SAD Semester 6

Naam: Rienk Engbrenghof

Studentnummer: 429059

Klas: S6-RB03T

# Introductie

In dit document zullen voornamelijk de technische keuzes omtrent mijn project gedocumenteerd staan. Hierbij moet worden gedacht aan een document met alle vereisten, keuzes en verschillende diagrammen die deze keuzes ondersteunen.

# Versiebeheer

|  |  |
| --- | --- |
| 21/2/2022 | Opzet gemaakt. |
| 23/2/2022 | Architectuurdocument toegevoegd. |
| 19/3/2022 | Aanpassingen aan stories. |
| 20/4/2022 | Feedback Marc toegepast. |
| 23/4/2022 | Scenario’s toevoegen. |
| 23/5/2022 | Scenario’s aangepast. |
| 8/6/2022 | Verder aanpassen scenario’s op basis leerdoel. |
| 15/6/2022 | Leerdoelen aangepast. |

# Repositories

Link naar repository: https://github.com/Pretter-S6

# Inhoudsopgave

[1. SAD 5](#_Toc106204706)

[1.1 Inleiding 5](#_Toc106204707)

[1.2 Userstories 5](#_Toc106204708)

[1.3 Usecases 6](#_Toc106204709)

[1.4 Non-functional requirements 9](#_Toc106204710)

[1.5 Diagrammen 10](#_Toc106204711)

[2. Voortgang 12](#_Toc106204712)

[2.1 Sprint 1 12](#_Toc106204713)

[2.2 Sprint 2 15](#_Toc106204714)

[2.3 Sprint 3 16](#_Toc106204715)

[3. Scenario’s 17](#_Toc106204716)

[3.1 Scalable Architectures 17](#_Toc106204717)

[3.1.1 Algemeen 17](#_Toc106204718)

[3.1.2 Toelichting 17](#_Toc106204719)

[3.2 Development and Operations 19](#_Toc106204720)

[3.2.1 Algemeen 19](#_Toc106204721)

[3.2.2 Toelichting 19](#_Toc106204722)

[3.3 Cloud services 21](#_Toc106204723)

[3.3.1 Algemeen 21](#_Toc106204724)

[3.3.2 Toelichting 21](#_Toc106204725)

[3.4 Security by design 23](#_Toc106204726)

[3.4.1 Algemeen 23](#_Toc106204727)

[3.4.2 Toelichting 23](#_Toc106204728)

[3.5 Distributed data 24](#_Toc106204729)

[3.5.1 Algemeen 24](#_Toc106204730)

[3.5.2 Toelichting 24](#_Toc106204731)

[4. Conclusie 25](#_Toc106204732)

# SAD

## 1.1 Inleiding

Het architectuurdocument wordt opgesteld om een duidelijk beeld te creëren over de applicatie. Hierin staan de keuzes gedocumenteerd. De eisen staan gedocumenteerd als userstories, deze worden vervolgens usecases.

De functionele en non-functionele eisen dienen allemaal uitgewerkt te worden. Op deze manier wordt de best mogelijke applicatie gemaakt.

## 1.2 Userstories

1. Als gebruiker wil ik kunnen inloggen, zodat iedereen een persoonlijk account heeft en geïdentificeerd kan worden.

2. Als gebruiker wil ik kunnen registeren, zodat ik een account kan aanmaken en hiermee kan inloggen.

3. Als gebruiker wil ik iemand kunnen volgen, zodat ik alle updates van deze persoon binnenkrijg op mijn feed.

4. Als gebruiker wil ik iemand kunnen ontvolgen, zodat ik zijn/haar updates niet meer krijg.

5. Als gebruiker wil ik mijn profiel kunnen beheren, zodat ik hierop aanpassingen kan doen en mijn informatie kan inzien.

6. Als gebruiker wil ik een post kunnen doen, zodat mensen die mij volgen kunnen zien wat ik post.

7. Als gebruiker wil iemand anders zijn profiel kunnen zien, zodat ik kan kiezen of ik deze persoon wil gaan volgen of niet.

8. Als gebruiker wil ik een foto toevoegen aan mijn post, zodat er een visuele representatie aan de post gekoppeld is.

9. Als gebruiker wil een post kunnen liken, zodat ik kan interacteren met posts van andere personen.

10. Als gebruiker wil ik kunnen reageren op een post van iemand anders, zodat ik mijn mening over de post kan achterlaten.

11. Als gebruiker wil ik kunnen uitloggen, zodat ik niet meer verbonden ben met mijn account.

12. Als gebruiker wil ik kunnen zoeken naar andere gebruikers, zodat ik hen kan gaan volgen.

## 1.3 Usecases

|  |  |
| --- | --- |
| Use case 01 | Inloggen |
| Actor(s) | Gebruiker |
| Aannames | Actor heeft al een account aangemaakt. |
| Basic flow | 1. Actor navigeert naar de login pagina. Voert zijn/haar emailadres en wachtwoord in. En klikt op inloggen. 2. Het systeem autoriseert de gebruiker en geeft bij de juiste combinatie toegang tot het systeem. Vervolgens wordt de actor naar de juiste pagina genavigeerd. |

|  |  |
| --- | --- |
| Use case 02 | Registreren |
| Actor(s) | Gebruiker |
| Aannames | Actor heeft nog geen account bij Pretter. |
| Basic flow | 1. Actor navigeert naar de register pagina. Voert zijn/haar gebruikersnaam, emailadres en wachtwoord (tweemaal) in. En klikt op registreer. 2. Het systeem controleert de gegevens. Wanneer deze correct zijn maakt het systeem het account aan. En wordt de actor naar de juiste pagina genavigeerd. |

|  |  |
| --- | --- |
| Use case 03 | Iemand volgen |
| Actor(s) | Gebruiker |
| Aannames | Actor volgt gebruiker nog niet. |
| Basic flow | 1. Actor navigeert naar iemand zijn pagina (Use case 07) en klikt op de knop om deze persoon te volgen. 2. Het systeem registreert deze handeling en zorgt ervoor dat de actor de gebruiker volgt. |

|  |  |
| --- | --- |
| Use case 04 | Ontvolgen |
| Actor(s) | Gebruiker |
| Aannames | Actor volgt gebruiker. |
| Basic flow | 1. Actor navigeert naar iemand zijn pagina (Use case 7) en klikt op de knop om deze persoon te ontvolgen. 2. Het systeem registreert deze handeling en zorgt ervoor dat de actor de gebruiker ontvolgt. |

|  |  |
| --- | --- |
| Use case 05 | Profiel beheren |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor navigeert naar zijn/haar eigen pagina. En kiest ervoor om 1 van zijn gegevens aan te passen. Bijvoorbeeld zijn gebruikersnaam. 2. Het systeem registreert de verandering, slaat deze op en voert deze door. |

|  |  |
| --- | --- |
| Use case 06 | Posten |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor navigeert naar de homepagina. Voert een tekst in voor de post die hij wil doen. En klikt vervolgens op: ‘plaatsen’. 2. Het systeem controleert de post, slaat deze op en voert het door voor de actor. Het systeem toont deze post aan de actor. |

|  |  |
| --- | --- |
| Use case 07 | Profiel inzien |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor navigeert naar iemand zijn pagina. Door middel van zoeken bijvoorbeeld (Use case 12) 2. Systeem laadt de pagina en toont deze aan de actor. |

|  |  |
| --- | --- |
| Use case 08 | Foto toevoegen |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor plaatst een post (Use case 06) en kiest ervoor om een foto te uploaden, gelinkt aan de post. 2. Het systeem slaat de foto en de post op. Vervolgens wordt de post getoond aan de actor. |

|  |  |
| --- | --- |
| Use case 09 | Post liken |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor ziet een post van een gebruiker die hij/zij volgt op zijn homepagina en klikt op ‘like’. 2. Het systeem slaat dit op en voert het door. |

|  |  |
| --- | --- |
| Use case 10 | Reageren op post |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor ziet een post van een gebruiker die hij/zij volgt op zijn homepagina en typt zijn reactie. Vervolgens klikt hij/zij op ‘reageer’. 2. Het systeem slaat dit op en voert het door. |

|  |  |
| --- | --- |
| Use case 11 | Uitloggen |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor navigeert naar zijn profielpagina en klikt hier op uitloggen. 2. Het systeem logt de gebruiker uit en stuurt de actor door naar de juiste pagina. |

|  |  |
| --- | --- |
| Use case 12 | Zoeken naar gebruikers |
| Actor(s) | Gebruiker |
| Aannames |  |
| Basic flow | 1. Actor bevindt zich op de homepagina en typt een naam van een gebruiker in de zoekbalk. Vervolgens klikt de actor op ‘zoek’. 2. Het systeem zoekt naar de naam en geeft de gevonden resultaten terug. 3. De actor klikt op een van de resultaten. 4. Het systeem stuurt de actor door naar het profiel waarop geklikt is. |

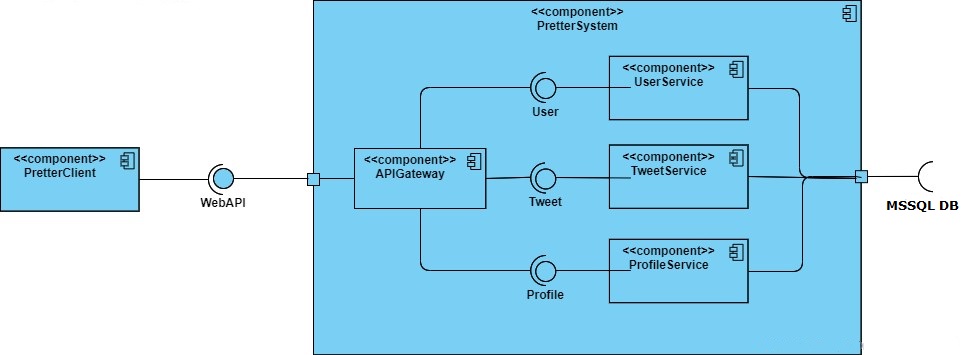
## 1.4 Non-functional requirements

|  |  |  |
| --- | --- | --- |
| NF 01 | Schaalbaarheid 1 | Tijdens het designen en opzetten van de architectuur wordt er rekening gehouden met de mogelijkheid om de applicatie makkelijk uit te kunnen breiden.  Hiervoor worden bepaalde keuzes gemaakt in de architectuur. |
| NF 02 | Schaalbaarheid 2 | De productie van de applicatie kan automatisch op en afschalen wanneer dit nodig is. Hier is rekening mee gehouden in het design. |
| NF 03 | Schaalbaarheid 3 | De productie kan omschakelen van instantie en kent meerdere instanties voor de services. |
| NF 04 | Performance | De snelheid van de applicatie is ook van belang. Deze wordt ook grotendeels gewaarborgd door een juiste architectuur. De applicatie dient te blijven werken onder zware load. Dit is in samenhang met schaalbaarheid. |
| NF 05 | Robuustheid | De applicatie gaat juist om met errors. Hij vangt deze af en zorgt ervoor dat het allemaal toegankelijk is voor de gebruiker.  Het is ook van belang dat binnen het systeem errors juist worden afgevangen. Zodat de uptime van het systeem gegarandeerd kan worden. |
| NF 06 | Gedistribueerde data | In het design en in de architectuur moet rekening worden gehouden met het juist opslaan van data. Hierbij moet je denken aan hoe data wordt opgeslagen en hoe deze wordt gedistribueerd. |
| NF 07 | Security | In het onderzoek wordt onderzocht welke risico’s het meest voorkomen bij soortgelijke applicaties. En vervolgens wordt er bij het design en de architectuur rekening gehouden met deze risico’s. Het systeem wordt zo geïmplementeerd dat deze risico’s beperkt worden. |
| NF 08 | Privacy | Er wordt rekening gehouden met de privacy regels omtrent het opslaan van data. |

## 1.5 Diagrammen

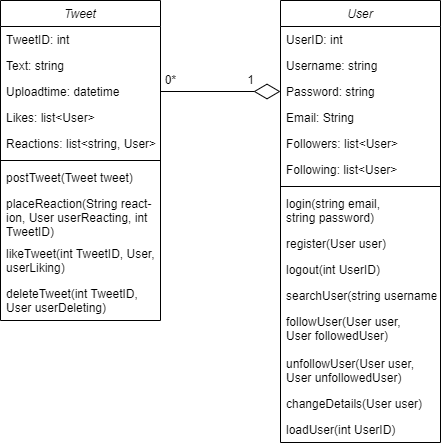
**Componentdiagram**

Dit diagram bevat een overzicht van de opdeling van de client (frontend) dat via de gateway de verschillende services aanroept. Deze services hebben vervolgens communicatie met de DB.



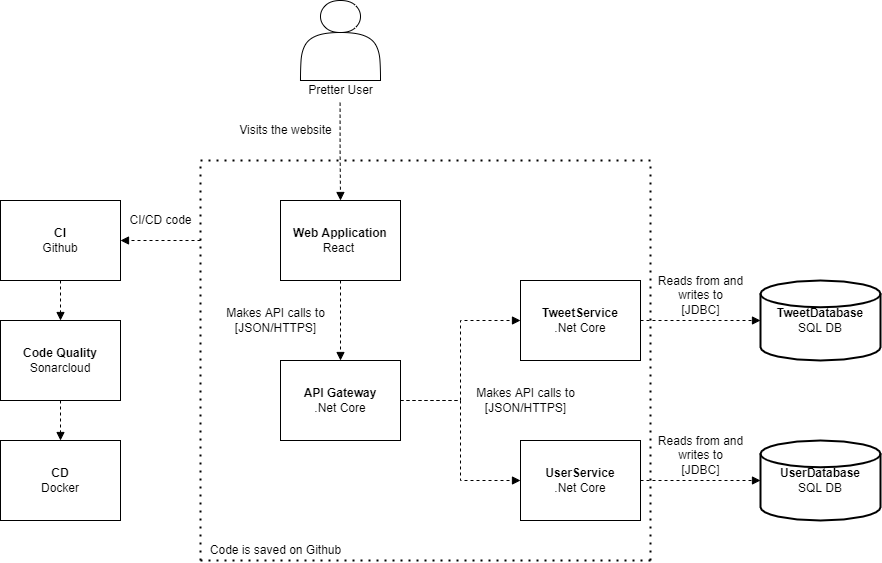
**ERD Diagram**

Dit diagram bevat een overzicht van hoe de modellen binnen de database en binnen de backend eruit gaan zien.



**Containerdiagram**

Dit diagram bevat alle componenten die zullen worden gerealiseerd voor mijn applicatie. Hierbij is alles binnen de gestippelde lijn geprogrammeerd door mij. Alles erbuiten zijn integraties.

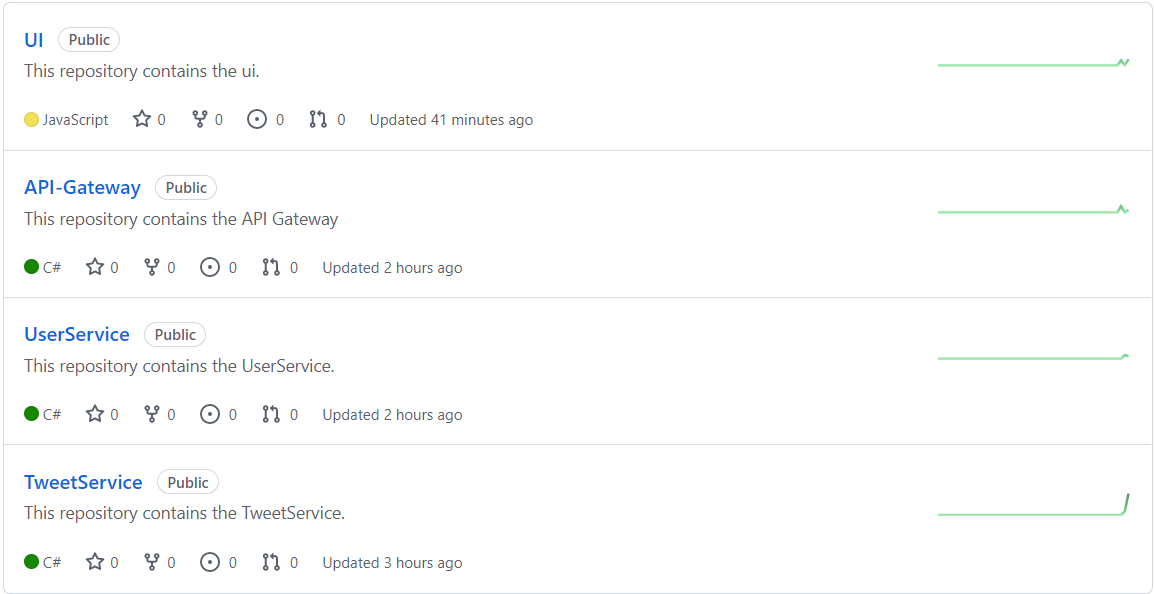
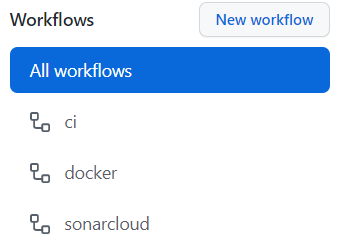


# Voortgang

## 2.1 Sprint 1

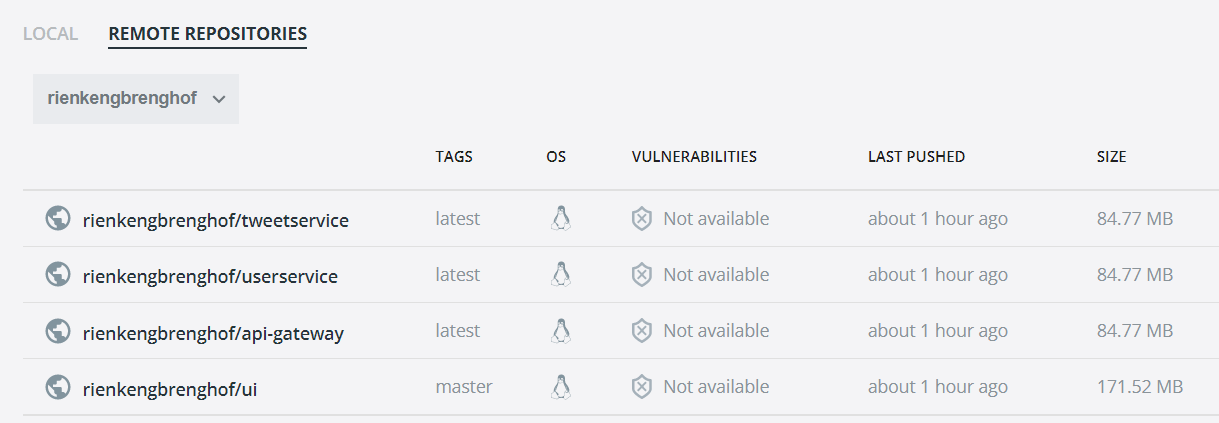
Binnen het groepsproject stond al vast welke technieken er worden gebruikt voor het behalen van de requirements. Hier is gekozen voor een VUE frontend, een API-gateway die gebruik maakt van het ocelot framework (.NET Core) en services die ook draaien op .NET Core. Deze keuzes staan al vast en zijn al goedgekeurd door de stakeholders.

Binnen mijn eigen project maak ik gebruik van een soortgelijke architectuur met dezelfde technieken. Hier gebruik ik zoals te zien in mijn architectuurdocument: react frontend, .NET Core gateway en .NET Core services. Op basis hiervan en het idee van mijn applicatie, heb ik verschillende eisen voor mijn applicatie opgesteld.

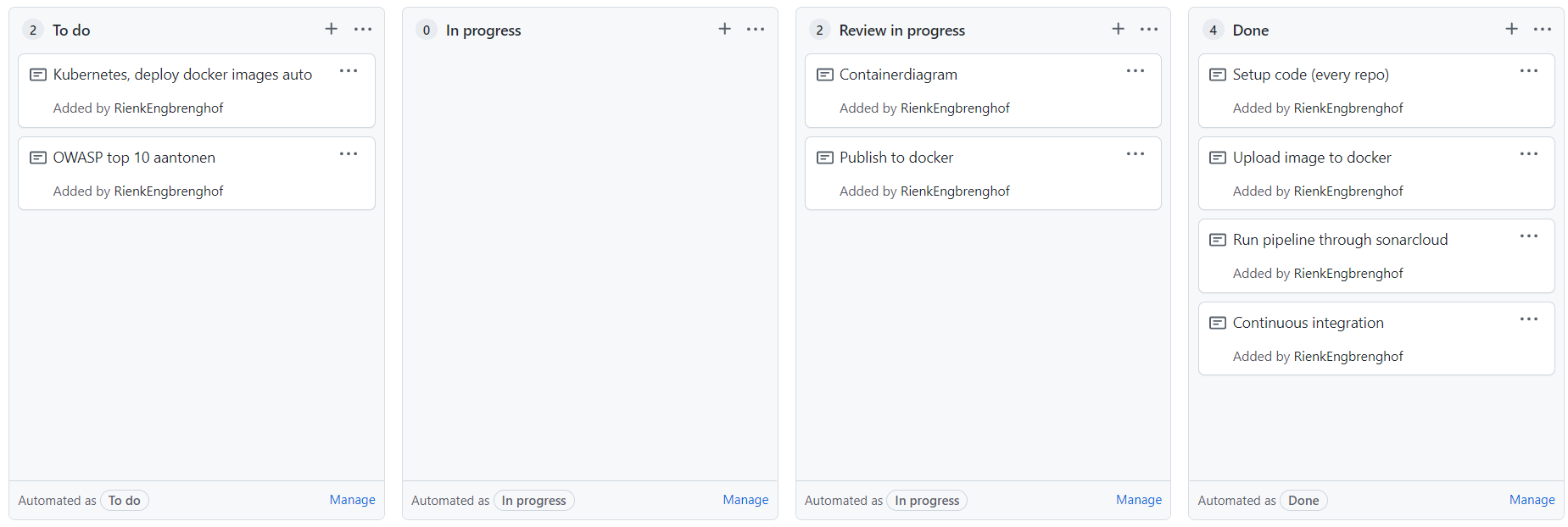


Bovenstaande foto laat mijn architectuur goed zien. Hier heeft elke service ook een toegewijde database. Elke repository bevat al een applicatie met code. En elke repository bevat CI, Code quality check en een image push naar Docker:

Hierbij heb ik mijn architectuur gedocumenteerd, aangepast wanneer nodig en is dit de initiële architectuur waartoe ik ben gekomen (walking skeleton).



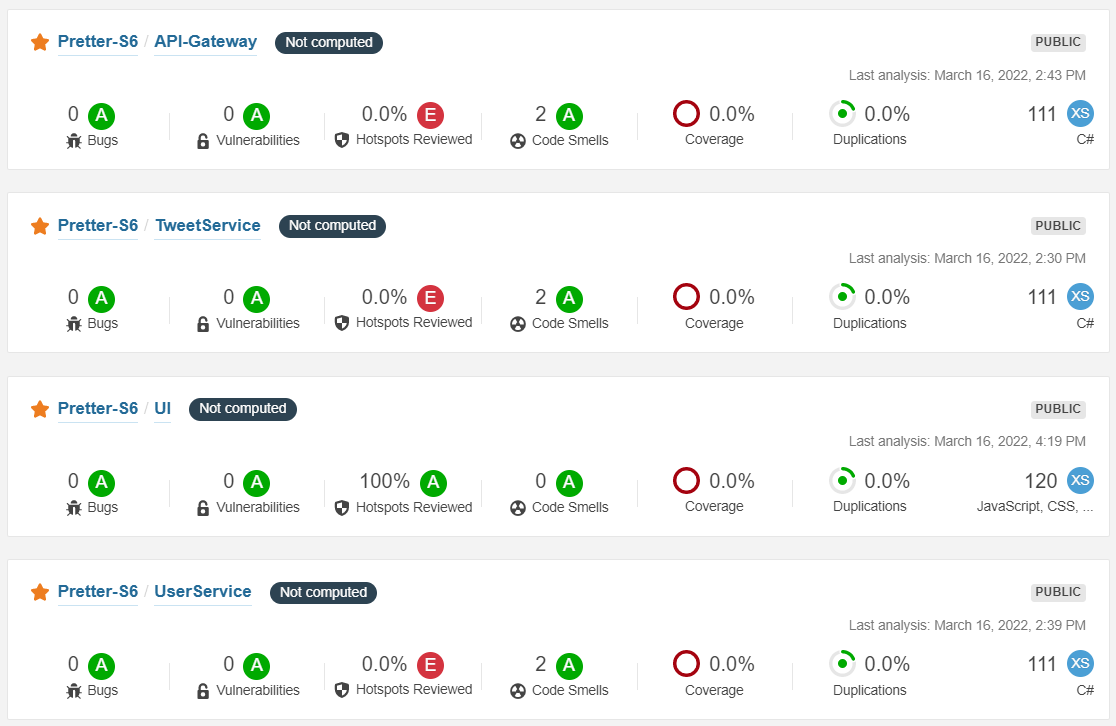
Heel mijn project is opgezet in Github, hierin hou ik ook bij welke taken nog moeten worden voltooid. Om het overzichtelijk te houden voor mezelf. Door middel van Github creëer ik voldoende flexibiliteit en onderhoudbaarheid.



The snelheid van de ontwikkeling binnen het project is afhankelijk van mijn individuele inzet en of de opzet juist is. Deze opzet (walking skeleton) bespreek ik dan ook uitvoerig met mijn docenten.

Omdat alles via CD online wordt gezet is het programma breed bruikbaar. En ook de architectuur en opzet van het project zorgen ervoor dat het een zeer bruikbaar programma wordt.

Zoals in 2.1 Enterprise Architectuur al is besproken, zijn alle repositories met code opgezet. Hierbij zijn de frontend, api-gateway en de microservices opgedeeld.



Als estimate voor total user base zou bij een beginnende situatie 1000 gebruikers een goede inschatting zijn. Vervolgens zouden hiervan ongeveer 100 een actieve gebruiker kunnen worden. Wanneer je 100 actieve gebruikers hebt dan zouden hiervan ongeveer 20 gelijktijdig de applicatie kunnen gebruiken. Hierbij zal de userservice minder worden gebruikt dan de tweetservice. De ui en api-gateway zullen het meeste worden gebruikt (highest load).

Hierbij is de keuze voor het opdelen van mijn architectuur dus gerechtvaardigd.

De gekozen technieken zijn eerder ook al verantwoord, ze zijn al breed in gebruik en zijn geschikt voor de Enterprise architectuur.

Load generation en het hierbij behouden van de juiste performance zijn erg van belang dit semester. Hiervoor maken we gebruik van kubernetes en docker. Die samen de load verdelen waar het nodig is. Er zijn ook verschillende tools die controleren of de performance op het juiste niveau blijft.

De non-functional requirements staan omschreven in hoofdstuk [1.3 Non-functional requirements](#_1.3_Non-functional_requirements).

Schaalbaarheid zal worden bereikt door de architectuur juist op te zetten. Dit voornamelijk door de applicatie op te delen in delen. Ook door het verspreiden van de load kan de applicatie makkelijk worden opgeschaald.

Performance zal voornamelijk worden gewaarborgd door het juist implementeren van technieken en het aanhouden van de juiste architectuur. Zoals gezegd is het ook mogelijk om tools te gebruiken om te controleren of de performance prima is.

Robuustheid kan ik bereiken door de applicatie veel te testen. Hierbij moet het afvangen van errors juist gaan en moet de applicatie up blijven tijdens onenigheden.

Gedistribueerde data zal worden bereikt door de data online, in delen, weg te schrijven.

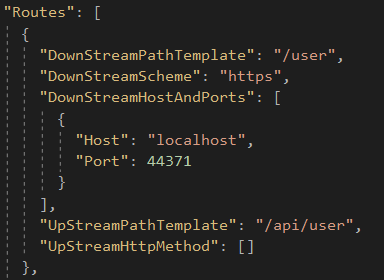
Security kan worden gewaarborgd door de juiste keuzes in de technieken en architectuur te maken. Ook kan er worden getest op verschillende zwaktes binnen een systeem.

Moet nog beginnen met het schrijven van (geautomatiseerde testen)

Sonarcloud is toegevoegd om de kwaliteit van het project te waarborgen. Omdat er nog niet veel code is geschreven is sonarcloud nog niet echt van toepassing. Wanneer er meer code staat dan zal sonarcloud aangeven wanneer er een zwakte waargenomen wordt.

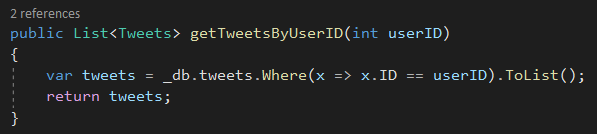
## 2.2 Sprint 2

Binnen sprint 2 ben ik bezig geweest met verscheidene dingen waaronder het aanpassen van de api-gateway naar een ocelot gateway. Hierbij wordt er geen gebruik meer gemaakt van de controller maar wordt alles geregeld via een JSON file:



Deze api-gateway werkt volledig en reroute alle incoming requests naar de juiste services.

Binnen de services ben ik bezig geweest met het aanpassen van het framework. Hier maak ik nu gebruik van het entity framework. Dat mij vooral helpt bij het ophalen en wegschrijven van data.



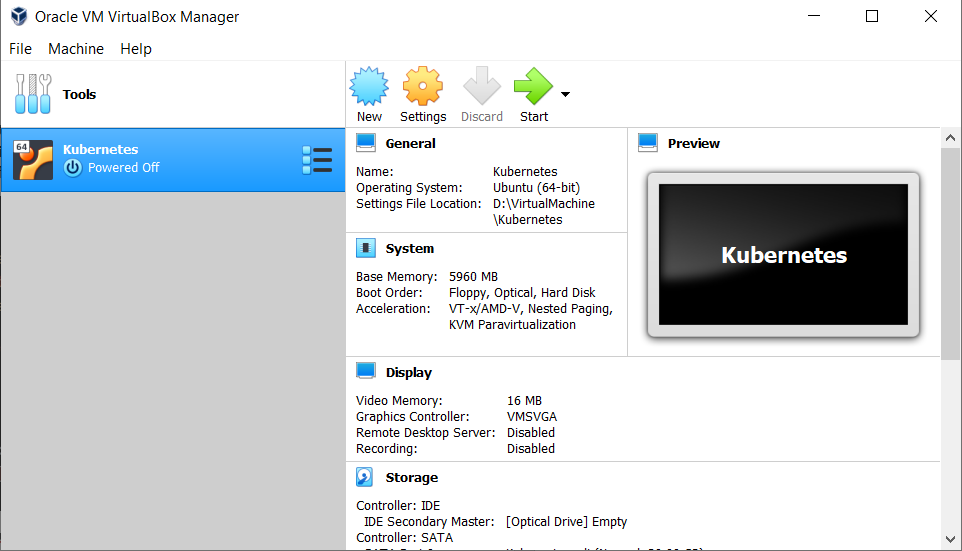
Dit entity framework werkt ook en ik ben nu vooral bezig met het implementeren van dit framework. Zodat mijn backend alle functionaliteiten bevat die ze nodig heeft.

Binnen mijn documentatie heb ik een testfase toegevoegd. Dit om de risico’s in kaart te brengen en ze waar nodig in te perken.

Deze sprint had ik graag meer werk verzet. Maar doordat ik verschillende aanpassingen binnen de structuur van mijn applicatie heb gedaan. Lijkt het alsof er minder werk is verzet. Echter zijn deze aanpassingen van belang voor de applicatie en de NFR’s van de applicatie.

## 2.3 Sprint 3

In deze sprint ben ik vooral bezig geweest met het opzetten van kubernetes. Ik heb het lokaal op een vm gezet en hierin al mijn services gehost. Hij pullt de images van docker en runt deze vervolgens op kubernetes.



Hiernaast heb ik alle documentatie bijgewerkt en online gezet in github.

Ook ben ik bezig geweest met het programmeren, zodat er nu een inlogscherm staat dat kan communiceren met de backend en de juiste methode. Hierdoor heb ik een integratie van frontend tot en met database.

Verder ben ik bezig geweest met het aanpassen van de docker images en de bijbehorende ip adressen voor de productie.

# Scenario’s

## 3.1 Scalable Architectures

### 3.1.1 Algemeen

Als developer, wil ik een schaalbare architectuur, zodat ik wanneer het nodig is kan upscalen, downscalen of kan switchen van instantie. Er zijn veel verschillende schaalbare architecturen. Zoals bijvoorbeeld: de N-Tier architectuur, waarin de applicatie is opgedeeld in 3 verschillende lagen: logic tier, presentation tier en de data tier. Een andere architectuur is de web-queue-worker architectuur. Hierin is de gebruiker het web en zit er een queue tussen de worker en het web, in de queue worden berichten opgeslagen zodat ze snel kunnen worden gebruikt. Daarnaast heb je ook nog het event-driven architectuur. Hierin heb je verschillende publishers, die een evenement aan de consumer doorgeven. Dit doen ze via een event manager (bijvoorbeeld een Kafka cluster). Tot slot heb je een microservice architectuur, waar binnen dit project voor is gekozen. Binnen het SAD staan de verschillende diagrammen die toelichten hoe deze architectuur in elkaar zit.

**Analyse en design**

* Binnen proftaak onderzoek gedaan naar de verschillende architectuur patterns die er zijn.
* Binnen het SAD zijn verschillende keuzes gemaakt die de schaalbaarheid van de applicatie en de architectuur waarborgen.

**Implementatie**

* Zoals in het SAD en de implementatie (Github) te zien, heb ik de microservice architectuur juist toegepast. Hierbij wordt schaalbaarheid gewaarborgd.
* Zoals in de Github repo van de API-Gateway te zien, heb ik Ocelot toegepast. Dit zorgt voor een (makkelijk) schaalbare API-Gateway.
* Zowel in Ocelot als in Kubernetes zit loadbalancing, hierbij heb ik op Kubernetes voor elke instantie 2 replicas aangemaakt. Hierdoor kan Kubernetes switchen wanneer dit nodig is. Kubernetes kent ook een dashboard waar je overzichtelijk kan zien wat de instanties doen.

**Scenario’s**

|  |  |
| --- | --- |
| **Scenario** | **Pretter** |
| Het verkeer naar de UserService wordt twee keer zo groot. | Kubernetes zal de load van de service balancen en wanneer nodig verkeer over een andere replica laten gaan. |
| De eerste replica van de TweetService is niet meer beschikbaar. | Kubernetes zal de load verschuiven naar de andere beschikbare replica. |

### 3.1.2 Toelichting

**Messaging**

Messaging kan erg handig zijn bij schaalbare architecturen. Wanneer er bijvoorbeeld een grote hoeveelheid aan verkeer langs een service gaat. Dan kan het van belang zijn dat je berichten worden opgeslagen. Dit kan zijn wanneer een service niet meer beschikbaar is. Of als er bijvoorbeeld moet worden geswitcht naar een andere instantie van de service.

RabbitMQ is een open source message broker die berichten kan managen. Het kan messages naar verschillende services tegelijk sturen, messages asynchroon versturen en eventueel ook tijdelijk opslaan.

**Keuzes**

Er is gekozen voor een microservice architectuur. Waar alle services, de API-Gateway en de frontend apart worden geployed als containers. Deze containers worden vervolgens in Kubernetes gebruikt. Er bestaan van elke container meerdere instanties, zodat er geschakeld kan worden. Kubernetes regelt ook de load balancing hier. En dit is zichtbaar op het dashboard.

## 3.2 Development and Operations

## 3.2.1 Algemeen

Als developer, wil ik een goede DevOps omgeving, zodat ik het agile en automatiseringsproces kan ondersteunen. En om een zo kortmogelijke release tijd en hoge software kwaliteit te garanderen.

**Analyse en design**

* Het proces is ingericht volgens OTAP (ontwikkeling, test, acceptatie en automatisering). Hierbij wordt eerst de code ontwikkeld, vervolgens geautomatiseerd getest en in geval van acceptatie gedeployed.

**Implementatie**

* De DevOps omgeving is opgezet in GitHub. Hier wordt de code en de taken opgeslagen. Ook de CI/CD pipeline bevindt zich in GitHub. De pipeline test eerst de code, vervolgens controleer SonarQube de kwaliteit en eventuele bugs. En hierna wordt de code eventueel gepusht naar Docker.
* Alle delen van de applicatie zijn individueel te deployen. Ze worden ook allemaal individueel als Docker images gepusht. In de virtuele omgeving van Kubernetes worden deze images automatisch gepulled.

**Scenario’s**

|  |  |
| --- | --- |
| **Scenario** | **Pretter** |
| Er is een visuele aanpassing gedaan op de frontend. | Wanneer de push wordt ontvangen. Zullen allereerst de test worden gerund, vervolgens zal sonarcloud de code scannen en tot slot zal bij goedkeuring er een docker image gebouwd worden. Wanneer dit gebeurd dan pullt Kubernetes deze en heeft de deployment de nieuwste versie. |
| Een test van de frontend push faalt. | De gefaalde test zal ervoor zorgen dat de pipeline faalt, hierdoor zal er eerst moeten worden voldaan aan deze test. |

## 3.2.2 Toelichting

**Management types**

Change management, change management gaat over de aanpak wanneer er een transitie of transformatie plaatsvind qua een organisatie haar doelen, processen of technologieën. Het doel van change management is om veranderingen effectief en gecontroleerd te laten verlopen. Ook helpt het personeel bij het accepteren van veranderingen. Agile support deze change omgeving. Omdat changes kunnen worden gescheduled als stories of features.

Risk management, risk management gaat over het identificeren, analyseren en controleren van risico’s voor een organisatie of systeem. Deze risico’s kunnen van alles zijn, bijvoorbeeld financiële risico’s, technologische risico’s, strategische risico’s of bijvoorbeeld ongelukken of rampen. Hierbij is het van belang om constant te monitoren, zodat je snel kan ingrijpen wanneer er een risico plaatsvind.

Security management, dit is een proces die moet zorgen dat er meer grip wordt verkregen op veiligheidsrisico’s. Het is een systematische kijk op de beveiliging van applicaties/organisaties. Hierin zit risk management ook verwerkt. Er wordt een risico analyse uitgevoerd, vervolgens vindt risk management plaats. Daarna worden er sancties en policies opgesteld en vervolgens worden het systeem geaudit.

Release management, release management is het managen en het toewerken naar een release. Het begint met plannen, bouwen, reviewen, testen. Vervolgens wordt het gedeployed, wanneer alles goedgekeurd is. Deze management vorm omvat alle stappen van development tot deployment.

Verwachtingsmanagement, hierbij doe je een poging om de kloof te dichten tussen wat een ander verwacht en wat er geleverd kan worden. Dus je stelt eisen aan de verwachtingen en koppelt dit terug aan belanghebbenden. Hiermee kan je belanghebbenden meer tevreden houden.

**Testen**

Er zijn verschillende soorten testen die de software quality kunnen waarborgen: Unit tests, Integration tests, UI tests, End-to-end tests en Acceptance tests zijn hier voorbeelden van. Binnen een applicatie is het van belang om zo veel mogelijk diversiteit qua testen toe te voegen. Door beknopte tijd binnen het project ben ik tot dusver alleen tot unittesten gekomen. Het is echter van belang om verder te kijken.

Integration tests gaan 1 stap verder dan Unit testen, binnen mijn applicatie zouden deze mogelijk zijn met bijvoorbeeld Enzyme. Enzyme rendert components in memory en hierdoor is het mogelijk om meer te testen dan alleen 1 klasse.

UI testen kan binnen react met react zelf, het biedt de mogelijkheid om de UI te testen. Voornamelijk of dat de UI gereageerd zoals beloofd.

End to end testen kan binnen een applicatie zoals de mijne met bijvoorbeeld playwright. Hierbij wordt verwacht dat de hele applicatie testbaar is. Hiervoor moet dus de hele applicatie gebouwd worden. Vervolgens kan er worden gekeken of de applicatie handelt zoals hij moet handelen.

Tot slot kan de acceptance test worden uitgevoerd, hierin controleer je of het systeem alles doet waar hij voor ontworpen is. Dit kan ook worden gezien als het aftekenen van de use-cases en test-cases.

## 3.3 Cloud services

### 3.3.1 Algemeen

Als developer, wil ik services in de cloud hosten, zodat ik de voordelen van cloudservices (schaalbaarheid, eenvoudigheid, betrouwbaarheid en veiligheid) kan benutten.

**Analyse en design**

* In het SAD zijn verschillende keuzes gemaakt over waarom sommige onderdelen wel of niet in de cloud gerund worden. Zo kon sonarcloud bijvoorbeeld ook lokaal gerund worden.

**Implementatie**

* Voor het controleren van de kwaliteit van de code wordt gebruik gemaakt van de sonarcloud service. Hierbij ontvangt de pipeline een report en indien er iets niet correct is wordt de pipeline gecanceld.
* Voor development wordt gebruik gemaakt van een online database. Alle images worden gedeployed naar Kubernetes.
* Er wordt ook gebruik gemaakt van Docker hub, hierbij worden mijn Docker images naar de cloudservice van Docker gepusht. Vervolgens zijn ze online beschikbaar voor Kubernetes.

**Scenario’s**

|  |  |
| --- | --- |
| **Scenario** | **Pretter** |
| Sonarqube heeft problemen door een windows update. | Mijn applicatie heeft hier geen last van omdat het gebruik maakt van de cloud service. |
| Er is erg veel dataverkeer. | Door het gebruik van een online database is er voldoende dataverkeer mogelijk. |
| Het login endpoint wordt erg veel gebruikt. | De applicatie laat het verkeer over de verschillende instanties gaan, en schaalt het CPU gebruik op. |

### 3.3.2 Toelichting

**Voordelen cloud services**

Het naar de cloud brengen van je services brengt voordelen met zich mee, voorbeelden hiervan zijn: het vermindert je eigen IT kosten, omdat je het niet meer zelf hoeft te hosten. Schaalbaarheid, omdat het gehost wordt bij een externe, moeten zij up of downscalen wanneer het nodig is. Het overstappen op cloud services dwingt ook om schaalbaar op te bouwen.

Het is een veilige omgeving waarin alles wordt opgeslagen. Hierdoor hoef je zelf niet meer aan de slag met het opzetten van een veilige omgeving. En komt je service minder vaak in gevaar.

Automatische updates, een cloud provider zal automatisch updates verschaffen, dit helpt ook met de veiligheid van jouw applicatie.

**Nadelen cloud services**

Cloud services brengen ook een aantal nadelen met zich mee, zoals dat je bijvoorbeeld afhankelijk wordt van de connectie naar de cloud services toe. Ook wordt je in het algemeen afhankelijk van de cloud provider. Zo kan er bijvoorbeeld daar een risico plaatsvinden waardoor je service niet meer beschikbaar is.

Een laatste nadeel zou kunnen zijn dat je kiest voor de verkeerde cloud provider. Waardoor je vast kan komen te zitten aan een langdradig contract. Of waardoor je service een langere periode niet meer beschikbaar is.

**Mogelijkheden**

Er zijn verschillende mogelijkheden om mijn services naar de cloud te deployen. Onder dit kopje zullen de verschillende cloudservices worden besproken.

**AWS**

|  |  |
| --- | --- |
| Is integreerbaar met mijn applicatie (ASP.NET, REACT en MSSQL). | JA |
| Staat bekend als veilig. | JA |
| Goede schaalbaarheid. | JA |
| Goede keuze in geval van 100.000 gebruikers. | NEE |
| Prijs in geval van 100.000 gebruikers. | +-10k |

Beschikbare monitoring tool voor AWS

AWS heeft een ingebouwde monitoring tool die CloudWatch heet. Het programma haalt metrics op om op basis daarvan statistiek te genereren. Hierdoor kan je verschillende patronen herkennen. Het is ook mogelijk om alarmen te zetten die services stoppen of starten.

**Interserver**

|  |  |
| --- | --- |
| Is integreerbaar met mijn applicatie (ASP.NET, REACT en MSSQL). | JA |
| Staat bekend als veilig. | JA |
| Goede schaalbaarheid. | JA |
| Goede keuze in geval van 100.000 gebruikers. | NEE |
| Prijs in geval van 100.000 gebruikers. | +-3k |

Beschikbare monitoring tool voor Interserver

Binnen Interserver zijn verschillende mogelijkheden. Een van deze mogelijkheden is Nagios. Dit is een extern programma dat de cloud kan monitoren in een dashboard. Nagios is ook mogelijk om op te zetten binnen mijn productie omgeving.

**Microsoft Azure**

|  |  |
| --- | --- |
| Is integreerbaar met mijn applicatie (ASP.NET, REACT en MSSQL). | JA |
| Staat bekend als veilig. | JA |
| Goede schaalbaarheid. | JA |
| Goede keuze in geval van 100.000 gebruikers. | JA |
| Prijs in geval van 100.000 gebruikers. | +-6k |

Beschikbare monitoring tool voor Azure

De monitoring tool binnen Azure heet Azure Monitor. Van de bovengenoemde monitoring tools is dit de meest uitgebreide tool. Je kan van alle services de resources zien, hoe het verkeer verloopt en zien waar het verkeer vandaan komt. Deze tool zorgt ervoor dat er een geheel nieuwe wereld opengaat, je krijgt toegang tot nieuwe functionaliteiten als: statistische inzichten, visualisaties in diagrammen, analyseren van logs, reageren op alarmen en het integreren van andere applicaties.

**Messaging**

Binnen al deze applicaties is het mogelijk om messaging via RabbitMQ op te zetten, deze service zou dan ook moeten worden gehost. Er zijn geen extra kosten voor het gebruik van RabbitMQ in een omgeving met 100.000 gebruikers, zolang je het dan op je eigen services host.

## 3.4 Security by design

### 3.4.1 Algemeen

Als developer, wil ik security implementeren vanaf het design stadium. Zodat ik eventuele beveiligingsrisico’s kan voorkomen en niet op het einde nog alles voor security hoef te doen.

**Analyse en design**

* Gebruikte frameworks geanalyseerd en hiervan de best practices gebruikt.
* OWASP implementeren.
* Testplan opgezet in documentatie en automatisch testen toegevoegd in de pipeline. Ook test sonarcloud op eventuele veiligheidsgebreken.

**Implementatie**

* In het juist opzetten van de architectuur heb ik voor veiligheidsredenen en best practices ervoor gekozen om de API-Gateway als enige punt van aanspraak te houden. Hierdoor is de attack surface een stuk kleiner.
* In de applicatie is authenticatie toegevoegd. Hierdoor kunnen onbevoegden niet alles doen.
* Binnen de applicatie worden errors juist afgevangen. Hierbij wordt er gezorgd dat er veilig gefaald kan worden.
* Sonarcloud toont beveiligingsrisico’s.
* Wachtwoorden zijn encrypted.

**Scenario’s**

|  |  |
| --- | --- |
| **Scenario** | **Pretter** |
| Er wordt van buitenaf een endpoint van de UserService aangesproken. | Endpoints van de services zijn alleen beschikbaar en bereikbaar voor de API-Gateway. |
| Een methode krijgt een error omdat er geen verbinding met de database is. | Errors worden afgevangen, de applicatie zal de data niet kunnen ophalen maar zal niet crashen. |

### 3.4.2 Toelichting

**GDPR**

Hoe lang een bedrijf persoonsgegevens mag opslaan op basis van de AVG, hangt af van waarvoor de gegevens gebruikt worden. Wel is het zo dat wanneer de data niet meer gebruikt wordt, het na een bepaalde tijd verwijderd dient te worden.

Bij een datalek waar het om persoonsgegevens gaat, is het belangrijk dat risk management in acht wordt gehouden. Een eventuele lek zou kunnen zorgen voor enorme boetes voor een bedrijf. Daarom is het van belang dat de data veilig wordt opgeslagen, idealiter worden er zo min mogelijk persoonsgegevens opgeslagen.

Binnen mijn applicatie zou ik dit kunnen verbeteren door OAuth2 of OpenID toe te voegen aan de applicatie. Dit zorgt ervoor dat gebruikers inloggen op een SSO van een andere instantie (bijvoorbeeld Google of Fontys). Hierdoor hoef ik zelf niet de persoonsgegevens op te slaan in mijn database. Dit ontneemt een hoop risico’s.

## 3.5 Distributed data

### 3.5.1 Algemeen

Als developer, wil ik mijn data distribueren, zodat ik de grote hoeveelheid aan data en de gevoeligheid ervan kan waarborgen. Ook de beschikbaarheid van de data is hierbij een belangrijke factor.

**Analyse en design**

* In het SAD is te zien dat er een opdeling van data plaatsvind. Hierbij is ook overwogen of er gebruik moest worden gemaakt van meerdere databases. Binnen de productie is gebruik gemaakt van meerdere replica’s van de database.
* Analyse over de data regulaties waar normaliter aan voldaan moet worden gedaan.

**Implementatie**

* In productie is de data opgedeeld in een User en Tweet database. Hierbij worden in Kubernetes meerdere instanties van de database aangemaakt. Dit ook voor beschikbaarheid en schaalbaarheid.
* De data is encrypted en volgens de geanalyseerde richtlijnen opgeslagen.

**Scenario’s**

|  |  |
| --- | --- |
| **Scenario** | **Pretter** |
| Er gaat een grote hoeveelheid aan data naar de UserDB op de productie. | Dit zou geen probleem moeten zijn, aangezien er meerdere replica’s van de database zijn. Kubernetes kan verkeer hiernaar omzetten. |
| Er vindt een datalek plaats in de UserDB. | Wanneer alles volgens richtlijnen is ingericht, zal er tijdens een lek ook volgens de richtlijnen gehandeld moeten worden. Maar risico´s zijn al kleiner omdat de richtlijnen in acht zijn genomen. |

### 3.5.2 Toelichting

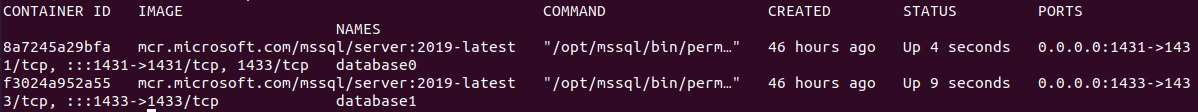
**Data syncing**

Met RabbitMQ is het mogelijk om data naar 2 databases tegelijk te sturen. Hierdoor zou je technisch gezien dezelfde data in beide databases krijgen. Vervolgens zou je deze data kunnen ophalen uit de database die het minst belast is.

Echter zijn er ook verschillende tools die ervoor zorgen dat de databases automatisch gesynchroniseerd worden. Zoals ApexSQL Data diff. Dit is een programma dat controleert op verschillen tussen databases. En ze vervolgens kan synchroniseren.

Op het internet staan ook verschillende oplossingen, die alleen SQL gebruiken. Hierbij wordt in de statement ook de andere database aangepast. Dit is echter een moeilijke implementatie aangezien wij gebruik maken van het entity framework.

Zoals onderstaande foto laat zien, is de data verdeeld over meerdere databases.



# Conclusie

Omdat er gebruik is gemaakt van een microservice architectuur, is de applicatie gedurende het hele semester goed schaalbaar geweest.

De applicatie werkt in zowel development als in productie. En in productie wordt de applicatie gehost in Kubernetes. In dit document zijn verschillende inzichten opgedaan over hoe de applicatie nog realistischer gehost zou kunnen worden.