FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Systeemontwerp

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 11 november 2018

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

Microservices

Hoofdstuk 1

Architectuur

1.1 Gelaagde stijl

Een bekende architectuur is het drie-lagenmodel waarbij drie belangrijke lagen aanwezig zijn: de **persistentielaag**, die de logica zal bevatten waarmee met een databank kan gecommuniceerd worden om gegevens persistent te maken, de **businesslaag** die de regels gedefinieerd door de use-cases bevat en de **presentatielaag.** behandelt de logica om met client applicaties te communiceren. Het drielagenmodel kent een aantal **voordelen**.

- Het reflecteert de structuur van vele organisaties.
- Het is eenvoudig om mockups te gebruiken tijdens het testen van applicaties. Zo kan de hele databanklaag een mockup.
- Wordt ondersteund door vele frameworks zoals Microsoft .NET en Java EE.

Er bestaan echter ook **nadelen**:

- Er is slechts één enkele presentatielaag, maar er kunnen meerdere types zijn zoals een webapplicatie of een mobiele applicatie.
- Er is ook maar één enkele persistentielaag. Bepaalde informatie geniet een voorkeur bij een ander soort databank (relationeel, grafen, document based, ...).
- Vaak wordt de dependency tussen de business en persistentielaag omgewisseld. De businesslaag zal repositoryinterfaces declareren die geïmplementeerd worden door de persistentielaag.

1.2 Hexagonale stijl

Bij deze stijl bestaat er nog altijd de algemene businesslaag. Deze laag zal nu verschillende **interfaces** bevatten zoals een repositoryinterface of een betalinginterface. Deze interfaces kunnen door

externe applicaties aangesproken worden. De implementatie van zo een interface is afhankelijk van de technologie dat gebruikt wordt. Deze technologiën worden via een **adapter** omgezet zodat ze gebruikt kunnen worden in de businesslaag.

Het voordeel is dat componenten nu zwak gekoppeld zijn aan elkaar en daardoor gemakkelijker te testen zijn. Deze structuur wordt doorgaans meer gebruikt bij moderne applicaties.

1.3 Monolitische stijl

Deze stijl zet een project om tot een enkelvoudig uitvoerbare unit. Een Java EE project kan verpakt worden in een *WAR* of *JAR* bestand. De **voordelen** zijn:

- Eenvoudig om te ontwikkelen. *IDE*'s zijn gemaakt om enkelvoudige applicaties te ontwikkelen.
- Eenvoudig om veranderingen toe te passen.
- Eenvoudig te testen aangezien de applicatie enkel moet draaien en dan de bijhorende testen moet uitvoeren.
- Eenvoudig om te deployen. Een WAR bestand functioneert onmiddelijk indien deze geplaatst wordt op een server met Tomcat geïnstalleerd.

Er zijn ook echter een aantal **nadelen** aan verbonden. Een codebase die zodanig groot wordt dat een ontwikkelaar niet alles meer kan begrijpen geeft aanleiding tot tragere compileertijden, destructieve bugs en tragere testen aangezien deze heel de testsuite moeten uitvoeren.

1.4 Microservices

Een microservice is een stijl dat een applicatie opdeelt in kleinere services. Deze services kunnen onafhankelijk van elkaar gebouwd worden. De services communiceren enkel via een API. Elke service heeft ook zijn eigen databank. De API bestaat uit een aantal **commands**. Deze commando's wijzigen de databank van een service. **Queries** daarentegen, vragen enkel gegevens op. Deze twee types operaties worden aangesproken door een client die gebruik wenst te maken van de API. Verder bevat een service ook nog **events**. Clients worden geinformeerd bij het gebeuren van een specifiek event. Een voorbeeld van zo een event is een confirmatie dat een bestelling gelukt is. Een microservice zal een kleine taak op zich nemen. Meestal is dit een groep van use-cases die nauw aan elkaar verbonden zijn. Het voordeel is dat er slechts kleine groepen van ontwikkelaars nodig zijn om elke individueel service to bouwen. De **Voordelen** van microservices zijn:

- Zwakke koppeling tussen microservices. Een verandering in een service mag geen invloed hebben op andere services. Hierdoor zijn API's heel belangrijk.
- Er kunnen verschillende technologiën gebruikt worden in elke individuele miicroservice.
- Elke microservice kan zijn eigen database schema gebruiken.

Enkele **nadelen** zijn:

- Een **incorrecte decompositie** (zie volgend hoofdstuk) zal leiden tot een gidistribueerde monolith.
- Microservices **communiceren** met elkaar over een netwerk. Dit is over het algemeen **trager** dan een lokale connectie. Juist hierom moet de API zo ontworpen zijn dat er minimaal verkeer moet zijn tussen de verschillende microservices.
- Een *IDE* bevat minder tools om microservices te testen en te ontwikkelen.

Hoofdstuk 2

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 3

Interactiestijlen tussen services

Microservices moeten met elkaar kunnen interageren. Bij het zoeken van een oplossing moeten volgende vragen gesteld worden:

- Hoe kan het aantal interacties tussen twee microservices geminimaliseerd worden?
- Hoe wordt de communicatie geïmplementeerd?
- Kan men zeker zijn dat een microservice zal antwoorden?

Een interactie kan op een **synchrone** manier gebeuren. Een client zal een aanvraag sturen naar een bepaalde service en zal wachten totdat er een antwoord terug ontvangen is. Tijdens de aanvraag en het antwoord zal de client blokkeren. Een **asynchrone** interactie daarentegen zal toelaten dat de client niet hoeft te wachten op het antwoord. Verder is er ook nog een opdeling mogelijk op vlak van het aantal microservices die een request behandelen. Een **one-to-one** geeft aan dat slechts één enkele microservice het request zal afhandelen. Een **one-to-many** interactie geeft aan dat meerdere microservices een request zullen afhandelen.

Deel II

Container deployment and orchestration

Hoofdstuk 4

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 5

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.

Daarom schakelt men over naar containers.

Deel III

Distributed Data Storage & Processing