FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Systeemontwerp

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

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 aangesproken worden 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 kunnen 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

Deel III

Distributed Data Storage & Processing