FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Systeemontwerp

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 2 januari 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

Inleiding

- Systeemontwerp = het ontwerpen van een infrastructuur waarbij verschillende componenten met elkaar kunnen interageren.
- Typische high level architectuurblokken:
  - transactiebehandeling: requests behandelen van gebruikers.
  - business intelligence: geproduceerde data analyseren.

1.1 Softwarearchitectuur

1.1.1 Systeemrequirements

- functionele requirements: specificatie wat een systeem moet **doen**.
- niet-functionele requirements: specificatie wat een systeem moet **zijn** (kwaliteitseisen).

1.1.2 4+1 view model

- Logical view:
  - Bevat de klassen, packages, relaties tussen klassen, associaties tussen de klassen (domain class diagram en entity-relationship diagram)
- Implementation view:
  - Bevat de output van het build systeem, zoals de verschillende modules (bv JARs) en componenten (executables, WARS).
  - Beschrijft de onderliggende relaties tussen alle modules en componenten (import, use, merge, ...)
- Process view:
  - Bevat de beschrijving van de werking van verschillende processen (een proces kan een hele module zijn) (activity diagram).
  - Een proces kan beheerd worden: starten, pauzeren, configureren, stoppen.
  - Heeft als doel om deadlocks en netwerkvertragingen te voorkomen en consistentie te bereiken.

- Deployment view:
  - Beschrijft op welke toestellen de processen moeten gedeployed worden, hoeveel toestellen er gebruikt worden.
  - Verschillende deploymentconfiguraties mogelijk per klant of geografisch gebied, maar ook of dat het een productie of ontwikkelomgeving is.
- +1 Use Cases/Scenarios:
  - Een use case beschrijft, binnen een view, hoe dat de componenten binnen die view met elkaar interageren voor een bepaalde situatie.
  - Is eigenlijk redundant omdat andere 4 views deze informatie ook al bevatten, maar use cases zijn toch nuttig:
    - ✓ Het valideert het ontwerp.
    - ✓ Het kan nieuwe systeemelementen ontdekken.

1.2 Reactive manifesto

4 kenmerken:

- Message Driven: asynchrone communicatie tussen componenten. Maakt gebruik van een wachtrij om de berichten te beheren. Dit heeft drie voordelen:
  - ✓ Zwakke koppeling: de verschillende componenten moeten enkel een protocol afspreken voor het bericht.
  - ✓ Loskoppelen van de tijd: Zender en ontvanger moeten niet wachten op elkaar.
  - ✓ Loskoppelen van locatie: De zender en ontvanger moeten niet in hetzelfde proces beschikbaar zijn, enkel de locator (analogie met gsm-nummer, ik kan eender waar naar iemand bellen, onafhankelijk van zijn locatie) moet bekend zijn.
- Responsief:
  - Lazy loading
  - Toon progressbar
  - Een trage service mag andere services niet beïnvloeden.
- Elastisch:
  - Predictieve en reactieve schaling
  - Resources moeten voor elk individueel component instelbaar zijn
  - Systeem moet responsief blijven
- Foutbestendig:
  - Systeem moet zichzelf kunnen herstellen
  - Fouten moeten snel opgespoord kunnen worden via monitoring
  - Voorzie fallback services

Deel I

Microservices

Hoofdstuk 2

Architectuurstijlen

- Δ Gelaagde stijl:
  - Δ Kan toegepast worden op elk view model.
  - Δ 3 lagen in het **logische view**: persistentie, presentatie en domeinlogica.
  - ✓ Robust systeem.
  - ✓ Eenvoudig om te ontwikkelen (veel frameworks ondersteunen dit: Java EE, .NET, ...).
  - ✓ De verschillende lagen kunnen gemockt worden om eenvoudig testen uit te voeren.
  - ! Slechts één presentatielaag, maar er kunnen verschillende clients zijn (desktop, mobile, tablet, ...).
  - ! Slechts één persistentielaag, maar er kunnen verschillende databasetechnologiën gebruikt worden.
  - ! De domeinlogicalaag definieert repositories, maar de persistentielaag implementeert deze. De dependency is dus omgekeerd.
- Δ Hexagonale stijl:
  - Δ Maakt gebruik van adapters. Deze adapters zijn interfaces en kunnen opgesplitst worden:
    - Δ **Inbound adapter**: Dit is API dat de domeinlogica openstelt, zodat deze kan aangeroepen worden door externen. Elke client kan nu onafhankelijk van elkaar ontwikkeld worden. Ze moeten enkel maar de API aanroepen.
    - Δ **Outbound adapter**: Dit is de API die de domeinlogica kan gebruiken (repository interface, payment interface).
  - ✓ presentatie-, persistentie- en domeinlogicalaag zijn losgekoppeld.
- Δ Monolithische stijl:
  - Δ Een monolitische applicatie bevat de drie lagen van de gelaagde stijl, en verpakt dit in één executable.
  - ✓ Makkelijk te ontwikkelen met een IDE.
  - ✓ Eenvoudig om wijzigingen door te voeren: edit → build → deploy.
  - ✓ Eenvoudig te testen.
  - ✓ Eenvoudig om te deployen.
  - ! Voor grote codebases wordt het moeilijk om elk detail van de codebase te kennen.
  - ! Een kleine wijziging resulteert in het rebuilden van de hele applicatie.

- ! Een bug kan het hele systeem onbereikbaar maken.
- ! Replicatie is haast onmogelijk.

- Δ **Microservices**:
  - Δ Decomposeerd een applicatie in kleine, zwak gekoppelde services, die individueel kunnen gedeployed worden.
  - Δ Services communiceren met elkaar via APIs.
  - Δ Wordt geïmplementeerd in de implementatie view. Elke service heeft zijn eigen logische view.
  - Δ Wat is een service?
    - Δ Een individueel component dat een bepaalde functionaliteit aanbiedt.
    - Δ Deze functionaliteit wordt via een API beschikbaar gesteld.
  - Δ Wat is een service?
    - Δ Een service dat goed gedefinieerd is in functie van de business.
    - Δ Voert een eenvoudige functie uit, en kan door een klein ontwikkelteam beheerd worden.
  - ! De juiste services vinden is moeilijk.
  - ! Foute decompositie leidt tot een gedistribueerd monolitisch systeem.
  - ! Gedistribueerde systemen zijn complex. (bv geen IDE die dit ondersteund).
  - ! Communicatie tussen services (over een netwerk) is traag.
  - ! Vele verschillende services kunnen op hetzelfde moment aan het draaien zijn.

Hoofdstuk 3

Decompositie van een applicatie

Het proces om een applicatie in te delen in verschillende microservices noemt men **decompositie**. Dit **iteratief** proces is belangrijk aangezien een foutieve methode leidt tot ongewenste resultaten. Een dergerlijk proces kan opgedeeld worden in drie stappen.

1. **Identificatie van de systeemoperaties.** Dit omvat het vertalen van de noden van één of meerdere gebruikers naar user stories en use-cases. Vaak wordt er hier overlegd met enkele domeinexperts. Het is belangrijk om te achterhalen wat belangrijke systeemoperaties zijn. Welke informatie moet er met een *create*, *update* of *delete* gewijzigd worden? Welke informatie moet met een *query* opgehaald worden? In deze fase worden er nog geen technische vaststellingen gedaan. De focus ligt namelijk op het vaststellen van de pre- en postcondities van de verschillende systeemoperaties.

2. **Identificatie van de services.** Services specificeren handelingen dat een bedrijf kan doen. Voorbeelden voor een online webshop zijn: *Sales*, *Marketing*, *Payment*, *Order Shipping* en *Order Tracking*. Deze services blijven lang stabiel en zullen haast nooit veranderen tenzij de business een shift van focus doen. Een obstakel dat zich kan voordoen zijn **godklassen**. Dit zijn klassen die te veel verantwoordelijkheid op zich dragen. Een oplossing hiervoor is om deze klasse in een centrale databank op te slaan en services die deze klasse nodig hebben kunnen die dan via de databank aanspreken. Dit is duidelijk een overtreding op de principes van de microservice architectuur. Er is nu een sterke koppeling tussen de microservices en de godklasse. Een betere oplossing is het opsplitsen van de klasse in verschillende klassen op basis van de bestaande services. Deze verschillende klassen kunnen in een microservice gestoken worden waarbij de definitie van de klasse sterk gedaald is (ze moet maar gelden binnen de microservice). Voorbeeld van een godklasse is een **Order** klasse voor pizza's. Denk aan de typische attributen: *status*, *requestedDeliveryTime*, *prepareByTime*, *deliveryTime*, *paymentinfo*, *deliveryAddress*, ... Het opsplitsen van deze klassen kan bijvoorbeeld gebeuren door enkel informatie die relevant is voor de keuken, in een keukenserivce te steken en informatie die enkel relevant is voor het bezorgen van een bepaalde order in een deliveryservice. Op die manier worden godklassen vermeden.

3. **Identificatie van de service API's.** Deze laatste stap zal nagaan welke operaties van een microservice publiek moeten gesteld worden aan de buitenwereld via een API.

Hoofdstuk 4

Interactiestijlen tussen services

	one-to-one	one-to-many
synchroon	request/response	/
asynchroon	request/async response one way notifications	publish/subscribe publish/async response

- Δ request/response: Een service stuurt een request en wacht op een response.
- Δ request/async response: Een service stuurt een request, maar wacht niet noodzakelijk op een response.
- Δ one way notifications: Een service stuurt een request en verwacht geen response.
- Δ publish/subscribe: Een service publiceerd een bericht en kan opgevangen worden door 0 of meerdere geïnteresseerden.
- Δ publish/async responses: Een service publiceerd een bericht en zal eventueel antwoorden opvangen van 0 of meerdere geïnteresseerden.

4.1 Synchrone communicatie

4.1.1 Voorbeelden

- Δ REST
  - Δ Vier levels:
    - 0 : Enkel HTTP POST mogelijk. Elke actie krijgt dan ook een ander endpoint toegewezen.
    - 1 : Maakt gebruik van resources zodat elk individuele resource een URI krijgt. Nog steeds enkel HTTP POST mogelijk.
    - 2 : GET, POST, PUT mogelijk. Een zelfde resource kan nu meerdere operaties ondersteunen.
    - 3 : HATEEOAS mogelijk: een GET request bevat, behalve het object, ook URLs die mogelijke acties op het object toelaten.
  - ✓ HTTP wordt niet geblokkeerd door een firewall.
  - ✓ Implementeerd request/response interactie.

- ! clients moeten de locatie (URL) kennen.
- ! Meerdere resources in één request opvragen is moeilijk.
- ! Soms is het moeilijk om de HTTP werkwoorden te mappen op een operatie.

- Δ gRPC
  - Δ Google Remote Procedure Call.
  - Δ API wordt gedefinieerd op basis van een Interface Definition File. Dit bestand bevat IDL (Interface Description Language) code.
  - Δ Deze code wordt gecompileerd afhankelijk van de gekozen client.
  - ! Het valt niet op dat de communicatie nu over het netwerk gebeurd.

4.1.2 Foutbestendigheid

Communicatie tussen twee services moet foutbestendig zijn. Problemen zoals trage netwerken, overbelaste microservices, enz... moeten oplosbaar zijn. Er zijn drie patronen die geschikt zijn om technische problemen te voorkomen en op te lossen.

- Δ **Netwerk-timeouts:** Zet een limiet op het aantal seconden dat een microservice wacht op een antwoord van een andere microservice.
- Δ **Bulkheads:** Zet een limiet op het aantal requests dat een client kan versturen naar een service. Dit kan bijvoorbeeld geïmplementeerd worden door bij elke service die andere services aanspreekt, een eigen thread pool bij te houden, zodat enkel de thread pool van één service vol komt te zijn.
- Δ **Circuit breaker pattern:** Monitor het aantal succesvolle en gefaalde operaties van een service. Wanneer de verhouding van gefaalde en succesvolle operaties een bepaalde limiet overtreedt, dan wordt de circuit breaker (die de operaties monitored), geactiveerd en zal geen enkele request nog lukken.

Wanneer deze problemen zich voordoen, is het nuttig om een aantal fallback strategiën te hebben. Meestal is dit een fout of een gecached antwoord terugsturen.

4.2 Asynchrone communicatie

- Δ Maakt gebruik van messaging.

4.2.1 Foutbestendigheid

Bij asynchrone communicatie kunnen er zich twee problemen voordoen:

- Δ **Competing consumers:** Het kan voorkomen dat er meerdere instanties van dezelfde service geïnteresseerd is in een bepaalde message stream. Er moet garantie zijn dan de berichten die toekomen in deze stream sequentieel afgewerkt worden. Het kan ook zijn dat verschillende services geïnteresseerd zijn in dezelfde message stream. Men zou de berichten dan moeten dupliceren, en dat gaat niet met een standard queue.

Deze problemen worden opgelost door een message broker.

Hoofdstuk 5

Saga

= garanderen dat een transactie ofwel volledig, ofwel niet uitgevoerd wordt.

- Traditioneel: 2-fasen-commit.
  - Eerst wordt elke databank naarwaar geschreven moet worden, op de hoogte gebracht van de informatie die ze moeten schrijven.
  - De eerste fase zal aan elke databank vragen of zij klaar zijn om weg te schrijven (prepare-fase).
  - De tweede fase zal effectief een commit uitvoeren op elke databank (commit-fase).
  - Van zodra één databank 'nee' antwoord, wordt het hele proces gestopt.
  - ! Synchroon
  - ! Niet elke database implementeert het concept van transacties en locking.
- **Saga**
  - Een sequentie van lokale interacties tussen verschillende microservices.
  - Elke systeemoperatie moet een saga hebben.
  -

Deel II

Container deployment and orchestration

Hoofdstuk 6

Productieomgeving

Een applicatie kan over een groot aantal services beschikken, die allemaal gebruik maken van verschillende technologiën. Een service kan eigenlijk beschouwd worden als een kleine applicatie, zodat er in plaats van één grote applicatie, meerdere kleinere applicaties in productie moeten draaien. Zo een productieomgeving moet vier functionaliteiten implementeren:

1. **Service management interface.** Het in staat zijn om services te creëren, configureren en updaten, vaak via een shell of GUI.
2. **Runtime service management.** De omgeving moet automatisch services kunnen herstarten indien deze gecrasht zijn. Ook als een fysieke server faalt, moet de omgeving een andere server aanspreken om de service op te draaien.
3. **Monitoring.** Informatie over elke service instance zoals logbestanden en metrieken voor die bepaalde service (aantal bezoekers per seconde, sucess rate, ...) moeten beschikbaar zijn voor de ontwikkelaars, en moeten ook gewaarschuwd worden indien een service niet aan de vooropgestelde criteria voldoet.
4. **Request routing.** De requests dat users versturen moeten naar de juiste service doorverwezen worden.

Volgende paragrafen bespreken hoe een aantal van deze zaken geïmplementeerd kunnen worden.

Een service heeft altijd een aantal configuratiegegevens (**environment variabelen** genoemd), die afhankelijk zijn van de omgeving waarin de service draait. Een service moet zo ontworpen zijn dat deze slechts éénmaal gecompileerd moet worden door de deployment pipeline, zodat deze meerdere malen in productie gezet kan worden. Het externalizeren van de configuratiegegevens betekent dat de configuratie van een service tijdens runtime bepaalt wordt. Hier zijn er twee modellen mogelijk:

- **Push model.** Bij dit model zal de service bij het opstarten configuratiegegevens verwachten, die door de deploymentomgeving meegegeven worden. Hoe deze configuratiegegevens gegeven worden (via bestand, of individuele parameters) maakt niet uit. De service en deploymentomgeving moeten wel onderling van elkaar weten hoe de structuur van de configuratiegegevens in elkaar zit. Het grootste nadeel van deze methode is dat een service haast niet meer gewijzigd kan worden na het initialiseren van de service. Een ander nadeel is dat de configuratiegegevens verspreidt over de services liggen.
- **Pull model.** Het pull model heeft bijna enkel voordelen tegenover het push model. De deploymentomgeving geeft bij de creatie van de service enkel de URL mee van een zogenaamde configuratieserver. Deze server bevat alle configuratiegegevens voor elke service. De service

zelf zal dan deze server, met behulp van de URL, aanspreken om de juiste configuratiegegevens op te halen. Dit biedt een aantal voordelen:

- Gecentraliseerde configuratie.
- Een service kan de server pollen om na te gaan of de configuratiegegevens aangepast zijn, en deze dan eventueel op te halen. De service moet hiervoor niet herstart worden.
- Sommige configuratiegegevens zijn gevoelig, zoals databaseinformatie. De server zal deze moeten encrypteren. De service wordt dan wel verwacht de publieke sleutel van de server te hebben zodat hij deze kan decrypteren. Sommige servers decrypteren de configuratiegegevens zelf.

Het grootste nadeel is echter dat de configuratieserver opnieuw een infrastructuur is dat moet opgesteld en onderhouden worden.

Om **monitoring** te implementeren moeten services 'waarneembaar' gemaakt worden. Er moeten hiervoor extra APIs aangemaakt worden, die niet mogen interfereren met de werkelijke functionaliteit van de service. Hiervoor zijn er een aantal hulpmiddelen om een service waarneembaar te maken:

- **Health check API.** Een eenvoudige API dat de status van de service teruggeeft.
- **Log aggregation.** Logbestanden zijn een goede manier om de werking van de service op te volgen. Deze logbestanden worden best geschreven naar een gecentraliseerde logserver, zodat zoeken ondersteund kan worden. Het is ook enkel de verantwoordelijkheid van de logserver om ontwikkelaars te waarschuwen. Traditioneel logt een applicatie naar een welbepaald logbestand op het filesysteem. Dit is hier geen goede oplossing, omdat sommige services zelfs geen filesysteem zullen hebben. Hierom moet elke service loggen naar stdout. De deploymentomgeving zal dan beslissen wat hij wil doen met deze uitvoer.
- **Distributed tracing.** Het oproepen van een endpoint van de API kan meerdere interne calls maken. Het is vaak moeilijk te achterhalen waarom zo een query traag verloopt. Distributed tracing kent aan elke endpoint een ID toe, en bekijkt de call chain die overlopen wordt. Deze gegevens worden naar een gecentraliseerde server verstuurd waarop analyse kan uitgevoerdt worden. Een endpoint wordt gepresenteerd door een trace. Zo een trace bestaat uit geneste spans, die elk een call voorstellen. De endpoint is de top span, en zal elke andere span bevatten.
- **Audit logging.** Het doel van audit logging is om acties van gebruikers te verzamelen. Elke audit log entry heeft een ID dat een gebruiker voorstelt, de actie dat ze uitgevoerd hebben en het domeinobject. Voorbeelden zijn gefaalde loginpoging, toevoegen van items in het winkelmandje, maar uiteindelijk niet betalen. Zulke logs dienen vooral voor customer support en om vreemde activiteiten op te sporen.
- **Application metrics.** Deze tak bestaat uit een metriekservice. Deze metriekservice vraagt gegevens op van de verschillende applicaties. Zulke gegevens zijn onder andere: CPU gebruik, geheugengebruik, schijfgebruik, aantal requests per seconde, request latency en domeinspecifieke gegevens. Een service moet ontworpen zijn zodat al deze gegevens naar de metriekservice kunnen verstuurd worden, en is afhankelijk van het gebruikte framework. De service kan ofwel zelf beslissen om zijn metrieken te versturen naar de metriekservice, of de metriekservice zal elke service pollen om de metrieken op te halen.

Hoofdstuk 7

Containers

Verschillende microservices kunnen onderling elk gebruik maken van verschillende technologiën. Om deze verschillende microservices te deployen zouden al de verschillende dependencies van elke technologie op de server geconfigureerd moeten worden. Dit is natuurlijk niet haalbaar op grote schaal, daarom zou men in eerste instantie virtuele machines kunnen gebruiken waarbij elke verschillende virtuele machine geshickt is voor een specifieke technologiestack. Virtuele machines nemen echter te veel opslag in beslag. Ze nemen zoveel beslag in omdat elke virtuele machine zijn eigen besturingssysteem en kernel bevat. De hypervisor (of virtual machine monitor) geeft elke virtuele machine de illusie dat enkel hun machine toegang heeft tot de resources van het hosttoestel (het toestel waarop de virtuele machines draaien). Er zijn twee types hypervisor:

- **Type 1.** Dit type hypervisor draait rechtstreeks op de hardware van de host en heeft geen behoefte aan een onderliggend besturingssysteem.
- **Type 2.** Dit type hypervisor heeft wel nood aan een besturingssysteem. Dit heeft als voordeel dat er ook applicaties op het hosttoestel kunnen draaien.

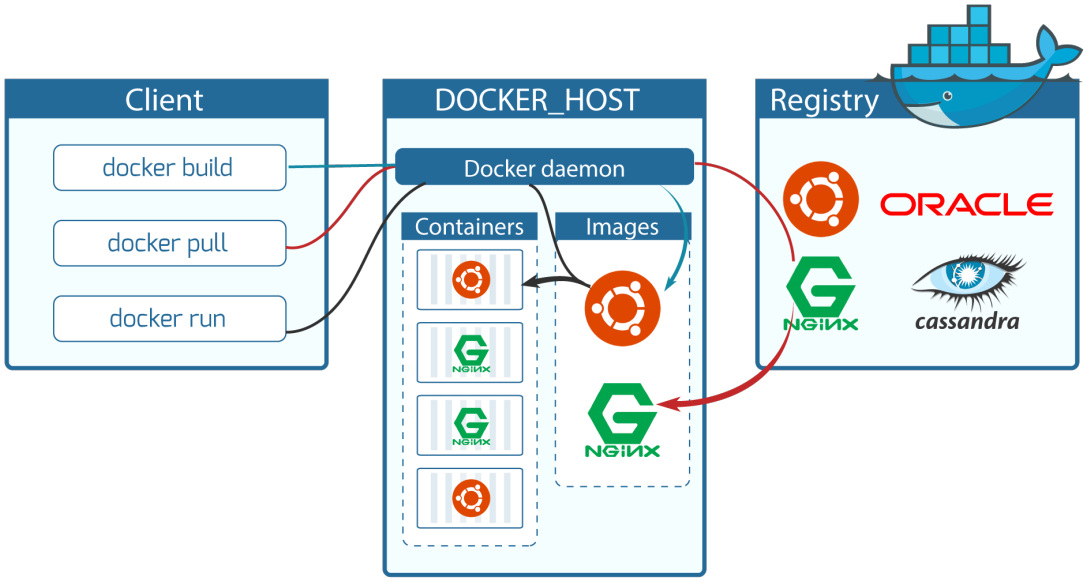
Containers zijn een virtualisatietechniek op besturingssysteemniveau. Een populaire container technologie is Docker, dat gebruik maakt van de linux container functionaliteit.

- **Linux Control Groups (cgroups).** Groeperen van processen, en privileges zetten op deze groepen.
- **Linux namespaces.** Een geïsoleerde weergave van de systeemresources.
- **Changing root (chroot).** Elk process de illusie geven dat ze vanuit de root-directory aangesproken worden.
- **Veilige containers (LSM).** Mogelijkheden per process vastzetten.

Docker:

- Automatisch verpakken en deployen van een applicatie.
- Beschikbaar op Linux, MacOS, Windows.
- Onafhankelijk. Minimale dependencies.
- Grote flexibiliteit in wat er in de container moet.
- Alle containers kunnen gestart en gestopt worden op dezelfde manier.
- Lightweight. Meerdere containers per host mogelijk (nog meer dan virtuele machines).

7.1 Docker Architecture



De **Docker daemon** is verantwoordelijk om containers te starten, stoppen, monitoren, alsook om images op te slaan en te builden. De **Docker client** kan de daemon aanspreken om de operaties van de daemon uit te voeren. De **Registry** bevat images die door de daemon gebruikt kunnen worden.

Een image aanmaken kan als volgt:

```
# Dockerfile to build an Apache2 image
# Base image is Ubuntu
FROM ubuntu:14.04
# Install apache2 package
RUN apt-get update && apt-get install -y apache2 && apt-get clean
```

Syntax van een Dockerfile:

```
1  FROM ubuntu:14.04
2
3  COPY html /var/www/html
4  ADD web-page-config.tar /
5
6  ENV APACHE_LOG_DIR /var/log/apache
7  USER 73
8
9  EXPOSE 7373/udp 8080
10
11 RUN apt-get update && apt-get install -y \ apache 2 && apt-get clean
12
13 ENTRYPOINT ["echo", "Dockerfile entrypoint Demo"]
```

△ **Lijn 1 FROM** De base image van de container. Alle volgende commandos bouwen verder op deze image.

- **Lijn 3 COPY** Bestanden kopieren van de docker host naar het bestandssysteem van de nieuwe image.
- **Lijn 4 ADD** Gelijkaardig aan COPY, maar kan ook .tar bestanden (die hij zal unzippen) en URLs (die hij zal downloaden) behandelen.
- **Lijn 6 ENV** Een omgevingsvariabele instellen in de nieuwe image.
- **Lijn 7 USER** Een gebruiker toevoegen met een specifiek UID. Deze UID wordt ook gebruikt in volgende RUN, CMD of ENTRYPOINT instructies.
- **Lijn 9 EXPOSE** Deze instructie informeert enkel dat deze poorten zullen gebruikt worden. Docker zal zelf deze poorten niet openzetten.
- **Lijn 11 RUN** Deze instructie bevat commando's die uitgevoerd moeten worden tijdens de buildfase. Het is beter om slechts één RUN instructie te hebben, omdat docker een nieuwe, read-only, laag aanmaakt voor elke instructie.
- **Lijn 13 ENTRYPOINT** Het startpunt van de applicatie. Als deze applicatie stopt, wordt de container ook automatisch gestopt.

Wanneer een container gestart wordt vanuit een image, dan zal Docker een extra, schrijfbare, laag toevoegen. Op deze laag kunnen dan nieuwe bestanden gezet worden. Als de container verwijderd wordt, zullen ook deze bestanden verwijderd worden. Op die manier kunnen dus verschillende containers, die toch dezelfde image hebben, aparte informatie bewaren.

Een container builden:

```
docker build -t dockerfile .
```

Hoofdstuk 8

Container orchestration

Drie functies van een orchestration framework:

1. **Resource management**: Een groep van machines als één cluster beschouwen, zodat het lijkt alsof deze groep één proces is met zijn eigen CPU-, RAM- en volumegebruik.
2. **Scheduling**: Het bepalen welke groep van containers op welke machine moet komen. De default werkwijze is om te kijken naar de systeemresources die een groep nodig heeft, en een daarbijhorende machine te vinden die deze systeemresources kan aanbieden.
3. **Service management**: Het blootstelling van de container aan de externe wereld. Ook zal het framework ervoor zorgen dat er genoeg instanties van een bepaalde service zijn. Verder zal het framework load balancing implementeren tussen deze instanties.

8.1 Kubernetes

Kubernetes is een voorbeeld van een orchestration framework.

- Een machine in een cluster kan twee vormen aannemen:
  - **Master:** deze machine beheert de cluster en bevat volgende componenten:
    - Het bevat een API server, die aangesproken kan worden door ontwikkelaars om de nodes aan te spreken.
    - De scheduler bepaalt op welke node een pod (≡ container, maar specifiek voor kubernetes) moet komen.
    - De controller manager is het proces dat verschillende kubernetes controllers bevat. Voorbeelden van controllers zijn:
      - Replication Controller, die ervoor zorgt dat er een bepaald aantal kopieën van een pod aanwezig zijn in het cluster.
      - Node Controller, die notificaties geeft wanneer een node niet meer bereikbaar is.
      - Endpoints Controller, die pods en services samenvoegt.
    - Tot slot is er nog de etcd storage, die configuratie en de staat van de cluster bewaart.

    Meestal heeft een cluster maar een klein aantal masters.
  - **Node:** Dit zijn de effectieve machines die de applicatie doen draaien en bevat volgende componenten:

    - De kubelet beheert de pods die aanwezig zijn op de node.
    - De kube-proxy voorziet een abstracte manier om de node te beheren.
    - De cAdvisor is een daemon that metrieken verzamelt.
    - De pods zijn de applicatieservices.

- Een pod is een eenheid van deployment in kubernetes, en bestaat uit één of meerdere containers die eenzelfde IP-adress en opslagvolume delen. Kubernetes maakt zelf de pods aan, met behulp van een document dat de gewenste staat beschrijft:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
 name: myapp-deployment
spec:
 replicas: 3
 template:
  metadata:
   labels:
        app: MyApp
  spec:
   containers:
   - name: nginx
         image: nginx:1.7.9
         ports:
         - containerPort:80
```

- Vaak wordt er per pod maar één container geassocieerd, maar het is ook mogelijk om meerdere containers per pod te associeren. Dit is handig als je extra hulpfunctionaliteit wil bieden aan een service zoals:

    - **Sidecar container**: een service die periodiek een git pull doet om de applicatie up te daten. Een andere mogelijkheid is een service die logbestanden verwerkt.
    - **Adapters**

- De service registry is een databank van services, die voor elke service al zijn instanties en hun locaties bijhoudt. Een instantie wordt registreerd in de databank bij creatie, en wordt er terug uitgehaald bij destructie. clients moeten enkel nog aan de service registery een query doen, om de instanties op te halen voor een specifieke service.

    - **Client-side discovery**: Een client is gekoppeld aan de service registry en zal zelf load-balancing toepassen. Het nadeel is dat de client nu sterk gekoppeld is aan de service registry en dat er nu discovery logic in elke client aanwezig moet zijn.
    - **Server-side discovery**: De client stuurt nu een request naar een load-balancer service, die op een bekende en vaste locatie ligt. De load-balancer zal nu zelf de service registry aanspreken en load-balancing implementeren. De client moet nu enkel een request sturen naar de load-balancer. Het nadeel is hier nu dat deze load-balancer opnieuw een component is dat moet onderhouden worden. Ook moet het gerepliceerd worden om beschikbaarheid te garanderen.

- Kubernetes maakt gebruik van server-side discovery en vereist het aanmaken van een Kubernetes Service:

```
kind: Service
apiVersion: v1
metadata:
```

```
  name: myapp-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 8080
    target-port: 9000
```

Deze service zal clients een juiste pod bezorgen.

- Een load balancer zal voor een bepaalde request bepalen welke instantie teruggeven moet worden. Enerzijds gebeurd dit op basis van de request, zoals bijvoorbeeld de geografische afstand, zodat enkel instanties in de omgeving relevant zijn. Anderzijds moet uit lijst van instanties degene gekozen worden op basis van de huidige capaciteit van een instantie.
- Er zijn drie interessante operaties die uitgevoerd kunnen worden met een load balancer.
  1. **Rolling update:** In plaats van elke instantie op hetzelfde moment up te daten, kan er altijd een selectie van instanties offline gehaald worden. De load-balancer stuurt dan de requests door naar de overige instanties.
  2. **Canary release:** Een kleine selectie van requests (bv 5%) worden naar een instantie gestuurd die een nieuwe software update bevat. Op die manier worden weinig mensen getroffen door bugs, indien deze aanwezig zouden zijn.
  3. **A/B testing:** Dit dient eerder voor marketingdoeleinden. De helft van de gebruikers krijgen versie A, en de andere helft krijgen versie B van een applicatie.

8.2 Dimensies van cloud computing

Drie dimensies in cloud computing

- Essentiële eigenschappen.
- Cloud service models.
- Cloud deployment models.

8.2.1 Essentiële eigenschappen

vijf eigenschappen

- **On-demand:** een resource moet op elk moment beschikbaar zijn.
- **Elasticiteit:** het aantal beschikbare resources moet instelbaar zijn, afhankelijk van de huidige nood.
- **Resource pooling:** De resources van de cloud provider worden gebruikt door meerder gebruikers. Een individuele gebruiker kan slechts op abstract niveau resources specificeren zoals het aanvragen van 6GB ram.
- **Metered:** Het gebruik van resources wordt in het oog gehouden. Meestal moet enkel voor de effectief gebruikte resources betaald worden.
- **Network access:** Een API is beschikbaar om de resources aan te spreken.

8.2.2 Cloud service models

eerst stakeholders beschrijven:

- **Cloud provider:** De provider beheert de hardware en software en stelt deze ter beschikking voor cloud users. De cloud provider is verantwoordelijk om de afgesproken requirements met de verschillende cloud users na te leven, meestal in de vorm van minimale garanties die de cloud provider moet leveren. Voorbeelden van cloud providers zijn: Amazon, Google en Microsoft.
- **Cloud users:** De cloud users gebruikt de resources van de cloud provider om applicaties te hosten, die dan gebruikt kunnen worden door end users. De cloud user heeft geen zicht op de interne structuur van de cloud provider, en kan enkel resources aanvragen. Er is geen formeel contract tussen cloud users en end users. Een voorbeeld van een cloud user is Netflix.
- **End users:** De end users maken gebruik van de cloud applicaties van de cloud users. Meestal is hier een bepaalde kost aan verbonden (jaarlijkse subscriptie).

vijf service models

- **Metal as a Service:** Huren van fysieke servers, kabels en electriciteit. De provider helpt met het instaleren van onder andere een besturingssysteem, configuratie van VLANs en andere toestellen op het netwerk ontdekken.
- **Infrastructure as a Service:** Huren van virtuele machines, opslag en netwerkinfrastructuur.
- **Container as a Service:** Huren van pregeconfigureerde orchestratieplatformen zoals Kubernetes.
- **Platform as a Service:** Huren van middleware producten zoals database management systemen, applicatieservers, message brokers, enz.
- **Software as a Service:** Huren van software.

8.2.3 Cloud deployment models

drie deployment models

- **Public cloud:** Iedereen kan de systeemresources huren en gebruiken.
- **Private cloud:**
- **Hybrid cloud:**

8.3 Elastische schaling

twee vormen van schaling

- **Statische schaling:** De systeembeheerder voert zelf een commando in om meer of minder resources te voorzien, meestal als het te laat is.
- **Elastische schaling:** Het systeem voorziet zelf schaling, in kleine intervallen, op basis van de huidige workload.

drie typen workloads

- Δ **Statische workload:** In dit geval is de werkload nagenoeg altijd dezelfde, zodat elastische schaling niet zo nuttig lijkt. Toch is het handig om, indien een machine faalt, snel een nieuwe machine werkend te hebben. Elastische schaling vergemakkelijkt dit proces.
- Δ **Periodieke workload:** Bij voorgedefinieerde pieken, zoals applicaties die enkel beschikbaar moeten zijn per dag, is het wel handig om elastische schaling te hebben. Bij statische schaling kan je vooraf genoeg resources voorzien, maar op normale momenten zijn er dan teveel resources beschikbaar, zodat er meer moet betaald worden.
- Δ **Dynamische niet-periodieke workload:** Deze niet-periodieke workloads kunnen vooraf geweten zijn (ticketverkoop) of niet (opeens meer interesse in bepaalde website).

Om elastische schaling toe te passen kan men steunen op twee types:

- ! **Verticale schaling:** De bestaande infrastructuur uitbreiden (meer geheugen, CPU, harde schijven in nodes steken). Dit is geen goede techniek, omdat het neerschalen niet eenvoudig toelaat. Alle nieuwe hardware zou er dan uit moeten gehaald worden.
- ✓ **Horizontale schaling:** Nieuwe nodes voorzien die dezelfde infrastructuur hebben. Op deze manier kan zelfs de capaciteit van de grootste node overschreden worden.

Deel III

Distributed Data Storage & Processing

Hoofdstuk 9

De uitdagingen van moderne data

Data-intensieve applicaties moeten rekening houden met volgende vier categorieën:

- **Volume.** De hoeveelheid opslag die over verschillende plaatsen moeten opgeslagen worden.
- **Velocity.** De snelheid waarop nieuwe informatie actueel wordt.
- **Variety.** De verschillende soorten types van data die bestaan.
- **Veracity.** De betrouwbaarheid van de data.

Hoofdstuk 10

Datamodellen

10.1 Het relationeel model

zie cursus relationele gegevensbanken, belangrijk is om gewoon de nadelen te kennen zoals:

- Er zou een brede tabel nodig zijn met honderden kolommen (waarvan de meeste dan NULL zijn) om bijvoorbeeld de producten van een winkel op te slaan. Elk product heeft diverse kenmerken die eigen zijn aan een bepaalde productcategorie.
- Men zou dit kunnen oplossen door een nieuwe tabel te maken per productcategorie, maar dit introduceert veel tabellen en relaties (te vergelijken met het verhogen van de normaalvorm).

10.2 Het document model

Informatie in een document model wordt in een boomstructuur met one-to-many relaties opgeslagen, en is daarom dus perfect voor one-to-many relaties. Een document wordt opgeslagen als één string, meestal in JSON of XML formaat, op deze manier volstaat één enkele query om een hele object, en zijn relaties op te vragen.

Een extreem voordeel van het document model is dat het geen restricties oplegt aan de data. Er kunnen twee producten zijn die in het systeem herkent worden als "Product"maar een andere interne structuur hebben. Het document model wordt dus best gekenmerkt door:

- Flexibiliteit in het schemamodel. Dit wordt ook wel "schema-on-read"genoemd aangezien de client niet op voorhand kan weten welke structuur het document zal hebben.
- Data lokaliteit. Hiermee wordt bedoeld dat een document als één enkelvoudige string wordt opgeslagen, en alle informatie zit dan ook in die string. Er is geen nood aan het join-of indexeringmechanisme. Een document wordt altijd in zijn geheel ingelezen, dit kan een nadeel zijn indien slechts een beperkt aantal informatie van dat document nodig is.

10.3 Het graaf model

Het document model volstaat voor one-to-many relaties, maar is niet perfect voor many-to-many of many-to-one relaties want dan moeten er toch "joins"gedaan worden, maar dan op documenten. Het graaf model kent twee soorten:

1. Een normale graaf met knopen en verbindingen, die beiden attributen kunnen hebben.
2. Een drievoudig model waarbij alle informatie opgeslagen wordt als: `SUBJECT` → `PREDICATE` → `OBJECT`

Gekende graafalgoritmen kunnen toegepast worden op dit model. Het graaf model heeft een aantal use cases:

- **Transportnetwerk.** Een graaf is de geschikte manier om een wegennet voor te stellen.
- **Linkanalyse.** Het zoeken van objecten die gerelateerd zijn aan een ander object (bv vrienden van vrienden zoeken).

10.4 Het kolomfamilie model

ToDo: xxx

Hoofdstuk 11

Gedistribueerde informatie

Waarom is het belangrijk dat informatie op verschillende nodes beschikbaar is?

- **Schaalbaarheid:** Een toestel heeft maar een maximum aantal geheugen, opslagplaats en schijfoperaties per seconde. Meerdere nodes betekent dat de belasting kan verdeeld worden tussen de nodes.
- **Fouttolerantie:** Een reverse voorzien voor in het geval dat een andere node uitvalt.
- **Latency:** Nodes geografisch verspreiden zodat connecties vanuit andere continenten niet traag zijn.

Er wordt best gebruik gemaakt van het horizontaal schaalschema. Dit heeft als voordeel dat er geen speciale hardware vereist is, en er gewoon machines kunnen bijgekocht worden indien dit nodig zou zijn.

Er zijn twee belangrijke patronen om data de distribueren:

1. **Replicatie:** Dit is eenvoudig alle data dupliceren op elke verschillende node, zodat ze allemaal dezelfde data bevatten. De voordelen zijn: hoge databeschikbaarheid en fouttolerantietegen het uitvallen van een node door de redundantie van de informatie. Het nadeel is: hoe moeten we ervoor zorgen dat alle nodes over dezelfde data beschikken (zie sectie 11.1)?
2. **Partitionering (sharding):** De grote hoeveelheid data kan ook gepartitioneerd worden, zodat elke node zijn unieke verzameling van gegevens bevat. Partitionering heeft een aantal voordelen: Elke partitie moet slechts zijn beperkte data behandelen. Hoe groter de dataset wordt, hoe minder operaties een bepaalde partitie zal moeten uitvoeren, aangezien er meer patitities zullen ingevoerd worden zodat de data meer verspreidt ligt over alle partities. Het nadeel is: hoe kunnen we bepalen op welke node een bepaald stukje informatie moet komen. Idealiter heeft elke node dezelfde workload.

In praktijk worden replicatie en partitionering gecombineerd. Eerst wordt partitionering toegepast, waarna deze partities ook nog gerepliceerd worden.

11.1 Replicatie

Er zijn twee modellen om aan replicatie te doen: het leader-follow model en het leaderless model. Een node die repliceerd wordt een replica genoemd.

11.1.1 Leader-Follower model

Dit model verloopt in drie stappen:

1. Een replica wordt tot leader gemaakt. Enkel op deze replica mogen er writeoperaties plaatsvinden.
2. Elke andere replica is een follower. Elke keer dat de leader naar zijn schijf schrijft, zal de leader ook de gewijzigde data doorsturen, in de vorm van een replication log, naar alle followers. Deze replication log bevat instructies dat elke follower moet ondernemen zodat ze de update juist kunnen uitvoeren.
3. Een client kan een readoperatie zowel aan de leader als aan een follower aanvragen.

Deze manier garandueerd dat de followers ooit zullen convergeren naar de juiste toestand.

11.1.2 Leaderless model