in4it's DevOps Handboek

Release 1.1

Edward Viaene

jan. 20, 2017

De themas:

Dit handboek is een technologie-agnostische bundel van documenten die helpen om software ontwikkeling en operations te kunnen verbeteren in elke organisatie. Het geheel van activiteiten die hierin beschreven worden vallen onder de noemer DevOps, een werkwijze die de samenwerking van development en operation teams moet bevorderen.

HOOFDSTUK 1

Introductie

Wat is DevOps

De term **DevOps** heeft verschillende definities, de werkelijke betekenis is verschillend in elke organisatie. Wij zien DevOps als een methodologie om software development en operations dichter bij elkaar te brengen: om beter, sneller, en efficiënter software te ontwikkelen, te testen en te releasen. Een andere vaak gebruikte term is de **delivery pipeline**. De delivery pipeline is het geheel van acties van development tot uiteindelijke release. Hoe sneller deze delivery pipeline uitgevoerd kan worden, hoe beter, en hoe sneller software in een organisatie ontwikkeld zal worden en in gebruik genomen zal kunnen worden.

Is dit van toepassing voor mijn organisatie?

DevOps is van toepassing in elke organisatie die aan development doet of zelf software ontwikkeld. Het verbeteren van de delivery pipeline in een DevOps gedreven organisatie is kostenverlagend (meer doen met minder mensen), wekt minder frustraties op (betere samenwerking), en zorgt voor een beter product voor de eindgebruiker (verhoging klantentevredenheid).

Wat bevat dit handboek?

Dit handboek heeft als doel een introductie te geven tot DevOps in de praktijk.

Wie heeft dit handboek gemaakt?

Dit handboek is gemaakt door in4it.io, een bedrijf gespecialiseerd in DevOps en Cloud. Het bevat onze ideeën en ervaringen die wij hebben opgedaan in de IT organisaties. In4it biedt advies en consultancy aan om deze beschreven manier van werken te implementeren in IT organisaties.

HOOFDSTUK 2

Version Control

Wat is Version Control

Version Control zorgt ervoor dat alle verschillende versies van de software bijgehouden worden in een centrale database of repository. Deze repository is de “golden source” van het software product. Het bevat de code die de developers schrijven en alle aanpassingen die gemaakt werden.

Waarom is Version Control belangrijk?

Een goed version control systeem zorgt ervoor dat developers samen kunnen werken in teamverband aan een softwareproduct. Zij kunnen een kopie van de code op hun systeem bewaren en deze periodiek synchroniseren met een centrale repository server. Het kan voorvallen dat developers tegelijk aan dezelfde code aan het werken zijn en dat conflicten opduiken wanneer beiden developers hun code naar de centrale repository sturen. Deze conflicten kunnen zeer tijdrovend zijn en de keuze van een goed versiecontrole systeem helpt om de productiviteit van de developers hoog te houden.

git

Git is de gouden standaard van versiecontrolesystemen. In het verleden werd er vaak gebruik gemaakt van CVS of SVN (Subversion), maar deze systemen zijn verouderd en zeker niet zo flexibel zoals git. Wat git zo degelijk maakt, is dat git een gedistribueerd versiecontrolesysteem is. Elke developer werkt met een volledige kopie van alle versies op hun systeem, maakt veranderingen en commit (bewaart) deze veranderingen lokaal. Daarna kunnen deze commits gepusht (verzonden) worden naar de centrale repository.

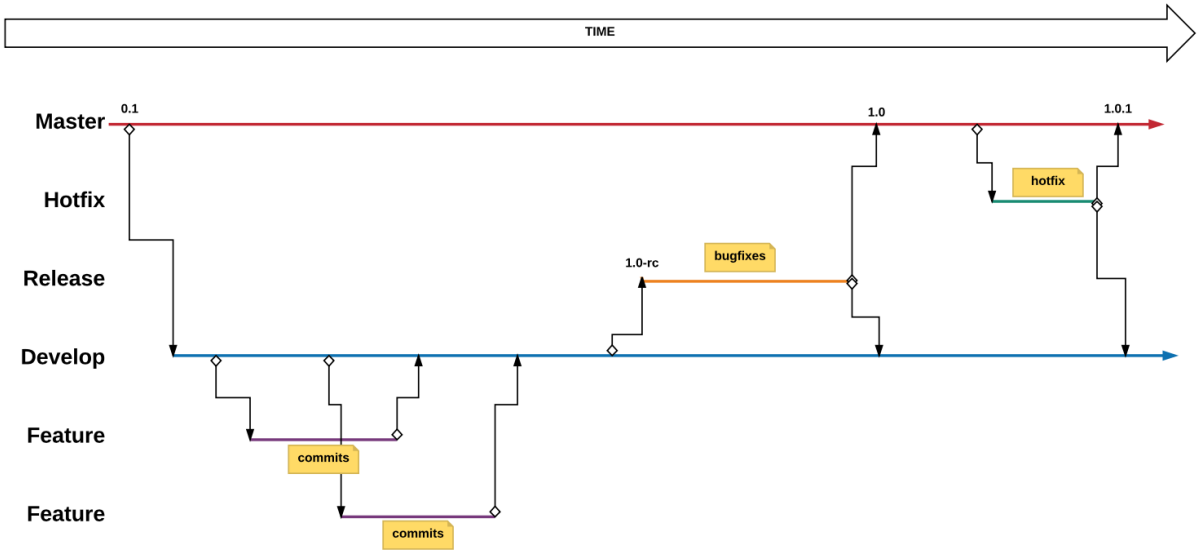
Conflicten kunnen nog steeds opduiken, maar met een goede git workflow implementatie, kunnen deze tot het minimum herleid worden. Hierna bespreken we git flow, een defacto-standaard git workflow dat door developers gebruikt wordt.

git flow

Eenmaal git in gebruik is, dient het hele team een goede workflow te volgen. Zonder een goede workflow te volgen gaat development te vaak gepaard met het oplossen van conflicten en is de kans groter dat ongeteste code

in een release terecht komt.

Git flow is een workflow dat er voor zorgt dat elke developer in zijn eigen versie kan programmeren, en deze eigen versie continu integreert met het werk dat andere developers op de gezamenlijke branch committen. Deze eigen versie noemt een branch in git.



git flow terminologie

Branch	Doel
master	Branch voor code dat uitgebracht is (release)
develop	Branch dat developers gebruiken voor te committen / branchen
feature/*	Branch waar de individuele developer kan committen
hotfix/*	Branch dat kan gebruikt worden voor hotfixes te onwtikkelen
release/*	Branch dat gebruikt wordt om een release voor te bereiden

git flow workflow

Git flow heeft altijd een master branch waar de code in zit die uitgebracht is in releases. Deze releases worden voorzien van een tag, bijvoorbeeld "1.0". Vanaf deze master branch wordt dan een nieuwe branch gemaakt met de naam "develop". Dit is de branch waar developers hun eigen feature/* branch vanaf kunnen starten. Wanneer de developer een nieuwe feature wenst uit te brengen start deze een nieuwe branch met als basis de develop branch. De developer ontwikkelt de nieuwe code en commit deze op zijn eigen feature branch. In geen geval gaat hij code rechtstreeks committen naar de develop of naar de master branch. Terwijl de developer code ontwikkelt, kan nieuwe (afgewerkte) code op de develop branch gezet worden. Dit betekent dat de developer nu in een feature branch zit te werken die niet up-to-date meer is. De developer zal daarom een "rebase" moeten uitvoeren op zijn feature branch om deze terug up to date te brengen. Op dat moment zal de developer mogelijke conflicten moeten oplossen, want zijn collega-developer heeft misschien aan dezelfde regels code gewerkt. De bedoeling is om continu, bijvoorbeeld elke dag (of vaker) een rebase uit te voeren, zodat de developer weet dat zijn feature branch up to date is met de develop branch. Wanneer de feature afgewerkt is, dan kan de developer een merge uitvoeren om de feature samen te voegen met de develop branch. Zijn feature wordt nu zichtbaar voor andere developers, die dan op hun beurt een rebase zullen moeten uitvoeren op hun feature branch.

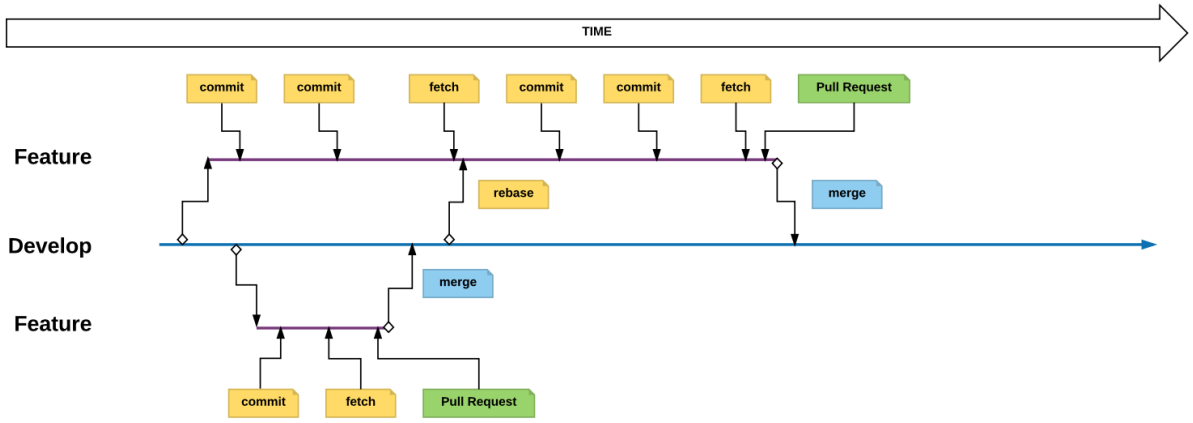
Wanneer er een release gepland wordt, kan een tijdelijke release branch in het leven geroepen worden. Deze branch dient om de release voor te bereiden. Eenmaal de release klaar is, wordt deze release branch weer samengevoegd met develop en master. In de master branch wordt dan een tag aangemaakt om de versie aan te duiden, bijvoorbeeld "1.1".

Duikt er later nog een probleem op in de release en moet er een hotfix geschreven worden, dan kan er een hotfix branch ontstaan uit de master branch. De hotfix wordt dan samengevoegd met zowel develop als master. Op master wordt er weer een nieuwe tag aangemaakt, bijvoorbeeld "1.1.1".

Door het volgen van deze methode is elke developer verantwoordelijk voor zijn feature branch. Elke developer dient zijn branch up to date te houden met de laatste versie van de applicatie in de develop branch. Elke developer is ook verantwoordelijk om zijn afgewerkte feature terug samen te voegen zonder dat er problemen ontstaan. Dit concept wordt ook Continuous Integration genoemd.

Feature workflow

Features worden ontwikkeld door developers. Dat wil zeggen dat er tegelijkertijd verschillende developers aan hun eigen features kunnen werken. Om dit in goede banen te leiden dient de developer steeds een rebase uit te voeren. Hieronder bevindt zich een gedetailleerde workflow van 2 developers die hun eigen feature ontwikkelen en de conflicten tot een minimum herleiden.



De eerste feature (bovenaan) wordt gecreëerd vanaf de develop branch. Een developer zal verschillende wijzigingen (commits) uitvoeren op deze feature branch. Terwijl maakt een andere developer zijn eigen feature branch, en maakt deze ook enkele wijzigingen. Door "fetch" uit te voeren gaat deze developer na of er wijzigingen aangebracht zijn aan develop. In dit voorbeeld zijn er geen wijzigingen, dus kan deze feature branch terug met de develop samengevoegd worden door middel van een merge. Het is mogelijk dat de developer zelf de merge niet kan uitvoeren, en een zogeheten "pull request (PR)" zal moeten uitvoeren. Deze pull request is een tussenstap voor een merge-actie. Het laat toe om andere developers zijn code te peer-reviewen alvorens deze met develop samengevoegd wordt. We gaan er vanuit dat de Pull Request succesvol was en dat de merge uitgevoerd kon worden.

De feature branch van de andere developer is op dit moment nog steeds open, en deze developer zal bij een fetch te weten komen dat de develop branch aangepast geweest werd. De nieuwe veranderingen in deze develop branch moeten nu samengevoegd worden in zijn feature branch. Dit gebeurt door middel van een "rebase". Rebase gaat er voor zorgen dat zijn features toegepast (gepatcht) worden op de nieuwe develop branch. Wanneer dit succesvol is, dan kan de developer weer enkele commits uitvoeren en dan uiteindelijk ook een Pull Request indienen om zijn feature samengevoegd te krijgen met develop.

git repository

Git repository software is noodzakelijk om git te kunnen gebruiken. Er zijn verschillende oplossingen beschikbaar, zowel self-hosted en als een service. Enkele bekende oplossingen zijn:

- Atlassian bitbucket (vroeger ook Stash genoemd - zowel hosted en als een service)
- Git Server (manueel voor kleine setups)
- GitHub (service)

- GitLab (service)
- Amazon AWS CodeCommit (service)

Testen uitvoeren

Nu dat we de code continu integreren, kunnen we op elke branch testen uitvoeren om na te gaan of de functionaliteit van onze code geen fouten bevat. Lees verder hierover in de sectie *Continuous Integration*.

HOOFDSTUK 3

Continuous Integration

Wat is Continuous Integration

Continuous Integration of CI, is een praktijk waarbij developers continu hun eigen geschreven code terug integreren met de code uit een centraal repository. Voor elke integratie wordt opnieuw het software build proces gestart, om zo fouten zeer vroeg te kunnen detecteren.

Het is belangrijk om een goede methode te vinden voor deze continue integratie van de code. Git flow (uitgelegd onder *Version Control*) is een goede methode die toelaat om code integratie gemakkelijk uit te voeren. Meermaals kan de developer een rebase uitvoeren op zijn eigen feature branch om zo nieuwe code te integreren met eigen code die ontwikkeld wordt in zijn feature branch.

Testing

Continuous Integration kan alleen goed werken wanneer de developers ook tests schrijven om de code te testen. Bij elke integratie van de code moet namelijk ook het hele buildproces van de applicatie doorlopen worden en dienen alle tests uitgevoerd te worden om na te gaan of de nieuwe aanpassingen geen nieuwe fouten heeft geïntroduceerd. Er zijn verschillende soorten testen die geschreven kunnen worden. De meest bekende types zijn:

- **Unit Test**: een test voor het kleinst mogelijk stuk code dat te testen valt
- **Integration Test**: een test die verschillende componenten samen neemt en nagaat of deze werken wanneer ze gecombineerd zijn
- **System Test**: een test die test of alle componenten tezamen werken wanneer ze gecombineerd zijn
- **Acceptance Test**: een test die ervoor moet zorgen dat de software “as designed” werkt, beschreven zoals de klant of eindgebruik het wenst
- **Regression Test**: een test die geschreven wordt wanneer er een bug opgelost wordt, om ervoor te zorgen dat dezelfde bug later niet opnieuw opduikt

Continuous Integration Software

Eerst en vooral moet er voor gezorgd worden dat het buildproces niet meer uit manuele stappen bestaat. Het buildproces dient hetzelfde te zijn voor zowel developers als voor de software die het buildproces automatisch zal uitvoeren wanneer nieuwe features ontwikkeld worden. Typisch wordt gebruik gemaakt van build tools. Deze

build tools gaan de nodige dependencies (afhankelijke modules) downloaden, de applicatie build uitvoeren, en alle testen uitvoeren. Afhankelijk van welke programmeertaal gebruikt wordt, zullen andere build tools gebruikt worden.

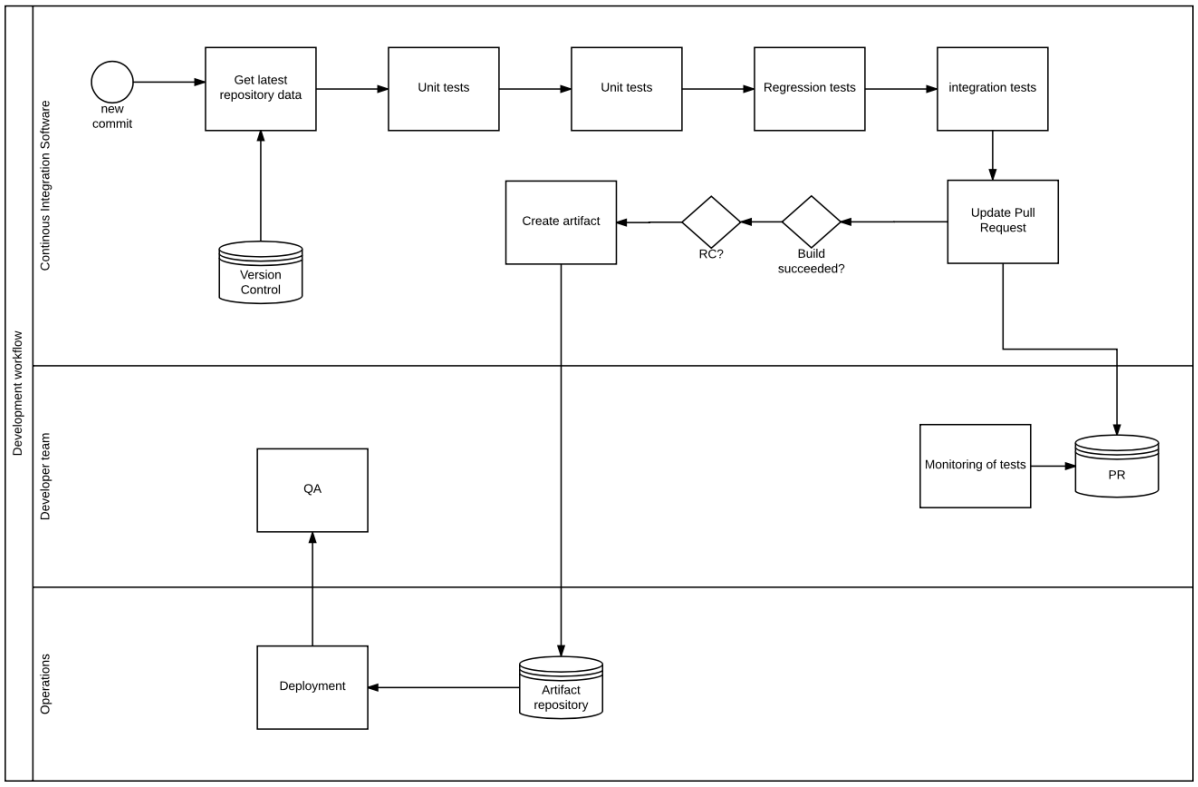
Eenmaal duidelijk is hoe het buildproces er uit ziet, kan het buildproces geautomatiseerd worden met de geschikte software. Enkele bekende software tools hiervoor zijn:

- Jenkins
- Atlassian Bamboo
- JetBrains TeamCity

Er zijn ook bedrijven die CI services aanbieden. Enkele bekende zijn:

- CircleCI
- Travis CI
- Codeship
- Wercker

Afhankelijk van welke programmeertaal gebruikt wordt om de applicatie te ontwikkelen, en afhankelijk van welke build tools gebruikt worden, zal de automatisatie er verschillend uitzien. De workflow zal echter vaak gelijkaardig zijn. Dit is een voorbeeld van een workflow:



Het is belangrijk om zo snel mogelijk te weten te komen wanneer er een nieuwe commit beschikbaar is op een branch, want bij elke commit kan er een nieuw build proces (inclusief tests) gestart worden. Indien de tests niet slagen, dan moet de developer zo snel mogelijk automatisch op de hoogte gebracht wordt, zodat deze zijn code kan aanpassen.

Pull Requests kunnen afgedwongen worden in plaats van de developer toe te laten om zelf een merge uit te voeren vanaf zijn feature branch op develop. De commits van een Pull Requests mogen alleen maar samengevoegd worden met de develop branch wanneer alle testen uitgevoerd werden en geen enkele test gefaald is.

HOOFDSTUK 4

Automated Testing

Inleiding

Zonder Automated Testing (geautomatiseerd testen), kan *Continuous Integration* niet goed werken. Wanneer de code geïntegreerd wordt, maar er worden tijdens de build & test fase geen tests uitgevoerd, omdat deze bijvoorbeeld niet bestaan, dan kan de developer niet onmiddellijk te weten komen of de code werkt.

Hoe langer de tijd tussen het schrijven van de code en het uitvoeren van de test, hoe groter de kost (technical debt) zal zijn om de fouten in de code op te sporen en op te lossen. De developer moet namelijk een context switch uitvoeren van de feature die op dit moment ontwikkeld wordt, naar het oplossen van fouten in een feature die voor de developer al (lange tijd) afgewikkeld was. De code zal ook niet meer vers in het geheugen zitten, wat extra tijd met zich mee brengt om de code logica weer te verstaan en de fouten op te lossen.

Daarom is het belangrijk om zoveel mogelijk relevante tests uit te voeren zo dicht mogelijk bij het afwerken van een feature. Als deze tests automatisch uitgevoerd kunnen worden, dan kunnen bepaalde fouten al opgespoord worden nog voor dat de developer zijn feature afsluit.

Zowel developers als testers zullen automated tests moeten schrijven. Developers zullen op hun eigen tests moeten focussen (unit tests, regression tests, etc). De testers focussen zich op de automatisatie van hun acceptance tests.

Betrek Testers mee in het development proces

Het is belangrijk om testers mee te betrekken in het development proces. Ze dienen integraal deel uit te maken van het team. In Agile methodologieën zoals Scrum is dit een vereiste. De verouderde waterfall methodologie betrok de testers veel te laat in het proces, pas nadat het development afgelopen was, wat vaak leidde tot vertragingen en hoge technical debt.

Wanneer testers deel uitmaken van de planning meetings in een sprint (in Scrum methodologie), kunnen zij onmiddellijk al de te ontwikkelen tests bepalen op basis van de features (user stories) die ontwikkeld moeten worden. De testers dienen werk in te plannen en schattingen te maken voor onder andere:

- Het aanmaken van test data
- Het aanmaken van acceptance tests voor de user stories
- Het design van test cases / acceptance test
- Het uitvoeren van de tests

- Design en/of verbeteringen van het test framework
- Andere scripting taken
- De juiste omgevingen op te zetten

Wanneer de sprint start dan gaan de testers aan de slag net zoals de developers. De testers kunnen focussen op het ontwikkelen van hun (geautomatiseerde) test cases.

In een volgende sprint kunnen de testers zich ook focussen op het ontwikkelen van automatische smoke tests en UI/regression tests dat niet ontwikkeld konden worden tijdens de sprint zelf. Het is ook belangrijk dat de testers mee deelnemen aan de review meeting en de retrospective. De testers kunnen bijvoorbeeld de interne of externe demo leiden.

Al deze acties die testers ondernemen zorgen er voor dat bugs zo snel mogelijk gevat worden zodat de kost om een bug op te lossen zo laag mogelijk blijft. Wanneer er veel later een fout opgelost moet worden dan is er niet alleen een stijging van de technical debt, maar er gaat ook kostbare tijd verloren van de developer, want deze moet een context switch uitvoeren om van huidige codebase naar de codebase van de bug over te gaan en omgekeerd. Door testers mee in het development team te brengen wordt van traditioneel testen afgestapt en wordt ook het testing team meer agile.

API tests

Applicaties hebben typisch meerdere lagen (layers). Tijdens de sprint is het al mogelijk voor de testers om volledige geautomatiseerde test cases te schrijven voor de (REST) API. Deze tests zullen eerst falen, want nieuwe API calls zullen nog niet geschreven zijn, maar naarmate het development vordert (de sprint), zullen de tests effectief de onderliggende business logic kunnen testen. Dit betekent dat van zodra een feature afgewerkt is, de tests onmiddelijk uitgevoerd kunnen worden. De feedback loop voor developer en tests is nu verkort tot uren, in plaats van dagen of weken in een waterfall methodologie.

Enkele producten die gebruikt kunnen worden voor API tests:

- HTTP Client Library for java (Google)
- Jersey Client & Jersey Test Framework (Oracle)
- Spring RestTemplate (Pivotal)
- Apache CXF Client

Het schrijven van tests

Testers hebben vaak geen (of maar een beperkte) developer achtergrond. De testers zullen een apart test framework moeten gebruiken om tests te ontwikkelen. Het is belangrijk om de implementatie van de test af te zonderen van de beschrijving van de test zelf. Dit maakt het mogelijk om testers een test scenario te laten schrijven zonder de onderliggende implementatie te hoeven verstaan. De implementatie zelf kan dan geschreven worden door een tester die meer development achtergrond heeft of een developer die tests mee helpt schrijven.

Er is software beschikbaar voor het opsplitsen van de beschrijving van de test en de implementatie zelf. Enkele bekende zijn:

- Cucumber
- JBehave

Behavior Driven Development

Software zoals Cucumber (een zeer populair pakket) laat toe om aan Behavior Driven Development (BDD) te doen. Het is mogelijk om de tests te schrijven voordat de software zelf ontwikkeld wordt. Dit kan dan dienen als specificatie voor de developers om de software te ontwikkelen. De developers zien dan onmiddellijk of ze een

fout gemaakt hebben in de implementatie van feature. De Cucumber scenarios kunnen zo geschreven worden dat toekomstige teamleden begrijpen wat het systeem doet door alleen maar de test scenarios te lezen.

Het is ook mogelijk om UI tests uit te voeren met een combinatie van bijvoorbeeld Cucumber + Selenium, maar de echte kracht van een framework zoals cucumber zit in het uitdrukken van business rules in test scenarios. Een voorbeeld van zo een scenario in Gherkin (de beschrijvende taal in Cucumber):

```
Scenario: Some determinable business situation
  Given some precondition
    And some other precondition
  When some action by the actor
    And some other action
    And yet another action
  Then some testable outcome is achieved
    And something else we can check happens too
```

Dit scenario kan dan in bijvoorbeeld in Java geïmplementeerd worden. De implementatie is losgekoppeld van het beschrijven van het scenario zelf. Elke tester kan dit scenario ontwikkelen, maar waarschijnlijk zullen niet alle testers even comfortabel zijn om de implementatie te schrijven voor de test.

HOOFDSTUK 5

Release Management

Wat is Release Management

Release Management is het proces van plannen, beheren, en uitvoeren van de software build, inclusief het testen en uitrollen (deployen) van de software release. De release wordt typisch uitgerold over verschillende omgevingen (ook wel zuilen genoemd). Deze omgeving start met een development omgeving, gaat dan naar een testomgeving, een Accept of QA (Quality Assurance) omgeving en dan pas naar een productie omgeving, waar de software beschikbaar is voor de eindgebruiker. Vaak is de test en Accept/QA omgeving dezelfde omgeving, dus in verdere paragrafen spreken we alleen over DEV, QA, en Productie (vaak afgekort als PROD of PRD).

In *Version Control* hebben we besproken hoe een release in git voorbereid kan worden. In het versiecontrolesysteem zal een versie klaar staan die het build proces succesvol heeft doorstaan en klaar is om gebruikt te worden. De term die vaak gebruikt wordt voor een versie die nog niet in productie staat is Release Candidate (RC). Wanneer gestart wordt met een nieuwe release, van bijvoorbeeld versie 2.0, dan worden er vaak eerst enkele release candidates uitgerold. Deze versies worden dan gemarkeerd met 2.0-rc1, 2.0-rc2, enz. Wanneer de build stabiel genoeg blijkt te zijn dan wordt de release gedaan van de finale versie.

Het uitrollen van de software op een omgeving wordt beschreven met de term deployment of kortweg deployen van een omgeving. De deployment procedure kan heel wat tijd in beslag nemen als deze niet geautomatiseerd is. Een van de eerste zaken dat daarom geautomatiseerd wordt bij een transitie naar een DevOps gedreven organisatie is het deployment proces.

Na de build

Nadat de build uitgevoerd is geweest kan de software deployment doorgaan. Het resultaat van het build proces is vaak een archief. Dit kan de vorm aannemen van een zip/tar/jar bestand, een installatiepakket voor linux (.deb, .rpm), of een installatiepakket voor andere besturingssystemen. Dit is het artifact, het bestand of de bestanden dat nodig zijn om de deployment te kunnen uitvoeren op de verschillende omgevingen.

Dit archief wordt best goed beheerd. Er bestaat software dat gespecialiseerd is in het managen van deze artifacts en hun dependencies:

- JFrog Artifactory
- Sonatype Nexus Repository
- apt/yum repositories voor Linux artifacts

Wanneer deze artifacts en hun dependencies goed beheerd worden is het gemakkelijker om oude releases terug te vinden, maar ook om rollbacks te kunnen uitvoeren of om de dependencies bij te houden in het geval dat een externe dependency niet meer teruggevonden kunnen worden.

Deployment

Het deployen van een applicatie is in veel organisaties een handmatig en intensief werk. Door middel van automatisatie kan echter de deployment van een applicatie op DEV, QA en PROD tot vrijwel een routine opdracht teruggedrongen worden. Er is veel software op de markt die helpt om het deployment proces te automatiseren. Aan de ene kant dienen de fysieke of virtuele machines klaar te staan met het juiste OS (besturingssysteem), aan de andere kant heb je het deployment proces zelf dat de software en libraries moet installeren.

Het beheren van de software op de servers zelf kan gedaan worden met automatiseringssoftware. Enkele bekende tools zijn:

- Puppet
- Saltstack
- Ansible
- Chef

Deze tools zijn allemaal zeer verschillend, maar hebben hetzelfde doel: server (of in het algemeen machine) administratie automatiseren. Ze doen dat allemaal met de "infrastructure as code"-methode, waarbij het operations team definities in code schrijft over hoe het uiteindelijke systeem er moet uitzien. Door code te gebruiken in plaats van manueel commandos uit te voeren zijn onmiddellijk enkele voordelen zichtbaar:

- Er is een audit log
- Historiek van alle veranderingen
- De regels kunnen periodiek opnieuw toegepast worden, om zo niet-goedgekeurde manuele wijzigingen tegen te gaan
- Dezelfde code kan uitgevoerd worden op meerdere machines, of meerdere omgevingen

Infrastructure automation

Het stopt niet bij de automatisatie van de software zelf. Ook de installaties van de virtuele / fysieke machines kan geautomatiseerd worden. Tegenwoordig is virtualisatie overal in de organisatie doorgedrongen. Software wordt niet meer op een fysieke machine geïnstalleerd, maar op een virtuele machine (VM). Deze VM kan vaak automatisch geïnstalleerd worden, opnieuw door gebruik van definities in code. Het is niet meer nodig om manueel een OS te installeren, dit kan allemaal automatisch gebeuren. Enkele software tools die hiervoor gebruikt worden:

- Terraform (voor infrastructure provisioning, werkt het best met public cloud providers)
- Vagrant (voor development)
- Packer (voor VMware of Cloud images)
- Docker (voor containers - zie de *Containers* pagina)

Dit betekent dat het volledige release proces van machine en OS tot dependencies en software volledig geautomatiseerd kan worden. Dit is belangrijk om de transitie te kunnen maken naar een echte DevOps-driven organisatie. Wanneer deployments zeer snel uitgevoerd kunnen worden, dan kan de tijd tussen 2 deployments drastisch verlaagd worden. Bedrijven die Devops toepassen, kunnen gemakkelijk verschillende deployments per dag uitvoeren.

Platform as a Service

Een alternatief voor infrastructure automation is Platform as a Service (PaaS). Wanneer PaaS gebruikt wordt, dan wordt het onderhoud van de onderliggende infrastructuur zoveel mogelijk uit handen gegeven. Deze platforms bieden de meest populaire programmeertalen aan, maar verwachten ook wel van de developers om bepaalde wijzigingen aan te brengen aan hun applicatie code om deze op hun platform te laten werken. Voor applicaties die op de cloud draaien kan dit een goede oplossing zijn, om zo snel applicaties te kunnen uitbrengen. Voor on-premise oplossingen, moet eerst de PaaS-software opgezet worden, wat redelijk wat tijd en mankracht in beslag kan nemen. Het is vaak dan ook alleen maar een optie voor grotere organisaties om dit on-premise te doen. Enkele populaire PaaS systemen zijn:

- Heroku (cloud)
- AWS Elastic Beanstalk (cloud)
- Google App Engine (cloud)
- Microsoft Azure PaaS (cloud)
- Dokku (single-node), Deis (multi-node) voor on-premise
- Openshift (on-premise)
- PaaS services for Azure Stack (on-premise)

Veel van deze oplossingen ondersteunen ook containers, gebruik makend van Docker. Lees hierover meer in de sectie over *Containers*.

Deployment Strategies

Zero downtime deployments

Er zijn verschillende strategieën die gevolgd kunnen worden om deployments uit te rollen. Gaat het om een web-applicatie, dan betekent een outage een mogelijk verlies in omzet. Daarom opteert men best voor zero downtime deployments: het uitvoeren van deployments zonder een seconde downtime. Dit is allemaal mogelijk zolang de applicatie hierop voorzien wordt. Alle PaaS oplossingen zijn bijvoorbeeld voorzien om zero downtime deployments uit te voeren, wat betekent dat dit een zeer populaire strategie om te volgen is. Hoe applicaties ontwikkeld moeten worden is beschreven in *12-Factor apps*

Blue/Green deployments

Blue/Green deployments worden gebruikt om zero downtime deployments te bereiken. Bij blue/green deployments wordt de nieuwe release eerst naast de huidige applicatie uitgerold. De stabiele, oude, release kan dan de green deployment zijn, de nieuwe deployment de blue deployment. Pas wanneer de blue deployment (de nieuwe release) volledig uitgerold is en live testen doorstaat, dan pas wordt de blue deployment op actief gezet. Op deze manier kunnen de Green/Blue omgevingen snel wisselen, om zo een zero-downtime deployments te kunnen uitvoeren. Deze techniek wordt zeer vaak toegepast in web applicaties.

Canary deployment

Een build dat succesvol is betekent nog niet dat de applicatie bugvrij is. Om de impact van nieuwe releases te beperken kan gekozen worden voor een Canary deployment strategie. In deze strategie wordt de applicatie slechts uitgerold voor een beperkt publiek, bijvoorbeeld 5% of 10% van de gebruikers. Zo is het mogelijk om snel fouten vast te stellen die alleen maar voorvallen bij deze kleine groep van eindgebruikers. Dit betekent dat niet alle eindgebruikers de impact zullen voelen als er een nieuwe bug geïntroduceerd werd in de release. Dit geeft een veel betere ervaring naar de eindgebruiker toe.

A/B Testing

A/B testing kan gebruikt worden voor feature testing. In dit geval worden bepaalde features uitgetest om de gevolgen ervan te kunnen waarnemen. Een grotere populatie zal de nieuwe feature zien. Deze groep wordt dan van dichtbij geobserveerd om na te gaan welke impact de nieuwe feature heeft en of de feature het gewenste effect bereikt.

HOOFDSTUK 6

Cloud

Wat is Cloud

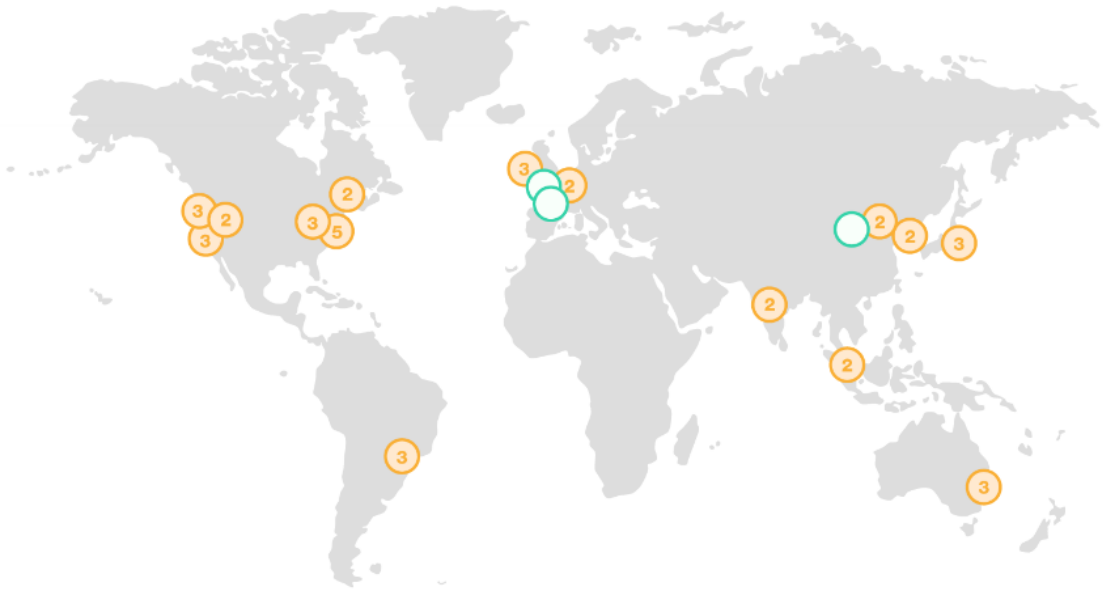
Cloud computing maakt het mogelijk om on-demand compute processing en data opslag ter beschikking te krijgen. De meeste cloud providers laten toe om te betalen per uur dat een resource gebruikt wordt, wat toelaat om kapitaalintensieve uitgaven te vervangen door een kostenmodel. Het eindresultaat is bijna altijd kostenbesparing, aangezien in het traditionele model aangekochte fysieke hardware vaak onderbenut gaat en snel verouderd is. Het pay-as-you-go model laat toe om een machine en/of opslag te huren voor een bepaalde tijd, om dan later te beslissen om een totaal ander (nieuwer) type machine te starten, afhankelijk van de noden. Door veel sneller te kunnen inspelen op de echte noden, kan een kostenbesparend resultaat gemakkelijk behaald worden.

Veel clouddiensten worden ook “As a Service” aangeboden, wat wilt zeggen dat het beheer uit handen gegeven wordt. Dit laat bedrijven toe te focussen op hun core business, en minder tijd en geld te hoeven investeren in het onderhouden van besturingssystemen, machines, netwerken, en opslagapparatuur. Bij on-premise datacenters gaat het zelfs ook over besparingen voor koeling, energie, backup-stroom en bewaking.

De trend, gestart in de Verenigde Staten, is om eigen datacenters af te bouwen in het voordeel van cloud en de volledige infrastructuur te laten managen door cloud providers. Enkele bekende publieke cloud providers zijn:

- Amazon AWS (de grootste speler)
- Microsoft Azure
- Google Cloud

Om een idee te geven hoe groot publieke cloud is, onderstaande afbeelding toont de globale infrastructuur van Amazon. Het nummer duidt het aantal Availability Zones aan (onafhankelijke zones in een regio).



De Amazon AWS diensten alleen al zijn goed voor meer dan 10 miljard dollar omzet per jaar en elk jaar groeit het belang van deze diensten.

Wat is Private Cloud

Door de opmars van de mogelijkheden van cloud computing, is er software ontwikkeld om de flexibiliteit van public cloud ook on-premise te kunnen hebben. Dit werd private cloud gedoopt. De hardware moet nog steeds aangekocht en onderhouden worden, maar de software laat toe om op deze hardware de management software te draaien die het toelaat om flexibeler compute en storage op te beheren. Bekende spelers zijn:

- Microsoft Azure Stack (zelfde technologie als Azure, maar dan on-premise)
- OpenStack

Private cloud is geschikt voor bedrijven die de schaalgrootte hiervoor hebben. Vaak kunnen ze van de regulators (bv. banken regulator) geen data in publieke cloud plaatsen. In alle andere gevallen is publieke cloud aangewezen.

Hoe past Cloud in het DevOps verhaal?

DevOps steunt op het snel kunnen uitvoeren van de delivery pipeline tussen development en release. Hoe sneller releases uitgerold kunnen worden hoe beter. Zonder een goede test en deployment strategie is dit moeilijk haalbaar. Het on-demand resourcemodel past zeer goed bij dit verhaal. Extra compute capaciteit kan gebruikt worden om meer testen uit te voeren, of om machines te starten in verschillende omgevingen zodat deze getest kunnen worden. Dit is ook haalbaar zonder publieke cloud, maar moeilijker en veel duurder te verwezenlijken. Extra en vooral tijdelijke compute capaciteit is vaak moeilijk beschikbaar of zeer duur in organisaties die geen publieke cloud technologieën beschikbaar hebben. De lead times voor nieuwe hardware lopen vaak op tot maanden wachttijd.

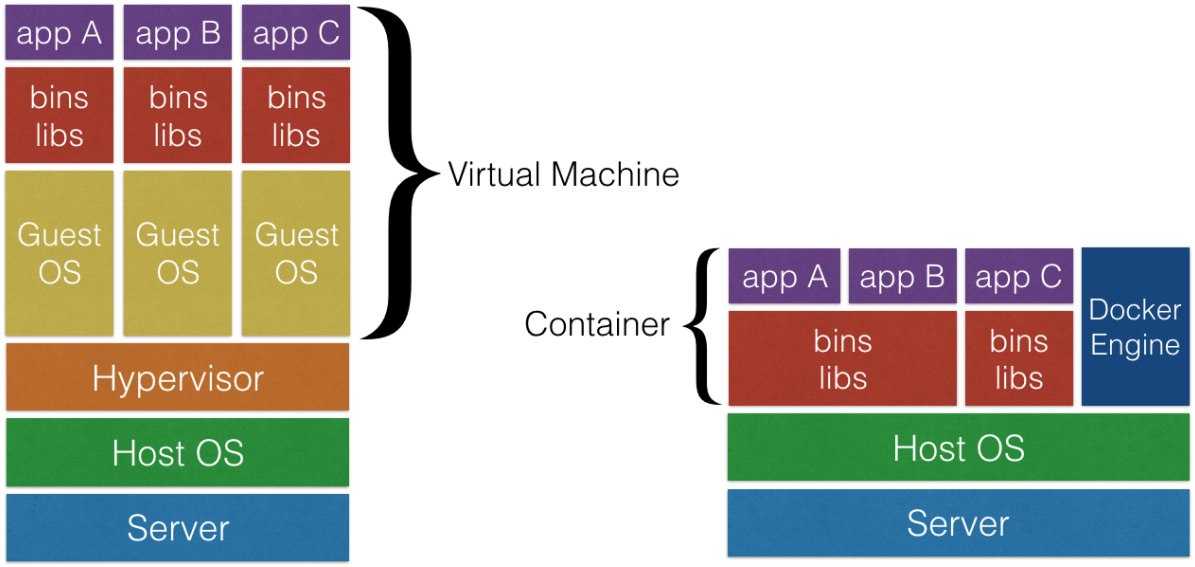
Het "as a Service"-model van publieke cloud past ook goed in het managen van deze delivery pipeline. Er zijn services beschikbaar die kunnen helpen om deze pipeline zo goed mogelijk te ontwikkelen zodat tests en deployments zeer snel uitgevoerd kunnen worden. Amazon gebruikt deze services zelfs intern voor hun eigen projecten, die ook deze DevOps principes gebruiken.

HOOFDSTUK 7

Containers

Wat zijn Containers?

Containers is een soort van virtualisatietechniek, maar dan op het niveau van het besturingssysteem. Traditionele virtualisatie gebruikt een hypervisor. Hierbij dient het hele bootproces van het besturingssysteem doorlopen te worden. Containers daarentegen zijn veel lichter, en kunnen veel sneller gestart worden. Op dit moment is het echter alleen maar mogelijk om linux-compatible-software in containers te gebruiken. Dat staat niet in de weg dat linux containers wel op windows besturingssystemen werken en ook ondersteund worden.



Wat is Docker?

Docker is de populairste container software voor Linux. Docker Engine is software dat geïnstalleerd kan worden om containers aan te maken, te managen en te gebruiken. Docker is beschikbaar voor zowel Windows, Linux, als MacOS. Er is ook Docker Hub, dat "as a service" producten aanbiedt om de build van containers online door hen te laten gebeuren. Ze bieden ook een repository server aan waar container images (vergelijkbaar met een VM image) gehost kunnen worden. Docker images kunnen zowel publiek als private (afgeschermd met credentials) gehost worden.

Wat zijn de voordelen?

Het gebruik van containers heeft tal van voordelen:

- Containers zijn kleiner dan VM images (megabytes in plaats van gigabytes)
- Containers kunnen gestart worden in minder dan een seconde (een VM starten is typisch enkele minuten)
- Containers kunnen gemakkelijk gebundeld worden (bv. webapp + database software in aparte containers)
- Dev-prod pariteit: dezelfde containers kunnen zowel in development als in productie gebruikt worden * Dit zorgt ervoor dat software dat lokaal getest wordt **ook** op de productieserver werkt
- Container images bestaan uit lagen (layers), wat meer flexibiliteit biedt en de images kleiner maakt

Hoe past dit in het DevOps verhaal?

Containers brengen developers en operations dichter bij elkaar. Door dezelfde container images te gebruiken op de development machines als op productie, verkleint een verschil tussen dev-qa-prod omgevingen. De flexibiliteit laat toe om zeer snel nieuwe containers te bouwen en deze gemakkelijk te verspreiden binnen de organisatie.

Containers in productie

Google is al meer dan 10 jaar overgestapt op een container-driven infrastructuur. Met al de ervaring dat Google heeft in de containerwereld hebben ze een open source software pakket uitgebracht voor het managen van containers in productie, Kubernetes genaamd. Deze software en software van andere leveranciers wordt gebruikt om containers in productie te gebruiken. Een overzicht van de belangrijkste spelers:

- Kubernetes (door Google)
- Mesos
- Docker swarm (door Docker)

Docker Orchestration tools zoals Kubernetes laten toe om containers zowel on-premise te gebruiken alsook op publieke cloud. Voordien was er altijd een groot verschil tussen software deployments on-premise en software deployments op de cloud. Deze orchestration tools laten toe om het datacenter volledig abstract te beschrijven om zo dezelfde manier van deployen toe te passen op eendere welke infrastructuur. Dit heeft het grote voordeel dat er geen verschillende systemen nodig zijn voor de verschillende datacenters/clouds dat een organisatie kan gebruiken.

Past Docker in mijn organisatie?

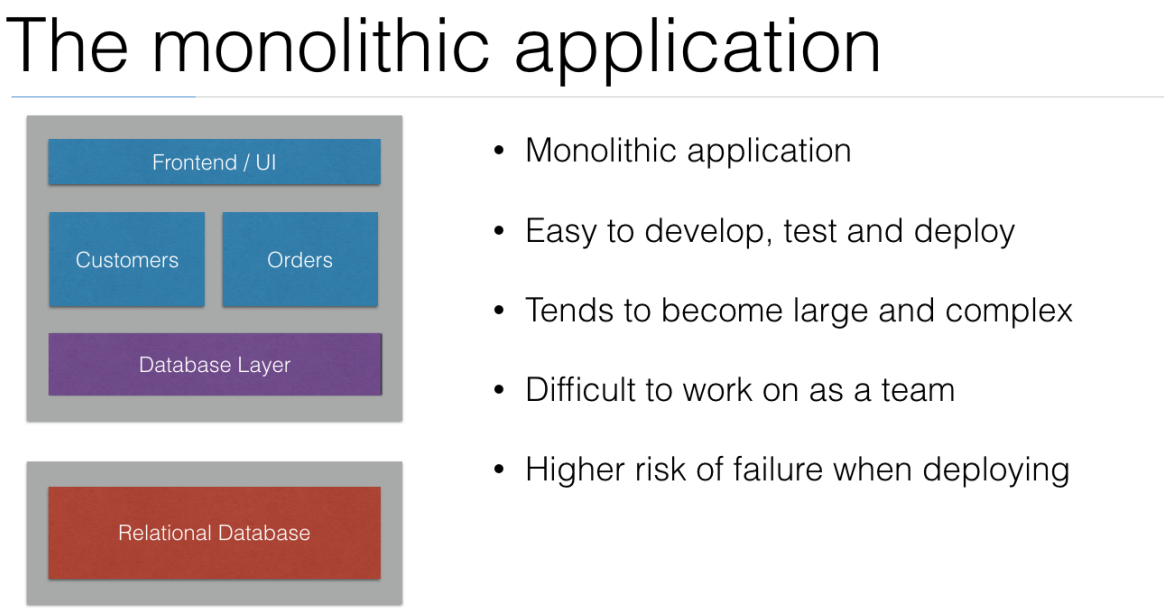
Zeer waarschijnlijk past Docker ook in uw organisatie. Veel leveranciers van software zijn mee op de containerkar gesprongen en bieden standaard containers aan voor hun software. Het kan een goede manier zijn om een flexibele en robuuste development omgevingen op te bouwen door middel van containers. Dat is ook waar de meeste organisaties mee beginnen: de development omgeving. Als het goed in development draait dan kan de stap naar QA/ACCEPT en productie gezet worden. Deze laatste stap is heel wat groter door de keuze van orchestration software die in productie dient te draaien.

HOOFDSTUK 8

Microservices

Wat zijn microservices

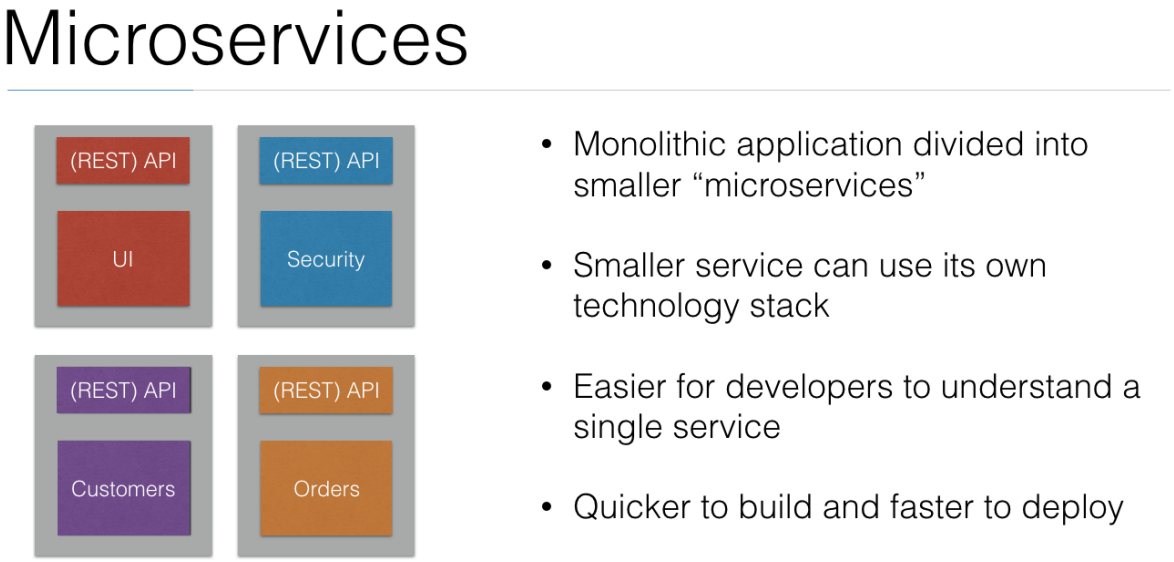
De microservices architectuur is een implementatiemethode in applicatieontwikkeling. Om microservices te begrijpen is het interessant om eerst het tegengestelde van microservices te bespreken, monolithische applicatie. Monolithische applicaties hebben één grote codebase dat als één geheel het build en deployment proces moet doorlopen. Het grote voordeel van een monolitisch app is dat deze gemakkelijk te ontwikkelen, testen en deployen is. Hieronder een voorbeeld hoe een monolithische applicatie er kan uitzien:



De nadelen van een monolithische app is dat deze over tijd groot en complex kunnen worden. Het is moeilijker om met in teamverband aan de applicatie te werken en deployen heeft een hoger risico. Gaat er iets mis bij de uitrol, dan is het risico niet beperkt tot een onderdeel, maar kan het invloed hebben tot heel de applicatie.

Microservices daarentegen zijn opgedeeld in kleinere onafhankelijke applicaties. Deze communiceren dan onderling met elkaar. De voordelen van microservices zijn dat het gemakkelijker is voor developers om 1 enkele microservice te begrijpen, het is niet onmiddellijk noodzakelijk de hele applicatie te verstaan. Omdat de applicatie kleiner is, is het build proces ook korter en zal de deployment sneller afgewerkt kunnen worden. Ook is het mogelijk om nieuwe technologieën te gebruiken voor een enkele microservice,

omdat deze toch onafhankelijk is van de rest van de applicatie. Een microservice kan er zo uitzien:



Zijn microservices interessant voor mijn organisatie?

Microservices zijn interessant wanneer er met een groot team aan een softwareproject gewerkt wordt. Voor kleinere teams is dit minder interessant. Voor grotere teams maakt de microservice architectuur het mogelijk om de teams op te splitsen in kleinere teams. Ieder team zal dan verantwoordelijk zijn voor hun eigen microservices, wat de teams ook onmiddellijk efficiënter maakt.

Wanneer de microservices architectuur gebruikt wordt, dan wordt vaak ook containers (meestal Docker) gebruikt als infrastructuurlaag. Dit omdat gemakkelijk verschillende containers met microservices gestart kunnen worden. Deze microservice kan dan uitgerold worden op een development machine, maar ook dezelfde image kan daarna in productie horizontaal geschaald worden op verschillende machines. Docker orchestration komt hier helemaal in het recht doordat het zeer snel containers kan plaatsen op machines waar er resources beschikbaar zijn.

HOOFDSTUK 9

Metrics voor de DevOps Organisatie

Evolueren naar een DevOps gedreven organisatie is een lange weg dat constante monitoring nodig heeft. Hieronder zijn enkele indicatoren die kunnen weergeven of er verbeteringen zijn in de delivery pipeline.

Build failures

Build failures is een belangrijke indicator. Hoe meer build failures er zijn, hoe vaker code gecommit wordt met fouten in, of code dat tests doet falen. Hoe vaker de developer aanpassingen moet maken aan zijn code om deze fouten op te lossen, hoe langer het zal duren om een feature af te werken. De trend dat een organisatie wenst te zien is minder build failures, wat kan duiden op hogere developer velocity.

Test Coverage

Zonder volwaardige test coverage kunnen fouten niet opgespoord worden. Het is belangrijk om tests te schrijven voor de applicatie, zodat fouten vroeger opgespoord kunnen worden. Software dat code analyse kan doen, zal deze test coverage indicatoren genereren.

Change Volume

Hoeveel commits en changes zijn aangebracht in een bepaalde tijd? Hoeveel tickets en stories werden afgewerkt? Deze indicator kan gebruikt worden om na te gaan of er veel verandering was in een bepaalde periode, en of dit gevolgen had op andere indicatoren, zoals bijvoorbeeld availability.

Lead time (van development tot productie)

Het is zeer belangrijk om op te volgen hoe lang het duurt om van de development van een feature te komen tot de uiteindelijke deployment van de feature in de productieomgeving. Hoe korter deze tijd, hoe sneller nieuwe features beschikbaar zijn voor de eindklant en hoe hoger de klant zijn tevredenheid over de applicatie zal zijn.

Deployment frequency

Hoe vaak worden deployments uitgevoerd? Hoe meer deployments elkaar kunnen opvolgen hoe beter, want dat wil zeggen dat nieuwe features en fixes uitgerold worden in productie. Lange deployment frequencies wijst vaak op lange, moeilijke deployments met heel veel manueel werk. Na de volledige automatisatie van de deployment procedure, zou de organisatie een verlaging van deze indicator moeten zien.

Failed deployments

Hoge deployment frequency met een hoge failed deployments indicator is dan ook weer niet goed. Elke failed deployment zal impact hebben naar de eindgebruiker en zal klantentevredenheid doen dalen. Het is belangrijk om zowel op korte termijn deployments te kunnen uitrollen, maar ook zonder problemen en zonder onderbrekingen.

Availability (beschikbaarheid)

Eindgebruikers verwachten de dag van vandaag 24/7 applicaties met amper tot geen onbeschikbaarheid. Availability wordt meestal gemeten in een percentage, bv 99.99% in de laatste maand. Eenmaal de deployments geautomatiseerd zijn en er genoeg test coverage is, dan zou er een stijging moeten waargenomen kunnen worden in de beschikbaarheid.

Mean time to recovery (MTTR)

Gaat er iets mis, hoe lang duurt het dan om een hersteloperatie uit te voeren? Deze indicator toont hoe goed dat de dev en ops teams kunnen omgaan met de veranderingen van de applicatie. De verwachting van deze indicator is dat deze verlaagd wordt over tijd.

Incident Volume

Incident volume indicators kunnen duiden op problemen met de applicatie of de onbeschikbaarheid ervan. Belangrijk is om het incident volume op te volgen en root cause analyses uit te voeren wanneer een verhoging in incident volume waargenomen werd.

HOOFDSTUK 10

12-Factor apps

Hoe bouw ik applicaties

de 12-Factor app methodologie geeft een antwoord hoe de dag van vandaag, in een wereld van private en public cloud, applicaties dienen gebouwd te worden. Deze pagina geeft onze interpratatie van de 12-factor app op basis van het originele document (externe link).

De 12-Factor applicatie is een methodologie voor webapplicaties of software ontworpen als software-as-a-service applicaties. Applicaties die ontwikkeld zijn met deze methodologie hebben de volgende kenmerken:

- De applicaties hebben geautomatiseerde setups door middel van declaratieve scripts
- Ze zijn Besturingssysteem onafhankelijk, en bereiken zo maximum portabiliteit
- Kunnen opgezet worden op moderne cloud platformen (public en private cloud)
- De omgeving heeft een minimaal verschil tussen development en productie
- De app moet continue uitgerold kunnen worden (Continuous Deployments)
- Schaalbaar zonder significante aanpassingen aan de tools, architectuur of development methodes

De 12-Factor app methodologie kan gebruikt worden voor eender welke programmeertaal, gebundeld met externe technologieën zoals databases, caching software, enz.

Voordelen

Enkele voordelen van de 12-factor app methodologie:

- Het voorkomen van software erosie. Applicaties die ontwikkeld worden, hebben vaak een hoge operationele kost. Het is meestal niet mogelijk de applicatie op nieuwere infrastructuur te plaatsen. Door de 12-factor methodologie te volgen wordt dit zoveel mogelijk gemitigeerd.
- De app kan zowel op private, als publieke, als container uitgerold worden
- De 12-factor app methodologie bestaat sinds 2012 en nog is steeds een goede methodologie voor applicatieontwikkeling

De 12 factors

Hieronder bespreken we de 12 factors. Wanneer deze factors toegepast worden tijdens het ontwikkelen van de applicatie, dan spreken we van een applicatie ontwikkeld met de 12-factor methodologie.

I. Codebase

Er is één enkele codebase voor de applicatie. Deze codebase zit in een versie controle systeem (bv git).

- Worden verschillende codebases gebruikt, dan dient elke app onafhankelijk te zijn, zoals in het geval van microservices (zie *Microservices*)
- Sharen verschillende apps een codebase om hergebruik van code te bevorderen, dan dient er met libraries en dependencies gewerkt te worden

II. Dependencies

Dependencies dienen expliciet gedefinieerd te zijn en dienen geïsoleerd te zijn van de applicatie. Enkele voorbeelden:

- Java heeft Maven / Ant / gradle om dependencies te beheren
- NodeJS heeft packages.json
- PHP heeft composer
- Ruby heeft Gems

III. Config

Een applicatie configuratie is alles wat kan verschillen tussen de verschillende omgevingen (dev, test/QA/accept, productie). Deze configuratie moet strikt gescheiden worden van de code. Dit zijn bijvoorbeeld credentials (logins en wachtwoorden), paden (paths), en dergelijke.

De twelve factor app raadt aan om deze in omgevingsvariabelen (environment variables) te zetten. Er zijn echter nog andere mogelijkheden:

- Een gedistribueerde key/value store zoals consul, zookeeper of etcd kan gebruikt worden om configuratie instellingen op te slaan
- Container Orchestration software (zoals Kubernetes) bieden vaak ook eigen oplossingen aan zoals een credential store en een config store. Vaak is dit toch ook wel een combinatie van een key value store en environment variables. Soms worden ook (tijdelijke) bestanden gebruikt op basis van de credential store / config store.
- Soms worden aparte repositories gebruikt puur om configuratie op te slaan, dit kan echter snel omslachtig worden

IV. Backing Services

Backing Services, zoals een database, caching server, search software, dienen gezien te worden als een externe service. Het enige dat de applicatie hoeft te weten is de connectie informatie die in de configuratie ingesteld dient te worden. De externe services worden dan onafhankelijk van de applicatie beheert en onderhouden.

V. Build, release, run

Een strikte scheiding van build, release, en run stages:

- De build stage gaat de software compilen, builden, testen en geeft als resultaat een build terug (artifact)
- De release stage combineert de build met de configuratie variabelen specifiek voor een omgeving (bv. build + production config)
- De run stage is de runtime, het is het opstarten van een specifieke release

Deze stappen zijn ook zeer duidelijk wanneer gebruik wordt gemaakt van containers. Een container wordt eerst gebouwd als een image (build). Daarna dient een specificatie geschreven te worden voor een release (build + config). Daarna kan de specificatie uitgevoerd worden en is de app live (run stage).

VI. Processes

De applicatie dient uitgevoerd te worden als 1 of meerdere stateless processen. Stateless betekent dat de applicatie geen "state" mag bewaren. Het kan geen bestanden lokaal wegschrijven en ook gebruikerssessie-informatie dient in een externe service opgeslagen te worden. Deze factor is zeer belangrijk, maar ook vaak een moeilijke om te implementeren als de applicatie niet stateless ontwikkeld geweest is.

Eenmaal de applicatie stateless is, kan deze gemakkelijk horizontaal geschaald worden. Dit schalen dient te gebeuren wanneer de applicatie meer requests af te handelen heeft. Omgekeerd ook, de applicatie kan horizontaal downscalen wanneer minder of geen eindgebruikers de applicatie nodig hebben, bijvoorbeeld buiten de kantooruren of 's nachts.

VII. Port binding

De applicatie dient volledig self-contained te zijn. Veel applicaties dienen nog vaak gebundeld te worden met een webserver (bv. Apache httpd / Tomcat). De applicatie dient er zelf voor te zorgen dat het aan poort binding doet en dat een vaste poort beschikbaar is om de applicatie aan te spreken. Enkele voorbeelden:

- Java kan Jetty gebruiken
- NodeJS kan een Express server opzetten

Dit neemt niet weg dat er uiteindelijk toch een webserver voor de applicatie geplaatst wordt (bv een load balancer of reverse proxy). Deze loadbalancer of proxy is dan volledig onafhankelijk en stuurt gewoon het dataverkeer door naar de onderliggende applicatielaag. Het voordeel van loadbalancers is dat de load gesplitst kan worden over de verschillende processen, en er geen dataverkeer gestuurd wordt naar processen die niet "healthy" zijn, bijvoorbeeld in het geval van software of hardware problemen.

VIII. Concurrency

Schalen dient te gebeuren via het process model. Er kunnen verschillende processen gestart worden voor verschillende workloads. Bijvoorbeeld voor long-running processen kan een apart worker proces gestart worden. Deze processen zijn volledig onafhankelijk van elkaar, ze mogen niets met elkaar delen, en moeten beiden onafhankelijk horizontaal kunnen schalen. Zo kan een worker bijvoorbeeld onafhankelijk meerdere processen starten om een piek op te vangen in het afhandelen van long-running processen.

IX. Disposability

De applicatie moet horizontaal kunnen schalen en zal dus verschillende instanties van elke applicatie draaien als afzonderlijke processen. Omdat onderliggende hardware altijd kan falen, en omdat een downscale event op elk moment kan voorvallen, dienen processen van applicaties gemakkelijk verwijderbaar (disposable) te zijn. Een applicatieproces moet snel zijn laatste acties kunnen afwerken en kunnen overgaan in een shutdown zonder gegevensverlies. Dit zorgt ervoor dat de applicatielaag robuust is en snel kan reageren in het geval van deployments.

Bij het vaak en snel uitvoeren van deployments moeten applicaties ook snel genoeg kunnen opstarten, anders dan duurt een deployment te lang, of kan er niet snel genoeg geschaald worden om de veranderingen (bvb meer gebruikers) op te vangen.

X. Dev/prod parity

Development, staging (test/QA/accept), en productie moeten zo gelijk mogelijk zijn.

XI. Logs

Logs zijn event streams en dienen niet per machine meer opgeslagen te worden in logfiles. Het is beter om de log file naar de standaard output te sturen, zodat deze dan opgevangen kan worden door andere software. Vaak wordt aggregatie software gebruikt om de output van de applicatie op te vangen en deze dan te aggregeren in een externe service. Deze service kan dan de logs weergeven in een dashboard.

XII. Admin Processes

Alle administratieve processen, zoals database migraties, of een terminal/console starten voor debugging, dienen altijd eenmalig te zijn en in dezelfde omgeving uitgevoerd te worden als de applicatie zelf. Deze code moet gebundeld worden met de applicatie zelf, zodat er geen versie mismatch kan zijn tussen de administratiecode en de applicatie