# Software architectuur

Functionele requirements: Wat het moet kunnen  
Niet functionele requirements: Hoe het het moet kunnen (verwezenlijked via software archi.)

Software architectuur = structuur componenten en hun relaties  
verschillende perspectieven: **4+1 model**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Logical view | De code/klassen (OO) | Classes, packages | Inheritance, association, depends-on | Class diagram |
| Process view | De processen. (wat sequentieel/parallel) | processes | IPC | Activity diagram |
| Physical(deployment) view | Welke executables staan waar / op welke hardware deploy ik dus | Fysieke of virtuele machines | networking | Deployment diagram |
| Development(implementation) view | gecompileerde code | Modules( bv jar)  Components (bv executables) | Dependencies (welke moet je nog uitvoerne) | Module/  subsystem diagram |
| Use cases |  |  |  |  |

# Reactive manifesto

Paar architectuur eigenschappen voor schaalbare robuste apps

* Message driven: Losgekoppelde componenten praten enkel via vastgelegde boodschapenformaten. Los in tijd (elk eigen tempo). Los in ruimte (ander proces, server of datacenter mogelijk)
* Responsive: Beter om loading bar te tonen dan niks. Trage delen mag de rest niet vertragen
* Elastisch: resources vermeerderen of verminderen afhankelijk workload
* Robuust: als 1 deel faalt moet de rest blijven doorgaan en moet het zichzelf herstellen. Checken op basis van: Service metrics (#schijfoperaties) en Business metrics (orders/min)

Microservices

## Architectuur

Elk van de views kan een layered, hexagonaal, monolitisch of microservice stijl hebben

**Layered**: Presentatie / business / persistentie

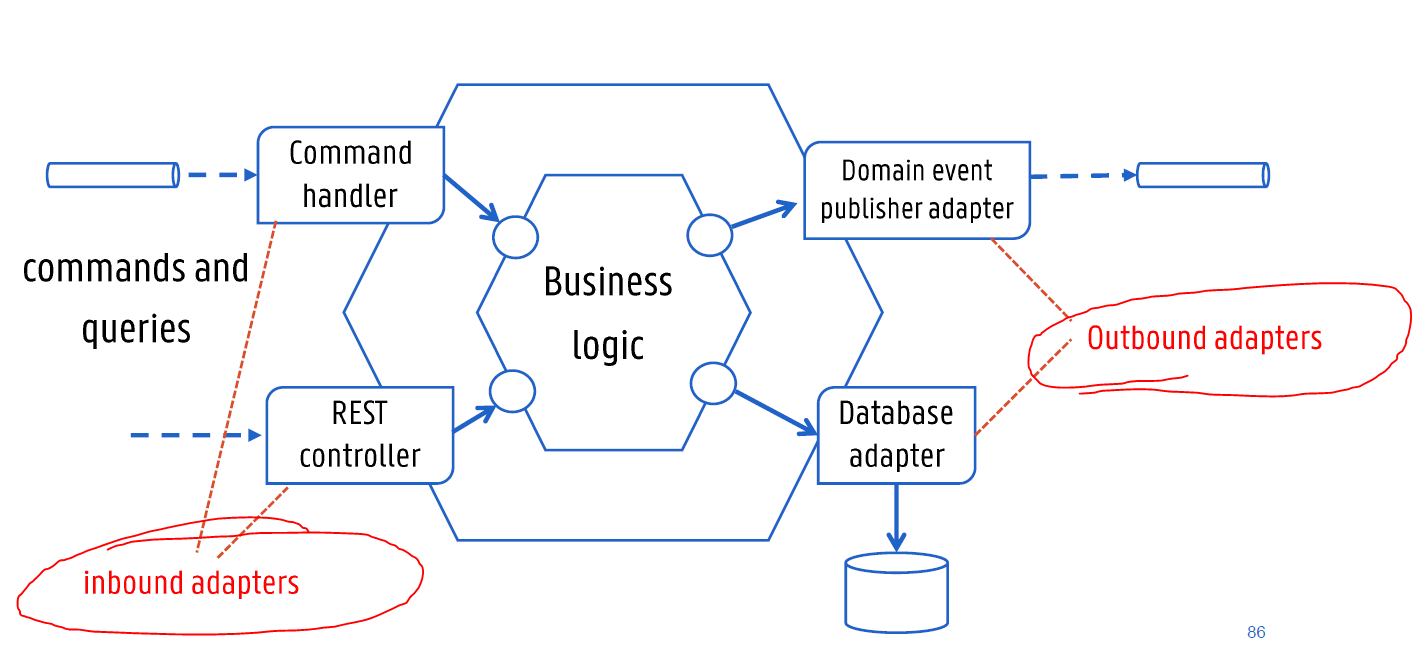
Bv. Java EE

+: solid. Makkelijk. Valt goed te testen (Mocks). Veel frameworks

-: verschillende type clients maar maar 1 laag. Meerdere type databanken maar maar 1 laag. Persistentie hangt veel af van businness terwijl dat niet mag

**Hexagonaal**: Business logic centraal

erond in/outbound ports die use cases implementeerd (technologie onafhankelijk)  
Inbound port: bv. Bedrijf.maakPizza da wordt tegen ander programma gezegd da pizza moet teruggeven  
Outbound port:



Adapters (lek injection) implementeert de uses cases gemapt op een bepaalde technologie.

**Monolitisch** één build die alles omvat

Bv. een Jar file

+: easy // makkelijke grote veranderingen want tmoet toch gans opnieuw builden // makkelijk te testen en deployen

-: moeilijk te onderhouden me veel code//moet alt gans rebuilden vo kleine change //moeilijk te scalen // 1 bug en alles plat // vast aan 1 technologie

Opletten: big ball of mud (god classes,..)

**Microservices**

* Meerdere onderdelen
* Elk een type logische view (hexa,..)
* Elk eigen db
* Communicatie via IPC
* Service registry houdt bij welke microservice waar bevindt
* Elke microservice heeft één taak die niet groter is dan nodig (dus niet zo klein mogelijk)
* **Looseley coupled** onderling: alle interacties via API / geen gedeeld databank schema / elk kan eigen technologie kiezen / dus aparte code base (dus polyglot=meertalig)/

-: moeilijk te vinden wat services van maken / gedistribueerde systemen die onderling moeten intrageren complex / meer operationele complexiteit (servers / updates / .. managen)

## Decomposition

Via 3 step iterative process (iterative want moet soms stap terug gaan. Trial error)

Stap 1: Identify system operations

* High level domeinmodel
* Op basis van vakjargon
* Zoeken achter werkwoorden en zelfstandige naamwoorden
* Doorvragen op precondities enzo

Stap 2: Define services by business capabilities

* Services (groeperingen dus) kiezen op basis van de verschillende business capabilities (hoe ze geld maken). Dus op basis van context groeperen of je krijgt een **god** class
* Maar opletten voor ‘te veel netwerkverkeer’ bij sommige onderverdelingen.
* Moet asychroon kunnen werken constant

Stap 3: Define service API’s

* De API’s van elke service nader definieren.
* 2 Type functies per API. Enerzijds voor client, anderzijds voor andere services communicatie
* Denken welke operatie in welke service en of die beschikbaar moet zijn voor buitenwereld. Of misschien hebben ze dingen uit die buitenwereld nodig

# Interactie stijlen tussen microservices

**One to one:** elke client request is verhandelt door 1 service  
 - Request/response (sychroon)  
 - Request/Async response (asychroon)  
 - One way notifications (geen reply verwacht)

**One to many**: Elke client request verhandelt door verschillende services  
 - Publish/subscribe (client pusblished en verwerkt door 0..N services)  
 - Publish/async response (client publishes request message; wacht dan ff voor antwoorden van geintereseerde services)

**Synchroon:** client wilt een snel getimed antwoord. En blokkeert misschien zelf tot hij het heeft

**Asychroon**: Niet persee onmiddelijk antwoord.

Les 2

Systeemontwerp les 2

Slide 41:

## Problemen met synchronisatie:

Je wacht en komt niks. Probleem ?

* Connectie (kabelbreuk bv)
* service ligt plat.

Proxy zelf kan da niet weten welke van de twee het is.   
In proxy: robust maken: blokkeert niet door een timeout in te stellen  
Fallback strategie nodig. Hierbij denken vanuit business perspectief: ofwel “kom straks terug” of iets anders tonen

## oplossing 1: Bulkhead

analogie tussenschotten in schip.

Technisch: probleemstelling. (schema linksboven)Alle threads zitten vast in microservice X dus er zijn er ook geen beschikbaar meer uit de threadpool voor microservice Y.

oplossing:(rechtsboven) microservice Y zal nu wel nog aangesproken kunnen worden.

## Oplossing 2: circuit breaker

bij elke rode pijl (call die niet werkt) gaat een teller omhoog. ‘Zekering’ houdt in oog hoeveel open verbindingen er zijn.. te veel? Dan doet ie fail.

Fallback strategieën: geen vaste strategie, hangt van de use case af.

Q: Synchroon heeft veel nadelen. Waarom gebruikt dan?   
A: Kan ni anders. Meeste dingen hebben REST nodig.   
 + wijd gekend en geen extra infrastrucuur nodig.

# Asychrone communicatie

Channel kan je zien als queue (soort buffer). Message niet alleen voor events, maar ook voor commando’s.

Een kanaal kan point to point zijn of point to multipoint(meerdere services verbonden).

Je geeft zelf aan service wat ie moet weten: zoaks returnaddress. Body is wat je wilt meegeven: commando/…

messageID: nodig want client doet ondertussen al verder (client moet dan zien voor welke request dat hij een reply krijgt).

Slide 55: zowel kanaal als type(Die R’en) en inhoud moet je specifieren.

Q: Verschil command/event?   
A: Commando kan je weigeren, bv order().  
 Event kan je niet weigeren, is al gebeurd.

## Broker

Service die **geen business logica** is maar deel van uw systeem.

Grote voordeel: betere **garanties**:   
als ik A ná B stuur.   
- Komt B dan eerst aan(message ordering)?  
- Komt het wel aan? (niet alle queues kunnen die garanties geven)  
- hoeveel krijg ik er door (scalability) ?...

### 2 grote problemen broker: (wordt in apache kapka opgelost)

#### Probleem 1: competing consumers

In 1 systeem meerdere instanties zelfde service. (bv tickets tomorrowland). Klant maakt order en cancelled onmiddelijk. (groene shit slide 57 is 1 zelfde service). Maar in 2de service komt cancelling toe en kan zijn da die service 2 andere processing time heeft door andere aantal dingen da al in queue zat. En dan kan cancel() eerst gebeuren en smijt ie da gwn weg want order bestaat nog niet. Dan pas ga den eerste de order aanmaken en wordt het dus nooit gecancelled.

#### 2de probleem

meerdere services da geinteresseerd zijn in een order: marketing\_service wilt hem ook moeien. Maar kan zijn da marketing die order uit die queue haalt en kitchen dus er niet meer aan kan of omgekeerd.

## Apache kafka

* meerdere topics (kanaal). Draaien partitities (deel v een topic)op verschillende machines (broker). Dus elk topic is gepartioneerd.
* Zookeeper dient voor synchronisatie tussen verschillende machines.
* Client moet Shard key geven (concreet bv. hash op een veld van een message). Zodat zijn shit ma in 1 partitie terecht komt.

Uitlezen (consuming): consumergroups= 1 of meerdere consumers zitten erin. Er is **maar** 1 client die 1 bepaalde partitie leest per consumergroep.   
consumer group B zouden bv alle kitchen services kunnen zijn.

Per identieke game service (scaled horizontaal): elk 2 rollen.

* Events accepteren en doorsturen naar events topic.
* Process: verwerkt alles bv van userID 1 tot 1000. Voordeel: als ie enkel da beperkt aantal moet doen. Dan kan ie cachen en gaat sneller.

Dus samengevat: eerst binnenkomen. Doorsturen en sorteren en dan terugsturen.

#### Ander probleem

ge zet dingen in queue en iemand stuurt acknoledgement (wordt dan verwijderd uit queue). Stel er zat syntax fout in message => receiver crashed. Andere receiver leest da nog eens en order bv 2x geplaatst.   
**Oplossing: atomaire transacties** = in consumer kant ook “insert processedmessage” (aankruisen dat het verwerkt is) en order update (order plaatsen) doen. moet alle 2 uitgevoerd worden of helemaal niks => idempotent

Slide 66: oplossing: Het kan goede oplossing zijn voor data te dupliceren (data opslag is cheap). Bv. User adres veranderd ni veel. Dus zal meestal geen probleem zijn. Order service kan nu in eigen tabel kijken en heeft call chain niet meer. Dus as er ene plat ligt no problem.

Andere oplossing: “ok ik heb u call gekregen. Maar kom straks ke terug”. Bij elke call da terug komt update je order status. Als client dan opnieuw polled, krijgt ie alt antwoord me wa nu beschikbaar is. ( we moeten afleren altijd volledige info in 1 keer te willen geven. Stukjes al geven intussen)  
  
api first: eerst naar calls kijken

saga

serie van transacties in bepaalde volgorde.

Motivatie:

* anders heb je altijd locks voor syncronisatie redenen
* NoSQL: zeer schaalbaar maar mindere garanties (heeft geen transacties) => oplossing: saga’s gebruiken

Create ticket kan je weigeren en kan dus falen

Approve ticket kan niet falen (business perspectief): want is al geauthoriseerd. Dus is gewoon nog ne update doen dus het **moet** gebeuren.

Heb ik een goede saga gemaakt?  
Laatste X aantal stappen (alles acknowledgen) slagen **altijd** dus heb je geen compensating transactions voor nodig.

Choreaography example: elke event trigger een aantal commando’s. ma ge hebt dus ni 1 plaats op u code waar staat hoe alles moet gebeuren.

orchestrator klasse maken: stuurt commando (in een message). Alle logica nu in dien klasse.  
voor alles ontkoppelt te houden. Laatste call ook nog doen ipv intern te doen.

# Business logic of a single microservice

Les 3

# Queries

Oplossen dat je data uit verschillende microservices moet halen:

1) API composition pattern (eerste keuze)

* Kunnen verschillende databanktechnologien zijn dus deze manier niet zo evident dan (sql, nosql,..)
* We steken die in Api gateway of service (lokaal netwerk dus sneller)
* Probleem: conflicten door syncronisatie problemen tussen de verschilledne databanken

2) zeer complexe queries: command-query responibility segragation (opsplitsen)  
ipv services apart aanspreken => 1 grote database

* 2 API’s en read afsplitsen naar query model.
* Service zal domain event genereren
* (Proactief luisteren )Query model vangt da op
* En slaagt die data in aparte databank op
* Die databank is bv. Wel gespecialiseerd in die complexe query
* Nadeel: extra service, extra management, replication lag (niet meest up to date systeem)
* Voordeel: lost nadelen composition pattern op

Deployment and orchestration

Runtime service management: zijn alle 5 instanties nog in goede staat? Is disk nog niet vol? Enz..

Monitoring interface: als er iets fout loopt moett development team op de hoogte gebracht worden

DevOps: zien dat uw services daje maakt ook kunnen runnen in deployment

Deployment pipeline: voert testen uit

Push based: Elke keer je iets uitrolt:

* Deployment environment: krijgt opdracht om iets te deployen
* Ie maakt eerst aantal **environment variabelen die configuratie definieren**

Pull based externaliszed config: pulled configuratie uit dingen als git (opletten met db gegevens)

# Observable services

Logging: voor analyse over microservices heen

Health check API : Is hij nog alive?

Auditing: wat doet de gebruiker

Metrics server: hoeveel requests worden afgehandeld? Extra instantie nodig?

Log aggregation:

* Weggeschreven naar file of remote db
* Maar je wilt service onafhankelijk plaats waar hij uitgevoerd (moet toegangrechten hebben)
* Algemene manier: output channel op stdout zetten (heb je altijd).
* Dan in service dingen da die stdout onderschept en dan via pipeline zet in 1 logging server.

Disttributed tracing:

* Omte debuggen
* Trace = externe partij die methode van je interface opvraagt
* Trace krijgt bij binnenkomen bepaald id
* Call chain dan
* Span: tijd tussen toekomen en antwoord op request
* Span daaronder: tussenliggende calls
* Allemaal ID’s aan geven
* Dan kijken zo bij wie het fout loopt in distributed tracing server waar alles samenkomt

Audit logging:

* Bv. Als je iets wilt kopen, maar niet betaald. Dan krijg je mails

Applicatie metrieken

* Hoe goed werkt uw applicatie?
* Houd bv uw CPU verbruik bij
* Je weet bv dat er 5 instanties zijn. Verbruiken ze allemaal evenveel CPU? Indien niet, hoe komt dat?
* Cloudwatch: trekt die metrieken bij en geeft alarm boven bepaalde drempel.

# Packagen: containers

Allemaal verschillende technologien kunnen deployen

Virtualizatie: virtueel beeld op onderliggende platform  
Hypervisor: neemt elk vd virtuele machines en geeft hun de illusie da die machine vo hunzelf is.

Os-level virtualization: 1 OS en alles verpakt in containers.

Docker: kernel code dingen gebruiksvriendelijk gemaakt want niemand werkte ermee

## Linux containers

Linux control groups: Controle mechanisme nodig: hoeveel CPU enzo mag iedereen gebruiken

## Docker

Client doet commando’s naar docker host. Op host aantal images ter beschikking.

Syntax:

From ubuntu van clean ubuntu beginnen

Dan webpagina toevoegen

Dan uw webserver nog toevoegen

Entrypoint: wat meot gebeuren als je docker container process start

Een laag is een aantal bestanden. Maintainer=neemt geen ruimte in bij besturingsysteem

Nut: lagen zijn read-only. Als je 2 container draait dan kunnen die zelfde laag gebruiken. Moet je dus niet dupliceren. Alleen writable layer wordt gedupliceerd voor bv log files.

# Container orchestration

Week 8: Dimensions of cloud computing

Essentials:

* Op aanvraag: serverpark dat uitegerust is voor dienst aan te bieden wanneer nodig
* Elastischiteit: soms veel gpu nodig andere keer cpu,...
* Cloud provider steek daarom resources in soort pool.
* Metered: te betalen naar verbruik
* Network access: je moet niet naar serverpark gaan ervoor

Zakenmodel:

* Cloud user: gebruikt de infrastrucuur
* Cloud provider: bied het aan
* End user
* Als gebruiker zie je niet individuele resources.
* Meerdere gebruikers op 1 infrastructuur zonder dajet als gebruiker door hebt door de pooling
* Metal as servicel: zo ne bak huren
* Virtuele hardware : infrastructure as a service:
* Daarin draait een OS: container as a service: die containers daje gemaakt hebt deployen
* Kafka of databank nodig 🡺 platform as a service. Biedt een bepaalde instantie van kafka/db aan (=middleware: tussen OS en applicatie. Met andere woorden geen applicatie opzich (bouwblok))
* Software as a service: zoals bv photoshop licensie huren / athena
* Spot pricing: prijs afhankelijk van dag/nacht en tot dag
* Containers as a service: vooral kubernetes.

deployment

* Hybride model: zelf basis infrastructuur voorzien maar bij piek zo een publieke cloud gebruiken maar gevoelige info deploy je zelf nog

# Elastisch schalen

* Belangrijkste reden voor cloud
* Constant monitoren automatisch en dan bijschafen
* Die bedjes linksonder: horizontaal schalen (meer van hetzelfde). Bedje rechtsonder:Verticaal schalen (upgrade uitvoeren)
* Verticaal schalen: je kan niet meer downscalen.
* Horizontaal schalen: beter kostengewijs
* Nooit volledige applicatie schalen
* Altijd bepaalde service die bottlenecked

Slide 96: hoeveel requests moeten binnen komen zodat backend A eentje omhoog gaat:

* Minstens 3
* Alarm gaat af na 80% 0,8 \* 100 maar ge hebt er drie X3
* 3\*100\*1\*0.8 \* 2
* Instances\*maxrate\*capacityscaler\*utilizationtarget \* #backends

# Data intensieve systemen

# Big data

* Volume: veel data (bv door sensoren enzo)
* Velocity: moet snel gaan. Bv eerst verwerken en n dan pas opslaan
* Variety: meer dan relationele data, foto’s, video’s,..
* Veracity: waarachtigheid: data kan rommelig zijn (tweet wordt niet in perfect nederlands geschreven of ruis bij sensoren // niet netjes afgelijnde data)
* Sql schiet te kort bij one to many relaties. Veel tabellen voor nodig, bij json zit alles samen
* Bij webshop, veel producte, veel eigenschappen dus veel null values in uw kolommen, en gigantisch veel kolommen => document beter. Me json doeje waje zelf wilt, schema ligt niet vast

# Data storage and retrieval