May 13, 2019

Terence Koch

Microservices

Fontys - JEA6

Inhoud

[Introductie 2](#_Toc10243877)

[Vooronderzoek 3](#_Toc10243878)

[Voor- en nadelen 3](#_Toc10243879)

[Ontwerp patronen 4](#_Toc10243880)

[Toepassingen 7](#_Toc10243881)

[Conclusie 7](#_Toc10243882)

[Bronnen 8](#_Toc10243883)

[Casus – Animal Shelter 9](#_Toc10243884)

[Omschrijving 9](#_Toc10243885)

[Requirements 9](#_Toc10243886)

[Non-functionele requirements 9](#_Toc10243887)

[Architectuur 10](#_Toc10243888)

[Tools 12](#_Toc10243889)

[Acceptatiecriteria 13](#_Toc10243890)

# Introductie

In dit document wordt de uitwerking voor de school opdracht “Capita Selecta: Micro Service” omschreven.

In deze opdracht komen de volgende onderwerpen aan bod:

* Een micro service architectuur kunnen ontwerpen en je ontwerpkeuzes kunnen verantwoorden
* Je kunt een casus uitwerken waarbij je gebruik maakt van de micro services architectuur.
* Je kunt de voor- en nadelen van je realisatie opsommen

Het eerste deel is gewijd aan het onderzoek en vervolgens wordt de casus uitgewerkt.

# Vooronderzoek

Er is een kort vooronderzoek gemaakt om kennis te maken met Microservices. Met onderstaande deelvragen wordt een antwoord gezocht op de vraag: “Wat is een microservice en hoe kan deze praktisch toegepast worden”.

**Hieruit zijn onderstaande deelvragen afgeleiden:**

* Wat zijn de voor- en nadelen van een Microservice architectuur.
* Welke ontwerp patronen kunnen toegepast worden in een Microservice architectuur.
* Wat zijn praktijkvoorbeelden van microservices waarin deze patronen toegepast kunnen worden.

## Voor- en nadelen

Onderstaande voor- en nadelen zijn opgesteld ten opzichte van een standaard, goed geschreven monolithische applicatie. Wat zijn de voor- en nadelen van een microservice architectuur:

|  |  |
| --- | --- |
| **Voordelen** | **Nadelen** |
| **DevOps:** Geeft de mogelijkheid om CI/CD optimaal te gebruiken | **Complxiteit:** Architectuur en deployment moeten goed opgezet en duidelijk zijn |
| **Schaalbaarheid:** Eenvoudiger te onderhouden van grote codebase | **Communicaties** tussen services moeten opgezet worden (REST) |
| **Schaalbaarheid:** Eenvoudig om uit te breiden met nieuwe services | **Beveiliging:** Meerdere ingangswegen die beveiligd moeten worden |
| **Foutafhandeling:** Fout met 1 service betekent dat andere services nog draaien (bv. Offlinemodus) | **Monitoring:** Overzicht van status alle systemen is complexer |
| **Testbaarheid:** Testen moeten niet alles omvatten | **Consistentie:** Data moete constant blijven tussen verschillende systemen |
| **Performance**: pakket wordt deel voor deel geleverd in plaats van alles tegelijk. |  |
| **Flexibiliteit:** Bij veel verkeer tijdens piekmomenten hoeft slechts 1 deel opgeschaald worden of er kunnen andere services uitgezet worden om het verkeer goed te verhelpen. |  |

## Ontwerp patronen

Verschillende patronen zijn veel voorkomend bij de communicatie en opzet van microservices. Hieronder enkele belangrijke patronen met een voorbeeld en eventuele architectuur.

**Domain driven design**: domein experten van iedere microservice aanstellen.

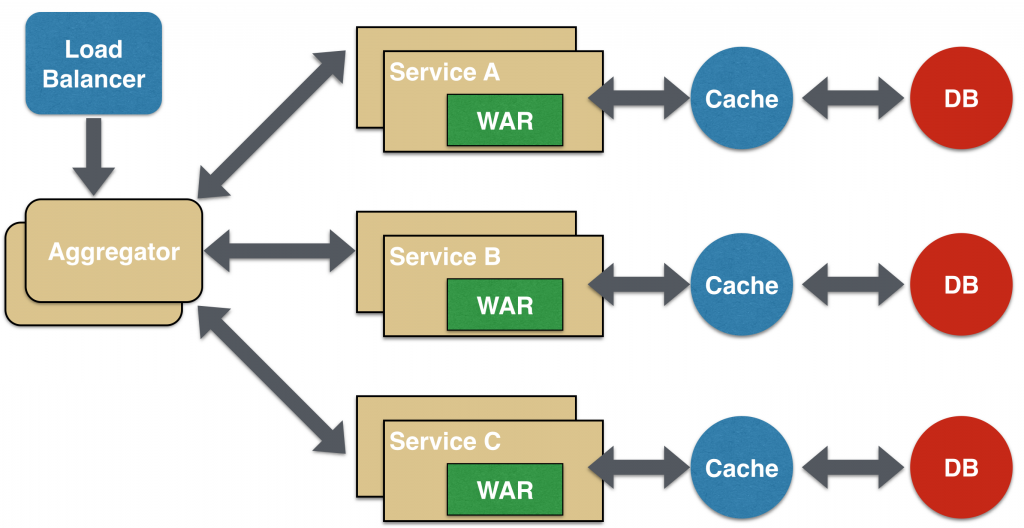
*Voorbeeld*: Een bank heeft een grote kennisbank applicatie dat door verschillende branches gebruikt wordt. Zo is er bijvoorbeeld een kennisbankservice voor de contactcenters, fysieke gebouwen, online bankieren. Elke service bevat informatie van andere domeinen dat belangrijk is voor de bank. Voor elke service is er dus 1 aangewezen domein expert die praktische ervaring heeft en zo precies kan bijdragen tot het stuk functionaliteit wat zij nodig hebben.

**Database per service**: Een aparte database wordt per service gekoppeld. Hiermee kunnen services helemaal los van andere microservices schalen en zich aan te passen.

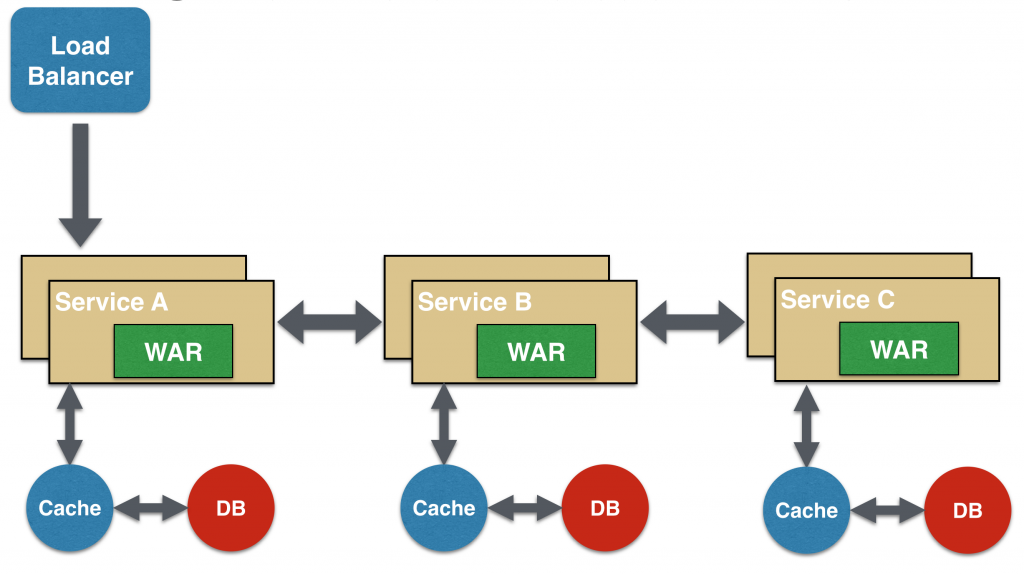
*Voorbeeld:* Bij de opdracht rekeningrijden zou het administratiesysteem een andere “User” tabel hebben als in de “user” tabel van de gebruikersapplicatie. Door deze te ontkoppelen ontstaat er ook minder verwarring tussen de subsystemen.

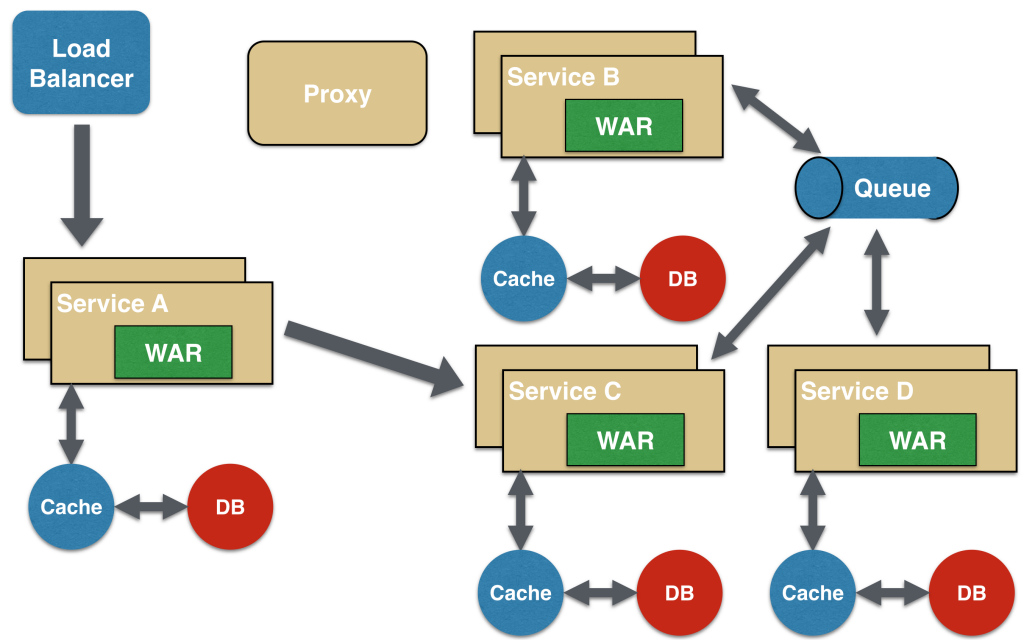
Een aansluitend patroon is “**shared database per service”** Hierbij worden verschillende services gekoppeld met eenzelfde database. Dit zorgt voor minder databases en dus meer consistentie van data maar ook voor minder ontkoppeling.

**Aggregator pattern:** Een systeem (microservice) dat met verschillende andere microservices communiceert. Een inlogmethode praat zo bijvoorbeeld met een authenticatie service en een accountservice. Beide worden gekoppeld aan de hand van een webpagina.



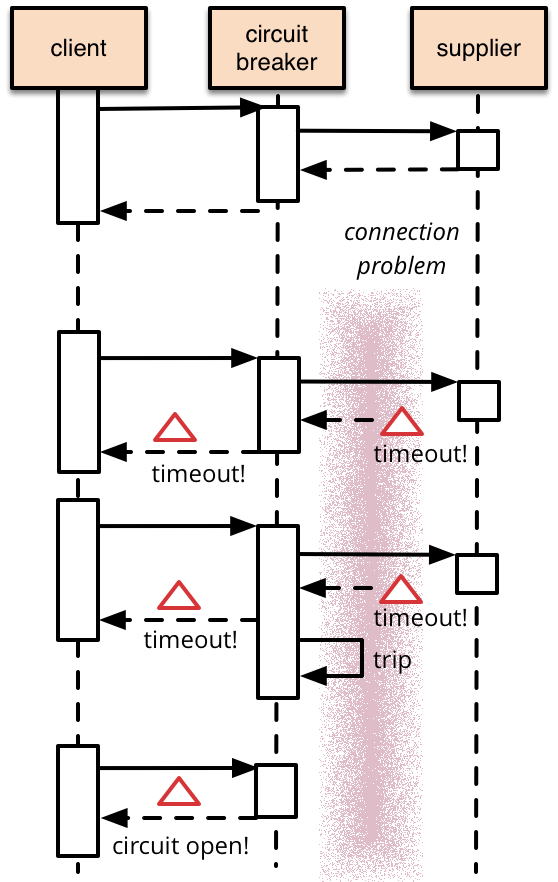
**Chained pattern:** Er wordt synchroon communicatie gelegd tussen verschillende microservices. Een voorbeeld zou het productorder process zijn.



**Asynchronous messaging pattern:** naast asynchroon communicatie met REST is het soms ook nodig om asynchrone communicatie te hebben. Het gebruik van bijvoorbeeld JMS kan zorgen dat berichten zeker aankomen ook als er geen ontvanger aanwezig is in het begin van het proces. Ook websockets zijn veelgebruikte communicatiemiddelen tussen microservices. 

**Circuit breaker pattern:** <https://martinfowler.com/bliki/CircuitBreaker.html>

Een manier om fouten op de juiste plaats op te merken. Als er een functie van communiceert van applicatie A -> B -> C en er gebeurt een fout bij applicatie B dan zal normaal gezien de fout bij A iets non gerelateerd zijn. Met dit patroon zet je een tussenobject tussen de call die de juiste error weergeeft. Dit patroon gaat vaak gepaard met een monitor systeem die de fouten juist weergeeft. De afbeelding maakt het duidelijk:



**Micro frontend**: normaal gesproken is de frontend ook in een microservice architectuur 1 geheel. Er zijn verschillende oplossingen om ook de frontend op te splitsen in micro frontends. Hiermee kun je bijvoorbeeld meerdere frontend technologieën zoals vue, angular of react in 1 website gebruiken. Hoewel het mogelijk is om meerdere frontenden te combineren is het niet de gewenste situatie. Net zoals je liefst alle microservices in 1 taal hebt wil je ook je frontend met dezelfde technologie bouwen. Verder komen er vaak problemen met conflecterende libraries en tragere website.

|  |  |
| --- | --- |
| https://cdn-images-1.medium.com/max/1000/1*l0sfQzUKdm8572t3Uwojwg.png  Figuur 1: standaard frontend architectuur | https://cdn-images-1.medium.com/max/1000/1*nQcL0KLROHsefxXp5wsT9Q.png  Figuur 2: micro frontend architectuur |

Voorbeeld: <https://single-spa.surge.sh/>

## Toepassingen

Er is gezocht naar een business case die voldoende inhoud bevat om het gebruik van microservices te onderbouwen.

De voornaamste praktische toepassing is bij grote bedrijven die problemen hebben hun **code base overzichtelijk te houden**. Door hun monolithische applicatie op te splitsen en domein experts voor elke microservice op te zetten verhoogd de kwaliteit en het overzicht van hun programma. Op deze manier kunnen per microservice kleine teams gezet worden in plaats van een grote groep die aan 1 applicatie werkt. Deze toepassing staat gelijk met Jeff Bezos’s “two pizza rule”— wat inhoud dat een softwareteam, dat niet gevoed kan worden met twee pizza’s, te groot is.

Een tweede case is bij applicaties die op momenten veel **piekmomenten** hebben, zoals webshops (solden) of chatapplicaties (nieuwjaar). Zij gebruiken microservices om bepaalde functies uit te schakelen of om meer capaciteit in te bouwen bij specifieke services. Zo merkt de gebruiker weinig verschil van het verhoogde verkeer en bespaart het bedrijf op kosten.

Een derde non-functional is als het **gebruik van 3e** partij applicaties nodig is. McDonalds heeft zo een integratie met UberEats. McDonalds wil niet dat hun applicatie een grote fout geeft als de koppeling tussen beide breekt. Daarom kunnen ze, dankzij hun microservice architectuur, met een druk op de knop hun link met UberEats uitschakelen zonder dat ze hun applicatie moeten aanpassen.

De **snelheid** waarop een functie op de markt gebracht wordt groeit ook. Bij monolitische applicaties wordt het naargelang de grote steeds moeilijker om monolithische applicaties uit te breiden omdat alles telkens opnieuw getest moet worden. Bij microservices moet enkel de koppeling en de service uitgebreid getest worden.

Geen stuk hardware zal 100% van de tijd werken en er zal altijd iets kapot gaan. Met de microservice architectuur kun je zeker zijn dat **ook als er een deel breekt, de andere nog kunnen werken**. Het goedgunstig opvangen van dit soort problemen is een groot voordeel voor toepassing zoals rekeningrijden waar een storing van enkele minuten al miljoenen euro’s kan kosten.

## Conclusie

Microservices zijn niet de oplossing voor alle applicaties. Als er geen probleem is met de schaalbaarheid of een te grote codebase is het niet verstandig om een monolithische applicatie om te bouwen naar microservices.

Het microservice architectuur geeft de mogelijkheid tot eenvoudigere parallelle development en integratie (CD/CI) en maakt het overzichtelijk om op applicatie niveau te testen. Het is altijd wel een goed idee om te kijken naar de (non-)functionele en om te kijken of een microservice architectuur van toepassing is.

## Bronnen

<https://dzone.com/articles/design-patterns-for-microservices>

<https://www.youtube.com/watch?v=-8FK9p_lLy0> AWS Case study McDonalds

<https://www.youtube.com/watch?v=vvhC64hQZMk> Architecture Whatsapp

<https://www.reddit.com/r/webdev/>

<https://dzone.com/articles/microservice-architecture-with-spring-cloud-and-do>

<https://blog.techcello.com/2016/04/top-10-critical-nfr-for-saas-applications-part-1/>

<https://microservices.io/patterns/microservices.html>

<http://microservices.io/>

<https://www.nginx.com/blog/introduction-to-microservices/>

<https://developer.ibm.com/wasdev/docs/writing-simple-microprofile-application/>

<https://dzone.com/articles/microservice-architecture-with-spring-cloud-and-do>

<http://blog.arungupta.me/microservice-design-patterns/>

<https://techlab.bol.com/swaps-from-monolith-to-small-web-applications/>

<https://monzo.com/blog/2016/09/19/building-a-modern-bank-backend/>

<https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116>

<https://www.businessinsider.nl/nederlandse-bedrijven-storten-zich-op-de-nieuwste-techtrend-microservices-maar-wat-is-het-en-wat-kun-je-ermee-davinci/>

# Casus – Animal Shelter

## Omschrijving

Animal Shelter™ dient een platform te zijn voor het vinden en opvangen van huisdieren. Zo kunnen honden geadopteerd worden, katten met een foto geïdentificeerd worden, een lijst met opvangers opgevraagd worden en voor geïnteresseerde kan er altijd naar ondergebrachte huisdieren gebrowset worden.

Deze opzet focust meer op het skelet van de applicatie en om non functionele requirements aan te tonen.

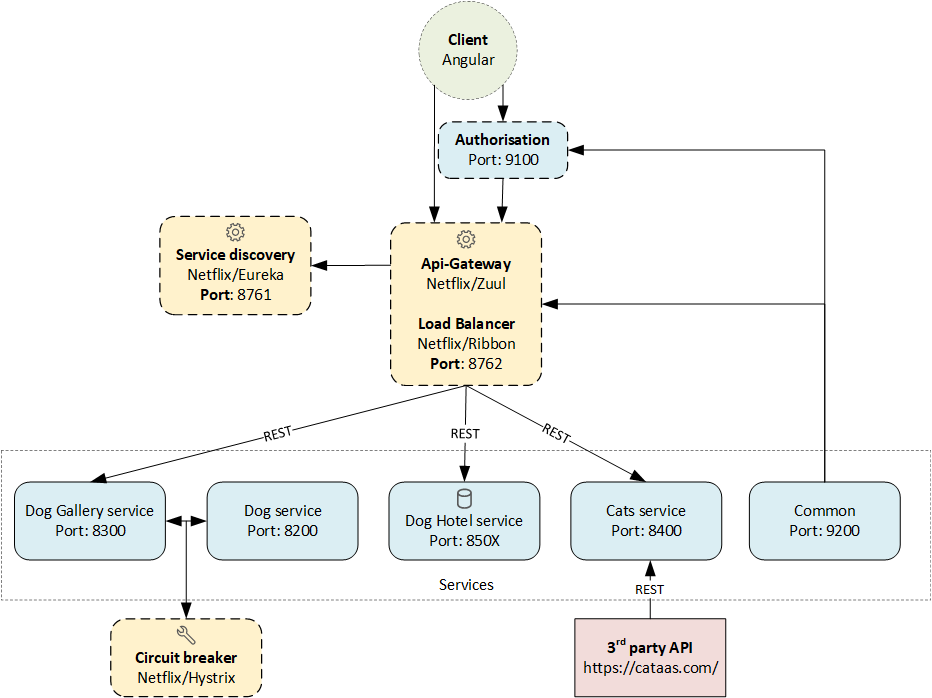
## Requirements

* **Honden:** hondenlijst ophalen
* **Honden:** informatie (NAW, foto, prijs) van een hond inzien
* **Honden hotel**: Moet tonen met welke server de gebruiker communicatie heeft.
* **Katten:** Er moeten willekeurige katten via een externe API opgehaald kunnen worden.
* **Autorisatie:** Een gebruiker moet kunnen inloggen (registreren niet vereist)

## Non-functionele requirements

* **Beschikbaarheid:** Als 1 service breekt moet de rest nog kunnen werken
* **Schaalbaarheid:** Het systeem moet eenvoudig uit te breiden zijn voor andere en functionaliteiten
* **Schaalbaarheid:** Er is een ontwerp die duidelijk het applicatielandschap schetst
* **Schaalbaarheid:** De applicatie is opgezet aan de hand van de SOLID principes
* **Schaalbaarheid:** Bij veel verkeer is het eenvoudig om flexibel op te schalen
* **Beveiliging:** De communicatie tussen systemen verloopt op een beveiligde lijn
* **Beveiliging:** Bepaalde functies zijn afgeschermd voor normale gebruikers
* **Beveiliging:** API van 3e moet afgeschermd zijn van de rest van de applicatie
* **Monitoring:** Het is mogelijk de systemen te monitoren
* **Logging**: Fouten moeten zo duidelijk mogelijk zijn en bij de juiste service aangeroepen worden.

## Architectuur



Figuur 3: Architectuur animal shelter

**Dog Gallery service** en **Dog service** hebben connectie met elkaar. De gallery service kan zo het dog object en lijsten query’s ophalen uit de dog service. Deze wordt via de zuul gateway aan de gebruiker getoond.

**Dog Hotel service** zorgt voor het vinden van dierenhuizen voor eigenaars die op vakantie zijn of die een achtergelaten huisdier gevonden hebben. Deze service is heel bekend geworden en draait dus op meerdere instanties. Met behulp van de load balancer wordt telkens de beste port gekozen tussen 8500 en 850X.

**Cats service** heeft een connectie met de API van cataas. Hiermee kunnen willekeurige katten plaatjes opgehaald worden. Indien nodig kan met een druk op de knop de link uitgezet worden – in geval dat de externe API wijzigt.

**Hystrix**

Hystrix is een implementatie van het circuit breaker pattern. Dit zorgt ervoor dat als er een fout optreedt, er in de juiste applicatie een oplossing voor gevonden kan worden.

Aanzetten gebeurt met de @EnableCirctuitBreaker annotatie in DogGalleryApplication.

Binnen de DogGalleryController kan een fout vanuit de Dog microservice komen. Met de Annotatie @HystrixCommand wordt er gekeken of dit voorkomt en naar de fallback methode gegaan.

**Eureka server**

De server wordt aangemaakt met de annotatie @EnableEurekaServer. Binnen de properties file worden ports doorgegeven en wordt het als server en niet als client geregistreerd.

Microservices worden elk met @EnableEurekaClient geannoteerd en in hun properties file wordt gelinkt naar de eureka server met de ingestelde port.

**Load balancing**

DogGallery maakt gebruik van RestTemplate wat door Eureka Server en Ribbon begrepen wordt. Eureka server zorgt ervoor dat het de service naam kan aanroepen (in plaats van localhost – zie DogGalleryController) en Ribbon zorgt automatisch voor load balancing.

**Zuul**

Zuul is de api gateway/proxy tussen de browser en de services. Iedere microservice draait op zijn eigen poort en wordt door eureka server herkend. Zuul is de entrypoint van alles. Het leest de properties uit en zorgt dat alle requests op slechts 1 port uitgevoerd moeten worden (zie postman ports – allemaal 8762).

Zuul zorgt voor load balancing (met ribbon).

Binnen zuul wordt ook de authenticatie en CORS behandeld.

**Authenticatie**

Authenticatie gebeurt met JWT tokens. In de Zuul gateway wordt gekeken naar:

1. Permissies (SecurityTokenConfig)
2. Token validator (JwtTokenAuthenticationFilter)
3. Fout afhandeling indien niet geautoriseerd waar nodig (JwtAuthenticationEntryPoint)

In de auth service wordt de gebruikers behandeld:

1. Bijhouden gebruikers (UserDetailsServiceImpl)
2. Inloggen (JwtUsernameAndPasswordAuthenticationFilter)

Het behandelen van de JWT token en zaken als gebruikersrollen (ADMIN en USER) moet in zowel de gateway als auth service gedaan worden. Hiervoor is een nieuwe “common” service opgezet wat de gedeelde code (JwtConfig en UserRole) bevat.

## Tools

Hieronder een lijst van gebruikte tools:

* Netflix Eureka: discovery server
* Netflix Ribbon: load balancing
* Netflix Hystrix: resilience module
* Netflix Zuul: HTTP routing between services
* Spring boot: Java framework for microservices
* Angular: frontend JavaScript framework

## Acceptatiecriteria

**Beschikbaarheid:** Als 1 service breekt moet de rest nog kunnen werken

*Oplossing:* Microservice architectuur

1. Start server en frontend
2. Navigeer naar dogs/login/cats pagina
3. API calls geven een foutmelding

**Schaalbaarheid:** Het systeem moet eenvoudig uit te breiden zijn voor andere en functionaliteiten

*Oplossing:* Microservice architectuur zorgt dat elk domein een eigen applicatie is. Experts op ieder domein kunnen zo werken aan hun eigen applicatie zonder dat andere delen breken.

**Schaalbaarheid:** Er is een ontwerp die duidelijk het applicatielandschap schetst

*Oplossing:* zie hoofdstuk “architectuur”.

**Schaalbaarheid:** De applicatie is opgezet aan de hand van de SOLID principes

*Oplossing:* Code is schaalbaar met SOLID principes opgezet. Zie bv. Common service voor duplicate code.

**Schaalbaarheid:** Bij veel verkeer is het eenvoudig om flexibel op te schalen

*Oplossing*: Ribbon en Eureka.

1. Draai server, zuul, doghotel, frontend
2. Wijzig de port in application.properties van de doghotel service en draai de nieuwe instantie
3. Refresh de dog hotel pagina enkele keer en je ziet dat verschillende porten reactie geven.

**Beveiliging:** De communicatie tussen systemen verloopt op een beveiligde wijze

*Oplossing:* JWT authenticatie, Spring security en BCrypt voor encryptie van wachtwoorden.

**Beveiliging:** Bepaalde functies zijn afgeschermd voor normale gebruikers

*Oplossing:* Met Spring security en user roles kunnen functies afgeschermd worden. Dit wordt gedefinieerd in de gateway (SecurityTokenConfig).

1. Start server, zuul, commen, auth, cat, frontend
2. Login met gebruikersnaam/paswoord: terry/12345
3. Navigeer naar de cats pagina en druk op de allow random cats knop -> unauthorized.
4. Log opnieuw in met gebruikersnaam/paswoord: admin/12345
5. Herhaal stap 3 -> succes.

**Beveiliging:** API van 3e moet afgeschermd zijn van de rest van de applicatie

*Oplossing:* Met het microservice patroon hoeft enkel het de cat service uitgezet worden.

1. Start server, zuul, cats
2. Doe een GET naar <http://localhost:8762/cat/randomCat>
3. Stop de cat service
4. Doe opnieuw dezelfde GET -> geen connectie met de random cat API.

**Monitoring:** Het is mogelijk de systemen te monitoren.

*Oplossing:* Eureka server voert automatisch health checks uit en laat zien welke systemen draaien.

**Logging:** Fouten moeten zo duidelijk mogelijk zijn en bij de juiste service aangeroepen worden.

*Oplossing*: Hystrix en het circuit breaker patroon.

1. Start server, zuul, doggallery
2. Doe een GET naar: <http://localhost:8762/dogGallery/1>

De call gaat naar de fallback methode van DogGalleryController.