Bouw een Big Data platform in de Cloud

Realisatie

Bachelor in de Elektronica-ICT
Keuzerichting: Cloud en Cybersecurity

Yari Van Doninck

Academiejaar 2021-2022

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

Inhoudsopgave

1. Inleiding

Gedurende 13 weken ben ik volledig in de wereld van Big Data gedoken op een stage bij het bedrijf Ordina in Mechelen. Ordina is een van de grootste IT-consultancy bedrijven in de Benelux. Zij proberen de nieuwste IT-trends bij te brengen bij klanten en ondersteuning te bieden op alle terreinen binnen IT om de concurrentie altijd een stapje voor te blijven.

Een van de vele takken binnen het bedrijf is DataDriven. Binnen deze unit wordt aan elk onderdeel van data in IT gewerkt. Dit gaat dan over het verwerken en modelleren van data, analyseren en visualiseren van data, Data Science en nog vele andere datatoepassingen.

De stageopdracht luidde om een fictief Big Data platform in de Cloud op te zetten waar later rapportage en andere datatoepassingen op konden worden uitgevoerd.
Dit document bevat de onderdelen die ik gedurende 13 weken onder de loep heb genomen om dit fictief Big Data platform op te bouwen.

ORDINA
Ahead of change

2. Onderdelen

Ik heb gedurende mijn stage verschillende onderdelen van data benaderd. Hieronder overloop ik kort wat elk van deze onderdelen inhield. Vanaf het volgende hoofdstuk in dit document worden deze onderdelen uitgebreider toegelicht.

Opstart van de stage

Voordat ik aan de slag kon gaan aan de technische onderdelen moest ik er eerst voor zorgen dat alle communicatietools tussen mijn stagementor op Ordina en mijn stagebegeleider van Thomas More in orde waren.

Data exploratie

Met dit onderdeel heb ik me de eerste weken bezig gehouden. Ik heb alle data waarmee ik aan de slag moest gaan eerst grondig bestudeerd. Zo wist ik goed waarover de data ging en hoe deze was opgebouwd.

Het Big Data platform

Al de data, zowel de oorspronkelijke data als later de bewerkte data, moest ergens op een centrale plaats opgeslagen worden. Hiervoor heb ik de Cloudservice Amazon Web Services gebruikt. AWS biedt namelijk enorm uitgebreide en gebruiksvriendelijke toepassingen aan om met data of Big Data te werken.

Data Engineering en Modellering

De oorspronkelijke data was een wirwar van tabellen en kolommen die verkeerde benamingen en corrupte gegevens bevatte. De data was niet echt bruikbaar voor eventuele rapportering, analyses en andere datatoepassingen. Tijdens deze fase van de stage heb ik de data opgeschoond en een volledig nieuw datamodel gebouwd dat wat meer afstemde op latere toepassingen die nog moesten gebeuren.

Visualisatie in AWS QuickSight

Nadat het nieuwe datamodel klaar was en de nieuwe tabellen en data gevormd waren kon ik starten met de eerste toepassing, namelijk het visualiseren van de data in AWS QuickSight op dashboarden. Deze visualisatie diende bijvoorbeeld om bepaalde conclusies te trekken of trends te ontdekken. Zonder de visualisatie zou dat vele moeilijker zijn.

Data Science toepassing

Een andere interessante toepassing was Data Science. Zo kon ik met behulp van een beetje Machine Learning bepaalde voorspellingen maken met de data. Dit zal in het bijhorende hoofdstuk meer verduidelijkt worden.

API toepassing

De laatste toepassing die ik benaderd en bestudeerd heb is om een externe API te koppelen aan de AWS omgeving om zo een constante stroom van data te creëren. Welke data en van waar deze werd opgehaald kan u vinden in het bijhorende hoofdstuk.

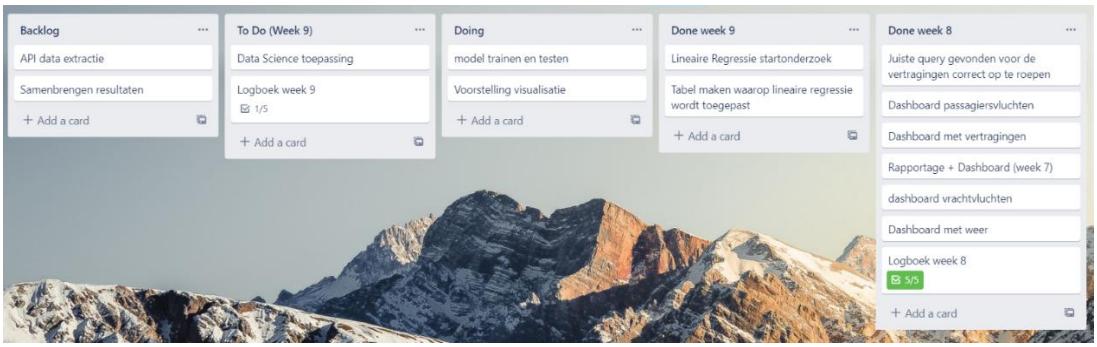
3. Realisatie

3.1 Opstart van de stage

Als eerste heb ik alle nodige communicatietools in orde gebracht zodat mijn mentor van Ordina en mijn begeleider van Thomas More via verschillende bronnen kon volgen waar ik nu juist aan bezig was en wat er nog op de planning stond.

Trello

De eerste tool die we gebruikt hebben om een uitgebreide planning te maken van alle componenten die aanbod zouden komen in de opdracht was Trello. Trello is een planningstool waar je gemakkelijk planningsborden kan maken zoals op de afbeelding hieronder getoond wordt. Hieronder ziet u een voorbeeld van mijn Trellobord in week 9 van de stage:



In de backlog stonden alle onderdelen die aan bod zouden komen tijdens de stage. Deze waren zeer algemeen dus deze werden in de weken wanneer aan een onderdeel gewerkt werd uit de backlog nog verder opgedeeld in kleinere componenten die dan samen een groot onderdeel van het geheel van de opdracht vormden.

GitHub

Een tweede tool die we hebben gebruikt is GitHub. GitHub is gebruikt geweest voor zowel versiebeheer en zodat mijn stagementor indien dat gewenst was ook de door mij geschreven code zou kunnen bekijken en opvolgen.

Logboek

Ik heb ook een logboek gemaakt dat wekelijks opgeleverd moest worden aan de begeleider van Thomas More waarin dagelijkse updates van de opdracht in neergeschreven werden zodat ook de begeleider van Thomas More kon volgen wat ik had gepresteerd elke week.

MS Teams

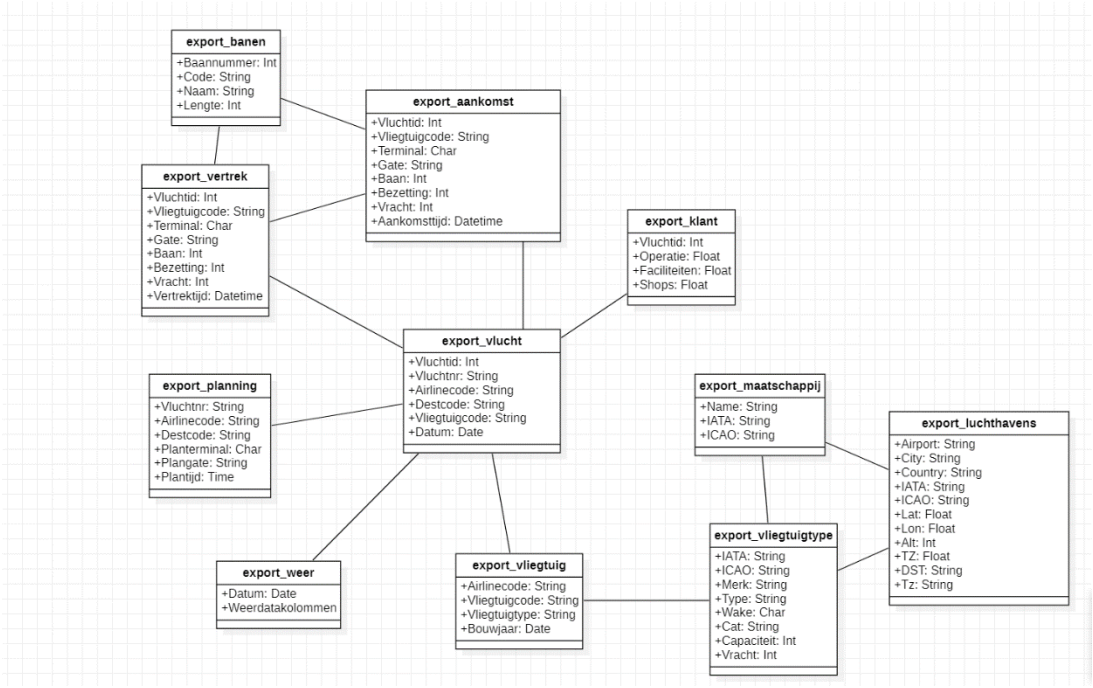
Microsoft Teams werd gebruikt als eerste communicatielijn met mijn stagementor van Ordina. Ik kon dus elke dag rechtstreeks bij mijn stagementor terecht indien ik vragen of problemen had. We hebben MS Teams ook gebruikt om wekelijks een meeting te houden om in meer details te bespreken wat ik in de voorbije week gerealiseerd had. Daarna werd besproken wat er de volgende week op de planning stond.

3.2 Data exploratie

In deze fase van de opdracht De oorspronkelijke dataset waarmee ik van start moest gaan staat op de figuur hieronder:

export_aankomst.txt
export_banen.csv
export_klant.csv
export_luchthavens.txt
export_maatschappijen.txt
export_planning.txt
export_vertrek.txt
export_vliegtuig.txt
export_vliegtuigtype.csv
export_vlucht.txt
export_weer.txt

Zoals u kunt zien bestond de oorspronkelijke dataset uit twee verschillende bestandsformaten, .csv en .txt. De namen van de bestanden verraden ook dat de dataset allerlei informatie bevatte over vluchten. De dataset bevat namelijk vluchtdata van de fictieve luchthaven VisionAirport. Het is allemaal historische data van het jaar 2014 tot het jaar 2018. Elk bestand bevatte exact één tabel waarin data zat in verband met de naam van het bestand. Ik heb elk bestand grondig bekeken en voor elke tabel opgeschreven welke kolommen en welke bijhorende data in elk bestand zaten. Nadat ik exact wist welke tabellen en kolommen er waren heb ik een datamodel gemaakt om een duidelijk overzicht te hebben van de verschillende tabellen, kolommen en hoe deze aan elkaar gelinkt waren.
Zie het datamodel op de figuur hieronder:



Vanaf dit punt had ik een beter beeld van al de data ter beschikking en kon ik ook helder nadenken hoe ik deze in de toekomst zou verwerken en modelleren.

3.3 Het Big Data platform

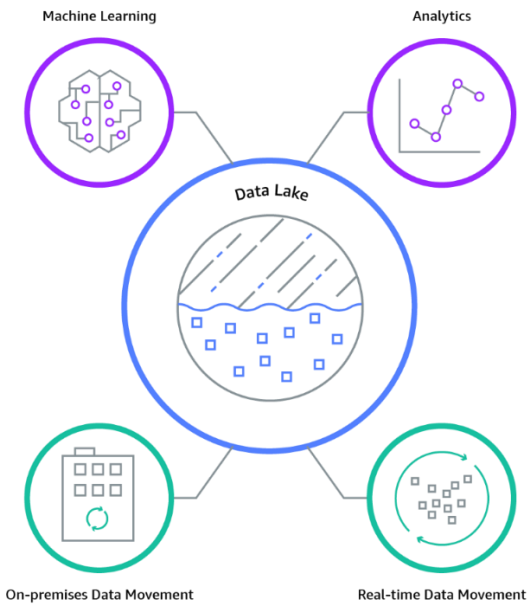
Het volgende onderdeel van de stageopdracht was het bouwen van het Big Data platform waar de data naartoe gestuurd moest worden en worden opgeslagen, een centrale opslagplaats voor de data voorzien eigenlijk. Er zijn in de vandaag moderne wereld van IT al een uitgebreid aantal oplossingen om zo'n platform te bouwen. Maar er zijn natuurlijk voor elke soort groepering van data aantrekkelijke en minder aantrekkelijke oplossingen. De meest aantrekkelijke keuze voor de dataset die ik ter beschikking had was een Data Lake

3.3.1 Wat is een Data Lake?

Een Data Lake is een soort van centrale opslagplaats waar allerlei soorten data, gestructureerd en ongestructureerd, kunnen bewaard worden op welke schaal je maar wil. De gegevens kunnen worden opgeslagen zonder ze eerst te structureren. Dankzij de werking van een Data Lake moet je dus geen rekening houden met op voorhand bepaalde schema's en kan je zonder zorgen je oorspronkelijke data dumpen in het Data Lake. Dat bespaart bedrijven enorm veel voorbereidingswerk en onnodige moeilijkheden alvorens ze gegevens ergens zo snel mogelijk willen dumpen. Dit was voor mijn situatie dan ook zeer interessant omdat mijn dataset ook uit verschillende bestandsformaten bestond en aan geen op voorhand bepaald schema heeft rekening gehouden.

Er zijn meerdere manieren om een Data Lake op te zetten. Dit kan lokaal worden gedaan indien dit gewenst zou zijn, maar evenzeer en zelfs gemakkelijker in de cloud. Zowel Amazon als Microsoft en andere cloud providers bieden in hun cloud services uitgebreide oplossingen aan om een Data Lake op te bouwen. In de cloud kunnen Data Lakes ook gemakkelijk gelinkt worden aan andere datatoepassingen. Deze toepassingen kunnen visualisatietools zijn waarop gegevens op gewenste manieren kunnen gevisualiseerd worden op dashboarden, maar dit kan ook een Analytics tool zijn of eender welke toepassing er beschikbaar is binnen de cloud provider. Dit is ook weer interessant voor onze dataset die later in het verhaal ook moest dienen voor visualisatie en voor andere toepassingen.

Een voorbeeld van een schematische schets van een Data Lake en gelinkte toepassingen kunt u zien op de afbeelding hieronder:



3.3.4 Data Lake op AWS

De AWS Cloud biedt veel van de bouwstenen aan die nodig zijn om klanten te helpen een veilig, flexibel en kosteneffectief Data Lake te implementeren. De Data Lake on AWS-oplossing bewaart en registreert datasets van elke grootte in hun oorspronkelijke vorm in het veilige, duurzame en zeer schaalbare Amazon S3.

Amazon S3

Amazon S3, ook bekend als Amazon Simple Storage Service, is een opslagruimte voor elke soort en hoeveelheid data dat je maar kan bedenken. Je kan Amazon S3 gebruiken om eender welke hoeveelheid gegevens op te slaan en op te vragen op eender welk moment, van eender waar op het web. Hoe gaat dat in zijn werk? In een notendop, Amazon S3 slaat om het even welke hoeveelheid gegevens op en haalt ze terug door gebruik te maken van zeer betrouwbare, snelle, goedkope en schaalbare gegevensopslag.

Amazon S3 is een opslagservice die data bewaart als objecten in buckets. Een object is een bestand en alle metadata die het bestand omschrijft zoals in ons geval de .csv en .txt bestanden met al hun data in. Een bucket dient als een container voor objecten. Je kunt het eigenlijk vergelijken met een geavanceerd versie van je eigen mappenstructuur op je PC.

Layers

Een Data Lake is in de meeste gevallen opgedeeld in de volgende lagen: een Landing Zone, een Raw Layer en een Cleansed Zone/Layer. Deze zorgen voor een correct onderscheid tussen verschillende soorten data en waarvoor de data gebruikt wordt. Elk van deze lagen of layers heeft zijn eigen functie.

- ✓ ***Landing Zone:***

Deze layer van het Data Lake dient gewoonweg als dumplocatie voor alle soorten bestanden die u wil bewaren in het Data Lake. U moet niets aanpassen of speciaal configureren aan de oorspronkelijke dataset. Deze layer zorgt ervoor dat de oorspronkelijke dataset een opslagplaats heeft zodat deze altijd zou kunnen worden opgeroepen als dat gewenst zou zijn.

- ✓ ***Raw Layer:***

Deze layer van het Data Lake dient als een source of truth van het Data Lake. In deze laag worden vaak de oorspronkelijke bestanden van de dataset bewaart, maar wel omgezet naar eenzelfde bestandsformaat zoals bijvoorbeeld .csv of .parquet. Ook kunnen andere data best practices in deze laag al toegepast worden. Voorbeelden hiervan zijn het verwerken van corrupte gegevens, zorgen voor correcte nul-waarden, correcte tabel en kolombenamingen en structuren maken in de bestanden waar dit nog niet het geval is enzovoort. Zo kunt u van een propere lei beginnen als u een nieuw datamodel wil creëren van de dataset en ook altijd terugkeren naar de propere versie van de oorspronkelijke dataset indien dat gewenst is.

- ✓ ***Cleansed Zone:***

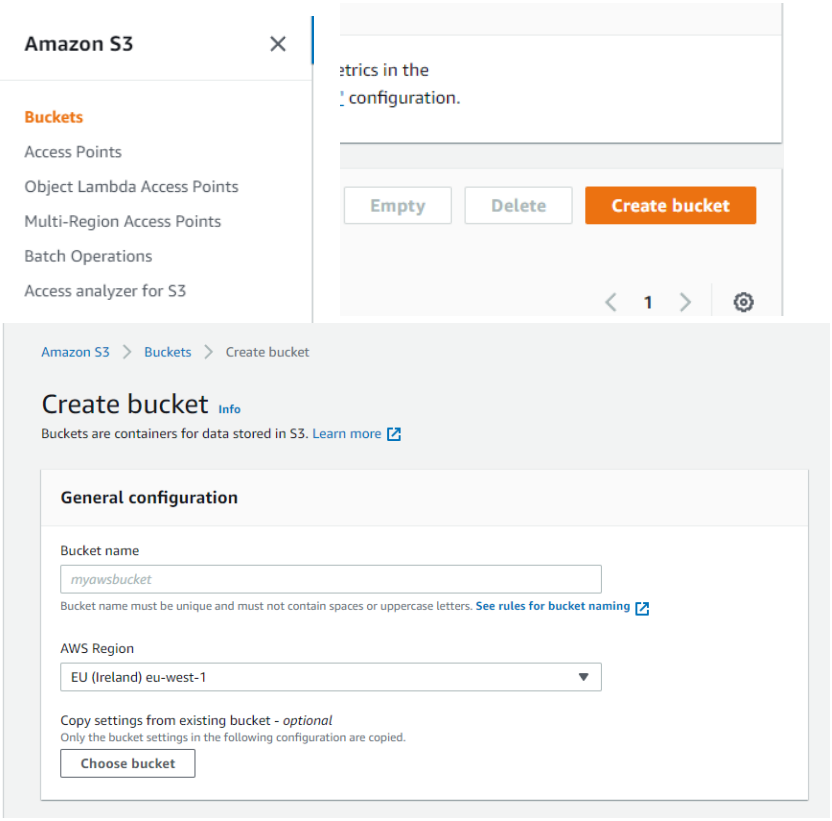
Dit is de laatste zone van het Data Lake waar de nieuwe dataset van het nieuwe datamodel in zit. Deze data is zo verwerkt dat het kan voldoen aan de nodige businessdoeleinden en waar rapportage en andere datatoepassingen zonder problemen op kunnen worden uitgevoerd op een efficiënte manier.

3.3.5 Wat is er gerealiseerd?

Als u een Data Lake op AWS wil opzetten kan dat op meerdere manieren. Ikzelf heb er namelijk drie geprobeerd en uitgetest.

GUI van AWS

De eerste manier dat ik gebruikt heb om te testen hoe ik het Data Lake kon opzetten was eerst manueel met behulp van de grafische interface van AWS. Dit is veruit de gemakkelijkste en snelste manier als je maar een kleine Data Lake structuur zou hebben. U moet gewoonweg een paar dingen aanklikken en klaar is kees. Maar als u Data Lake in grootte en complexiteit toeneemt wordt dit steeds meer problematischer om dit met de grafische interface te doen.



Python

Met behulp van python scripts kan je ook zeer snel en makkelijk S3 buckets en files aanmaken. Python heeft enkele libraries die heel goed samenwerken met AWS en een vlotte verbindingen legt tussen de python omgeving en de AWS omgeving.

Voorbeeld van python code om een bucket, folder en een lokale file uit de dataset in S3 te krijgen:

```python
#imports and variables
import boto3
import os

s3_resource = boto3.resource("s3")
s3_client = boto3.client("s3")
```

```python
#Functions:

#creation bucket data-lake-yari
def create_bucket(s3_bucket_name, region):
    s3_resource.create_bucket(Bucket=s3_bucket_name, CreateBucketConfiguration={"LocationConstraint": region})
    return "Bucket", s3_bucket_name, "is created!"

#creation of an object (folder) in s3 bucket
def folder_in_s3(s3_bucket_name, folder_name):
    s3_client.put_object(Bucket=s3_bucket_name, Key=(folder_name + '/'))
    return "Folder", folder_name, "has been created in", s3_bucket_name + "!"

#upload file to a folder
def upload_file_to_s3(s3_bucket_name, local_file_path, s3_file_path):
    s3_resource.Bucket(s3_bucket_name).upload_file(local_file_path, s3_file_path)
    return s3_file_path, "is uploaded to", s3_bucket_name + "."
```

```python
create_bucket("yari-test-functie", "eu-west-1")
```

```
('Bucket', 'yari-test-functie', 'created.')
```

```python
folder_in_s3("yari-test-functie", "test-folder")
```

```
('Folder', 'test-folder', 'has been created in', 'yari-test-functie!')
```

```python
upload_file_to_s3("yari-test-functie", "../datasets_visionairport/export_banen.csv", "test-folder/test_file.csv")
```

```
('test-folder/test_file.csv', 'is uploaded to', 'yari-test-functie.')
```

Terraform

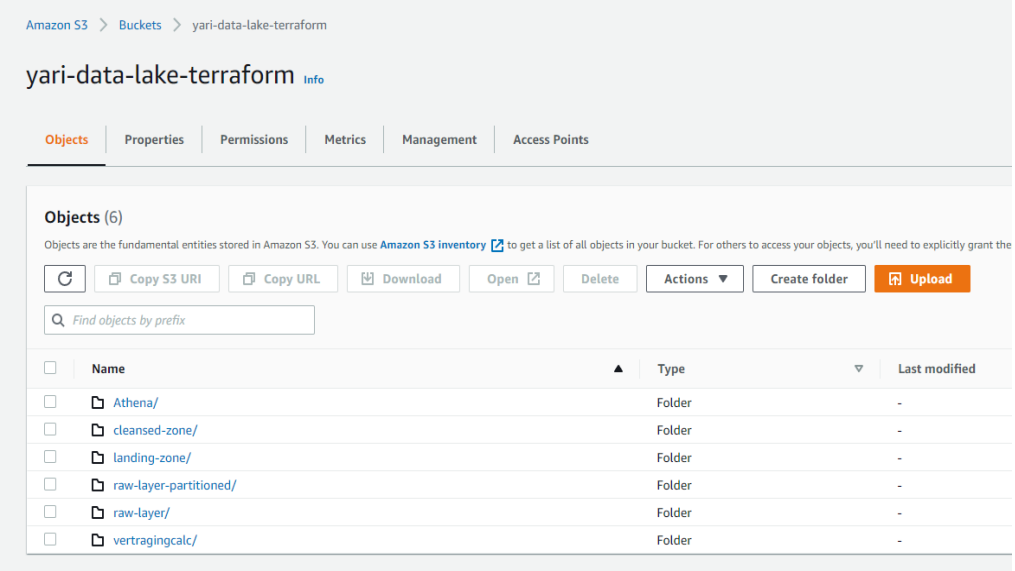
In de moderne wereld van IT wordt steeds meer gebruik gemaakt van Infrastructure as Code tools. Zo kan er met behulp van Terraform een configuratie file worden aangemaakt in human-readable formaat waarin heel de configuratie van het Data Lake neergeschreven is en met enkele commando's opgezet kan worden en ook weer afgebroken worden. Dit is een enorme verbetering in efficiëntie aangezien de code in veel gevallen in grote lijnen hetzelfde is enkel unieke namen moeten worden aangepast. Dit versnelt dus enorm het proces van het opzetten en ook afbreken van Data Lakes. Dit is ook de laatste manier dat ik heb uitgetest om mijn Data Lake op te zetten in S3.

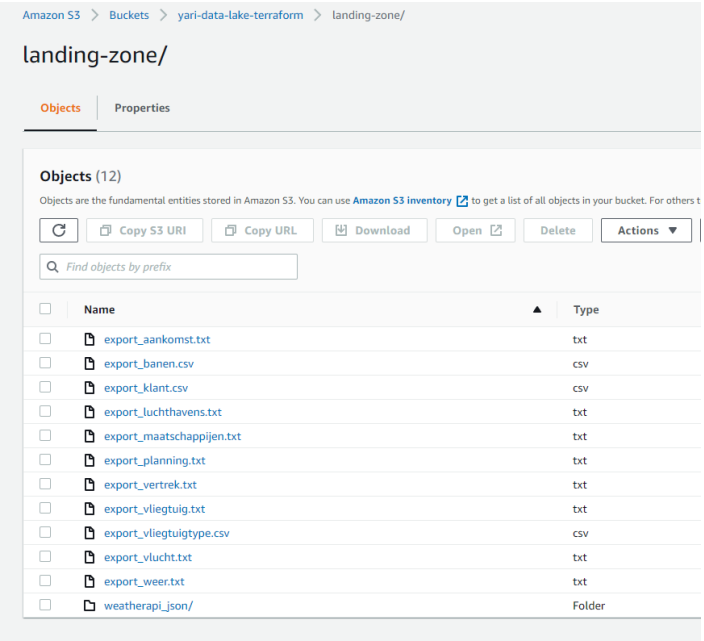
In onderstaande foto ziet u de configuratie file om een nieuwe S3 bucket en folders aan te maken op AWS. Tenslotte wordt de dataset die lokaal in een map op mijn PC zit ook mee doorgestuurd naar de Landing Zone van het Data Lake die diende als dump folder voor alle bestanden in hun oorspronkelijke vorm.

```
# AWS credentials from folder aws
provider "aws" {
  profile = "default"
  region = "eu-west-1"
}

# AWS S3 bucket
resource "aws_s3_bucket" "data-lake-bucket" {
  bucket = "yari-data-lake-terraform"
}

# Folder in S3 bucket
resource "aws_s3_object" "landing-zone-folder" {
  bucket = aws_s3_bucket.data-lake-bucket.id
  key = "landing-zone/"
}

resource "aws_s3_object" "raw-layer-folder" {
  bucket = aws_s3_bucket.data-lake-bucket.id
  key = "raw-layer/"
}

resource "aws_s3_object" "cleansed-zone-folder" {
  bucket = aws_s3_bucket.data-lake-bucket.id
  key = "cleansed-zone/"
}

# Upload files to S3 bucket
resource "aws_s3_bucket_object" "upload_source_files_to_folder" {
  for_each = fileset("C:\\Users\\YaVa\\OneDrive - Ordina\\Bureaublad\\Stageopdracht_Yari_Big_Data\\Terraform\\Datasets-VisionAirport","*")
  bucket = aws_s3_bucket.data-lake-bucket.id
  key = "landing-zone/${each.value}"

  source = "C:\\Users\\YaVa\\OneDrive - Ordina\\Bureaublad\\Stageopdracht_Yari_Big_Data\\Terraform\\Datasets-VisionAirport\\${each.value}"
  etag = filemd5("C:\\Users\\YaVa\\OneDrive - Ordina\\Bureaublad\\Stageopdracht_Yari_Big_Data\\Terraform\\Datasets-VisionAirport\\${each.value}")
}
```

Eindresultaat Data Lake infrastructuur

Op onderstaande foto's ziet u enkele voorbeelden van de infrastructuur van mijn Data Lake. Er zijn hier natuurlijk nog meer folders aanwezig dan diegene die al besproken zijn. Deze worden verder toegelicht in de volgende hoofdstukken.





Toen ik dit gerealiseerd had kon ik beginnen met de oorspronkelijke files uit de dataset te gaan verwerken en opschonen zodat deze klaar waren voor in de volgende lagen van het Data Lake te komen, de Raw Layer en tenslotte in de Cleansed Layer.

3.4 Data Engineering en Modellering

Nu dat de infrastructuur van het Data Lake was opgezet kon ik beginnen met het verwerken van de data uit de oorspronkelijke dataset en deze klaar maken voor naar de volgende layers van het Data Lake te sturen, de Raw en Cleansed Layer. Elk van deze layers hebben andere doeleinden.

Er zijn meerdere manieren om aan data engineering te doen. Zo zijn er meerdere programmeertalen of tools om hiermee aan de slag te gaan. Ik heb gewerkt met de programmeertaal Python, een zeer beginnersvriendelijke programmeertaal met veel behulpzame python libraries zoals Python Pandas. Die python libraries zoals Python Pandas maakt het zeer gebruiksvriendelijk om tabellen en kolommen op te roepen en te bewerken naar eigen wens. Er zijn ook heel veel behulpzame documentatie en voorbeelden te vinden op het internet over deze libraries.

3.4.1 Raw Layer

De Raw Layer wordt dus gebruikt als opslagplaats waar data als source of truth wordt opgeslagen zoals in het hierboven toegelichte hoofdstuk. Nu hoe heb ik dat nu gedaan met mijn dataset?

Realisatie Raw Layer

Ik ben eerst gestart met enkele best practices toe te passen op de tabellen en kolommen. Dit allemaal met behulp van python libraries zoals Pandas. Ik ben begonnen met elk bestand in te lezen in mijn python omgeving, Jupyter Notebooks.

De eerste aanpassing die ik heb uitgevoerd aan de dataset is het aanpassen van het bestandsformaat van elk bestand naar .parquet en deze dan verder geüpload naar de raw-layer folder in de S3 bucket op AWS. Ik had dit gedaan om latere eventuele format issues te vermijden. Zie een voorbeeld van de code hieronder:

```
#Functions:

#download file from landing-zone in s3
def download_and_read_file(file_key):
    obj = s3_client.get_object(Bucket=bucketname, Key=file_key)
    df = pandas.read_csv(io.BytesIO(obj['Body'].read()), sep=';', header=0, low_memory=False)
    return df

#convert .csv to .parquet and upload to s3 raw-layer
def convert_and_upload_parquet_file(s3_uri):
    s3_url = s3_uri
    new_parquet_df.to_parquet(s3_url, compression='snappy')
    print('The parquet file is uploaded to the raw layer.')
```

```
#Executing functions:

#update df variable
new_parquet_df = download_and_read_file(".csv file in landing-zone")

convert_and_upload_parquet_file("s3://bucketname/foldername/new_file.parquet")
```

```
The parquet file is uploaded to the raw layer.
```

De volgende best practice die ik heb toegepast op de dataset is alle kolomnamen lowercase maken. Dit wordt in de meeste gevallen altijd gedaan zodat er geen onduidelijkheden voorkomen in de spelling van een kolomnaam (voorbeeld is bijvoorbeeld de kleine letter l en hoofdletter i = I). Dit kan eventuele problemen veroorzaken als de correcte benaming niet overal gebruikt wordt. Zie een voorbeeld van de gebruikte code hieronder:

```
#Functions:

#downloading and reading parquet file from AWS
def download_and_read_parquet_file(bucketname, filekey):
    buffer = io.BytesIO()
    object = client.Object(bucketname, filekey)
    object.download_fileobj(buffer)

    df = pd.read_parquet(buffer)
    display(df)
    return df

#columns to lowercase
def lowercase_columns(df):
    df.columns = df.columns.str.lower()
    print("The columns have changed to lowercase!")
    display(df)
    return df

#uploading the updated parquet file to AWS
def uploading_file_to_aws(s3_uri):
    final_df.to_parquet(s3_uri, compression="snappy")
    print("The updated file is uploaded to", s3_uri + "!")
```

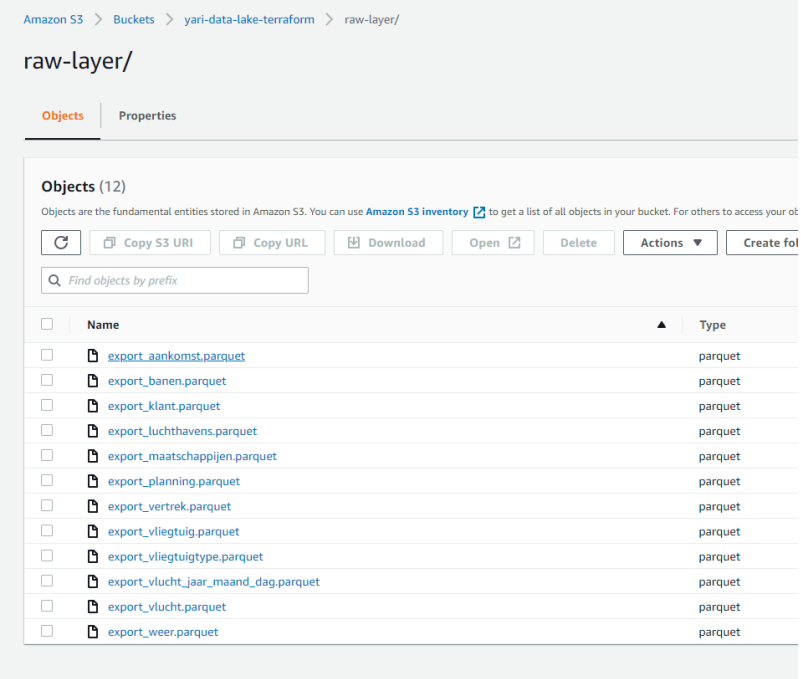
```
#Executing functions:

#update df variable
downloaded_df = download_and_read_parquet_file(bucketname, "file key")

#update df variable again
final_df = lowercase_columns(downloaded_df)

#execute final function to upload the updated file to AWS
uploading_file_to_aws("s3 URI")
```

De laatste best practice die ik in deze fase had toegepast was het verwijderen van kolommen met dezelfde namen in een tabel. Er waren enkele kolomnamen wanneer deze naar lowercase gebracht werden dezelfde spelling kregen, vandaar dat ik sommige kolomnamen heb aangepast naar een synoniem van wat er eerst stond. Hieronder is een voorbeeld van de code die ik daarvoor gebruikt heb:

```
#Functions:

#downloading and reading parquet file from AWS
def download_and_read_parquet_file(bucketname, filekey):
    buffer = io.BytesIO()
    object = client.Object(bucketname, filekey)
    object.download_fileobj(buffer)

    df = pd.read_parquet(buffer)
    display(df)
    return df

def renaming_column(df, old_column, new_column):
    df.rename(columns={old_column : new_column}, inplace=True)
    print(old_column,"has changed to", new_column + ".")
    display(df)
    return df

#uploading the updated parquet file to AWS
def uploading_file_to_aws(s3_uri):
    final_df.to_parquet(s3_uri, compression="snappy")
    print("The updated file is uploaded to", s3_uri + "!")
```

```
#Executing functions:

#update df variable
downloaded_df = download_and_read_parquet_file(bucketname, "file with duplicate columns")

#update df variable again
updated_df = renaming_column(downloaded_df, "old column name", "new column name")

#update df variable again until you don't have to change column names anymore
final_df = renaming_column(updated_df, "old column name", "new column name")

#execute final function to upload the updated file again to AWS
uploading_file_to_aws("s3 URI")
```

Hieronder ziet u ook een foto van de .parquet files in de S3 bucket op AWS in de raw-layer folder:



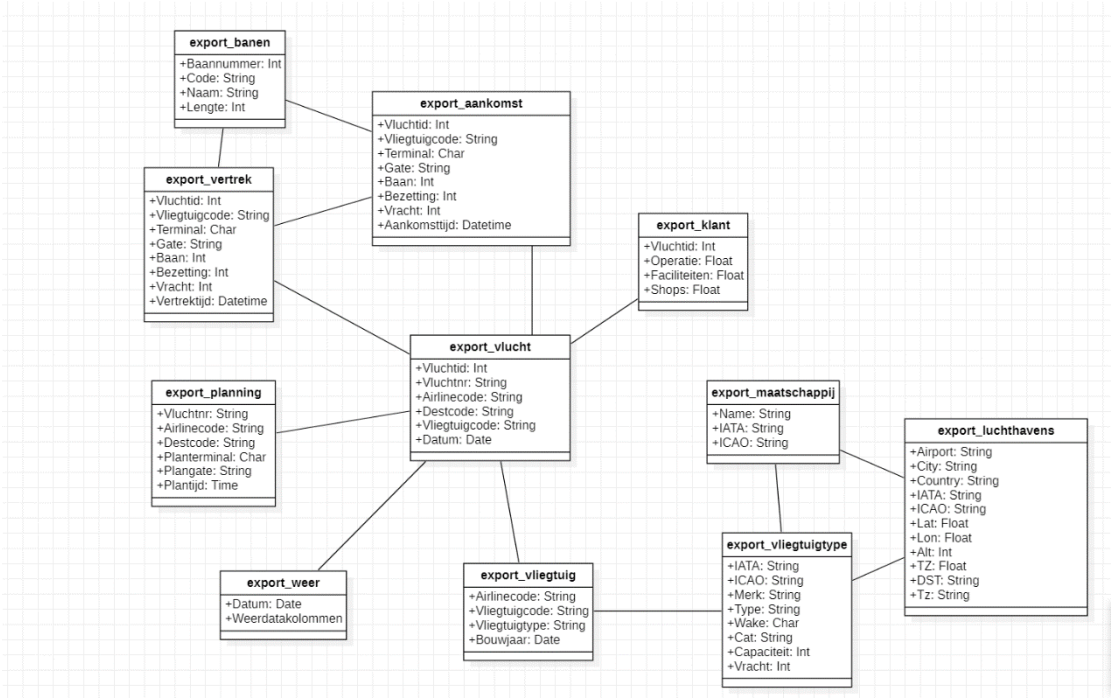
Door deze aanpassingen te doen had ik een nieuwe propere dataset om een nieuw datamodel te vormen in de Cleansed Zone van het Data Lake.

3.4.2 Cleansed Zone/Layer

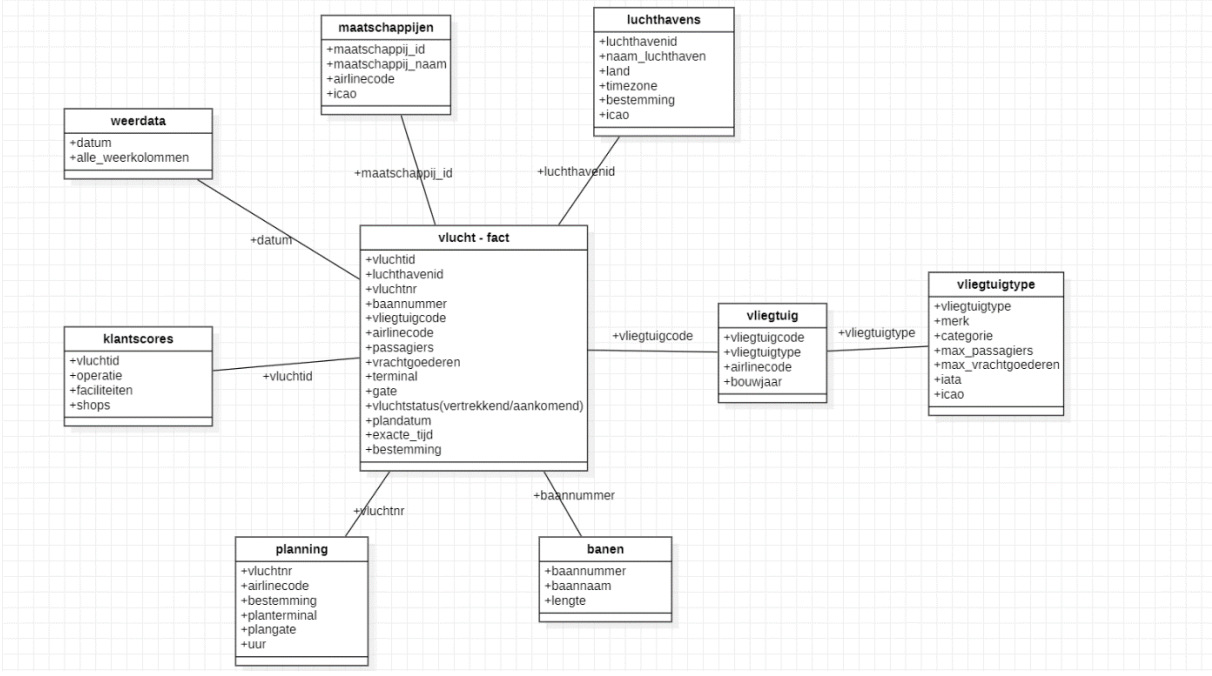
Tijdens dit onderdeel was de grote uitdaging om het nieuwe datamodel te ontwerpen. Ik moest dus van de oorspronkelijke dataset een nieuwe dataset maken die bijvoorbeeld aan in dit geval fictieve businessdoeleinden van de luchthaven zou moeten voldoen. Deze nieuwe dataset zou andere datatoepassingen zoals visualisatie, analyses, Data Science… mogelijk moeten maken.

Datamodel

Ik ben dan beginnen nadenken en uittekenen wat in ons geval een goede upgrade zou zijn van de dataset. Hieronder kan u het datamodel zien dat ik gemaakt had van de oorspronkelijke dataset (zonder aanpassingen aan de originele dataset):



Dit is het nieuwe datamodel:



Het nieuwe model volgt het principe van een snowflake model. Meestal staat er één fact tabel centraal (dit kunnen er in bepaalde gevallen ook meer zijn) in dit model waaruit alle andere tabellen verder gaan en eigenlijk een sneeuwvlokje als figuur vormen. De tabellen aan de buitenkant kunnen altijd nog een keer uitgebreid worden zoals bij de tabel *vliegtuig* die nog verder gaat naar *vliegtuigtype*. Er zijn dus heel wat tabellen aangepast en samengevoegd om de dataset wat compacter en efficiënter te maken.

Realisatie Cleansed Zone

Ik heb dus wel heel wat aanpassingen aangebracht aan de originele dataset. Ik ga niet van elke tabel tonen hoe dat ik dat heb gedaan, maar ik ga wel de grootste en moeilijkste tabel toelichten, namelijk de fact tabel.

De fact table, in het datamodel wordt deze vermeld als de tabel *vlucht*, is samengesteld door informatie samen te voegen vanuit drie tabellen uit de originele dataset: export_vlucht, export_vertrek en export_aankomst. Dit is dus de centrale tabel waaruit het nieuwe snowflake model is opgebouwd en alle andere tabellen met elkaar verbindt. In een fact table worden vaak de belangrijkste algemene en numerieke gegevens bewaard. De fact table die ik gemaakt heb bestaat namelijk uit allerlei algemene gegevens die bij een vlucht horen en ook een paar interessante cijfers die bij de vlucht horen, bijvoorbeeld: belangrijke ID's die linken naar andere tabellen, vertrek en aankomsttijd, aantal passagiers/vrachtgoederen, of het een vertrekkende vlucht of aankomende vlucht is, de geplande tijd van de vlucht, enzovoort…

Gebruikte code om de fact table te maken:

```
import pandas as pd
import pyarrow.parquet as pq
import s3fs
import awswrangler as wr
import boto3
import numpy as np
import io
import datetime as dt
import warnings
warnings.filterwarnings('ignore')

s3 = s3fs.S3FileSystem()
bucketname = "yari-data-lake-terraform"
client = boto3.resource("s3")
```

```
#downloading en reading parquet file van AWS
def download_and_read_parquet_file(bucketname, filekey):
    buffer = io.BytesIO()
    object = client.Object(bucketname, filekey)
    object.download_fileobj(buffer)

    df = pd.read_parquet(buffer)
    return df

#verander kolom records van float naar int
def column_float_to_int(df ,column_name):
    df[column_name] = df[column_name].fillna(0).astype(int)
    return df

#verander datatype kolom naar dateformat
def object_to_dateformat(df, column_name):
    df[column_name] = pd.to_datetime(df[column_name], errors='coerce')
    return df
```

```
#download parquet file from partitioned raw layer
df_export_vlucht = wr.s3.read_parquet(path='s3://yari-data-lake-terraform/raw-layer-partitioned/export_vlucht/', dataset=True)
df_export_vertrek = wr.s3.read_parquet(path='s3://yari-data-lake-terraform/raw-layer-partitioned/export_vertrek/', dataset=True)
df_export_aankomst = wr.s3.read_parquet(path='s3://yari-data-lake-terraform/raw-layer-partitioned/export_aankomst/', dataset=True)
```

```
#download other needed parquet files
df_luchthavens = download_and_read_parquet_file(bucketname, 'cleansed-zone/luchthavens/luchthavens.parquet')
```

```
df_fact_main = df_export_vlucht
```

```
#rename to make the merge between luchthavens and main table possible
df_fact_main = df_fact_main.rename(columns={'destcode': 'bestemming'})
```

```
#merge luchthaven_id into the main table
df_fact_main = df_fact_main.merge(df_luchthavens[['bestemming', 'luchthaven_id']])
```

```
#determine the type of the flight
df_fact_met_vluchtstatus = df_fact_main.assign(vluchtstatus=df_fact_main['vluchtid'].isin(df_export_vertrek['vluchtid']))

replace_booleans = {True: 'vertrekkende vlucht', False: 'aankomende vlucht'}
df_fact_met_vluchtstatus['vluchtstatus'] = df_fact_met_vluchtstatus['vluchtstatus'].map(replace_booleans)
```

```
#merge vertrekdata into the main table
df_fact_met_vertrekdata = df_fact_met_vluchtstatus.merge(df_export_vertrek[['vluchtid', 'vertrektijd', 'terminal', 'gate', 'baan', 'bezetting', 'vrach
```

```
#merge of the two tables to add the needed columns to the fact table
df_fact_met_aankomstdata = pd.merge(df_fact_met_vertrekdata, df_export_aankomst, on='vluchtid', how='left')
```

```
#merge all duplicate columns
df_fact_met_aankomstdata['vertrektijd'].fillna(df_fact_met_aankomstdata['aankomsttijd'], inplace=True)
df_fact_met_aankomstdata['terminal_x'].fillna(df_fact_met_aankomstdata['terminal_y'], inplace=True)
df_fact_met_aankomstdata['gate_x'].fillna(df_fact_met_aankomstdata['gate_y'], inplace=True)
df_fact_met_aankomstdata['baan_x'].fillna(df_fact_met_aankomstdata['baan_y'], inplace=True)
df_fact_met_aankomstdata['bezetting_x'].fillna(df_fact_met_aankomstdata['bezetting_y'], inplace=True)
df_fact_met_aankomstdata['vracht_x'].fillna(df_fact_met_aankomstdata['vracht_y'], inplace=True)
```

```
#drop all unnecessary columns
df_fact_met_aankomstdata.drop(['vliegtuigcode_y', 'aankomsttijd', 'bezetting_y', 'vracht_y', 'terminal_y', 'gate_y', 'baan_y', 'gepland', 'jaar_y', 'd
```

```
#give all columns userfriendly names
df_fact_rename_columns = df_fact_met_aankomstdata.rename(columns={'bezetting_x': 'bezetting', 'vracht_x': 'vracht', 'vliegtuigcode_x': 'vliegtuigcode'
```

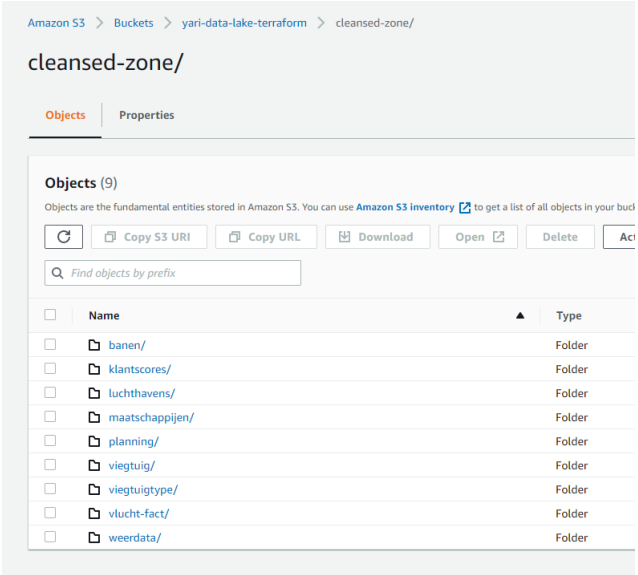
```
df_fact_final_table = df_fact_rename_columns[['vluchtid', 'vluchtnr', 'airlinecode', 'bestemming', 'luchthaven_id', 'vliegtuigcode', 'vluchtstatus', '
```

```
df_fact_final_table.to_parquet(
    path="s3://yari-data-lake-terraform/cleansed-zone/vlucht-fact/",
    compression="snappy",
    partition_cols=["jaar", "maand", "dag"]
)
```

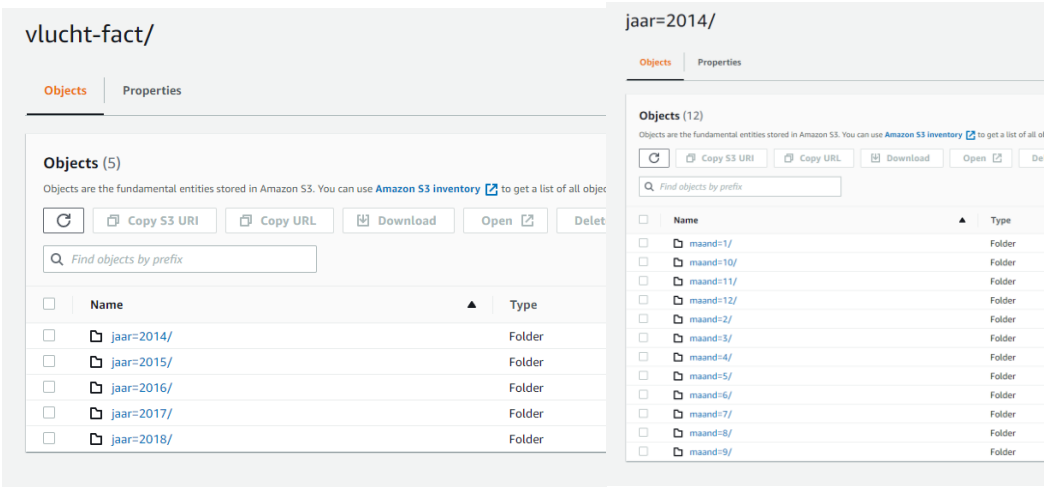
Deze tabel bevatte uiteindelijk dan ook het meest aantal records, bijna 800 duizend. Als er grote hoeveelheden data in een tabel zitten wordt deze tabel in de meeste gevallen gepartitioneerd. Dat wil zeggen dat er een opdeling wordt gemaakt van het totaal aantal gegevens in een tabel op basis van een of enkele parameters. Vaak hoort er bij verschillende soorten gegevens altijd een of andere tijdseenheid of datum. Een veel gebruikte partitioneringsmethode is dan ook het opdelen van de data per datum. Zo heb ik in mijn voorbeeld hierboven de data opgedeeld op basis van jaar, maand en dag. Het resultaat hiervan was dat het ene grote bestand met 800 duizend records in werd verdeeld over allemaal deelbestandjes gesorteerd onder de bijhorende datum per record. Dit brengt als voordeel met zich mee dat als u data wil oproepen van een bepaalde periode gaat er eerst gezocht worden naar de map met de correct bijhorende datumgegevens van de opgevraagde periode en geeft u dan die bijhorende data. Als er niet gepartitioneerd wordt moet u elke keer heel het bestand opvragen wat natuurlijk extra verbruik, tijd en kosten met zich meebrengt hoe groter de dataset is die wordt opgevraagd. Vandaar dat ik de grote fact table heb opgedeeld in jaar, maand en dag.

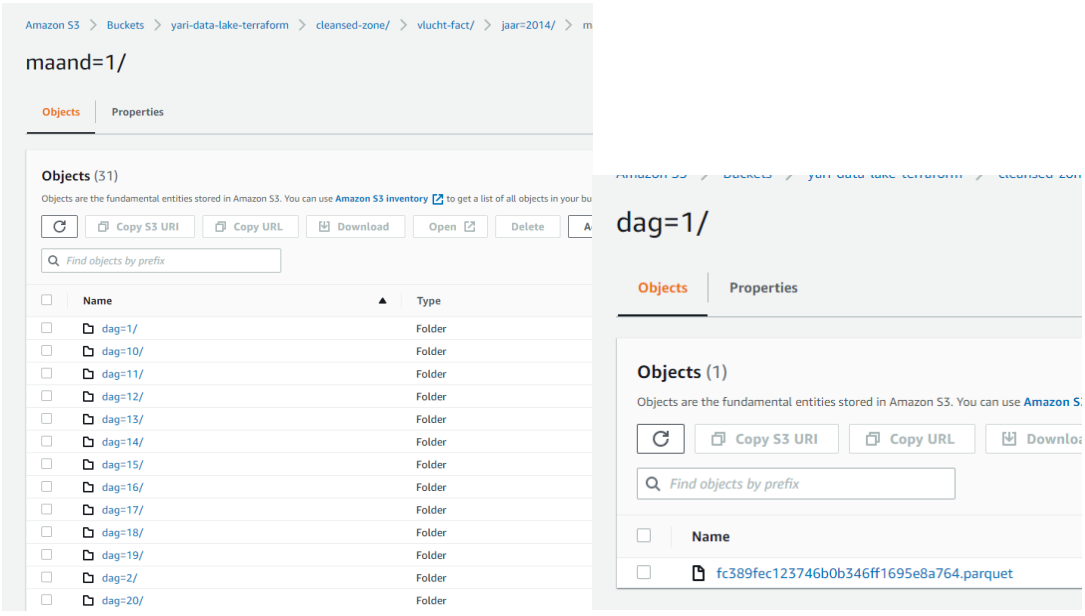
Resultaat in het Data Lake op AWS

Hieronder ziet u een foto van de Cleansed Zone van het Data Lake:



Hieronder ziet u voorbeelden van de partitionering van de *vlucht-fact* tabel:





3.5 Visualisatie in AWS QuickSight

Toen dat het Data Lake volledig gevuld was en de data correct was omgezet volgens het nieuwe datamodel kon ik beginnen met het opzetten van de eerste uitbreiding op het Data Lake, namelijk data visualiseren op dashboards in AWS QuickSight. Dit kon ik niet meteen doen zonder enige voorbereiding.

3.5.1 AWS Athena

Wat is AWS Athena?

Amazon Athena is een interactieve query service. Deze service maakt het zeer gemakkelijk om rechtstreeks data uit Amazon S3 buckets te analyseren met behulp van SQL. Met maar een paar uit te voeren acties in de AWS Management Console, kan u Athena verbinden met uw data in Amazon S3 en hier standaard SQL queries op uitvoeren. Resultaten worden zichtbaar in no time.

Athena is serverless, er is dus geen infrastructuur nodig om Athena te configureren of beheren, en u betaalt enkel voor de queries die uitgevoerd worden. Athena heeft automatische schaalbaarheid, resultaten worden zeer snel gevonden en getoond, zelfs met grote datasets en complexe queries.

Waarom AWS Athena gebruiken?

Athena is zeer behulpzaam en effectief om unstructured, semi-structured en structured data opgeslagen in S3 te analyseren. Bijvoorbeeld CSV, JSON of kolomsgewijze dataformaten zoals Apache Parquet. Athena kan gebruikt worden om SQL queries uit te voeren zonder dat eerst al de data opgeslagen in S3 moet worden ingeladen in de Amazon Athena omgeving.

Athena integreert met de AWS Glue Data Catalog, die een persistente metadata opslag biedt voor uw data in Amazon S3. Dit stelt u in staat om tabellen aan te maken en data te bevragen in Athena op basis van een centrale metadata store die beschikbaar is in uw gehele Amazon Web Services account en geïntegreerd is met de ETL en data discovery functies van AWS Glue.

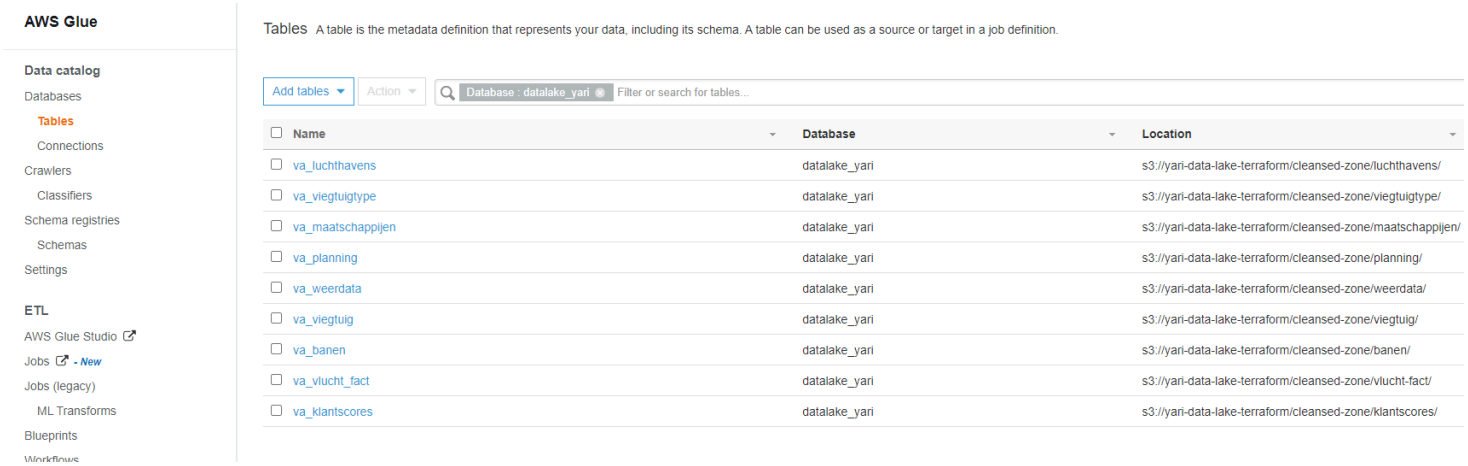
Athena werkt ook goed samen met Amazon QuickSight zodat we de opgevraagde data in Athena gemakkelijk kunnen visualiseren indien nodig.

Amazon Athena maakt het simpel om interactieve queries uit te voeren rechtstreeks op data uit S3 buckets zonder dat het de data moet herformatteren of aparte infrastructuur moet gebruiken om te werken. Dus als u geen extra infrastructuur en clusters wil opzetten en beheren. Het is dus enorm gemakkelijk om SQL queries uit te voeren op data opgeslagen in S3 zonder extra servers op te zetten en te beheren.

Realisatie AWS Glue Data Catalog & AWS Athena

Voordat ik kon beginnen met het maken van dashboarden moest ik er eerst voor zorgen dat mijn data uit S3 in de AWS Glue Data Catalog opgeslagen is. Om dit te bereiken heb ik een Glue Crawler gemaakt die de gewenste data uit S3 zou ophalen en opslagen in de Data Catalog.

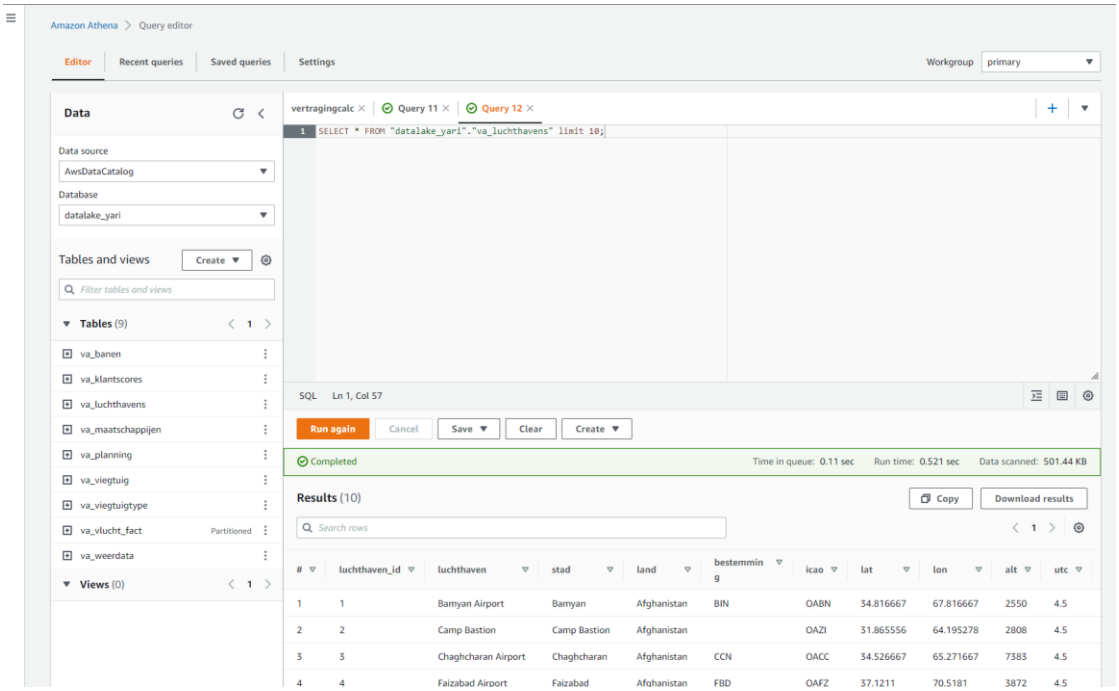
U kunt een crawler gebruiken om de AWS Glue Data Catalog te vullen met tabellen. U geeft met de crawler mee op welke data store hij zijn werk moet doen. In ons geval was dit de Cleansed Zone van het Data Lake. Na voltooiing creëert of actualiseert de crawler een of meer tabellen in de Data Catalog in uw AWS omgeving. Als dit gelukt is kan AWS Athena rechtstreeks data opvragen uit te net gemaakte tabellen die data bevatten uit onze S3 bucket. Zo storen we onze S3 omgeving niet elke keer als we data willen opvragen in Athena of als we data willen genereren op dashboarden in AWS QuickSight.

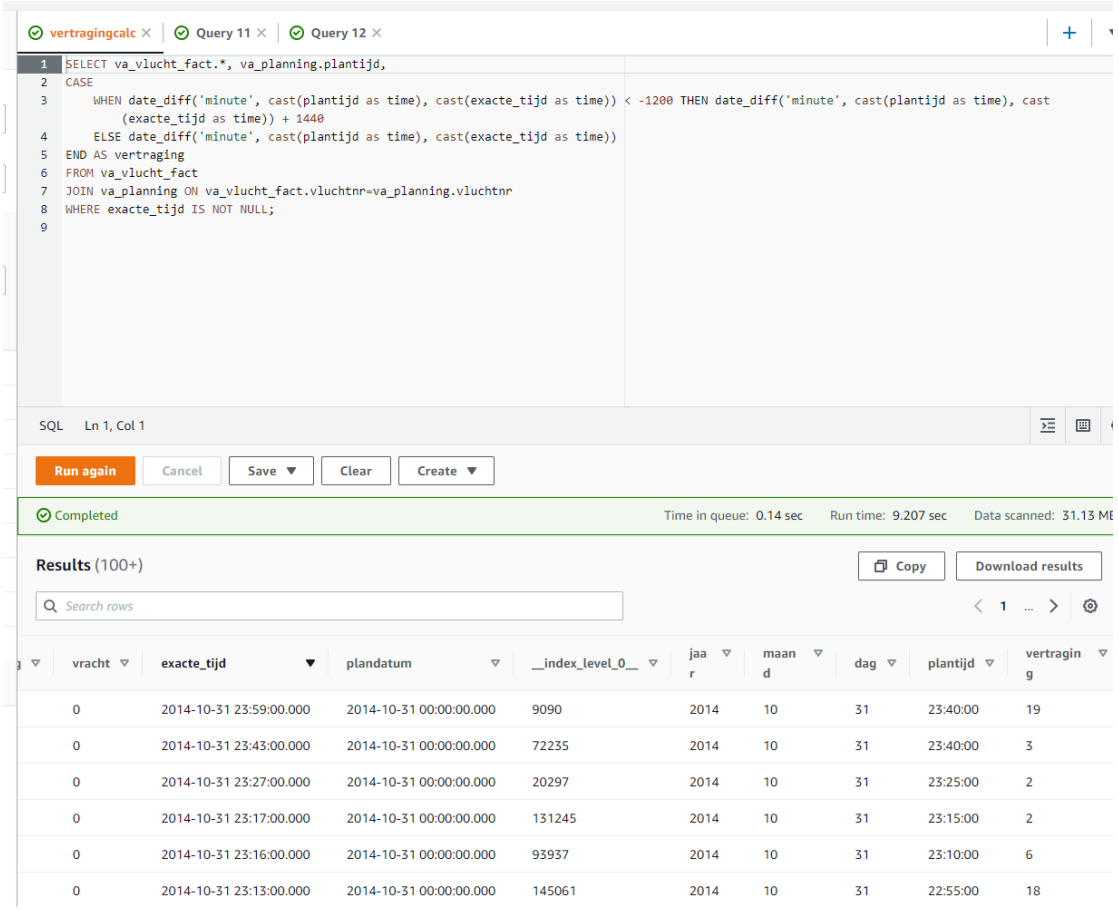


Hierboven kan u de nieuwe aangemaakte database met tabellen zien in de AWS Glue Data Catalog. Hiervoor heb ik dus een Glue Crawler gebruikt die ik via Terraform gedeployed heb. Ik heb met de crawler meegegeven op welke data store deze moet crawlen en in welke database de gevonden tabellen moet worden opgeslagen in de Glue Data Catalog.

```
resource "aws_glue_crawler" "datalake_yari_crawler" {
  database_name = "datalake_yari"
  name          = "datalake-yvd-crawler-tf"
  role          = "service-role/AWSGlueServiceRole-AWSGlueServiceRoleDefault"

  s3_target {
    path = "s3://s3://yari-data-lake-terraform/cleansed-zone"
  }
```

De nieuwe database en tabellen zijn dus succesvol toegevoegd aan de AWS Glue Data Catalog waar we nu in de Athena Query Editor SQL queries op kunnen uitvoeren. Zie voorbeelden hieronder:


Amazon Athena > Query editor
Editor
Recent queries
Saved queries
Settings
Workgroup primary
Data
Data source
AwsDataCatalog
Database
datalake_yari
Tables and views
Create
Filter tables and views
Tables (9)
va_banen
va_klantscores
va_luchthavens
va_maatschappijen
va_planning
va_viegtuig
va_viegtuigtype
va_vlucht_fact
Partitioned
va_weerdata
Views (0)
vertragingcalc
Query 11
Query 12
1 SELECT * FROM "datalake_yari"."va_luchthavens" limit 10;
SQL Ln 1, Col 57
Run again
Cancel
Save
Clear
Create
Completed
Time in queue: 0.11 sec
Run time: 0.521 sec
Data scanned: 501.44 KB
Results (10)
Copy
Download results
Search rows
luchthaven_id luchthaven stad land bestemming icao lat lon alt utc
1 1 Bamyan Airport Bamyan Afghanistan BIN OABN 34.816667 67.816667 2550 4.5
2 2 Camp Bastion Camp Bastion Afghanistan OAZI 31.865556 64.195278 2808 4.5
3 3 Chaghcharan Airport Chaghcharan Afghanistan CCN OACC 34.526667 65.271667 7383 4.5
4 4 Faizabad Airport Faizabad Afghanistan FBD OAFZ 37.1211 70.5181 3872 4.5


vertragingcalc
Query 11
Query 12
1 SELECT va_vlucht_fact.*, va_planning.plantijd,
2 CASE
3 WHEN date_diff('minute', cast(plantijd as time), cast(exacte_tijd as time)) < -1200 THEN date_diff('minute', cast(plantijd as time), cast (exacte_tijd as time)) + 1440
4 ELSE date_diff('minute', cast(plantijd as time), cast(exacte_tijd as time))
5 END AS vertraging
6 FROM va_vlucht_fact
7 JOIN va_planning ON va_vlucht_fact.vluchtnr=va_planning.vluchtnr
8 WHERE exacte_tijd IS NOT NULL;
9
SQL Ln 1, Col 1
Run again
Cancel
Save
Clear
Create
Completed
Time in queue: 0.14 sec
Run time: 9.207 sec
Data scanned: 31.13 MB
Results (100+)
Copy
Download results
Search rows
vracht exacte_tijd plandatum __index_level_0__ jaar maand dag plantijd vertraging
0 2014-10-31 23:59:00.000 2014-10-31 00:00:00.000 9090 2014 10 31 23:40:00 19
0 2014-10-31 23:43:00.000 2014-10-31 00:00:00.000 72235 2014 10 31 23:40:00 3
0 2014-10-31 23:27:00.000 2014-10-31 00:00:00.000 20297 2014 10 31 23:25:00 2
0 2014-10-31 23:17:00.000 2014-10-31 00:00:00.000 131245 2014 10 31 23:15:00 2
0 2014-10-31 23:16:00.000 2014-10-31 00:00:00.000 93937 2014 10 31 23:10:00 6
0 2014-10-31 23:13:00.000 2014-10-31 00:00:00.000 145061 2014 10 31 22:55:00 18

3.5.2 AWS QuickSight

De database en tabellen zijn toegevoegd aan de AWS Glue Data Catalog, we hebben onze Athena tables, we kunnen de data opvragen met SQL queries in de Athena Query editor. We zijn klaar om de data te verbinden met AWS QuickSight.

Wat is AWS QuickSight?

Amazon QuickSight is een Business Intelligence service in de cloud (AWS) waar dashboards, insights en analytics van bepaalde gegevens op een eenvoudig te begrijpen manier gedeeld kunnen worden aan de mensen van u organisatie. Omdat deze service in de cloud draait is uw QuickSight omgeving bijna altijd en overal beschikbaar om te bekijken. De service maakt achter de schermen een verbinding met de gegevens opgeslagen in de cloud (S3, AWS Glue Data Catalog…) met behulp van SQL queries. Dit gebeurt allemaal ook weer serverless. Er is geen infrastructuur die u moet opzetten om AWS QuickSight te gebruiken in al zijn glorie.

Waarom AWS QuickSight?

In mijn geval was het eenvoudiger om mijn data die al op AWS aanwezig was hieraan te linken, dan nog een externe tool te gebruiken voor enkele gegevens uit het Data Lake te visualiseren. QuickSight integreert efficiënt met allerlei verschillende data stores op AWS. De Athena tables had ik al gebruikt voor in de Athena Query editor data op te vragen, deze waren dus een goede keuze om opnieuw te gebruiken om data uit te halen en te visualiseren in QuickSight.

Een ander groot voordeel aan deze service is dat het, zoals hierboven al vermeld is, serverless is. Ik moest geen extra infrastructuur opzetten of managen om gegevens uit mijn Athena tables te visualiseren.

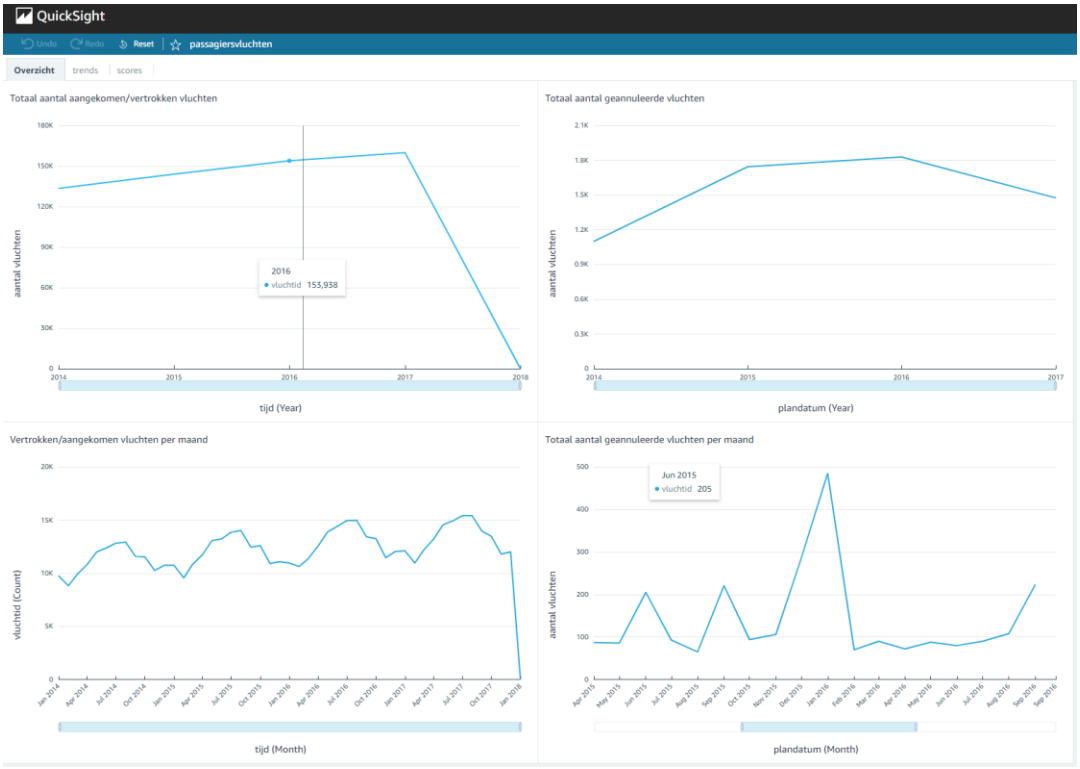
Realisatie

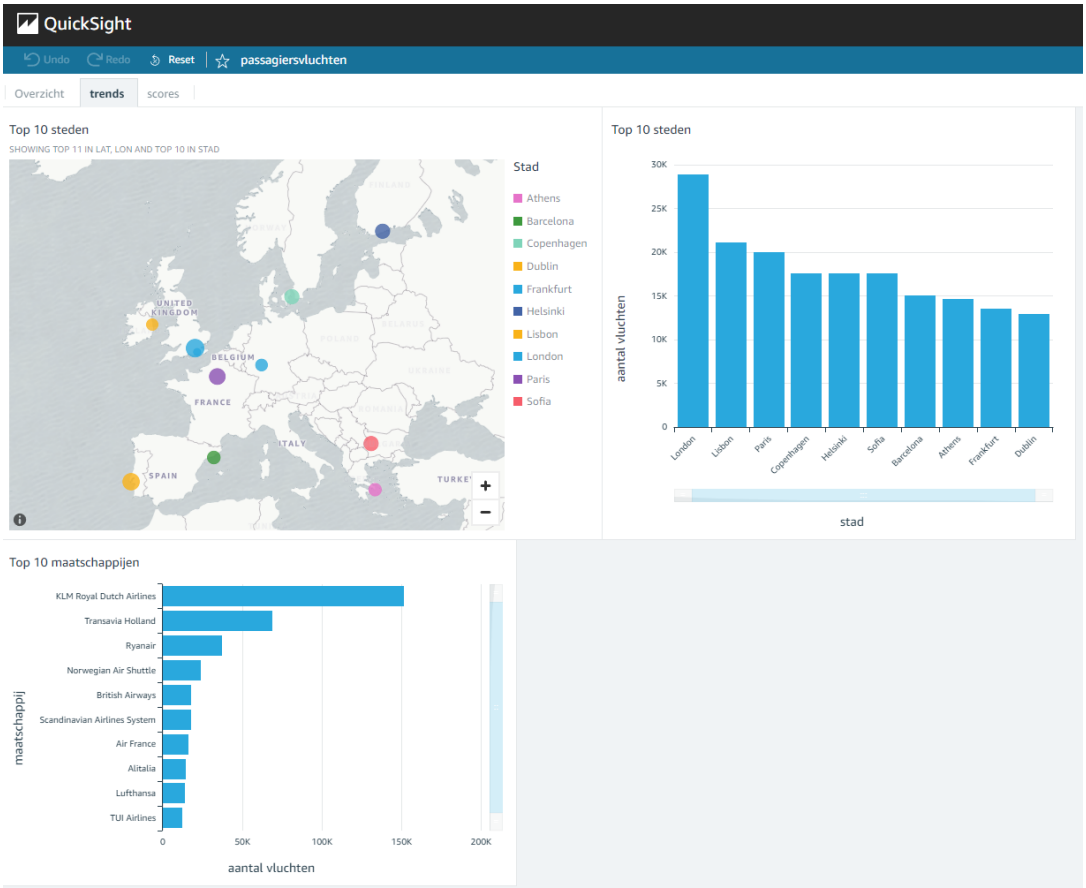
De dashboarden die ik gemaakt heb halen hun gegevens dus elke keer op uit de Athena tables, niet dus rechtstreeks uit mijn S3 bucket. De data die ik moest visualiseren mocht ik zelf kiezen. Ik had dus geen van te voren afgesproken gegevens die zeker gevisualiseerd moesten worden. Zo kon ik op mijn eigen manier exploreren hoe ik dit het beste aanpakte.

Voordat een dashboard gemaakt kan worden moest er met behulp van een SQL query bepaalde gegevens geselecteerd worden uit een of meerdere Athena tables die ik wou visualiseren op een dashboard. Zo kon ik gegevens die niet geladen moesten worden achterwegen laten zodat de gegevens die gevisualiseerd moeten worden zo snel mogelijk zichtbaar worden op de dashboards. Dit moest ik doen per apart dashboard dat ik wou.

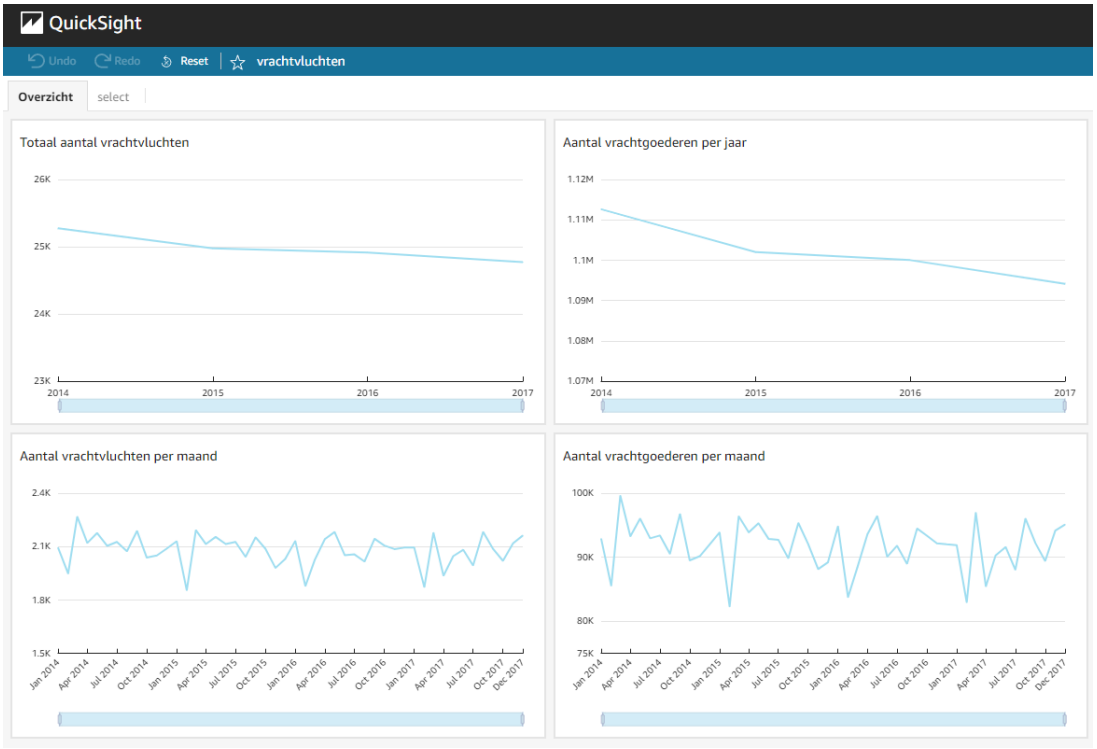
Ik heb drie dashboarden gemaakt: een dashboard waar gegevens van passagiersvluchten gevisualiseerd worden, een dashboard waar gegevens van vrachtvluchten gevisualiseerd worden en een dashboard waarop vluchten met vertraging en bijhorende gegevens over getoond worden.

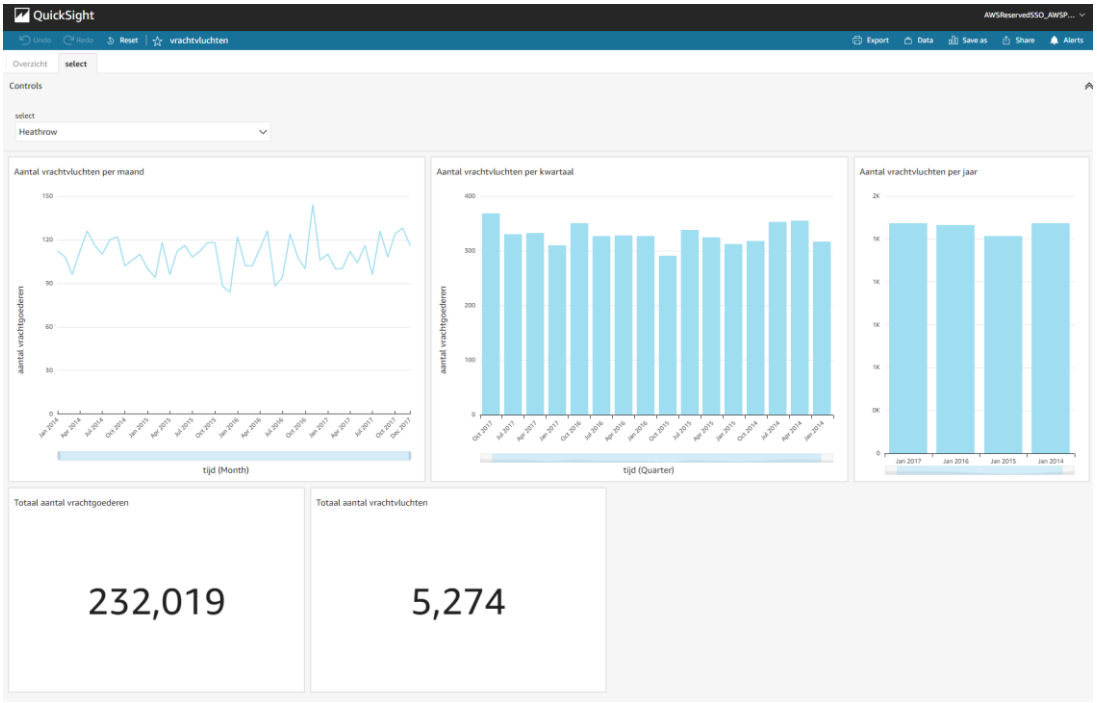
✓ ***Passagiersvluchten***



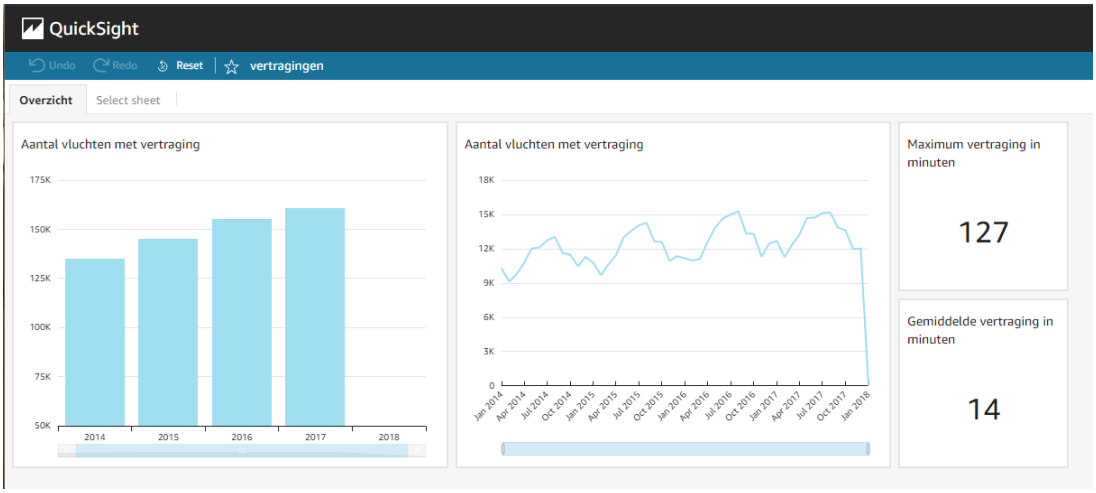


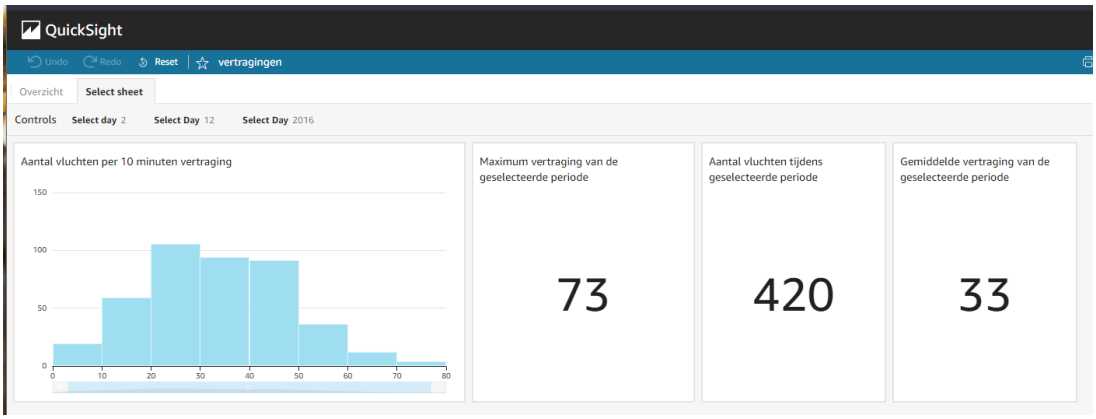
✓ ***Vrachtvluchten***





✓ ***Vertragingen***





De visualisatie in QuickSight was een van de drie extra datatoepassingen die ik gerealiseerd heb tijdens de stage. In de volgende twee hoofdstukken worden de andere twee extra toepassingen op onze dataset uitgelegd.

3.6 Data Science toepassing

De volgende uitbreiding die ik aan het Data Lake had toegevoegd was een Data Science toepassing, Machine Learning.

Mijn dataset bestond volledig uit historische data. In die historische data konden we allerlei informatie over vluchten terugvinden. Sommige vluchten hadden echter vertraging opgelopen. Vertraging kon veroorzaakt worden door enkele parameters, de belangrijkste was het weer. In de dataset zat er namelijk ook een tabel dat allerlei weerdata bevatte van 2014 tot 2018. Het was dus een interessante uitbreiding om het verband tussen het weer en de vertragingen te bestuderen.

Het uiteindelijk doel van de Data Science toepassing was om uiteindelijk vertraging te voorspellen door wat Machine Learning toe te passen op de dataset.

3.6.1 Realisatie van de Data Science toepassing

Ik ben begonnen met speciaal een nieuwe tabel te maken om deze toepassing op uit te voeren. In deze tabel heb ik alle vluchten met hun geplande tijd en hun exacte tijd dat ze zijn vertrokken of aangekomen toegevoegd. Ook het verschil tussen de geplande en exacte tijd toegevoegd wat dan de uiteindelijke vertraging van de vlucht is. Tenslotte heb ik dan de belangrijkste weerdata ook toegevoegd aan de tabel zoals windsnelheid, neerslag, min. en max. temperatuur… eigenlijk alle belangrijkste factoren die vertraging zouden kunnen veroorzaken.

De volgende uitdaging was dan het trainen van de dataset. Ik heb de dataset eerst opgedeeld in twee groepen. De trainingsgroep die 80% van de data uit de tabel representeerde en waar we ons model op gaan trainen. De tweede groep van de resterende 20% is dan de test groep om te vergelijken of de voorspelde resultaten overeenkwamen met de echte data uit de tabel.

Hieronder is een foto van de tabel waarop ik Machine Learning hebben uitgevoerd:

	vluchtid	vluchtnr	bestemming	exacte_tijd	plantijd	vluchtstatus	vertraging	windsnelheid	windstoot	temperatuur	neerslag	minimum_zicht	maximum_zicht	datum
0	935992	TK297	ESB	2014-01-01 02:46:00	2014-01-01 02:25:00	vertrekkende vlucht	21	38	110	-16	-1	57	75	2014-01-01
1	936179	TK298	ESB	2014-01-01 12:06:00	2014-01-01 11:55:00	aankomende vlucht	11	38	110	-16	-1	57	75	2014-01-01
2	936199	TK299	ESB	2014-01-01 12:55:00	2014-01-01 12:40:00	vertrekkende vlucht	15	38	110	-16	-1	57	75	2014-01-01
3	936421	TK296	ESB	2014-01-01 22:30:00	2014-01-01 22:20:00	aankomende vlucht	10	38	110	-16	-1	57	75	2014-01-01
4	935993	DL5829	DTW	2014-01-01 02:57:00	2014-01-01 02:15:00	vertrekkende vlucht	42	38	110	-16	-1	57	75	2014-01-01
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
658938	1647468	KL4377	NRT	2017-09-09 22:54:00	2017-09-09 22:25:00	aankomende vlucht	29	24	90	127	133	41	80	2017-09-09
658939	1647313	DL3952	BOS	2017-09-09 16:45:00	2017-09-09 16:05:00	aankomende vlucht	40	24	90	127	133	41	80	2017-09-09
658940	1647351	DL3953	BOS	2017-09-09 17:36:00	2017-09-09 17:20:00	vertrekkende vlucht	16	24	90	127	133	41	80	2017-09-09
658941	1647118	HV733	LPA	2017-09-09 11:57:00	2017-09-09 11:35:00	aankomende vlucht	22	24	90	127	133	41	80	2017-09-09
658942	1647143	HV734	LPA	2017-09-09 12:31:00	2017-09-09 12:20:00	vertrekkende vlucht	11	24	90	127	133	41	80	2017-09-09

658943 rows × 15 columns

Hieronder kunt u foto’s zien van de gebruikte code:

Data splicing

Our next step is to divide the data into independent variables and dependent variables, whose values are to be predicted. To predict the 'vertraging'. Create the two datasets and next, split 80% of the data to the training set and 20% to the test set.

```
X = df_final[['neerslag', 'windsnelheid', 'windstoot', 'temperatuur', 'minimum_zicht', 'maximum_zicht']].values
y = df_final['daggemiddelde_vertraging'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

```
print(X_train)
```

```
[[ 72  35 100  24   8  75]
 [  0  35  90  81  25  60]
 [114  45 180 137  50  75]
 ...
 [ 10  30  90 185  56  75]
 [ 37  33 120 149  61  75]
 [  1  23 170 208   0  80]]
```

```
print(y_train)
```

```
[10.96082949  2.88235294 18.76547842 ... -0.47311828  5.96461825
 -1.99786325]
```

Train the model

Now train the model.

```
model = LinearRegression()
model.fit(X_train, y_train) # training the model
```

```
LinearRegression()
```

Predictions

Now that we have trained our model, it's time to make some predictions. Do the prediction on test data.

```
y_pred = model.predict(X_test)
```

```
compare_df = pd.DataFrame({'Actual': y_test.flatten(), 'Predicted': y_pred.flatten()})
compare_df
```

	Actual	Predicted
0	7.083673	8.836239
1	1.180422	0.916108
2	19.838068	16.991994
3	10.012255	2.006188
4	6.673160	5.829542
...	...	...
131784	10.921154	10.495635
131785	1.247059	3.882167
131786	12.451444	18.673329
131787	-3.699640	7.211808
131788	-1.142086	5.805986

131789 rows × 2 columns

```
# Importing mean_absolute_error from sklearn module
from sklearn.metrics import mean_squared_error, mean_absolute_error

# printing the mean absolute error
print(mean_absolute_error(y_test, y_pred))
```

```
4.368514895235402
```

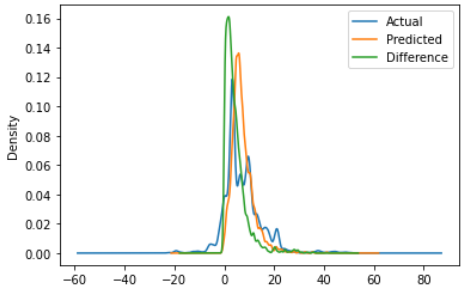
```
df_compare_with_diff = compare_df
df_compare_with_diff['Difference'] =  pd.Series.abs(df_compare_with_diff['Actual'] - df_compare_with_diff['Predicted'])
df_compare_with_diff
```

	Actual	Predicted	Difference
0	7.083673	8.836239	1.752566
1	1.180422	0.916108	0.264315
2	19.838068	16.991994	2.846074
3	10.012255	2.006188	8.006067
4	6.673160	5.829542	0.843618
...	...	...	...
131784	10.921154	10.495635	0.425518
131785	1.247059	3.882167	2.635108
131786	12.451444	18.673329	6.221886
131787	-3.699640	7.211808	10.911448
131788	-1.142086	5.805986	6.948072

131789 rows × 3 columns

```
bell_curve = df_compare_with_diff.plot.kde()
bell_curve
```

```
<AxesSubplot:ylabel='Density'>
```



Op de laatste bovenstaande foto kunnen we zien dat over het algemeen het verschil (groene lijn op de figuur) tussen de actuele data en de voorspelde data in de meeste gevallen niet ver van 0 afwijkt op de gausscurve. Het model was dus vrij accuraat in de meeste gevallen op een paar uitzonderingen na.

3.7 API toepassing

De laatste uitbreiding op het Data Lake dat ik tijdens de stage benaderd heb is het verbinden van een API met onze data storage op AWS. Met deze API zou ik een constante stroom aan data hebben in plaats van dat ik één keer al de data verstuur naar het Data Lake zoals ik met de originele dataset gedaan had.

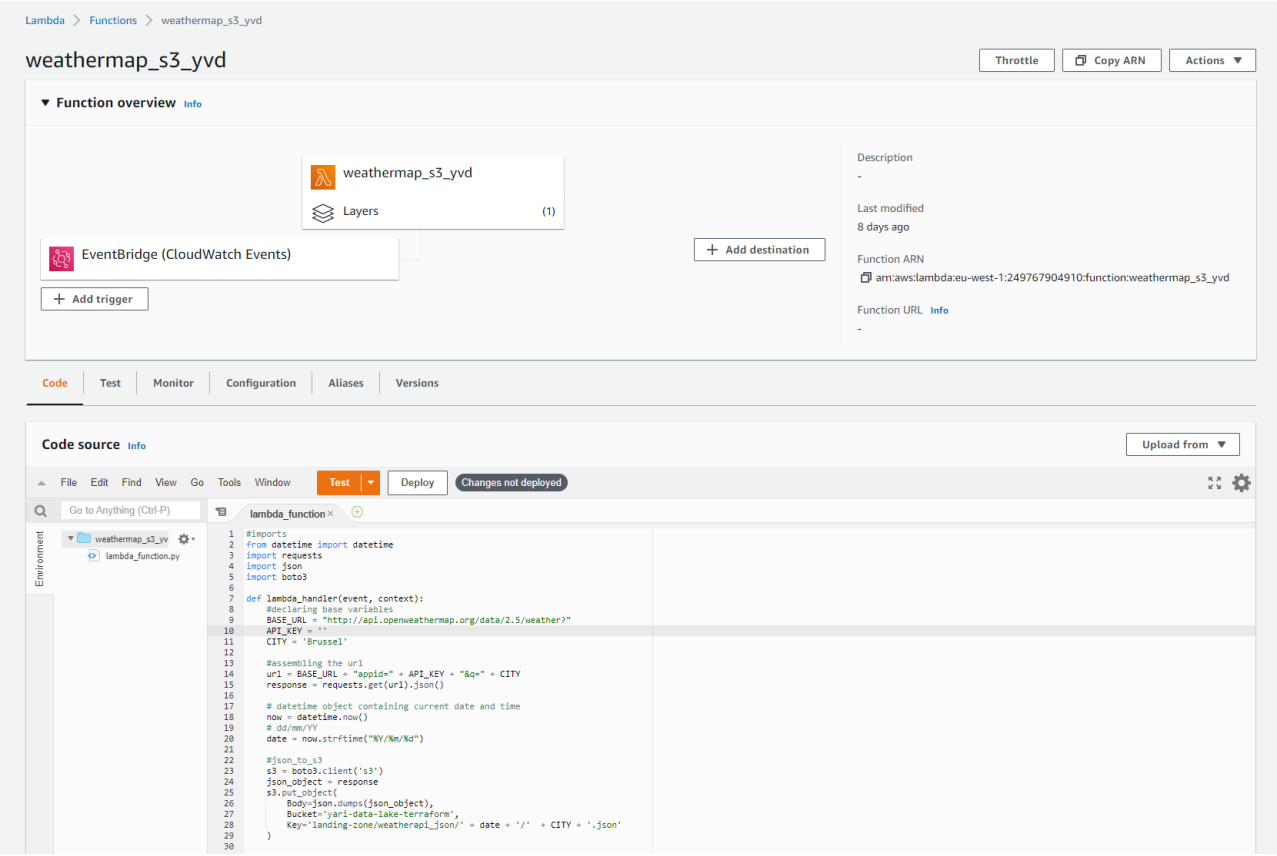
Realisatie API toepassing

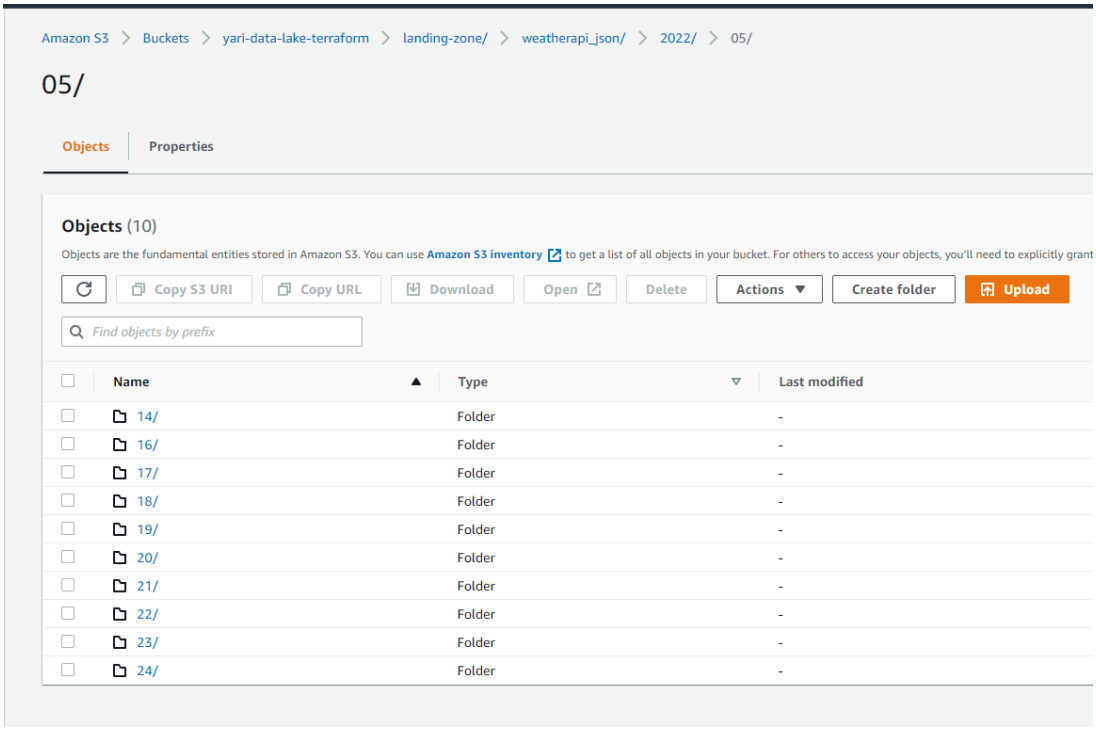
De API die ik gebruikt had gebruikt voor deze toepassing is de API van openweathermap.org. Met behulp van deze API kon ik actuele weerdata ophalen van een bepaalde locatie. Zo kon ik bijvoorbeeld om drie uur in de namiddag de API aanspreken voor weerdata door te sturen van de stad Brussel en dan kreeg ik allerlei weerdata van de stad Brussel van dat exacte moment zoals minimum en maximum temperatuur van de dag, windsnelheid, zichtbaarheid, eventueel neerslag als er neerslag zou zijn, wanneer de zon opgekomen is en wanneer deze zou ondergaan enzovoort… Het was dus een interessante oefening dat ik die weerdata automatisch volgens een tijdsschema zou doorsturen en bewaren in het Data Lake voor verder gebruik.

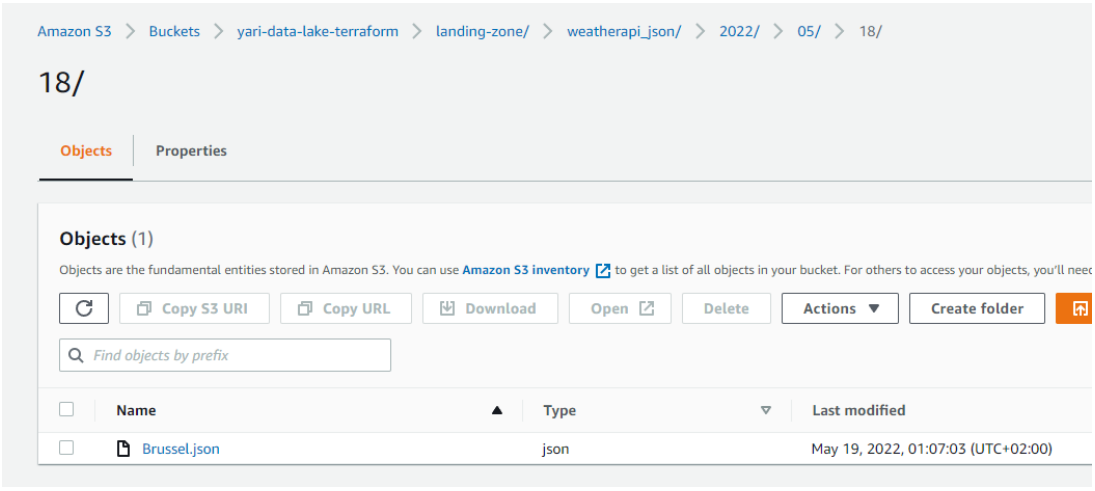
Het script kon ik hosten op een service van AWS, AWS Lambda genaamd. Deze service zorgt ervoor dat het gebruikte script om de API aan te spreken zonder enige infrastructuur of andere omgevingen gerund kon worden. AWS Lambda kan ook rechtsreeks verbonden worden met de Event manager van AWS zodat het script een bepaald tijdsschema kan volgen wanneer dat het uitgevoerd zou moeten worden.

Tenslotte werd de data dat werd opgehaald door het script opgeslagen in de Landing Zone van het Data Lake onder de bijhorende datum in JSON-formaat.

Hieronder is een foto van de gebruikte code en het resultaat in het Data Lake:







Om kort samen te vatten, elke dag om 13:07 uur werd het script uitgevoerd en werd de opgehaalde JSON data van openweathermap.org onder de bijhorende datum van wanneer het is opgehaald opgeslagen in de Landing Zone van het Data Lake.

Dit was de laatste technische toepassing dat ik bestudeerd had tijdens de stageperiode.

4. Besluit

Gedurende 13 weken ben ik volledig ondergedompeld geweest in de wereld van Big Data. Tijdens die weken heb ik mij enorm geamuseerd en is mijn interesse in een toekomst in IT alleen maar toegenomen. Op relatief korte tijd heb ik enkele van de meest belangrijke basics van Big Data getackeld. Data Engineering, Data Visualisatie, Data Science, API integratie en natuurlijk het opzetten van een Data Lake, dit waren de belangrijkste onderdelen die ik bestudeerd heb tijdens de stageperiode. Ik heb deze onderdelen voor het grote deel volledig op mijn eigen benaderd en bestudeerd. Dit was op sommige momenten zeker een uitdaging, ook omdat ik geen achtergrond had in de wereld van Big Data voordat ik aan de stage begon. Maar ik heb elk onderdeel een voor een aangepakt met eventuele ondersteuning van de stagementor om zo toch tot een efficiënt resultaat te komen. Zo is er toch een nieuwe passie ontstaan voor de wereld van Big Data.

Ik ben tenslotte zeer dankbaar voor de kans die mij gegeven is om aan de slag te gaan als stagiair bij Ordina en voor de ondersteuning die ik heb gehad van mijn stagementor gedurende heel de stageperiode. Ik sluit het avontuur af met een zeer positief gevoel.