Preview J-Spring

31 mei verzamelen Java developers zich in TivoliVredenburg in Utrecht voor dé Java conferentie van Nederland. In het midden van deze editie van Java Magazine zit het programmaboekje van TEQnation (inclusief volledige timetable en de plattegrond) daarna vind je de preview van J-Spring.

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.

Angular en NativeScript {N}

Bouw je eigen mobiele apps

Steeds meer Java ontwikkelaars komen de laatste jaren in aanraking met front-end ontwikkeling. Voor deze groep ontwikkelaars is het nog maar een kleine stap om tot mobiele app-ontwikkeling over te gaan door middel van NativeScript. In dit artikel proberen we duidelijk te maken wat de mogelijkheden zijn van native hybride app ontwikkeling met behulp van NativeScript.

Wat is NativeScript {N}?

{N} is een open source framework (onder de Apache 2 licence) om native IOS en Android apps te bouwen met behulp van Typescript en Angular. {N} is een andere technologie dan de hybride-frameworks, zoals Ionic en Phonegap. {N} is een runtime, geen web technologie. Je app zal niet draaien als een mini website in een WebView en is daarom meer performant. Met {N} heb je direct toegang tot al de Native APIs van je device.

TypeScript:

Voor Javanen is het interessant om te beseffen dat TypeScript (TS) heel veel overeenkomsten met Java heeft. Je kunt code kloppen waarvan je je afvraagt of het TS of Java is. TS is de beste manier om {N} apps te schrijven, al dan niet gecombineerd met Angular. Verder is het schrijven van native code in TS voor Android vrij eenvoudig, omdat je in de native Java code veel één-op-één kunt overnemen (bijvoorbeeld: new io.java.File() is zowel geldige Java als JS/TS. Dat ligt bij Objective C net even wat lastiger). Het is heel fascinerend om te zien hoe NativeScript het voor elkaar krijgt om met puur JavaScript alle native constructies te implementeren. Van String Arrays tot interfaces en implementeren van abstract classes.

Waarom NativeScript?

- Eén van de argumenten om NativeScript {N} te gebruiken, is hergebruik van 'Skills'. Dat wil zeggen iemand met kennis van JavaScript / Typescript kan direct aan de slag met {N}. {N} wordt namelijk geschreven in JavaScript of Typescript.
- Hergebruik van code. Schrijf Native Mobile apps voor iOS en Android met één enkele codebase en één gezamelijke interface. Dit is bijvoorbeeld niet het geval bij Xamarin waar je toch twee interfaces moet bouwen voor IOS en Android met één common layer. Het is niet zo vreemd dat de slogan van {N} luidt: 'Write once, run everywhere'.
- Het is tevens gemakkelijk uit te breiden met behulp van {N} modules (zie de voorbeelden in dit artikel) en npm modules. Eigenlijk de plug-ins van Cordoba / Phonegap.
- Tenslotte wordt geen gebruik gemaakt van WebViews, zoals bij vele hybride frameworks. Met JavaScript hebt je direct toegang tot de Native APIs en is daardoor beter performant.

Wat is er nodig voor NativeScript?

Op **https://docs.nativescript.org/start/quick-setup** staat een duidelijke beschrijving hoe {N} moet worden geïnstalleerd. Omdat deze installatie niet altijd feilloos verloopt, is het verstandig om 'NativeScript Sidekick' te installeren.

Met Sidekick is het mogelijk om apps te builden in de cloud. Als je ervoor kiest om je apps te builden in de cloud, dan kan je onafhankelijk van je operating system waarop je werkt je apps ontwikkelen. Je kan dan uiteraard zelfs iOS apps builden op een Windows machine. Het is ook mogelijk om met Sidekick lokaal aan de slag te gaan, maar daarvoor moet je je eigen omgeving inrichten met iOS Xcode en Android SDK. Sidekick heeft nog veel known issues, maar werkt al aardig. Het mooie aan {N} is ook Live Sync. Dus als je een regel in de code aanpast, zal dit zelfs met de cloudbuild direct op de telefoon worden 'gepushed' en zo is het resultaat van de wijziging direct zichtbaar.

Peter Eijgermans is een enthousiaste Java Web Developer bij Ordina JTech. Onlangs geridderd tot Codesmith.

Rogier van Apeldoorn is een innovatieve Mobile Developer en Codesmith bij Ordina JTech.

Hoe werkt NativeScript met behulp van JavaScript?

In **Listing 1** volgt een simpel voorbeeld van JavaScript in {N}, die een Objective-C based IOS UIAlertView control instantieert.

Omdat web developers geen iOS en Android specifieke APIs willen aanleren, biedt {N} een set van {N} modules aan. Die modules abstraheren de iOS en Android details in simpele JavaScript APIs. De bovenstaande UIAlertView-gebaseerde code kan herschreven worden met de {N} 'Dialog module' (zie **Listing 2**).

Deze dialogs.alert()call levert ons ook de android.app.AlertDialog voor je Android app. Ondanks dat dit 'dialog'' voorbeeld eenvoudig is, kan dezelfde techniek worden ingezet voor het bouwen van robuuste apps. Hiervoor kun je gebruik maken van de reeds bestaande en volwassen native iOS en Android UI componenten.

Wat kan Angular toevoegen aan NativeScript?

{N} kan nu ook geschreven worden in Angular. Als je kennis hebt van Angular, dan is het een kleine stap om Angular in {N} te gebruiken. Het grote verschil met Angular is dat de browser-based HTML-elementen, zoals en , niet beschikbaar zijn in {N}. Je dient daarvoor in de plaats {N} UI componenten te gebruiken. In {N} wordt geen DOM of browser gebruikt. {N} UIs zijn native UIs en dus losgekoppeld van de DOM. Omdat Angular een agnostisch framework is en losgekoppeld is van de DOM, kan dit framework gemakkelijk geïntegreerd worden met {N}. Onderstaand voorbeeld gaat hierop in. *AngularJS* is in tegenstelling tot Angular niet geschikt voor {N}, omdat dit framework gekoppeld is aan de DOM.

Door gebruik te maken van Angular in {N}, heb je de mogelijkheid om code te delen tussen je bestaande webapplicatie en je Native apps. Laten we kijken naar een voorbeeld (zie **Afbeelding 2**).

Hergebruik van code tussen web en mobile apps

Laten we als voorbeeld een 'Grocery List' tonen in een webapplicatie (zie **Listing 3**). In **Listing 3** is een Angular Component gedefinieerd, die een array van 'groceries' vult, via de constructor. Er wordt gebruik gemaakt van TypeScript. Dit is een 'typed superset' van JavaScript en is de standaard voor het schrijven van Angular applicaties.

```
var myAlert = new UIAlertView();
myAlert.message = "NativeScript rocks!";
myAlert.show();
```

Listing 1

```
var dialogs = require("ui/dialogs");
dialogs.alert({ message: "NativeScript rocks!" });
```

Listing 2

```
import {Component} from '@angular/core';

@Component({
    selector: 'grocery-list'
    templateUrl: 'grocerylist.template.html'
})

export class GroceryListComponent {
    groceries: string[];

    constructor()  {
        this.groceries =['milk, 'bread']
    }
}
```

Listing 3

```
....
    <grocery-list></grocery-list>
....
```

Listing 4

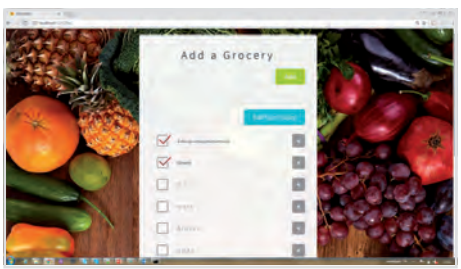
```
<p>Groceries: </p>
  <ul>
    <li *ng-for = "let grocery of groceries">
            {{grocery}}
    </li>
  </ul>
```

Listing 5

Hoe werkt dit component?

Via de *selector* behorend bij het Component, kan je Angular vragen om dit component te instantiëren en te renderen, waar het een <grocery-list> - tag vindt in de HTML (zie **Listing 4**).

Voor het renderen van de 'Grocery list', dient er tenslotte een template gedefinieerd te worden met de naam: grocerylist.template.html (zie **Listing 5**). Deze template staat ook gedefinieerd onder de *templateUrl* van het Component. Er wordt door de lijst met 'groceries' geïte-



Afbeelding 2

Afbeelding 1

```
<StackLayout *ngFor = “let grocery of groceries”>
                <Label [text] = “grocery”></Label>
</StackLayout>
```

Listing 6

```
var http = require(“http”);  à import HTTP module

http.getJSON(‘https://api.groceries.com’)
        .then(function (result)) {
                …
       });
```

Listing 7

reerd, met behulp van de *ng-for* directive van Angular. De vraag is nu: wat moeten we anders programmeren om bovenstaande code werkend te krijgen voor je NativeScript iOS en Android apps?

Het uitgebreide antwoord is: het component in **Listing 3** blijft hetzelfde. Deze TypeScript-code kan je dus hergebruiken tussen de webapplicatie en mobile apps. Het enige wat je dient te wijzigen is de template voor dit component. Je dient {N} UI componenten te gebruiken in plaats van HTML-elementen in je template (zie **Listing 6**). Wat opvalt is dat je nog steeds de *ng-for* directive van Angular kan gebruiken voor het itereren door de lijst.

Het omzetten van een template naar {N} UI componenten is een eenvoudige exercitie, omdat voor ieder HTML-element wel een Native UI component te vinden is. Dit voorbeeld laat ook zien dat Angular losgekoppeld is van de DOM en dus met {N} UI componenten kan omgaan.

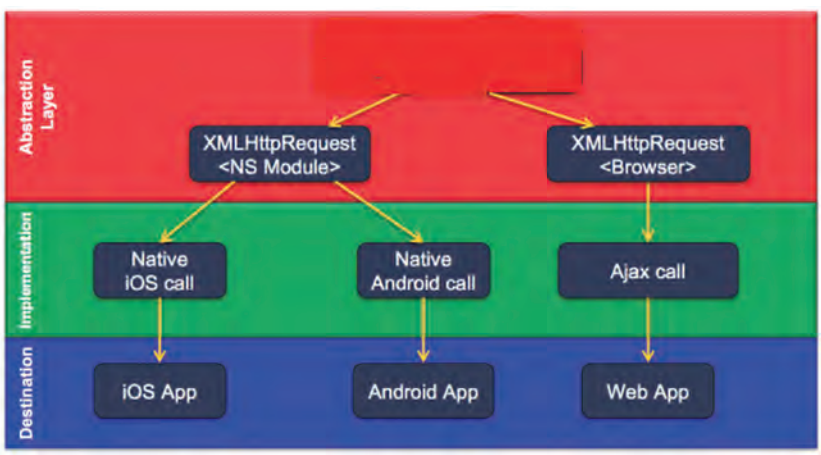
HTTP Module

Een ander voorbeeld voor code hergebruik is het volgende. {N} biedt support voor web APIs, zoals XMLHttpRequest en fetch(). Je kan hiervoor de HTTP module gebruiken binnen {N}. Deze module maakt het mogelijk om web requests te verzenden en web responsies te ontvangen (zie **Listing 7**).

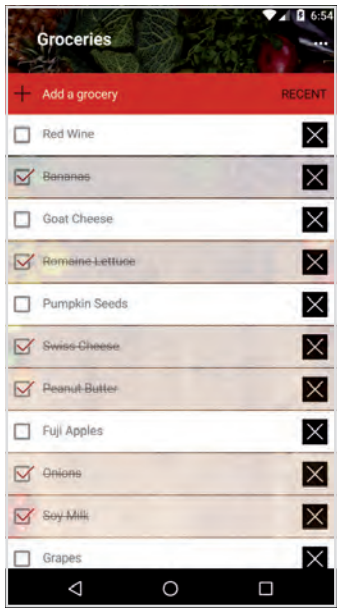
In **Afbeelding 4** zie je aan de linkerzijde dat {N} de XMLHttpRequests web API vertaalt naar Native code voor iOS en Android. Tevens wordt aan de rechterzijde van dit schema de implementatie van de XMLHttpRequest getoond die gebruikt wordt voor de webapplicatie. Dit is een andere implementatie dan de NativeScript variant.

Het probleem hier is dat we de XMLHttpRequest implementaties niet kunnen hergebruiken tussen de webapplicatie en mobiele app. Er is hier een ''missing link'', namelijk een generieke HTTP-module voor de webapplicatie en de mobiele app. Dit kan opgelost worden met Angular. Angular voegt een extra abstractielaag toe voor het afhandelen van HTTP-requests/responses. Dit kan met de HTTP-module van Angular. Deze module kunnen we hergebruiken tussen de webapplicatie en de mobiele app.

De te hergebruiken code ziet er dan bijvoorbeeld als volgt uit in Angular (zie **Listing 8**).

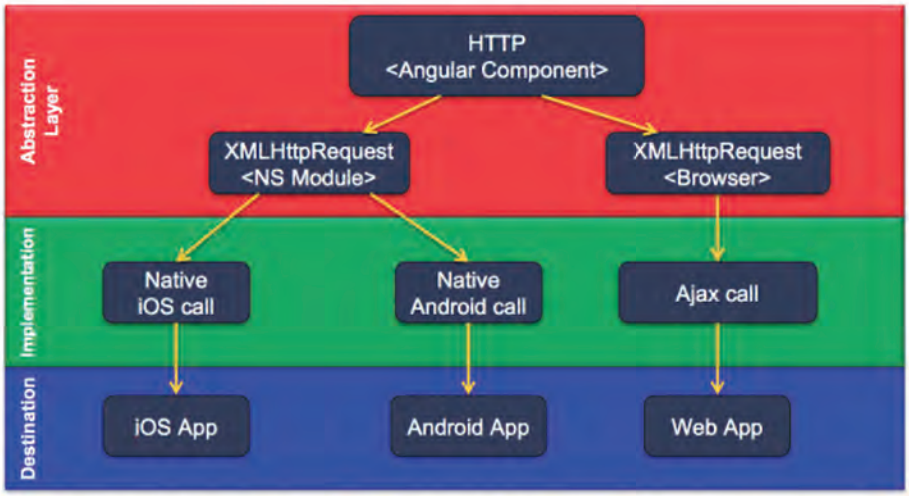


Afbeelding 4



Afbeelding 3

WRITE ONCE, RUN EVERY-WHERE



Afbeelding 5

Meer tools en features

{N} biedt ons naast hierboven genoemde voorbeelden ook nog andere featues, zoals het gebruik van plug-ins, scaffolding en de {N} Playground. Verder kan je momenteel ook met het *Vue.js framework* een {N} app bouwen.

Plug-ins

Het interessante aan {N} is dat er veel plug-ins beschikbaar zijn in de marketplace. **https://market.nativescript.org**. Je hoeft dus niet alles zelf te bouwen. Denk aan Facebook, fingerprint authenticatie, 'text to speech', advertenties met Admob of in-app purchase. Mocht de plug-in er nog niet zijn, dan kan deze altijd zelf worden gebouwd.

NativeScript Playground

{N} Playground is een tool waarmee op eenvoudige wijze een app kan worden gebouwd met behulp van een web interface. Zie **https://play.nativescript.org**. Naast de website moet op een device twee apps worden geïnstalleerd:

- Nativescript Playground
- Nativescript Preview

Na het scannen van de QR-code van de **https://play.nativescript.org** website met behulp van de playground app, zal de app worden geïnstalleerd op het device. Met behulp van Live Sync zullen alle wijzigingen direct zichtbaar worden in de app. Supercool. Probeer het eens uit! Met Playground kan je eenvoudig je {N} code en issues delen met andere ontwikkelaars. Het is een middel om snel aan de slag te gaan met {N} en de voordelen ervan te ervaren.

Tot slot

Hergebruik van code is een zeer belangrijke uitdaging voor development in het algemeen. Op dit moment kan je met de *Angular CLI* een webapplicatie genereren. En voor {N} gebruik je de *NativeScript CLI*. Dit is een probleem, omdat de CLIs ons geen mogelijkheid geven om één project te maken voor de webapplicatie en de Native applicatie.

Natuurlijk is het mogelijk om twee aparte projecten te onderhouden en de gedeelde files tussen de twee projecten te 'copy-pasten'. Dit kan ook opgelost worden door bijvoorbeeld het volgende 'seed project' te gebruiken: **https://github.com/TeamMaestro/angular-native-seed**. Hiermee kan je met één codebase je webapplicatie en NativeScript onderhouden. Samenvattend, is {N} een krachtig framework om native cross-platform mobiele applicaties te bouwen met Angular, Vue.js, TypeScript of JavaScript. Je hebt direct toegang tot de Native platform API's. {N} werkt anders dan hybride frameworks, zoals Phonegap. Maar is uitermate geschikt voor webdevelopers. Het is beter performant en eenvoudig te leren. ■

```
        import  {Http} from '@angular/http'

        export class GroceryListService {

           constructor(private http: Http)  {}

getGroceries(){
return this.http.get(`https://api.groceries.com`)
.map((res:Response) => res.json());
          }

        }
```

Listing 8

Cloud Native CI/CD met Spring Cloud Pipelines

Inmiddels onmisbaar!

Met de opkomst van de Cloud Native Computing, Microservices en DevOps is een goede CI/CD pipeline onmisbaar geworden. Omdat het idee achter microservices en DevOps is dat teams onafhankelijk en met grote snelheid kunnen ontwikkelen, moet het pad van code naar productie zoveel mogelijk geautomatiseerd zijn. Teamleden moeten het vertrouwen hebben dat ze nieuwe functionaliteit kunnen toevoegen met zo weinig mogelijk risico op productieverstoringen.

Pipeline als code is niets nieuws. Eind 2016 schreef Bert Jan Schrijver al in dit blad over de toen nieuwe pipeline-functionaliteit in Jenkins 2. Toch bestaat nog veel onduidelijkheid en verschil van mening over de precieze opbouw van de pipeline. Spring Cloud Pipelines reikt een geopinieerde standaard aan om een robuuste pipeline op te bouwen.

Dit artikel geeft een overzicht van de verschillende concepten van Spring Cloud Pipeline en zal onderweg ook nog wat interessante technologieën introduceren.

Spring Cloud Pipelines

Spring Cloud Pipelines, opgestart in 2016 door Marcin Grzejszczak, bestaat uit een conceptuele pipeline en een set templates en implementaties voor verschillende CI/CD servers en Cloud Platforms. Met deze pipeline kunnen Spring Boot applicaties gebouwd, getest en gedeployed worden binnen een microservices-platform. De pipeline weerspiegelt een aantal best practices en subjectieve keuzes van het Spring Cloud team.

De pipeline geeft invulling aan de volgende doelen:

- Test de applicatie in isolatie;
- Test de backwards compatibility van de applicatie, zodat een rollback mogelijk is;
- Faciliteer het testen van de gedeployede applicatie;
- Faciliteer acceptatie-tests en performance-tests van de gedeployede applicatie;
- Deploy naar productie.

Ondersteunde CI/CD servers en pipeline-formaten:

- Jenkins Job DSL plugin;
- Jenkins 2 Jenkinsfile;
- Concourse.

Ondersteunde deployment-platforms:

- Kubernetes;
- Cloud Foundry.

Verdere gerelateerde technologieën:

- Git;
- Maven of Gradle voor de build;
- Artifactory of Nexus als artifact repository;
- In het geval dat Kubernetes gebruikt wordt een Docker repository;
- Spring Cloud Contract, voor de implementatie van Consumer Driven Contracts met Spring.

David Caron was 15 jaar als tech lead betrokken bij Java/Spring-projecten en werkt momenteel als Platform Architect bij Pivotal.

Build en upload

Volgens het bekende "build once and run anywhere" principe worden alle artifacts éénmaal gebouwd en opgeslagen voor later gebruik:

- De applicatie wordt gebouwd met Maven of Gradle en gepubliceerd naar de artifact repository.
- Stubs voor de (REST) interfaces worden gepubliceerd naar de artifact repository (zoals hieronder beschreven onder het kopje "Spring Cloud Contract".
- In het geval dat Kubernetes het doelplatform is, wordt een Docker image gebouwd en gepubliceerd.

API compatibility check

In deze stap wordt getest of de nieuwe versie van een API de contracten met de afhankelijke client-services die momenteel in productie staan, niet verbreekt. Dit gebeurt met behulp van Spring Cloud Contract. De technische invulling hiervan wordt verderop in het artikel toegelicht.

- Er wordt gekeken of er al een eerdere productie-release geweest is.
- Bij deze -nu in productie staande- release zitten API-contracten (als aparte jar-file in de maven repository). De API-compatibility tests in de nieuwe applicatie worden getest tegen de contracten van de huidige productie versie.

Deploy naar testplatform

- De applicatie wordt gedeployed naar een eigen Cloud Foundry space of Kubernetes namespace.
- Alleen deze applicatie wordt hier in isolatie gedeployed met de benodigde stubs.
- De testdatabase wordt gemigreerd met bijvoorbeeld Flyway of Liquibase.

Testplatform smoke test

De smoke tests worden uitgevoerd tegen de nieuwe applicatie. Dit zijn dus tests die uitgevoerd worden tegen de gedeployede applicatie en zijn stubs. De aanbeveling is dat het hier gaat om een paar cruciale use cases; het gaat hier niet om volledigheid.

Testplatform rollback

Nu wordt de huidige productieversie weer gedeployed. De database, die door de nieuwe applicatie is gemigreerd, blijft gewoon staan.

Testplatform rollback smoke test

De smoke tests van de huidige productieversie worden weer uitgevoerd. Als deze slagen is er behoorlijk wat vertrouwen dat een eventuele rollback van de komende productie-deploy zal slagen.

Deploy naar acceptatieplatform

De applicatie wordt gedeployed naar het acceptatieplatform waar alle andere microservices ook aanwezig zijn. Deze stap, en de volgende, zijn alleen nodig en/of wenselijk als de organisatie er nog niet genoeg vertrouwen in heeft dat fouten in productie snel genoeg gedetecteerd kunnen worden.

Acceptatieplatform end-to-end test

De end-to-end tests worden uitgevoerd tegen het hele systeem. Aangezien dit soort testen vaak onderhoudsgevoelig zijn, is het het beste om hier hooguit enkele van te hebben.

Deploy naar productieplatform

- Er wordt in git een tag gezet met prod/${version}.
- Door middel van blue-green deployment wordt productie overgeschakeld naar de nieuwe versie.

Productieplatform finaliseren

Als we zeker weten dat de deploy gelukt is en we geen rollback nodig hebben, dan kan de oude versie worden opgeruimd. Dit is een handmatige stap.

Productieplatform rollback

- Als de productie-deploy fout is gegaan, dan wordt het verkeer weer naar de oude applicatie geleid.
- De laatste productie-tag in git wordt gewist.

Zelf aan de slag

Spring Cloud Pipelines omvat een combinatie van zeer uiteenlopende technologieën die behoorlijk wat opswerk vereisen om goed te installeren en op te zetten. Het voert daarom te ver door om hier een volledige tutorial neer te zetten. Bovendien is een aantal zeer uitgebreide tutorials te vinden op de Spring Cloud Pipelines GitHub pagina.

Misschien is de simpelste manier om de pipeline helemaal te zien werken en alle aspecten te doorgronden om de tutorial van Cora Iberkleid te doen die hoort bij haar presentatie met Marcin op SpringOne Platform afgelopen december 2017. Deze tutorial maakt gebruik van de Concourse CI/CD server en Cloud Foundry. In drie stappen migreert ze een tweetal simpele Spring Boot applicaties (gebaseerd op het "fortune teller" Spring Cloud voorbeeld) naar het gebruik van Spring Cloud Pipelines. De drie stappen zijn:

1. Eerst worden de minimale wijzigingen aangebracht die nodig zijn voor applicatie om door de pipeline heen te komen.
2. Vervolgens worden verschillende tests en test-profielen toegevoegd die de tests in verschillende stappen inhoud geven. Ook worden database-migraties met Flyway toegevoegd.
3. In de laatste stap worden de Spring Cloud Contract tests toegevoegd zodat de onderlinge afhankelijkheden worden geverifieerd.

Hier is de URL van het relevante gedeelte van de documentatie: **https://github.com/spring-cloud/spring-cloud-pipelines#step-by-step-cloud-foundry-migration**.

Gebruikte tools

De volgende tools worden gebruikt als infrastructuur voor de pipeline.

GitHub

In GitHub kun je de relevante projecten forken. In een branch genaamd "version" wordt in een file de release-versie bijgehouden. Builds worden getagd met dev/$VERSION. Als alle tests geslaagd zijn wordt de tag prod/$VERSION gezet.

Concourse

Concourse is een open source CI/CD server waarin pipeline als code centraal staat. Waar een Jenkins-installatie door configuratie in een GUI en door de verschillende plugins en plugin versies vaak onderhevig is aan *configuration drift*

```
fly --target local login --concourse-url http://localhost:8080 --username concourse --password xxxxxxxx
fly --target local set-pipeline --pipeline greeting-ui --config
"${SCP_HOME}/spring-cloud-pipelines/concourse/pipeline.yml" --load-vars-from
"${SCP_HOME}/credentials/credentials-greeting-ui.yml" -n
```

Listing 1

biedt Concourse veel sterkere garanties dat de pipeline code op een willekeurige Concourse-installatie werkt.

Ook worden stappen in de pipeline geïsoleerd in Docker containers uitgevoerd, zodat de dependencies van de test volledig voorspelbaar zijn. Verder is de GUI van Concourse ontwikkeld om op een scherm een duidelijke weergave van de staat en de activiteit van de pipeline te geven.

Je interacteert met Concourse door middel van een command-line-interface (zie **Listing 1**).

Je kunt Concourse lokaal draaien als VM of met docker-compose: **http://concourse.ci/installing.html**.

Zorg wel dat je een goede internetverbinding hebt. Voor iedere stap worden Docker-images gedownload. Als je krediet hebt op AWS, dan is Concourse-up de makkelijkste manier van installeren: **https://github.com/EngineerBetter/concourse-up**.

JFrog Bintray

JFrog Bintray is een publieke gehoste Maven Repository waarvoor gratis accounts aangemaakt kunnen worden: **https://bintray.com/signup/oss**.

Pivotal Web Services

Pivotal Web Services is een publiek gehoste versie van Cloud Foundry. Cloud Foundry is een open source cloud native platform. Applicaties kunnen met een commandline interface direct naar het platform gedeployed worden en gekoppeld aan backend services, zoals databases en message queues. Ze zijn daarna meteen beschikbaar onder een URL. Specifiek voor Spring Cloud kunnen gehoste versies van de Spring Cloud Netflix aangemaakt worden, zoals Eureka voor service discovery, Config server en Circuit breaker. Op Pivotal Web Services (**http://run.pivotal.io/**) kun je een gratis proefaccount aanmaken. Om een idee te geven hoe simpel deloyment naar Cloud Foundry is hier een klein voorbeeld (zie **Listing 2**).

Spring Cloud Contract

Spring Cloud Contract is een Spring Cloud project onafhankelijk van Spring Cloud Pipeline. Voor het begrip van sommige stappen in de pipeline is het belangrijk om te weten wat Spring Cloud Contract is. Spring Cloud Contract levert een verzameling tools om op verschillende manieren API compatibility te testen. Met Consumer Driven Contracts geven consumer services aan wat hun verwachtingen zijn van de producer service. Dit zorgt ervoor dat het team dat de producer service ontwikkelt weet wat de impact is van wijzigingen op hun consumers.

Het contract

In een Domain Specific Language (DSL) wordt het contract van de service met een specifieke consumer-service gedefinieerd. In **Listing 3** zie je een voorbeeld in Groovy (Sinds kort zijn ook andere formaten beschikbaar, zoals YML).

Producer

In het project aan de server-kant van de API voeg je een Maven- of Gradle plugin toe die op basis van een base class (die je zelf definieert) een unit test genereert voor het contract dat je hebt gespecificeerd (zie **Listing 4**). Op basis van deze base class wordt de volgende test gegenereerd (zie **Listing 5**).

Wanneer de test slaagt, dan worden de contracten en gegenereerde stubs in het project gebundeld als een jar-file en samen met het artifact van de applicatie naar de repository gepubliceerd.

Consumer

In het project aan de client-kant van de API wordt als dependency een implementatie van "Spring Cloud Contract Stub Runner" toegevoegd. Vervolgens kunnen tests gedefinieerd worden waarin de API-client getest worden tegen de eerder gegenereerde server stubs. De "Spring

```
# clone een voorbeeldapplicatie
git clone https://github.com/scottfrederick/spring-music.git
cd spring-music
# bouw en deploy
./gradlew assemble && cf push
# maak een database schema aan
cf create-service cleardb spark music-db
# bind de app aan de db
cf bind-service spring-music music-db
# restart
cf restart spring-music
```

Listing 2

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
   description("""
should return a fortune string
""")
   request {
       method GET()
       url "/"
   }
   response {
       status 200
       body "foo fortune"
   }
}
```

Listing 3

Cloud Contract Stub Runner" haalt de aangewezen server stubs uit de Maven repository en start vervolgens WireMock (zie **Listing 6**).

Uiteraard is dit allemaal een hoop werk en is er nog volop discussie in de community over Consumer Driven Contracts, maar het levert natuurlijk wel wat op.

- Specificatie van de API
- Pijlsnelle integratietests met stubs die je niet zelf hoeft te onderhouden.
- Testen van backwards compatibility (waarover later meer).

De ruimte is hier te beperkt om een volledig voorbeeld op te nemen, maar ik verwijs je graag naar de voorbeelden op de website van Spring Cloud Contract.

Conclusie

Het opzetten van een CI/CD pipeline met Spring Cloud Pipeline is niet triviaal en vergt best wat trail and error om voor elkaar te krijgen. Als het eenmaal is gelukt, heb je wel een pipeline die een behoorlijke mate van betrouwbaarheid garandeert. Ook weet iedereen die met deze pipeline gewerkt heeft waar hij aan toe is bij een volgend project met deze pipeline.

Zeker wanneer er veel services zijn, dan lijkt Spring Cloud Pipeline zeker de moeite van het overwegen waard! ■

```
public class BaseClass {

   @Before
   public void setup() {
       FortuneService service = BDDMockito.mock(FortuneService.class);
       BDDMockito.given(service.getFortune()).willReturn("foo fortune");
       RestAssuredMockMvc.standaloneSetup(new FortuneController(service));
   }
}
```

Listing 4

```
public class Greeting_uiTest extends BaseClass {

  @Test
  public void validate_shouldReturnAFortune() throws Exception {
     // given:
        MockMvcRequestSpecification request = given();

     // when:
        ResponseOptions response = given().spec(request)
              .get("/");

     // then:
        assertThat(response.statusCode()).isEqualTo(200);
     // and:
        String responseBody = response.getBody().asString();
        assertThat(responseBody).isEqualTo("foo fortune");
  }
}
```

Listing 5

REFERENTIES

- **https://www.cncf.io/about/charter/**
- **https://en.wikipedia.org/wiki/DevOps**
- **http://www.nljug.org/databasejava/pipeline-code/**
- **https://cloud.spring.io/spring-cloud-pipelines/**
- **http://concourse.ci/**
- **https://cloud.spring.io/spring-cloud-contract/**
- **https://martinfowler.com/articles/consumerDrivenContracts.html**
- **https://12factor.net/build-release-run**
- **https://www.youtube.com/watch?v=vJUn80y63yM**
- **http://wiremock.org/**
- **https://twitter.com/jkuipers/status/964606807737028608**
- **https://cloud.spring.io/spring-cloud-contract/**

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = GreetingUIApplication.class, webEnvironment = SpringBootTest.WebEnvironment.
NONE,
       properties = {"spring.application.name=greeting-ui", "spring.cloud.circuit.breaker.enabled=false",
"hystrix.stream.queue.enabled=false"})
@AutoConfigureStubRunner(ids = {"io.pivotal:fortune-service:+"},
       //repositoryRoot = "${REPO_WITH_BINARIES}"
       workOffline = true
)
public class FortuneServiceTests {

   @Autowired FortuneService fortuneService;

   @Test
   public void shouldSendRequestToFortune() {
       // when
       String fortune = fortuneService.getFortune();
       // then
       BDDAssertions.then(fortune).isEqualTo("foo fortune");
   }
}
```

Listing 6

Development automation met Atomist

Meer tijd overhouden voor ontwikkelwerk dat je écht leuk vindt

Als ontwikkelaars willen we vooral bezig zijn met het schrijven van toffe code. Helaas zijn we een flink deel van onze tijd kwijt aan allerlei saaie randzaken. In dit artikel nemen we daarom het Atomist platform onder de loep. Atomist neemt een groot deel van het saaie werk voor zijn rekening, zodat jij je weer kan concentreren op wat je leuk vindt: gave code schrijven!

Hoeveel uur per dag ben je nu eigenlijk bezig met het daadwerkelijk leveren van toegevoegde waarde voor je klanten? Naast coderen ben je bezig met het onderhouden van je build pipeline, het upgraden van library dependencies over meerdere componenten, het met terugwerkende kracht integreren van die nieuwe NoSQL store in elk component van je project, het heen en weer switchen van je IDE naar chat naar webbrowser, *ad inifinitum*. Behalve dat die werkzaamheden vaak saai zijn, heb je elke keer weer te maken met een *context switch*. Elke keer dat je terugkeert naar je IDE, moet je weer even 'op gang' komen.

Atomist

Atomist (**https://atomist.com**), bedacht door Rod "Spring Framework" Johnson, is een platform dat ontwikkelwerk vergaand automatiseert. Zelf automatiseren we natuurlijk al veel in onze buildstraten, maar sommige taken zijn lastiger te automatiseren. Denk aan library upgrades, de daarbij benodigde codewijzigingen, het sluiten van issues wanneer fixes zijn uitgerold naar productie of een rollback op productie wanneer die mooie nieuwe feature toch onverwachts problemen geeft.

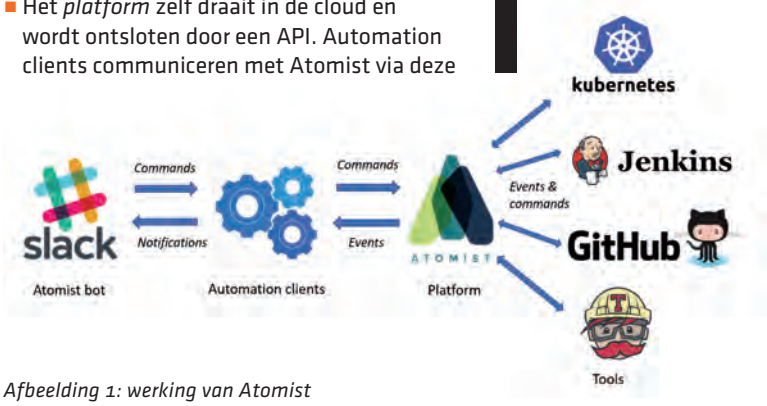
De belangrijkste onderdelen van Atomist zijn:

- *Single point of interaction* om context switches tegen te gaan.
- *Automatisering* van bekende werkzaamheden als issue tracking, het opzetten van nieuwe componenten en wijzigingen op bestaande code.
- *Project context-awareness*. Atomist is op de hoogte van de context van het project, bijvoorbeeld de architectuur, procesafspraken en de gebruikte programmeertalen.
- *API* voor het schrijven van je eigen automatiseringscode.

Architectuur

Een overzicht van de globale werking van Atomist vind je in **Afbeelding 1.**
De belangrijkste onderdelen van Atomist zijn:

- *Atomist Bot*. Deze Slack Bot is je toegangspunt tot Atomist en is je *single point of interaction*. Via de bot stuur je commando's naar Atomist om taken uit te voeren, zoals het genereren van een project, het vastleggen van een issue in GitHub of het starten van een build in je build server. De Bot biedt tweerichtingsverkeer: je kunt notificaties die vanuit je buildstraat binnenkomen, ook direct zien. Omdat alles in Slack samenkomt, neemt interactie weinig tijd in beslag en blijf je beter in je workflow zitten.
- *Automation clients*. Elk Bot commando start een automation client. Dit is een TypeScript-applicatie die één specifieke automatiseringstaak implementeert. Naast aanroepen van meegeleverde clients kan je ook zelf clients schrijven.
- Het *platform* zelf draait in de cloud en wordt ontsloten door een API. Automation clients communiceren met Atomist via deze



Afbeelding 1: werking van Atomist

Michel Schudel is Full Stack Software Engineer bij **Craftsmen.nl**. Hij deelt graag zijn kennis van nieuwe technologie met anderen middels workshops, presentaties en blogs.

API door het sturen van commands en het ontvangen van events.
- De *tooling* in je buildstraat wordt door Atomist aangestuurd. Om wijzigingen te kunnen maken, moet Atomist geautoriseerd worden tot bijvoorbeeld je Git repositories. Dit wordt tijdens de installatie geregeld. Omgekeerd worden http callbacks (*webhooks*) gebruikt om Atomist op de hoogte te stellen van events.

Installatie

Via de website van Atomist installeer je een Bot in Slack. Je kunt dan direct aan de slag. De Slack Bot *@atomist* zorgt voor communicatie met het platform. Je nodigt deze uit in het Slack channel met het commando:

```
/invite @atomist
```

Daarna kan je met het commando:

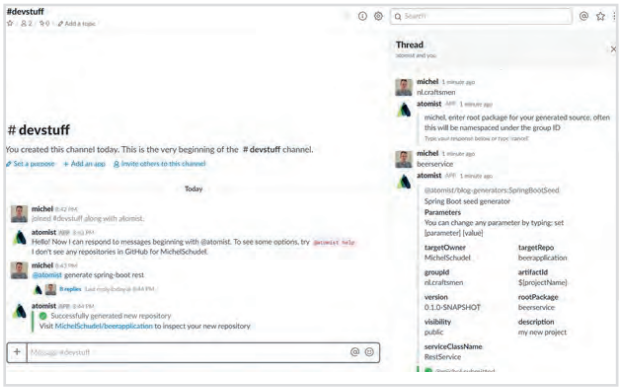
```
@atomist help
```

alle commando's opvragen. De opties die je dan krijgt, zijn out-of-the-box automation clients. Een paar voorbeelden:
- *create issue.* Maakt een nieuw issue aan in GitHub.
- *list skills.* Geeft een uitgebreide lijst van alle ingebouwde automation clients die aanwezig zijn.
- *repo.* Koppelt een GitHub repository aan het huidige Slack channel. Eventuele notificaties van GitHub zullen dan in dat channel verschijnen.

Zie een interactie voorbeeld in **Afbeelding 2**. Hier zijn de volgende zaken te zien:
- Atomist is als Slack Bot actief in het channel *#devstuff*.
- Met de opdracht @atomist generate springboot-rest wordt een *generator* automation client gestart die een nieuwe Spring Boot-applicatie genereert. De conversatie over de details, zoals root package, target repository en dergelijke, staat rechts in de conversatie thread.

Generators

Een *generator* is een automation client die op basis van een aantal parameters nieuwe code voor je genereert. Nu is dit op zich niet zo bijzonder; een tool als Spring Initializr (**https://start.spring.io/**) kan dit ook. Deze tools zijn meestal template-based. Op dat punt onderscheidt Atomist zich: Generators werken altijd op basis van een *seed*, wat een *werkend*



Afbeelding 2: interactie met Atomist in Slack

prototype is. Op dit prototype worden vervolgens een aantal transformaties losgelaten om zo tot jouw specifieke project te komen. Een Atomist generator bestaat dan ook altijd uit werkende code én transformatieregels.

Spring Boot applicatie

De transformatieregels van de Spring Boot generator staan in **Listing 1** (te vinden op **https://github.com/atomist/spring-automation**). De Spring Boot *seed* staat in een andere repository (**https://github.com/atomist-seeds/spring-rest-seed**) en is hier weggelaten. Een paar interessante transformaties in deze generator:
- doUpdatePom (met de parameters die via Slack aan de gebruiker zijn gevraagd) – zorgt voor invulling van een Maven pom.
- inferStructureAndMovePackage – zorgt voor transformatie van het standaard package in de seed naar jouw eigen root package, meegegeven via de rootPackage parameter (te zien in **Afbeelding 2**).

Het voordeel van generatie op basis van een werkend prototype is duidelijk. Je prototype is je werkende referentie-architectuur. Je maakt daar wijzigingen in. Vervolgens gebruik je Atomist om op basis van de nieuwste inzichten nieuwe projecten te starten... of te wijzigen. Dit laatste doe je met een ander soort automation client, genaamd een *editor*.

Editors

Niet-triviale code-wijzigingen op meerdere plekken tegelijk is meestal handwerk. Om dit te automatiseren gebruikt Atomist *editors*, automation clients die bestaande code transformeren. Denk aan het upgraden van library-versies over meerdere subprojecten of het verwijderen van onnodige code. Een voorbeeld van dit laatste staat in **Listing 2**. Deze out-of-the-box editor spoort alle Spring beans op

ATOMIST ONDERSTEUNT HET AUTOMATISEREN VAN ONTWIKKEL-TAKEN WAAR-VOOR TOT NU TOE NOG GEEN HAPKLARE TOOLS WAREN

```
export function springBootProjectEditor(params: SpringBootGeneratorParameters):
    AnyProjectEditor<any> {
    const starterEditors: Array<AnyProjectEditor<any>> =
        params.starters.map( callbackfn: starter =>
            addSpringBootStarter( artifact: "spring-boot-starter-" + starter));
    logger.debug("Starters: [%s]. Editor count=%d", params.starters.join(),
      starterEditors.length);
    const editors: Array<AnyProjectEditor<SpringBootGeneratorParameters>> = [
        curry(cleanReadMe)(params.target.description),
        curry(cleanTravisBuildFiles)(slackTeamTravisWebhookUrl(params.slackTeam)),
        curry(doUpdatePom)(params),
        curry(inferStructureAndMovePackage)(params.rootPackage),
        curry(inferSpringStructureAndRename)(params.serviceClassName),
    ];
    return chainEditors(
        ...editors.concat(starterEditors),
```

Listing 1: generator voor het genereren van een Spring Boot applicatie

en verwijdert de @Autowired annotatie van de default constructor als dat de enige constructor op die bean is.

De editor bestaat uit twee delen:

- De *actie*: in dit geval verwijderen van constructors met de functie removeAutowiredOnSoleConstructor. Je ziet hier Java termen gebruikt worden als JavaSourceFiles en Constructors; deze editor is *language-aware* en maakt gebruik van de Abstract Syntax Tree (AST) van Java. Je kan dus wijzigingen in je code maken zonder allerlei lastige string parsing. En er zijn ook parsers voor andere talen als Kotlin en TypeScript.
- Definitie van een *command handler*, in dit geval removeAutowiredOnSoleConstructorCommand. Dit zorgt voor de integratie met Slack; door de Atomist bot in Slack de opdracht @atomist remove unnecessary autowired te geven (de *intent* property verwijst naar de naam van het commando) start je deze editor. Voor de complete code verwijzen we naar het Spring automation project in de referenties.

Je eigen automation clients bouwen

Tot nu toe hebben we alleen twee out-of-the-box automation clients gezien. Het creëren van je eigen automation client is eenvoudig. Geef Atomist in Slack de opdracht:

```
@atomist generate automation
```

Dit zet een out-of-the-box generator aan het werk die in een nieuwe GitHub repository de basis voor jouw nieuwe automation client neerzet. Het project kun je nu clonen en bouwen met npm install. Het resultaat dat in de repository landt, staat in **Listing 3**.

Deze client haalt op basis van een GraphQL query ("*graphql/person*"), getoond in **Listing 4**, extra informatie op over de gebruiker die de client aanroept. Hierna checkt de client of deze persoon inderdaad bekend is in Slack. Als laatste stap wordt de *messageClient* API aangeroepen met een bericht "Hello..." met daarin de naam die als parameter meegegeven is, samen met de voor- en achternaam van de gebruiker die het commando *@atomist hello <name>* heeft gegeven.

GraphQL is een querytaal die Atomist gebruikt om met je buildstraat te praten. In de query in **Listing 4** worden de variabelen $teamId en $slackUser gebruikt om meer informatie uit Slack te halen over de gebruiker die de *hello* automation client heeft aangeroepen. Voor meer informatie over de mogelijkheden van GraphQL, zie de tutorial op de GraphQL-site: **http://graphql.org/learn/**

Je start de automation client lokaal met npm start. De automation client registreert zichzelf als skill op jouw Slack channels via security

GRAPHQL IS EEN QUERYTAAL DIE ATOMIST GEBRUIKT OM MET JE BUILDSTRAAT TE PRATEN

```
const Constructors = `//classBodyDeclaration[//constructorDeclaration]`;

export const removeAutowiredOnSoleConstructor: SimpleProjectEditor = p => {
    return findMatches(p, JavaFileParser, JavaSourceFiles, Constructors)
        .then(constructors => {
            if (constructors.length === 1
             && constructors[0].$value.includes( searchString: "@Autowired")) {
                constructors[0].$value =
                constructors[0].$value.replace(/@Autowired[\s]+/, "");
            }
        }).then(() => p.flush());
};
export const removeAutowiredOnSoleConstructorCommand: HandleCommand =
    editorHandler(() => removeAutowiredOnSoleConstructor,
        BaseEditorOrReviewerParameters,
        "RemoveAutowiredOnSoleConstructor",
        {
            description: "Remove @Autowired on sole constructor",
            tags: SpringBootEditorTags,
            intent: "remove unnecessary Autowired",
        },
    );
```

Listing 2: editor voor verwijderen van overbodige annotaties in een Spring applicatie

tokens. Die bepreken we hier verder niet, maar op de site van Atomist worden deze uitgebreid besproken. Clients kunnen ook draaien in Cloud Foundry, of in een Docker container.

Events

Er zijn ook gebeurtenissen in je buildstraat die geen gevolg zijn van een commando van een gebruiker, maar die plaatsvinden door een extern event. *Event handlers* zijn automation clients die dergelijke events afhandelen.

In **Listing 5** staat een event handler die als gevolg van een GitHub push event een notificatie naar Slack stuurt. Met een GraphQL event (verwijzing via *"graphql/push"*, vergelijkbaar met de GraphQL query in Listing 4) stelt de eerdergenoemde *messageClient* API het Slack channel op de hoogte met een notificatie.

Tot besluit

De highlights nog even samengevat:

- Atomist ondersteunt het automatiseren van ontwikkeltaken waarvoor tot nu toe nog geen hapklare tools waren.
- Dankzij 1 point of interaction (Slack) brengt de focus terug en reduceert het voortdurend switchen van context.
- Atomist versnelt het genereren van nieuwe projecten op basis van werkende code.
- Het biedt een groot aantal out-of-the-box automation clients voor issue management en veel voorkomende code-wijzigingen.
- Biedt een TypeScript-based programming model voor je eigen automatisering.

Een uitgebreide lijst met op dit moment ondersteunde buildstraat tools is te vinden op: **https://github.com/janekdb/awesome-atomist** ■

```
@CommandHandler("sends a hello back to the channel", "hello world")
@Tags("hello")
export class HelloWorld implements HandleCommand {
  @Parameter({
      displayName: "Name",
  })
  public name: string;
  @MappedParameter(MappedParameters.SlackUserName)
  public slackUser: string;

  public handle(ctx: HandlerContext): Promise<HandlerResult> {
    return ctx.graphClient.executeQueryFromFile<Person.Query, Person.Variables>(
        "graphql/person", teamId: ctx.teamId, slackUser: this.slackUser })
        .then( onfulfilled: result => {
            if (result && result.ChatTeam && result.ChatTeam[0] &&
                result.ChatTeam[0].members[0] && result.ChatTeam[0].members[0].person) {
                return result.ChatTeam[0].members[0].person;
            } else {
                return null;
            }
        })
        .then( onfulfilled: person => {
            if (person) {
                return ctx.messageClient.respond( msg: `Hello ${this.name}
                from ${person.forename} ${person.surname}`);
            } else {
                return ctx.messageClient.respond( msg: `Hello ${this.name}`);
            }
        })
        .then(success, failure);
  }
}
```

Listing 3: gegenereerde automation client

```
query Person($teamId: ID!, $slackUser: String!) {
  ChatTeam(id: $teamId) {
    members(screenName: $slackUser) {
      person {
        forename
        surname
      }
    }
  }
}
```

Listing 4: GraphQL query graphql/person

REFERENTIES

Atomist platform en documentatie: **https://atomist.com**
Spring automation project: **https://github.com/atomist/spring-automation**
Custom editor voorbeeld: **https://github.com/MichelSchudel/myautomation**
Jessica Kerr, "develop your development automation": **https://www.youtube.com/watch?v=Ztoqv1LNDzc**

```
@EventHandler("notify repo channels when push", GraphQL.subscriptionFromFile( path: "graphql/push"))
@Tags("push", "notification")
export class NotifyOnPush implements HandleEvent<graphql.PushWithRepo.Subscription> {

    public handle(e: EventFired<graphql.PushWithRepo.Subscription>, ctx: HandlerContext):
                  Promise<HandlerResult> {
        return Promise.all(e.data.Push.map( callbackfn: p => {
            if (p.repo && p.repo.channels && p.repo.channels.length > 0) {
                return ctx.messageClient.addressChannels( msg: `Got a push with sha \`${p.after.sha}\``
                    p.repo.channels.map( callbackfn: c => c.name));
            } else {
                return Success;
            }
        }))
            .then(success, failure);
    }
}
```

Listing 5: event handler

Craftsmanship

Checklist voor succes

Wat maakt een developer nou succesvol? Zijn het de developer skills of juist de persoonlijke eigenschappen? Om succesvol te kunnen zijn en om als craftsman gezien te worden, moeten er veel factoren samenkomen in één persoon. Dit artikel geeft inzicht in deze verschillende factoren beschrijft craftsmanship, waarbij vanzelfsprekend ook craftwomanship mee wordt bedoeld.

In de loop der jaren is de software ontwikkelaar geëvolueerd van software developer naar software craftsman. Zijn hiermee de kennis, taken en verantwoordelijkheden veranderd of zijn het slechts synoniemen van elkaar?

Waar vroeger de software ontwikkelaar acht uur per dag achter het beeldscherm kon zitten, zit de dag van een craftsman tegenwoordig juist vol met stand-up meetings, event storming, retrospective, meetups, etcetera. Een ontwikkelaar heeft tegenwoordig veel meer in huis dan enkel software skills.

Persoonlijke eigenschappen

Er wordt veel verwacht van een craftsman, te beginnen bij zijn/haar persoonlijke vaardigheden. Waarom bereikt één persoon meer dan de ander? Het is de combinatie van de persoonlijke eigenschappen, die iemand effectief kunnen maken. Als je schitterende ideeën hebt, maar niet de executiekracht hebt om ze uit te voeren, dan is het lastig om toch succesvol te zijn. In een team kunnen elkaars tekortkomingen gecompenseerd worden, maar een craftsman is ook als individu succesvol.

(Bewust) risico's nemen

Een craftsman moet risico's durven nemen. Kom met innovatieve en weldoordachte voorstellen en wees voorbereid op negatieve terugkoppeling. Dat is spannend en uitdagend, maar laat je daar niet door weerhouden. Dit zijn de momenten waarom je craftsman bent en waar je van leert, ongeacht het resultaat.

Wat erg goed werkt, is disruptief denken. Leg de lat hoog qua doelen stellen en geloof daar heilig in! Ik geloof erin dat jouw omgeving zich naar de doelen gaat gedragen op het moment dat je deze geloofwaardig kunt overbrengen. Neem daarbij geen risico's om het risico nemen, maar doe dit wel met een bepaald idee: het is van grote toegevoegde waarde voor de software en/of de gebruikers.

Doorzettingsvermogen hebben

Ben je iemand die doorgaat, óók als het even tegenzit? Maak je altijd af waar je aan begint, ondanks de risico's? Om een (succesvolle) craftsman te zijn, moet je over een flinke dosis doorzettingsvermogen beschikken. Je moet veel van jezelf kunnen eisen, je goed kunnen concentreren, een helder doel voor ogen hebben en dat ook willen bereiken. Intrinsieke motivatie is hier essentieel.

Intrinsieke motivatie is het ultieme middel om je doel te bereiken. Het probleem hierbij is dat je intrinsiek gemotiveerd bent, of je bent het niet. Je kunt dat niet aanzetten. Je zou doelen/resultaten moeten formuleren waar jij intrinsiek gemotiveerd voor bent.

Creatief zijn

Je hebt creativiteit nodig om jouw idee op zo'n manier uit te bouwen, dat het leidt tot een sublieme oplossing. Kun je verder én "out of the box" denken? Heb je het in je om jouw idee te voorzien van een extra laag, die voor dat "yes!"-gevoel zorgt? Dan zit het

Koen Aerts is werkzaam binnen Ordina JTech. Sinds 1999 is hij actief in de ICT-sector, eerst zelf als software ontwikkelaar en momenteel aan de organisatorische kant.

met jouw creativiteit waarschijnlijk wel goed. Je zult het nodig hebben, ook om anderen te kunnen overtuigen van je plannen.

Overtuigingskracht

Klopt jouw argumentatie altijd en weet je anderen daarmee ook te overtuigen? Oftewel: ben je in staat ideeën te verkopen en anderen daarbij te enthousiasmeren? Het is belangrijk dat je duidelijk kunt maken wat je wilt (en waarom) en daarbij vertrouwen kunt scheppen.

Voor dat laatste heb je ook empathisch vermogen nodig. Mensen goed in kunnen schatten, goed luisteren en tijdens een gesprek in staat zijn om snel te achterhalen wat de behoeften van de gebruikers zijn.

Resultaatgericht werken

Hou strak voor ogen welke resultaten je wilt behalen en handel daar ook naar. Maak snel beslissingen en houd altijd de grote lijnen in de gaten. Wees helder richting je team welke resultaten jij geformuleerd hebt. Hoe kan jouw team hierbij helpen? Als zij het doel helder voor ogen hebben, mogen ze de details zelf invullen.

Craftsmanship manifesto

Developers skills staan beschreven in een heus Software Craftsmanship Manifesto, die iedere ontwikkelaar kan ondertekenen. Bijna 25.000 software craftsmen hebben dit manifesto ondertekend. Dit betekent dat zij zich voor-nemen "*well-crafted software, steadily adding value, a community of professionals en productive partnerships*" te maken. Via de website **http://manifesto.softwarecraftsmanship.org/** kun jij ook jouw commitment geven aan dit manifesto.

Well-crafted software

Iedere craftsman is trots op zijn/haar software! Hij/zij is als een smid, die een zwaard vervaardigt. Iedereen herkent een goede smid aan de zwaarden. Bij software craftsman is dit niet anders. Zorg ervoor dat je nooit onder druk gezet wordt, waardoor je niet meer trots kunt zijn op het product dat je maakt. Als jij er niet trots op bent, dan is het niet van de juiste kwaliteit en straalt dat af op jouw vakmanschap.

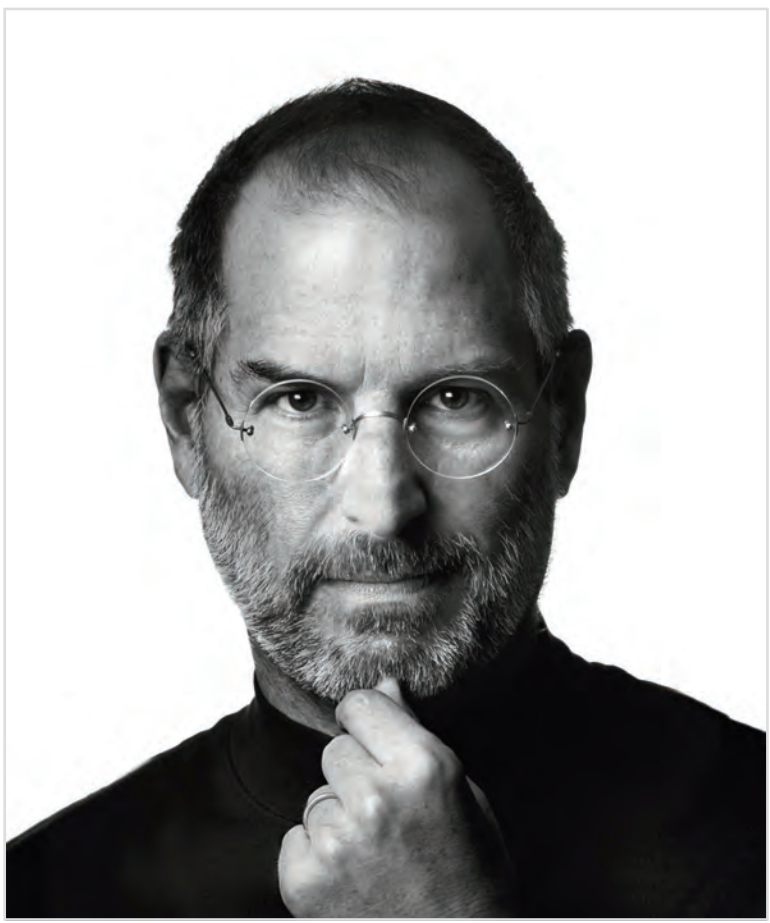
Zorg dat je trots bent op de software, maar ook op het proces om er te komen. Laat zien dat je efficiënt en effectief software kunt ontwikkelen door een innovatieve software-straat, waarin veel geautomatiseerd is. Deze straat stelt jou in staat om keer op keer snel software op te leveren.

Steadily adding value

Je kunt je steeds meer focussen op de gebruikers, die ondersteund moeten worden. Waarom wordt deze software gemaakt? Stuur aan op outcome in plaats van output. Output zijn louter regels code of schermenm die gemaakt worden, maar outcome is de toegevoegde waarde voor de business. Waar zijn de gebruikers nu echt mee geholpen? Stel je hier kritisch op en vraag door op het moment dat die toegevoegde waarde niet helemaal duidelijk is.

A community of professionals

Een craftsman heeft een groot netwerk. Dit netwerk gebruikt hij/zij om zijn/haar kennis te delen en ideeën te toetsen. Op het moment dat je geeft aan de community, dan krijg je al deze energie terug. Nieuwe



"Zie kansen in je eigen tegenslagen" - Steve Jobs

kansen ontstaan en jouw skills stijgen snel. Kom uit je comfortzone en verleg je grenzen! Sta open voor kritiek, want dat is enorm leerzaam!

In de Java- en open source community zijn er mogelijkheden te over om dit netwerk op te bouwen. Er zijn ontelbaar veel open source projecten, conferenties, meetups en blog/Twitter mogelijkheden. Een craftsman is actief op al deze gebieden. Kennis brengen en halen is de key!

Productive partnerships

Vanzelfsprekend kent een craftsman de mogelijkheden wat betreft tooling, frameworks en libraries. Of in ieder geval de manieren om hiernaar te zoeken. Daarnaast beschikt hij/zij ook over waardevolle partners. Samenwerking met cloud providers, bedrijven die software packages beschikbaar stellen en natuurlijk een netwerk om alles te maken, waar hij/zij niet goed in is (bijvoorbeeld UX design). Investeer in dit netwerk en onderhoud het goed, want het is continu in beweging.

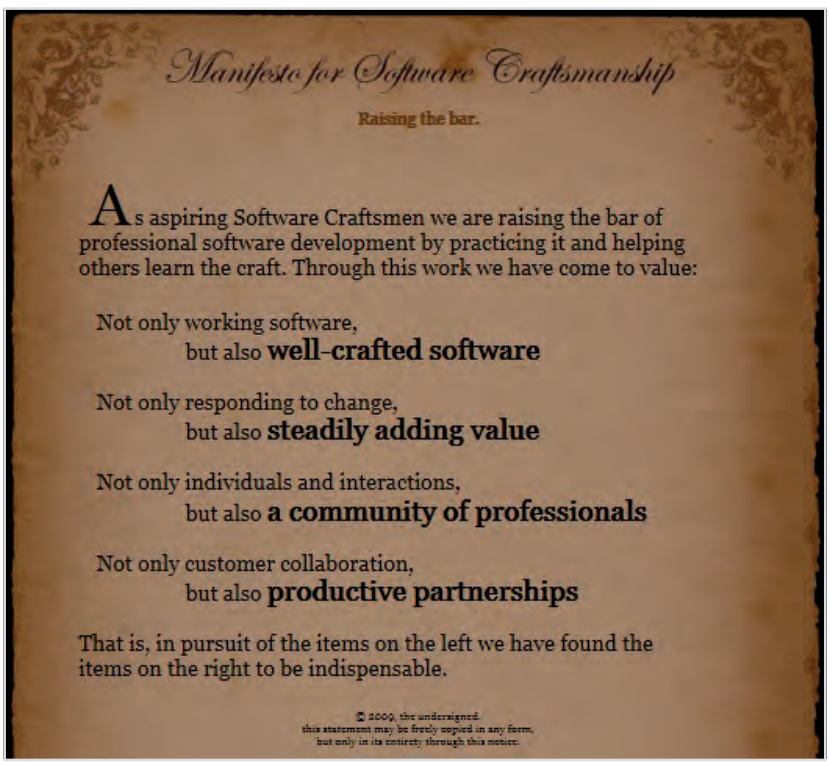
Conclusie

Als alle vaardigheden en eigenschappen, die hierboven beschreven staan, samenkomen in één persoon, dan is succes gegarandeerd.

Helaas ontbreken er bij de meeste personen wel een paar van deze eigenschappen. Dit onderkennen en er naar acteren is belangrijk! Het is namelijk gemakkelijk om dit door een collega of heel team te laten compenseren. Wees je ervan bewust en maak het bespreekbaar.

Sleutelwoorden uit dit artikel zijn disruptief denken, intrinsieke motivatie en een solide netwerk. Stel dus vandaag nog een mooi disruptief doel voor jezelf, waarbij je intrinsieke motivatie voelt.

En als laatste: het solide netwerk. Tegenwoordig gaat het netwerk veel verder dan organisatie en collega's. Dit is belangrijk, want je kunt hierdoor in andere keukens kijken en tunnelvisie doorbreken. Geef jouw energie aan de open source community en je krijgt het direct terug. ■



Manifesto for Software Craftsmanship

Raising the bar.

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software,
but also **well-crafted software**

Not only responding to change,
but also **steadily adding value**

Not only individuals and interactions,
but also **a community of professionals**

Not only customer collaboration,
but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

KENNIS BRENGEN EN HALEN IS DE KEY!

Terugblik op de eerste editie van JVMCON

Een zeer geslaagd concept!

Op 30 januari 2018 was het dan eindelijk zover! Na anderhalf jaar voorbereiding vond de eerste editie van JVMCON in Cinemec Ede plaats. Alweer Cinemec Ede? Jazeker, door de bioscoopopstelling heb je daar goed zicht op het scherm en de spreker. Ook ligt Ede lekker centraal en is het goed betaalbaar.

De insteek van JVMCON is uniek. Het grootste gedeelte van de sessies is door het publiek gekozen! We hebben 177 sessievoorstellen ontvangen van vele sprekers en spreeksters. Alle bezoekers die een early bird ticket besteld hadden, konden vervolgens iedere sessie met maximaal vijf sterren beoordelen. Die beoordeling gaven ze puur om de titel en de omschrijving van de sessie. Ze konden verder niet zien wie het voorstel had ingediend. We geloven in gelijke kansen en vinden de inhoud het belangrijkste. Uiteindelijk hebben we de lijst gesorteerd en hebben we de best beoordeelde voorstellen opgenomen in het programma. Het selecteren van sessies ging nog nooit zo gemakkelijk. Met dank natuurlijk aan de bezoekers die het zware werk voor ons gedaan hebben!

Daarna werd het spannend. We hadden een gaaf programma dat was verspreid over drie zalen, maar zouden we ook bezoekers krijgen? Voor mij was dat nog de meest zenuwslopende periode van de conferentie. De laatste weken voor de conferentie was ik eigenlijk best relaxed. Zeker toen we de dag voor JVMCON zagen dat het laatste kaartje verkocht was en we officieel uitverkocht waren. Met 336 bezoekers verspreid over drie zalen en een algemene ruimte was het gezellig druk en was er plek genoeg voor iedereen.

Het concept sloeg aan en dat bleek ook wel door één van de 'klachten' die ik gedurende JVMCON hoorde: 'Waarom zijn er zoveel goede sessies tegelijk?'. De sessies waren over het algemeen dan ook erg goed beoordeeld, dus blijkbaar hebben jullie de juiste sessies uitgekozen.

Het zal jullie wellicht niet verbazen dat de beste beoordeelde sessie van Venkat was, met een 2e plek voor Hadi Hariri en een 3e plek voor Tom Cools. Het mooie was dat er allerlei soorten sprekers waren, van internationale topsprekers tot sprekers die hun eerste conferentie sessies presenteerden. Het was gaaf om te zien dat ze allemaal hun visitekaartje af hebben gegeven.

Voor meer informatie -inclusief foto's en de aftermovie- kun je terecht op:
http://www.jvmcon.com

Johan Janssen
is trainer bij Info Support en bedenker/organisator van JVMCON.

Full-stack reactive

De kracht van Observables

Ontwikkelaars krijgen steeds meer tools aangereikt om asynchrone afhandeling van applicatielogica mogelijk te maken. Asynchrone programmalogica biedt voordelen t.o.v. synchrone code, omdat efficiënter gebruik kan worden gemaakt van de beschikbare resources. Zo kan je bijvoorbeeld een database query uitvoeren en terwijl je wacht op het resultaat een webservice request kunnen uitvoeren.

Er zijn verschillende mogelijkheden om asynchrone code te schrijven. Voor javascript ontwikkelaars is het bijvoorbeeld vanzelfsprekend gebruik te maken van callback-functies. Sinds versie 8 bestaat deze mogelijkheid ook in Java in de vorm van Lambda statements. Ook kan gebruik gemaakt worden van multithreading, van patterns zoals een eventloop, van co-routines (nodejs, kotlin) en van libraries als reactive extensions (RxJava, RxJS). Asynchrone afhandeling is vooral voordelig wanneer we te maken hebben met interactie tussen applicaties. Het meest populaire protocol hiervoor is REST. Gebaseerd op HTTP is dit een synchroon protocol.

In dit artikel kijken we aan de hand van een voorbeeld naar een alternatief in de vorm van websockets. We zullen een systeem bouwen met een Angular frontend en een Vert.x backend. Voor de asynchrone afhandeling maken gebruik van Observables in RxJS en RxJava. RxJS en RxJava zijn implementaties van het Reactive Extentions project. De kracht van Observables Het voorbeeld gebruikt websockets, omdat dit een populaire standaard is. Het principe kan echter ook worden toegepast op vergelijkbare protocollen zoals bijvoorbeeld TCP.

Use case

Aan de hand van de volgende usecase zullen we laten zien wat de kracht van Reactive Extentions is. We zullen verschillende operators gebruiken en bespreken. De usecase is gekozen, zodat we te maken hebben met asynchrone data communicatie die over tijd wordt uitgevoerd. Dit is precies het domein waar Reactive Extentions het best tot zijn recht komt. Door gebruik te maken van RxJS en RxJava houden we de code compact en leesbaar. We maken een webapplicatie waarin gebruikers kunnen zoeken naar films. De hoofdfunctionaliteit bestaat uit een tekstveld waar zoektermen ingetikt kunnen worden. De applicatie toont vervolgens direct de gevonden films, terwijl ze uit de database worden gestreamd.

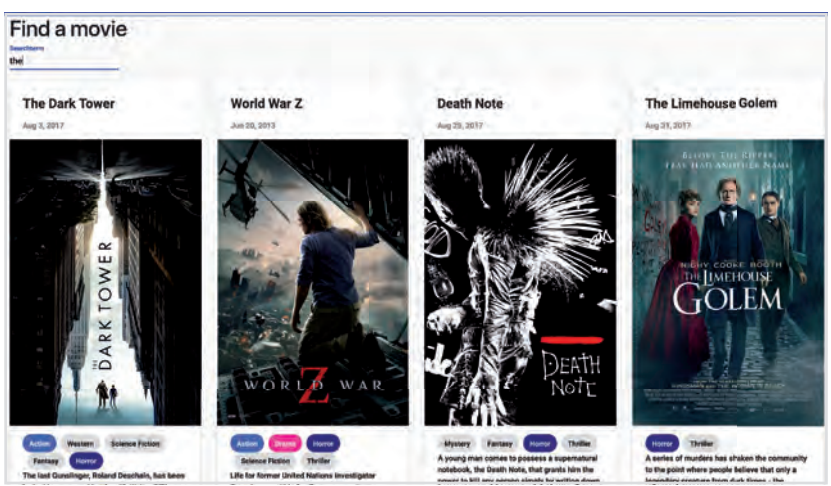
De filmdatabase waar de applicatie gebruik van maakt, kan veel films bevatten. Een zoekresultaat kan dus potentieel veel resultaten opleveren. Het resultaat zal asynchroon naar de browser moeten worden gepushed, zodat de gebruiker niet lang hoeft te wachten totdat films op het scherm verschijnen. Elke keer wanneer de gebruiker een letter intikt, zal het zoekresultaat moeten worden aangepast. Hierbij is het belangrijk dat de gebruiker geen oude zoekresultaten voorgeschoteld krijgt.

Technologie

Voor het versturen van de zoekterm en het ontvangen van de resultaten zullen we gebruik maken van een Websocket verbinding. Websockets maken het mogelijk om full-duplex asynchrone communicatie uit

Erwin de Gier is Software Architect bij Trifork Amsterdam.

Roy Straub is Software Engineer bij CodeSquad



te voeren. Dit in tegenstelling tot REST, waarbij altijd sprake is van een synchrone Request en Response. Met REST is het voor ons dus niet mogelijk het zoekresultaat naar de browser te pushen in de vorm van een Stream. In dit voorbeeld maken we gebruik van een Angular 4 frontend en een Vert.x 3 backend. Beide frameworks bevatten websocket API's die gebruik maken van Reactive Extensions. In Angular kan je gebruik maken van RxJS en in Vert.x van RxJava. Beide kanten van de lijn maken gebruik van Observables, wat ervoor zorgt dat de frontend en backend API's erg op elkaar lijken.

```
HttpServer server = vertx.createHttpServer(); //1
Observable<ServerWebSocket> wsStreamObservable
      = server.websocketStream().toObservable(); //2
    wsStreamObservable.subscribe( //3
      socket -> { //4
        socket.toObservable() //5
          .map(buffer ->
              Json.decodeValue(buffer.toString("UTF-8"), WSAction.class)) //6
          .filter(action -> action.isSearch()) //7
          .map(action -> action.getBody()) //8
          .filter(searchTerm -> searchTerm.length() >= 3) //9
          .switchMap(movieService::findMovies) //10
          .map(movie -> movie.encode()) //11
          .subscribe(socket::writeTextMessage); //12
      }
    );
```

Listing 1

Backend

De backend heeft voor iedere browser-client een websocket connectie. Via deze connectie komen berichten binnen. In ons geval representeren de berichten een zoekterm voor het zoeken naar films. Elke keer als de gebruiker een nieuwe letter intypt, krijgen we een nieuwe zoekterm binnen. Het is dus zaak dat deze zoekacties snel achter elkaar kunnen worden uitgevoerd. Hiernaast is het nodig elke vorige zoekactie te annuleren als een nieuwe zoekactie voor dezelfde client binnenkomt. Dit doen we om te voorkomen dat de gebruiker oude resultaten ziet bij zijn nieuwe zoekterm. Het volgende code snippet toont de belangrijkste functionaliteit voor het verwerken van deze zoekacties (zie **Listing 1**).

Op regel 1 maken we een nieuwe Vert.x HTTP server aan. Deze kan HTTP(s) en Websocket verbindingen voor verschillende clients faciliteren. Op regel 2 vragen we de websocket stream op voor deze server. Deze transformeren we tot een Observable. Dit maakt het mogelijk ons te abonneren op nieuwe websocket connecties (wsStreamObservable.subscribe, regel 3). Per client krijgen we een nieuwe socket binnen (regel 4). Deze socket representeert de verbinding tussen de server en de client. Via deze socket ontvangen we dus berichten en kunnen we berichten terugsturen. Hiervan maken we wederom een Observable (regel 5). Tot zover het opzetten van de verbindingen. De berichten komen binnen als bytes. Deze gaan we eerst vertalen naar een JSON representatie, welke we vervolgens direct omzetten naar een Java object. Hiervoor maken we gebruik van de Rx map operator (regel 6). De map operator voert een functie uit op elk van de items en retourneert het resultaat als een Observable. In dit geval komt voor elke zoekactie een buffer instantie binnen die wordt omgezet naar een WSAction object, zie hiervoor het marble diagram in **Diagram 1**. Marble diagrams zijn visuele representaties van functionele operators.

Vervolgens willen we alleen zoekacties gaan afhandelen. Het is mogelijk dat we via dezelfde websocket connectie andere acties binnen krijgen. We willen in onze stream alleen de zoekacties doorlaten en andere acties negeren. Hiervoor gebruiken we een filter (regel 7). Een filter laat alleen items door die voldoen aan het meegegeven predicaat. In dit geval is dat action.isSearch(). Dit resulteert weer in een nieuwe Observable, die alleen de doorgelaten items bevat.

We willen van deze zoekactie de body hebben, deze bevat namelijk de zoekterm. Hiervoor gebruiken we wederom een map operator(8). Om ervoor te zorgen dat we alleen zoekacties van 3 of meer karakters verwerken, gebruiken we wederom een filter (9).

Als de gebruiker een nieuw karakter intikt, krijgen we via de websocket verbinding een nieuwe zoekterm binnen. Op dat moment zijn we nog de vorige zoekacties aan het verwerken. Deze moeten we annuleren. Als je dit zou implementeren zonder Observables, dan zou de code eruit kunnen zien als in **Listing 2**.

In deze implementatie van de zoekfunctionaliteit houden we een map van subcriptions bij op basis van het textHandlerID. Dit ID representeert een verbinding met een client. We moeten hier dus expliciet controleren of er een

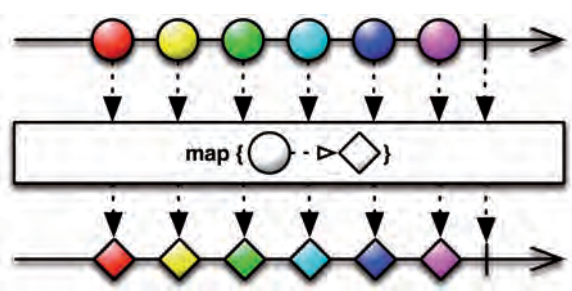


Diagram 1: map

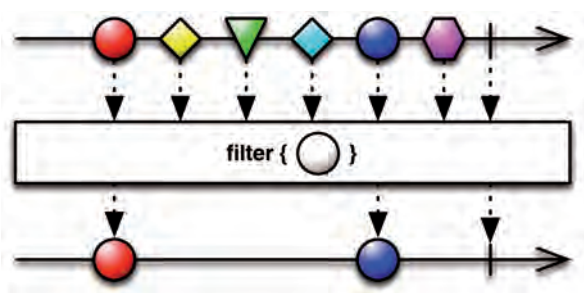


Diagram 2: filter

bestaande zoekactie is (1). Als deze bestaat, roepen we hier unsubscribe aan, zodat de actie wordt geannuleerd (2). We moeten vervolgens een nieuwe Subscription aanmaken (3) en deze aan de map toevoegen (4). Hierdoor introduceren we een expliciete variabele waarin we state moeten bijhouden, de map met subscriptions.

De switchmap kan dit voor ons afhandelen. Een switchmap subscribed op een Observable, in ons geval op het resultaat van het filter op regel 9 van **Listing 1**. Voor elke zoekterm beginnen we met het ophalen van films door het aanroepen van de findMovies methode op onze movieService. Deze findMovies geeft een Observable van films terug in JSON formaat (**Listing 3**). Per zoekterm krijgen we dus meerdere films binnen over tijd. Op het moment dat een nieuwe zoekterm binnenkomt uit de Observable van het filter in regel 9, dan zorgt de switchmap ervoor dat een unsubscribe plaatsvindt op de Observable, die weer terugkomt uit findMovies methode en die behoorde bij de oude zoekterm. De findMovies methode wordt opnieuw uitgevoerd met de nieuwe zoekterm en er vindt een subscribe plaats op de nieuwe Observable die de films bevat die horen bij deze nieuwe zoekterm.

Tot slot gebruiken we nogmaals een map operator om de films om te zetten naar een JSON formaat wat we als websocket response kunnen teruggeven. Het terugsturen van de films over de websocket verbinding doen we in de subscribe methode (regel 12).

Front-end

Nu we een mooie api hebben staan, moeten we uiteraard nog films gaan tonen aan de gebruikers. Zoals genoemd, gebruiken we hiervoor Angular 4 in combinatie met TypeScript. Het project is gegenereerd met de Angular CLI[1] . We hanteren daarnaast zoveel mogelijk de officiële styleguide van Angular[2].

De front-end bestaat uit verschillende pagina's, wat onze container componenten zijn. De meeste actie

```
WSAction action = null;

try {
  action = Json.decodeValue(message, WSAction.class);
} catch (DecodeException e) {
  ws.writeTextMessage(new JsonObject()
          .put("status", "invalid request").encode());
}

if (action != null && action.isSearch()) {
    Subscription existingSearch = subscriptions.get(ws.textHandlerID());
    if (existingSearch != null) { //1
      existingSearch.unsubscribe(); //2
    }

    Subscription newSearch
      = movieService.findMovies(action.getBody()).subscribe(movie -> { //3
      ws.writeTextMessage(movie.encode());
    });

    subscriptions.put(ws.textHandlerID(), newSearch); //4
}
```

Listing 2

```
public Observable<JsonObject> findMovies(String keyword);
```

Listing 3

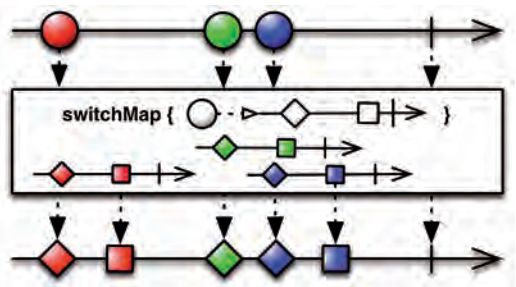


Diagram 3: switchmap

vindt plaats op de zoekpagina. Hiervoor is een search component aangemaakt. Op de zoekpagina vinden we een tekstveld waarmee we uiteindelijk searchterms over de websocketconnectie versturen naar de backend. De resultaten die terugkomen, renderen we netjes met een apart component.

Het eerste wat we moeten doen, is het opzetten van een websocket connectie. In RxJS is hiervoor een speciale operator, namelijk de **webSocket** operator. Deze is te vinden in een speciale **dom** folder in de rxjs module. Op regel 1 van **Listing 4** is te zien dat we de websocket laten verwijzen naar de opgezette websocket in de backend. De websocket is een speciale Observable. Hij fungeert namelijk zowel als een Observable (dus een bron van data), maar ook als Observer (consument van data). Zo'n Observable die ook Observer interfaces implementeert, noemen we ook wel een **Subject**. De conventie is tevens om Observables te denoteren met eindigend met een $.

Het tweede wat we moeten doen, is luisteren naar veranderingen in het zoekveld (in de code searchTerm). Dit zetten we op, op het moment dat het component in de DOM wordt gezet. In Angular gebruiken we hiervoor de **OnInit** lifecycle hook van een component (zie **LIsting 5**). Vervolgens laat Angular nog meer krachtige mogelijkheden zien. We kunnen namelijk met regel 1 code een Observable krijgen van het zoekveld met **valueChanges**. Dit werkt overigens alleen met een **FormControl**.

Nu kunnen we natuurlijk direct waardes over de lijn sturen, maar we kunnen het veld ook nog wat slimmer maken. Een eerste stap die we hierin maken, is door alleen te acteren op drie of meer karakters. Hiervoor gebruiken we de reeds bekende **filter** operator (zie regel 2). Een verdere optimalisatie, is bijvoorbeeld door alleen

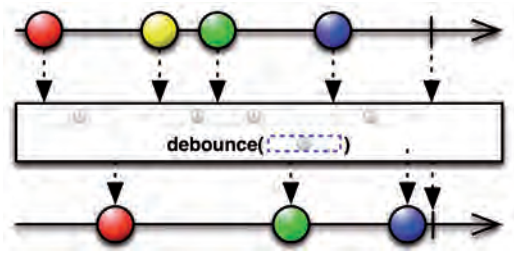


Diagram 4: debouncetime

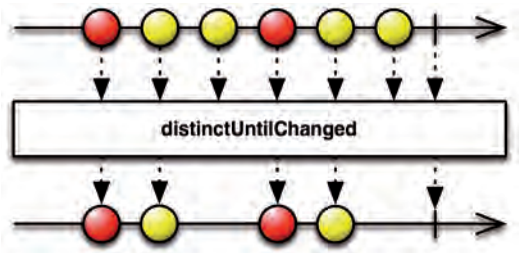


Diagram 5: distinctUntilChanged

```
private socket$ = Observable.webSocket('ws://localhost:8080'); // 1
```

Listing 4

```
ngOnInit() {
  this.searchTerm.valueChanges                              // 1
    .filter(searchTerm => searchTerm.length >= 3)           // 2
    .debounceTime(300)                                      // 3
    .distinctUntilChanged()                                 // 4
    .subscribe(searchTerm => this.search(searchTerm));      // 5

    // Setup for websocket results
}
```

Listing 5

```
search(searchTerm: string) {
    this.movies = [];          // 1
    this.socket$.next(         // 2
      JSON.stringify({         // 3
        action: 'search',      // 4
        body: searchTerm       // 5
      })
    );
  }
```

Listing 6

een zoekterm te versturen als we kunnen aannemen dat de gebruiker klaar is met typen. Hiervoor kunnen we de **debounceTime** operator toepassen (zie regel 3). Hiermee geven we aan dat we pas verder willen gaan met de data uit de Observable als minstens 300ms geen nieuwe data uit de Observable komt.

We kunnen de client nóg efficiënter maken door alleen zoektermen te versturen die ongelijk zijn aan de vorige verstuurde zoekterm. Aangezien deze toch dezelfde resultaten zullen opleveren. Ook hier heeft ReactiveX een operator voor. Dat is de zogenaamde **distinctUntilChanged** operator. Hierna subscriben we op deze hele ketting aan operatoren en gaan we de zoekterm afhandelen (zie regel 4).

Elke keer als we een nieuwe zoekaktie starten voor een nieuwe zoekterm, maken we de resultaten van de vorige zoekaktie leeg. Op regel 2 zien we dat we met de **next** methode een nieuwe waarde over de websocket sturen (zie **Listing 6**). We converteren op regels 3 t/m 5 een nieuw object met de gespecificeerde zoekterm en actie tot een string die door de backend wordt geaccepteerd.

De afhandeling van de resultaten zetten we ook op in de component initialisatie. We subscriben op de socket en voor elk stuk data kijken we of het een film betreft. Op regel 2 van **Listing 7** is dit te zien met behulp van wederom een **filter**. Vervolgens zetten we de resultaten in de movies array op regel 4. Dit wordt in de view direct gerenderd door Angular's templating engine. Tot slot maken we op regel 5 en 6 een basale error respectievelijk completion afhandeling aan.

Conclusie

Ondanks de vrij grote mate van complexiteit in het opzetten en afhandelen van alle asynchroniteit in de applicatie, is de code zeer goed te volgen. Het is compact, to the point en semantisch redelijk vanzelfsprekend. Het afhandelen van een datastream, door middel van een pipe waar de data doorheen gaat, is krachtig en gemakkelijker om mee te werken dan bijvoorbeeld callbacks. Daarnaast geeft het scheiden van databron (Observable) en data-consument (Observer) ons flexibiliteit. Het is op deze manier mogelijk meerdere en verschillende afhandelingen voor eenzelfde databron te definiëren. Ook geeft het ons de mogelijkheid eenzelfde API te gebruiken in zowel de front-end (RxJS) als de backend (RxJava). Bekend zijn met de ReactiveX operators heeft hier dus voor zowel front-end als backend werk toegevoegde waarde.

ReactiveX is al met al een goede toevoeging aan de toolkit van elke developer. De mogelijkheden en toepassingen hiervoor zijn eindeloos. Daarnaast is het zeer goed gedocumenteerd. Denk de volgende keer bij een project met asynchroniteit dus zeker aan de Reactive Extensions library!

De volledige source code van dit project is te vinden op: **https://github.com/erwindeg/ultimate-async-stack**. ■

```
ngOnInit() {
    // Search field observable setup

    this.socket$                                          // 1
      .filter(movie => movie.hasOwnProperty('_id'))       // 2
      .subscribe(                                         // 3
        movie => this.movies.push(movie as Movie),        // 4
        err => console.log('error:', err),                // 5
        () => console.log('complete')                     // 6
      );
  }
```

Listing 7

REFERENTIES

[1] https://cli.angular.io

[2] https://angular.io/guide/styleguide

Foto: Frans van Bragt

.nl. jug

J·SPRING nl jug

J-Spring: een NLJUG Java conferentie

31 mei 2018

TivoliVredenburg - Utrecht

Bestel nu je tickets op: http://jspring.nl/tickets/

Bert Ertman
Director of Technology Outreach @ Luminis

Daniel Bryant
Independent Technical Consultant @ InfoQ

Peter Hilton
Consultant @ Signavio

Martijn Verburg
CEO @ jClarity

Sebastiaan Daschner
Author of Architecting Modern Java EE Applications @ IT-Beratung

Simon Maple
Java Champion

Siren van der Hof
CSO @ Min Doktor

Roy van Rijn
Software Craftsman @ JPoint

Sven Peters
Evangelist @ Atlassian

Emond Papegaaij
Java Ontwikkelaar Topicus Security

Ray Tsang,
Technology Architect

CO-SPONSOREN

topicus

Rabobank

EXPOSANTEN

vijfhart IT-OPLEIDINGEN

sultansofjava

YACHT

CarthagoICT CONSULTING. ICTPROJECTS. WEBSOLUTIONS.

CGI

GROUP9 ELITE JAVA DEVELOPMENT

OVSOFTWARE

Profit4Cloud professionals in cloud engineering

Rijksoverheid

Developing tomorrow's bank today

Rabobank is looking for new IT Colleagues

With our IT team you will be at the heart of tomorrow's Rabobank. Whether it is our Rabobank App, used by over 3 million customers, our responsive web platform or the latest apps and widgets we're creating for smartwatches, you will be improving and mindshifting the way our customers interact with Rabobank.

Being part of this means you have the experts around you to learn, the less experienced to coach and space for your experience to flourish. We believe in having you in the driver's seat and guiding us to the future. Join us and become part of one of the biggest IT teams within the financial market where your work and your next step is all about you.

Interested in our opportunities? Check rabobank.nl/ict

Rabobank

Nieuw dit jaar: Profit4Cloud op J-Spring 2018

Java, Cloud en food.

Op 31 mei barst het weer los: de latest and greatest in Java-land. Ditmaal op een nieuwe locatie: Tivoli in Utrecht met een nieuwe exposant: Profit4Cloud. Als business-partner van NLJUG konden we haast niet ontbreken op J-Spring. Vandaar. Onze Java-adepten gaan deze dag natuurlijk een aantal sessies bijwonen terwijl een aantal andere collega's in de stand aanwezig zal zijn om mee te kunnen sparren over zaken rondom Java en Cloud. Want dat je in Java mooie applicaties slim en efficiënt kunt ontwikkelen met behulp van Cloud is een feit.

In ons nog jonge bestaan vanaf begin 2015 hebben we al een mooie track-record opgebouwd van projecten waaraan we hebben meegewerkt. Voor grote wereldspelers maar ook MKB-bedrijven en start-ups. Samen met onze 40 collega's bedenken we elke dag mooie en slimme oplossingen voor de toepassingen van morgen. En dat doen we niet alleen op locatie bij onze opdrachtgevers maar ook intern op onze kantoren in Amsterdam, Apeldoorn en Eindhoven. En graag geven wij onze kennis terug aan de community: samen met NLJUG organiseren wij twee keer per jaar een Tech Update.

Vragen over Java en Cloud? Bezoek dan onze stand.
Wij zorgen voor een goed gesprek en een gevulde maag.

Wil je meer weten over ons? Check dan ook eens www.profit4cloud.nl

Profit4Cloud
professionals in cloud engineering

Spring and Azure

Options for Java cloud developers

Spring is here and it only seems like the right time to talk about spring.io, and all the things you can do with Spring on Azure. At Microsoft, we do not stop at offering a great platform for cloud-native apps. We also work with partners and the Java community to offer the most productive tools for developers to build agile applications across multiple environments. With that goal in mind, we love working with Pivotal in supporting their efforts and offerings for spring and Cloud Foundry. Here are some of the results of our partnership.

Azure Services in Spring Initializr

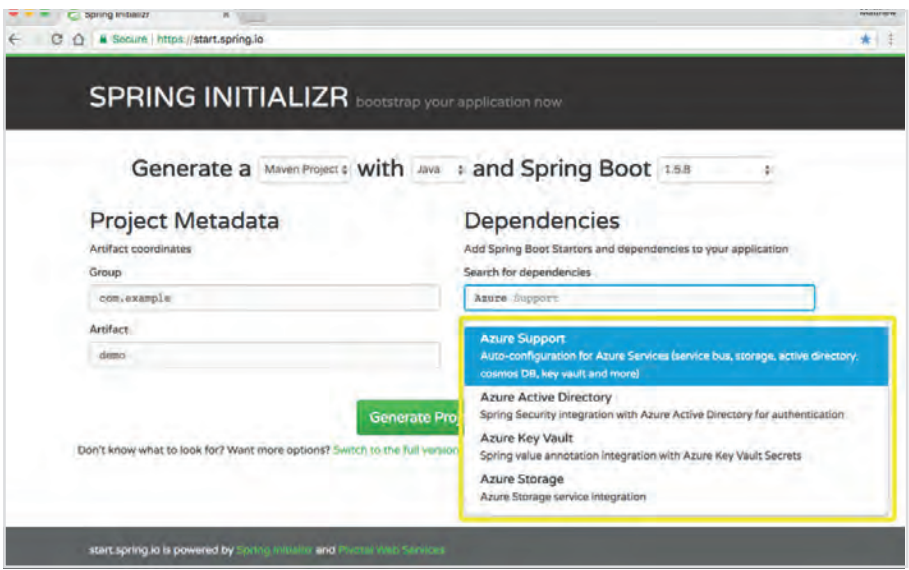
Last December we created the **Spring Boot Starters for Azure**, providing Java developers a shortcut to apply Spring technologies to Azure. They're available on the Spring Initializr (at **start.spring.io**) and enable developers to handle the dependency management and make the bootstrapping process for Spring projects with Azure much easier.

If you're depending on the latest Spring Boot support for your projects, you;ll be happy to know that Azure Starters were recently been updated to support Spring Boot 2.0.

Java developers can now get started with their Spring applications on Azure quickly by typing "Azure" inside **Spring Initializr** to choose the dependencies for Azure services, or by selecting options they want to include from the new Azure section on the full version of the site. You can also access Azure dependencies from the cloud foundry CLI, Visual Studio Code, Eclipse, and IntelliJ. All of our Spring Boot support is open source and on **GitHub.**

We are currently offering the following **Spring Starters for Azure**, with more to come:

- **Azure Support:** Provides support for all services currently available via Spring Boot Starters.
- **Azure Active Directory:** Enterprise grade authentication using Azure Active Directory.
- **Azure Key Vault:** Manage application secrets using Azure Key Vault.
- **Azure Storage:** Integration with Azure Storage including object storage, queues, tables, and more.

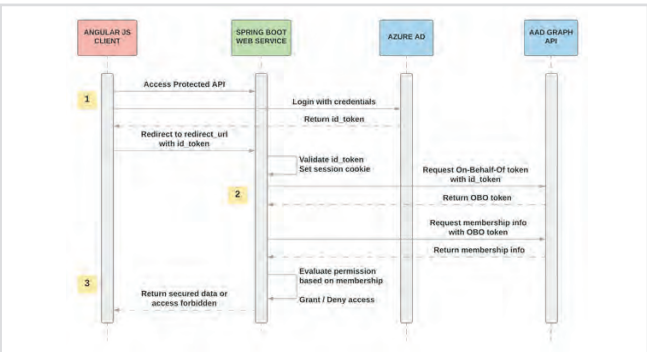


Microsoft

Brian Benz, Senior Cloud Developer Advocate, Java at Microsoft.

Enterprise grade authentication and authorization with Spring Security Azure AD

Here's a sample app that was shown by **Erich Gamma at SpringOne 2017**. It's a great example of integrating Spring Security and Azure Active Directory into your applications **The sample** is composed of two layers: and Angular client and a Spring Boot RESTful web service. It illustrates the flow for login and retrieving user information using AAD Graph API.



Authorization Flow Chart

The authorization flow is composed of 3 phrases:

1. Login with credentials and validate via Azure AD.
2. Retrieve a token and membership information from Azure AD Graph API.
3. Evaluate the membership for role-based authorization.

The demo used **Spring Initializer** to quick-start a new project with dependencies of Spring Security and Azure Active Directory. Once you specify the Azure AD connections and wire up AAD AuthFilter in your project, you can easily **set up AAD authentication and role-based authorization** with the following features:

- @PreAuthorize: Implement Spring's @PreAuthorize annotation to provide method-level security with roles and permissions of logged-in users.
- isMemberOf(): provide access control with roles and permissions based on a specified Azure user group.
- Easily use Azure AD Group for access control by adding or removing group members.

To get started, first **register a new application in Azure Active Directory**. After the app is ready, generate a client key and **grant permissions to the app**.

Spring Boot extensions for Visual Studio Code

We also offer a new set of **Visual Studio Code extensions** that add first-class support for Spring Boot developers. With the new extensions from Pivotal, you get full code-completion support, validation and assistance for application property files, navigation shortcuts, and the ability to inspect running apps. Combine that with Spring Initializr support, and you can quickly initiate, develop, and deploy applications anywhere, including Pivotal Cloud Foundry on Azure and Azure Stack.

Pivotal Cloud Foundry on Azure Stack

For organizations with hybrid cloud needs, Pivotal recently announced **Cloud Foundry support for Azure Stack**. The addition of Azure Stack support enables you to deploy apps written using any language and framework, including Java and Spring, to the public or private Azure cloud securely and easily with Pivotal Cloud Foundry. Pivotal and Microsoft have been working together to offer a consistent experience on premises or on the cloud. For example, Pivotal Cloud Foundry uses the same operations manager and provisioning agent for Azure Stack and the public Azure cloud. You can also use the **Open Service Broker for Azure** in Pivotal Cloud Foundry on Azure Stack to connect to services on the public Azure cloud like **Cosmos DB**, Service Bus, and more.

Lighter image for Pivotal Cloud Foundry on the Azure Marketplace

The Azure Marketplace is a valuable way for you to provision software in the cloud. That's why we're pleased that Pivotal chose the Azure Marketplace as the first place to offer a **new, lightweight image of Pivotal Cloud Foundry**. The *Small Footprint Runtime* offers very similar functionality to the traditional image, but requires 70% fewer VMs. This is great for small environments like proof-of-concepts, edge locations, or departmental solutions that want velocity, but with the smallest infrastructure possible.

Moving forward

We're excited about all of the announcements that are coming from the Microsoft and Pivotal partnership. We're working to make Azure a great platform for all developers, including those working with Java and Spring, enabling them to deploy to virtual machines, container, Functions and third-party platforms like Pivotal Cloud Foundry. We're also enabling first-class hybrid scenarios with Azure Stack.
If you're building apps in Spring, check out our developer hub for **Spring on Azure**, including how to use the new Azure starters in Spring Initializr.

Deploy your Spring apps to Azure

Once you've built your Spring apps, you can easily deploy them to Azure, leveraging services like:

- Fully-managed Kubernetes on **Azure Container Service** (AKS)
- **Linux Virtual Machines** supporting your choice of distributions
- Bring a Docker container to **Web Apps for Containers**
- For more information about using Spring on Azure, visit the following pages:
- GitHub: **Spring Boot Starters for Azure Services**
- Tutorial: **Spring on Azure Homepage**
- Tutorial: **Java support on Azure Homepage**

If you don't have an Azure subscription, you can get started with our **trial offer**, including $200 in Azure credits and 12 months of popular services, for free. ■

TRAINING

BY DEVELOPERS FOR DEVELOPERS

OFFICIAL PARTNER OF PIVOTAL IN THE NETHERLANDS

Ace the Spring Professional Certification by taking the **ONLY** Pivotal certified Core Spring training in The Netherlands. The training is focused on **Spring 5** and led by Trifork's Spring experts at the GOTO Academy in Amsterdam.

UPCOMING COURSE DATES

- May 15-18
- July 24-27
- September 11-14
- November 13-16
- December 18-21

€112 DISCOUNT

USE PROMO CODE
"NLJUGSPRING"
www.gotoacademy.nl

Optional...

What else?!

Optional. Een nieuw concept dat sinds Java 8 is geïntroduceerd. In zijn nog korte levensduur heeft Optional al voor veel opschudding gezorgd. De ene persoon vindt het een prachtige oplossing, de ander vindt het zo lelijk als de nacht. Echter, wat je mening ook is, het is onderdeel van de taal en je zal er hoe dan ook mee te maken krijgen. Ondanks dat er al veel meningen zijn, zijn niet alle meningen gegrond en weet ook niet iedereen wat nu eigenlijk het doel is van Optional. In dit artikel laat ik mijn licht schijnen op Optional. Hierbij worden een aantal valkuilen aangestipt, zodat deze ons niet meer verrassen.

Wat is een Optional?

Optional is op een bepaalde manier de Java implementatie van de Maybe monad. Nu moet je niet gelijk schrikken of afhaken vanwege het gevreesde 'M' woord. Eigenlijk is een monad helemaal niet zo afschrikwekkend en zeker niet de Maybe monad. Een monad is niks meer of minder dan het omsluiten van een type om een specifieke casus of gedrag te bedienen. In dit geval het feit dat het type de waarde *null* kan zijn met een mogelijke *NullPointerException* tot gevolg. Om het te simplificeren, kunnen we het eigenlijk beschouwen als een wrapper die een gebruiker dwingt om te kijken of de waarde wel of niet aanwezig is.

Voor wie niet zo bekend is met Optional hier in vogelvlucht de belangrijkste zaken. Stel dat we in Java een functie hebben die een User teruggeeft. De waarde van User kan null zijn wanneer deze bijvoorbeeld niet te vinden is. Hierdoor moet een gebruiker altijd eraan denken om een null check uit te voeren, voordat we iets aanroepen op de teruggeven User (zie **Listing 1)**.

In plaats daarvan kunnen we een Optional teruggeven. Nu kunnen we niet zomaar iets aanroepen op User, maar zijn we verplicht om eerst de Optional "uit te pakken" (zie **Listing 2**).

```
public User getUser() {
    return user;
}
```

Listing 1

```
public Optional<User> getUser() {
    return Optional.ofNullable(user);
}
```

Listing 2

```
Optional<String> a = Optional.of("Hello World");

Optional<String> b = Optional.empty();

Optional<String> c = null //doe dit niet !!!!!!
```

Listing 3

Door het teruggeven van Optional geef je het signaal richting de gebruiker van de functie dat de waarde mogelijk niet aanwezig is.

Logischerwijs zou je kunnen concluderen dat een Optional dus twee mogelijkheden heeft. Of hij is gevuld of hij is niet gevuld. Echter, we leven in een objectgeoriënteerde wereld. Dit betekent dat het mogelijk is om de waarde null toe te kennen aan Optional (zie **Listing 3**).

Als ik je vandaag maar één ding kan meegeven, dan is dat het volgende. Zorg ervoor dat je nooit, maar dan ook echt nooit, de waarde null toekent aan een Optional in je eigen code!

Brian Vermeer is Senior Software Engineer bij Blue4IT.

Ondanks dat Optional bedoeld is om de NullPointerException te voorkomen, kan het helaas dus nog steeds gebeuren. Het vergt dus nog steeds discipline van de ontwikkelaar om te zorgen dat het mechanisme werkt zoals het zou moeten werken.

Hoe om te gaan met optionals

Oké, leuk. Nu hebben we een Optional en dan? Om de daadwerkelijke waarde uit een Optional te halen, moeten we hem "uitpakken". Dit zou kunnen door simpelweg **`get()`** op de optional aan te roepen. Maar, als de Optional leeg is en we roepen **`get()`** aan, dan krijgen we te maken met een NoSuchElementException. Dit is ook niet iets wat we willen. Daarom zullen we eerst moeten kijken of de Optional überhaupt een waarde bevat voordat we **`get()`** kunnen aanroepen (zie **Listing 4**).

Het aanroepen van **`get()`** zonder te checken of de Optional gevuld is, kan dus leiden tot een Exception. Het zou verboden moeten worden! Daarom ben ik van mening dat de **`get()`** methode in z'n huidige vorm deprecated zou moeten zijn. Daarnaast is de imperatieve stijl van coderen, zoals hierboven, ook geen elegante oplossing. We moeten nu een **`isPresent()`** check uitvoeren waar we voorheen, voor het gebruik van Optional, een null check moesten uitvoeren. Wat mij betreft lood om oud ijzer en geen echte oplossing.

We kunnen Optional ook op een wat meer functionele of declaratieve manier gebruiken. Dit doen we door **`map()`** aan te roepen op de Optional. De lambda die we meegeven aan deze higher-order function zal alleen worden uitgevoerd als de Optional daadwerkelijk een waarde bevat (zie **Listing 5**).

Wanneer we **`execute()`** uitvoeren krijgen we, zoals verwacht, de output:

```
only run if optional is filled
```

Als we de waarde van **`maybeString`** veranderen naar **`Optional.empty()`** en **`execute()`** wordt wederom uitgevoerd, zien we vervolgens dat er geen output is in de console. Dit ziet er al een stuk interessanter en leesbaarder uit.

Alternatieve flows

Prima! Maar hoe moeten we dan omgaan met een alternatieve flow? Als we iets willen doen wanneer de Optional leeg is, dan hebben we drie keuzes:

- orElse()
- orElseGet()
- orElseThrow()

Met de laatste optie is het mogelijk om, met behulp van een supplier, een exceptie te gooien als de Optional leeg is. Dit is mogelijk in combinatie met **`map()`** maar het is niet per definitie nodig (zie **Listing 6**).

Wanneer we **`execute()`** aanroepen in bovenstaand voorbeeld, dan treed -zoals verwacht- een RuntimeException op.

Maar goed, we willen geen exceptie, maar daadwerkelijk een alternatieve functionaliteit toepassen wanneer de Optional geen

```
public String doSomething() {
   Optional<String> foo = Optional.empty();
   if (foo.isPresent()) {
       return foo.get();
   }
   return "";
}
```

Listing 4

```
public void execute() {
   Optional<String> maybeString = Optional.of("foo");
   maybeString.map(this::runIfExist);
}
private String runIfExist(String str) {
   System.out.println("only run if optional is filled ");
   return str;
}
```

Listing 5

```
public void execute() {
   Optional<String> maybeString = Optional.empty();
   maybeString
       .map(this::runIfExist)
       .orElseThrow(() -> new RuntimeException("Optional was empty"));
}
```

Listing 6

```
public void execute() {
   Optional<String> maybeString = Optional.of("foo");
   String newString = maybeString
           .map(this::runIfExist)
           .orElse(runIfEmpty());
   System.out.println(newString);
}
private String runIfExist(String str) {
   System.out.println("only run if optional is filled ");
   return str;
}
private String runIfEmpty() {
   System.out.println("only run if empty");
   return "empty";
}
```

Listing 7

waarde heeft. Onderstaand voorbeeld lijkt op het eerste gezicht vrij voor de hand liggend (zie **Listing 7**).

Als we de code simpelweg van boven naar beneden lezen, dan ontstaat de verwachting dat **`newString`** de waarde **`"foo"`** gaat krijgen. De Optional **`maybeString`** is immers gevuld, dus logischerwijs zou de **`map()`** zijn werk moeten doen. Wanneer we daadwerkelijk **`execute()`** runnen, dan krijgen we de volgende output in de console:

```
only run if optional is filled
only run if empty
foo
```

De verwachting dat **`newString`** de waarde **`"foo"`** zou krijgen, klopt nog steeds. Iets wat we hier nog meer zien, is dat de string **`"only run if empty"`** ook in de console output staat. Dit was iets dat je wellicht niet zou verwachten. Het betekent dus, dat ondanks dat de Optional een waarde heeft, de methode in de **`orElse()`** alsnog uit- gevoerd wordt.

We veranderen de methode **`execute()`** zodat we **`orElseGet()`** gebruiken in plaats van **`orElse()`**. Dit betekent dat we de method call naar **`runIfEmpty()`** ook zullen moeten veranderen naar een supplier (zie **Listing 8**).

Als we nu **`execute()`** aanroepen, zien we het volgende:

```
only run if optional is filled
foo
```

Nu zien we, zoals we in eerste instantie zouden verwachten, dat **`runIfEmpty()`** niet uitgevoerd is. Wat mij betreft, is dit niet echt intuïtief. Je zou eigenlijk verwachten dat dit het algemene gedrag zou zijn, ook bij de normale **`orElse()`**.

Conclusie

Als eerste hebben we gezien dat Optional de waarde **`null`** kan krijgen. Technisch gezien behouden we dus de mogelijk op een **`NullPointerException`**. Echter, met verstandig gebruik en daarmee dus nooit een Optional de waarde null toekennen, kan een hoop ellende voorkomen worden.

Daarnaast zien we dat als we **`orElse()`** gebruiken om alleen een alternatieve

```
public void execute() {
   Optional<String> maybeString = Optional.of("foo");
   String newString = maybeString
            .map(this::runIfExist)
            .orElseGet(() -> runIfEmpty());
   System.out.println(newString);
}
```

Listing 8

waarde toe te kennen wanneer een Optional leeg is, er geen enkel probleem is. Je moet er dan wel absoluut zeker van zijn dat de code in de **`orElse()`** geen enkel side effect heeft. Deze code wordt immers, ongeacht de waarde van de Optional, alsnog uitgevoerd. Over het algemeen is **`orElseGet()`** de veiligere en juiste optie, zeker als we daadwerkelijk logica willen uitvoeren.

Door de karakteristieken van het declaratief programmeren is de mix-up snel gemaakt. Daarnaast zie je het resultaat pas wanneer de code daadwerkelijk uitgevoerd is. Uiteraard zou je goede testcode moeten schrijven, maar het is nog beter om het verschil bij voorbaat te weten. Het verschil is subtiel, maar de impact kan enorm zijn wanneer je niet voorzichtig bent. ■

ZORG ERVOOR DAT JE NOOIT, MAAR DAN OOK ECHT NOOIT, DE WAARDE NULL TOEKENT AAN EEN OPTIONAL IN JE EIGEN CODE

Pragmatisch programmeren

Java of Go?

Al meer dan 20 jaar is Java een aannemelijke keuze om software te ontwikkelen. De taal is gevestigd, de Java Runtime Environment is stabiel en er is een enorme keuze aan kennis en kunde te vinden in de vorm van boeken en libraries, die een ontwikkelaar helpen met het realiseren van een (software-)doel. Door de jaren heen zijn er uitdagers voor Java als techniek. Nieuwe programmeertalen (zoals Ruby) worden populair, maar worden uiteindelijk ook weer vervangen door een volgende hype. Alternatieven (zoals .NET) vestigen zichzelf in een duidelijk deel van de markt en bouwen eigen bestaansrecht op.

Het is als Java ontwikkelaar mooi om te zien hoe de Java community de opkomst van andere 'hypes' oppakt als uitdaging. Hierdoor kunnen we nu ook functioneel programmeren op de JRE en inmiddels hebben we tal van 'alternatieve' frameworks om backends te maken.

Dit artikel gaat echter niet over deze hype cycle. Althans, niet in de klassieke manier. In dit artikel verkennen we geen nieuwe ideeën en modellen en behandelen we ook geen taalfeatures die elke taal eigenlijk zou moeten hebben. We gaan wel kijken naar een 'nieuwe' taal en wat de gedachtes daarachter zijn.

Go is een taal, ontwikkeld door Google. Het wordt ook vaak 'golang' genoemd. Dit maakt het immers makkelijker om op te zoeken met... Google. De ironie hoe een zoekgigant een taal ontwikkeld, die qua naam slecht vindbaar is, is overduidelijk.

Wat interessant is aan Go, is dat het een hele eigenwijze taal is. Andere 'nieuwe' talen of frameworks lijken vooral uit te breiden op bestaande features, en voegen meer en meer toe aan wat je als ontwikkelaar allemaal zou moeten kennen. Go wijkt hiervan af, vanwege het feit dat het niet veel nieuwe features heeft. Daarentegen zijn er juist veel features die ontbreken. Het is dus een nieuwe taal, waarmee je vooral 'minder' kunt dan dat je gewend bent met Java of een andere taal.

Toch staat Go als beperkte taal in de top 10 van talen op GitHub, komt het in alle rankings terug

Corstijan Kortsmit is engineer bij Info Support. Haalt plezier uit logica vinden in dingen die vooral onlogisch lijken.

```
func main() {
  s := make([]string, 0) // Maak een lege slice (initial size 0) van type string.

  s = append(s, "Hello") // Merk op dat append een keyword is in de taal
  s = append(s, "world") // en dus geen methode op een slice

  fmt.Printf("%s %s \n", s[0], s[1]) // Print de waardes als geformateerde string


  m := make(map[string]string) // Een map met als key string, en als waarde string

  m["een"] = "eerste waarde" // Geen put methode; maar waardes worden eenvoudig
  m["twee"] = "tweede waarde" // toegekend

  fmt.Printf("%v\n", m)
}
```

Listing 1

```
// Definieer de interface met 2 methodes
type vehicle interface {
  move(int) int
  describe() string
}

// Definieer een 'Car' met 'brand'
type car struct {
  brand string
}

// De methodes die we ook terug zien in de vehicle interface.
// Let wel; Op deze manier wordt car impliciet een Vehicle, omdat de methodes van de interface
vehicle geimplementeerd worden op car.
func (c car) move(distance int) int {
  return distance / 120
}
func (c car) describe() string {
  return fmt.Sprintf("Car(%s)", c.brand)
}

// Definieer een truck met dezelfde methodes van de vehicle interface, met andere implementatie
type truck struct{}

func (t truck) move(distance int) int {
  return distance / 80
}
func (t truck) describe() string {
  return "Truck"
}

func main() {
  // We gebruiken hier dus onze vehicle interface waarmee we het gedrag opgeven wat we nodig
hebben in deze slice
  vehicles := make([]vehicle, 0)
  vehicles = append(vehicles, truck{})
  vehicles = append(vehicles, car{"Volvo"})
  vehicles = append(vehicles, car{"Renault"})

  for _, v := range vehicles {
    fmt.Printf("Vehicle type %s takes %v hours to travel %v kilometers\n", v.describe(),
v.move(240), 240)
  }
}
```

Listing 2

als populaire taal en wordt het door zo'n beetje elke groot bedrijf zichtbaar en volop gebruikt. Laten we eens bekijken waarom dit zo is.

Less is more

Wat opvalt als we kijken naar Go is dat er ontzettend veel niet kan. In Java hebben we een uitgebreide Collections API, waardoor allerlei verzamelingen als **`ArrayList`**, **`LinkedList`**, **`HashSet`**, **`TreeSet`** of **`Map`** op een standaard manier kunnen benaderen. Dit is typisch voor object-georiënteerde talen; er zijn vooraf gedefinieerde interfaces en de ontwikkelaar kan kiezen welke implementatie het beste past voor de keuze.

De wereld van Go is eenvoudiger. Je hebt standaard alleen native datatypes voor een array, een slice en een Map. Daarbij is een slice een wrapper om een array heen. Alles wat voor een array geldt, geldt ook voor een slice, en meer. In de praktijk zie je in Go dus eigenlijk vooral slices en maps. Geen sets en geen implementaties, die anders geordend worden.

Op papier is dit een punt voor Java. Immers, meer keuze lijkt fijn. Het is interessant om eens bij jezelf te rade te gaan hoe vaak je nu echt een verzameling tegenkomt in Javacode, die geen **`ArrayList`**, **`HashSet`** of **`HashMap`** als implementatie heeft. Natuurlijk komt dat voor, maar het merendeel van verzamelingen in Java valt onder deze drie types.

Stel dus, dat we met de kennis van alle Javacode een 'nieuwe' Java moesten maken. Agile als we zijn: hoe zou een 'Minimal Viable Collections Api' er dan uitzien?

Dit is een manier van redeneren, die de ontwikkelaars van Go voor ogen hadden. De taal en de standaard bibliotheek zijn haast 'spartaans' om mee te werken. Een Java ontwikkelaar is gewend om keuze op keuze te maken.

Deze keuzes zijn vaak hetzelfde, maar toch: er is keuze. In Go is deze keuzevrijheid minder. Als je een HTTP endpoint wilt maken, dan doe je dat gewoon met de standaard go Library. JSON serialisatie, cryptografie, DateTime, logging: allemaal zaken die in Go standaard goed genoeg werken en waar je geen extra libraries (keuze) voor nodig hebt.

Andere grondslag

De keuze voor minder komt Go ook wel eens te staan op kritiek van ontwikkelaars. Verreweg het meest pijnlijke punt voor (Java) ontwikkelaars die met Go werken, is het gebrek aan generics. Als Java ontwikkelaars dit voor het eerst zien, krijgen ze vaak traumatische flashbacks naar de tijd voor Java 5. In deze tijd moesten we als Java ontwikkelaar alles wat je uit een List haalden, controleren en casten naar een type voor je het kon gebruiken.
In de Java wereld, met de sterke focus op objectoriëntatie voor collections, zijn generics nodig om iets van pragmatisme te hebben, als het gaat om werken met deze collections. In Go zijn de collections echter native datatypes en dus per definitie al getypeerd.

Nu klopt het dat je in Java inderdaad ook bij verzamelingen generic constructies kunt toepassen als `List<? extends Vehicle>` om een lijst aan te duiden, waarin alleen maar subtypes van `Vehicle` kunnen. Dit kan, op het eerste gezicht niet in Go. Echter, vanuit het perspectief van Go is het ook niet nodig. Je wilt dat items in de list extenden van `Vehicle`, omdat je gebruik wilt maken van een methode op `Vehicle`.

In Java moet je interfaces als `Vehicle` definiëren bij het object. Doordat Go werkt met 'impliciete' interfaces, hoef je bij een `car` niet op te geven dat het een `Vehicle` is. Je definieert de interface met methode die je nodig hebt, en die interface gebruik je voor je array, slice, of methode signatuur. De Go compiler controleert dan in ieder geval of alles wat in of uit de verzameling of methode komt, voldoet aan de interface.

Impliciete interfacing is een heel andere manier van denken ten opzichte van de 'traditionele' Java objectoriëntatie. In Java is 'een Eend een Eend', omdat je dat als ontwikkelaar opgeeft. In Go is het veel meer 'als het loopt als een Eend, en het kwaakt als een Eend, dan vind ik het een Eend'. Je kunt dus ook prima zelf interfaces verzinnen, die prima passen op types, die je gebruikt uit een andere library of module. Vaak zie je in Go dus ook kleine, specifieke interfaces, die horen bij bijvoorbeeld een bepaalde slice, omdat daar het gedrag nodig is. De compiler controleert vervolgens wat er mogelijk allemaal in kan komen. De compilatie faalt wanneer code iets probeert toe te voegen, dat niet voldoet aan de interface.

Natuurlijk zijn er scenario's te verzinnen in Java met Generics, die in Go niet mogelijk zijn. De vraag is echter hoe vaak men in de praktijk een dergelijk geval tegenkomt, die niet op een andere gelijkwaardige manier opgelost kan worden.

Pragmatisch, maar anders

Waar we met Java en 'nieuwere' talen vooral pragmatisme invullen als 'hoeveel' (of juist niet) werk een bepaald probleem is, gaat het bij Go vooral om 'hoe eenvoudig' het programma blijft. Deze nadruk op eenvoud zorgt - interessant genoeg - vaak voor beknopte en behapbare code. Een goed voorbeeld hiervan is exception handling. In Java kennen we hiervoor `throw` met `try en catch` voor Checked Exceptions. In de ogen van de Go ontwikkelaar is dit nodeloos gecompliceerd. Er is onderscheid tussen checked en unchecked exceptions, maar alle exceptions kunnen altijd ook weer doorgegooid worden door de aanroeper. Het is een mechanisme dat openstaat voor verschillende interpretaties en gebruiken. Het is inderdaad te zien dat verschillende producten en teams exceptions op verschillende wijzen afhandelen.

Dus als we denken vanuit een 'minimal viable product': why bother? Go als taal kent multiple return values, waarbij een methode dus meer dan een return waarde kan hebben. In de Go community is er een (geschreven)

GO: "ALS HET LOOPT ALS EEN EEND, EN HET KWAAKT ALS EEN EEND, DAN VIND IK HET EEN EEND"

```
func betterDivide(arg1, arg2 int) (int, error) {
  if arg2 == 0 {
    // Let wel dat er nog steeds een return value *moet* zijn omdat go
    // geen 'nil' of 'null' voor integers heeft.
    return -1, errors.New("Can't divide by zero!")
  }

  return arg1 / arg2, nil // Nil als error; alles was immers goed
}

func main() {
  result, err = betterDivide(10, 0)
  if err != nil {
    fmt.Printf("%v\n", err)
  } else {
    fmt.Printf("10 / 0 = %v\n", result)
  }
}
```

Listing 3

afspraak, dat de laatste return value altijd een error is als de methode een error gooit. De aanroeper kan hier dan iets mee doen, of niet. Want laten we eerlijk zijn, een Exception 'checked' maken, geeft geen enkele extra garantie dat de afvanger van de exception er verantwoordelijk mee om gaat.

En zo is Go code vaak heel eenvoudig te lezen. Er zijn geen `try/catch` constructies die weer afwijken van de 'flow' van de code. Er zijn wel veel `if(err){...}` constructies. Dat is ook precies wat er gebeurd: indien er een error is, doe dan dit. Wanneer je als ontwikkelaar geen errors afhandelt? Dan borduur je verder op aannames die niet kloppen, waardoor je programma snel genoeg crasht vanwege iets als een `nil pointer dereference.`

Niet waarom, maar wanneer

Het begint nu hopelijk duidelijk te worden dat de ontwikkelaars van Go op een andere manier kijken naar het vakgebied software engineering. Qua denkwijze lijkt Go nog het meest op het klassieke C (geen c++), compleet met pointers en een totaal gebrek aan exception handling. Het is overduidelijk dat de mensen achter Go een duidelijke eigen wijze (eigenwijze) hebben.

Go heeft sinds de release van versie 1.0 in 2009 geen enkele taalwijziging gehad. Dit alleen geeft al aan dat de ontwikkelaars van Go niet heel erg bezig zijn met andere talen en nieuwe features, die zich daar ontwikkelen. Go is vooral een beknopte, relatief eenvoudige taal met een krachtige basis library en ontzettend snelle ontwikkeltooling. Iedere nieuwe release zijn er optimalisaties in de compiler, waardoor Go programma's sneller en veiliger worden, zonder dat er opnieuw code geschreven hoeft te worden.

Go code is daarnaast 'compile, copy, run'. Go code wordt gecompileerd naar één statische executable. Er zijn geen extra libraries of runtime environments, die je mee moet kopiëren naar een productiemachine of container. Dit zorgt ervoor dat Go programma's vaak relatief klein zijn en dat is ideaal voor moderne, schaalbare cloud deployments. Hier scheelt het nogal of het gaat om een applicatieserver van 500Mb of een enkele executable van 10Mb.

De oorsprong van Go ligt bij Google en het werkt dan ook het beste voor backend software. Het schrijven van services en API's is iets dat eenvoudig werkt in Go, waarbij de standaard bibliotheek vaak al toereikend is. De code compileert ontzettend snel en tools voor het linten en formatteren van je code zijn gestandaardiseerd over de gehele Go community heen.

Daarnaast zien we sinds de opkomst van Docker dat Go veel gebruikt wordt in allerlei DevOps tools. Populaire projecten, zoals Docker en Kubernetes, zijn geschreven in Go. Echter ook allerlei tools daaromheen, zoals Etcd, Vault en Hugo zijn geschreven in deze lichtgewicht taal. Zodoende is de kans groot, dat ook als je zelf geen Go schrijft, je toch al wel gebruik maakt van één of meerdere tools, die geschreven zijn in deze taal.

Het is dus niet echt een vraag waarom je Go zou moeten gebruiken. Het is een keuze die je kunt maken. Alles wat je met Go kunt, kan je zeker ook met Java of met een andere taal. Dat gezegd hebbende, is het interessant om te kijken wanneer je Go zou *kunnen* gebruiken. De randvoorwaarden waaronder Go goed werkt, zijn redelijk eenvoudig;

- Het gaat om een backend of CLI oplossing.
- Er hoeft geen gebruik gemaakt te worden van een bestaande library met bijvoorbeeld business rules, geschreven in Java.
- Eenvoudige deployments zijn belangrijk. Om gebruik te vereenvoudigen of om makkelijk snelle schaalbaarheid te bereiken.
- Opstart- en ontwikkelsnelheid is belangrijk.

Vaak zijn we met software engineering bezig om allerlei engineering principes toe te passen op een nieuwe taal of framework. We schrijven voor of spreken af in onze teams hoe we logging doen, hoe we exceptions doen en hoe we omgaan met interfaces en verzamelingen.

Go is een taal waar men niet zozeer dit soort principes op toepast. Het is zelfs eerder omgekeerd. Er zijn principes over software engineering en de engineers van Google hebben daar een minimale, maar wel bruikbare, taal omheen opgemaakt. Het resultaat smaakt als de bittere citroenfrisdrank: 'een beetje vreemd, maar wel lekker'. ■

Alles over gRPC

High performance micro-services

gRPC is een framework dat zich richt op de communicatie tussen back-end services met hoge prestaties, waarbij gebruik wordt gemaakt van HTTP/2 en Protobuffer 3. In dit artikel gaan we in op wat gRPC precies is en wat de voor- en nadelen hiervan zijn. Aan de hand van een case wordt de basis van het gRPC framework doorlopen.

Hoge prestaties

Nu steeds meer applicaties worden opgebouwd uit verschillende microservices, worden de eisen voor snelle onderlinge communicatie steeds hoger. De gebruiker is gewend aan een snelle verbinding en wil informatie direct beschikbaar hebben. Daarnaast nemen mobiele apparaten een steeds belangrijkere rol in. Voor het gebruik van een dataverbinding onder minder ideale omstandigheden is gRPC ontwikkeld met een mobile-first gedachte. Niet alleen gebruikers stellen steeds hogere eisen, ook ontwikkelaars willen snel nieuwe functionaliteiten kunnen ontwikkelen en zo min mogelijk tijd besteden aan het schrijven van boilerplate code. Om aan deze eisen te kunnen voldoen, heeft Google Stubby ontwikkeld. Een variant van Stubby is later door Google beschikbaar gemaakt als open source project onder de naam gRPC. Voluit betekent gRPC: "gRPC Remote Procedure Call", wat het een recursief acroniem maakt. De "g" in gRPC staat dus niet voor Google, zoals vaak wordt gedacht. Bij RPC zullen velen denken aan CORBA of RMI en zijn dan ook huiverig tegenover gRPC. Anderen denken mogelijk: we hebben toch REST en JSON, wat biedt gRPC mij dan?

Onderliggende techniek

Momenteel wordt veel gebruik gemaakt van REST en JSON. Een bericht in JSON is gemakkelijk te lezen voor mensen, maar is voor een machine niet de meest efficiënte vorm om te verwerken. Daarom maakt gRPC standaard gebruik van de *Interface Definition Language (IDL) Protocol Buffers*, kortweg: Protobuf. Protobuf maakt het mogelijk om data te serialiseren naar een binair formaat. Dit levert kleinere berichten op en maakt ook het afhandelen van berichten efficiënter. De Protobuf berichten worden beschreven in een bestand met de extensie .proto. Een voorbeeld is te zien in **Listing 1**.

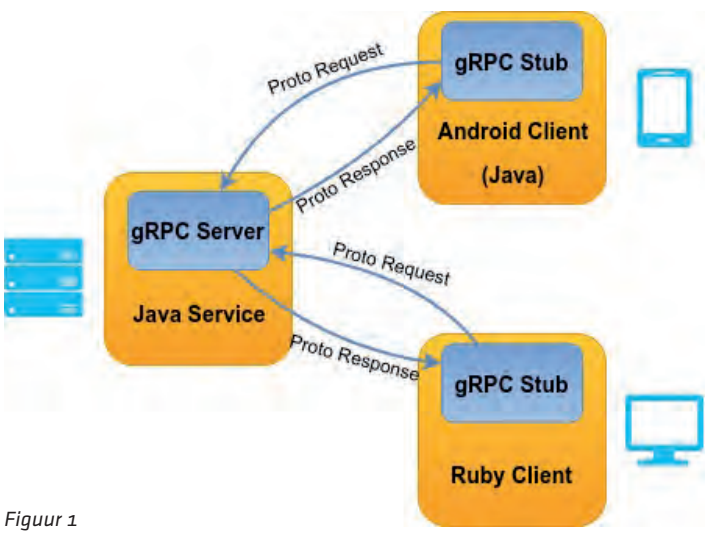
Protobuf is niet platform- of taalafhankelijk. Dit is ook te zien in de Protobuf beschrijving (**Listing 1**). De datatypes in de beschrijving komen het meest overeen met de datatypes uit de taal C++. Dit betekent dat Protobuf strongly typed is. Voor het Java datatype

```
syntax = "proto3";

message Person {
   string firstName = 1;
   string lastName = 2;
   int32 age = 3;
}
```

Listing 1: Protobuf beschrijving

Bart Kerkvliet is software developer bij Quintor Den Haag. Momenteel werkt hij voor KPN New Business aan een IoT platform.



Figuur 1

long wordt in Protobuf het datatype *int64* gebruikt. Doordat Protobuf en gRPC niet platform- en taalafhankelijk zijn, is het mogelijk om services in verschillende talen te ontwikkelen en deze met elkaar te laten communiceren (**Figuur 1**). Door gRPC wordt ook winst geboekt door gebruik te maken van HTTP/2. Met HTTP/2 is het niet meer nodig om voor elk request een nieuwe connectie open te zetten, wat veel efficiënter is. Een bijkomend voordeel is dat het mogelijk is om data te streamen over HTTP/2.

Getting started

Als voorbeeldimplementatie is een project aangemaakt, waarin enkele punten van gRPC worden behandeld. De broncode is te vinden op **https://gitlab.com/BKer/grpc-book-finder-example**.

Het voorbeeld betreft een bibliotheek, waarbij met de naam van een auteur boeken kunnen worden gezocht. Het project maakt gebruik van Gradle en de Protobuf compiler is toegevoegd als plugin. Hierdoor is het niet nodig om Protobuf te installeren en maakt iedereen gebruik van dezelfde versie. Het project zal ook automatisch de .proto bestanden compileren en .java bestanden genereren die gebruikt zullen worden in het project. Na het uitvoeren van "*./gradlew clean build*" zal in de build folder een mapje "generated/sources" te zien zijn. Hierin staat onder andere een *abstract class* die geïmplementeerd dient te worden. Eén van de voordelen van het kunnen genereren van zogenaamde client stubs (soms clients genoemd) is dat het maken van libraries voor een service of product gemakkelijker wordt. Er kan immers direct voor verschillende talen een client stub worden gegenereerd.

Protobuf

Het voorbeeldproject zal een soort bibliotheek/catalogus voorstellen, waarbij men met de naam van een auteur een boek of serie boeken kan opvragen. Er worden twee voorbeeld methodes beschreven voor het opvragen van boeken. De eerste methode zal de boeken in één keer teruggeven, terwijl de tweede methode gebruik zal maken van een stream om de boeken terug te geven. Bij de tweede methode krijgt de client dus een boek toegestuurd, zodra dit boek is gevonden. De input is dus een auteur en de output een boek. De modellen van auteur en boek kunnen als volgt worden beschreven in Protobuf (**Listing 2**).

> Meer informatie over de verschillende datatypes is te vinden op **goo.gl/xFV2P1**

```
message Author {
    string author = 1;
}

message Book {
    string title = 1;
}
```

Listing 2

```
service BookFinderService {
   rpc FindBooksByAuthor(Author) returns (BookList) {}
   rpc FindBooksByAuthorAsStream(Author) returns (stream Book) {}
}
```

Listing 3

```
message BookList {
    repeated Book books = 1;
}
```

Listing 4

```
server = ServerBuilder.forPort(port)
        .addService(service)
        .build();
```

Listing 5

Nu de twee modellen bekend zijn, kan de service (zoals dit heet binnen Protobuf) beschreven worden. In **Listing 3** is de beschrijving van de service te zien.

In de service wordt gebruik gemaakt van het model *BookList*, maar deze is (nog) niet beschreven. Omdat het *Book* model geen lijst is, dient een nieuw type beschreven te worden. Dit type is beschreven in **Listing 4**.

Voor de tweede methode, waarbij gebruik wordt gemaakt van een stream, is het niet nodig om een lijst te definiëren. De stream zal de boeken sturen, zodra deze beschikbaar zijn en niet eerst een lijst opbouwen.

De server

Nu de service is beschreven, is het tijd om de server te bekijken. Het gRPC framework maakt gebruik van een builder-pattern. Bij het opbouwen van de server zal hier gebruik van worden gemaakt. Om een server te starten, dient het poortnummer opgegeven te worden en kan een service worden toegevoegd (zie **Listing 5**).

De server kan gestart worden door *server.start()* aan te roepen. Om ervoor te zorgen

HET GEBRUIK VAN GRPC BLIJFT OP DIT MOMENT VOORAL BEPERKT TOT BACK-END SYSTEMEN

dat de server niet direct weer afsluit, dient de functie *blockUntilShutdown()* aangeroepen te worden. Deze functie zorgt ervoor dat de JVM blijft draaien en de server connecties kan ontvangen.

De service

De service is waar de functies geïmplementeerd worden, zoals staat beschreven in het .proto bestand. Omdat al veel gegenereerd is met de Protobuf beschrijving, is de service relatief eenvoudig te implementeren. In dit geval dient de service *BookFinderServiceGrpc.BookFinderServiceImplBase* te extenden. Daarna zijn de twee methodes, zoals beschreven in de Protobuf definitie, te overriden. De editor zal waarschijnlijk al voorstellen doen, maar de parameters van de methodes zijn in beide gevallen een request in de vorm van een *Author* en een *StreamObserver<V>*.

De *StreamObserver* is een interface met drie methodes, namelijk *onNext*, *onError* en *onCompleted*. Om iets naar de client te sturen, wordt de *onNext* methode gebruikt. In het geval van *BookList* mag deze methode eenmalig worden aangesproken waarna de *onCompleted* aangeroepen dient te worden. Als *onNext* nog een keer zou worden aangeroepen, dan krijgt men een *exception*, waarin wordt aangegeven dat er te veel responses zijn verzonden. In het geval van de stream mag *onNext* wel vaker worden aangeroepen en wordt de stream afgesloten door *onCompleted* aan te roepen. Dit is voor de client het signaal dat deze geen verdere responses hoeft te verwachten. Mocht een exceptie optreden, dan kan deze naar de client worden gestuurd door middel van *onError*. De methode *onError* accepteert een *Throwable*, maar het is beter om deze te wrappen in een *Status* object. Met *Status* kan de client een duidelijke melding krijgen.

gRPC kent een aantal statuscodes en deze zijn ook beschikbaar als enumeration. Aan deze statussen is een omschrijving en detailomschrijving mee te geven en kan de werkelijke exceptie worden toegevoegd als cause. Omdat een *Status* geen *Throwable* is, heeft de *Status* een methode *asException* en *asRuntimeException*. Door hier gebruik van te maken, voorkom je dat de client een status UNKNOWN terugkrijgt.

De client

Net als de server wordt ook de client aangemaakt met een builder-pattern. Hiervoor dient eerst een channel aangemaakt te worden. Dit channel krijgt een adres en poortnummer mee. Als het channel is aangemaakt, kunnen stubs worden gemaakt. Er zijn drie type stubs: een blocking stub, een asynchronous stub en een future stub. De blocking stub is de simpelste van de drie. Bij een methode aanroep verwacht de blocking stub alleen een *Author* object en geeft deze de volledige response terug op moment dat deze binnen is gekomen. Dit betekent wel dat de gehele flow van het programma even stil komt te staan. De asynchrone stub maakt gebruik van het zogenaamde Hollywood principe: “Don’t call us, we’ll call you”. Deze stub verwacht bij een methode aanroep zowel een *Author* als een *StreamObserver*. Nu is de *StreamObserver* een interface en dit betekent dat de drie methodes nog geïmplementeerd dienen te worden.

De *onNext* verwacht een *Book* object. Deze wordt dan ook steeds aangeroepen als een *Book* door de server is verstuurd. De *onCompleted* wordt aangeroepen als al de *Book* objecten zijn verzonden door de server. Omdat de asynchrone stub niet blocking is, heeft deze ook geen return type en zal het programma ook niet worden geblokkeerd en direct verder gaan. Het verwerken van de data dient dan ook allemaal opgelost te worden met behulp van de *StreamObserver* interface methods. Ondanks dat de blocking stub geen *StreamObserver* verwacht, is het uiteraard wel mogelijk om een streaming methode aan te roepen. Er wordt alleen gewacht tot al de objecten binnen zijn gekomen. Als laatst bestaat nog een future stub, maar deze kan geen streams verwerken.

Deadline

Clients kunnen ervoor kiezen om een deadline te zetten. De deadline is de tijd die de client maximaal wacht. Stel: een client zet een deadline van 200 milliseconden waarin een serie micro-services een actie dienen uit te voeren. Elke service zal bepalen of de deadline al is verstreken en als deze is verstreken, dan zullen geen verdere services meer worden aangesproken. Als de deadline verloopt, wordt een DEADLINE_EXCEEDED exceptie gegooid en wordt de keten gestopt.

Ter illustratie is in **Figuur 2** een voorbeeld van het verlopen van de deadline met vier services weergegeven. In het figuur is te zien dat de deadline verloopt en dat de groene service niet meer wordt aangesproken. Dit zorgt

ervoor dat geen onnodige services worden aangeroepen. Mocht de deadline van de client niet verlopen, dan zal het proces volledig worden afgerond. In dit geval zou een deadline van 2 seconden ruim voldoende zijn om het proces af te ronden. Dit flexibele model zorgt ervoor dat clients die meer tijd hebben een aanroep wel kunnen afronden, maar dat clients met minder tijd niet onnodig resources verspillen.

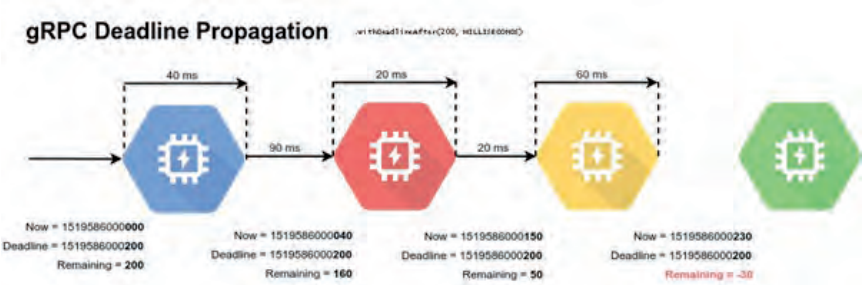
Nadelen

Geen techniek komt zonder nadelen en deze zitten ook aan het gebruik van gRPC. De meeste front-end libraries maken gebruik van REST en JSON en op dit gebied kan gRPC nog niet worden ingezet. Hierdoor blijft het gebruik van gRPC op dit moment vooral beperkt tot back-end systemen. Daarnaast bestaan talloze blogs, video's en cursussen over het gebruik van REST en JSON, maar voor gRPC is de hoeveelheid informatie nog wat beperkt. Een andere uitdaging zit in het testen, een REST endpoint is eenvoudig aan te roepen met *curl* of een vergelijkbare toepassing. Door het gebruik van Protobuf gaat dit niet op voor gRPC. Het is mogelijk om Protobuf te vervangen, maar overige content-types zijn nog niet geschikt voor een productieomgeving. Een ander content-type is dus mogelijk, maar in het toch al nieuwe landschap vormt dit een extra hindernis die dan genomen dient te worden. Wel blijft het mogelijk om gRPC en REST interfaces naast elkaar aan te bieden. Door een goede scheiding aan te brengen in de applicatie kunnen zowel de REST- als gRPC-laag gebruikmaken van de service laag.

Tot slot

In dit artikel is de basis van gRPC besproken met behulp van een kleine case. Hierin is uitgelegd hoe een basisopzet te maken en van daaruit is het mogelijk om door te groeien in het gebruik van gRPC. Bepaalde zaken, zoals discovery en loadbalancing, zijn achterwege gelaten, omdat ze niet binnen de scope van dit artikel vallen.

Steeds meer bedrijven maken gebruik van gRPC, denk hierbij aan grote partijen zoals: Netflix en CoreOS. Tevens is gRPC opgenomen door de Cloud Native Computing Foundation. Door deze sterke community is gRPC geen hobbyproject, maar een serieus alternatief voor een nieuwe service. ■



Figuur 2

STEEDS MEER BEDRIJVEN MAKEN GEBRUIK VAN GRPC, ZOALS: NETFLIX EN COREOS

Evolueren naar microservices

Met Axon Framework

Axon is een volwassen Java framework, dat is gebaseerd op Domain-Driven Design (DDD), Command Query Responsibility Segregation (CQRS) en Event Sourcing (ES). Het is inmiddels zo'n 600.000 keer gedownload en wordt wereldwijd gebruikt, onder andere in de financiële sector. Recent is er veel belangstelling voor Axon in verband met microservices. De DDD/CQRS/ES-concepten zijn weliswaar ouder dan microservices, maar passen daar heel goed bij. En Axon daarom ook.

In dit artikel bespreken we eerst (kort) de kernconcepten, daarna kijken we naar de structuur van een Axon applicatie aan de hand van een voorbeeld en tenslotte bespreken we hoe microservices hierin passen.

DDD

DDD is een brede benadering van softwareontwikkeling, met name bedoeld om applicaties met complexe business logica goed te kunnen ontwikkelen en onderhouden. DDD kent zowel strategische aspecten als hele concrete "building blocks". Building blocks, die je ook in Axon tegenkomt, zijn:

- **Aggregate**: één of meer entiteiten, die vanuit een oogpunt van persistentie en consistentie als een eenheid worden beschouwd. Verschillende aggregates kunnen hooguit indirect naar elkaar verwijzen. In databasetaal: er kunnen foreign key constraints zijn tussen de entiteiten binnen een aggregate, maar niet tussen entiteiten in verschillende aggregates.
- **Repository**: een interface waarmee aggregates kunnen worden gepersisteerd en teruggelezen, onafhankelijk van de onderliggende databasetechnologie. Die komt pas in de repository implementatie om de hoek kijken.
- **Domain Event**: een representatie van iets dat gebeurd is in het domein en waar andere componenten op kunnen reageren. Events zijn immutable (want we kunnen het verleden niet veranderen). In DDD zijn de Domain Events niet puur technisch, maar herkenbaar voor de business owners van de applicatie. Een voorbeeld zou kunnen zijn: "KlantHeeftOrderBevestigd".

Een belangrijk uitgangspunt van DDD dat ook door Axon wordt gestimuleerd, is om business logica in het domein model en in het bijzonder in de aggregates zelf te plaatsen (en dus niet in een "service layer", die is gescheiden van het datamodel).

CQRS en ES

Het kernidee van CQRS is dat we een onderscheid maken tussen commando's (wijzigen van state) en queries (opvragen van state zonder deze te wijzigen). Bij CQRS vinden die twee zaken plaats op aparte databases. Dat levert extra complexiteit op, omdat de read-side consistent moet worden gehouden met de command-side. Daar staat tegenover dat de read-side exact kan worden afgestemd op de functionaliteit van de applicaties (zowel qua technologiekeuze als schema). Queries kunnen daarom erg simpel blijven en dus

Frans van Buul werkt als evangelist bij AxonIQ sinds de oprichting in 2017. Daarvoor heeft hij gewerkt als Java ontwikkelaar en presales consultant.

```
data class IssueCmd(val id: String, val amount: Int)
data class IssuedEvt(val id: String, val amount: Int)
data class RedeemCmd(@TargetAggregateIdentifier val id: String, val amount: Int)
data class RedeemedEvt(val id: String, val amount: Int)
```

Listing 1

ook erg snel worden uitgevoerd. In veel gevallen is het acceptabel als de read-side asynchroon wordt bijgewerkt ten opzichte van de write-side (eventual consistency). Daardoor blijft ook command processing simpel en snel.

ES is op het eerste gezicht een ongerelateerd concept. ES gaat over persistentie van entiteiten. Met een traditionele CRUD-aanpak (bijvoorbeeld met JPA) wordt actuele state van entiteiten gepersisteerd en die wordt telkens bijgewerkt. Bij ES slaan we de geschiedenis van entiteiten op als een serie van events (append-only) en wordt de actuele state afgeleid van de events. De motivatie voor ES zit in de waarde van historische informatie. Denk aan auditing/compliance, machine learning, analytics, etcetera.

CQRS en ES worden vaak in één adem genoemd. CQRS is prima mogelijk zonder ES, maar omgekeerd niet. Een ES-systeem kan vanuit de opgeslagen events efficiënt individuele entiteiten teruglezen, maar niet efficiënt willekeurig queries uitvoeren op de gehele dataset. Daarom wordt ES in de praktijk vrijwel altijd gecombineerd met CQRS.

Praktisch voorbeeld

We bespreken hieronder enkele kernpunten van een Axon voorbeeld dat in z'n geheel kan worden gedownload op GitHub (**https://github.com/AxonIQ/giftcard-demo-series**). We beperken ons tot de command-side.

Het voorbeeld maakt gebruikt van de Axon Spring Boot starter. Axon kan ook zonder Spring (Boot) gebruikt worden, maar Spring Boot maakt het erg gemakkelijk. Alle benodigde infrastructuur wordt dan geconfigureerd met redelijke defaults, zonder dat we iets hoeven te doen.

Het domein van de applicatie is het uitgeven en weer inwisselen van cadeaukaarten (giftcards), die een bepaalde waarde vertegenwoordigen. Dit is een geschikt simpel domein om CQRS/ES te illustreren. Er zijn slechts twee events: er kan een nieuwe cadeaukaart worden uitgegeven (issuing) en een cadeaukaart kan worden gebruikt voor een betaling (redeeming).

Het startpunt bij CQRS/ES zijn commando's en events, die in de applicatie worden gerepresenteerd als immutable objects. Die kun je in standaard Java coderen, maar dat leidt tot veel boilerplate. Meestal wordt daar een andere oplossing voor gekozen, zoals Lombok @Value, Kotlin data classes of Scala case classes. In **Listing 1** zien we een voorbeeld met Kotlin.

Het enige Axon-specifieke aan de listing is de **@TargetAggregateIdentifier**. Dit commando is bedoeld voor een reeds bestaand aggregate. De annotatie vertelt aan Axon dat dit veld de identifier bevat van het aggregate waar het commando voor bedoeld is.

De commando's en events moeten worden verwerkt door het GiftCard aggregate. Deze is weergegeven in **Listing 2**. De class is geannoteerd met **@Aggregate (1)**. Hierdoor zal Axon de class automatisch vinden, een repository instantiëren en scannen voor command en event handlers.

Voor beide commando's uit **Listing 1** hebben we een **@CommandHandler (4)**. Voor het aanmaken van nieuwe cards is dat een constructor, voor het gebruiken van een bestaande card is dat een reguliere methode. In de command handler methods wordt besloten of een commando door kan gaan. In dit geval moeten de bedragen positief zijn en mag er niet meer worden uitgegeven dan nog op de kaart staat. In de command handlers wordt echter geen state gewijzigd. Dat gaat via events. De command handlers eindigen dus met het aanroepen van Axon's **apply** method, waarmee een event eerst wordt toegepast op het aggregate en daarna wordt gepubliceerd voor externe listeners.

Uiteraard moeten we ook nog events afhandelen. Daarvoor hebben we twee **@EventSourcingHandler methods (5)**. Het is belangrijk om te beseffen dat bij ES deze

CQRS EN ES WORDEN VAAK IN ÉÉN ADEM GENOEMD. CQRS IS PRIMA MOGELIJK ZONDER ES, MAAR OMGEKEERD NIET

methods niet alleen maar worden aangeroepen direct nadat een commando een event publiceert, maar ook iedere keer dat dit aggregate opnieuw wordt teruggelezen. De handlers moeten dus geen side effects hebben (die zouden veel te vaak worden uitgevoerd) en geen beslissingen nemen (dat kan tot inconsistente resultaten leiden). De enige gebeurtenis is het veranderen van de eigen state.

Bij het lezen van een bestaand aggregate moeten events worden toegepast op een nieuw, leeg object zonder de commando-constructor aan te roepen. Daarom hebben we ook de lege constructor nodig (3).

De state die door de class wordt bijgehouden (2) bestaat uit de identifier (geannoteerd met **@AggregateIdentifier**) en er

```
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.commandhandling.model.AggregateIdentifier;
import org.axonframework.eventsourcing.EventSourcingHandler;
import org.axonframework.spring.stereotype.Aggregate;

import static org.axonframework.commandhandling.model.AggregateLifecycle.apply;

@Aggregate
public class GiftCard {

    @AggregateIdentifier
    private String id;
    private int remainingValue;

    public GiftCard() {
    }

    @CommandHandler
    public GiftCard(IssueCmd cmd) {
        if(cmd.getAmount() <= 0)
            throw new IllegalArgumentException(
                "amount <= 0");
        apply(new IssuedEvt(
            cmd.getId(), cmd.getAmount()));
    }

    @CommandHandler
    public void handle(RedeemCmd cmd) {
        if(cmd.getAmount() <= 0)
            throw new IllegalArgumentException(
                "amount <= 0");
        if(cmd.getAmount() > remainingValue)
            throw new IllegalStateException(
                "amount > remaining value");
        apply(new RedeemedEvt(
            id, cmd.getAmount()));
    }

    @EventSourcingHandler
    public void on(IssuedEvt evt) {
        id = evt.getId();
        remainingValue = evt.getAmount();
    }

    @EventSourcingHandler
    public void on(RedeemedEvt evt) {
        remainingValue -= evt.getAmount();
    }
}
```

1. Class annotatie waardoor Axon repository en command routing configureert.

2. Identifier en (voor command handling benodigde) state.

3. Lege constructor, nodig voor event sourcing.

4. Command handlers: validatie, mogelijk side effects, geen state wijziging. Laatste stap (bij positieve validatie) is het aanroepen van de 'apply' method met een event als parameter.

5. Event sourcing handlers: geen beslissingen en geen side effects. Uitsluitend wijzigen van aggregate state.

Listing 2

```
/* Usually injected */
private final CommandGateway commandGateway;

...

/* Example: Vaadin click listener sending command. */
submit.addClickListener(evt -> {
    IssueCmd cmd = new IssueCmd(
        id.getValue(),
        Integer.parseInt(amount.getValue()));
    commandGateway.sendAndWait(cmd);
    Notification.show("Success", Type.HUMANIZED_MESSAGE);
});
```

Listing 3

is data nodig om commando's te valideren. Functioneel willen we in de voorbeeldapplicatie ook de oorspronkelijke waarde van de kaart laten zien en niet alleen de resterende waarde. Echter, omdat die oorspronkelijke waarde niet wordt gebruikt om commando's te valideren, hoeft die ook niet te worden opgeslagen in het aggregate. De gebruiker krijgt uiteindelijk inzicht in die oorspronkelijke waarde via de read-side.

Integratie

Op basis van de code, die we hierboven hebben besproken, kan Axon commando's gaan verwerken. Zaken als het laden van de aggregate vanuit de repository, het aanroepen van de command handler, opslag van events en transactie management worden allemaal door het framework afgehandeld.

Voor Axon maakt het niet uit welk (web) framework je gebruikt om je applicatie te schrijven. Het integratiepunt is Axon's CommandGateway. In **Listing 3** laten we zien hoe een stukje code (gebaseerd op Vaadin) commando's kan laten uitvoeren.

Evolutionaire microservices

Wat is nu het verband tussen DDD/CQRS/ES, Axon en microservices? Dat zit in een aantal dingen. Bij een microservices systeem ontstaat van nature de vraag welke services er moeten zijn. DDD biedt daar natuurlijke antwoorden op. Het strategische DDD-concept van de bounded context kan worden gebruikt om (groepen van) microservices af te bakenen. Nog fijnmaziger is het prima mogelijk om individuele aggregates als microservice ter beschikbaar te stellen, omdat aggregates per definitie een consistentiegrens afbakenen en dus goed onafhankelijk kunnen functioneren.

CQRS/ES voegt daar nog het concept aan toe dat alle informatie wordt uitgewisseld met commands, events en queries. In principe maakt het niet uit of die berichten binnen hetzelfde proces worden afgehandeld of door een ander proces.

Axon Framework helpt om de stap van afhandeling binnen het proces (monoliet) naar afhandeling in een ander proces (microservices systeem) te maken. Alle berichtuitwisseling vindt logisch plaats via een bus: de CommandBus, EventBus en QueryBus (Java interfaces in Axon). Standaard worden er in-process implementaties van deze interfaces geïnstantieerd. Zodra we het systeem willen opschalen, kunnen deze gemakkelijk worden vervangen door gedistribueerde implementaties, zonder de business logica te wijzigen. Zo is het mogelijk te beginnen met een monoliet en te evolueren naar een microservices systeem, zodra dat nodig is.

Meer informatie

Uiteraard hebben we hier alleen de basis van Axon kunnen behandelen. Mocht je interesse gewekt zijn, dan kun je veel meer informatie vinden op **https://axoniq.io**, zoals info over trainingen, conferenties en case studies van gebruikers. ■

AXON FRAMEWORK HELPT OM DE STAP VAN AFHANDELING BINNEN HET PROCES (MONOLIET) NAAR AFHANDELING IN EEN ANDER PROCES (MICROSERVICES SYSTEEM) TE MAKEN

Redux State Management

Voorspelbaar state management voor alle JavaScript SPA frameworks

Door de toename van het gebruik van Single Page Application (SPA) frontends, zoals Angular, verschuift steeds meer logica van de backend naar de frontend. Het managen van state in een SPA web app wordt daardoor steeds lastiger.

Naast state afkomstig van de SPA zelf (bijvoorbeeld door gebruikers invoer), zal een SPA vrijwel altijd ook gegevens opvragen bij een backend via REST end-points. Bij wijzigingen dient de state ook te worden gesynchroniseerd met de backend.

Een complicerende factor bij state management van een SPA is de dynamische aard hiervan. Waar 'vroeger' de backend naar de browser enkel de HTML retourneerde, communiceert de SPA nu door middel van data met de backend en wordt op basis hiervan de HTML dynamisch aangepast.Terwijl de SPA web app communiceert met de backend, staat de gebruiker ook niet stil. Zo kan de gebruiker inmiddels zijn eerder ingevoerde postcode alweer hebben gewijzigd, terwijl de response van de eerdere REST call nog niet eens ontvangen is. Of de gebruiker kan de actuele pagina alweer hebben gewijzigd door bijvoorbeeld de browser back-knop gebruikt te hebben.

Soorten van state in SPA frontends

In een SPA frontend wordt er state bijgehouden op allerlei plaatsen, waaronder:

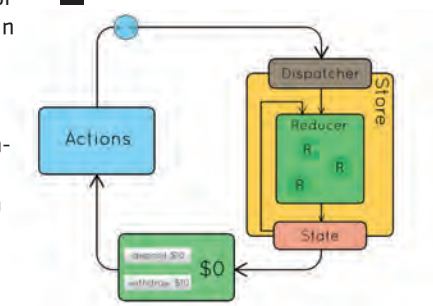
- **In de browser DOM**: het object model van het door de browser getoonde document.
- **In zijn UI componenten**: die worden gesynchroniseerd met de browser DOM door middel van templates en data binding.
- **URL in de browser**: bijvoorbeeld door het id van een entity (bijvoorbeeld een order) op te nemen als path variabelen of door filter waarden op te nemen als query parameters (achter de `?`). Ook haalt de router benodigde gegevens op aan de hand van de informatie uit de URL.
- **De router**: reageert op wijzigingen in de browser URL met als primaire doel om op basis hiervan de actieve view te wijzigen.
- **In cookies en browser storage oplossingen**: zoals session- en local storage.
- **In stateful services**: in tegenstelling tot backend is het gebruik van stateful services in de frontend niet per definitie fout.
- **Globale variabelen**: waarvan het gebruik bij voorkeur dient te worden vermeden.

State management zonder Redux

Een veel voorkomende (anti-)pattern in SPA web applicaties is dat de state benodigd voor de applicatie (en presentatie) verborgen zit in UI componenten.
De quick & dirty oplossing voor de implementatie van het tonen van todo items uit TodoMVC is om een tweetal UI componenten te maken:

- `` `<todo-mvc-item-list>` ``: met daarin de todo items array.
- `` `<todo-mvc-item>` ``: met daarin een todo item object met de properties text en marked (completed).

Emil van Galen
is Senior Java Full-Stack Developer bij JDriven.



Afbeelding 1

```
class TodoStore {
    constructor() {
        /** @type {Array.<{text: string, marked: false}>} */
        this.todos = [];
    }

    addTodo(text) {
        this.todos.push({text: text, marked: false});
    }
    markTodo(index) {
          this.todos[index].marked = !this.todos[index].marked;
    }
    // …
}

export const todoStore = new TodoStore();
```

Listing 1

Nadeel van bovenstaande oplossing is dat de state management één op één gekoppeld is aan de UI componenten. Het is beter om de benodigde state en logica hiervoor los te trekken in services. Voor TodoMVC zouden we bijvoorbeeld een **`TodoStore`** service kunnen maken, zoals in **Listing 1**.

Ondanks het gebruik van services, zie je vaak dat applicatie logica toch (onbedoeld) belandt in UI componenten. Zo is het bijvoorbeeld wel erg verleidelijk om in plaats van '**`.markTodo(..)`**' door two-way binding op de checkbox direct de waarde van de "marked" property te wijzigen.Uiteraard kunnen gewenste wijzigingen van state objecten worden voorkomen, door deze immutable te maken. Zo zouden we de `TodoStore` kunnen aanpassen om gebruik te maken van de Seamless Immutable (**http://bit.ly/2uQ9cdB**) library zoals in **Listing 2**.

State management met Redux

Met de juiste discipline kun je, door gebruik te maken van services (zoals **`TodoService`**), de applicatie logica en bijbehorende state op een juiste manier scheiden van de presentatie logica in de UI componenten.
Nadelen van deze aanpak zijn wel dat:

- De state van de applicatie logica versnipperd zal zijn over diverse services.
- Er soms ook data geduplicerd zal worden bij gebruik van meerdere services.
- Er geen standaard mogelijkheid is om services onderling op elkaar te laten reageren.
- Het expliciet zelf immutable maken van state objecten makkelijk kan worden vergeten.

In plaats van de Services-aanpak kunnen we ook gebruik maken van Redux:

- Hiermee wordt de gehele state op één plek bijgehouden en is het voor iedereen toegankelijk.
- Hierdoor is er geen data duplicatie meer nodig.
- State transities worden geïnitieerd door het versturen (dispatch) van action objecten, waar vervolgens alle geïnteresseerden (de reducers) reageren door een nieuwe state (slice) te creëren.
- Door het wijzigen van een regel kunnen we afdwingen dat de state altijd immutable is.

```
import Immutable from 'seamless-immutable';

class TodoStore {
    constructor() {
        /**  @type {Array.<{text: string, marked: false}>} */
        this.todos = Immutable.from([]);
    }

    addTodo(text) {
        this.todos =
            this.todos.concat({text: text, marked: false});
    }
    markTodo(index) {
        this.todos = Immutable.setIn(this.todos,
                [index, 'marked'],
                !this.todos[index].marked);
    }
    // …
}

// …
```

Listing 2

Het mooie van Redux is dat state transities enkel tot stand kunnen komen door een "one way data flow" en slechts alleen door het versturen (dispatch) van een action object (zie **Afbeelding 1**).

Het diagram in **Afbeelding 1** beschrijft ook de werking van Redux:

- De frontend gebruikt de Redux state, die beheerd wordt door de Redux store. Dit gebeurt door bijvoorbeeld databinding toe te passen op de HTML template.
- Na een gebruikersactie (bijv. indrukken "deposit" knop) zal de frontend een action object creëren en deze doorsturen naar de Redux store.
- De Redux store roept vervolgens de state reducer aan om op basis van de actie en de huidige state de nieuwe state te creëren.
- Tenslotte zal de frontend de nieuw Redux state representeren (bijv. door databinding).

TodoMVC applicatie logic met Redux

Het creëren van de Redux store instantie (de conventie is één instantie per web app) kan eenvoudig worden gedaan met de volgende regels code (zie **Listing 3**). De Redux store heeft slechts weet van één reducer: de "root reducer". Echter, door gebruik te maken van de **`combineReducers`** kan een (root) reducer worden gecreëerd, die de state management van state slices delegeert naar andere reducers. Zo wordt in dit geval het management van de todos state slice gedelegeerd naar de **`todoReducer`**.

Door de standaard **`combineReducers`** functie te vervangen door de third-party library **redux-seamless-immutable** kunnen we tevens garanderen dat de Redux state altijd immutable is (zie **Listing 4**). Om dubbele action types te voorkomen, is het een conventie om alle action types te definiëren in één enkel bestand. Dit bestand heet vaak **`action-types.js`** (zie **Listing 5**).

De constanten uit de **`action-types.js`** worden gebruikt in **`todoReducer`** reducer om de state management te verzorgen van de **`todos`** state slice (zie **Listing 6**).

```
import {createStore, combineReducers} from 'redux';
import {todoReducer} from './todo-reducer'

const rootReducer = combineReducers({todos: todoReducer});

export const store = createStore(rootReducer);
```

Listing 3

```
import { createStore} from 'redux';
import { combineReducers } from 'redux-seamless-immutable';
// …
```

Listing 4

```
export const ADD_TODO = 'ADD_TODO';
export const MARK_TODO = 'MARK_TODO';
// …
```

Listing 5

Integratie Redux in UI componenten

Redux is framework-agnostisch en dus te gebruiken met ieder JavaScript framework. Voor het gebruik van Redux in de UI zijn de volgende Redux store methoden relevant:

- **`getState()`**: retourneert de door Redux state instantie.
- **`dispatch(..)`**: verstuurt een action object naar Redux om een state transitie te initiëren.
- **`subscribe(..)`**: registreert een listener functie, die na iedere state transitie zal worden aangeroepen.

De Redux state kan eenvoudig worden gepresenteerd door data-binding in HTML templates. Als ultieme bewijs dat Redux inderdaad framework-agnostisch is, gebruikt de voorbeeld web app van dit artikel helemaal geen UI framework. Dit wordt ook wel VanillaJS genoemd. Om dit mogelijk te maken, gebruiken we Custom Elements v1 en doen we data binding met behulp van ES6 Template Literals (zie **Listing 7**).

Aangezien er bij iedere state transitie door Redux een nieuwe state instantie zal worden gecreëerd, kunnen we niet zomaar (een gedeelte van) de state eenmalig toewijzen aan een (instantie) variabele. In plaats hiervan maken we een calculated property aan op basis van een JavaScript getter, die bij elke aanroep de gewenste state zal opvragen uit de Redux store.

Bij sommige UI frameworks (zoals AngularJS / Angular) is enkel data-binding op het resultaat van de getter voldoende om de betreffende state te renderen in je HTML template. Echter, aangezien VanillaJS standaard geen change detection heeft, gebruiken wij de **subscribe(..)** method van de Redux store om te genotificeerd te worden van iedere state transitie (zie **Listing 8**).

```
import {ADD_TODO, MARK_TODO, /* , … */} from './action-types';
import Immutable from 'seamless-immutable';

const initialState = Immutable.from([
    {text: 'Use JavaScript', marked: true},
    {text: 'Explore Custom Elements v1', marked: false},
    {text: 'Reduxify your app logic', marked: false}
]);

export function todoReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return state.concat({text: action.payload, marked: false});
    case MARK_TODO:
      return Immutable.setIn(state,
              [action.payload, 'marked'],
              !state[action.payload].marked);
    // …
    default:
      return state
  }
}
```

Listing 6

Om een nieuw todo item toe te voegen, wordt in de keydown event handler van de **<input type="text">** een dispatch gedaan van een Redux action met type **`ADD_TODO`** na het indrukken van de ENTER-toets (zie **Listing 9**).
Voor verdere vereenvoudiging van de integratie van Redux met UI frameworks zijn er diverse binding libraries beschikbaar, zoals o.a. voor AngularJS (1.x), Angular (2+), React, Vue en Polymer (zie: **http://bit.ly/2olJIoi**)

Redux actions

Een action in Redux is een object met ten minste een **type** property, die de action type specificeert.

```
{type: ADD_TODO} .
```

Desondanks is het verstandig om standaard opbouw van action objecten te

```
import html from 'html-template-tag';
import {store} from "../redux/store";
// …

export class TodoList extends HTMLElement {
    // …
    connectedCallback() {
        this.render();
        /** … */
    }
    // …
    get todos() {
        return store.getState().todos;
    }

    render() {
        this.innerHTML = html`
            <!-- … -->
            <ul>
            ${this.todos.map((todo) => html`
                <!-- simplified version of actual HTML -->
                <li>${todo.text}</li>
            `)}
            </ul>`;
    }
}

customElements.define('todo-list', TodoList);
```

Listing 7

```
// …
import {store} from '../redux/store';
// …

export class TodoList extends HTMLElement {
    // …
    connectedCallback() {
        this.render();
        this.unsubscribe =
            store.subscribe(() => { this.render(); });
    }

    disconnectedCallback() { this.unsubscribe(); }
    // …
}
```

Listing 8

```
// …
import {store} from '../redux/store';
import {ADD_TODO} from './action-types';

export class TodoList extends HTMLElement {
    constructor() {
        super();
        this.addEventListener('keydown', this);
    }
    // …
    handleEvent(event) {
        if (event.target.id === 'to-be-added-todo' &&
                event.type === 'keydown' &&
                event.key === 'Enter') {
            store.dispatch(addTodo(event.target.value));
        }
    }
    // …
    render() {
        this.outerHTML = html`
            <input type="text"
                id="to-be-added-todo"
                placeholder="What needs to be done?">
            <!-- … -->
        `;
    }
}

// …
```

Listing 9

hanteren. Gebruik hiervoor bijvoorbeeld de Flux Standard Action (FSA) specificatie, die stelt dat er gebruik dient te worden gemaakt van een payload property.

```
{type: MARK_TODO, payload: 1}
```

Verder is een naamgevingsconventie voor action types aan te raden, zoals: *<werkwoord>_<zelfstandig-naamwoord>* (bijv. **ADD_TODO**).

Tot slot is het gebruikelijk dat action objecten worden gecreëerd door een action creator functie (zie **Listing 10**).

Het voordeel van een action creator is dat bij gebruik hiervan altijd het betreffende action object op dezelfde wijze zal worden gemaakt. Bij gebruik van de third-party library redux-actions (zie **http://bit.ly/2oqHdB0**) kun je tevens ook gelijk het gebruik van de FSA specs afdwingen.

Geen out-of-the-box ondersteuning voor async

Een minpunt van Redux is dat deze standaard geen ingebouwde ondersteuning heeft voor asynchroniteit (ook wel side-effects genoemd). Gelukkig kan deze functionaliteit wel aan Redux worden toegevoegd door middleware.

Nadeel hiervan is wel dat er grote verscheidenheid van middleware hiervoor bestaat, met elk een heel verschillende aanpak. Zo is er middleware die:

- Action creators de mogelijkheid geeft tot het retourneren van een functie (redux-thunk).
- Een REST call uitvoert op basis van de gegevens van de meta property uit een action object (redux-api-middleware).
- De mogelijkheid geeft om direct asynchrone taken te initiëren van een reducer (redux-reducer-effects).
- Het nieuwe concept introduceert voor de afhandeling van side-effects (redux-saga).

```
import { ADD_TODO, MARK_TODO } from './action-types';

export function addTodo(text) {
    return {type: ADD_TODO, payload: text};
}

// …
```

Listing 10

Persoonlijk ben ik het meest gecharmeerd van oplossingen, die een nieuw concept introduceren voor side-effects in plaats van de bestaande Redux concepten en conventies aan te passen. Voor een overzicht van de meest bekende oplossingen ga je naar **http://bit.ly/2HMjlQi**.

Redux DevTools

Een interessante aanvulling op Redux zijn de Redux Developer Tools. Deze geven je allerlei interessante mogelijkheden tijdens development, zoals:

- Inkijk in de huidige of voorgaande Redux state.
- Overzicht van alle verstuurde acties.
- Per actie de state wijziging, die het tot gevolg had.
- Mogelijkheid tot het terugspringen naar een vorige state (tijdreizen).
- Playback van voorgaande acties en de bijbehorende state.
- En tenslotte: het vasthouden van Redux state na een F5 (page reload).

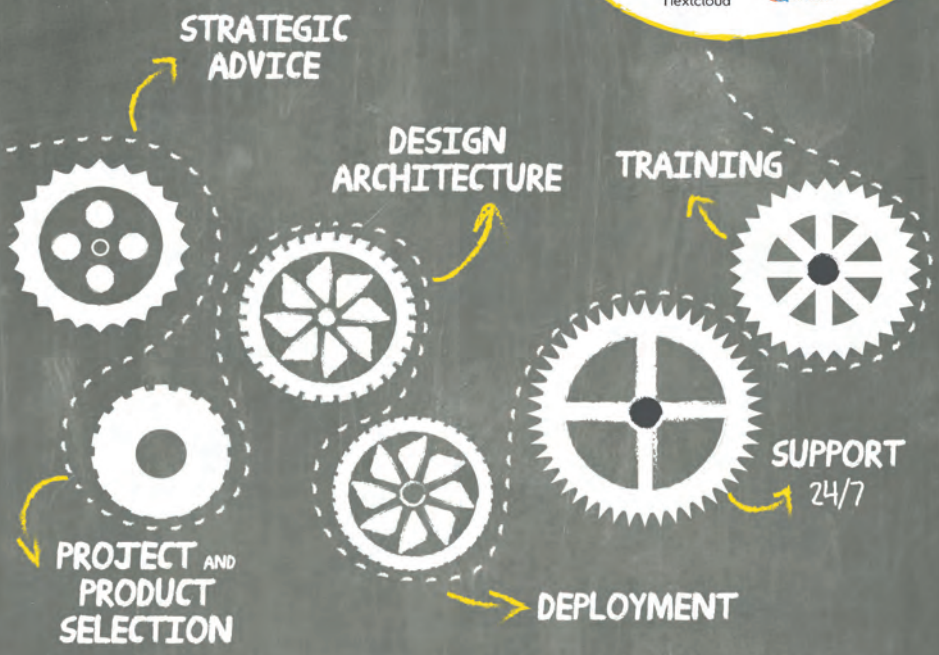
De Redux DevTools zijn beschikbaar als browserextensie voor Chrome en Firefox, maar ook als een module, die kan worden meegebundeld met je web app. Meer info over de Redux DevTools is hier te vinden: **http://bit.ly/2FbnaQM**.

Voorbeeld web app

Als basis voor dit artikel is er een voorbeeld web app ontwikkeld met een implementatie van TodoMVC, die hier te bekijken is: **http://bit.ly/2EVp88x**. ■

EXPERTS IN LINUX AND OPEN SOURCE

kangaroot
Linux & Open Source Solutions



MEET US AT

TEQNATION
THE FUTURE IS OPEN

25 April 2018
Jaarbeurs Utrecht
TEQnation.nl/register

7TH OF JUNE 2018
REGISTER NOW

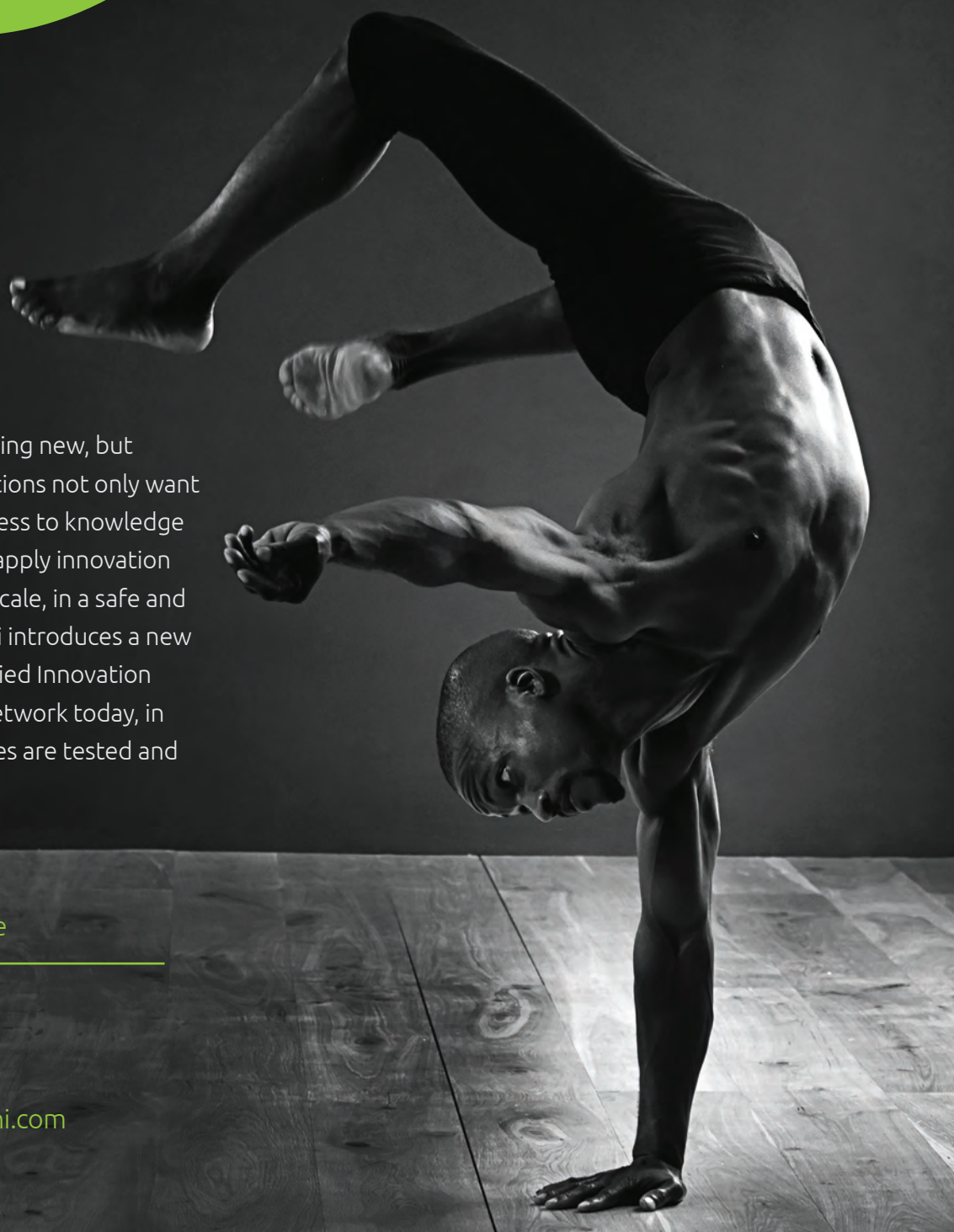
WWW.OPEN18.BE

OPEN ('18)
ENTERPRISE OPEN SOURCE 5TH EDITION

WWW.KANGAROOT.NET KANGAROOTLINUX KANGAROOT-LINUX-SOLUTIONS @KANGAROOTLINUX

Capgemini

Applied **Innovation** Changes **Everything.**



Innovation is no longer something new, but something standard. Organizations not only want technology, they also want access to knowledge that matters. Discover how to apply innovation at the right pace, on the right scale, in a safe and sustainable manner. Capgemini introduces a new and unique approach: the Applied Innovation Exchange. Connect with this network today, in which your ideas and prototypes are tested and turned into true solutions.

For more information:
www.capgemini.nl/
applied-innovation-exchange

Or please contact:
Peter Paul Tonen
Director Applied Innovation Exchange
peterpaul.tonen@capgemini.com
+31 6 4617 1806

Talent

Joop Lanting is vaste columnist voor Java Magazine.

Er is een Garfield (gekke kat) strip, waarin de kat en de hond elkaar achterna zitten en uiteindelijk hoog in een boom eindigen. Hun baasje komt langs en roept: "*Honden kunnen niet in een boom klimmen*", waarna Garfield wijsantwoordt: "*Je moest eens weten wat je allemaal kunt als maar niemand zegt dat je het NIET kunt*".

Wat is talent precies? Een vaardigheid? Nee, eerder het vermogen om een vaardigheid te ontwikkelen. We hebben allemaal wel één of meer talenten, maar die komen niet noodzakelijkerwijs boven water. Er zijn gunstige en ongunstige omstandigheden die bepalen of een talent zijn kans krijgt. De Engelstaligen noemen het 'gifted', alsof je het van iemand aangereikt krijgt. Nou, misschien wel. Maar, "*Het plantje heeft ook water, mest en een goede pot nodig*" en daar wil ik het eens over hebben.

■ **Erfelijkheid**

Soms slaat de natuur een generatie over, maar meestal is de voorouder met een aanleg wel aan te wijzen. Mijn opa had absoluut gehoor en speelde viool. Door omstandigheden moest hij zijn instrument weg doen. Mijn vader en moeder waren amuzikaal, maar mijn broer en ik speelden in orkestjes.

■ **Bekendheid**

Wanneer je niet weet wat er 'te koop' is, loop je het mis. Het Montessori onderwijs liet me maar wat aan rommelen, waardoor ik deels onwetend bleef. Het reguliere onderwijs liet me (gedwongen) overal kennis mee maken. Helaas zijn allerlei vaardigheden gekoppeld aan bepaalde goederen en diensten. Denk maar aan de computer, zodat je jezelf pas kunt ontwikkelen wanneer je er al middenin zit.

■ **Stimulans**

Menig ouder wil dat een kind iets bereikt, maar twijfelt op de 'drempel'. "*Dat is voor ons soort niet weggelegd*". Dergelijke bescheidenheid is besmettelijk en desastreus. Iedereen heeft een coach nodig die in je gelooft. Eén van mijn klasgenoten mocht thuis geen Engels spreken, want zijn vader schaamde zich voor zijn eigen opleiding.

■ **Interesse**

Ik ging ooit met een vriendin naar een ander stel. Voor de gezelligheid gingen we sjoelen (houten schijven door poortjes schuiven). Mijn vriendin had geen zin, dus we speelden met gering succes met zijn drieën. Uiteindelijk deed mijn vriendin ook een rondje mee. Ze haalde de hoogst mogelijke score en ging weer zitten lezen, ons totaal verbijsterd achterlatend: ze vond er gewoon niks aan.

Nog een voorbeeld. Aan de universiteit deden we ooit een kunstmatige intelligentie workshop. De bedoeling was een programma te schrijven dat een verhaaltje zou inlezen en vragen daarover beantwoorden. Eén van de (psychologie) studentes schreef in één weekend een programma dat daaraan voldeed. Het kon 'onthouden' welke personen waren genoemd, wanneer een vraag 'hem' of 'haar' betrof. Wij waren verpletterd, want het was haar eerste programma ooit. Maar ook haar laatste: ze vond programmeren niet leuk en ging liever verder met haar andere studie.

■ **Ambitie**

Hoeveel talent je ook hebt, het gaat niet vanzelf. Schaker Aleksandr Aljechin studeerde zelfs nog zeven uur per dag toen hij al wereldkampioen was. Herman van Veen vertelde eens dat hij, ondanks zijn succes als cabaretier, nog elke dag uren viool studeerde, omdat hij volwaardig viool wilde spelen op het toneel.

■ **Gelegenheid**

En tot slot is het leven een spel van kansen. Je kunt nóg zoveel van plan zijn, een begin maken staat gelijk met een gesloten deur openen. In veel gevallen kun je niet zomaar ergens naar binnen lopen. Verder is het soms bijna onmogelijk op een ingeslagen weg terug te komen (bijv. de schoonmaakster die zangeres wil worden). Wat mezelf betreft, mijn Natuurkunde/Informatica studie was in het slop geraakt, toen mijn vroegere biologielerares me vroeg of ik wat wilde verdienen met tuinieren. Terwijl ik de plantjes deed, vertelde ze dat ze aan de Universiteit werkte en pas een computer had aangeschaft voor een ton. Aangezien er geen geld meer was voor personeel vroeg ze of ik er verstand van had. Ik zei ja en verknoeide mijn studie definitief door twee jaar bijna fulltime op haar machine te werken en te experimenteren. Doordat ik zo, voor die tijd unieke, kennis opdeed, kreeg ik een uitstekende baan als systeemprogrammeur aan de Universiteit van Amsterdam. Had ik die kans niet gekregen, dan had ik niet in JAVA geschreven, maar het blad rondgebracht.

Het is natuurlijk ook een kwestie van geluk hebben, in de juiste tijd en op de juiste plaats te leven. Wie in de bush wordt geboren heeft weinig kans om concertpianist te worden. En wanneer ik met mijn IT-talent 3.000 jaar eerder was geboren in Hellas, dan had ik een baan kunnen krijgen in een atelier voor gevorderde telramen met een eigen rijtuig en een knaapje van de zaak... Nou, dank je feestelijk.

;JOOP!

Van het bestuur

Carrière-advies uit Brazilië en een vooruitblik naar een aantal events dit jaar

Java developer career workshop

Even geleden was ik bij de Amsterdam Java User Group om samen met Bruno Souza een workshop te geven. Bruno staat bekend als de 'Brazilian Javaman' en is op veel conferenties te herkennen als een energiek figuur die met een Braziliaanse vlag om zijn schouders rondloopt.

We hebben Bruno's 'Java Developer Career Workshop' gepresenteerd. Dit is een sessie die een aantal carrièretips geeft aan Java-developers. In deze bestuurscolumn deel ik een aantal van Bruno's adviezen.

Zijn eerste tip was "*Start with finding your focus*". Waar ben je goed in, wat vind je leuk, waar help je anderen mee (collega's, klanten, de wereld) en wat levert dat op? Als oefening liet hij het publiek de volgende zin invullen: "*I help (people) to do (this) so they can have (that)*".

Daarna benoemde hij vijf essentiele vaardigheden voor developers:

1. read code: zorg dat je vloeiend wordt in het lezen en begrijpen van code van anderen. Dat kun je onder andere doen door code van open source projecten te bekijken.

2. write code: schrijf zelf code en vraag anderen om het te reviewen en feedback te geven.
3. deliver code: zorg dat je niet stopt na het schrijven van code, maar dat je alle stappen beheerst om code vanuit versiebeheer in productie te krijgen.
4. solve problems: één van de meeste waardevolle eigenschappen als developer is het vermogen om problemen op te lossen. Probeer geoefend te worden in het gestructureerd aanpakken van problemen. Dat kun je onder andere doen door samen met iemand anders een probleem aan te vallen en te bekijken hoe de ander het probleem benadert.
5. share what you know: misschien wel één van de belangrijkste vaardigheden om verder te komen in je carrière: deel je kennis en ervaring! Dat kun je doen met collega's en met anderen door te bloggen, artikelen te schrijven (bijvoorbeeld voor Java magazine) of te presenteren op meetups en conferenties. Zou je dit wel willen, maar je weet niet waar je moet beginnen? Stuur me dan gerust een mailtje op **bert.jan.schrijver@nljug.org** – ik help je graag verder.

TEQnation en J-Spring

We zijn alweer volop bezig met de organisatie van twee events in het voorjaar: TEQnation en J-Spring. TEQnation is de opvolger van de IoT Tech Day en is een tweedaagse conferentie op 25 en 26 april 2018 in de Jaarbeurs in Utrecht. De eerste dag staat in het teken van Open Source en de tweede dag is gericht op Smart technology; denk hierbij aan AI, Big data, IoT en AR/VR. Neem voor meer informatie een kijkje op **www.teqnation.nl**.

Vorig jaar hebben we voor het eerst sinds lange tijd weer een J-Spring georganiseerd. Dat beviel zó goed dat we dat dit jaar weer gaan doen! Op 31 mei 2018 kun je je bij Tivoli Vredenburg in Utrecht weer een dag lang te goed doen aan de beste Java-sprekers uit de hele wereld. Voor de sprekers en het programma, kijk je op **www.jspring.nl**.

Tot slot

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

En noteer alvast in je agenda: J-Fall is dit jaar op donderdag 8 november 2018. Op 7 november 2018 is er weer zoals gebruikelijk een pre-conference. Deze bevat een aantal workshops, de Masters of Java en 's avonds een diner met alle deelnemers en sprekers. Zie je daar ;-)

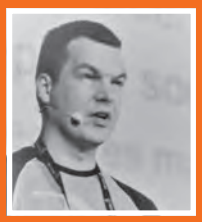
groeten, Bert Jan

Bert Jan Schrijver
Bestuurslid NLJUG.

Maven to GIT

Meer met Maven

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



Robert Scholte
Freelance ICT consultant, Java architect en chairman Apache Maven.
Twitter: @rfscholte

Tot voor kort stonden de sources van Apache Maven in Subversion. Dat heeft ons heel lang voor een dilemma gesteld. Git is de nieuwe standaard en via Github is het erg eenvoudig geworden om Pull Requests aan te bieden. We willen echter niet het overzicht kwijtraken van de jobs in Jenkins, onze Continuous Integration Server.

Maven bestaat in totaal uit ruim 85 projecten, waarvan ongeveer de helft de plugins zijn. Voor plugins willen we testen met zoveel mogelijk combinaties van besturingssystemen, JDKs en Maven versies. In Subversion konden we dit oplossen door alle plugins in een gezamenlijke directory te plaatsen. Met behulp van een aggregator pom (een pom met alleen maar modules, waarbij het zelf geen parent is) kon je dan alle plugins in één run draaien. Het was niet optimaal, want met een enkele commit op één plugin werden alle plugins gebouwd. Maar op deze manier bleef het aantal jobs overzichtelijk en goed te beheren.

Een overstap naar Git zou betekenen dat we voor elke plugin een aparte repository moeten maken, elk met een eigen set aan jobs om op gemiddeld 2 operating systems, 4 JDKs en 3 Maven versies te testen. We hebben git-modules overwogen, maar dat zou maar een deel van de problemen wegnemen.

Ondertussen biedt Jenkins ook steeds nieuwe features. Stephen Connolly, committer voor zowel Maven als Jenkins, is de uitdaging aangegaan om een oplossing te vinden voor dit probleem. Zijn ultieme goal: uiteindelijk is er nog maar 1 job, welke gekoppeld is aan een organisatie (in ons geval AsfMaven) en alle repositories die onder deze organisatie vallen zullen automatisch gebouwd worden, inclusief de branches. En uiteraard geldt ook hier weer: convention over configuration. Dit betekent dat een project conform een aantal defaults wordt gebouwd, maar deze kun je overrulen wanneer nodig.

Stephen heeft zijn weg naar de oplossing opgenomen en als een twaalfdelige unedited “Watch me code”-reeks gepubliceerd op YouTube. Daarin vertelt hij stap voor stap wat hij precies van plan is te gaan doen. Je zult zien hoe hij van een idee tot een implementatie komt. Het risico van live-coding is dat je onbewust fouten maakt en dat blijkt in dit geval ook een aantal keren te gebeuren. Echter krijg je ook te zien hoe deze tijdens het testen aan het licht komen, hoe Stephen de fout weet te vinden en vervolgens weet op te lossen.

De hele reeks is te vinden via **https://www.cloudbees.com/blogs/stephen-connolly**. Een aantal episodes gaan wellicht iets te diep, waardoor het soms lastig volgen is. Over het geheel genomen is het zeker het bekijken waard om te zien hoe een idee langzaamaan vormt krijgt. Het resultaat: elke repository hoeft slechts één Jenkinsfile in de root van het project te plaatsen. In Jenkins is het overgrote deel van de jobs overbodig geworden. Daarnaast heeft het INFRA team van Apache ook belangrijke stappen gezet. Het is ondertussen mogelijk om Pull Requests rechtstreeks vanuit Github te accepteren en welke in de originele Git repository van Apache gemerged worden (zie: **https://gitbox.apache.org/repos/**).

Dit alles moet het nog eenvoudiger maken om bij te dragen aan de open source projecten van Apache. ■

Triodos Bank

Waarom Triodos Bank?

Ik heb de eerste jaren na mijn studie op allerlei manieren geprobeerd mijn draai te vinden in verschillende rollen, voordat ik ontdekte dat mijn werkplezier niet alleen uit de dagelijkse werkzaamheden komt, maar ook uit het uiteindelijke doel van de software die ik ontwikkel. Bij Triodos Bank draag ik bij aan de missie van het bedrijf en kan ik de wereld een beetje mooier maken door te doen wat ik goed kan: software ontwikkelen.

Welke uitdagingen zijn er op technisch vlak?

We zijn een groeiende organisatie met een systeem bestaande uit zo'n 1,5 miljoen regels Java-code die zijn opgebouwd in de afgelopen 15 jaar. Dit met een groeiende club van inmiddels zo'n 60 developers verdeeld over 15 scrumteams, dus je kunt nagaan hoe complex dat is! Uitdagingen zat dus: hoe zorgen we dat nieuwe developers hun weg kunnen vinden in de codebase (modulariteit, scheiding van verantwoordelijkheden, leesbare code, goede testcoverage etc.)? Hoe blijven we up-to-date en zetten we nieuwe technologie in waar dat nuttig is? Hoe vertalen we een businesswens naar iets dat in te zetten is voor alle internationale branches, op een toekomstbestendige manier?

Hoe ziet jouw ideale dag eruit?

Mijn ideale dag begint met een lekkere (en eerlijke!) koffie. Gedurende de dag staan er veel developers aan mijn bureau met vragen over gewenste oplossingsrichtingen of code reviews. Naast de nodige meetings en af en toe een sollicitant, kom ik tussendoor gelukkig ook nog toe aan ontwikkelwerk. In het Quality & Maintenance team waarin ik zit houden we ons vooral bezig met vanuit de afdeling geïnitieerde projecten op het gebied van schaalbaarheid, onderhoudbaarheid en kwaliteit. Dat kunnen bijvoorbeeld performanceverbeteringen zijn, maar ook aanpassingen aan onze ontwikkelomgeving die het werk voor alle developers weer wat makkelijker maken.

Martijn – Developer/Component Architect

Waar heb jij je de laatste jaren in kunnen ontwikkelen?

Ik leer veel on the job, of uit gesprekken met collega's. Daarnaast biedt Triodos Bank me ook de mogelijkheid om te leren. Zo reizen we ieder jaar met een flink gezelschap naar Antwerpen om Devoxx bij te wonen. Ook op het gebied van de missie en waarden van Triodos Bank word ik uitgedaagd: zo is er bijvoorbeeld het Values Seminar waarin je met een groep collega's uit de hele Triodos groep (dus internationaal) twee dagen lang inzoomt op de waarden waarop Triodos Bank gebaseerd is. Super inspirerend!

Hoe blijven jullie op de hoogte van de technische ontwikkelingen en hoe wordt dat toegepast binnen Triodos Bank?

Ik denk dat het belangrijkste is dat we proberen om niet geïsoleerd te raken. Naast het werk in de scrumteams en inhoudelijke afstemming en code reviews tussen teams, hebben we ook iedere twee weken een meeting met alle developers. Hier presenteren collega's vaak een technisch aspect van een project, een nieuw framework of juist nieuwe features in de IDE, of hebben we een spreker van buiten. Zo blijven we elkaar inspireren!

TRIODOS.NL

WIL JE BIJDRAGEN AAN DEZE UITDAGINGEN? WIJ ZIJN OP ZOEK NAAR NIEUWE DEVELOPERS (FULLSTACK EN FRONT-END!)

18-9566 March 2018

Ons eigen Java bier

Wil jij zo'n gaaf bierpakket ontvangen?

Omdat wij geloven dat naast inspanning ook ontspanning belangrijk is, hebben wij bij Ordina ons eigen biertje gebrouwen. Wij willen dit graag delen met jou.

Meer over Java bij Ordina

Onze Java club bij Ordina laat zich niet leiden door standaard loopbaanpaden. Als je goed bent, ben je goed. We hebben geen "up or out" mentaliteit, maar je kunt wel pijlsnel groeien als je die ambitie hebt. J-Tech is Ordina's Java en open source unit met 150 ervaren software developers. Front-end, backend en app developers die met cutting edge technologie werken voor onze klanten.

Het delen van kennis en sociale activiteiten zijn het meest belangrijk voor JTech.

Wil jij zo'n gaaf bierpakket ontvangen?

Stuur jouw gegevens naar JTech@ordina.nl en de reden waarom jij ons biertje optimaal kunt testen en je krijgt het testpakket thuis gestuurd!

Neem ook eens een kijkje op www.werkenbijordina.nl

ORDINA ICT voor mensen

JTECH