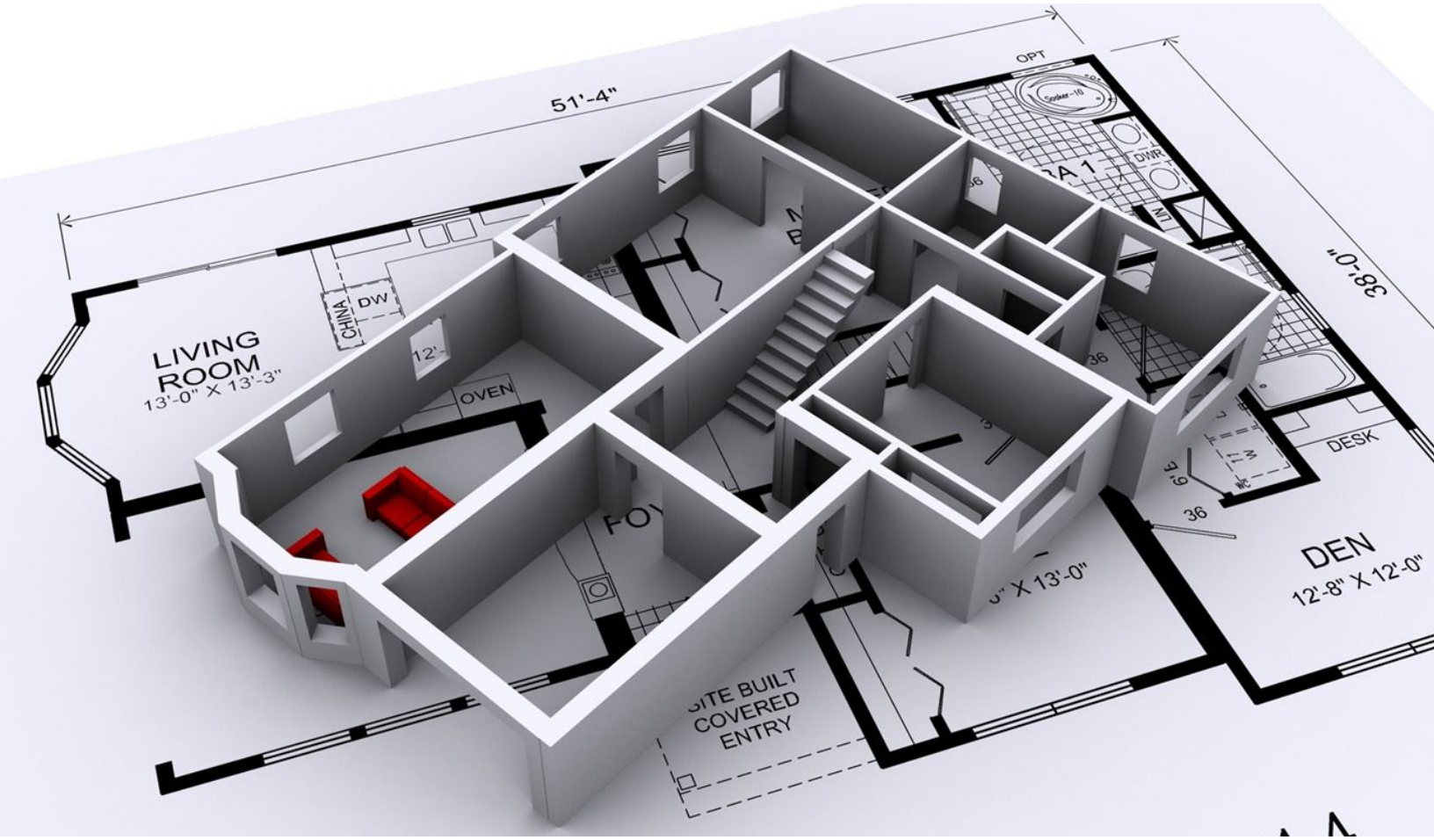
A&D



Gemaakt door: Gerbrecht Ghijsels
Email: Gerbrecht.Ghijsels@hva.nl
Studentnummer: 500736298
Datum: 01-01-2020
Versie: 3.0

Inhoudsopgave

Versies:

1.0	Eerste iteratie
2.0	Tweede iteratie
3.0	Documenteren van installatie processen

Figurenlijst

Code base

In via de onderstaande link is de code base te verkrijgen
https://github.com/GerbrechtGhijsels/IOTApplications/tree/Api/cardsudemy

Inleiding

Dit architecture and design document wordt uitgevoerd om te onderzoeken of de knmi weers dataset kan worden gebruikt om een weers api te ontwikkelen. Dit idee is ontstaan uit gebrek van een weers api die niet kunnen worden toegepast bij de case study.
In dit rapport staat vermeld hoe deze case study is opgebouwd met behulp van architecture and design keuzes.

Case Study

Voor mijn case study heb ik gekozen om een applicatie te maken die weers data van de KNMI en data van een weer API toont op verschillende manieren aan de gebruiker door middel van een API achtige Back-End. Deze backend moet dan beschikbaar zijn via een url door de gebruiker. De data komt deels uit een dataset van Kaggle: https://www.kaggle.com/sinaasappel/historical-weather-in-the-netherlands-19012018 en van de knmi website http://projects.knmi.nl/klimatologie/daggegevens/selectie.cgi waar een tweede dataset is gegenereerd om de dataset van kaggle aan te vullen met recente weers data. Hiernaast moet de gebruiker ook nog zijn eigen weers data kunnen uploaden die alleen toegankelijk is voor hem.

Dataset
De dataset bestaat uit de volgende modellen:

Measurements:
YYYYMMDD = Datum (YYYY=jaar MM=maand DD=dag);
DDVEC   = Vectorgemiddelde windrichting in graden (360=noord, 90=oost, 180=zuid, 270=west, 0=windstil/variabel). Zie http://www.knmi.nl/kennis-en-datacentrum/achtergrond/klimatologische-brochures-en-boeken;
FHVEC   = Vectorgemiddelde windsnelheid (in 0.1 m/s). Zie http://www.knmi.nl/kennis-en-datacentrum/achtergrond/klimatologische-brochures-en-boeken;
FG   = Etmaalgemiddelde windsnelheid (in 0.1 m/s);
FHX   = Hoogste uurgemiddelde windsnelheid (in 0.1 m/s);
FHXH   = Uurvak waarin FHX is gemeten;

FHN = Laagste uurgemiddelde windsnelheid (in 0.1 m/s);
FHNH = Uurvak waarin FHN is gemeten;
FXX = Hoogste windstoot (in 0.1 m/s);
FXXH = Uurvak waarin FXX is gemeten;
TG = Etmaalgemiddelde temperatuur (in 0.1 graden Celsius);
TN = Minimum temperatuur (in 0.1 graden Celsius);
TNH = Uurvak waarin TN is gemeten;
TX = Maximum temperatuur (in 0.1 graden Celsius);
TXH = Uurvak waarin TX is gemeten;
T10N = Minimum temperatuur op 10 cm hoogte (in 0.1 graden Celsius);
T10NH = 6-uurs tijdvak waarin T10N is gemeten; 6=0-6 UT, 12=6-12 UT, 18=12-18 UT, 24=18-24 UT
SQ = Zonneschijnduur (in 0.1 uur) berekend uit de globale straling (-1 voor <0.05 uur);
SP = Percentage van de langst mogelijke zonneschijnduur;
Q = Globale straling (in J/cm2);
DR = Duur van de neerslag (in 0.1 uur);
RH = Etmaalsom van de neerslag (in 0.1 mm) (-1 voor <0.05 mm);
RHX = Hoogste uursom van de neerslag (in 0.1 mm) (-1 voor <0.05 mm);
RHXH = Uurvak waarin RHX is gemeten;
PG = Etmaalgemiddelde luchtdruk herleid tot zeeniveau (in 0.1 hPa) berekend uit 24 uurwaarden;
PX = Hoogste uurwaarde van de luchtdruk herleid tot zeeniveau (in 0.1 hPa);
PXH = Uurvak waarin PX is gemeten;
PN = Laagste uurwaarde van de luchtdruk herleid tot zeeniveau (in 0.1 hPa);
PNH = Uurvak waarin PN is gemeten;
VVN = Minimum opgetreden zicht; 0: <100 m, 1:100-200 m, 2:200-300 m,..., 49:4900-5000 m, 50:5-6 km, 56:6-7 km, 57:7-8 km,..., 79:29-30 km, 80:30-35 km, 81:35-40 km,..., 89: >70 km)
VVNH = Uurvak waarin VVN is gemeten;
VVX = Maximum opgetreden zicht; 0: <100 m, 1:100-200 m, 2:200-300 m,..., 49:4900-5000 m, 50:5-6 km, 56:6-7 km, 57:7-8 km,..., 79:29-30 km, 80:30-35 km, 81:35-40 km,..., 89: >70 km)
VVXH = Uurvak waarin VVX is gemeten;
NG = Etmaalgemiddelde bewolking (bedekkingsgraad van de bovenlucht in achtsten, 9=bovenlucht onzichtbaar);
UG = Etmaalgemiddelde relatieve vochtigheid (in procenten);
UX = Maximale relatieve vochtigheid (in procenten);
UXH = Uurvak waarin UX is gemeten;
UN = Minimale relatieve vochtigheid (in procenten);
UNH = Uurvak waarin UN is gemeten;
EV24 = Referentiegewasverdamping (Makkink) (in 0.1 mm);

Stations:

stn = station nummer;
lon = Longitude;
lat = Latitude;
alt = Altitude;
name = Naam van het station;

1 Tactieken

Ik heb gekozen voor de volgende tactieken om het architecture and design process te leiden.

- Het ontwikkelen van de project architectuur
- Limit structural complexity.

2 System Requirements

2.1 User stories

ID	Story
US-001	Als een gebruiker wil kunnen inloggen zodat ik de enige ben die mijn gegevens zou kunnen zien.
US-002	Als een gebruiker wil ik mijn eigen weers data kunnen toevoegen zodat ik mijn weers data later kan terug zien.
US-003	Als een gebruiker wil ik mijn eigen weers data kunnen delen met andere mensen zodat andere mensen mijn weers data kunnen zien.
US-004	Als een gebruiker wil ik de meeste recente weers data kunnen zien in mijn omgevingen of een locatie naar keuze zodat ik weet wat voor weer het is op die locatie.
US-005	Als een gebruiker wil ik mijn eigen weers data kunnen zien van vroeger van een bepaalde locatie in nederland zodat ik het weer kan zien van vroeger.
US-006	Als een gebruiker wil ik de weers date kunnen filteren en sorteren op bepaalde velden zodat ik een beter overzicht heb van de data.
US-007	Als een gebruiker wil ik andere mensen hun weers data kunnen zien zodat ik een idee heb van wat van andere weerstations er zijn.
US-008	Als een gebruiker wil ik wil een weersvoorspelling kunnen zien van een bepaalde locatie in nederland zodat ik me kan voorbereiden op dat weer.
US-009	Als een developer wil ik weinig module dependencies zodat de complexiteit van de applicatie gelimiteerd blijft.

2.2 Quality Attribute Scenarios

ID	Quality Attribute	Scenario	User Story
QA-01	Performance	Het systeem moet maximaal 4000 gelijktijdige request aan kunnen op hetzelfde moment. Een devops analyst moet per week gaan deze data ophalen en gaan controleren of deze doelen ook bereikt worden. Een issue wordt aangemaakt als dit niet gebeurd.	US-002 US-003 US-004 US-006 US-007 US-008
QA-02	Performance	Verzoeken die door gebruikers zijn verzonden, moeten binnen 2 seconden worden verwerkt. Het systeem registreert de operaties en de hoeveelheid tijd om de operaties te verwerken. Als de tijd langer is dan 2 seconden, wordt er een bericht naar de beheerders gestuurd.	US-002 US-003 US-004 US-006 US-007 US-008
QA-03	Security	Alle weers data moet op nederlandse servers worden bewaard.	US-001
QA-04	Security	Het systeem mag geen enkele mogelijkheid hebben voor api-injecties en MITM-aanvallen. Aan het einde van elke ontwikkel sprint moeten de ontwikkelaars controleren of dit het geval is.hebben	US-001
QA-05	Reliability	De server mag maar slechts een down tijd van 3 uur per maand, wanneer de down tijd van de server meer dan 2 uur bedraagt wordt er in 15 minuten na deze twee uur contact opgenomen met de beheerders en devops om zo het probleem op te lossen.	US-001 US-009
QA-06	Testability	Het systeem moet unit-getest zijn. Een code dekking van 75% wordt verwacht. Elke sprint wordt de code dekking van het systeem getest door de ontwikkelaars en testers.	US-001 US-009

2.3 Constraints

ID	Constraint
CON-1	Alle gegevens moeten worden opgeslagen op Nederlandse servers
CON-2	Het systeem moet 4000 gebruikers tegelijkertijd aan moeten kunnen.

2.4 Architectural Concerns

ID	Concern
CRN-1	4000 gebruikers kunnen te veel mensen zijn en dus te veel vragen van het systeem. Hierdoor zouden de kosten van het systeem hoog kunnen liggen
CRN-2	Het systeem zou mogelijk te slecht presteren om binnen 3 seconden request af te handelen.

3 Inputs Review

<table>
<tr><td>Category</td><td>Details</td></tr>
<tr><td>Design Purpose</td><td>Dit is een compleet nieuw systeem. En het doel is om het systeem uit te rollen in de eerstvolgende release.</td></tr>
<tr><td>Primary Functional Requirements</td><td>Door middel van de user stories die staan vermeld in hoofdstuk 2.1 zijn de volgende user stories beschouwt als primair:
US-002
US-003
US-004
US-006
US-007
US-008
Deze user stories zijn gekozen om een mvp mee te maken.</td></tr>
<tr><td>Quality Attribute Scenarios</td><td><table>
<tr><td>Scenario ID</td><td>Importance to customer</td><td>Difficulty of implementation according to architect</td></tr>
<tr><td>QA-1</td><td>High</td><td>Medium</td></tr>
<tr><td>QA-2</td><td>High</td><td>Medium</td></tr>
<tr><td>QA-3</td><td>Medium</td><td>Medium</td></tr>
<tr><td>QA-4</td><td>High</td><td>Medium</td></tr>
<tr><td>QA-5</td><td>High</td><td>Low</td></tr>
<tr><td>QA-6</td><td>Low</td><td>High</td></tr>
</table></td></tr>
<tr><td>Constraints</td><td>Alle constraints voor dit project zijn gedefinieerd in hoofdstuk 2.3</td></tr>
<tr><td>Architectural Concerns</td><td>Alle architectuur concerns zijn vermeld in hoofdstuk 2.4 en vermeld in de drivers.</td></tr>
</table>

Eerste iteratie

Step 2: Iteratie doelen

Het doel van de eerste iteratie is het creëren van een functionerende microservice applicatie op die gebaseerd is op de mvp die staat vermeld in de system requirements. De uitdagingen zullen dan ook vooral zitten in het ontwerpen van de opzet architectuur.

Op basis van onze systeemvereisten willen we het volgende in gedachten houden tijdens de eerste iteratie:

- QA:
    - QA-1: Performance gerelateerd
    - QA-2: Performance gerelateerd
    - QA-5: Reliability gerelateerd
- Architectural Concerns:
    - CRN-1: Performance gerelateerd
    - CRN-2: Performance gerelateerd

Step 3: Choose one or more elements to refine

Gedurende de eerste iteratie gaan ik me focussen op het uitwerken van de basis functionaliteit en waarmee het gebouwd moet worden.

Step 4: Choose one or more design concepts that satisfy the selected drivers

De volgende tabel toont de ontwerpbeslissingen die tijdens de eerste iteratie zijn genomen met betrekking tot de ontwerpconcepten.

Design decisions and location	**Rational**
Microservices gebruiken in plaats van een monolithic approach	Waarom is de keuze voor microservice een betere optie van een monolithische approach? Het gebruik van microservices geeft de volgende voordelen ten opzichte van een monolithische approach: - Met een op microservices gebaseerde applicatie is de applicatie opgesplitst in beter beheersbare services. Die zijn gemakkelijker te begrijpen, te onderhouden en te ontwikkelen (Fowler, 2012). - Microservices kunnen individueel worden ontwikkeld; ontwerp autonomie (Richardson, 2015). - Microservices kunnen afzonderlijk worden ingezet. Dit zou maakt continue deployment mogelijk en verlaagt de downtime (Kharenko, 2015). - Microservices vereenvoudigen het schalen. Indien nodig kunnen er zonder problemen meer diensten draaien (Richardson, nd). **Nadelen**: - Het inzetten van meerdere Microservices is moeilijker dan het inzetten een monolithische applicatie (Richardson, n.d). - Het is moeilijker om wijzigingen op meerdere systemen toe te passen (Richardson, 2015). - Testen is moeilijker dan met een monolithische applicatie (Fowler, 2012). - Algehele memory footprint neemt toe door replicatie van verschillende klassen en bibliotheken in elke microservice bundel. Op basis van de bovenstaande informatie heb ik besloten om microservices te gebruiken. Met mijn doelen wegen de voordelen op tegen de nadelen. Twee van mijn huidige aandachtspunten deze iteratie zijn 4.000 gelijktijdige aanvraag (QA-1) en snelle

	bezorging van request (QA-2). Ervoor zorgen dat QA-1 wordt gehaald is veel gemakkelijker met Microservices, als we 4.000 verzoeken niet kunnen afhandelen, moeten er gewoon meer services toegevoegd worden en hierdoor is het ook meer toekomstbestendig. Hetzelfde gaat voor QA-2, als het te veel request moet verzenden en het niet aan kan , kunnen er meer diensten worden toegevoegd. QA-7 stelt ook dat er slechts een downtime van 3 uur per week mag zijn. Dit is veel gemakkelijker te bereiken met microservices want je hoeft niet meer de gehele applicatie offline te halen maar een deel of een service. **Gecanceld alternatief:** Monolithische api. **Voordelen**: - Eenvoudig te ontwikkelen (Richardson, n.d.) - Eenvoudig te testen en te debuggen (Richardson, n.d.) - Eenvoudig te implementeren (Kharenko, 2015) **Nadelen**: - Een grote applicatie wordt complex, moeilijk te begrijpen en moeilijk te onderhouden (Richardson, n.d.). - De hele applicatie moet opnieuw worden opgestart voor een update (Kharenko, 2015). - Betrouwbaarheid is vaker wel dan niet lager dan op microservice gebaseerde applicaties (Kharenko, 2015). - Moeilijker om continue development te implementeren (Richardson, n.d.). - Moeilijk te schalen (Richardson, n.d.) - Door de grote van het systeem kan het opstarten ook langer duren(Richardson, n.d.). - Monolithische applicaties worden geïmplementeerd met een single development stack, die de beschikbaarheid van “the right tool for the right job” limiteerd. - Betrouwbaarheid, omdat bijvoorbeeld een bug potentieel kan het hele proces kan laten crashen (Kharenko, 2015). De reden voor het niet uitvoeren van dit idee is het feit dat de voordelen van microservices beter aansluiten bij mijn User stories. In theorie zou een monolithicum applicatiekunnen werken, maar het zal andere problemen met zich meebrengen.

Welke programmeertaal is het meest geschikt om een microservice applicatie mee te maken?	Vanwege de aard van deze opdracht, waar geen lessen voor worden gegeven en geen programmeertaal vereisten voor waren, heb ik gekozen om typescript en nodejs te gebruiken in combinatie met kubernetes. Hiervoor heb ik gekozen omdat ik zelf geen kennis had over hoe je een microservice op een verstandige manier moet bouwen en dus een cursus heb gevolgd via Udemy. Het gaat om deze cursus: Microservices with Nodejs and React. Deze keuze staat los dus van alle andere keuzes die verder worden genomen doordat er geen onderzoek achter staat.
Hoe moet de architectuur van de microservices er uit zien als het gaat om databases?	Om erachter te komen hoe de database van de applicatie er uit moet komen te zien moet eerste de beslissing gemaakt worden welke vorm van database er gebruikt gaat worden. Omdat de applicatie in kubernetes gemaakt gaat worden met Nodejs en Typescript wordt de database gemaakt door middel van Mongodb. Alleen deze Mongodb database kan worden neergezet op twee methodes die ik zie als de meest prominenten methodes. De eerste is de database per service pattern en de tweede is de shared database pattern. Database per service Deze pattern zorgt ervoor dat elke service zijn eigen database krijgt. Alle data van de service blijft dan ook in die service staan. Voordelen: - Dit helpt zorgt ervoor dat de services losjes worden gekoppeld. Wijzigingen in de database van één service hebben geen invloed op andere services. - Elke service kan een database gebruiken dat het best past bij hun taak Nadelen: - Veel complexiteit voor het beheren van meerdere sql databases. - Als een process over meerdere databases heen gaat kan er al snel veel onnodige complexiteit ontstaan voor het afhandelen van alle taken database taken. - Data die moet worden verkregen door joins zitten nu in meerdere databases. Shared Database Voordelen: - Er hoeft maar een database worden beheerd. - Je hoeft nooit bang te zijn voor data inconsistentie.

	Nadelen: - Alle veranderingen in een service die te maken hebben met de database beïnvloeden ook alle andere services tijdens ontwikkeling. - Als een tabel wordt gelocked daar een service kan een andere service er niet bij. - Als er een hoge aanvraag is voor een tabel beïnvloedt dat ook de performance van alle andere tabellen. De keuze is gemaakt om de pattern database per service te gebruiken omdat een van de tabellen veel meer data bevat dan de andere is het delen van de database hindert dan alle andere tables. En ten tweede zijn er geen complexe verbindingen tussen de verschillende tabellen. Dus de pattern shared database is hier overbodig.

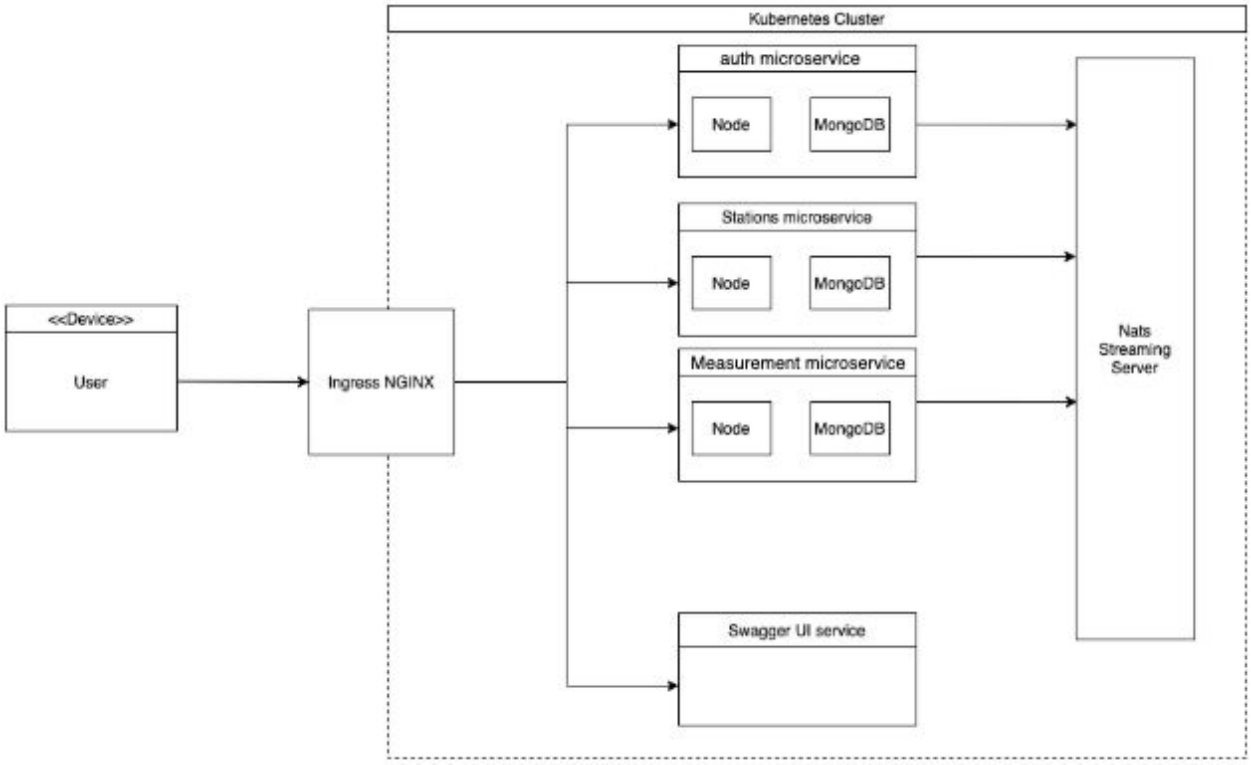
Step 5: Instantiate architectural elements, Allocate Responsibilities and Define interfaces

Design decisions and location	**Rationale**
Microservices bouwen	Sommige van onze zorgen en QA's hebben verbindingen met performance. Door het developen met behulp van Microservices wordt dit probleem aangepakt. De api zal bestaan uit een reeks kleine diensten, zoals OATH, weerberichten delen en weerstations delen , die zelfstandig werken en met elkaar communiceren. Elke service kan door meerdere diensten worden gebruikt.
Deployment strategie	Een van de grootste veranderingen tijdens het overstappen van een monolithische app naar microservices is de manier waarop we onze applicatie implementeren. **Service-instance per container.** Dit patroon is vergelijkbaar met het patroon dat we kiezen, maar gebruikt containers, zoals docker bijvoorbeeld in plaats van een VM. Een van de voordelen over VM is het feit dat containers veel gemakkelijker zijn en sneller te bouwen dan VM (Richardson, 2016). Voordelen: - Makkelijker op te zetten dan VMs - Scalability is veel beter - Het beheren van de verschillende containers is makkelijker te beheren dan de VMs Nadelen: - Alles containers hebben 1 access dus een persoon door de beveiliging is kan ie bij alle containers komen. - Het is minder volwassen dan VMs **Service Instance per Host Pattern** Deze pattern zorgt ervoor dat een vm wordt gebruikt per service dus iedere service krijgt in gehele omgeving om in te draaien in plaats van de Service-instance per container waar meerdere service draaien in een omgeving. Voordelen: - Elke service-instantie wordt geïsoleerd uitgevoerd (Richardson, 2016).

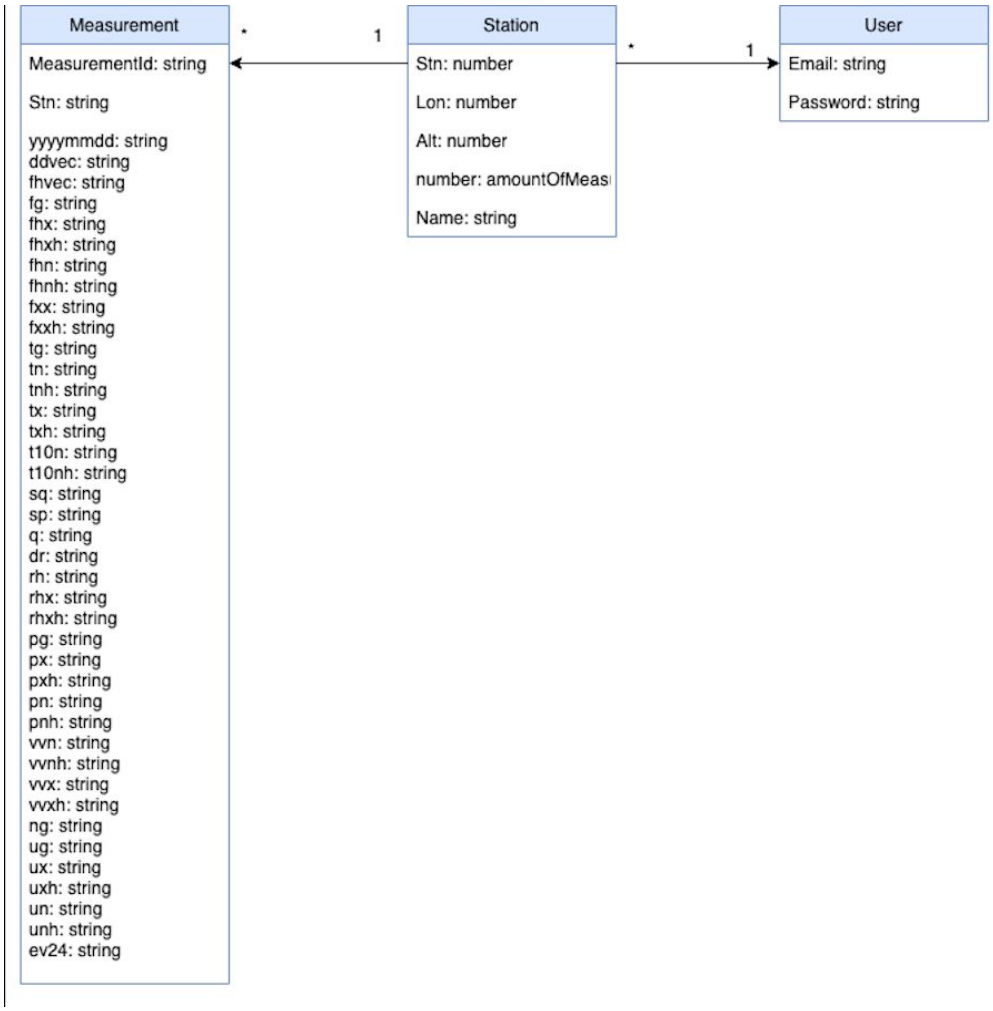
	- Elke service-instantie heeft een vaste hoeveelheid ram en cpu en kan geen middelen stelen van andere services (Richardson, 2016). - Het service-exemplaar is ingekapseld zodra het is verpakt als VM wordt een black box (Richardson, 2016). - VM-machines draaien op de meeste grote hostingproviders (Richardson, 2016). Nadelen: - Minder efficiënte toewijzing van middelen (Richardson, 2016). - Elke VM heeft de overhead van een VM (Richardson, 2016). - Het implementeren van een nieuwe service is traag. (Richardson, 2016). De pattern die gekozen is als deployment strategy is de **Service-instance per container.** Omdat deze het meest past bij de QAS en de grote van het systeem. De api in een vm bouwen is te complex en met behulp van containers blijft het een overzicht baar project.

Step 6: Sketch views and record design decisions

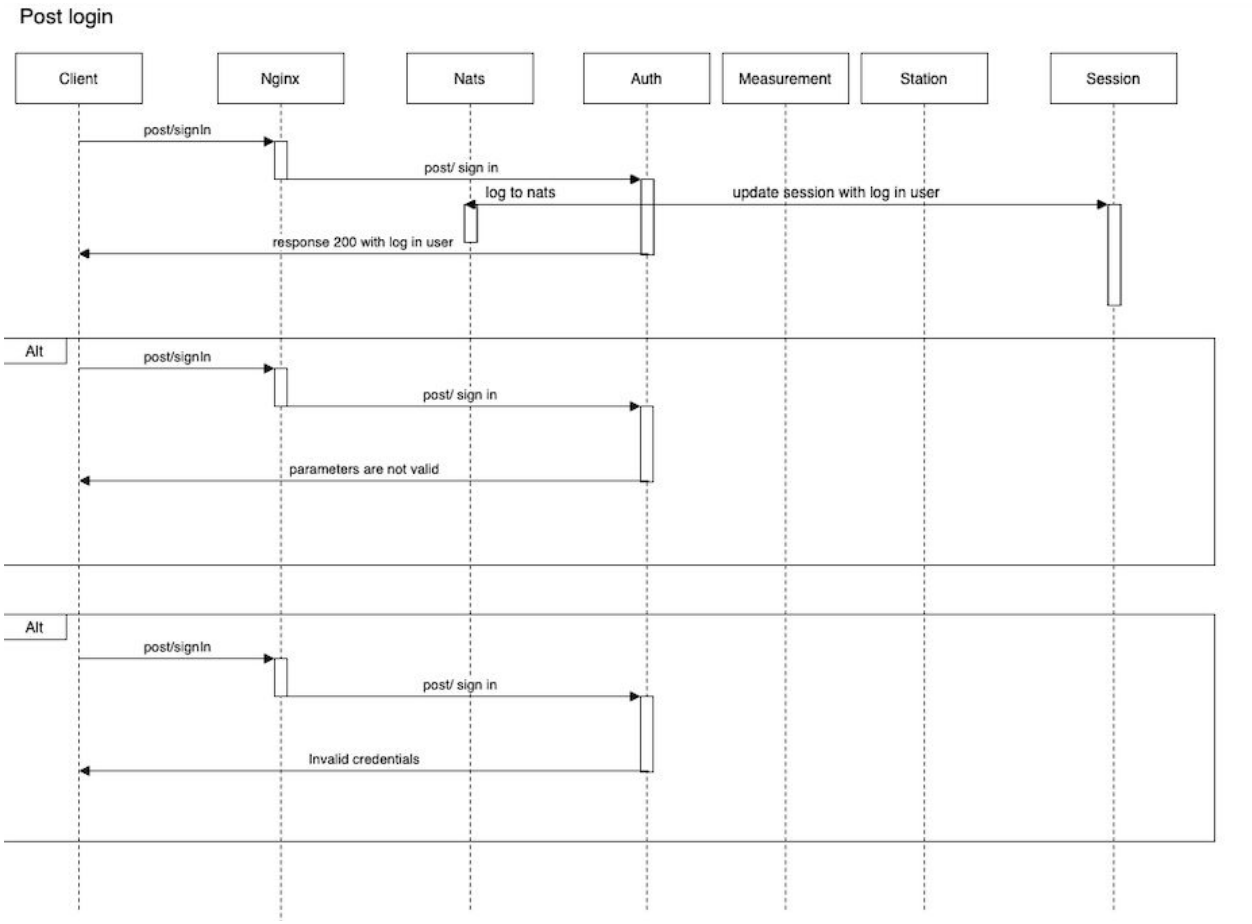
Deployment strategy en database strategy
De onderstaande diagram laat de deployment strategie en database strategie zien die is gekozen tijdens deze iteratie. Hiernaast is ook het domain model te zien met alle sequence diagrammen.



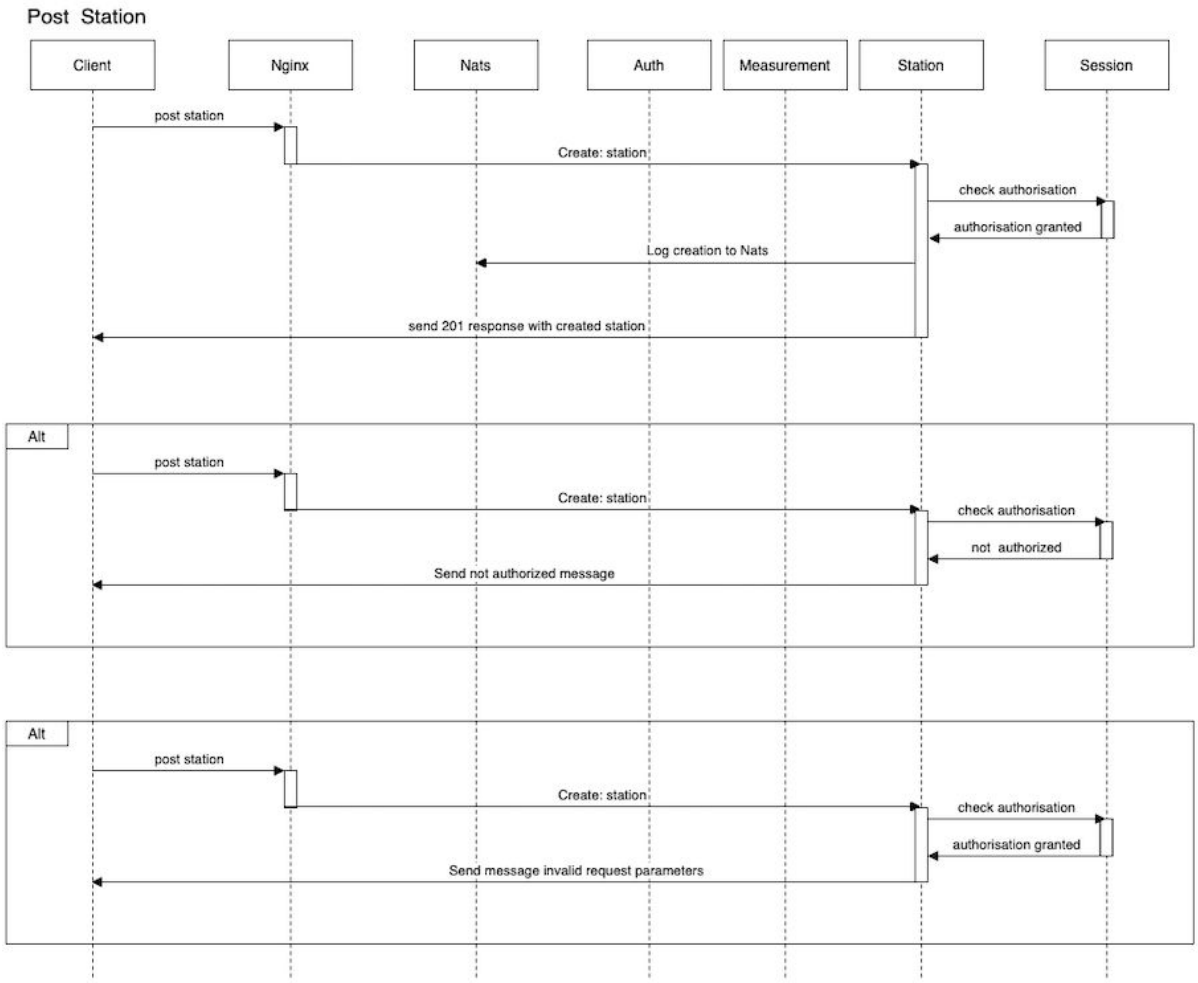
Figuur 1. deployment en database strategie



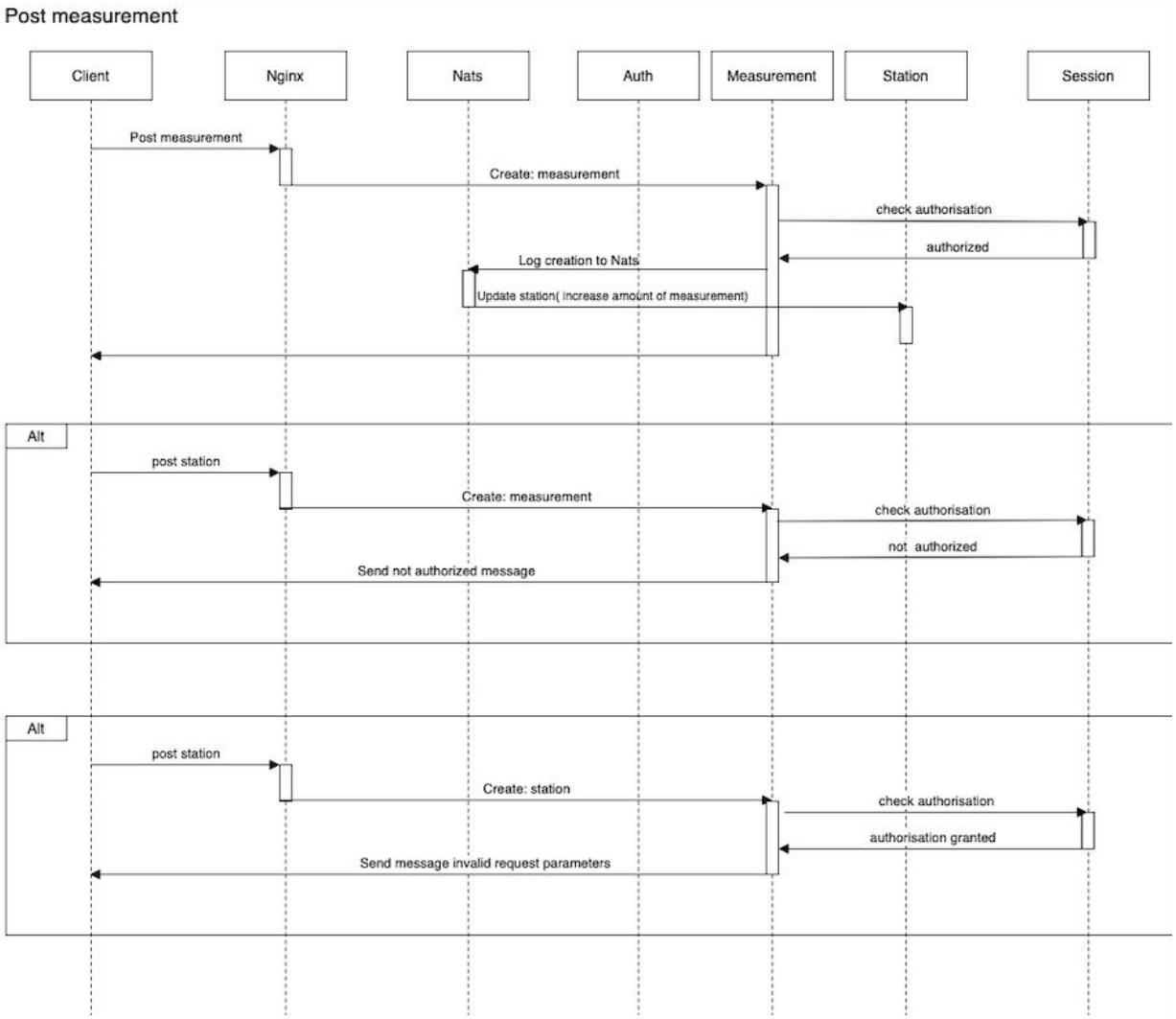
Figuur 2. Domain model



Figuur 3. Sequence diagram post login



Figuur 4. Sequence diagram post station



Figuur 5. Sequence diagram post measurement

Step 7: Perform analysis of the current design and review the iteration goals and achievement of the design purpose.

In the following table we show our design progress:

Not addressed	**Partially addressed**	**Completely addressed**	**Design decisions made during the iteration**
		QA-01	Door de implementatie van de microservices is de load op de server lager en is de performance hoger en dus sneller.
	QA-02		Nog niet geheel opgelost.
		QA-03	De cluster komt te draaien op nederlandse servers.
QA-04			
		QA-05	De implementatie van microservices zorgt ervoor dat de down time minimaal is doordat niet het gehele systeem offline hoeft worden gehaald bij updates.
QA-06			

Omdat de user stories gekoppeld zijn aan de QA zijn deze niet vermeld in deze tabel.

Tweede iteratie

Step 2: Iteratie doelen

Het doel van de tweede iteratie is om voort te bouwen op de basis van de microservices die is neergezet in de eerste iteratie en zo de primaire functionaliteit in te bouwen. De iteratie focust dus op usability en functionality.

Op basis van de systeemvereisten wordt het volgende in gedachten gehouden tijdens de tweede iteratie:

- QA:
    - QA-2: Performance gerelateerd
- Architectural Concerns:
    - CRN-1: Performance gerelateerd
    - CRN-2: Performance gerelateerd

Step 3: Choose one or more elements to refine

Gedurende de twee iteratie gaan ik me focussen op het uitwerken van de functionaliteit en gebruiksvriendelijkheid.

Step 4: Choose one or more design concepts that satisfy the selected drivers

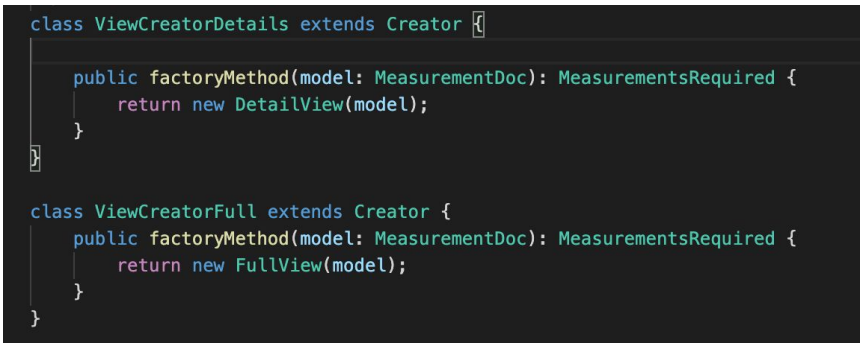
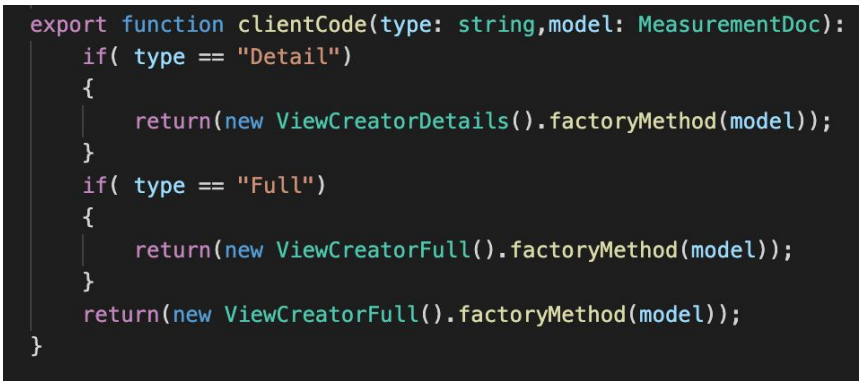
De volgende tabel toont de ontwerpbeslissingen die tijdens de eerste iteratie zijn genomen met betrekking tot de ontwerpconcepten.

Design decisions and location	**Rational**
Weergeven van weers metingen.	Om ervoor te zorgen dat de weers data makkelijk kan worden bekeken worden door de gebruiker moet er een converter gemaakt worden voor de weers metingen die de nogal uitgebreide weers data ook klein moet maken zodat het begrijpbaar wordt. Dit moet worden gedaan door bepaalde patterns in te zetten. Deze patterns zijn de factory pattern en de iterator pattern. De factory pattern wordt gebruikt om een overzicht mee op te bouwen. Zoals het nu is komt alle data uit de api in een lange lijst te staan per measurement. Dit moet veranderd worden. De gebruiker moet kunnen kiezen wat voor overview die komt te zien een volledige of alleen de details. Dit wordt gedaan door de measurement en keuze voor de overview door te sturen naar de factory pattern. Deze convert de measurement dan. Om de gehele lijst te converteren wordt gebruikt gemaakt van de iterator pattern. Deze iterate over de lijst heen en stuurt die per measurement door naar de factory converter.
In de eerste sprint is de structuur vast gezet van alle kubernetes pods en services. In deze sprint worden deze structuur gemaakt tot een werkend systeem.	Alle beslissingen van deze implementatie zijn al gemaakt in de eerste sprint. Alle kleine veranderingen zullen hier niet worden besproken omdat dat onderdeel is van het development process.

Api calls zichtbaar maken	Om ervoor te zorgen dat de verschillende api calls overzichtelijker worden moet er op een bepaalde manier een overzicht komen van de verschillende calls. Dit kan op meerdere manieren en de twee keuze die ik heb gekozen om naar te kijken zijn Postman en Swagger UI. **Postman** Postman is een api platform waarmee api kunnen worden ontwikkeld in teamverband en daarnaast gebruikt kan worden op api calls te registreren voor development doelen. Op deze manier kan je een bibliotheek maken aan calls specifiek voor een bepaald project. **Voordelen:** - Er kunnen calls worden getest op regressie doordat je steeds dezelfde calls met dezelfde parameters kan maken en de response kan vergelijken. - Doordat het een eigen applicatie is zit je niet vast aan een opzet en beheer van de applicatie. - De hoevelen keuzes die je hebt per call als het gaat om auth, headers, body type enz. zorgt ervoor dat er nog opties zijn om vrijwel alle soorten calls te maken. **Nadelen:** - Doordat het niet direct onderdeel uitmaakt van het api systeem ben je afhankelijk van de uitgever van postman. - Je kan niet de api calls readonly maken hierdoor kunnen mensen opties veranderen waardoor de calls niet meer overheen komen met postmann. **SwaggerUI** Swagger UI is onderdeel van Swagger.io en is een omgeving waarmee door middel van een config bestand een api documentatie wordt gemaakt waarin api calls kunnen gemaakt worden. **Voordelen:** - Omdat het een image is valt het binnen het systeem en daardoor kan jezelf swagger-ui beheren. - Door de readonly nature van swagger-ui zal de documentatie altijd hetzelfde zolang je niet het

	afgeschermde config file gaat aanpassen. **Nadelen** - Het is te gelimiteerd dus aanpassingen aan auth, headers, query zijn moeilijk te maken. - Niet bedoeld voor development Omdat geen van beide precies alle requirements invult voor api call gebruik is besloten om beide manieren te implementeren.

Step 5: Instantiate architectural elements, Allocate Responsibilities and Define interfaces

Design decisions and location	**Rationale**
Factory pattern implementeren	Hieronder is de implementatie te zien van de factory pattern.  Figuur 6. Factory pattern view creator De factory maakt een view aan, aan de hand van de string die wordt meegegeven aan de cliënt code methode.  Figuur 7. Factory pattern client code
Iterator pattern implementeren.	Hieronder is de implementatie te zien van de iterator pattern.

```
constructor(collection: MeasurementsCollection, reverse: boolean = false) {
    this.collection = collection;
    this.reverse = reverse;

    if (reverse) {
        this.position = collection.getCount() - 1;
    }
}

public rewind() {
    this.position = this.reverse ?
        this.collection.getCount() - 1 :
        0;
}

public current(): MeasurementDoc {
    return this.collection.getItems()[this.position];
}

public key(): number {
    return this.position;
}

public next(): MeasurementDoc {
    const item = this.collection.getItems()[this.position];
    this.position += this.reverse ? -1 : 1;
    return item;
}

public valid(): boolean {
    if (this.reverse) {
        return this.position >= 0;
    }
```

Figuur 8. Factory pattern collection

```
export class MeasurementsCollection implements Aggregator {
    private measurements: MeasurementDoc[];



    constructor(measurements: MeasurementDoc[]) {
        this.measurements = measurements;
    }

    public getItems(): MeasurementDoc[] {
        return this.measurements;
    }

    public getCount(): number {
        return this.measurements.length;
    }

    public addItem(measurement: MeasurementDoc): void {
        this.measurements.push(measurement);
    }

    public getIterator(): CustomIterator<MeasurementDoc> {
        return new FieldIterator(this);
    }

    public getReverseIterator(): CustomIterator<MeasurementDoc> {
        return new FieldIterator(this, true);
    }
}
```

Figuur 9. Factory pattern Iterator

De iterator hierboven zorgt ervoor dat de verschillende weerberichten kunnen worden uitgelezen.

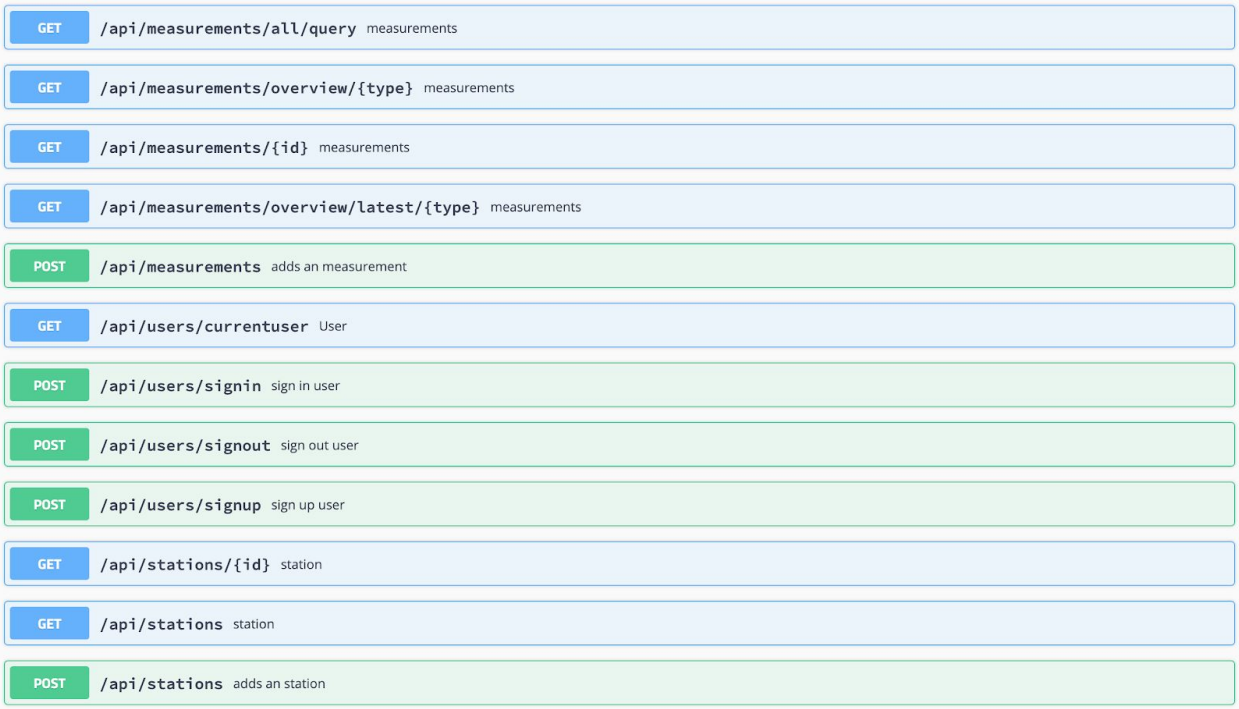
Step 6: Sketch views and record design decisions

Implementatie van het Cluster

```
NAME                                      READY   STATUS    RESTARTS   AGE
auth-depl-5cf644874-2mm89                 1/1     Running   0          27h
auth-mongo-depl-fcd4cc6c6-bz6gw           1/1     Running   0          27h
measurements-depl-75b6cc8c5c-czn9h        1/1     Running   0          27h
measurements-mongo-depl-65f88788dd-lmkb9  1/1     Running   0          27h
nats-depl-6cfdd57765-gdbgl                1/1     Running   0          27h
stations-depl-54c5597695-xmcs2            1/1     Running   0          27h
stations-mongo-depl-59495dc9d8-swjmc      1/1     Running   0          27h
swagger-ui-8574d7d697-kj2j8               1/1     Running   0          27h
swagger-ui-8574d7d697-qwjw4               1/1     Running   0          27h
```

Figuur 10. Cluster

Implementatie van Swagger UI



Figuur 11. Swagger UI

Voor een volledige demonstratie van dit systeem kunt u het beste een bericht sturen naar: Gerbrecht.Ghijsels@hva.nl

Step 7: Perform analysis of the current design and review the iteration goals and achievement of the design purpose.

In the following table we show our design progress:

Not addressed	**Partially addressed**	**Completely addressed**	**Design decisions made during the iteration**
		QA-01	Al geadresseerd
		QA-02	Door het gebruik van de patterns is de snelheid waarmee de request gemaakt mee worden verhoogd doordat de gebruiker gemakkelijker informatie kan verkrijgen en uitlezen.
		QA-03	Al geadresseerd
QA-04			
		QA-05	Al geadresseerd
QA-06			

Omdat de user stories gekoppeld zijn aan de QA zijn deze niet vermeld in deze tabel.

Installatie

Omdat de development op een mac is gedaan is deze installatie guide voor een mac bedoel. Mocht u dit systeem op een windows computer willen draaien verzoek ik u om per stap uit te zoeken welke commands u zelf nodig heeft om het systeem te draaien.

Download de source code van Github via de onderstaande link. De cardsudemy map moeten worden gezien als de root van het project. Alle andere mappen zijn alleen voor ondersteuning tijdens de development.
Source code: https://github.com/GerbrechtGhijsels/IOTApplications/tree/Api

Stap 1.

Installeer docker desktop.
Als docker desktop geïnstalleerd zijn kan er via het preferences scherm kubernetes worden geïnstalleerd.

Installer kubectl om kubernetes commands te runnen:
brew install kubectl

Installer ingress door dit command:
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/cloud/deploy.yaml

Installeer skaffold zodat het project kan worden opgebouwd:
brew install skaffold

Stap 2.

Elke service moet eerst worden gebuild om te zorgen dat het opstart process slaagt. Hiervoor moet in de root map van het project deze commands worden gedraaid:

```
docker build -t gg/measurements ./measurements
docker build -t gg/stations ./stations
docker build -t gg/auth ./auth
docker build -t gg/auth ./requester
docker build -t gg/auth ./client
```

Stap 3.

Maak een JWT key aan als secret zodat de oath werkt. Dit kan doormiddel van dit command:
kubectl create secret generic jwt-secret --from-literal=JWT_KEY=asdf

Stap 4.

In de file etc/host moet het volgende worden toegevoegd zodat dit url: https://measuring.dev doorwijst naar 127.0.0.1. Dit kan door middel van dit command:
sudo nano /etc/hosts

En dan dit onderaan toe te voegen aan de file.
127.0.0.1 measuring.dev

Stap 5.

Als je hier bent gekomen zonder probleem kan je doormiddel door het command: skaffold dev het systeem opstarten. Skaffold dev moeten worden gerunt in de folder met de skaffold.yaml file.

Doordat skaffold een development tool is waarmee je alle pods en serives mee kan opstarten ben je als je skaffold stop alle databases kwijt. Door het command skaffold dev --no-prune te gebruiken in plaats van skaffold dev worden je databases bewaard. Skaffold delete namelijk alle pods als ie wordt afgesloten dus ook alle data in de pods. Door --no-prune er achter te zetten in het command delete ie niet de data van de pods.

Alle databases zijn bereikbaar via de volgende ips:

Oath Host: 127.0.0.1 Port 30009
Stations Host: 127.0.0.1 Port 30008
Measurements Host: 127.0.0.1 Port 30007

Een programma wat je hier voor kan gebruiken is MongoDB Compass.

Stap 6.

Als het goed is draait er nu een kubernetes cluster met de volgende pods en services:

Kubectl get pods

```
GerbrechtsAir3:cardsudemy gerbrechtghijsels$ Kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
auth-depl-5544fc6f4c-gn6n6              1/1     Running   0          31m
auth-mongo-depl-54bb5d74b5-dnvbz        1/1     Running   0          31m
client-depl-5d756787c-gvlvv             1/1     Running   0          31m
measurements-depl-554c485b8b-qflmp      1/1     Running   0          31m
measurements-mongo-depl-67b5658f9-mrrhr 1/1     Running   0          31m
nats-depl-7f7fd7f97b-xxt2l              1/1     Running   0          31m
requester-depl-6489884f67-q2cqv         1/1     Running   0          31m
stations-depl-7bf5c6bd8-4dcsm           1/1     Running   0          31m
stations-mongo-depl-5f54797bf9-92k6w    1/1     Running   0          31m
swagger-ui-5467679d64-rqx7f             1/1     Running   0          31m
swagger-ui-5467679d64-sr7nw             1/1     Running   0          31m
```

Figuur 12. Kubectl Pods

Kubectl get services

```
auth-mongo-srv          NodePort    10.101.146.243   <none>        27017:30009/TCP     45m
auth-srv                ClusterIP   10.110.172.128   <none>        3000/TCP            45m
client-srv              ClusterIP   10.103.224.240   <none>        3000/TCP            45m
kubernetes              ClusterIP   10.96.0.1        <none>        443/TCP             50d
measurements-mongo-srv  NodePort    10.103.51.26     <none>        27017:30008/TCP     45m
measurements-srv        ClusterIP   10.108.218.173   <none>        3000/TCP            45m
nats-srv                ClusterIP   10.107.153.69    <none>        4222/TCP,8222/TCP   45m
requester-srv           ClusterIP   10.100.76.78     <none>        3000/TCP            45m
stations-mongo-srv      NodePort    10.104.96.172    <none>        27017:30007/TCP     45m
stations-srv            ClusterIP   10.107.209.217   <none>        3000/TCP            45m
swagger-ui              NodePort    10.104.26.3      <none>        80:30010/TCP        45m
```

Figuur 13. Kubectl Services

Alle services behalve mongodbstations moeten aanwezig zijn.

Nu heb je een werkend systeem alleen zonder een dataset. Deze dataset moet je toevoegen door deze commands te draaien als het systeem draait.

mongoimport --host 127.0.0.1:30007 --jsonArray --db stations --collection stations --drop --file stations.json

mongoimport --host 127.0.0.1:30008 --jsonArray --db measurements --collection measurements --drop --file measurements.json

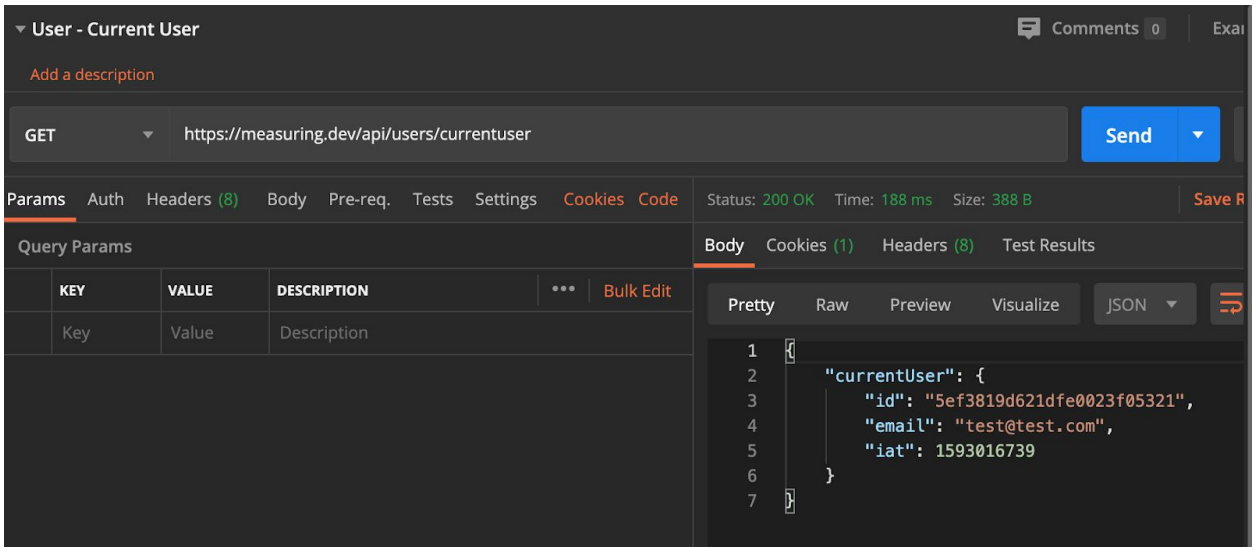
Stap 7.

Door naar het url https://measuring.dev/swagger-ui te gaan kan je inzicht krijgen in alle api calls die gemaakt kunnen worden.

Met behulp van het programma postman kunnen makkelijk de api calls gemaakt worden. In de root folder staat een export bestand van postman met daarin alle calls. Deze kan geimporteerd worden in postman.

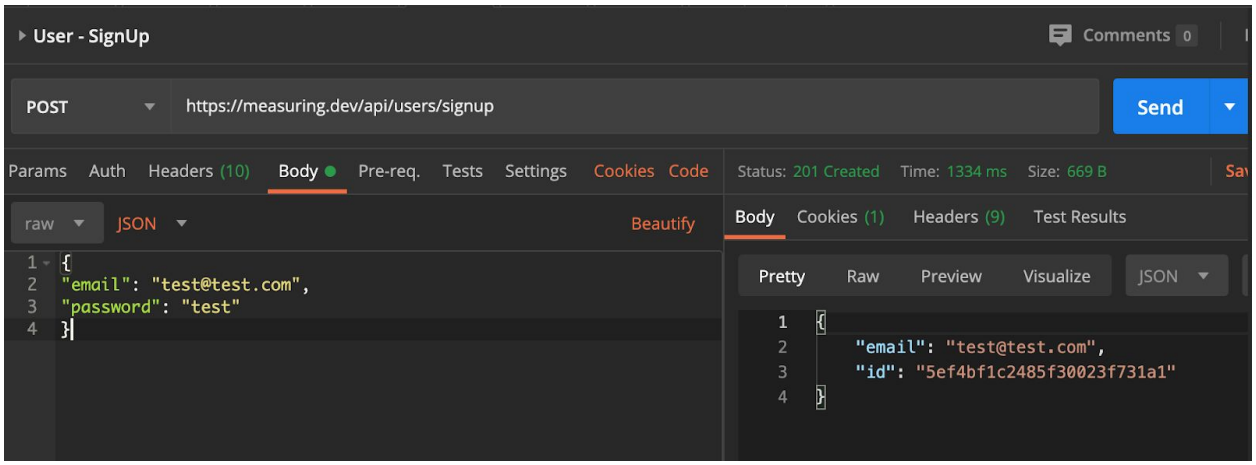
Postman

Api call om de huidige user te krijgen



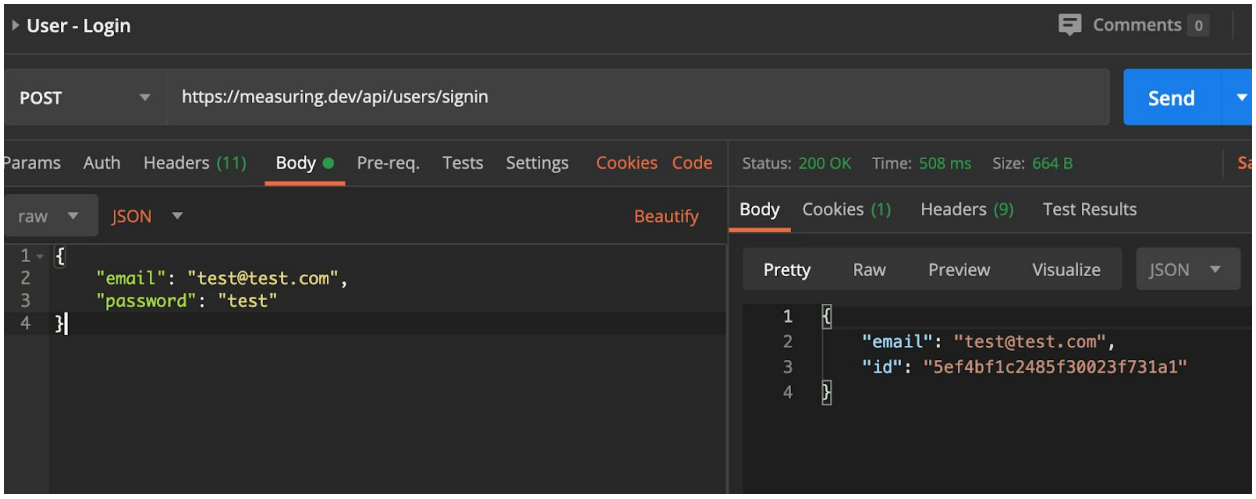
Figuur 14. Api call get Current user

Api call om in te registreren



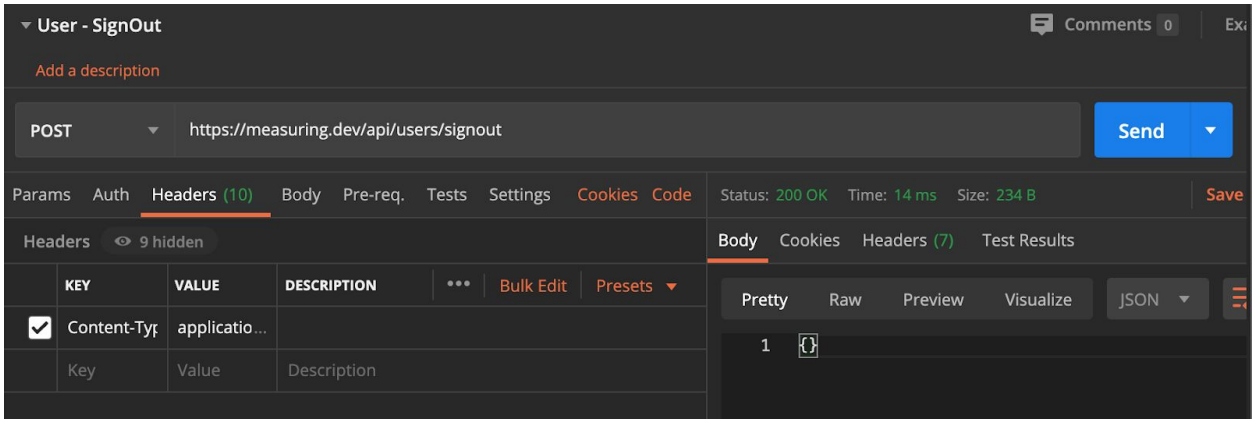
Figuur 15. Api call sign up

Api call om in te loggen



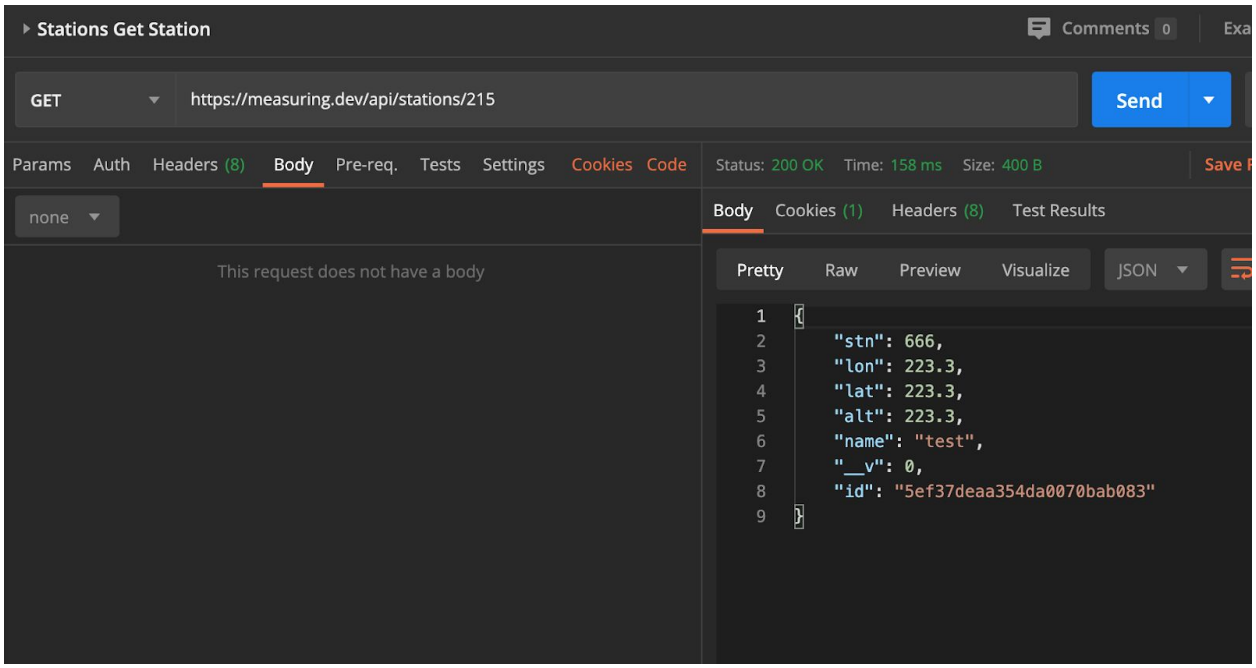
Figuur 16. Api call Sign In

Api call om uit te loggen



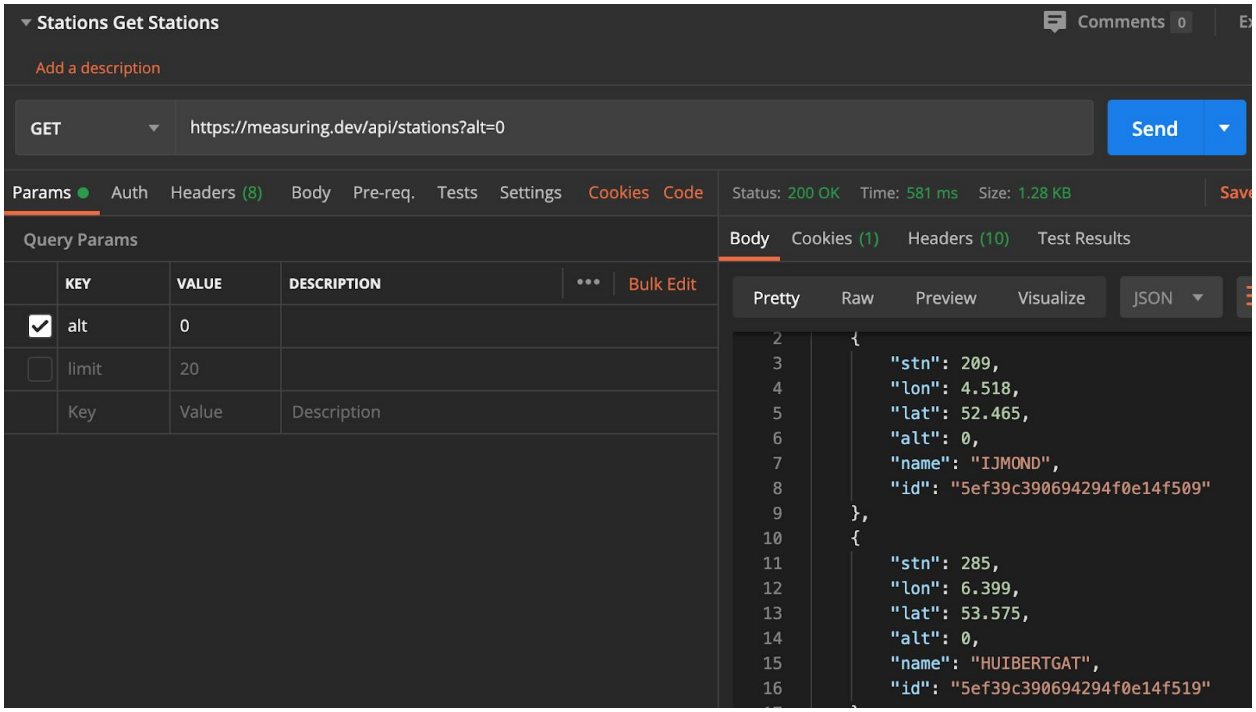
Figuur 17. Api call logout

Api call om een station te krijgen door een id



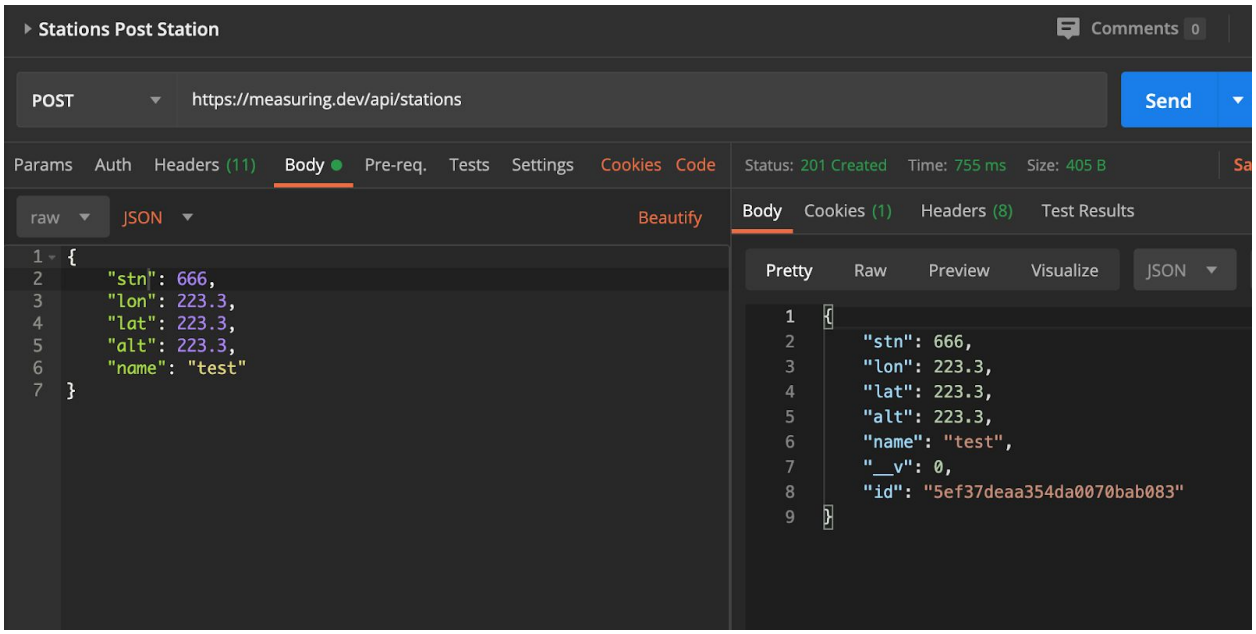
Figuur 18. Api call get station

Api call om alle stations te krijgen met mogelijk om te filteren om velden



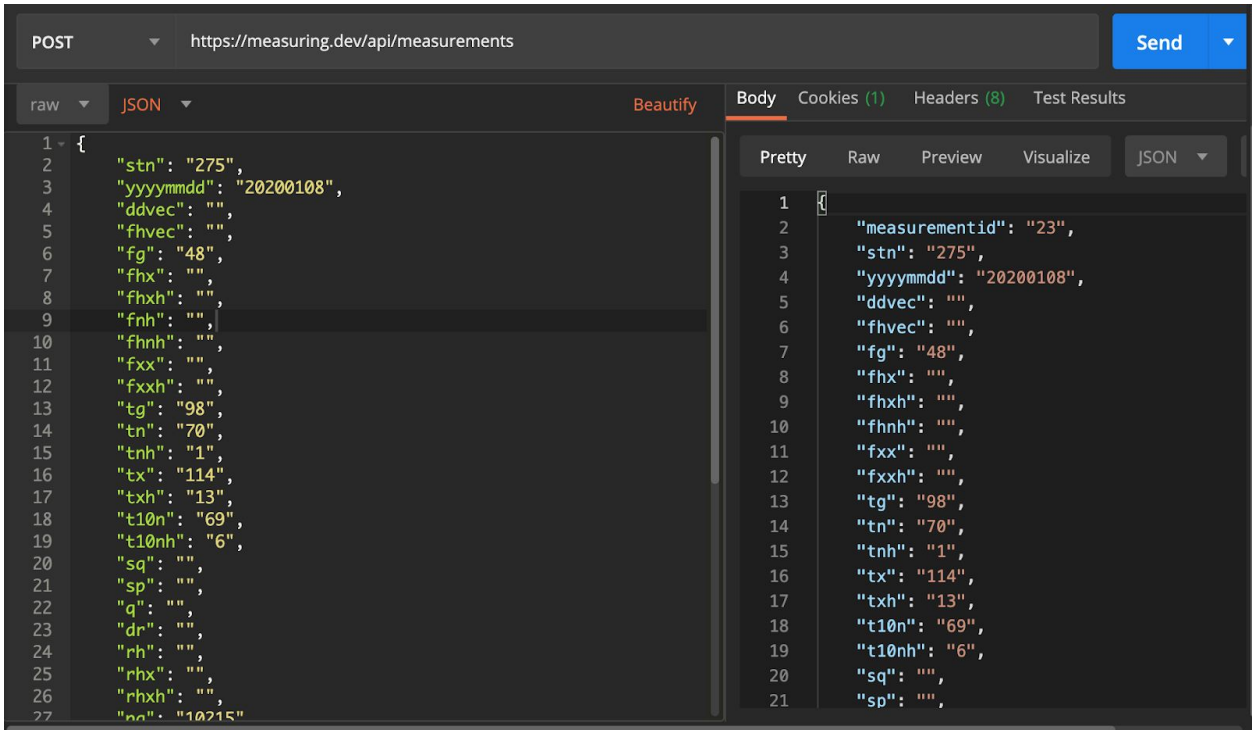
Figuur 19. Api call get station filter

Api call om een station te posten



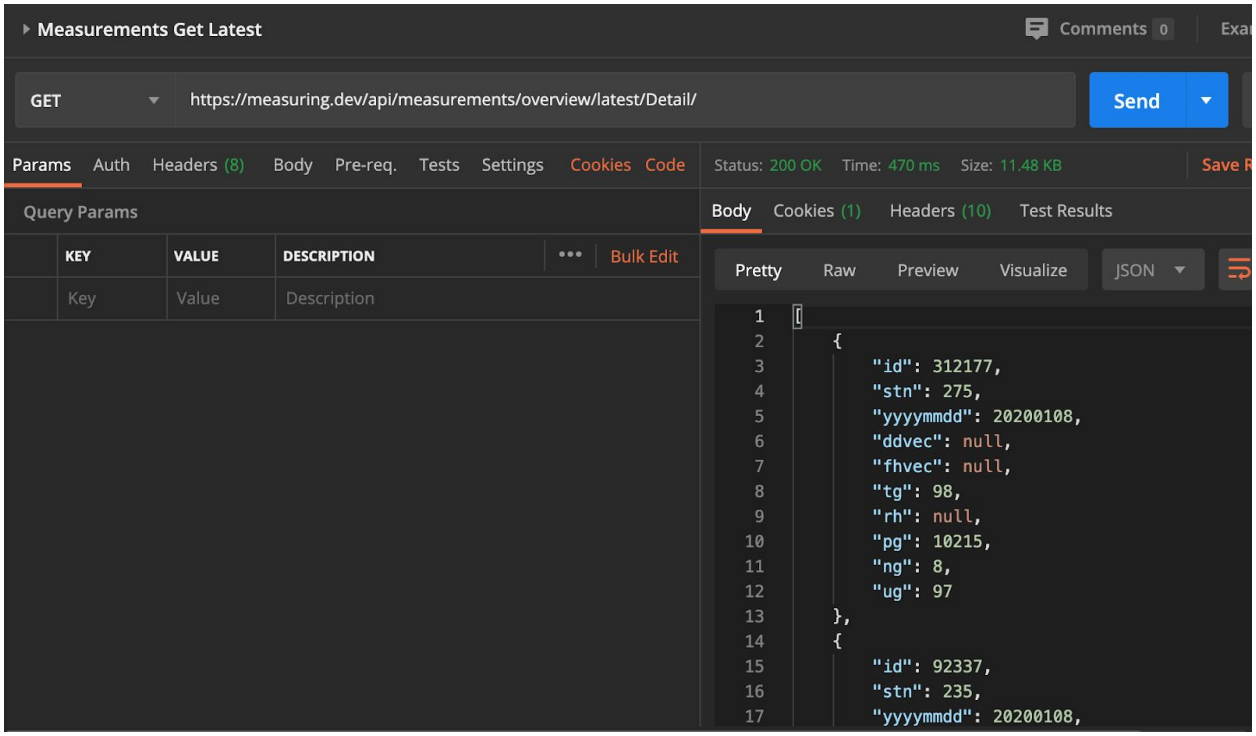
Figuur 20. Api call post station

Api call om alle measurements te krijgen met een limiet van 1000



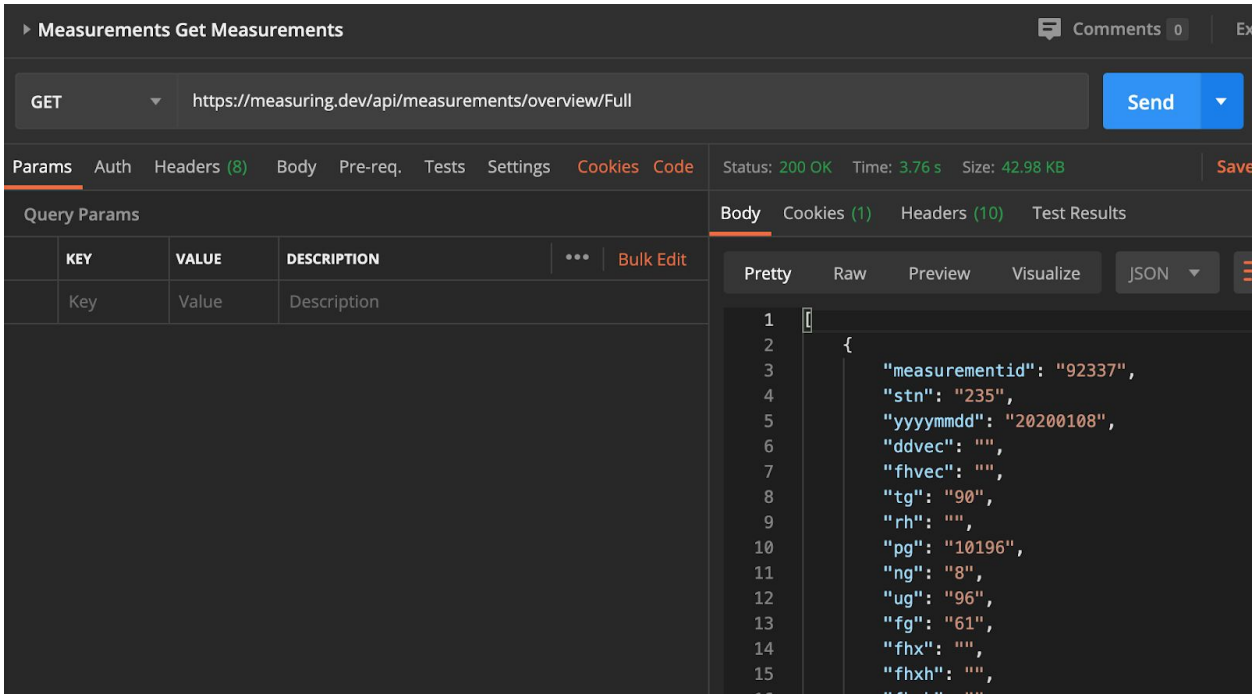
Figuur 21. Api call get measurements

Api call om de laatste measurement te krijgen en te kunnen laten zien in het formaat Detail of Full



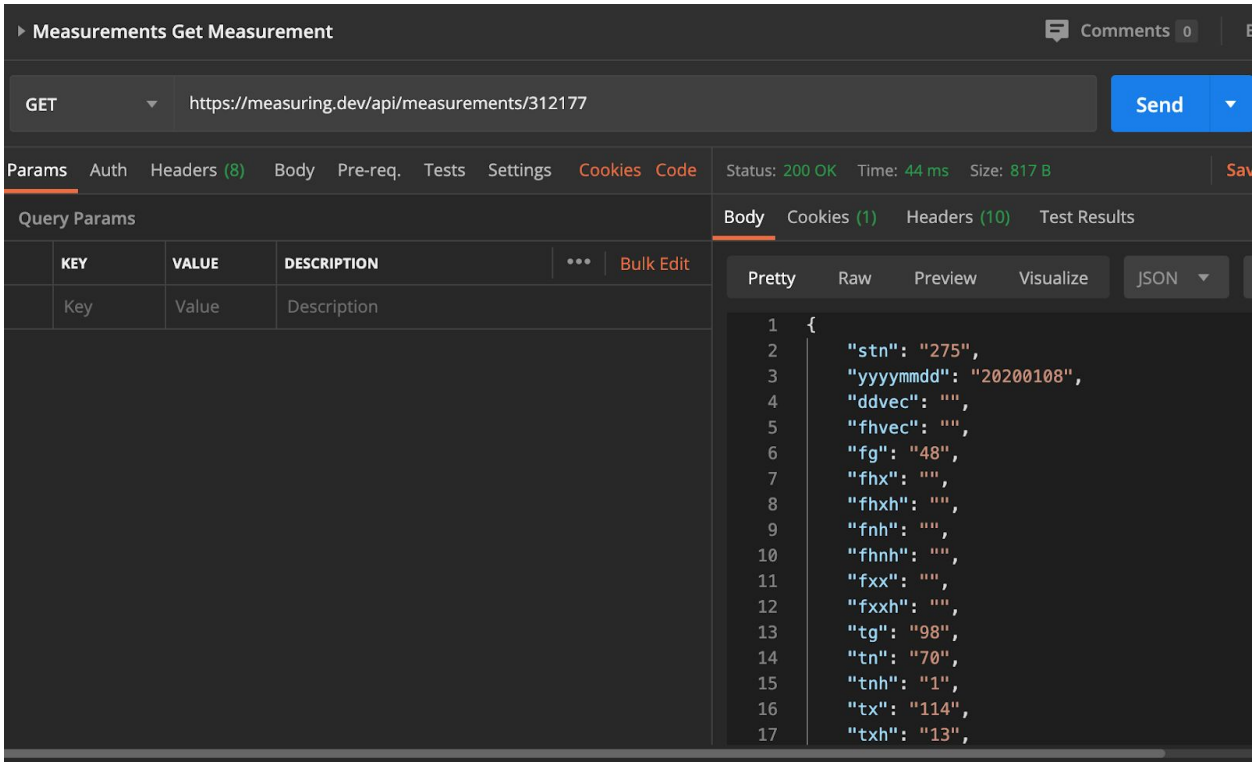
Figuur 22. Api call get last measurement

Api call om measurement te krijgen en te kunnen laten zien in het formaat Detail of Full



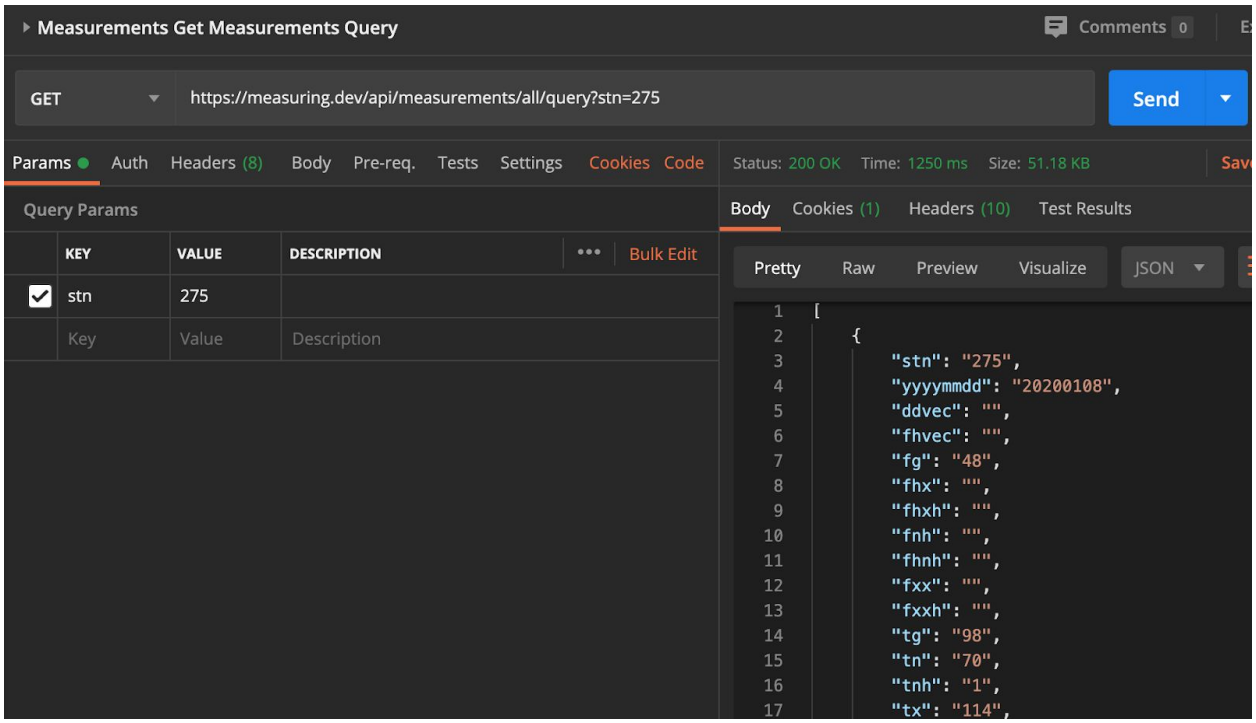
Figuur 23. Api call get measurement detail of full

Api call om een measurement te krijgen



Figuur 24. Api call get measurement

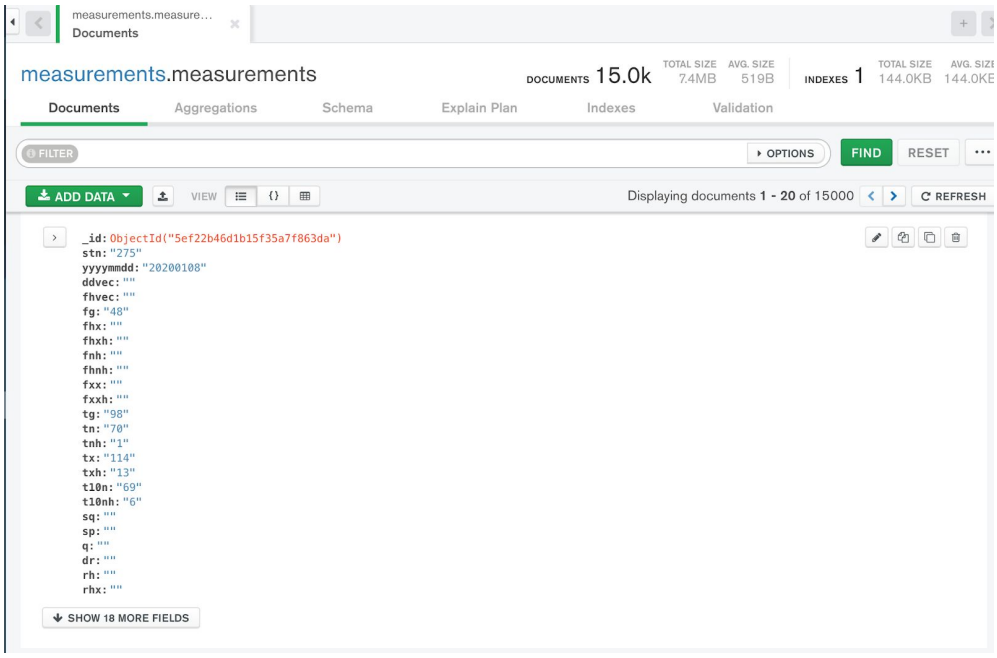
Api call om alle measurement te krijgen en te filteren op bepaalde velden



Figuur 25. Api call get measurement query

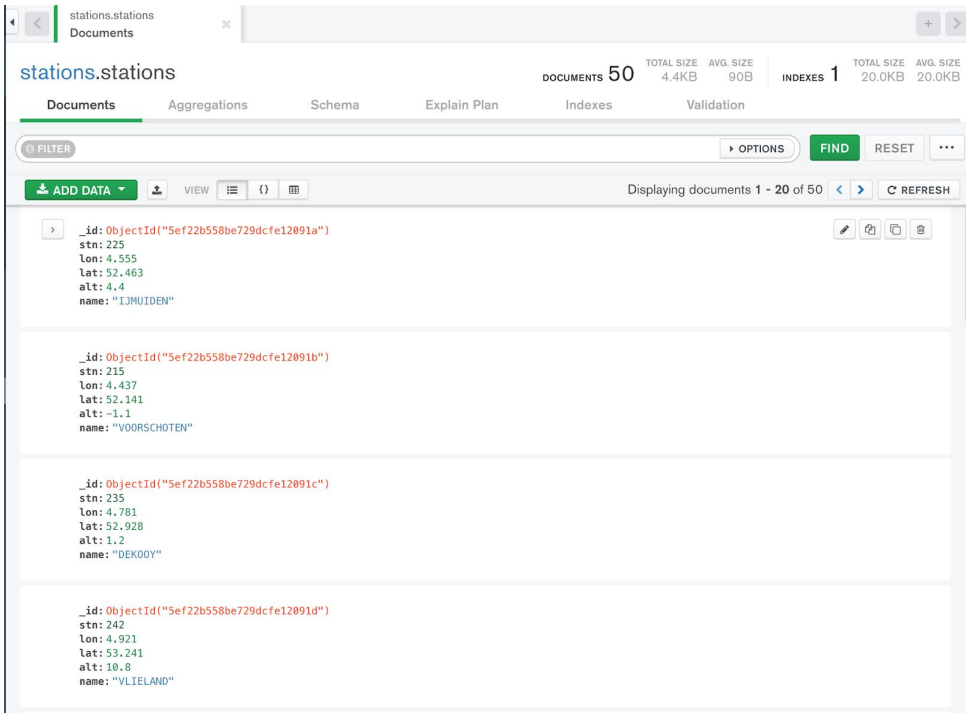
Databases

Measurement database 15000 rows



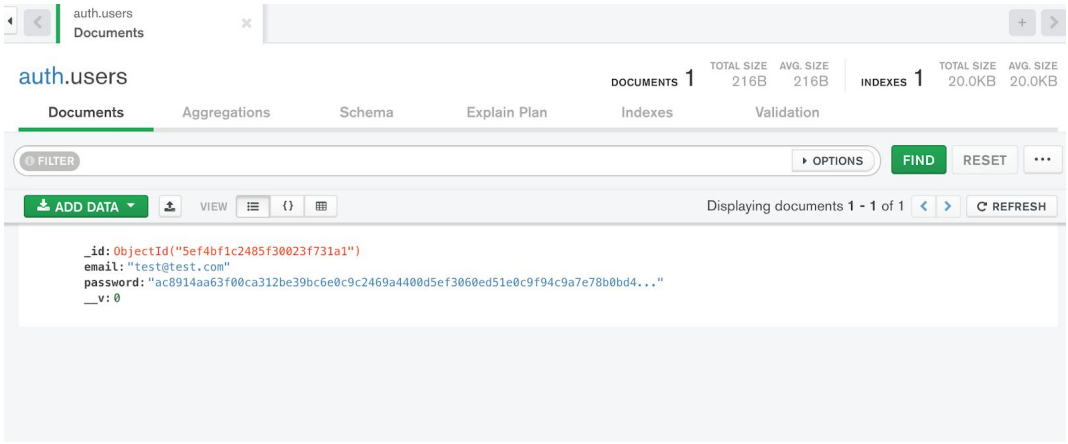
Figuur 26. Database measurements

Stations database 50 rows



Figuur 27. Database stations

User database 1 row



Figuur 28. Database Users