De Java ontwikkelomgeving van 2016

Ontwikkelomgevingen blijven vaak een onderwerp van discussie. Mag iedereen zijn eigen favoriete Integrated Development Environment (IDE) gebruiken? Hoelang duurt het voordat een nieuwe ontwikkelaar zijn ontwikkelomgeving volledig werkend heeft? Draai je de omgeving op je laptop, op een centrale server of gevirtualiseerd? Hoe gaat de code vervolgens naar productie? Deploy je een war vanaf je laptop, of gebruik je Jenkins of iets dergelijks? Dit artikel biedt een blik in de keuken bij een aantal bedrijven. De auteurs hebben ruime ervaring met een flink aantal verschillende oplossingen voor ontwikkelomgevingen. Hierna kun je lezen wat de ervaringen en de voor- en nadelen van de verschillende oplossingen zijn.

Ontwikkelomgeving in een Docker container

Containers en Docker zijn tegenwoordig erg hip, maar worden nog vooral gebruikt om niet-grafische applicaties in te draaien. Dus men gebruikt containers om een applicatieserver in te draaien, maar niet om een IDE in te draaien. Bij ING/Info Support waren we benieuwd hoe goed het zou werken om een hele ontwikkelomgeving in Docker te draaien.

Er bleken drie opties te zijn om verbinding te maken naar de ontwikkelomgeving in Docker. De bekendste opties zijn Remote Desktop Protocol (RDP) en Virtual Network Computing (VNC). Je gebruikt een applicatie, die RDP en/of VNC ondersteunt en daarmee maak je verbinding naar een draaiende Docker container. Een interessant alternatief is het gebruik van het X Windows System (X11). Met X11 kun je ervoor zorgen dat als je een Docker container start, dat je direct een IDE te zien krijgt. Dit zorgt ervoor dat je niet eerst een container hoeft te starten en daarna een verbinding moet maken naar de container, zoals met RDP en VNC.

De ontwikkelomgeving bij onze ING-teams stond ofwel op een centrale gedeelde Linux server waarop een heel team moest werken of op persoonlijke Windows machines. Aangezien Windows nog geen ondersteuning bood voor Docker maakten we gebruik van Virtualbox images met daarin de Docker containers. Om verbinding te maken gebruikten we RDP en VNC.

Overzicht van de omgeving

Voor meer informatie over Docker verwijs ik je naar het artikel in de links (zie kader "Referenties"). Wat wel handig is om te weten, is dat een container een draaiende instantie is van een image.

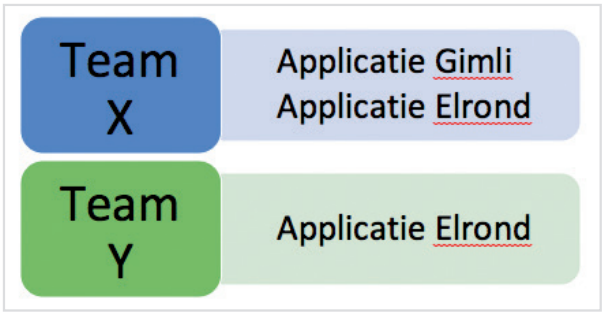
Met Docker kun je images stapelen, een soort van inheritance in Java. In eerste instantie gebruikten we een basis image en daarbovenop een team specifiek image. In het basis image

Johan Janssen is trainer, architect en Competence Center Lead Java bij Info Support.

zitten algemene dingen zoals Java, Maven, NPM, Tomcat en security configuratie. In het team image zitten team specifieke zaken, zoals de applicaties en instellingen van dat team.

In het begin werkte dat prima, totdat we sommige onderdelen wilden delen met andere teams en bepaalde onderdelen juist niet. In **Figuur 1** is te zien dat applicatie Elrond zowel voor team X beschikbaar moet zijn als voor team Y. Nu zouden we de applicatie in het Shared image kunnen zetten, maar alle applicaties zitten in Tomcat servers, die automatisch opstarten tijdens het starten van het image. Dus als er applicaties in de container zitten, die je eigenlijk niet nodig hebt, dan gebruiken ze nog wel resources.

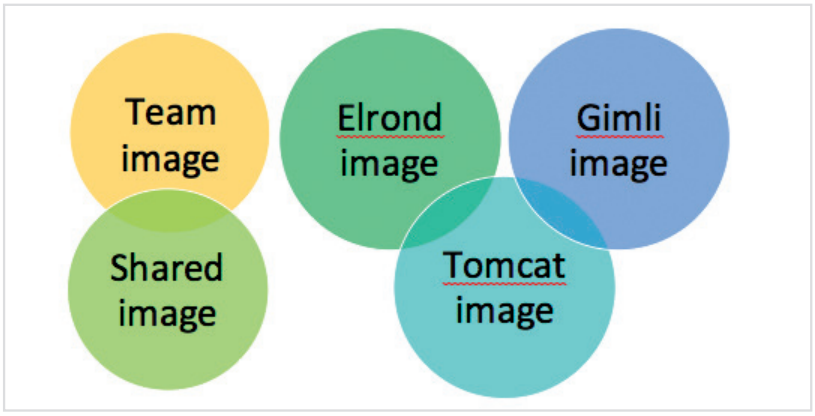
De oplossing hiervoor is om de applicaties in losse Docker containers te draaien. Naast het Shared en team image is er ook een Tomcat image met daarin alle Tomcat zaken, die voor iedere applicatie nodig zijn. Bovenop het Tomcat image is er een Elrond image en een Gimli image. Beide images kunnen los gestart worden, waardoor de gebruiker zelf kan kiezen welke hij of zij wil gebruiken. Vervolgens kan het team image gebruikmaken van de verschillende applicatie images (zie **Figuur 2**). Dit werkt al aardig, alleen moet je nu een hele set aan images starten. Daarnaast moet je op basis van IP-adressen en poorten een verbinding maken vanuit het team image naar de applicatie images. Ook dat kan makkelijker, namelijk met Docker compose. Hiermee kun je in een bestand definiëren welke images er gestart moeten worden en wat de onderlinge relaties tussen de containers zijn. In het Docker compose bestand is te zien welke drie omgevingen gebouwd worden. Uiteindelijk



Figuur 1

zijn ze via de standaard RDP-poort 3389 beschikbaar. Vanuit de development environment zijn er links naar de beide applicatie containers. Met het commando 'docker-compose up' worden alle beschreven acties in het Docker compose bestand uitgevoerd. Het resultaat is dat je omgeving draait en dat je vanuit je ontwikkelomgeving naar de Gimli applicatieserver kunt verbinden op **http://gimli:8080**. Dus in plaats van het IP-adres kun je de naam gebruiken, die in de Docker compose file is gedefinieerd. De verschillende images slaan we op op een private Docker registry (soort van Git voor images), zodat iedereen ze kan gebruiken.

```
tomcatgimli:
  build: TomcatGimli
tomcatelrond:
  build: TomcatElrond
developmentenvironment:
  build:  DevEnv
  ports:
  - "3389:3389"
links:
  - tomcatgimli:gimli
  - tomcatelrond:elrond
```



Figuur 2

Nadat de code geschreven is, wordt die door de ontwikkelaars in Git opgeslagen. Een commit zorgt ervoor dat een Jenkins build start en de codekwaliteit gecontroleerd wordt met SonarQube. Als de kwaliteit goed genoeg is, wordt de artifact (ear/war) opgeslagen in Nexus. Bij ING gebruiken we Nolio om vervolgens iets te deployen op de OTAP straat, maar bij andere klanten gebeurt dat ook wel met Jenkins of andere software. Applicaties als Jenkins, SonarQube, Git en Nexus/Artifactory worden bij sommige klanten met de hand geïnstalleerd, maar op de meeste plekken gebeurt dat gelukkig geautomatiseerd met bijvoorbeeld Chef/Puppet en/of Docker containers.

Waarom hebben we voor deze omgeving gekozen?

Elke keer als er bij ING nieuwe collega's kwamen of er omgevingen wijzigden, dan kostte het veel tijd voor de ontwikkelaars om hun omgeving bij te werken. Daarom wilden we graag een makkelijke manier hebben om ontwikkelomgevingen uit te rollen. We hadden ook kunnen kiezen voor virtual machines, echter zijn die wat lastiger te upgraden en vrij groot. Wel is het zo dat iedere klant en soms zelfs ieder project eigen wensen heeft. Daarom bieden we dit soort oplossingen aan, maar als men liever iets anders wil, dan kan dat natuurlijk.

Voor- en nadelen

De opzet is erg flexibel, want je kunt blijven uitbreiden door containers te stapelen of naast elkaar te draaien. De eerste keer kost het wel behoorlijk wat tijd om alles om te zetten naar Docker containers, maar dat kun je ook stapsgewijs doen door in eerste instantie nog wat handmatige acties uit te voeren. Sommige ontwikkelaars willen liever de IDE buiten Docker draaien. Gelukkig kan dat gemakkelijk en kun je vanuit die IDE verbinden naar de Docker containers. Hierdoor houden ontwikkelaars de vrijheid om hun eigen IDE te kiezen en te draaien waar ze dat willen. Bij sommige hardware ondervonden we helaas wat problemen met de VNC/RDP verbinding. Dus voordat je alles omzet in Docker containers kun je beter eerst even proberen of het goed werkt. Gelukkig ging het bij andere hardware wel goed. Al met al zijn we erg tevreden met de huidige oplossing en het wordt ook door behoorlijk wat ontwikkelaars gebruikt. Hierdoor kunnen nieuwe teamleden nu sneller beginnen en kunnen bestaande omgevingen in Docker snel bijgewerkt worden. ■

REFERENTIES

Docker voorbeelden met X11/VNC/RDP:
https://bitbucket.org/johanjanssen/dockeride
Docker intro:
www.nljug.org/databasejava/docker

Ontwikkelstraat op basis van OpenShift

Kleine zelfsturende teams, die (binnen kaders) de vrijheid hebben om architectuur en implementatie keuzes te maken, die optimaal zijn voor het ontwikkelen van hun deel, zijn een belangrijke succesfactor. Als uitgangspunt hebben we een aantal standaarden. Zo moet het mogelijk zijn om een omgeving volledig in te richten op basis van een checkout uit versiebeheer, met zo min mogelijk externe afhankelijkheden.

Overzicht van de omgeving

Voor het beheren van de verschillende omgevingen gebruiken we Docker images. De basis images worden beheerd in een eigen Docker registry. Dit geeft het voordeel, dat we onze eigen omgevingen kunnen beheren, met bijvoorbeeld de laatste security patches en ook custom images kunnen maken, zoals bijvoorbeeld een combinatie van Jenkins en SonarQube in één image. Ook kunnen we klant specifieke images maken. Door gebruik te maken van de layering mogelijkheid van Docker images kunnen we basis images bieden met bijvoorbeeld Java 8 en JBoss EAP, die vervolgens door de ontwikkelteams verder uitgebreid kunnen worden. Deze teams kunnen hun eigen images vervolgens weer opslaan in de Docker registry.

We hebben ervoor gekozen om al onze omgevingen binnen het OpenShift platform van Redhat te draaien. Naast ondersteuning voor Docker levert OpenShift ook orchestration in de vorm van Kubernetes, hoge beschikbaarheid, limieten op gebruikte resources en strikte isolatie van containers onderling. Dit geeft ons de mogelijkheid om de volledige ontwikkelstraat binnen OpenShift te draaien. Door de extra abstractielaag ten opzichte van kale VM's, kunnen we ontwikkelteams veel vrijheid en

Erwin de Gier is software architect en trainer bij de business line Open Source van Sogeti.

volledige verantwoordelijkheid geven voor hun applicaties. Aan de andere kant kunnen we optimaal gebruik maken van standaardisatie door het bieden van standaard Docker images.

Waarom hebben we voor deze omgeving gekozen?

Onze vorige ontwikkelstraat bestond uit centrale servers voor Jenkins (master en meerdere slaves), Artifactory, Jira, Confluence en Subversion. Hiernaast waren er per project (gevirtualiseerde) omgevingen voor de databases en applicaties. Vooral het beheer van de Jenkins omgeving voor een groot aantal projecten werd een steeds grotere uitdaging. De verschillende projecten bleken vaak veel verschillende eisen te stellen aan plug-ins, versies en omgevingen. Een tweede nadeel van de oude opzet was dat het aanvragen van servers vrij lang duurde. Als er een nieuwe omgeving nodig was voor een klant (Proof of Concept of afstudeerproject) dan gingen hier weken overheen. Ook moesten servers weer expliciet vrijgegeven worden en moest centraal bijgehouden worden welke server waarvoor werd gebruikt.

De Redhat OpenShift omgeving maakt het mogelijk om een eigen ontwikkelstraat per project in te richten. Door gebruik te maken van een eigen Docker registry is het toch mogelijk om standaarden af te dwingen en beheer te doen op omgevingen. Zo hoeven de gebruikers zich niet druk te maken om OS-versies en security patches en hebben ze toch de vrijheid om hun omgeving zelf in te richten. Binnen OpenShift heeft elk project zijn eigen omgeving. Binnen deze omgeving is de ontwikkelstraat ingericht op basis van Docker images. Op deze manier heeft elk project bijvoorbeeld zijn eigen Jenkins instantie. De OpenShift omgeving maakt het ook heel makkelijk om nieuwe omgevingen aan te maken. Via de commandline client of de webportal kunnen omgevingen worden toegevoegd. Ook is het mogelijk om op basis van een Jenkins build een nieuwe omgeving uit te rollen.

De configuratie van een omgeving is in principe vastgelegd in een Docker file in de Git-repository. Na een commit van de sourcecode wordt via een webhook een Jenkins build uitgevoerd, die de applicatie artefacts bouwt. Deze Jenkins build start vervolgens een OpenShift build. We maken gebruiken van de OpenShift "Docker" build strategie. Deze creëert op basis van de Docker file en de applicatie artefacten een Docker container en start deze binnen een zogenaamde pod. "Pod" is een Kubernetes begrip en duidt op een verzameling Docker containers, die gezamenlijk een beheerbare omgeving vormen met een eigen IP-adres. Containers binnen een pod delen hun lokale opslag en netwerk.

Het voordeel van deze opzet is dat de omgevingen portable worden en het eenvoudiger is om meerdere omgevingen uit te rollen en te beheren. De Docker files kunnen immers ook gebruikt worden op de lokale omgeving van de ontwikkelaar. Fouten, die ontstaan door configuratieverschillen, komen op deze manier nagenoeg niet meer voor. Ook het opzetten van een omgeving kost minder tijd en behoeft minder documentatie. In de praktijk maakt dit het eenvoudiger om omgevingen te beheren en is het mogelijk om configuratiewijzigingen te volgen in versiebeheer.

Op de lokale omgeving van de ontwikkelaars wordt verder nog Docker compose gebruikt om meerdere containers tegelijkertijd te starten en met elkaar te verbinden, bijvoorbeeld voor het lokaal draaien van een database. Docker compose maakt gebruik van een yml configuratiebestand. In dit bestand worden de verschillende containers gedefinieerd en geconfigureerd. Dit kan door gebruik te maken van een Docker file of door middel van het specificeren van een image. Het is best-practice om applicatieconfiguratie via omgevingsvariabelen te doen. Binnen de OpenShift omgeving kunnen deze namelijk eenvoudig worden beheerd.

Voor- en nadelen

Het grootste voordeel van onze omgeving is de mogelijkheid voor selfservice voor diverse ontwikkelteams. Teams kunnen zelf omgevingen aanmaken, beheren en weggooien. Hiernaast kunnen ze gebruik maken van standaard images, die voor hun beheerd worden. De

Component	Tool
Versiebeheer	Git
Build tool	Maven
OS	Ontwikkelomgeving op developer laptops, testomgevingen binnen OpenShift
IDE	Vrije keuze developers, (Eclipse, IntelliJ en Netbeans)
Configuratie-omgevingen	Sogeti images in private docker registry, Docker files in versiebeheer
Databasemanagement	Centrale databases met data voor het reproduceren van specifieke gevallen of performance tests. Lokale databases op de laptops van de ontwikkelaars om onafhankelijk en snel te kunnen ontwikkelen.

met de laatste versies van Java, Spring en de bekende Javascript frameworks. Momenteel ziet onze ontwikkelomgeving voor deze componenten er als volgt uit. Wij werken met sprints van twee weken. Voordat er een story "in sprint" genomen wordt zijn er een aantal refinement slagen geweest waar de gewenste functionaliteit en de oplossingsrichting besproken zijn. Dit administreren wij momenteel in Phabricator. Dit is een open source suite van web-based software development tools, vergelijkbaar met de commerciële Atlassian suite. Voorheen gebruikten we Redmine voor onze projectadministratie. Die tool voldeed echter niet aan onze werkwijze en werd uiteindelijk een belemmering. Na een snelle review van wat alternatieven proberen we nu dus Phabricator. Het is vooralsnog een verademing.

Ons bouwproces wordt gedirigeerd door Jenkins. Het opleveren van elk product naar onze centrale repository is in zogenaamde pipelines gedefinieerd. Een pipeline bestaat uit kleinere autonome stappen, zoals bouw, test, package, deploy, etc. Zo'n stap wordt een job genoemd. De pipeline wordt getriggerd door een push naar git.

Voor de job definities gebruiken we de "pipeline as code" features van Jenkins 2. Dit heeft een aantal voordelen. Op deze manier is het bouwproces gestandaardiseerd, zijn bouwstappen herbruikbaar over alle builds heen en zijn alle job en pipeline definities onder versiebeheer. Wij hebben de deployment op deze development-omgevingen geautomatiseerd met Rundeck. Dit is de laatste stap in onze Jenkins pipelines.

In een ideale situatie draait de gewenste functionaliteit autonoom in een microservice. Wij streven er daarom naar de backend processen embedded in onze testen te kunnen draaien. Dit lukt niet altijd. In zulk soort gevallen vallen we terug op mock services waartegen de testen kunnen draaien. Deze mock services draaien in een speciale omgeving, die Jenkins voor ons opzet, zodra we de testen uitvoeren. Af en toe kiezen we er ook voor om integratie-tests te schrijven, die met een "productie-like" omgeving communiceren, zodat de effecten van een gewijzigde, afhankelijke, API direct duidelijk zijn.

De virtualisatietechniek is identiek aan onze productie-omgevingen. Immers hebben wij al onze infrastructuur geautomatiseerd. Deze scripts halen dan de nieuwe producten op vanuit de centrale repository, zodra wij een update nodig achten. Deze werkwijze zorgt er voor dat je als ontwikkelaar vrij bent om je IDE en OS te kiezen. Een deel ontwikkelt met OS X en een deel met Linux. Deze keuzevrijheid leidt tot kleine variaties in de versies van JDK, Maven, Node, e.d. per ontwikkelaar.

Hiermee ontstaat de onvermijdelijke "works-on-my-machine" bug. Dat zien we niet direct als een probleem. Het leidt tot beter begrip over de interne werking van je product, het geeft beter inzicht in het gewenste autonome design en het forceert je beter op de standaarden van de taal te focussen.

Verder gebruiken we SonarQube en enkele JavaScript frameworks om de technische kwaliteit van onze producten te bewaken.

De huidige opzet werkt goed. Het bouwproces verloopt over het algemeen vlot. Door de vergaande standaardisatie heb je snel overzicht van de verschillende interacties binnen het applicatielandschap. Nieuwe teamleden kunnen snel aan de slag met hun favoriete tools en als ontwikkelaar heb je veel invloed om het buildproces te optimaliseren. Dat laatste is volgens ons van essentieel belang voor een effectieve ontwikkelomgeving. Uiteindelijk maakt het niet uit welke tool of mooie spulletjes je hebt, want als het niet precies aansluit bij de gewenste werkwijze van het team, dan leveren deze enkel ruis op een effectieve productiegang. ■

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

Bert Jan Schrijver is software craftsman bij JPoint, momenteel werkzaam bij de Nationale Politie.

Conclusie

Uiteindelijk blijkt dat de ideeën achter alle oplossingen grotendeels vergelijkbaar zijn. Allemaal proberen we zoveel mogelijk te automatiseren om tijd te besparen. Daarnaast is het belangrijk dat de oplossingen flexibel zijn en de teams ondersteunen. Het is niet onze bedoeling om teams oplossingen door de spreekwoordelijke strot te duwen, want we willen ze immers helpen. Helaas zien we ook nog weleens dat er bijvoorbeeld slechts één Jenkins instantie is voor alle teams, die niet aanpasbaar is. De ervaring heeft ons echter geleerd dat dat de voortgang van de teams niet helpt. De uiteindelijke implementatie verschilt nog wel iets, maar dat is ook afhankelijk van de al beschikbare kennis en interesses.

Belangrijk in de keuze van het proces en de tools is, dat je wilt dat ontwikkelaars zich aan bepaalde standaard houden en dat je het ze zo gemakkelijk mogelijk moet maken. Vergaande automatisering is hier een krachtig mechanisme voor. Neem als voorbeeld het geval waarbij je als standaard hebt, dat het opzetten van een omgeving volledig gedocumenteerd is, zodat een nieuwe ontwikkelaar deze zonder hulp kan inrichten. Om dit af te dwingen is het beter om omgevingen te beschrijven in uitvoerbare configuratiebestanden (zoals bijvoorbeeld Docker files) en deze op te nemen in versiebeheer, dan om ontwikkelaars te vragen om een installatiehandleiding op een wiki bij te houden.