Microservices

In de praktijk

Microservices is de laatste tijd het nieuwe buzzword in applicatie-architectuur. Het doel is om kleinere, doelgerichte applicaties te bouwen. Het is een antwoord op de problemen die ontstaan in de alom vertegenwoordigde, complexe, monolithische applicaties. Onder een monoliet verstaan we applicaties die heel veel verschillende businessfunctionaliteit bevatten en als één geheel worden gedeployed. Een groot nadeel van zo'n architectuur is dat een wijziging op één stuk businessfunctionaliteit leidt tot een nieuwe deployment van de gehele applicatie.

Een belangrijk element in een microservices-architectuur is het groeperen van functionaliteit. Als object-georiënteerde programmeurs zijn we wel gewend om de businesslogica te ordenen en te structureren op class- en packageniveau. Microservices past deze technieken toe op applicatieniveau.

Elke applicatie, groot of klein, heeft zijn eigen businessdomein. Doordat je nu een traditionele applicatie met een vaak complex businessmodel opdeelt in meerdere applicaties, wordt het totale businessmodel inzichtelijker. Immers heeft elke microservice zijn eigen domein, dat een onderdeel is van het totale businessmodel van je uiteindelijke applicatie.

De interacties tussen microservices heb je door de splitsing expliciet gemaakt. Dit geeft niet altijd een overzicht. Er kunnen heel veel interacties tussen microservices ontstaan naar mate je landschap van services groeit.

De grenzen tussen deze deeldomeinen zijn daarom cruciaal. Het is de kunst om als team te bepalen welke functionaliteit nu bij welke domein (lees: microservice) hoort. De ervaring leert dat een generieke interface uiteindelijk leidt moeilijk onderhoudbare services.

Het is verstandig om de interfaces tussen microservices zeer specifiek te benoemen. Het benoemen van grenzen en de interfaces tussen services vraagt om een goede, eenduidige, afstemming tussen de business-experts, ontwikkelaars en beheer.

De technieken uit Domain Driven Design geven je verscheidende handvatten om deze grenzen tussen domeinen te bepalen en zo je applicatie op te delen in verschillende microservices. Het voert wat ver om deze in dit artikel te behandelen, maar het is zeker een aanrader om je hierin te verdiepen als je met microservices aan de slag wilt gaan.

In deze serie beschrijven we in een aantal cases hoe verschillende projecten aan de slag zijn gegaan met microservices. Uit de verschillende cases blijkt dat het onderwerp veel breder is dan enkel de manier waarop je code organiseert. Het heeft een directe impact op je manier van werken, je organisatie en het opleveringsproces. Dit effect is niet te onderschatten.

Je ondergaat een leerproces met deze verandering van architectuur. Daar is geen goed of slecht in. Stadia zijn ook lastig te herkennen. Misschien zijn ze zelfs onwenselijk om te benoemen, omdat ze als meetlat gaan dienen. En het daarmee vergelijken leidt eerder af van de onderliggende uitdagingen.

Zoals zo vaak is het beter om klein te beginnen en je successen en tegenslagen te blijven communiceren en analyseren. Op die manier kom je vanzelf uit op het uiteindelijke doel: een model dat het beste past bij je organisatie en bedrijfsvoering. En wie weet... Misschien is het wel een vorm van microservices waar de volgende auteurs over spreken. ■

Jethro Bakker is freelance software ontwikkelaar en architect met een bijzondere interesse in big data en enterprise search oplossingen. Hij is momenteel werkzaam bij de Nationale Politie

Eelco Meuter is Software craftsman bij JPoint en momenteel werkzaam bij de Nationale Politie

Microservices bij Malmberg

Malmberg behoort tot de top drie van educatieve uitgeverijen van Nederland in het basis-, voortgezet- en middelbaar onderwijs. Binnen Malmberg werken een aantal DevOps-teams aan digitale leerplatformen voor verschillende onderwijstypen. Deze case deelt de ervaringen vanuit het oogpunt van die teams.

Zo'n twee jaar geleden is Malmberg een aantal projecten gestart op een nieuwe technologie-stack: Vert.x, MongoDB en AngularJS, draaiend in de Amazon cloud. Vert.x is een toolkit voor het bouwen van *reactive* applicaties op de JVM [1]. De keuze voor Vert.x is gedreven door de reactive eigenschappen (*responsive, resilient, elastic, message driven*) [2] en het krachtige module-systeem. Die laatste is belangrijk, omdat we wisten dat we een set herbruikbare services voor educatieve applicaties wilden gaan bouwen. Destijds was het concept van micro-services nog niet wijdverspreid. Het feit dat we nu met een microservices-architectuur werken, zou je dan ook kunnen zien als 'bij-vangst' van de keuze voor een modulair, reactive platform. Inmiddels draaien 8 projecten met deze technologiestack in productie.

Wat gaat er goed?

Een applicatie bestaat doorgaans uit een stuk of 10 tot 15 services. Een klein aantal hiervan zijn projectspecifiek en handelen zaken als http-requests en flows over meerdere services heen af. Het merendeel van de services is echter generiek voor alle projecten. Deze worden door een centraal team ontwikkeld en door de projecten als Maven dependencies naar binnen gehaald. Elk team heeft dus zijn eigen instanties en versies van de generieke services draaien. Dat kan, omdat de teams wél functionaliteit delen, maar geen data. De teams zijn in productie dus niet afhankelijk van andere teams. Doorgaans stappen de teams vrij snel over naar de nieuwste versies van de generieke services.

De services communiceren met elkaar via de gedistribueerde eventbus in Vert.x. Services hebben één duidelijk doel, bijvoorbeeld sessiebeheer. Als het handig is dat een service meerdere verantwoordelijkheden heeft (bijvoorbeeld omdat het als proxy dient voor een remote webservice met meerdere endpoints), dan doen we daar niet moeilijk over. Hetzelfde geldt voor het delen van code tussen services: als dat handig is, dan doen we dat.

We testen de services individueel en in combinatie met elkaar, onder andere via integratietests, waarin we tijdens een test een service en afhankelijkheden opbrengen.

We kunnen services los uitrollen en upgraden met minimale impact op andere services. Toch deployen we onze applicaties in de praktijk meestal nog wel als geheel. Onze deployment-tooling werkt dan overigens alleen de gewijzigde services bij. We tunen JVM-settings en Vert.x-configuratie op serviceniveau. Services die veel gebruikt worden of veel geheugen gebruiken (bijvoorbeeld voor caching), kunnen we dan meer resources toekennen. Draaien in losse JVM's levert ook voordelen op bij probleemanalyse, want je ziet sneller welk proces (en dus in welke service) in de problemen zit.

Bert Jan Schrijver is Software craftsman bij JPoint, momenteel werkzaam bij Malmberg. Hij is erg geïnteresseerd in Java, Continuous Delivery en DevOps

Een applicatie-node bestaat uit een webserver met één stateless applicatie, opgebouwd uit een set services. Onze applicatie-nodes zijn geclusterd: doorgaans draaien we voor een productieomgeving met 2 tot 6 servers met een loadbalancer ervoor. Die voert een healthcheck op alle nodes uit, die intern per node weer alle services afgaat. Zodra er een service niet (tijdig) reageert, valt de node uit de loadbalancer en gaat er geen verkeer meer naartoe, totdat de healthcheck weer aangeeft dat alles in orde is.

Voor deployments van Vert.x modules gebruiken we open source tooling, die we zelf ontwikkeld hebben [3]. Op elke applicatie-instance draait een deployment agent, die we via REST benaderen vanuit Jenkins. De services die we deployen komen als binary uit Nexus. De deployment tooling interfacet met de Amazon API, zodat we seamless node-voor-node kunnen upgraden zonder impact op de eindgebruikers.

We hebben geen development teams, maar DevOps teams, die de volledige verantwoordding over hun project hebben: van code tot productieomgeving. Alle infrastructuur wordt automatisch opgebouwd met Puppet en CloudFormation en we kunnen automatisch schalen op basis van load. De teams houden hun productieomgevingen zelf in de gaten via allerlei metrieken, dashboards en gecentraliseerde, gestandaardiseerde logging en sturen bij waar nodig.

We werken bewust niet met containeroplossingen, zoals Docker. Waarom niet? Dat is een artikel op zich waard ;-) Zie [4] voor een toelichting.

Wat kan er nog beter?

Alhoewel we het op DevOps-vlak al behoorlijk goed doen en er weinig misgaat, kan de adoptie binnen de organisatie nog wel beter. Dit is niet alleen een technisch proces. Het gaat echt om een andere manier van werken en dat soort verandering kost tijd en moeite.

Werken met microservices blijkt toch wel wat lastiger dan met een monoliet. We moeten ons her en der in wat bochten wringen om met afhankelijkheden tussen services om te gaan. We hebben er bewust voor gekozen om nog niets met service discovery te doen en om ook nog niet meerdere versies van een service tegelijkertijd te draaien. Dat brengt wat beperkingen met zich mee, maar daar is in de praktijk goed mee te leven.

Een typische applicatie bestaat uit 10 tot 15 services en dus 10 tot 15 JVM's. Door de JVM- en OS-overhead moeten we af en toe wat kruidenieren met geheugen en is een machine met 2GB RAM hierdoor al aan de krappe kant. We proberen door het monitoren van productieomgevingen zo goed mogelijk inzicht te krijgen in de geheugen-distributie over JVM's heen.

Conclusie

Voor ons werkt een microservices-architectuur behoorlijk goed, zeker gezien de behoefte om met generieke, herbruikbare bouwblokken te werken. We doen niet in alle gevallen microservices 'volgens het boekje', maar proberen er praktisch mee om te gaan. Werken met microservices is zonder meer lastiger dan met een monoliet, maar het levert je ook veel op. ■

REFERENTIES

[1] *http://vertx.io*
[2] *http://www.reactivemanifesto.org*
[3] *http://github.com/msoute/vertx-deploy-tools*
[4] *http://www.slideshare.net/BertJanSchrijver/geecon-microservices-2015-swimming-upstream-in-the-container-revolution*

Microservices bij Bol.com

In de afgelopen zestien jaar heeft bol.com de transformatie gemaakt van een winkel voor boeken en entertainment naar een verzameling van speciaalzaken met in totaal zo'n negen miljoen artikelen. Inmiddels heeft bol.com meer dan 5,5 miljoen klanten in Nederland en België. Met deze groei ontstond ook de groeipijn en het kraken van het IT-landschap. De systemen waar we ooit mee waren begonnen, groeiden uit tot grote, logge monolieten met een veelheid aan complexe, verweven functionaliteiten en verantwoordelijk voor evenzoveel domeinen. Niemand had meer echt het overzicht. Daar bovenop kwam de personele groei, waarbij steeds meer Scrum-teams in dezelfde monolieten werkten. We konden niet meer schalen (in de breedste zin van het woord). De wens van de business om nieuwe functionaliteit werd met de dag lastiger te vervullen.

Maarten Dirkse werkt bij bol.com in het DPI team waar hij code schrijft en dingen automatiseert. Hij focust zich momenteel voornamelijk op Mayfly en de ver-microservicing van het bol.com platform

Maarten Roosendaal werkt als IT Architect bij bol.com in het Architectuur team met focus op (schaalbare) zoekoplossingen en SEO

Vanaf 2009 is bol.com gestart met de overgang naar een servicegeoriënteerde architectuur. We hebben destijds een service gedefinieerd als:

"[E]en logische eenheid die een verzameling taken, bevoegdheden en verantwoordelijkheden heeft. Er moet sprake zijn van een hoge interne samenhang en een lage externe samenhang."

Daarbij is van belang dat je aan de business kunt uitleggen wat de rol van een service zou zijn. Services moeten immers de business in staat stellen om hun visie en wensen te realiseren. Toen ergens in 2012 dan ook bij een plan door de business zelf om een nieuwe service werd gevraagd, wisten we dat we ze van onze aanpak hadden overtuigd.
Een van de essentiële aspecten van een service is de ontkoppeling. Een service moet bestaansrecht hebben zonder dat het (te) afhankelijk is van andere services. Maar hoe definieer je die functionele afbakening en splits je grote monolieten in de juiste services? Iets als een AccountService of SearchService (oorspronkelijk onderdeel van de monoliet 'WebShop') lijken triviaal, maar wat behelst die service dan precies? Wat is een Account en wat kun je allemaal Search'en? Een nieuwe *greenfield*service is wat dat betreft relatief eenvoudig. Je begint met één, via een API ontsloten stukje functionaliteit in een losse deploybare eenheid. Maar een bestaande monoliet opsplitsen langs de juiste grenzen en soms ook combineren met iets nieuws is een forse uitdaging. We kiezen dan ook vaak voor een Agile-aanpak, waarbij we trachten een goed onderbouwde visie en keuze te maken voor een service en dan simpelweg te starten met bouwen. Het zal dan wellicht niet helemaal juist of volledig zijn, maar niet kiezen is geen optie en inzicht komt met tijd en ervaring. Als de praktijk anders uitwijst, dan stellen we die visie bij. We erkennen dat we nog veel moeten leren qua servicedesign en zijn al regelmatig tot nieuwe inzichten gekomen.

Op het technisch vlak hebben we ook nog de nodige stappen te zetten. In ons huidige landschap zijn service instanties gekoppeld aan een VM, is er een omvangrijk proces om een nieuwe service te laten ontstaan en duren deployments lang. Dit zijn allemaal barrières voor het creëren van, en snel itereren op, kleine services. Daarnaast hebben we ook nog een uitdaging in onze huidige testomgeving: voor het hele bol.com ITlandschap hebben we momenteel één testomgeving waar iedereen op werkt. Dit was een logische keuze toen ons landschap uit sterk verweven monolieten bestond, maar inmiddels voldoet deze werkwijze niet meer. Als een service of applicatie in het testlandschap faalt, dan hebben (in potentie) alle Scrum-teams daar last van. Daarom hebben we anderhalf jaar geleden besloten om onze development en productieplatformen zodanig te verbeteren, zodat ze meer geschikt zijn voor een (micro) service-architectuur.

De oplossing voor de development uitdagingen is alweer enkele maanden in gebruik en heet Mayfly. Mayfly biedt Scrum-teams een geïsoleerde testomgeving *per user story*, die in uitvoering is. Niet alleen de service waaraan het team werkt wordt in deze omgeving geïnstantieerd, maar ook test databases, test properties en een instantie van Critter (een service test tool) worden beschikbaar gesteld. In plaats van een grote, gedeelde testomgeving hebben teams nu de mogelijkheid om een testomgeving per story te creëren, compleet met feature branches, die automatisch worden gemaakt en gemerged door Mayfly. Hierdoor hebben problemen, die geïntroduceerd worden door codewijzigingen van een story, geen impact op andere teams of zelfs andere stories. De oorspronkelijke visie van onafhankelijkheid en ontkoppeling tussen services wordt hierdoor ook werkelijkheid in de dagelijkse ontwikkelervaring van de Scrum-teams.

Ook de deployment kant van de medaille behoefde aandacht. Als bol.com willen we onze 40+ Scrum-teams de vrijheid geven om een fijnmazige service-architectuur te creëren. De oorspronkelijke constructies, die we hanteerden voor het lanceren van een nieuwe service, waren te zwaar en te zeer afhankelijk van IToperations om goed te kunnen schalen. Lijvige nonfunctional requirements documenten, uitvoerige provisiontrajecten van VM's (met volledig statische configuraties) voor services en tijdrovende deployments kwamen de ontwikkelsnelheid niet ten goede. We zijn momenteel bezig met de evolutie van ons platform naar een meer dynamisch model. We gebruiken nieuwe technologieën, zoals Docker en Mesos om deployment en provisioning van service instanties te versnellen en versimpelen. Statische serviceconfiguratie wordt vervangen door tools, zoals Consul (voor service discovery) en onze eigen property service Kevlar (voor database wachtwoorden etc.) Daarnaast maakt een aantal Scrum-teams inmiddels gebruik van Flyway en myBatis Migrations om te zorgen dat database deployments binnen de service kunnen worden geregeld in plaats van via een apart deploy proces.

Naast de technische oplossingen, die we doorgevoerd hebben om onze service-architectuur te realiseren, zit het serviceconcept inmiddels ook bij de hele IT-afdeling goed tussen de oren. Er wordt zelfs gesproken over het "verservicen" van enkele vroege service,s die op hun beurt weer monolieten dreigen te worden. Het doorvoeren van onze (micro-)service aanpak geeft ons evenwel het vertrouwen dat we de huidige en toekomstige problemen kunnen oplossen. ■

Microservices bij de Nationale Politie

Bij de Politie is veel aandacht voor het verbeteren van de ICT-omgeving, zowel bij het bouwen van de nieuwe organisatie (van 26 korpsen naar één politie), als in het ondersteunen van de operatie. Een goed voorbeeld is het project waar wij aan werken. Dit is een onderzoeksomgeving, waarmee eindgebruikers digitale informatie op een veilige, forensische, geborgde manier kunnen gebruiken. Een onderdeel van deze omgeving is een applicatie waar een eindgebruiker digitaal materiaal verzamelt ten behoeve van opsporing en onderzoek.

Deze applicatie is begonnen als elk ander Java web-project met haar typische opbouw. Deze opbouw bevatte de traditionele presentatie-, business- en datalaag. Naarmate het project groeide werd steeds duidelijker dat deze opbouw niet voldeed.

Dit had enerzijds te maken met de toenemende complexiteit en integratie van de functionele wensen, alsmede dat het applicatielandschap een state-of-the-art Openstack private cloud omgeving is met haar specifieke wensen. Daarnaast merkte we als team, dat er steeds meer verwevenheid ontstond in onze code. Het werd tijd voor een wat onconventioneler aanpak: een kleine revolutie binnen de organisatie.

Opsplitsen naar functionaliteit

We zijn begonnen met het herschikken van de verschillende functionaliteit. Hiervoor hebben we losjes de principes van Domain Driven Design gehanteerd. Dit houdt in dat elk specifiek onderdeel van de totale applicatie een eigen (sub)domein heeft.

De raakvlakken met andere domeinen worden ontsloten met een REST interface voor point-to-point communicatie en Kafka voor pub-sub communicatie. We hebben gekozen voor REST en Kafka omdat dit makkelijk horizontaal schaalt in een cloudomgeving.

Versiebeheer

Het beheren van versies is door de splitsing naar meerdere applicaties heel belangrijk. Versiebeheer is en blijft een uitdaging ondanks de verscheidenheid aan tooling, zoals Maven. De keuze voor microservices kan zelfs zorgen voor een grotere uitdaging. Immers moet je nu voor elke microservice de verschillende versies tussen services en de third-party dependencies bijhouden.

Het opsplitsen van de applicatie in meerdere subdomeinen maakt de verwevenheid inzichtelijk en beperkt het aantal third-party dependencies per service. Dit vereenvoudigt de impact van een upgrade en maakt het uiteindelijke beheer eenvoudiger.

Jethro Bakker is freelance software ontwikkelaar en architect met een bijzondere interesse in big data en enterprise search oplossingen. Hij is momenteel werkzaam bij de Nationale Politie

Eelco Meuter is Software craftsman bij JPoint en momenteel werkzaam bij de Nationale Politie

Onafhankelijke deployments

We hebben voor Spring Boot gekozen als basis voor deze microservice architectuur. De domeinen zijn relatief fijn gegranuleerd. Dit betekent dat de uiteindelijke implementatie klein en overzichtelijk is. Een expliciete keuze voor één framework geeft eenduidigheid in onze code base. Dit komt de productiviteit ten goede.

Elke applicatie draait in zijn eigen virtuele omgeving die we automatisch opbouwen. De services worden gedeployed met Rundeck en Puppet. Rundeck is verantwoordelijk voor het uitvoeren van de juiste Puppet-commando's binnen onze private cloud instances. Dit is een integraal onderdeel van ons continuous integration proces.

Samenwerken is een must

Doordat de services autonoom draaien wordt een deel van de verwevenheid tussen domeinen verplaatst van de traditionele monoliet naar de middleware zoals de load balancer. Dit vraagt dus meer afstemming tussen ontwikkelaars en beheerders.

Binnen ons project staat DevOps hoog in het vaandel. Beheerders en ontwikkelaars zitten bij elkaar en kunnen continu met elkaar afstemmen. We werken op deze manier veel beter samen op het gebied van monitoring, security en optimalisatie van de verschillende applicaties in relatie tot netwerken, virtualisatie en zelfs hardware.

De samenwerking met beheer leidt tot applicaties waar de reactieve foutafhandeling snel en efficiënt wordt gedetecteerd. De volgende stap is dat onze services zelf een oplossing hebben voor een eventuele verandering in het landschap. Immers kan een afhankelijke REST service niet beschikbaar zijn. Een andere volgende stap is de implementatie van elastische load balancing.

De heldere splitsing naar businessfunctionaliteit en domeinen schept veel duidelijkheid.

Door het definiëren van specifieke interfaces en een eenduidig protocol wordt veel miscommunicatie voorkomen.

De vergaande integratie in een hoog gedistribueerde omgeving kent veel uitdagingen. Het vereist dat zowel ontwikkelaars als beheerders samenwerken om zo tot een goede oplossing te komen. Dit vergt kennis en begrip van elkaars werkveld.

De complexiteit en het feit dat we in een klein team werken, betekent dat we alles automatiseren wat we kunnen automatiseren. Dit voorkomt herhaalbare fouten en is tegelijkertijd een goede borging van onze gezamenlijk opgebouwde kennis. ■

REFERENTIES

Openstack: *http://www.openstack.org*
Kafka: *http://kafka.apache.org*
Rundeck: *http://rundeck.org*
Puppet: *https://puppetlabs.com*
Spring boot: *http://projects.spring.io/spring-boot/*