whitehorses

member of the human network

WHITEBOOK

Maak Software Schaalbaar Met Microservices

April 2018

Auteur:
Patrick Sinke
INTEGRATIE SPECIALIST

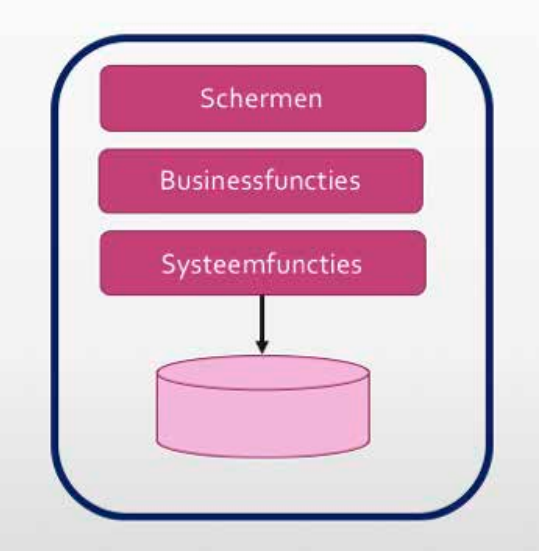
De geschiedenis van softwarearchitectuur

Inleiding

Een softwarearchitectuur is een set van regels en principes die de elementen, de structuur, het gedrag en de relatie tussen verschillende componenten beschrijft. Om te begrijpen wat de positie van microservices is, is het nuttig om de eigenschappen en evolutie van softwarearchitecturen te benoemen.

Monolieten

De eerste grootschalige softwaresystemen waren volgens het monolithische principe gebouwd. Monolithische applicaties bestaan uit één enkel programmeerplatform met meerdere modules. De opdeling van modules is op basis van bedrijfsproces of technische functionaliteit, en alle lagen (presentatie, business, database) in de architectuur zijn onderdeel van dezelfde applicatie.



Figuur 1 -Een monolitische applicatiearchitectuur

Monolithische software was dé standaard in de jaren 80, maar al snel werden de beperkingen van deze architectuur duidelijk. De onderhoudbaarheid is laag, doordat een wijziging aan één onderdeel invloed heeft op grote delen van het systeem. Daardoor neemt de kans op introduceren van bugs exponentieel toe met het aantal regels code, alsook de benodigde inspanning voor een release.

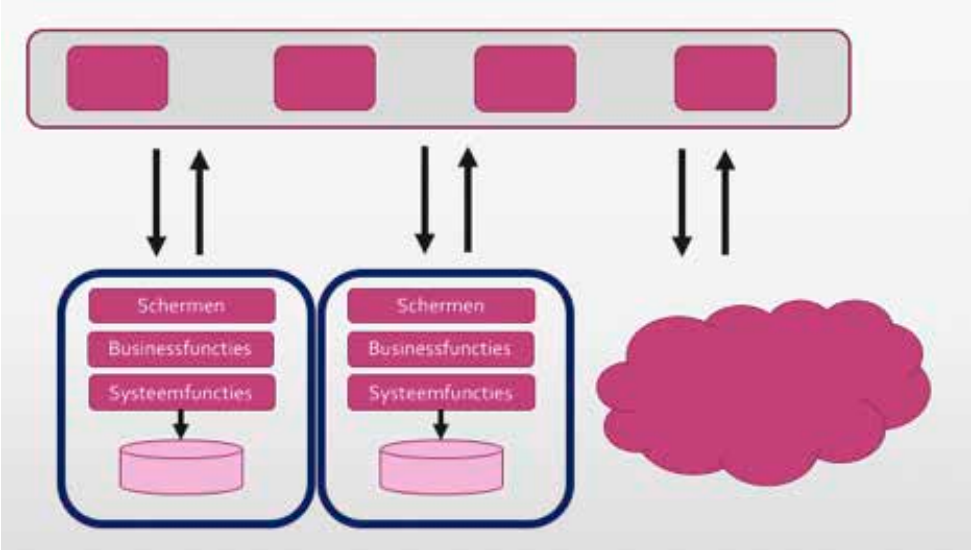
De oplossing daarvoor was het bouwen van meerdere applicaties naast elkaar, waarbij het probleem ontstond dat deze applicaties met elkaar moesten communiceren, waardoor een wirwar van onderlinge connecties ontstond. Ook dit was op een gegeven moment niet meer onderhoudbaar.

Een nieuwe aanpak

Het antwoord op deze problematiek was de Enterprise Application Integration; een architectuurlaag die verschillende systemen op een eenduidige manier met elkaar verbindt door middel van een centrale hub. Dat principe zien we nog steeds terug in de Enterprise Service Bus.

De volgende stap daarin is de service-georiënteerde architectuur, of SOA. Door het introduceren van een diensten-leveranciermodel en het introduceren van applicatielagen worden componenten in een applicatielandschap ontkoppeld en worden diensten (services) onafhankelijk van hun afnemers, meer modulair en generieker.

SOA wordt zo een abstractielaag over bestaande architectuur en koppelt legacysoftware aan business services van externe partijen (waaronder clouddiensten). Oude applicaties kunnen naadloos met nieuwe softwarecomponenten samenwerken!



Figuur 2 - een service-georienteerde architectuur (SOA)

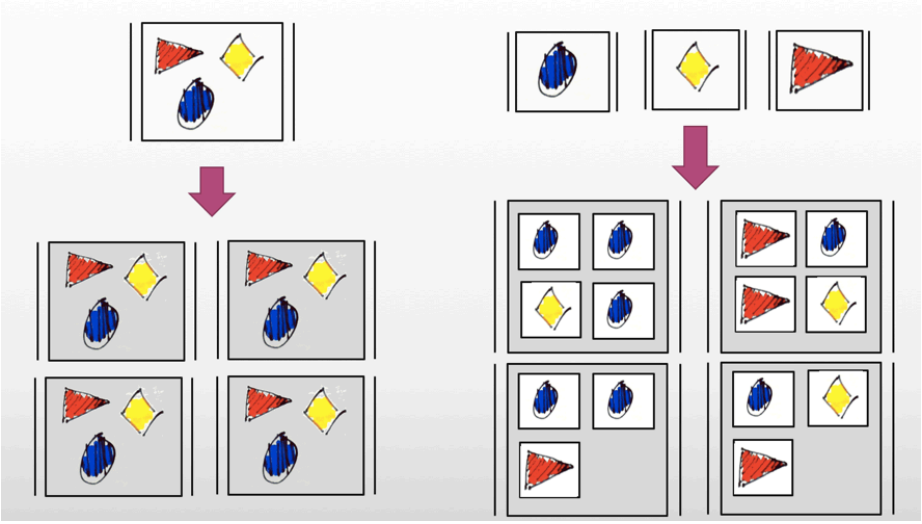
Zoals met alle architecturen, heeft ook SOA zijn toepassingsgebied. Het is ontstaan als oplossing voor bestaande applicatie-integratie en ontvlechten van functionaliteit uit monolieten. Dit werkt goed, en doen we nog dagelijks!

Soms is er echter de wens om nog sterker te ontkoppelen, sneller te deployen met CD/CI (daarover straks meer), en nog sterker agile te werken. Zeker als er minder afhankelijkheid is van legacy software kan het zijn dat SOA niet de meest passende keuze is. Welke architectuurkeuzes moet je maken om aan die wensen te voldoen?

Microservices

Als iemand zegt dat de Microservices architectuur (MSA) oude wijn in nieuwe zakken is, is dat misschien ten dele waar. Veel van de concepten in Microservices zien we ook terug in SOA, of zijn daar een evolutie van. De Microservices architectuur wordt niet voor niets soms “Fine grained SOA” genoemd; SOA met de nadruk op kleine, mogelijk kortlevende, “vluchtige” componenten. Maar wel componenten die één ding heel erg goed kunnen; dat is een belangrijke filosofie van MSA.

Het hele idee van MSA is om *business functions* op te breken in kleine taken, en voor elke taak is er een aparte microservice. Een typisch systeem kan uit 2 microservices bestaan, maar ook uit honderden. Dit alles is afhankelijk van het aantal en soort business requirements en hoe klein de functionaliteiten opgebroken kunnen worden.



Figuur 3 - Schaling van monolitische applicaties (links) vergeleken met microservices (rechts)

Neem als voorbeeld een HRM pakket met verschillende functies: een indiensttreding, de salarisbetaling, het fiscaal jaaroverzicht. Later komt daar misschien een “serviceaanvraag laptop” bij. Elke functie heeft andere eigenschappen met betrekking tot frequentie van gebruik en benodigde resources. Als we figuur 2 bekijken dan zien we links een monolitische applicatie met meerdere modules. Een gele module kan zijn een service aanvraag laptop, een functie die weinig gebruikt wordt. De blauwe module “salarisverwerking” vraagt veel meer resources. Een klassieke applicatie is niet in staat om per module aan de behoefte te voorzien en schaalt daarom niet efficient. Rechts is een microservices applicatie waar de losse modules vrij verdeeld kunnen worden over het serverpark en daarmee is de hardware veel efficienter in te zetten.

Kenmerken

Wat maakt nu dat een applicatie als Microservices architectuur bestempeld kan worden? Er zijn verschillende kenmerken die, wanneer zij aanwezig zijn in een applicatie, het tot een MSA maken.

Ten eerste is er de meest voor de hand liggende, het systeem moet uit meerdere modules bestaan die elk hun functionaliteit als services aan de buitenwereld tonen. Elke module heeft een achterliggende businessfunctie en er mag niet of nauwelijks koppeling tussen modules zijn; "*loosely coupled*".

Het volgende onderscheid is dat elke module in een *andere taal of framework gebouwd* kan zijn. In de praktijk is het niet verstandig voor elke functie een andere toolset te gebruiken, aangezien de nadelen van het leren van nieuwe frameworks al snel tegen de voordelen ervan opwegen. Maar het is zeker het overwegen waard om in specifieke gevallen toch die ene taal te gebruiken die een nuttige toevoeging is aan je DevOps-omgeving.

Continuous Integration, Delivery en Deployment (*kortweg CI/CD*) wordt genoemd als een van de cruciale gereedschappen in een succesvolle microservices implementatie. Het biedt, wanneer goed geïmplementeerd, een agile ontwikkelcyclus. De uitdaging zit vooral in de grote hoeveelheid build pipelines die vanzelfsprekend ook veel diverser zijn dan in een monolithische omgeving.

Een ander belangrijk kenmerk van de MSA zijn API's. Afhankelijk wie je het vraagt, zal het antwoord zijn dat die twee (bijna) hetzelfde zijn, dat zonder API's geen microservices zouden bestaan, of dat de ontwikkeling van die twee hand in hand gaan.

Aan microservices wordt een hoge mate van schaalbaarheid toegekend, mede door moderne, elastische back-end architecturen, echter wil dat niet zeggen dat de applicaties als geheel ook meteen *schaalbaar* zijn. Gecentraliseerde systemen of databases (die niet altijd te vermijden zijn) moeten die schaling ook kunnen volgen, en een microservices architectuur kan het uiterste vragen van de gebruikte API's, waarbij we op het volgende punt komen:

API Management is van groot belang, Service discovery en API catalogs zijn hulpmiddelen om al die, potentieel honderden, microservices vindbaar te houden. Hierbij is Service Discovery typisch iets voor runtime beheer: op welk adres bevindt zich een service en welke methodiek heb ik beschikbaar om een service te vinden. API catalogs zijn feitelijk niets meer dan bibliotheken (in de vorm van softwareproducten) waarin beschreven staat welke services er zijn en welke functionaliteit zij beschikbaar stellen.

Tenslotte gaan microservices niet alleen om techniek en architectuur, maar ook over werkwijze. Multidisciplinaire teams zijn een belangrijke factor in het realiseren van microservices-applicaties.

Met de introductie van agile teams waren de zogenaamde techniekSilo's, waarbij de DBA's bij elkaar zaten en de ontwerpers ook bij elkaar maar op een andere locatie, al niet meer in zwang. Waar vroeger de teamstructuur een afspiegeling van de organisatie was en niet geoptimaliseerd voor het op te leveren product, is het streven in multidisciplinair teams om weinig afhankelijkheden tussen de teams te hebben en ze een hoge mate van autonomie en verantwoordelijkheid te geven.

Conway's wet stelt: *Organisaties die softwaresystemen maken zijn beperkt tot het produceren van ontwerpen die kopieën zijn van de communicatiestructuren van die organisaties*. De les die hieruit te leren valt, is dat de enige goede organisatiestructuur van softwareteams er één is die aansluit bij de eisen van het op te leveren product. Daar is dus ook geen one size fits all oplossing voor en moet per project opnieuw bekeken worden. En dat is natuurlijk precies wat Agile werken ook inhoudt.

Onmisbare tools voor het inzetten van microservices

Wie zoekt naar informatie over microservice development, komt al snel het begrip CI/CD tegen. Deze werkwijze is een voorwaarde voor het succesvol inzetten van microservices. De tools die dit ondersteunen hebben een aantal eigenschappen die mogelijk maken dat er in hoog tempo nieuwe releases van kleine componenten gedaan worden, die onafhankelijk en lichtgewicht zijn en die gezamenlijk zich toch als een robuuste, consistente applicatie gedragen. Wat zijn deze producten en wat kunnen zij?

Een eigenschap van Continous Integration and Delivery is de zogenaamde "nightly build", waarin alle wijzigingen van die dag 's nachts worden gecompileerd, zodat er de volgende ochtend een nieuwe werkende versie van de software klaar staat voor de testers. De grote uitdaging is om te voorkomen dat die build compilatiefouten heeft door afhankelijkheden tussen de wijzigingen. *Jenkins* is een automation platform voor CI/CD pipelines die alle wijzigingen tussen code repositories en software platforms beheert en zoveel mogelijk taken daarin automatiseert.

Docker is het bekendste voorbeeld van een “software container platform”, ofwel software die in staat is elke applicatie als een onafhankelijke, verplaatsbare “container” te deployen en te beheren. Het is een tussenvorm tussen klassieke runtimes en een virtual machine. Met Docker heeft elke container zijn eigen CPU, geheugen en netwerkbronnen, maar delen de containers wel één OS Kernel. Dit maakt containers lichtgewicht en schaalbaar, maar het heeft ook nadelen. Zo zijn ze niet gemaakt om geupdate te worden; een nieuwe versie van de software betekent een nieuwe instance van die container. Ook is er geen persistente data-opslag mogelijk, en zijn ze ook niet geschikt voor bijvoorbeeld agents of daemons. Dit comt omdat Docker *stateless* is. Simpel gezegd: Docker onthoud geen informatie over de containers.

Kubernetes is een ander platform dat in staat is dergelijke containers te schalen en beheren over grote clusters. Zo heeft Kubernetes functies zoals *service naming and discovery*, load balancing, health monitoring, rolling updates, high availability en horizontal autoscaling (horizontal scaling gaat over het toevoegen van meerdere machines aan een cluster, vertical scaling is het toevoegen van meer rekenkracht en geheugen aan een bestaande server). En het mooie is, Kubernetes kan in een Docker container draaien.

Conclusie

Wat zijn nu de voordelen van Microservices architecturen ten opzichte van hun voorgangers? Zoals inmiddels wel duidelijk is, is het een combinatie van factoren. Veel van de valkuilen van monolitische en SOA software zijn onderkend en de MSA bouwt duidelijk voort op de geleerde lessen. Er zijn de laatste jaren veel hulpmiddelen ontwikkeld die de snelheid waarmee software gedeployt kan worden drastisch versnellen, met een kortere *time to market* als gevolg. Applicaties kunnen schaalbaarder worden door inzet van modulaire containers . De eisen zijn ook veel hoger; datavolumes zijn niet meer te vergelijken met 10 jaar geleden, en ook het aantal gelijktijdige gebruikers van een dienst is duizelingwekkend. Dat lukt gewoonweg niet meer met een monolitische applicatie gebouwd door een klassiek watervalteam. En daar schuilt misschien wel de belangrijkste kracht van een MSA: het inherent loslaten van oude organisatiepatronen en softwaredogma’s. Het geeft de ruimte om per businessonderdeel, per product en zelfs per release te kiezen wat nodig is om een snelle time to market te realiseren. Het is wel goed om te realiseren dat microservices ook complexiteit met zich meebrengen, dus zet ze alleen in waar schaalbaarheid, een sterk ontkoppelde architectuur en een korte development lifecycle onontbeerlijk zijn.

Verder lezen en referenties

- https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd
- https://diginomica.com/2014/07/02/virtualization-dead-long-live-containerization/
- https://www.infoworld.com/article/3253284/containers/what-you-need-to-know-about-microservices-tools.html
- https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html