HO
GENT

Faculteit Bedrijf en Organisatie

GraphQL vs REST: een vergelijkende studie

Sam Brsbaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Guy Dekoning
Co-promotor:
Yannick Borghmans

Instelling: Ventigrate

Academiejaar: 2020-2021

Tweede examenperiode

Faculteit Bedrijf en Organisatie

GraphQL vs REST: een vergelijkende studie

Sam Brsbaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Guy Dekoning
Co-promotor:
Yannick Borghmans

Instelling: Ventigrate

Academiejaar: 2020-2021

Tweede examenperiode

Woord vooraf

Ik heb gekozen voor het onderwerp omdat ik vorig jaar voor het eerst in aanraking kwam met APIs en het me erg interesseerde. Ik kwam vooral in contact met REST APIs, maar dan begon ik over GraphQL te lezen en wou ik weten of het al zijn beloften kon nakomen om een echte verbetering te vormen voor REST.

Ik zou gebruik willen maken van dit voorwoord om een aantal mensen te bedanken. Eerst en vooral zou ik graag mijn promotor Guy Dekoning voor zijn bijstand in dit onderzoek en goed adviezen. Ik zou ook mijn co-promotor Yannick Borghmans willen bedanken voor zijn hulp. Ook zou ik graag Marina Kantcur willen bedanken voor de steun en het luisterend oor.

Samenvatting

In deze bachelorproef wordt nieuwkomer in de API wereld GraphQL vergeleken met de veelgebruikte REST architectuur voor een data-gebaseerde API. De keuze heeft een grote invloed op de structuur van de API, de aanpasbaarheid, de documentatie en op de performantie. Dit laatste zowel op basis van de hoeveelheid data dat verstuurd moet worden als op de snelheid van de verzoeken. Om deze zaken te onderzoeken werden er twee proof-of-concepts opgesteld, een met REST, en een met GraphQL als technologie voor de API. Dit liet ons toe om een concrete vergelijking te maken tussen beide. Uit deze vergelijking bleek GraphQL de betere keuze te zijn, op een aantal vlakken scoort het gelijkaardig aan REST, maar op een aantal, vooral performantiegebaseerde, vlakken kwam het duidelijk als overwinnaar naar boven.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

1.1 Context

Voor het ontstaan van het wereldwijde web, verliep de communicatie tussen computers meestal intern binnen een organisatie op een lokaal netwerk. Toen alle communicatie binnenshuis gebeurde, wist elke computer alles over de interne werking van alle andere computers op het netwerk. Dit systeem was voldoende voor de noden van de tijd, maar toen in 1989 het wereldwijde web ontwikkeld werd door CERN en zeker daarna toen het rond het jaar 2000 explodeerde in populariteit, werd duidelijk dat er een nieuwe manier van werken moest komen.

Het moest mogelijk zijn om over het internet te communiceren en daarvoor moesten een aantal afspraken vastgelegd worden. Voorbeelden hiervan zijn HTTP, FTP, enz. Dit zijn protocollen die vastleggen hoe data over een netwerk, in dit geval het wereldwijde web, verstuurd kan worden. Door gebruik te maken van deze protocollen gaat er een hele wereld open van communicatie, hetzij intern binnen een organisatie (eventueel op andere locaties) of extern tussen verschillende organisaties. Ze hebben echter niets te zeggen over de inhoud van de data zelf, of hoe deze data opgevraagd kan worden. Om dat te bepalen was een ander systeem nodig, namelijk het idee van een API.

1.2 Probleemstelling

Eens Ventigrate, of een gelijkaardige organisatie, gekozen heeft dat men een API wil maken voor een softwareproject, resteert nog de belangrijke keuze welke technologie men hiervoor zal gebruiken. De meest populaire keuze is al een hele tijd REST, er was

lang weinig concurrentie voor. Voor sommigen lijken API en REST zelfs onlosmakelijk verbonden te zijn, maar dat is nu althans niet meer het geval. Er is sinds 2015 een nieuwe speler, namelijk GraphQL dat veel belofte toont. Dan komt dus de vraag, blijf je bij het oude, vertrouwde REST, of geef je de nieuwe, veelbelovende GraphQL een kans?

1.3 Onderzoeksvraag

Welke technologie is beter om een API te maken voor een data-gebaseerde applicatie, REST of GraphQL?

Om de vergelijking te maken tussen REST en GraphQL hebben we een aantal requirements opgesteld, enerzijds functionele en anderzijds niet-functionele.

Functionele requirements:

- Er moet een mogelijkheid zijn om documentatie automatisch te genereren
- Het moet losgekoppeld zijn van de persistentie
- Het moet mogelijk zijn om de API te updaten

Niet-functionele requirements:

- Het moet stabiel zijn, met weinig dataverlies
- Het moet snel zijn
- Er moeten libraries zijn voor de meest gebruikte programmeertalen
- De hoeveelheid data dat over het netwerk gaat moet beperkt zijn
- Het aantal verzoeken moet beperkt zijn.

1.4 Onderzoeksdoelstelling

Dit onderzoek is een vergelijkende studie over de meest geschikte technologie voor een API tussen REST en GraphQL. Om de vergelijking te maken worden er twee proof-of-concept applicaties aangemaakt, een met REST en een met GraphQL. Het gaat om een simpele data-gebaseerde applicatie waar keuringen voor producten in kunnen worden bijgehouden. Omdat dit onderzoek als opzettelijk doel heeft om deze twee technologieën met elkaar te vergelijken, is het opstellen van een long list overbodig.

Het verwachte resultaat is dat GraphQL zal slagen in zijn opzet om een verbetering te zijn op REST, zeker als het gaat over geneste data met veel attributen.

1.5 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt context gegeven, eerst en vooral wordt duidelijk gedefinieerd wat een API is, daarna wat REST is en tenslotte wat GraphQL is.

In Hoofdstuk 3 wordt de methodologie bekeken. Er wordt besproken welke proeven uitgevoerd worden om REST en GraphQL aan de requirements te toetsen.

In Hoofdstuk 4 wordt de proof-of-concept toegelicht en wordt dieper ingegaan op de keuze van platform en achterliggende technologieën.

In Hoofdstuk 5 wordt aan de hand van de proof-of-concept de vergelijking gemaakt op basis van de gekozen requirements, en worden de resultaten van de performantietests gegeven.

In Hoofdstuk 6 wordt een conclusie gevormd op basis van wat er uit het onderzoek naar boven gekomen is. Er wordt een keuze gemaakt tussen de twee technologieën en er wordt bekeken wat mogelijk toekomstig onderzoek over dit onderwerp zou kunnen zijn.

2. Stand van zaken

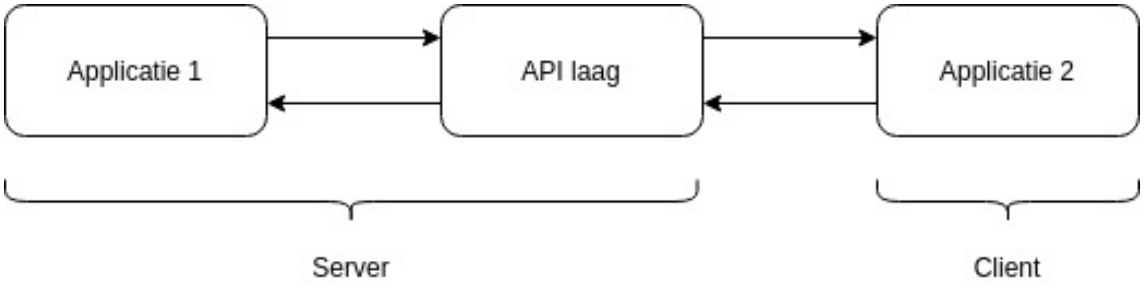
Omdat deze bachelorproef een vergelijking maakt tussen twee technologieën die gebruikt worden voor APIs, moet eerst worden uitgelegd wat een API juist is. Er wordt bekeken wat ze juist kunnen, welke probleem ze willen op lossen en hoe ze daarvoor te werk gaan. Daarna wordt er dieper ingegaan op de de verschillende aanpak van de twee technologieën.

2.1 Wat is een API?

Een API is een interface tussen een programma en de buitenwereld. Het definieert welke verzoeken ernaar gemaakt kunnen worden, hoe ze aangeroepen kunnen worden en welk formaat de data in de verzoek moet hebben.

De API definieert dus de regels waarmee dat de communicatie zal gebeuren. De werking van API's kan best uitgelegd door middel van een voorbeeld, hier wordt de manier van bestellingen opgenomen in een restaurant bekeken. In dit voorbeeld is de ober van het restaurant de API en het menu de lijst van mogelijke verzoeken. Eenmaal de klant een keuze heeft gemaakt, geeft hij dit door aan de ober. Op dat moment is hij een verzoek aan het maken. De ober zal de bestelling opnemen, en na enige tijd komt hij later terug met het gerecht dat de klant besteld had. Hierbij heeft de klant geen idee wat er allemaal achter de schermen gebeurd is, maar heeft hij wel net wat hij gevraagd heeft.

Een belangrijk onderscheid is dat in ons fictief restaurant alleen de klant de ober tot bij zich kan roepen, omwille van het gebruik van het client-server model (zie later). In dit geval is de klant de client en het restaurant (vertegenwoordigd door de ober) de server. De client is altijd diegene die het contact maakt en initiatief neemt. In dit restaurant zal de ober dus nooit zelf bij klant komen vragen of hij nog iets wilt.

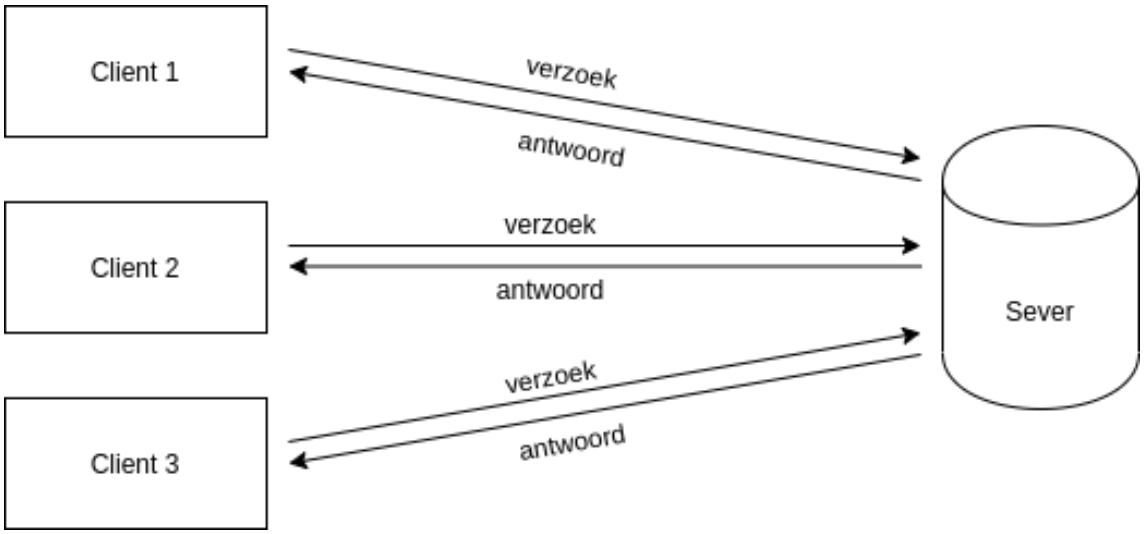


Figuur 2.1: Grafische voorstelling API

Een ander voorbeeld is het aanvragen van een paspoort. De klant moet hiervoor een aantal formulieren invullen en documenten afgeven. Dit gebeurt allemaal op een zeer goed uitgelijnde en specifieke manier. Als er iets fout ingevuld is of een document ontbreekt (gedeeltelijk of volledig), wordt het verzoek geweigerd. Zo gaat dat ook bij een API, daarin wordt een lijst van mogelijke verzoeken opgesteld en een heel specifieke manier waarop ze moeten worden aangevraagd waar men zich aan moet houden.

2.1.1 Client-servermodel

Zoals tevoren besproken, steunen API's op het client-servermodel. Dit is een model om de samenwerking tussen computersystemen mogelijk te maken. De server biedt diensten aan waar clients, ofwel klanten, gebruik van kunnen maken. Deze diensten zijn typisch computerprogramma's. Het kan zijn dat de dienst een calculatie of andere complexe operatie uitvoert, maar in veel gevallen gaat het over data. Zowel het ophalen, als het aanpassen, verwijderen of toevoegen ervan. Dit wordt vaak een CRUD-applicatie (zie later) genoemd en hierop zal de focus liggen in dit werk.



Figuur 2.2: Client-server model

Heel wat zaken worden tegenwoordig met het client-serverprincipe geregeld. Voorbeelden zijn file servers, het wereldwijde web (WWW), en e-mail. Een computer kan zowel de rol

van client als van server spelen. Een computer kan ook meerdere diensten hosten, dus voor meerdere diensten als server dienen. Uiteraard kan een client ook van meerdere servers gebruik maken.

De server staat in de meeste gevallen op elk moment klaar om verzoeken van clients te aanvaarden. De clients contacteren de server met een verzoek en nooit omgekeerd, het is dus eenrichtingscommunicatie.

Omwille van het client-servermodel hoeft de client de onderliggende werking van de server niet weten, maar moet het enkel weten welke verzoeken er gemaakt kunnen worden en hoe.

2.1.2 CRUD

Het principe van CRUD is een architecturale stijl voor software, het staat voor Create, Read, Update, Delete (J. Martin, 1983). Het idee ervan is dat dit alle mogelijke operaties omvat die men nodig zou hebben voor het manipuleren van opgeslagen data.

Deze vier operaties beschrijven de meerderheid van het soort verzoeken dat bij een API gedaan worden. In feite zijn er ook nog andere soorten operaties mogelijk, zoals bijvoorbeeld het uitvoeren van een berekening. In dit bachelorproef, zullen we dit soort verzoek behandelen als een read-verzoek.

In de meeste gevallen zal de persistentie van deze data geregeld worden door middel van een relationeel databank (Nayak e.a., 2013), maar dat is absoluut niet noodzakelijk. Het is juist de sterkte van een API dat implementatie verborgen is, dus hierdoor is het meestal perfect mogelijk om bijvoorbeeld gebruik te maken van een NoSQL databank (zoals bv. MongoDB of Redis).

2.2 Waarom API?

Nu verduidelijkt is wat een API is, stelt de vraag zich welke waarom men zou kiezen om een API te maken. In dit deel wordt bekeken wat zijn troeven zijn en voor welke redenen men hiervoor zou kiezen.

De voornaamste reden is om de achterliggende implementatie van software de verbergen. Er zijn een aantal goede redenen waarom men dat zou willen doen, namelijk:

1. Om de exacte details van hoe iets geïmplementeerd is geheim houden zodat concurrenten niet te weten komen hoe iets bereikt wordt. Dus om je concurrentieel voordeel te behouden en je intellectueel eigendom te beschermen.
2. Security. Als de code verborgen blijft, is het moeilijker voor hackers om bugs en fouten te vinden die gebruikt kunnen worden om zonder toestemming toegang te krijgen of iets laten mislopen in uw programma.
3. Als er alleen een interface naar de buitenwereld beschikbaar is, maakt het het

mogelijk om in de toekomst aanpassingen te maken aan de implementatie zonder dat de gebruikers van de API daar iets voor moeten aanpassen.

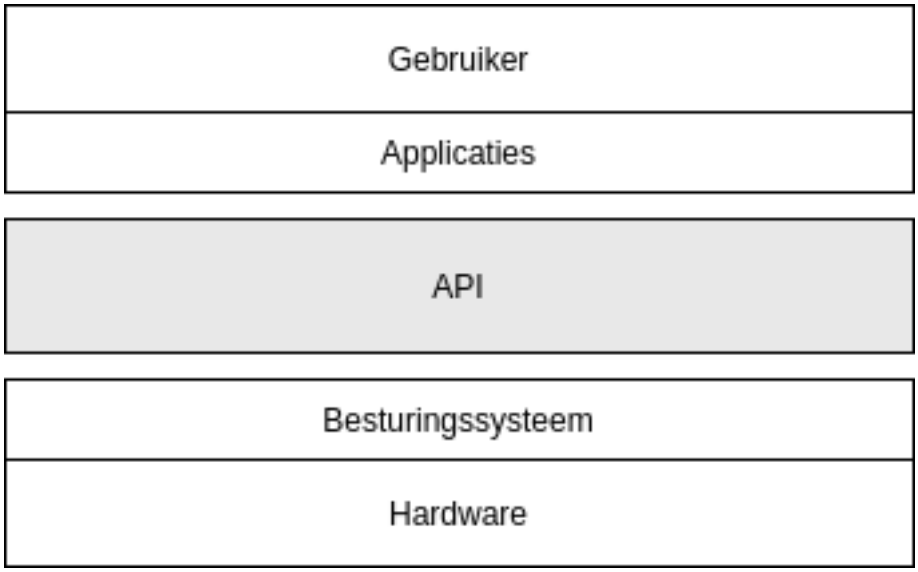
Een goed ontworpen API vervult juist een taak of use case. Binnen een programma (dit zou bijvoorbeeld zelf een API kunnen zijn) kan er gebruik gemaakt worden van meerdere aparte APIs. Dit zorgt allemaal voor loose coupling, wat een zeer belangrijk kenmerk is van clean code (R. C. Martin, 2018).

2.3 Soorten API

Het is belangrijk om te vermelden dat er verschillende soorten API zijn, die allemaal andere doelen vervullen. In dit deel zal even worden ingegaan op de verschillende soorten.

2.3.1 API voor besturingssystemen

Een besturingssysteem bestaat om computer hardware en software te beheren en te zorgen dat alle middelen (bv. RAM, processor, opslagruimte) op een juiste manier verdeeld worden over de verschillende programma's die ze willen gebruiken. Het definieert ook een aantal algemene diensten die door elk programma gebruikt kunnen worden, bijvoorbeeld om toegang te krijgen tot de files op het systeem of om de systeemtijd aan te vragen.



Figuur 2.3: API voor besturingssytemen

Voor deze diensten wordt een API gedefinieerd. Zo bestaat er voor UNIX-gebaseerde besturingssystemen (bv. Linux, MacOS) het POSIX API dat functionaliteiten zoals stat() om de status van bestanden te krijgen en time() om de systeemtijd te krijgen (IEEE, 2018). Exacte implementatiedetails worden niet gedefinieerd in deze APIs en dus kunnen die verschillen van systeem tot systeem. Alleen de manier waarop de verzoeken kunnen opgevraagd worden en wat verwacht wordt als output staat vast. Het is van uiterst belang

dat de definitie van dit soort low-level API niet zomaar veranderd, want dat zou ver reikende gevolgen hebben (Reddy, 2011). Heel wat programma's zouden in dat geval namelijk niet meer correct functioneren.

Op figuur 2.3 wordt een grafische weergave gegeven van een API voor een besturingssysteem. De API is een laag tussen het besturingssysteem en applicaties. Alle communicatie tussen de twee gaat via de API.

2.3.2 API voor programmeertalen

Een ander belangrijk soort API wordt gebruikt bij programmeertalen. Zo heeft de taal C bijvoorbeeld een API, de C standard library, waar opgelijst staat welke functionaliteit allemaal meegeleverd moet worden. Een ander voorbeeld is de Java API dat hetzelfde fungeert als de C standard library, maar dan voor de programmeertaal Java.

Naast standaardbibliotheken zijn er ook tal van andere APIs die proberen te voorkomen dat programmeurs altijd maar dezelfde code opnieuw moeten schrijven. Zo zijn er bijvoorbeeld libraries om encoding van audio en video te doen (bv. JAVE2 voor Java), of libraries voor complexe wiskundige operaties (bv. NumPy voor python).

Het is belangrijk om even te vermelden dat een library en een API niet hetzelfde zijn. Een library is namelijk niet alleen een oplijsting van vereistes, maar het bevat namelijk ook de implementatie. Binnen de context van een library kan de API gezien worden als een logische representatie van wat er allemaal mogelijk is met de library, dus met andere woorden welke verzoeken uitgevoerd kunnen worden en wat de output daarvan dient te zijn.

2.3.3 Web API

Als men het vandaag de dag over APIs heeft, gaat het meestal over dit soort API. Het zijn APIs die bedoeld zijn om de communicatie tussen verschillende stukken software over een netwerk te faciliteren, in veel gevallen over het internet (Espinha e.a., 2015). Deze communicatie kan zijn tussen programma's binnen een organisatie. Het kan ook dat een organisatie zijn API openstelt aan derden zodat zij onderling kunnen communiceren.

Voorbeelden van dit soort API zijn:

- Skyscanner Flight Search - informatie over vluchten van verschillende maatschappijen
- Open Weather Map - informatie over de huidige weersomstandigheden op verschillende locaties
- Yahoo Finance - nieuws en data over de beurs
- Twitter - toegang tot functionaliteiten zoals tweets, persoonlijke berichten, enz.

Er zijn dus allerlei soorten Web APIs, het wordt duidelijk waarom men hiervan gebruik zou willen maken. Er is door gebruik te maken van dit soort API informatie te verkrijgen

die moeilijk of onmogelijk is om zelf te achterhalen, maar waar andere organisaties al over beschikken.

Het is bij web APIs dat REST en GraphQL gebruikt kunnen worden, dus als we het in de rest van deze bachelorproef hebben over een API, gaat het over een web API.

2.4 REST

REST (Representational State Transfer) is een software-architectuur voor API's dat vastgelegd is door Roy Fielding in zijn doctoraatsthesis "Architectural Styles and the Design of Network-based Software Architectures" in 2000 (Fielding, 2000).

In tegenstelling tot HTTP of FTP is het geen protocol, maar zoals Fielding het zelf benoemt een "architectural style". Het verschil zit in het feit dat een protocol exact beschrijft hoe berichten opgesteld moeten worden, maar bij REST is het mogelijk om gelijk welk formaat te gebruiken. Wat REST wel definieert is een aantal vereisten waar een API aan moet voldoen. Als het aan al deze vereisten voldoet, kan de API een 'RESTful API' genoemd worden.

De vereisten worden door Roy Fielding uitgelijnd in zijn thesis. Ze zijn als volgt:

- Client-Server
- Statelessness
- Cache
- Uniform Interface
- Layered System
- Code-On-Demand

Aangezien deze vereisten aan de basis liggen van REST, is het belangrijk om hier even op dieper in te gaan en de bedoeling van elke vereiste te gaan ontleden.

2.4.1 Client-Server

Zoals eerder besproken, is het client-server model van uiterst belang bij een API. REST vormt hier zeker geen uitzondering op. Het is belangrijk om een client-server model te hanteren om ervoor te zorgen dat je "separation of concerns" hebt, wat een van de hoofdzakelijke redenen is om een API te gebruiken. Via dit model zorg je ervoor dat verschillende softwareprojecten los van elkaar blijven en dus niet weten wat er bij elkaar achter de schermen gebeurt.

2.4.2 Stateless

De server houdt geen data bij over de client. Elk verzoek moet dus alle nodige informatie bevatten zodat de server het kan begrijpen. Dit wil zeggen dat als er geen context gepaard

is aan een verzoek vanuit het standpunt van de server. Alle contextuele informatie moet dus door de client worden bijgehouden. Volgens Fielding is dit om zichtbaarheid, betrouwbaarheid en schaalbaarheid in de hand te werken. Zichtbaarheid omdat de server enkel naar de data meegeleverd met het verzoek moet kijken, en niets anders. Betrouwbaarheid omdat het een atomaire operatie is en dus minder plaatsen heeft waarop het kan falen. Schaalbaarheid omdat het feit dat de server geen data hoeft bij te houden ervoor zorgt dat er meer middelen overblijven voor andere verzoeken.

2.4.3 Cache

Cache zorgt ervoor dat als dezelfde (of toch gelijkaardige) verzoeken herhaaldelijk gestuurd worden, dat de server niet telkens hetzelfde werk opnieuw moet doen. Dit wordt bereikt door de uitkomst van verzoeken geheel of gedeeltelijk bij te houden. Dit kan de gemiddelde wachttijd van een verzoek helpen reduceren.

2.4.4 Uniform Interface

Net zoals interfaces bij Object-georienteerd programmeren (Design Principles and Design Patterns, Robert C. Martin), wordt bij REST gebruik gemaakt van interfaces. De bedoeling is dat er een soort contract wordt opgesteld tussen de server en de client waardoor de achterliggende implementatie kan veranderen terwijl het contract onveranderd blijft. (eventueel nog schrijven over “the four constraints for this uniform interface”).

2.4.5 Layered System

Het layered system vereiste heeft weer te maken met “separation of concerns”. De client gebruikt de API om te communiceren met de server, maar dat gebeurt niet rechtstreeks. Er kunnen een onbepaald aantal andere lagen tussen zitten. Een voorbeeld van een van de lagen dat hier tussen kan zitten is een load balancer. Hierdoor kan de server in werkelijkheid eventueel meerdere servers zijn. Wanneer een verzoek binnenkomt, zal de load balancer dan kiezen welke server het zal verwerken. Er kunnen ook meerdere servers samenwerken om een enkel verzoek te vervullen. Zo zou het bijvoorbeeld kunnen dat een server het verzoek binnen krijgt, en daarvoor data moet opvragen aan een andere server. De client merkt hier helemaal niets van omdat er gewerkt wordt met een gelaagd systeem.

2.4.6 Code on demand

Dit omvat dat het mogelijk moet zijn om naast puur met data te werken, ook code te versturen van de server naar de client. Deze vereiste is volgens Fielding optioneel en zullen we dus niet verder over uitwijden.

2.5 GraphQL

GraphQL staat voor Graph Query language, zoals de naam doet vermoeden is het een query taal. Het mag wel niet verward worden met iets zoals SQL (Structured Query Language), wat een taal is om rechtstreeks aan de data in een relationele databankmanagementsysteeem te geraken. GraphQL daarentegen is bedoeld om over een API te gebruiken en dus niet rechtstreeks op de onderliggende data. Het vervult dus dezelfde functie als REST, maar op een andere manier en met een andere focus. De andere manier is dat de verzoeken heel anders worden opgebouwd (hier later meer over). De andere focus is om enkel de gewenste data op te halen, niet meer en niet minder.

2.5.1 Ontstaan

GraphQL werd ontworpen door facebook in 2012. Ze waren toen bezig met het omzetten van hun mobile apps naar native apps in plaats van wrappers voor de mobile website. Hiervoor hadden ze nood aan een API om de data van de news feed te versturen van hun servers naar deze apps, want voordien werd er gewoon html van de server gestuurd.

Hiervoor hebben ze onderzocht of REST genoeg was om hun noden te vervullen, maar ze merkten dat er een grote kloof was tussen de data die ze wilden en de server queries die daarvoor nodig zouden zijn bij een REST API (Byron, 2015). Ze wouden dus dat de queries die ze konden doen om data op te halen van de server nauwer aansloot bij de effectieve structuur van die data. Ze hadden als doel dat er daardoor meer specifieke requests konden uitgevoerd worden om net de data op te halen die ze wilden.

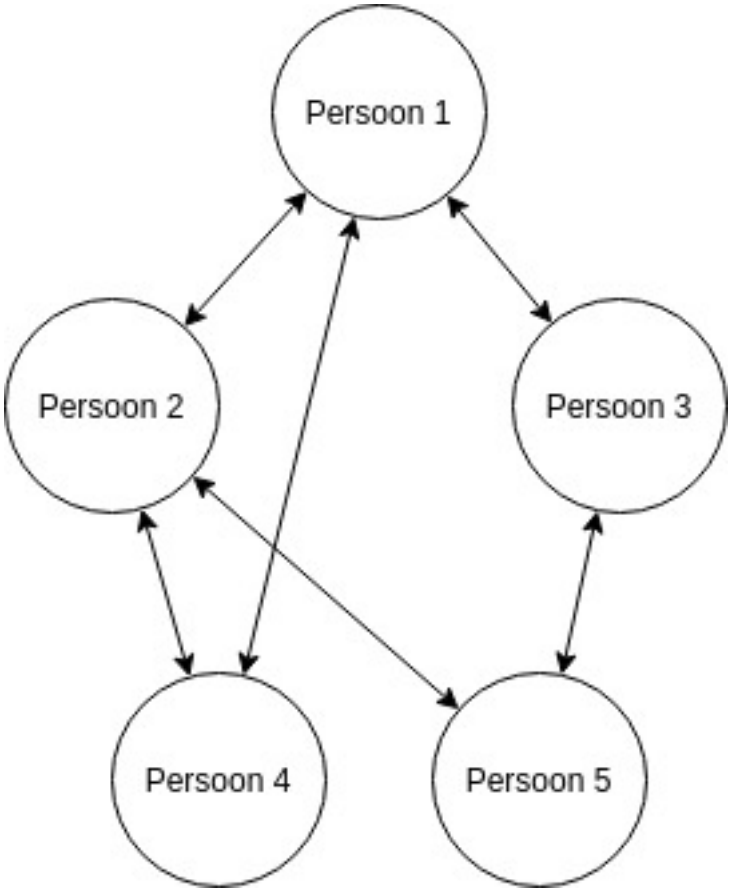
2.5.2 Grafen

Het ander deel van de naam (naast QL) is Graph, dit is omdat GraphQL gebaseerd is op het idee dat je veel datastructuren en hun onderlinge relaties kan voorstellen door middel van grafen.

Een graaf is een verzameling van knopen (ook gekend als nodes) en zijden (ook gekend als bogen). De knopen kan je zien als de data-objecten, en de bogen als hun onderlinge relaties. Grafen kunnen gericht zijn, in dit geval krijgen de bogen een richting (vaak aangeduid met een pijl).

Om dit wat concreter te maken zullen we kijken naar een voorbeeld. Op sociale media-platformen zoals facebook heb je accounts, en elk van die accounts hebben relaties met andere accounts. Deze relaties worden vaak vriendschappen genoemd. In deze situatie kan men dus de accounts voorstellen door knopen, en hun relatie als bogen. Bij een sociale media met vriendschappen, zullen de bogen van de graaf telkens in beide richtingen wijzen. Als een account vrienden is met een andere account, is dat namelijk ook waar in de andere richting. Figuur 2.4 toont hoe dit grafisch voorgesteld wordt.

Er zijn ook sociale media waarbij de relaties tussen de accounts niet per se in beide



Figuur 2.4: Graaf van facebook vrienden

richtingen gaan, zoals bijvoorbeeld instagram. Hierbij is er een systeem van volgers in plaats van een van vriendschappen. Hierbij kunnen de bogen in een richting gaan, of in beide als de accounts elkaar volgen.

Zo begint duidelijk te worden hoe men alle data in een computersysteem kan voorstellen door middel van een graaf. Dit is eigenlijk niet anders dan hoe data wordt opgeslagen in een relationele databank. Het steunt op hetzelfde principe, namelijk dat er entiteiten zijn met kenmerken en onderlinge relaties. GraphQL steunt dus veel dichter aan bij de databank dan REST. Achter de schermen worden de queries, of dat nu met REST of GraphQL werd gedaan, toch gewoon op de databank uitgevoerd, in de meeste gevallen met SQL. Bij REST is het enkel mogelijk de data die men wil opvragen/aanpassen te specificeren met eindpunt URL's. Bij GraphQL gebeurt dit door middel van een GraphQL query, en ligt dit dus veel dichter bij de representatie in de databank.

2.5.3 GraphQL en SQL

GraphQL en SQL zijn beide query talen, ze vervullen echter elk wel een ander doel. SQL is bestemd voor gebruik rechtstreeks op de databank, terwijl GraphQL bedoeld is om bij API's gebruikt te worden. In feite hoeft de data waar GraphQL aan kan zich helemaal

niet in een relationele databank te bevinden. De data kan bijvoorbeeld ook in een niet-relationele databank zijn, zoals MongoDB of Redis. De syntax van GraphQL en SQL is ook volledig anders. Bij SQL wordt een SELECT statement gebruikt om data op te halen, terwijl dit bij GraphQL met Query() gedaan wordt.

Ondanks het feit dat ze niet net hetzelfde zijn, is het feit dat ze op elkaar lijken geen toeval. Het was namelijk de bedoeling van de makers GraphQL om ervoor te zorgen dat de manier waarop data wordt opgevraagd meer lijkt op de manier waarop de data ook binnen het systeem bijgehouden wordt (Byron, 2015).

2.6 Bestaande literatuur

Aangezien GraphQL op het moment van schrijven nog maar 6 jaar bestaat, is er nog niet heel erg veel onderzoek over gedaan. Dat gezegd zijnde zijn er wel al een aantal interessante studies over gepubliceerd die we hier kort bespreken.

Brito en Valente (2020) onderzochten welke winst men kon maken op vlak van programmeersnelheid door GraphQL te gebruiken in plaats van REST. Ze vonden dat er wel degelijk een groot verschil is tussen de twee, men kon gemiddeld namelijk een derde van de programmeertijd wegwerken door over te schakelen naar GraphQL. Dit geeft aan dat er voordelen zijn van GraphQL naast performantie of features, dat het namelijk ook gemakkelijker is in gebruik.

Er zijn ook een aantal artikels die de overgang beschrijven van softwareprojecten die gebruik maken van REST naar GraphQL, bijvoorbeeld door Brito e.a. (2019). Nadat men een keuze heeft gemaakt tussen de twee (wat het onderwerp is van deze bachelorproef), kunnen dit soort artikels helpen om over te stappen naar GraphQL.

Eizinger (2017) geeft een mooi overzicht van de verschillen tussen GraphQL en REST en de mogelijke voordelen en nadelen van beide. Het is een goede aanvulling voor deze bachelorproef aangezien het dieper ingaat op de theorie, en minder praktisch is van aanpak.

3. Methodologie

Om dit onderzoek uit te voeren is er gekozen om een experiment op te stellen. In dit onderdeel wordt uitgelegd hoe we te werk zijn gegaan om dit experiment op te stellen en uit te voeren.

3.1 Proof-of-concept

Om onze requirements te evalueren en te vergelijken bij REST en GraphQL werd voor elk een proof-of-concept opgesteld. Dit liet toe om een zowel te zien hoe beide systemen in werking zijn, als om hen empirisch te testen met een aantal proef queries. De proof-of-concept applicaties worden uitvoerig besproken in hoofdstuk 4.

3.2 Structuur van de data

Bij een API is de structuur van de data van groot belang, aangezien dit juist hetgeen is dat naar de buitenwereld gebracht wordt. Onze data werd zodanig opgesteld dat er goed getest kon worden op performantie, daarvoor waren twee zaken van groot belang. Als eerste moesten er geneste structuren in zitten, hierdoor werd het mogelijk om te zien welke oplossingen REST en GraphQL boden om deze data toegankelijk te maken. Als tweede was het belangrijk dat er voldoende en voldoende grote (qua opslagruimte) attributen waren van de data-entiteiten, want hierdoor konden de verschillen inzake de hoeveelheid data dat verstuurd wordt naar boven komen. De specifiek gekozen structuur van de data wordt besproken in deel 4.1.1.

3.3 Proef-queries

Om de performantie van beide applicaties te vergelijken, was het noodzakelijk om een aantal voorbeeldqueries op te stellen. Er werden vijf gekozen met een oog voor variatie en volledigheid, dus om een zo goed mogelijk beeld te geven van de mogelijkheden van de APIs om onze data op te vragen. De vijf queries zijn als volgt:

1. Haal de naam en het id op van alle producten.
2. Haal van een categorie met een bepaalde id de naam en beschrijving op en ook het id en gewicht van alle producten binnen deze categorie.
3. Haal voor een keuring met een bepaalde id de kwaliteitsscore op dat toegekend werd, de naam van het product dat gekeurd werd, alsook de organisatie die het keurde.
4. Haal voor een keuring met een bepaalde id de datum van de keuring op, de kleur van het product en de beschrijving van de categorie waartoe het product behoort.
5. Haal voor een product met een bepaalde id de barcode van het product op, de naam en beschrijving van de categorie waar het product toe behoort en de datum en het verslag van elk van zijn keuringen.

3.3.1 Script om proef-queries uit te voeren

Deze queries werden uitgevoerd en getest aan de hand van een script dat de commandline-tool curl gebruikt. Met deze script was het mogelijk om de snelheid waarmee de queries uitgevoerd konden worden te testen en ook de hoeveelheid data dat hiervoor over het netwerk moest gaan. De code voor de script wordt nader toegelicht in deel 4.4.

3.3.2 Server

Omdat web APIs bestemd zijn om over het internet te gaan, was het onvoldoende om de applicaties enkel lokaal te laten draaien en uit te testen, met andere woorden om ze op localhost uit te voeren. Om dichter bij de werkelijkheid te komen werd gebruik gemaakt van een server om de applicaties op uit te voeren. De queries werden dan vanop een computer op een ander netwerk uitgevoerd en gingen over het internet.

De specificaties van de server zijn als volgt:

- CPU: Intel Atom C2350 - 1.7 GHz - 2 core(s)
- RAM: 4GB - DDR3
- Hard Schijf: 1x 1TB (HDD SATA)
- Bandbreedte: 1Gbps
- Besturingssysteem: Linux

4. Proof of Concept

Om REST en GraphQL te toetsen aan de meetcriteria was het noodzakelijk om een proof of concept te schrijven. Aan de hand hiervan konden we een aantal tests uitvoeren en bespreken wat de voor- en nadelen zijn van beide. In dit deel zullen we bespreken met welke technologieen de proof of concept werd opgesteld en hoe we hiervoor te werk gingen.

Omdat er een vergelijking gemaakt wordt tussen GraphQL en REST, is ervoor gekozen om twee verschillende applicaties te schrijven, een gebruik makend van REST en de andere van GraphQL voor de API. Het grootste deel van deze applicaties komt overeen tussen beide, in feite alles behalve de API zelf. Eerst wordt besproken wat beide applicaties gemeen hebben, en daarna waar ze verschillen.

Als proof-of-concept werd gekozen om een applicatie voor producten te maken waar keuringen in bijgehouden kunnen worden. Dit omdat het een typisch voorbeeld is van het soort project waar Ventigrate mee aan de slag gaat.

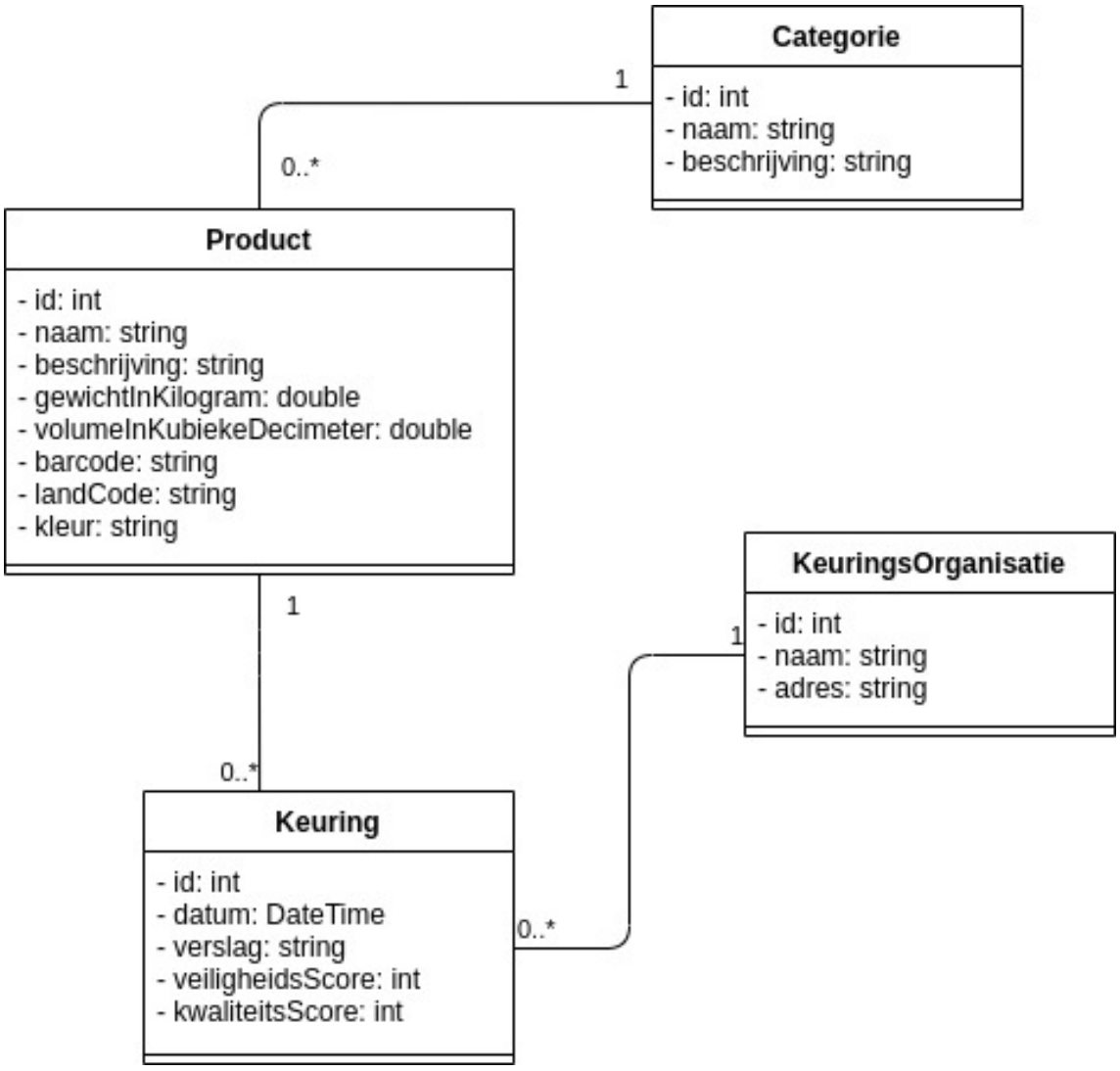
4.1 Gemeenschappelijk tussen beide applicaties

In dit deel wordt uitleg gegeven bij de onderdelen die gemeenschappelijk zijn aan beide applicaties.

4.1.1 Structuur van de data

De structuur van de data is grafisch weergegeven in figuur 2.4. Er zijn producten met een naam, een beschrijving, een gewicht, enz. Elk product heeft ook een categorie en een lijst van keuringen. Deze keuringen hebben op hun beurt een datum, een verslag, scores voor kwaliteit en veiligheid en de organisatie die de keuring heeft uitgevoerd.

Met deze structuur is het mogelijk om een aantal verzoeken uit te voeren, zowel simpele als meer complexe waar geneste data inzit.



Figuur 4.1: Grafische weergave van de structuur van de data

4.1.2 Keuze van platform

We hebben ervoor gekozen om de proof-of-concept in C# op het .NET platform te schrijven, want dat is wat er het meeste bij Ventigrate gebruikt wordt. C# behoort tot de top vijf

meest populaire programmeertalen (Carbonnelle, 2021) en bevat goede tooling voor het opstellen van APIs. We gebruiken .NET 5, wat op het moment van schrijven de recentste stabiele versie is.

4.1.3 Databank

Voor onze databank maakten we gebruik van ORM (Object–relational mapping), want dit liet ons toe om een databank te gebruiken, maar wel nog alles te sturen vanuit ons programma zelf. We gebruiken EF Core, wat het ingebouwd systeem voor ORM is in .NET. Het gekozen databanksysteem is SQLite, maar door het gebruik van EF Core was gelijk welk databanksysteem (bv. MySQL) gangbaar was geweest.

4.1.4 Databank seeding

Omdat we de mogelijkheid wilden hebben om grote hoeveelheden data op te vragen met onze APIs, was het belangrijk dat de databank opgevuld werd. Dit proces heet *databank seeding* en hebben we voor ons project toegepast door gebruik te maken van een library dat *Bogus* heet (Chavez, 2021).

Hieronder een snippet van de code dat gebruikt werd om de data op te bouwen.

```
Randomizer.Seed = new Random(4732891);
[...]
const int aantalCategorieen = 5;
[...]
// Genereer Categorieen.
var categorieId = 1;
var categorieFaker = new Faker<Categorie>()
  .RuleFor(c => c.CategorieId, f => categorieId++)
  .RuleFor(c => c.Naam, f => f.Commerce.Categories(1)[0])
  .RuleFor(c => c.Beschrijving, f => f.Lorem.Paragraph());
var categorieen = categorieFaker.Generate(aantalCategorieen);
[...]
modelBuilder
.Entity<Categorie>()
.HasData(categorieen);
```

Door gebruik te maken van een seed voor de randomizer wordt gegarandeerd dat de code in de databank elke keer dat de databank aangemaakt wordt hetzelfde is.

De categorieën worden aangemaakt met een *faker*, waar regels aan gegeven worden om de verschillende attributen aan te maken. Ten slotte worden de gegenereerde categorieën doorgegeven aan de model builder die ervoor zorgt dat ze in de databank terecht komen.

4.1.5 Docker

Om het uitvoeren van de applicaties gemakkelijker repliceerbaar te maken werd gebruik gemaakt van Docker en Docker Compose, dit liet ons ook toe om de APIs gemakkelijk op een server te runnen, wat voor realistische resultaten zorgt bij het uitvoeren van de tests.

4.2 Specifiek voor REST

Waar de proof of concept pas echt interessant wordt is waar de applicaties van elkaar verschillen. In dit deel worden de onderdelen van de applicatie dat enkel in de REST applicatie zijn besproken.

4.2.1 Controllers

In de applicatie zijn er vier verschillende soorten entiteiten die we naar buiten willen brengen, namelijk producten, categorieën, keuringen en keuringsorganisaties. In de REST versie van onze applicatie krijgen elk van hen een eigen controller. Deze controllers voorzien een aantal methoden waarmee data opgehaald kan worden, in dit geval telkens een methode om alle entiteiten van een soort te krijgen en een om een entiteit met een bepaalde id op te vragen. Daarnaast is er afhankelijk van of er een lijst in zit ook nog een methode om deze lijst op te vragen. In de ProductController zit er bijvoorbeeld een methode *GeefKeuringenVanProductMetId(int id)*.

Hieronder een snippet van de code van de CategorieController.

```
[HttpGet("{id:int}")]
public IActionResult GeefCategorieMetId(int id)
{
    var categorie = _categorieRepo.GeefMetId(id);

    if (categorie == null) return NotFound(
    $"De categorie met id {id} is niet gevonden.");

    return Ok(categorie);
}
```

4.2.2 Endpoints

De methodes in de controllers komen overeen met volgende endpoints:

- /producten
- /producten/{id}
- /producten/{id}/keuringen
- /categorieen

- /categorieen/{id}
- /categorieen/{id}/producten
- /keuringen
- /keuringen/{id}
- /keuringsorganisaties
- /keuringsorganisaties/{id}
- /keuringsorganisaties/{id}/keuringen

Deze endpoints vormen de toegang tot onze data voor de buitenwereld. Elke endpoint komt overeen met telkens een methode in een controller klasse, bijvoorbeeld de code voor `/categorieen/\{id\}` bevindt zich in de `GeefCategorieMetId(int id)` methode van de vorige sectie.

De keuze van endpoints is zeer belangrijk bij REST, want deze mogen nooit veranderd worden zodat de software van de clients blijft werken.

4.2.3 Nesting binnen endpoints

Het is altijd mogelijk om meer endpoints te maken waarmee men meer specifiekere verzoeken mee kan behandelen.

We hadden er bijvoorbeeld voor kunnen kiezen om minder geneste endpoints te maken. Het is namelijk mogelijk om in plaats van een endpoint te hebben om de keuringen van een bepaald product op te vragen (`/producten/\{id\}/keuringen`), gewoon de gebruikers van de API de endpoint voor keuringen te gebruiken (`/keuringen`) en er zelf client-side de juiste informatie uit te halen.

Volgens Microsoft (2018) is het best practice om met een endpoint niet verder te gaan dan collectie/entiteit/collectie. Dat hebben we hier toegepast, daarom dat we bijvoorbeeld geen endpoint hebben zoals `/categorieen/\{id\}/producten/{id}/keuringen`, om deze informatie te krijgen zal de client dan maar meerdere verzoeken moeten uitvoeren. Het is ook duidelijk dat als men voor alle geneste een nieuwe endpoint moet maken, dat het aantal endpoints zeer snel zal toenemen.

Bij het kiezen van de endpoints moet er dus een soort balans gemaakt worden. Enerzijds betekent meer endpoints dat er minder onnodige data doorgestuurd moet worden en dat er minder verzoeken verstuurd moeten worden. Anderzijds maakt het de API moeilijker te onderhouden en is het moeilijk om achteraf nog aanpassingen te maken aan de achterliggende structuur van de data. Als er te veel endpoints zijn, maakt het de API ook minder overzichtelijk voor clients.

4.3 Specifiek voor GraphQL

In dit deel wordt meegedeeld wat de unieke onderdelen zijn van de GraphQL versie van de API.

4.3.1 Query types

In plaats van controllers zoals bij de REST applicatie, wordt er in de GraphQL applicatie gebruik gemaakt van query types. Voor elke soort entiteit die men zou willen opvragen, wordt er een klasse aangemaakt. Zo is er voor categorieën bijvoorbeeld de CategorieQuery klasse. Deze bevat een methode om een enkele categorie op te vragen op basis van zijn id, en een andere methode om alle categorieën op te halen. Deze methoden zijn grotendeels analoog aan de REST versie.

Hieronder een snippet van de code van de CategorieQuery klasse dat dezelfde functionaliteit heeft als het voorbeeld dat bij REST getoond werd in deel 4.2.1.

```
public Categorie GetCategorie(
[Service] IRepository<Categorie> categorieRepo,
IResolverContext context, int id)
{
    var categorie = categorieRepo.GeefMetId(id);
    if (categorie == null)
        context.ReportError(
            ErrorBuilder.New()
                .SetMessage(
                $"De categorie met id {id} is niet gevonden."
                )
                .Build());
    return categorie;
}
```

Bij de REST applicatie waren er ook methoden nodig zoals *GeefKeuringenVanProductMetId(int id)*, maar bij GraphQL is dat niet nodig omdat deze informatie met een query kan opgehaald worden. De query zou er bijvoorbeeld als volgt kunnen uitzien:

```
query {
    product(id: 1) {
        keuringen {
            datum,
            veiligheidsScore
        }
    }
}
```

Afhankelijk van welke data men wil ophalen van de keuringen, kunnen de attributen in de query aangepast worden.

4.4 Script om de applicaties te testen

Zoals uitgelegd in deel 3.3.1 werd er een script opgesteld om de gekozen proef-taken uit te voeren en om de snelheid en datatoevoer te meten. Deze script werd geschreven in bash en de code ervoor is als volgt:

```
#!/usr/bin/env bash

readonly formaat="%{size_download}\t%{time_total}\t"
readonly url="http://163.172.219.242:5500/api"
readonly taken="./taken/*"
readonly output_map="./output/$(date +"%FT_%T")"
readonly aantal_keer_uitvoeren="100"

main() {
    # Maak de output map aan.
    mkdir -p $output_map

    # Ga door alle bestanden in de taken map en voer ze
    # herhaaldelijk uit.
    for bestand in $taken
    do
    herhaal_taak $aantal_keer_uitvoeren $bestand
    echo "Klaar met $(basename $bestand)."
    done
}

voer_taak_uit() {
    local bestand=${1}

    # Ga door alle verzoeken in het bestand en voer ze uit.
    while read verzoek
    do
    curl \
    -s \
    -o /dev/null \
    -w "$formaat" \
    "$url"/"$verzoek"
    done < $bestand
}

herhaal_taak() {
    local aantal_keren=${1}
    local bestand=${2}
    local output_bestand="$output_map/$(basename $bestand).tsv"

    # Voer de taak herhaaldelijk uit en schrijf de uitkomst
```

```
    # in het output bestand.
    for i in $( seq 1 "$aantal_keren")
    do
    voer_taak_uit $bestand >> $output_bestand
    echo >> $output_bestand
    done
}

main
exit
```

De uit te voeren taken worden in aparte bestanden gehouden in de taken map, met telkens een lijntje per verzoek. Deze verzoeken worden uitgevoerd met het commandline-programma *curl*. Elke taak wordt herhaaldelijk gedraaid en de snelheid waarmee de verzoeken gebeurd zijn, alsook de hoeveelheid data dat ermee verstuurd is, wordt opgeslagen in een tsv bestand. Deze data kan dan geanalyseerd worden. In hoofdstuk 5 worden de resultaten van deze tests besproken.

5. Vergelijking REST en GraphQL

In dit hoofdstuk worden REST en GraphQL getoetst aan de requirements en met elkaar vergeleken met behulp van de proof-of-concepts. Ook komen de resultaten van de tests die uitgevoerd werden door middel van het script (zie deel 4.4) hier aan bod.

Hierbij de lijst van functionele en niet-functionele requirements.

Functionele requirements:

- Er moet een mogelijkheid zijn om documentatie automatisch te genereren
- Het moet losgekoppeld zijn van de persistentie
- Het moet mogelijk zijn om de API te updaten

Niet-functionele requirements:

- Het moet stabiel zijn, met weinig dataverlies
- Het moet snel zijn
- Er moeten libraries zijn voor de meest gebruikte programmeertalen
- De hoeveelheid data dat over het netwerk gaat moet beperkt zijn
- Het aantal verzoeken moet beperkt zijn.

Deze worden nu een voor een behandeld en vergeleken tussen REST en GraphQL. Voor elke requirement werd er een score op 5 gegeven, dit liet ons toe om de vergelijking duidelijk te maken.

5.1 Automatische generatie van documentatie

Documentatie is een heel belangrijk onderdeel van een API, want zonder documentatie is het onmogelijk voor gebruikers van de API te weten welke verzoeken uitgevoerd kunnen worden. Men kan manueel documentatie schrijven, maar het is een groot voordeel als (tenminste een deel) van de documentatie automatisch gegenereerd kan worden.

5.1.1 REST

In de thesis van Fielding (2000) waar REST in uitgelijnd staat, worden er geen vereisten opgelegd aan documentatie. In feite is de programmeur dus helemaal vrij om zelf zijn documentatie op te stellen zoals hij dat wil. Het nadeel hiervan is dat er niets van documentatie automatisch kan worden aangemaakt.

Toch is er een manier om automatisch documentatie te laten genereren met REST, namelijk door gebruik te maken van een specificatie bovenop REST zoals OpenAPI (ook gekend als swagger). Dat hebben we in onze proof-of-concept gedaan en dat liet ons toe om documentatie te laten aanmaken zonder het volledig manueel te moeten doen.

Dit werd gedaan door simpelweg een paar lijnen code toe te voegen aan de Startup.cs file:

```
[...]
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo {
        Title = "KeuringAppREST", Version = "v1"});
});
[...]
app.UseSwagger();
app.UseSwaggerUI(c => c.SwaggerEndpoint(
"/swagger/v1/swagger.json", "KeuringAppREST v1"));
[...]
```

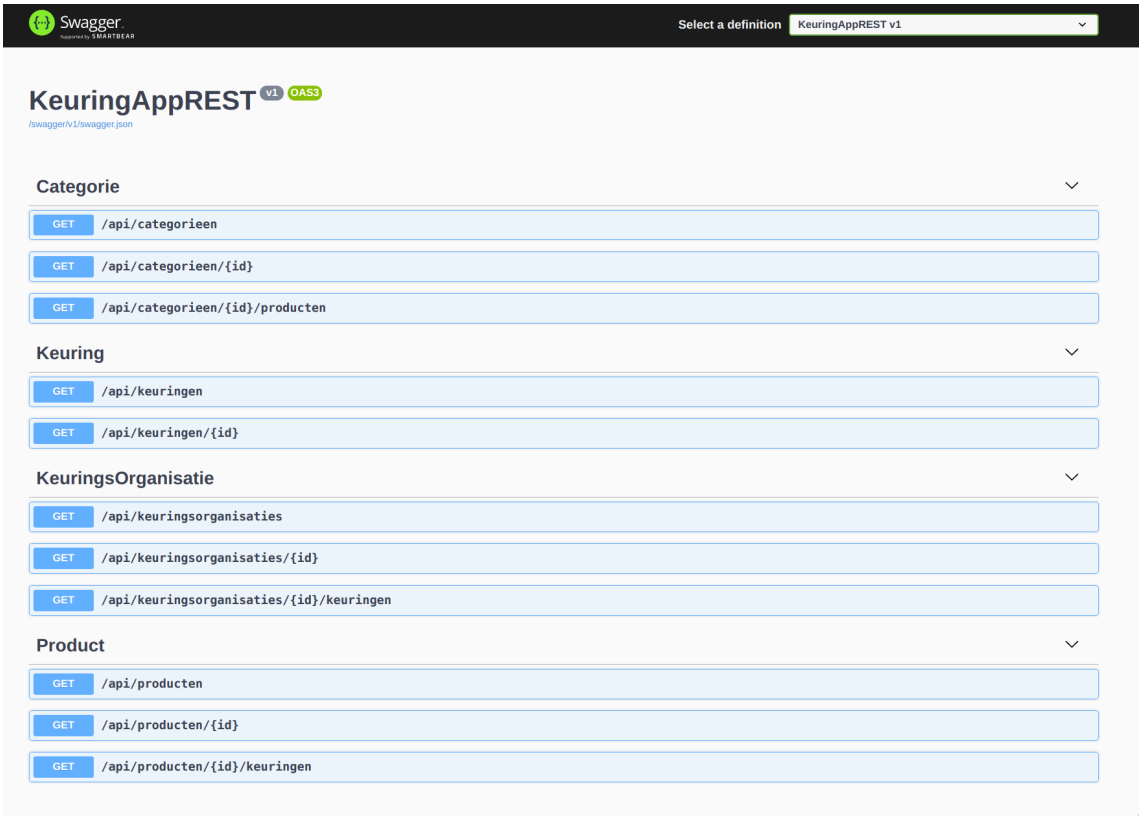
Hierdoor wordt dan automatisch de documentatiepagina aangemaakt die te zien is in figuur 5.1

Op deze pagina is een overzicht te zien van de verschillende endpoints die gebruikt kunnen worden, en bestaat ook de mogelijkheid om ze hier uit te proberen.

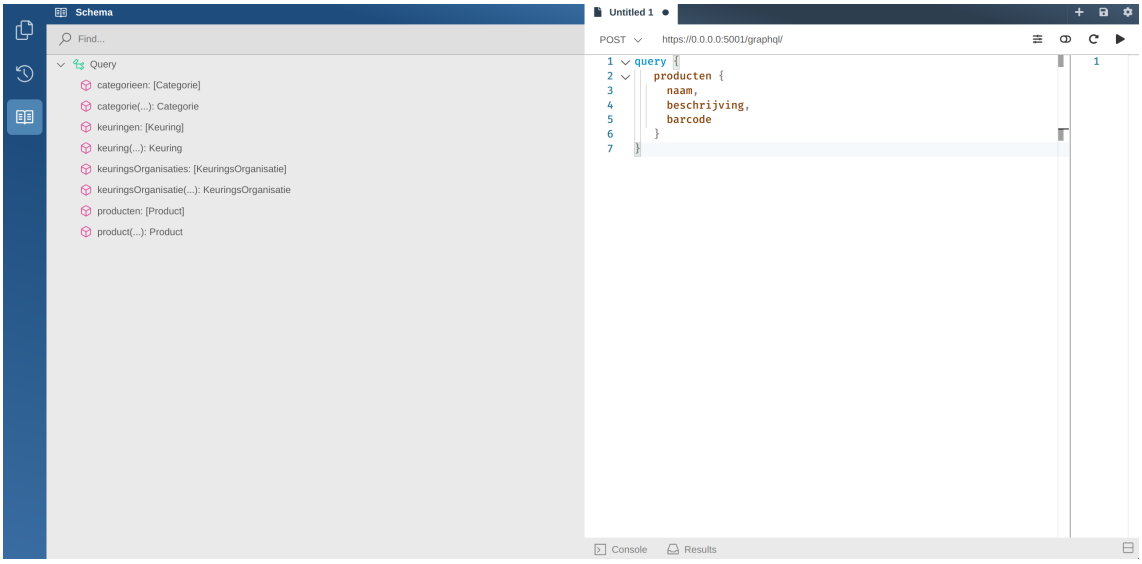
5.1.2 GraphQL

Bij GraphQL staat er een vereiste voor documentatie vermeld in de specificatie, hierdoor ben je gegarandeerd dat als er gebruik wordt gemaakt van GraphQL dat er automatisch documentatie aangemaakt wordt (Facebook, 2018).

Bij de GraphQL proof-of-concept ziet die documentatie er uit zoals te zien is in figuur 5.2.



Figuur 5.1: Automatisch gegenereerde documentatiepagina voor REST



Figuur 5.2: Automatisch gegenereerde documentatiepagina voor GraphQL

Op deze pagina ziet men links de mogelijke queries die uitgevoerd kunnen worden en rechts is het mogelijk om queries uit te testen met automatisch aanvullende suggesties.

5.1.3 Scores

De functionaliteit om documentatie automatisch te genereren zit bij GraphQL ingebakken, maar bij REST moet men een extra library gebruiken.

Hierdoor krijgt GraphQL een lichte voorkeur boven REST.

REST	GraphQL
4	5

Tabel 5.1: Scores automiatische generatie van documenatie

5.2 Loskoppeling van de persistentie

Zowel REST als GraphQL stellen geen eisen aan de achterliggende implementatie voor de persistentie van data. Het is dus perfect mogelijk om een relationele databank te gebruiken, maar een NoSQL databank of nog iets anders is ook een optie.

Het is zelfs mogelijk om bijvoorbeeld voor een REST of GraphQL API geen persistentie te hebben, maar in plaats daarvan de data ophaalt van andere APIs. Zo kan men een bestaande REST API blootleggen met een GraphQL API als men dat gunstig vindt.

In onze proof-of-concept hebben we ervoor gekozen om de gangbare relationele databank te gebruiken, weliswaar via een ORM-systeem (zie deel 4.1.3 voor meer info).

5.2.1 Scores

REST en GraphQL voldoen beide volledig aan de requirement, hierdoor krijgen ze elk 5/5.

REST	GraphQL
5	5

Tabel 5.2: Scores loskoppeling van de persistentie

5.3 Beschikbaarheid van libraries in programmeertalen

Zowel REST als GraphQL kunnen gebruikt worden in gelijk welke programmeertaal. Dit omdat ze elk geen implementatiedetails specificeren. In plaats hiervan bepalen elk een aantal regels waaraan voldaan moet worden.

Hoewel het dus perfect mogelijk om zowel REST als GraphQL in gelijk welk project te gebruiken, zal men in de werkelijkheid voorkeur geven aan het gebruik van een al bestaande implementatie in plaats van er zelf een te schrijven. Als het harde werk al door een andere gedaan is, heeft het weinig zin om het wiel te heruitvinden.

We hebben gekeken naar de meest populaire programmeertalen en geven een overzicht van of er wel of geen libraries bestaan voor GraphQL en voor REST.

De populariteit van programmeertalen vergelijken is geen simpele opdracht. Veel bedrijven achterhouden namelijk de details van hun achterliggende implementaties, en dus kan het moeilijk zijn om te achterhalen welke juist gebruikt worden. Omwille hiervan is ervoor geopteerd om de PYPL index te gebruiken (Carbonnelle, 2021). Deze lijst maakt gebruik van het aantal uitgevoerde zoekopdrachten op zoekmachines om een schatting te maken van de populariteit van programmeertalen. Het is dus geen perfecte lijst, maar het volstaat voor onze behoeftes.

Rang	Programmeertaal	Aandeel	REST	GraphQL
1	Python	29.9 %	x	x
2	Java	17.72 %	x	x
3	JavaScript	8.31 %	x	x
4	C#	6.9 %	x	x
5	C/C++	6.62 %	x	x
6	PHP	6.15 %	x	x
7	R	3.93 %	x	x
8	Objective-C	2.52 %	x	x
9	Swift	1.96 %	x	x
10	TypeScript	1.89 %	x	x
11	Matlab	1.71 %	x	x
12	Kotlin	1.62 %	x	x
13	Go	1.42 %	x	x
14	VBA	1.33 %	x	
15	Rust	1.13 %	x	x

Tabel 5.3: Meest populaire programmeertalen volgens PYPL

Er werd gekeken naar de top 15 meest populairste talen. Zoals te zien is op tabel 5.3 is er dus zowel voor GraphQL als voor REST library support in de populairste programmeertalen, op een taal na (VBA bij GraphQL).

5.3.1 Scores

Libraries zijn voor beide technologieën beschikbaar in de meeste programmeertalen, maar REST heeft toch net iets meer dekking.

Hierdoor krijgt REST een lichte voorkeur boven GraphQL.

REST	GraphQL
5	4

Tabel 5.4: Scores beschikbaarheid van libraries in programmeertalen

5.4 API updaten

APIs zijn vaak constant aan het evolueren. Als men nieuwe functionaliteit wil toevoegen, moet dit mogelijk zijn. Liefst gebeurt dit ook op een manier waarop de gebruikers van de API (dus de clients) niets moeten aanpassen aan hun software en dat alle oude functionaliteit dus op dezelfde manier blijft werken.

Om dit te garanderen zijn er twee opties:

- versioning
- een doorlopende evolutie van eenzelfde versie

In het algemeen gebruikt REST versioning, en GraphQL niet, maar hier bestaat wel enige nuance in. Dit wordt later nog uitgelegd.

Met 'versioning' wordt bedoeld dat er telkens als men aanpassingen wil maken aan de API, er een nieuwe versie gemaakt wordt.

5.4.1 REST

Hoewel versioning geen vereiste is bij REST, het wordt namelijk helemaal niet vermeld in de originele paper die rest definieert, wordt het in de werkelijkheid wel zeer vaak gebruikt (Sohan e.a., 2015). De reden hiervoor is dat de manier waarop REST is opgesteld, het bijna noodzaakt om versioning te gebruiken. Het is namelijk zo dat REST APIs een eindig en op voorhand bepaald aantal endpoints hebben. Deze endpoints geven vaak heel wat data mee met een heel specifieke structuur. De structuur mag nooit veranderen, want daardoor zouden bestaande programma's breken. Als de maker van de API dus een kleine aanpassing wil maken, is dit vaak niet mogelijk. Tenzij men gebruik maakt van versioning.

Elke versie krijgt een ander endpoint URL. Zo zouden de endpoints voor verschillende versies er bijvoorbeeld uitzien om alle producten op te halen:

- https://www.voorbeeldapi.be/api/1/producten
- https://www.voorbeeldapi.be/api/2/producten
- https://www.voorbeeldapi.be/api/3/producten
- enz.

Dit zorgt ervoor dat programmeurs die gebruik maken van deze REST API er zeker van kunnen zijn dat de functionaliteit van de versie die ze gebruiken nooit zal veranderen. Dit heeft voor- en nadelen. Het voordeel is dat hun code altijd zal blijven werken en dat het niet zal mislopen omdat de API veranderd is. Het nadeel is dat als men gebruik wil maken van nieuwe functionaliteit, dat men zal moeten overgaan naar een hogere versie van de API. Omdat de makers weten dat de API mooi opgesplitst is in versies, zullen ze vaak grote veranderingen brengen in de structuur van de API, dus welke endpoints gebruikt kunnen worden en welke data deze teruggeven. Hierdoor kan het zijn dat bij het overgaan naar een nieuwe versie van de API, de programmeur heel wat veranderingen moet aanbrengen in zijn programma.

5.4.2 GraphQL

Bij GraphQL is geopteerd om geen versioning te gebruiken. Dit is mogelijk door het feit dat er bij GraphQL veel exacter gespecificeerd kan worden welke data men wil ophalen. In tegenstelling tot REST, waarbij de makers van de API moeten proberen anticiperen wat de noden van de gebruikers gaat zijn, moet men bij GraphQL hier geen rekening mee houden, want de gebruikers stellen zelf hun eigen queries op.

Hierdoor is er ook geen nood aan versioning. Als men een nieuw veld aan de data wil toevoegen, kan men dat gewoon doen. Als gebruikers van de GraphQL API deze willen opvragen, doen ze dat gewoon. Als ze het niet willen gebruiken, vragen ze het niet op en wordt het ook niet meegegeven.

Stel dat we een verandering willen maken in onze proof-of-concept API. Om de datum en het verslag van een keuring op te halen kan men volgende query gebruiken:

```
query {
    keuring(id: 1) {
        datum,
        verslag
    }
}
```

Stel nu dat beslist wordt om het formaat van het verslag aan te passen van string naar een pdf-bestand. Naast het veld *verslag* kan men dan een veld verslag_pdf maken. We behouden het oud veld, want we willen de functionaliteit van de clients niet breken. Een client kan dus dezelfde query blijven gebruiken en het zal blijven werken. Een client die gebruik wil maken van het nieuw formaat doet dit door middel van volgende query:

```
query {
    keuring(id: 1) {
        datum,
        verslag_pdf
    }
}
```

Om duidelijk te maken aan de klanten dat de oude versie van het verslag niet de nieuwste versie is dat bij voorkeur gebruik wordt gemaakt van het nieuw veld, kan men het deprecated annotatie gebruiken om aan te geven dat het gebruik ervan verouderd is. Dan zal dit in de documentatie getoond worden naast het oud veld. Toegepast op ons voorbeeld ziet dat er zo uit:

```
type Keuring {
    datum,
    verslag @deprecated(reason: "gebruik verslag_pdf"),
    verslag_pdf
}
```

Het is uiteraard ook mogelijk om ook bij REST het nieuwe veld gewoon toe te voegen, maar dit zorgt wel voor onnodig dataverkeer, aangezien beide velden dan verstuurd worden. Bij een string zoals verslag is dat misschien zo veel data niet, maar bijvoorbeeld bij een video file zou dat wel heel wat data kunnen zijn als zowel het oude als het nieuwe formaat verstuurd moeten worden.

5.4.3 Scores

REST legt niets op over hoe een API geüpdatet moet worden, maar omwille van de manier waarop het is opgebouwd is men vaak genoodzaakt om versioning te gebruiken. GraphQL daarentegen heeft geen nood aan versioning. Dit geeft een GraphQL API meer flexibiliteit dan een REST API.

Er werd gekozen om REST een score van 3/5 te geven en GraphQL een score van 5/5.

REST	GraphQL
3	5

Tabel 5.5: Scores API updaten

5.5 Resultaten proef-queries

De resultaten van de vijf queries die werden uitgelijnd in hoofdstuk 3 en uitgevoerd werden met het script dat werd besproken in 4.4 komen in dit deel aan bod.

Hier nogmaals de opsomming van de proef-taken:

1. Haal de naam en het id op van alle producten.
2. Haal van een categorie met een bepaalde id de naam en beschrijving op en ook het id en gewicht van alle producten binnen deze categorie.
3. Haal voor een keuring met een bepaalde id de kwaliteitsscore op dat toegekend werd, de naam van het product dat gekeurd werd, alsook de organisatie die het keurde.
4. Haal voor een keuring met een bepaalde id de datum van de keuring op, de kleur van het product en de beschrijving van de categorie waartoe het product behoort.
5. Haal voor een product met een bepaalde id de barcode van het product op, de naam en beschrijving van de categorie waar het product toe behoort en de datum en het verslag van elk van zijn keuringen.

In de volgende delen worden het aantal benodigde verzoeken, de hoeveelheid data dat verstuurd moest worden en de snelheid waarmee het werd uitgevoerd, besproken.

5.6 Aantal verzoeken

Het aantal verzoeken dat nodig was voor elk van de proef-taken is opgelijst in tabel 5.6

Taak	REST	GraphQL
1	1	1
2	2	1
3	3	1
4	3	1
5	3	1
Gemiddelde	**2,4**	**1**

Tabel 5.6: Aantal verzoeken nodig per taak

Bij heel simpele verzoeken zoals taak 1 die perfect overeenkomen met een van de REST endpoints, is er maar een verzoek nodig. Bij meer complexe queries, waarbij er aan geneste data moet gekomen worden, zijn er 2 of 3 verzoeken nodig. In een nog complexere structuur met meer entiteiten met onderlinge relaties zou dit nog kunnen oplopen.

Bij GraphQL is er telkens maar een verzoek nodig. Het is dan ook het doel van GraphQL om in een verzoek alle data te kunnen ophalen die men wilt (Byron, 2015), ook geneste data. Het probleem dat er te veel verzoeken gedaan moeten worden heet *underfetching* en was een van de grote problemen van REST waar GraphQL een oplossing voor wou bieden.

5.6.1 Scores

Aangezien REST gemiddeld 2.4 keer het aantal verzoeken nodig had dat GraphQL nodig had, krijgt REST een score van 2/5 en GraphQL een score van 5/5.

REST	GraphQL
2	5

Tabel 5.7: Scores aantal verzoeken

5.6.2 Hoeveelheid data

De hoeveelheid data in bytes dat verstuurd moest worden over het netwerk voor elke taak is te zien in tabel 5.8

Taak	REST	GraphQL
1	419 866	48 962
2	37 702	5 303
3	795	134
4	1 090	297
5	1 681	1 117
Gemiddelde	**92 226,8**	**11 162,6**

Tabel 5.8: Hoeveelheid data per taak in bytes

Bij GraphQL moest er dus aanzienlijk minder data over het netwerk gaan om de taken uit te voeren dan bij REST. Dit verbaast niet, want GraphQL is ervoor gemaakt om net de data op te halen die men nodig heeft, niet meer en niet minder. Het probleem dat er bij REST soms te veel data verstuurd wordt heet *overfetching* en is een ander groot probleem dat GraphQL wou aanpakken.

5.6.3 Scores

REST moest bij onze queries gemiddeld meer dan acht keer zoveel data over het netwerk versturen als GraphQL. REST krijgt hierdoor een score van 1/5 en GraphQL een score van 5/5.

REST	GraphQL
1	5

Tabel 5.9: Scores hoeveelheid data

5.7 Snelheid

In tabel 5.10 worden de snelheidsresultaten voor de REST API weergegeven voor elke query.

Taak	Gemiddelde	Mediaan	Minimale waarde	Maximale waarde	Standaarddeviatie
1	585	571	535	1067	61,0
2	146	144	128	185	8,3
3	215	191	179	1209	143,2
4	147	146	133	190	7,7
5	179	161	153	1183	113,9

Tabel 5.10: Tijd in ms per taak in REST

De resultaten voor de GraphQL API zijn te vinden in tabel 5.11

Taak	Gemiddelde	Mediaan	Minimale waarde	Maximale waarde	Standaarddeviatie
1	496	478	462	982	77,2
2	54	53	47	72	4,3
3	42	42	34	60	4,0
4	42	42	35	54	3,3
5	58	57	50	79	4,5

Tabel 5.11: Tijd in ms per taak in GraphQL

Elke taak is 100 keer uitgevoerd geweest per API om toeval te vermijden.

Bij GraphQL liggen de gemiddelde snelheden dus beduidend lager dan bij REST. Ook de standaarddeviatie ligt in de meeste gevallen lager, wat erop duidt dat GraphQL ook meer consequent is dan REST.

Dit verschil in snelheid is voor het grootste deel te wijten aan de grotere hoeveelheid data dat bij REST over het netwerk moet gaan in vergelijking met GraphQL.

5.7.1 Scores

De gemiddelde tijd over alle taken heen is bij REST 254,4 ms en bij GraphQL 138,4 ms, wat betekent dat GraphQL meer dan 1,8 keer sneller is dan REST.

Daarom werd besloten om REST een score te geven van 3/5 en GraphQL een score van 5/5.

REST	GraphQL
3	5

Tabel 5.12: Scores snelheid

5.8 Dataverlies/stabiliteit

	Dataverlies (bytes)	Aantal gefaalde verzoeken
REST	0	0
GraphQL	0	0

Tabel 5.13: Dataverlies en aantal gefaalde verzoeken

In alle voorgaande tests is er geen een keer sprake geweest van dataverlies of gefaalde verzoeken, zowel bij REST als bij GraphQL niet. Aangezien het gaat over 500 queries bij elk (100 per taak), kunnen we concluderen dat beide API technologieën stabiel zijn en geen problemen hebben met dataverlies.

5.8.1 Scores

Zowel REST als GraphQL vertoonden geen dataverlies of gefaalde verzoeken, ze krijgen allebei een score van 5/5.

REST	GraphQL
5	5

Tabel 5.14: Scores dataverlies en stabiliteit

6. Conclusie

Dit onderzoek is gestart met de vraag wat een betere technologie is om te gebruiken voor een data-gebaseerde API, REST of GraphQL? Dit werd met behulp van de gemaakte proof-of-concepts getoetst aan de hand van enkele functionele en niet-functionele requirements. Voor elke requirement werd er een score op 5 gegeven en die zijn te zien in tabel 6.1.

Requirement	REST	GraphQL
Automatische generatie van documentatie	4	5
Loskoppeling van de persistentie	5	5
Beschikbaarheid van libraries in programmeertalen	5	4
Updatebaarheid	3	5
Beperkt aantal verzoeken	2	5
Beperkte hoeveelheid data	1	5
Snelheid	3	5
Dataverlies/stabiliteit	5	5

Tabel 6.1: Scores van REST en GraphQL

Het antwoord op onze onderzoekvraag is helder naar boven gekomen doorheen het onderzoek. De conclusie is duidelijk, GraphQL scoorde even goed of beter op bijna alle requirements en is dus de betere keuze. Binnen deze context is de enige reden om REST te gebruiken omdat het beter gekend is onder programmeurs, hierdoor is de wrijving eventueel wat minder om aan een nieuw project te beginnen. Maar de performantie voordelen van GraphQL zijn moeilijk om naast te kijken, dus is het aan te raden voor elke organisatie dat een API maakt om de overstap naar GraphQL serieus te overwegen.

Dit onderzoek heeft gebruik gemaakt van een beperkte en artificiële databank, verder onderzoek zou performantietests kunnen uitvoeren op een meer complexe en grotere

hoeveelheid data dat op natuurlijke wijze ontstaan is om na te gaan of GraphQL het daarbij ook beter doet dan REST.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Vandaag de dag worden API's (Application Programming Interface) uitgebreid gebruikt, zowel intern als extern. Dit omdat het een aantal heel belangrijke sterktes heeft. Een van de belangrijkste hiervan is *Separation of Concerns*, in dit geval om verschillende programma's van elkaar gescheiden te houden, maar toch nog te laten communiceren met elkaar. Deze communicatie gebeurt typisch over een netwerk. Door gebruik te maken van een API moeten de aparte programma's niet de onderliggende werking van elkaar kennen en kunnen ze hierdoor veel gemakkelijker data opvragen en wijzigen.

Op dit moment is REST (Representational state transfer) het meest gebruikte paradigma om de structuur van een API te bepalen (Motroc, 2017). Het is een duidelijk uitgelijnd en gestructureerd systeem dat bestaat uit een aantal regels die gevolgd moeten worden en methodes die dienen te worden voorzien (Fielding, 2000). Het grootste voordeel ervan is dat het relatief simpel is om te gebruiken en dat velen er al ervaring mee hebben omdat het zo alomtegenwoordig is. Hierdoor is het heel gemakkelijk om met een nieuw RESTful API aan de slag te gaan.

REST heeft wel een aantal belangrijke nadelen. Omwille van de restricties die het oplegt moeten er vaak meerdere verzoeken gedaan worden, wat inefficiënt kan zijn. Ook is er een probleem gekend als *overfetching*, dit wil zeggen dat men vaak veel meer data terugkrijgt van de server dan men nodig heeft. Dit belast het netwerk onnodig.

GraphQL werd ontworpen in 2012 als interne tool bij Facebook (Byron, 2015) en had als doel om de vermelde zwaktes van REST te verbeteren. Met GraphQL is het namelijk mogelijk om zeer specifiek enkel de gewenste data op te vragen en dit met minder verzoeken. Sinds 2015 is het beschikbaar voor iedereen en is de broncode open-source geworden.

Omdat REST de standaard is voor API's en GraphQL een nieuwe speler is, kan men zich gaan afvragen of het de moeite waard is om tegen de stroom in te gaan in plaats van gewoon bij REST te blijven. Het doel van deze bachelorproef is om te onderzoeken wat de juiste keuze is voor een datagestuurd project. Er zal bekeken worden of de complexiteit van de data (hoeveel nesting er is) een rol speelt. Ook de hoeveelheid data dat opgevraagd wordt kan een verschil maken en zal worden onderzocht. Hetgeen dat getest zal worden bij verschillende taken (bijv. nieuwe gegevens toevoegen, bestaande gegevens opvragen) op onze data is het volgende:

- de grootte van de data dat over het netwerk verstuurd wordt
- de snelheid waarmee de taak wordt uitgevoerd
- het aantal verzoeken

A.2 State-of-the-art

Er is nog niet erg veel onderzoek gedaan over GraphQL omdat het pas zo recent ontworpen is. Dat gezegd zijnde, zijn er wel een aantal interessante studies over gepubliceerd.

Brito en Valente (2020) onderzochten welke winst men kon maken op vlak van programmeersnelheid door GraphQL te gebruiken in plaats van REST. Ze vonden dat er wel degelijk een groot verschil is tussen de twee, men kon gemiddeld namelijk een derde van de programmeertijd wegwerken door over te schakelen naar GraphQL. Dit is een zeer interessant artikel omdat het sterke evidentie geeft dat er voordelen zijn voor GraphQL naast pure efficiëntie. Het vult de inhoud van deze bachelorproef dus goed aan door te focussen op het gemak van programmeren, wat moeilijk te onderzoeken zou zijn zonder proefpersonen.

Er zijn ook een aantal artikels die de overgang beschrijven van softwareprojecten die gebruik maken van REST naar GraphQL, bijvoorbeeld door Brito e.a. (2019). Dit kan zeer interessant zijn eens men de keuze heeft gemaakt om te migreren naar GraphQL van REST, maar om die keuze te maken zou deze bachelorproef kunnen helpen.

Eizinger (2017) geeft een mooi overzicht van de verschillen tussen GraphQL en REST en de mogelijke voordelen en nadelen van beide. Het is echter zeer theoretisch en academisch, deze bachelorproef zou meer praktisch zijn en empirisch onderzoeken wat de verschillen zijn qua performantie.

A.3 Methodologie

Om te onderzoeken wat de potentiële voordelen zijn van GraphQL tegenover REST zal er een simpel project opgesteld worden, we kiezen voor een applicatie waar keuringen bijgehouden worden voor producten omdat dit het soort project is waar Ventigrate typisch aan werkt. We zullen zowel een scenario bekijken met heel simpele data, als een scenario met een meer complexe, geneste structuur. Ook zal er onderzocht worden welk verschil de hoeveelheid data maakt, namelijk of er maar enkele datapunten zijn of (zoals regelmatig het geval is in een gelijkaardig project) duizenden.

Voor het voorbeeldproject zal Express.js gebruikt worden, dit is een framework bovenop Node en laat toe om gemakkelijk en met weinig code een simpele server op te stellen die in staat is om HTTP-verzoeken te behandelen. De tool Postman zal gebruikt worden om de verzoeken naar de server te sturen en de snelheid te meten.

A.4 Verwachte resultaten

Er wordt verwacht dat het aantal benodigde verzoeken bij GraphQL lager zullen liggen dan bij REST. De hoeveelheid data dat over het netwerk verstuurd moet worden zal ook lager zijn. Bij grote en complexe datasets is de verwachting dat GraphQL efficiënter zal zijn per taak. GraphQL zou dus op alle onderzochte criteria een verbetering zijn op de traditionele REST.

A.5 Verwachte conclusies

Aangezien verwacht wordt dat GraphQL REST zal verbeteren op elk mogelijk criteria, zal de conclusie waarschijnlijk zijn dat GraphQL de beste keuze is. Enkel voor een softwareproject met weinig data dat niet complex is, zou REST de voorkeur hebben aangezien het meer simpel is om op te zetten en de voordelen van GraphQL niet of nauwelijks tot uiting kunnen komen.

Bibliografie

Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F. & Secret, A. (1994). The world-wide web. *Communications of the ACM*, *37*(8), 76–82.

Bradshaw, A. C. (2001). Internet users worldwide. *Educational Technology Research and Development*, *49*(4), 112–117.

Brito, G., Mombach, T. & Valente, M. T. (2019). Migrating to GraphQL: A practical assessment. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 140–150.

Brito, G. & Valente, M. T. (2020). REST vs GraphQL: A Controlled Experiment.

Byron, L. (2015). *GraphQL: A data query language*. https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/

Carbonnelle, P. (2021). *PYPL PopularitY of Programming Language*. https://pypl.github.io/PYPL.html

Chavez, B. (2021). *Bogus - A simple and sane fake data generator for C#, F#, and VB.NET.* https://github.com/bchavez/Bogus

Eizinger, T. (2017). API design in distributed systems: a comparison between GraphQL and REST.

Espinha, T., Zaidman, A. & Gross, H.-G. (2015). Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, *100*, 27–43.

Facebook. (2018). *GraphQL Specification, 2018 edition*. https://spec.graphql.org/June2018/

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Deel 7). University of California, Irvine Irvine.

IEEE. (2018). *The Open Group Base Specifications Issue 7, 2018 edition*. https://pubs.opengroup.org/onlinepubs/9699919799/

Martin, J. (1983). *Managing the data base environment*. Prentice Hall PTR.

Martin, R. C. (2018). *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall.

Microsoft. (2018). *Web API design*. https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

Motroc, G. (2017). *The State of API Integration: SOAP vs. REST, public APIs and more*. https://jaxenter.com/state-of-api-integration-report-136342.html

Nayak, A., Poriya, A. & Poojary, D. (2013). Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, *5*(4), 16–19.

Reddy, M. (2011). *API Design for C++*. Elsevier.

Sohan, S., Anslow, C. & Maurer, F. (2015). A case study of web API evolution. *2015 IEEE World Congress on Services*, 245–252.