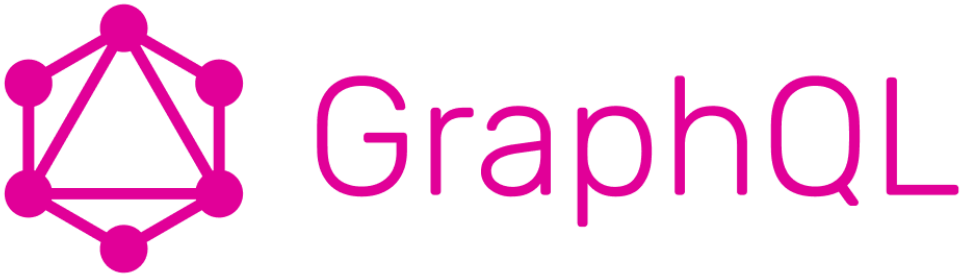
Afstudeerverslag

Onderzoek naar het gebruik van GraphQL binnen de API van Moneybird



Naam	Wouter Kamp
Studentnummer	423078
School	Saxion Enschede
Opleiding	HBO-ICT Software Engineering
Opdrachtgever	Moneybird Holding B.V.
Afstudeerbegeleider	M. Siekmans
Bedrijfsbegeleiders	Gijs Abbring & Hugo Olthof
Datum	18 juni 2023

Afstudeerverslag

Onderzoek naar het gebruik van GraphQL binnen de API van Moneybird

Gegevens student

Naam	Wouter Kamp
Studentnummer	423078
Opleiding	HBO-ICT Saxion Enschede
Email	wouter@moneybird.nl 423078@student.saxion.nl

Gegevens afstudeerdocent

Naam	M. Siekmans
Email	m.siekmans@saxion.nl

Gegevens bedrijf

Naam	Moneybird
Website	www.moneybird.nl
Adres	Moutlaan 35
Postcode en plaats	7523 MC Enschede
Email	support@moneybird.nl

Gegevens begeleider 1

Naam	Gijs Abbring
Functie	Software developer
Opleiding	HBO-ICT Saxion Enschede
Email	gijs@moneybird.nl

Gegevens begeleider 2

Naam	Hugo Olthof
Functie	Software developer
Opleiding	HBO-ICT Saxion Enschede
Email	hugo@moneybird.nl

Versiebeheer

Versie	Wijzigingen	Datum
0.1	Initiële opzet	4 juni 2023
0.2	Feedback docent verwerkt	15 juni 2023
1.0	Feedback bedrijfsbegeleiders verwerkt	18 juni 2023

Tabel 0-1: Versiebeheer

Voorwoord

Voor u ligt het afstudeerverslag “Onderzoek naar het gebruik van GraphQL binnen de API van Moneybird”. Dit verslag is geschreven als afronding van de studie HBO-ICT Software-Engineering aan Saxion Enschede.

Tijdens de afstudeerperiode van 17 februari tot 7 juli heb ik onderzoek gedaan naar de voor- en nadelen van het gebruik van GraphQL binnen Moneybird. In dit document is te lezen hoe dit onderzoek is verlopen en tot welke conclusies dit onderzoek heeft geleid.

Bij dit onderzoek ben ik vanuit Moneybird begeleid door Gijs Abbring en Hugo Olthof. Graag wil ik hen bedanken voor hun begeleiding tijdens dit traject. Zij hebben mij geholpen tijdens alle stappen van het afstuderen. Zonder hun begeleiding had ik dit afstudeerproject niet af kunnen ronden.

Ook wil ik mijn vriendin LeAnn bedanken voor haar steun tijdens dit traject. Tot slot wil ik ook mijn afstudeerdocent, Manasse Siekmans, bedanken voor zijn begeleiding tijdens dit afstudeertraject.

Ik wens u veel plezier toe tijdens het lezen van dit document.

Wouter Kamp

Enschede, 18 juni 2023

Samenvatting

Moneybird maakt een boekhoudpakket voor ondernemers om het bijhouden van de boekhouding gemakkelijk te maken. Dit boekhoudpakket met dezelfde naam beschikt ook over een REST API. Met deze API biedt Moneybird de mogelijkheid aan ontwikkelaars om software te maken die integreert met Moneybird. Moneybird is altijd op zoek naar manieren om haar applicatie te moderniseren. Een van deze manieren is het gebruik van GraphQL. In dit verslag is te lezen wat de voor- en nadelen zijn van de integratie van GraphQL in de Moneybird applicatie.

Tijdens dit onderzoek zijn GraphQL en REST op vier vlakken vergeleken. Tijdens het eerste deel van het onderzoek is onderzocht waar de verschillen liggen in het gebruik van GraphQL en REST vanuit het perspectief van de gebruiker van de API. Daarna is onderzocht waar de verschillen liggen vanuit het perspectief van de Moneybird ontwikkelaar. Het derde deel van het onderzoek gaat in op de integratie van GraphQL in Moneybird. Tot slot is onderzocht waar de verschillen liggen op het gebied van performance.

Uit dit onderzoek is gebleken dat GraphQL veel voordelen met zich meebrengt. Zo kan een gebruiker met GraphQL zelf aangeven welke data van de server verwacht wordt. Dit zorgt ervoor dat de server geen onnodige data hoeft te versturen. Ook maakt GraphQL het mogelijk om in één request data op te halen die gerelateerd is aan het opgevraagde object. Bij de klassieke REST API zouden hier meerdere requests voor nodig zijn.

Met behulp van bestaande GraphQL libraries voor Ruby on Rails is GraphQL geïmplementeerd in de Moneybird applicatie. Deze implementatie is zonder grote problemen verlopen, mede dankzij de stabiele codebase van Moneybird, en vormt een basis voor de verdere ontwikkeling van de GraphQL API.

Naast het gebruiksgemak biedt GraphQL ook voordelen op het gebied van performance. Een belangrijk aspect hiervan is de mogelijkheid om zelf aan te geven welke gegevens van de server verwacht worden. Hierdoor hoeft minder data verstuurd te worden, waardoor de latency over het algemeen lager ligt.

Op basis van de resultaten van het onderzoek is het advies voor Moneybird om zowel de GraphQL API als de bestaande REST API te ondersteunen. Op deze manier kan de nieuwe functionaliteit aangeboden worden aan klanten, terwijl backward compatibility behouden wordt.

Inhoudsopgave

Figuren- en tabellenlijst

Begrippenlijst

Begrip	Definitie
API	De afkorting API staat voor Application Programming Interface. Deze interface zorgt ervoor dat het voor andere software mogelijk is om met een programma te communiceren.
API token	Een API token is een sleutel die meegegeven kan worden bij het communiceren met een API. Deze sleutel maakt het mogelijk om de gebruiker te identificeren. Een API token hoort altijd bij een enkele gebruiker.
API documentatie	Een technische documentatie van een API. Deze bevat instructies voor het gebruik van de API.
API request	Een API request is een medium waarmee verschillende API's met elkaar communiceren. Het is een bericht dat naar een server verstuurd kan worden om bepaalde informatie op te vragen of aan te bieden.
Endpoint	Een endpoint is een locatie in de API waar requests afgehandeld worden.
Query	Query is het Engelse woord voor aanvraag.
Software stack	Een complete verzameling van alle software die gebruikt wordt tijdens het bouwen en onderhouden van een applicatie.

Tabel 0-2: Begrippenlijst

[1] Inleiding

Moneybird biedt ondernemers een platform om bijhouden van de boekhouding gemakkelijk te maken. Dit platform met dezelfde naam kan gebruikt worden om facturen op te stellen, bonnetjes in te boeken en rapportages in te zien. Klanten kunnen Moneybird op verschillende manieren gebruiken. Zo biedt Moneybird een eigen interface aan, maar ook een API. Deze API heeft op dit moment twee use-cases: ontwikkelaars die Moneybird willen integreren in hun eigen administratie, en ontwikkelaars die een applicatie willen bouwen voor andere Moneybird gebruikers. Deze applicaties maken respectievelijk gebruik van een API token en OAuth 2.0 autorisatie.

REST

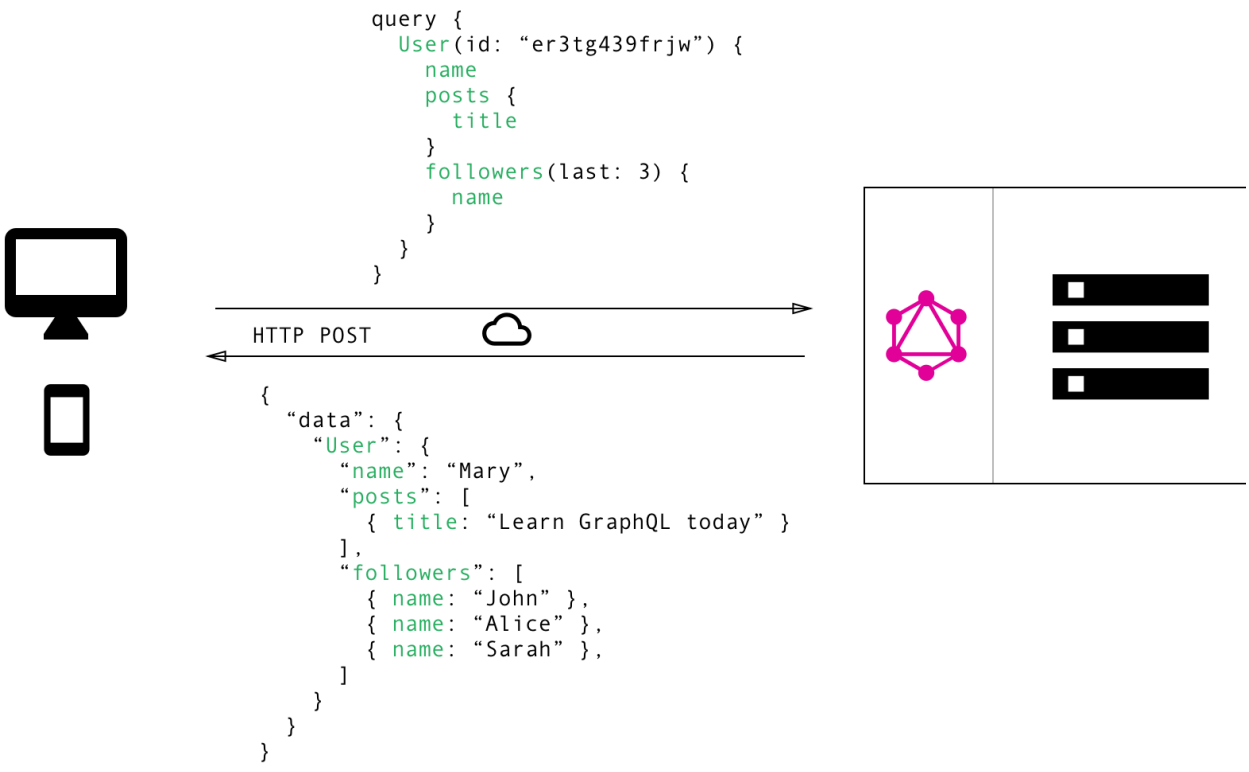
De huidige API van Moneybird maakt gebruik van REST. Hoewel dit goed werkt voor de meeste doeleinden, heeft het ook nadelen. Zo is de data die de API aanbiedt altijd in een vast formaat. Dit kan leiden tot uitwisseling van onnodige data, waar de performance van de API onder kan lijden.

Figuur 1-1: REST API calls (Burk, 2021)

Stel, een website met een blog wil data ophalen van een REST API. Een visualisatie van dit voorbeeld is te vinden in figuur 1-1. Voor de blog zijn drie stukken data nodig: de naam van de auteur, de titels van de posts van de auteur, en de drie meest recente volgers van de auteur. Om deze data op te vragen van een klassieke REST API zijn drie verschillende requests nodig. In een eerste request wordt de informatie over de auteur opgehaald. Vervolgens is een tweede request nodig om de data van de posts van de auteur op te halen. In een derde request worden de volgers van de auteur opgehaald. Elke van deze requests geeft meer data terug dan nodig is voor de blog. Zo bevat de informatie over de auteur ook een adres en een verjaardag, terwijl deze in dit geval niet nodig zijn.

GraphQL

GraphQL is een alternatief voor de klassieke REST API. Met GraphQL kan de gebruiker specificeren welke data nodig is van de server. Op deze manier is slechts één request nodig, en wordt geen onnodige data uitgewisseld. In figuur 1-2 is het voorbeeld met de blog nogmaals gevisualiseerd, maar dit keer met GraphQL in plaats van REST.



Figuur 1-2: GraphQL API call (Burk, 2021)

Combinatie

Het is ook mogelijk om zowel een REST API als een GraphQL API te ondersteunen. Uit dit onderzoek zal blijken of het voor Moneybird beter is om gebruik te maken van alleen REST, alleen GraphQL of een combinatie van beide.

[2] Context

In dit hoofdstuk is informatie te vinden over de organisatie waarbij de opdracht is uitgevoerd, en wat de aanleiding hiervoor was. In de eerste paragraaf is de organisatiebeschrijving te vinden. In de tweede paragraaf is de aanleiding voor het onderzoek beschreven.

2.1 Organisatiebeschrijving

De opdracht is aangeboden door Moneybird. Moneybird is in 2008 opgericht door Joost Diepenmaat, Edwin Vlieg, en Berend van Bruijnsvoort. Zij zochten een goede oplossing voor het maken van facturen, en besloten hier zelf iets op te bedenken. Het doel was om boekhouden niet alleen makkelijker te maken, maar ook leuk. Inmiddels is Moneybird een grote naam in boekhouding. Met hun platform “Moneybird” kunnen ondernemers nu gemakkelijk facturen maken, bonnetjes inboeken en rapportages inzien. Op deze manier biedt Moneybird veel inzicht in de financiën.

Moneybird heeft een kantoor aan de Moutlaan in Enschede. Hier werkt een team van meer dan 50 mensen. Een groot deel hiervan zijn softwareontwikkelaars. Alle collega's hebben echter hetzelfde doel: het waarmaken van ambities van ondernemers, door ze te ondersteunen bij het bouwen van een gezond bedrijf (Moneybird, z.d.).

Vanuit de organisatie zijn twee begeleiders aangewezen. Deze begeleiders zijn het eerste aanspreekpunt voor de student. Zij leveren informatie over het bedrijf, ondersteunen de student bij het maken van keuzes, en controleren de kwaliteit van de geschreven code en documenten. Er zijn twee begeleiders aanwezig, waardoor de afwezigheid van één van de begeleiders niet tot problemen leidt.

2.2 Aanleiding

Moneybird heeft een eigen interface om het boekhouden voor klanten makkelijk te maken. Om meer opties te bieden voor klanten heeft Moneybird ook een API. Deze REST API ondersteund twee use-cases:

- ontwikkelaars die hun administratie verder willen automatiseren. Deze ontwikkelaars kunnen door middel van een API token wijzigingen aanbrengen aan hun eigen boekhoudingen; en
- ontwikkelaars die een app willen voor klanten van Moneybird. Klanten kunnen deze applicatie door middel van OAuth 2.0 autorisatie toegang verlenen om wijzigingen aan te brengen in hun boekhoudingen.

Er is niets mis met de bestaande API van Moneybird. Toch is Moneybird altijd op zoek naar nieuwe technieken om hun applicatie te moderniseren. Één van deze moderne technieken is GraphQL.

GraphQL is een gestandaardiseerde query taal waarmee een client kan specificeren welke data verwacht wordt van een server. GraphQL is ontwikkeld om extra efficiëntie en flexibiliteit toe te voegen aan een API. Met GraphQL bepaalt de ontwikkelaar welke data uitgewisseld wordt. Omdat ontwikkelaars niet altijd alle data van een API nodig hebben, kan dit leiden tot een daling in de verstuurde hoeveelheid data. GraphQL is een populaire optie, waardoor er libraries beschikbaar zijn om de ontwikkeling van GraphQL applicaties eenvoudiger te maken, wat leidt tot minder werk voor de ontwikkelaars.

[3] Opdracht

Dit hoofdstuk bevat informatie over de inhoud van het onderzoek. In de eerste paragraaf wordt de probleemstelling beschreven. In de tweede paragraaf wordt de doelstelling beschreven. In de derde paragraaf wordt de afbakening van het onderzoek besproken. In de vierde paragraaf zijn de eindproducten te vinden. In de vijfde paragraaf zijn de onderzoeksvragen te vinden.

3.1 Probleemstelling

Om de applicatie te moderniseren wil Moneybird onderzoek doen naar GraphQL. GraphQL is een taal die gebruikt kan worden voor het structureren van API requests. Met GraphQL kan een gebruiker specificeren welke data zij op willen vragen van de API. Op deze manier is het voor de gebruiker mogelijk om enkel de benodigde data op te vragen. Dit kan voordelig zijn voor de performance van de API, omdat er geen ongebruikte data verstuurd wordt.

Voordat GraphQL geïmplementeerd kan worden door Moneybird, moet onderzocht worden of Moneybird hier baat bij heeft. Dat zal onderzocht worden tijdens dit afstudeertraject.

3.2 Doelstelling

Het doel van dit onderzoek is inzicht krijgen in het proces van het integreren van GraphQL in de Moneybird software stack en de gevolgen hiervan. Moneybird wil weten welke voor- en nadelen er zitten aan GraphQL ten opzichte van de bestaande REST API. Het moet duidelijk worden of er klanten zijn die baat hebben bij een GraphQL API, en welke klanten dit zijn.

Er wordt geen nieuwe functionaliteit toegevoegd aan de Moneybird applicatie, anders dan de ondersteuning van GraphQL. De GraphQL API endpoints moeten dezelfde functionaliteit aanbieden als de endpoints uit de REST API.

Dit onderzoek moet leiden tot de volgende vier meetbare doelen:

- Achterhalen of klanten baat hebben bij GraphQL, en welke klanten dit zijn;
- Achterhalen hoe GraphQL in de Moneybird stack geïntegreerd kan worden;
- Achterhalen wat de performance van de GraphQL API is ten opzichte van de bestaande REST API; en
- Achterhalen wat de mogelijkheden zijn van het genereren van API documentatie met GraphQL.

3.3 Afbakening

Tijdens het afstudeertraject worden verschillende onderdelen van GraphQL onderzocht. In deze paragraaf is te vinden welke onderdelen meegenomen worden in het onderzoek, en welke onderdelen niet.

Integratie in de bestaande stack
Er wordt onderzocht hoe GraphQL geïntegreerd kan worden in de bestaande software stack van Moneybird. Hierbij wordt gekeken wat de 'beste' methode is om dit te bereiken. Er wordt rekening gehouden met de tijd die nodig is voor het integreren, de complexiteit van het integreren, en de gevolgen die de integratie heeft voor de gebruikers van Moneybird.

Onderhoud van GraphQL
Wanneer GraphQL onderdeel van de Moneybird software stack is, moet hier rekening mee gehouden worden tijdens de ontwikkeling van nieuwe software. Het is belangrijk dat de integratie van GraphQL niet in de weg gaat zitten tijdens deze ontwikkelingen. Er wordt onderzocht of dit het geval is, en of hiervoor een oplossing is.

Performance voor- en nadelen
Ook wordt onderzocht wat het verschil is qua performance tussen GraphQL en de huidige REST API. Hierbij wordt specifiek gekeken naar de verschillen in performance met betrekking tot de functionaliteit van de huidige Moneybird API. Hierbij wordt voornamelijk gekeken naar de latency van de requests en de schaalbaarheid van GraphQL.

GraphQL API documentatie
GraphQL beschikt over de mogelijkheid om API documentatie te genereren. Dit kan veel tijd schelen bij het documenteren van de API. Tijdens het onderzoek wordt gekeken hoe dit onderdeel geconfigureerd kan worden en of de gegenereerde documentatie voldoet aan de standaard van Moneybird.

3.4 Eindproducten

Deze doelstelling leidt tot een aantal eindproducten. de volgende producten worden aan het eind van de afstudeerperiode opgeleverd:

- Een onderzoeksrapport waarin de resultaten van het onderzoek te vinden zijn;
- Een integratierapport waarin te vinden is hoe de integratie van GraphQL is verlopen;
- Een prototype van de implementatie van GraphQL in Moneybird;
- Een proof-of-concept voor het automatisch genereren van API documentatie met GraphQL.

3.5 Onderzoeksvragen

Op basis van de doelstelling is voor dit onderzoek de volgende onderzoeksvraag opgesteld:

“Op welke manier kan de implementatie van GraphQL de ervaring van zowel klanten als ontwikkelaars verbeteren?”

Om het antwoord op de onderzoeksvraag te vinden, zijn een aantal deelvragen opgesteld. De antwoorden op deze deelvragen zullen leiden tot het antwoord op de onderzoeksvraag. De deelvragen staat op de volgorde waarin zij behandeld zullen worden. Voor dit onderzoek zijn de volgende deelvragen opgesteld:

1. In welke opzichten is het gebruik van een GraphQL API anders dan het gebruik van een REST API?
2. Welke gebruikers zouden baat hebben bij een GraphQL API?
3. Hoe eenvoudig is het om een GraphQL API te implementeren in de bestaande Moneybird stack?
4. Hoe eenvoudig is het onderhouden van een GraphQL API?
5. Welke performance voor- en nadelen heeft het gebruik van GraphQL ten opzichte van de bestaande REST API?

Er is ook een zesde deelvraag opgesteld met betrekking tot het genereren van API documentatie. Wegens gebrek aan tijd is deze deelvraag uiteindelijk niet meegenomen in het onderzoek.

6. Welke mogelijkheden biedt GraphQL voor het genereren van API documentatie?

3.6 Requirements

Vanuit Moneybird zijn een aantal requirements opgesteld. Deze hebben met name betrekking op de opdracht.

- De oplossing moet werken met de bestaande software stack van Moneybird: het Ruby on Rails framework;
- De oplossing moet de huidige vormen van authenticatie ondersteunen;
- De oplossing moet dezelfde functionaliteit bieden als de huidige REST API.

[4] Proces

In dit hoofdstuk wordt het proces beschreven. In de eerste paragraaf zijn de gebruikte tools te vinden. In de tweede paragraaf wordt de projectorganisatie behandeld. In de derde paragraaf wordt de kwaliteitswaarborging behandeld. In de vierde paragraaf is de risicoanalyse te vinden.

4.1 Tools

Tijdens het onderzoek worden verschillende stukken hard- en software gebruikt. Hier worden al deze tools besproken. Het is mogelijk dat tijdens het onderzoek blijkt dat nog meer tools nodig zijn.

Laptop
Elke werknemer van Moneybird krijgt een Macbook in bruikleen. Deze wordt gebruikt tijdens het gehele afstudeertraject. De andere tools die gebruikt worden tijdens dit project moeten daarom compatibel zijn met de Macbook.

Slack
Voor alle communicatie binnen Moneybird wordt Slack gebruikt. In Slack zijn verschillende kanalen beschikbaar, waaronder een kanaal specifiek voor dit afstudeertraject. Dit kanaal wordt gebruikt voor communicatie tussen de student en de afstudeerbegeleiders vanuit Moneybird.

Google Suite
Moneybird maakt gebruik van de suite van Google. Deze suite wordt gebruikt voor het opslaan van documenten (Google drive), het beheren van e-mail, en als kalender.

Ruby
Binnen Moneybird wordt alle software geschreven in de taal Ruby. Hierbij wordt gebruik gemaakt van het Ruby on Rails framework. Om de geschreven code te kunnen implementeren in de software stack van Moneybird zal deze geschreven worden in Ruby.

Git
Git wordt gebruikt als versiebeheer voor de code van Moneybird. Tijdens dit project wordt gebruik gemaakt van Git, zodat de code gemakkelijk te gebruiken is in combinatie met de bestaande projecten van Moneybird.

Phabricator
Binnen Moneybird wordt Phabricator als projectmanagement systeem gebruikt. In Phabricator worden tickets aangemaakt en beheerd. Zo is het voor alle betrokkenen duidelijk waar aan gewerkt wordt. Daarnaast wordt Phabricator ook gebruikt voor het beheren van wijzigingen in de code.

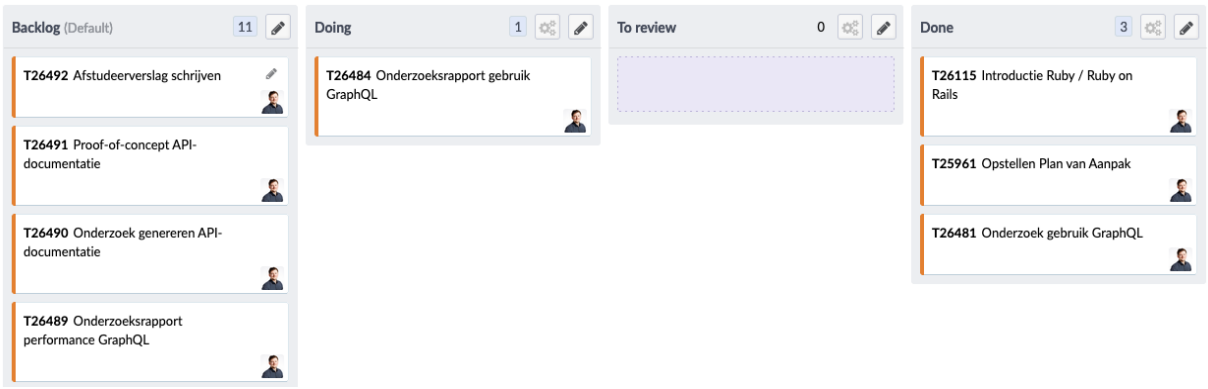
Arcanist
Arcanist is een command line tool voor Phabricator. Wanneer een ticket is afgerond, wordt Arcanist gebruikt om een zogenaamde ‘diff’ aan te maken in Phabricator. In deze diff is makkelijk te zien welke wijzigingen aangebracht zijn. Arcanist kan ook gebruikt worden om de code op te halen uit Phabricator tijdens een review. Wanneer een diff goedgekeurd is, wordt de code gemerged naar de development branch.

Buildkite
Wanneer code gepushed wordt, gaan er automatisch taken runnen in Buildkite. Hierbij wordt de linter uitgevoerd om de kwaliteit van de code te verbeteren en worden alle tests uitgevoerd. Wanneer een van deze taken niet succesvol afgerond kan worden, mag de code niet naar de development branch gemerged worden.

Docker
Moneybird maakt veel gebruik van Docker. Docker wordt gebruikt om de code op een virtueel besturingssysteem uit te voeren. Omdat de omgeving waarin de software draait altijd hetzelfde is, zijn er vrijwel geen problemen waarbij de software wel werkt op computer A maar niet op computer B.

4.2 Projectorganisatie

Om het project te managen wordt gebruik gemaakt van de Kanban methode. Omdat dit project niet in teamverband wordt uitgevoerd, is het gebruik van een methode als Scrum niet nodig. Om het project toch overzichtelijk te houden, is gekozen voor de kanban methode. Voor elke stap in het proces wordt ook een ticket aangemaakt op het projectbord op Phabricator. Elk ticket bevat minstens een duidelijke omschrijving van het doel, en een definition-of-done. Eventuele documentatie is ook onderdeel van het ticket, en wordt als voorwaarde meegenomen in de definition-of-done.



Figuur 4-1: Kanban bord in Phabricator

Ter bevordering van het projectmanagement wordt ook iedere week de voortgang opgenomen in de projectdocumentatie. Hierin wordt voor de desbetreffende week aangegeven welke taken gepland waren, welke taken afgerond zijn, en tegen welke obstakels de student is aangelopen.

4.3 Kwaliteitsbewaking

Om de kwaliteit van alle aspecten van het afstudeertraject te waarborgen, zijn een aantal kwaliteitscontroles opgesteld. Deze controles duiden de standaarden aan waar aan voldaan moet worden.

4.3.1 Proces

Tijdens het afstudeerproces staan er wekelijkse sessies ingepland met de begeleiders vanuit Moneybird. Tijdens deze sessies wordt de voortgang besproken, en is er mogelijkheid tot feedback. Daarnaast voert de student eens in de vier weken een voortgangsgesprek met de docent. Ook hier wordt de voortgang besproken, en is er mogelijkheid tot feedback. Omdat deze momenten regelmatig plaatsvinden wordt het snel duidelijk als het proces niet voldoet aan de normen van Saxion of Moneybird.

Documenten
Tijdens de wekelijkse sessies met de begeleiders, en de sessies met de docent worden ook de wijzigingen aan de documenten besproken. Op deze manier kunnen alle documenten tijdig bewerkt worden indien nodig.

Naast de feedback van de begeleiders en de docent, zorgt de student dat alle documenten voldoen aan de richtlijnen van Saxion, zoals gespecificeerd in het document “Rapportscan HBO-ICT”, te vinden op Google Drive.

4.3.2 Code

Tijdens het afstudeertraject wordt een prototype gebouwd door de student. Om de kwaliteit van de code te kunnen waarborgen, worden de methodes die ook binnen Moneybird gebruikt worden aangehouden. Naast het prototype wordt mogelijk ook code geschreven die alleen tijdens het onderzoek gebruikt wordt. Deze code hoeft niet te voldoen aan dezelfde standaarden als de code die mogelijk in productie komt bij Moneybird, en wordt niet onderworpen aan deze controles.

Code reviews
Alle geschreven code wordt door minimaal één van de begeleiders bekeken en goedgekeurd. Zolang dit niet het geval is, wordt de code niet als ‘af’ beschouwd. In het geval dat de geschreven code afgekeurd wordt, wordt deze op basis van de feedback van de begeleiders verbeterd.

Automated tests

Voor alle functionele code worden ook tests geschreven. Wanneer de code naar de GitHub repository gepushed wordt, worden deze tests automatisch uitgevoerd. De code wordt niet als ‘af’ beschouwd indien:

- er geen tests geschreven zijn voor de code;
- de geschreven tests niet slagen;
- bestaande tests niet langer slagen door de toegevoegde code.

4.4 Risicoanalyse

Risico	Kans	Impact	Oorzaak	Voorkomen	Oplossen
Een van de onderzoeksfases kost meert tijd dan verwacht	middel	hoog	Nodige tijd van de werkzaam-heden onderschat.	Ruim voldoende tijd inschatten voor de werkzaamheden.	Overleggen met de docent en bedrijfsbegeleiders welke onderzoeken de hoogste prioriteit hebben, en deze eerst afronden.
uitval van bedrijfs-begeleider.	laag	middel	Wegens omstandig-heden kan een bedrijfs-begeleider uitvallen.	Vanuit Moneybird zijn twee begeleiders aangewezen.	De tweede begeleider neemt de begeleiding over.
De complexiteit van de opdracht blijkt te hoog.	laag	hoog	Gebrek aan kennis bij de student.	Door tijdig hulp te vragen bij de begeleiders kan voorkomen worden dat de voortgang stil komt te staan.	Overleggen met de docent en bedrijfsbegeleiders of de opdracht aangepast kan worden om de complexiteit te verminderen.

Tabel 4-1: Risicoanalyse

[5] Theoretisch Kader

In dit hoofdstuk wordt informatie verzameld over het onderzoek. In de eerste paragraaf wordt besproken welke informatie reeds beschikbaar is over het onderwerp. In de tweede paragraaf is een onderzoek naar REST te vinden. In de derde paragraaf wordt de basis van GraphQL besproken.

5.1 Literatuurverkenning

Voor aanvang van het onderzoek is een literatuurverkenning gedaan naar informatie die bekend is over het onderwerp. Bij de literatuurverkenning zijn vier onderwerpen meegenomen.

Gebruik GraphQL
Er is veel informatie beschikbaar over de verschillen tussen GraphQL en REST. Wiesbauer (2019) schreef bijvoorbeeld een whitepaper over de voordelen van GraphQL ten opzichte van REST. Daarin beschrijft hij hoe GraphQL de hoeveelheid data die verzonden wordt kan verminderen.

Integratie GraphQL
Over de implementatie van GraphQL in Ruby on Rails is ook veel informatie beschikbaar. De GraphQL Ruby (n.d.) website bevat veel informatie over de installatie en het gebruik van de Ruby on Rails GraphQL gem. Daarnaast is de community voor GraphQL voor Ruby on Rails erg groot. Hierdoor is veel informatie beschikbaar over het onderwerp op websites als GitHub (https://www.github.com) of StackOverflow (https://www.stackoverflow.com).

GraphQL performance
Naast informatie over het gebruik en de installatie van GraphQL zijn er ook meerdere artikelen over de verschillen in performance tussen een GraphQL API en een REST API. Een voorbeeld hiervan is het artikel van Lawi et al. (2021). Hierin wordt beschreven hoe een grote hoeveelheid tests gebruikt zijn om de verschillen in performance tussen de twee technieken te meten. Uit dit onderzoek blijkt dat de klassieke REST API een lagere latency en hogere throughput heeft dan een GraphQL API, terwijl de GraphQL API minder CPU en geheugen vraagt van de server. Of dit ook het geval is binnen de software van Moneybird zal blijken uit dit onderzoek.

API documentatie
Ook voor API documentatie zijn meerdere oplossingen beschikbaar. Er zijn twee vormen die veel voorkomen. De eerste vorm is een implementatie voor de documentatie in dezelfde applicatie als de API zelf, zoals Graphdoc-ruby (https://github.com/alpaca-tc/graphdoc-ruby). De tweede optie is een externe applicatie die de API uitleest om documentatie te genereren, zoals DociQL (https://github.com/wayfair/dociql) of SpectaQL (https://github.com/anvilco/spectaql).

5.2 GraphQL

Tijdens het afstudeertraject wordt onderzocht waar de verschillen tussen REST en GraphQL liggen. Het is hiervoor nodig om kennis van beide onderwerpen te hebben. Tijdens de literatuurverkenning is dan ook onderzoek gedaan naar beide technieken. In het onderzoeksrapport in bijlage A is meer informatie te vinden over de literatuurstudies. Hierin wordt ook dieper ingegaan op de implementatie van de technieken met het HTTP protocol. In dit verslag is alleen het onderzoek naar GraphQL meegenomen.

5.2.1 GraphQL

GraphQL is een query taal ontwikkeld door Facebook in 2012. Net als REST, wordt GraphQL gebruikt bij communicatie volgens een server-client model. In 2015 is de ontwikkeling van de open standaard van GraphQL begonnen. In 2017 is de eerste specificatie goedgekeurd. GraphQL biedt een intuïtieve en flexibele syntaxis voor het beschrijven van data-uitwisseling en interacties (GraphQL contributors, 2021).

GraphQL maakt gebruik van een schema om een API te veranderen in een database. Op deze manier kan een client zelf aangeven welke data van de server verwacht wordt. Een GraphQL schema bestaat uit verschillende velden. Ieder veld heeft een naam en een type, zoals int, float of string. Ook kan een veld van het type object zijn. Een object bevat zelf weer meerdere velden. Op deze manier komt het schema er uit te zien als een grafiek, vandaar de Engelse naam GraphQL. Het is ook mogelijk om zelf nieuwe types toe te voegen.

Schema
GraphQL heeft ook een eigen taal, genaamd SDL (Schema Definition Language). In figuur 5-1 is een voorbeeld te zien van deze taal. In dit voorbeeld worden twee types beschreven die gebruikt kunnen worden om een blog te bouwen: post en auteur. Een post heeft velden voor ID, auteur, titel en body. Een auteur heeft velden voor ID, naam en e-mail. Van ieder veld wordt eerst de naam genoemd, gevolgd door het type. De meeste velden hebben string als type, maar het auteur-veld op de post heeft het type auteur. Dit is een verwijzing naar het auteur-type, dat er onder beschreven wordt. Tot slot bevat het voorbeeld ook een query type. Dit type wordt gezien als de root van de API. Het query type bevat alleen een post-veld, waarmee een object van type post opgehaald kan worden.

```
type Post {
  id: String!
  author: Author
  title: String
  body: String
}

type Author {
  id: String!
  name: String
  email: String
}

type Query {
  post(id: String!): Post
}
```

Figuur 5-1: GraphQL blog schema (Brito et al., 2019)

Query

Een client kan nu een query gebruiken om data uit het schema te halen. In figuur 5-2 is een voorbeeld te zien van een query genaamd Post By Title And Author. In deze query wordt de post opgehaald op basis van het ID, in dit geval 1000. De query geeft aan dat de velden titel en auteur verwacht worden. Omdat het type auteur eigenlijk een object is, moet de query ook aangeven welke velden van auteur verwacht worden. In dit geval is dat alleen de naam.

```
query PostByTitleAndAuthor{
    post(id:"1000"){
      title
      author {
        name
      }
    }
}
```

Figuur 5-2: GraphQL post query (Brito et al., 2019)

Response

De server zal op de query reageren met de opgevraagde data. In dit voorbeeld wordt uitgegaan van een JSON formaat voor de data. In figuur 5-3 is een mogelijke reactie van de server te zien op de query uit figuur 5-2.

```
{ "data": {
    "post":{
       "title": "GraphQL: A data  query language"
       "author":{
          "name": "Lee Byron"
       }
    }
  }
}
```

Figuur 5-3: GraphQL post response (Brito et al., 2019)

Mutation

Bij het opvragen van data wordt gebruik gemaakt van een query. Vaak is het ook nodig om data op de server aan te passen. Bij GraphQL gebeurt dit door middel van een mutation. In figuur 5-4 is een voorbeeld te zien van een mutation type met de naam add Post. Deze mutation verwacht een post object als argument, en bevat ook een veld om een post op te halen.

```
type Mutation {
   addPost(post: Post): Post
}
```

Figuur 5-4: GraphQL post mutation (Brito et al., 2019)

[6] Methodologie

In dit hoofdstuk wordt besproken welke methoden toegepast zijn tijdens het onderzoek. In de eerste paragraaf is informatie te vinden over de indeling van de onderzoeksvragen. In de tweede paragraaf is te vinden over de ICT research methoden. In de derde paragraaf is te vinden welke methoden toegepast zijn per deelvraag.

6.1 Indeling onderzoek

Dit onderzoek raakt verschillende vlakken van GraphQL. Om het onderzoek overzichtelijk te houden zijn de deelvragen ingedeeld in vier verschillende categorieën. De eerste categorie bevat deelvragen die gaan over GraphQL vanuit het perspectief van de gebruikers van Moneybird. De tweede categorie bevat deelvragen die gaan over GraphQL vanuit het perspectief van de ontwikkelaars van Moneybird. De vragen in de derde categorie gaan over de performance van GraphQL ten opzichte van de huidige REST API. De vragen in de laatste categorie gaan over het automatisch genereren van API documentatie.

GraphQL voor de klanten van Moneybird
De Moneybird API wordt gebruikt door klanten. De complexiteit van de API mag hier niet te hoog worden, zodat klanten nog steeds gebruik kunnen maken van de API. De complexiteit ligt hier met name bij de extra hoeveelheid werk die een klant in de software moet stoppen om de juiste data op te halen van de GraphQL API. De API is gericht op ontwikkelaars, waardoor enige technische kennis verwacht mag worden. Naast de complexiteit moet ook rekening gehouden worden met backwards compatibility. Deze categorie bevat de volgende deelvragen:

1. In welke opzichten is het gebruik van een GraphQL API anders dan het gebruik van een REST API?

2. Welke gebruikers zouden baat hebben bij een GraphQL API?

GraphQL voor de Moneybird ontwikkelaars
Het is van belang dat de integratie van GraphQL niet te veel complexiteit toevoegt aan de Moneybird software stack. Extra complexiteit zou ervoor zorgen dat ontwikkeling langer duurt, en kan zorgen voor de introductie van bugs. Complexiteit voor de ontwikkelaars ligt met name in de hoeveelheid extra werk die het onderhouden van een GraphQL API met zich meebrengt. Als voor de GraphQL API bijvoorbeeld alle logica opnieuw geschreven moet worden, is de complexiteit te hoog.

De deelvragen in deze categorie bieden inzicht in de gevolgen voor de Moneybird ontwikkelaars in het geval van integratie met GraphQL. Deze categorie bevat de volgende deelvragen:

3. Hoe eenvoudig is het om een GraphQL API te integreren in de bestaande Moneybird stack?

4. Hoe eenvoudig is het onderhouden van een GraphQL API?

Performance van GraphQL
Performance is een belangrijk aspect van de integratie van GraphQL. Als de snelheid en schaalbaarheid van GraphQL niet vergelijkbaar of beter zijn dan die van de REST API, is de integratie van GraphQL het mogelijk niet waard. De deelvragen in deze categorie zullen inzicht bieden in de performance van GraphQL. Deze categorie bevat de volgende deelvragen:

5. Welke performance voor- en nadelen heeft het gebruik van GraphQL ten opzichte van de bestaande REST API?

API documentatie met GraphQL
GraphQL biedt ook de mogelijkheid om automatisch API documentatie te genereren. Dit kan interessant zijn voor zowel de ontwikkelaars als de klanten van Moneybird. Het kan ervoor zorgen dat ontwikkelaars minder tijd nodig hebben voor het opstellen van API documentatie. De klanten kunnen er op deze manier zeker van zijn dat de documentatie correct en up-to-date is. Deze categorie bevat de volgende deelvragen:

6. Welke mogelijkheden biedt GraphQL voor API documentatie?

6.2 ICT Research Methoden

Om methodisch te werk te gaan tijdens het onderzoek, is het DOT framework toegepast (https://ictresearchmethods.nl). Dit framework biedt verschillende onderzoeksmethoden aan, gebaseerd op het doel van de gebruiker. Tijdens het uitvoeren van het onderzoek zijn deze methoden voornamelijk als richtlijnen gebruikt. Hier is voor gekozen, omdat het DOT framework grotendeels bedoeld is voor de ontwikkeling van een product, terwijl tijdens dit project onderzoek gedaan wordt naar een bestaand product.

6.3 Onderzoeksmethoden per deelvraag

De gebruikte onderzoeksmethoden verschillen per deelvraag. Hieronder is te vinden welke methoden zijn toegepast per deelvraag.

In welke opzichten is het gebruik van een GraphQL API anders dan het gebruik van een REST API?
Aan het begin van het onderzoek is een literatuurstudie gedaan om meer informatie te vinden over de werking van GraphQL voor de gebruikers van Moneybird, evenals de verschillen tussen GraphQL en REST. Bij de literatuurstudie wordt informatie verkregen uit bestaande bronnen. Deze bronnen worden bestudeerd en samengevat in het onderzoeksverslag. De literatuurstudie helpt met het vinden van betrouwbare informatie over het onderwerp. Deze methode is gekozen

omdat aan het begin van het onderzoek nog niet veel kennis aanwezig is over het onderwerp. De resultaten van deze literatuurstudie zijn te vinden in hoofdstuk 5.

Naast de literatuurstudie is ook gekozen voor de prototypingmethode om ervaring op te doen met GraphQL. Voor deze methode wordt een kleine API opgezet met ondersteuning voor REST en GraphQL. Naar deze API worden een aantal REST requests en een aantal GraphQL requests voor dezelfde acties gestuurd. Hierdoor wordt duidelijk waar de verschillen tussen REST en GraphQL liggen. Met de prototypingmethode wordt snel duidelijk hoe de technieken in de praktijk werken. Omdat beide API's met dezelfde data werken, zijn de technieken goed te vergelijken.

Welke gebruikers zouden baat hebben bij een GraphQL API?
De informatie die verkregen is tijdens de eerste deelvraag wordt ook meegenomen tijdens de fieldstrategie. Hiervoor zijn de observatiemethode en taak-analysemethode gekozen.

Het doel van de observatiemethode is het verkrijgen van informatie over de manier waarop de klanten de huidige API gebruiken. Tijdens deze methode worden nog geen conclusies getrokken. Het gaat enkel om het verzamelen en structureren van de gegevens. Voor dit onderzoek is gekozen om de logs van de Moneybird API te analyseren. Op deze manier kan achterhaald worden waar de klanten de API voor gebruiken, zonder dat contact met de klanten nodig is.

Nadat de gegevens gestructureerd zijn, worden deze onderzocht met de taak-analysemethode. Tijdens deze methode wordt onderzocht waarom klanten specifieke API requests maken, en tegen welke limieten zij hierbij aanlopen. Deze methode biedt inzicht in hoe de huidige API gebruikt wordt, en op welke gebieden GraphQL de klanten kan helpen.

Hoe eenvoudig is het om een GraphQL API te integreren in de bestaande Moneybird stack?
Om te achterhalen hoe eenvoudig het is om GraphQL te integreren in Moneybird, worden verschillende methoden toegepast. Allereerst is de literatuurstudie methode gekozen om te leren welke opties er zijn voor het implementeren van GraphQL in Ruby on Rails. Op basis van de resultaten van deze methode wordt gekozen op welke manier GraphQL geïmplementeerd wordt in Moneybird.

Wanneer bekend is op welke manier GraphQL geïmplementeerd kan worden, wordt de implementatie uitgevoerd. Hierbij is opnieuw gekozen voor de prototypingmethode. Om zeker te zijn dat alle delen van de implementatie werken zijn ook een aantal showroom methoden toegepast. Moneybird maakt gebruik van continuous integration. Geschreven code wordt bijvoorbeeld door een linter gehaald om te zorgen dat aan de standaarden van Moneybird voldoet. Daarnaast worden code reviews uitgevoerd door de begeleiders. Ook hier wordt gekeken of de code voldoet aan de standaard van Moneybird.

Tot slot worden tijdens de ontwikkeling ook unit tests en system tests geschreven. Op deze manier wordt de kwaliteit van de code verder gewaarborgd.

Hoe eenvoudig is het onderhouden van een GraphQL API?
Om erachter te komen hoe eenvoudig het is om een GraphQL te onderhouden wordt onderzocht wat er nodig is om nieuwe functionaliteit toe te voegen aan een GraphQL API. Hiervoor worden de workshop en lab strategieën toegepast.
Ook hier is gekozen voor de prototypingmethode. Er wordt functionaliteit toegevoegd aan de GraphQL implementatie, om te leren welke stappen hiervoor nodig zijn. Uiteraard worden tijdens de ontwikkeling weer unit tests en system tests geschreven om te controleren of de toegevoegde functionaliteit aan alle eisen voldoet. De ervaring van het toevoegen van functionaliteit aan de GraphQL implementatie biedt voldoende inzicht om de deelvraag te beantwoorden.

Welke performance voor-en nadelen heeft het gebruik van GraphQL ten opzichte van de bestaande REST API?
Een belangrijk aspect van het onderzoek naar GraphQL is het verschil in performance ten opzichte van REST. Om dit verschil te onderzoeken is de benchmark test methode toegepast. Bij deze methode worden gestandaardiseerde tests uitgevoerd op zowel de GraphQL API als de REST API. Op deze manier kunnen de resultaten direct met elkaar vergeleken worden.

[7] Onderzoeksresultaten

In dit hoofdstuk worden de resultaten van het onderzoek besproken. De zes paragrafen bevatten elk de resultaten van één deelvraag.

7.1 Verschil in gebruik

Tijdens de literatuurstudie zijn de REST en GraphQL concepten besproken. In dit document worden de HTTP implementaties van REST en GraphQL met elkaar vergeleken, omdat deze implementaties relevant zijn voor Moneybird. In het onderzoeksrapport zijn de volledige onderzoeken naar zowel REST als GraphQL te vinden met meer voorbeelden. In dit document worden voorbeelden gebruikt uit het gebouwde prototype. Dit prototype draait op de lokale machine op poort 9090.

7.1.1 Prototype

Tijdens het onderzoeken van de verschillen tussen REST en GraphQL is een prototype opgesteld. Dit prototype maakt het mogelijk om de praktische verschillen tussen REST en GraphQL te vinden. Het prototype beschikt over zowel een REST API als een GraphQL API. De API's werken met dezelfde data. Op deze manier zijn er geen verschillen tussen de omgevingen waarin de API's getest worden.

Het prototype is een simpele blog. Het is mogelijk om artikelen op te halen of te plaatsen. Hiervoor zijn twee modellen gebruikt. Het eerste model is de gebruiker. Een gebruiker heeft drie eigenschappen: ID, naam en e-mail. Het tweede model is een artikel. Een artikel heeft vier eigenschappen: ID, titel, inhoud en auteur. Het auteur veld bevat de ID van de gebruiker die het artikel heeft aangemaakt. Zowel artikelen als gebruikers kunnen aangemaakt worden met de REST API en de GraphQL API.

7.1.2 Vergelijking REST en GraphQL

REST en GraphQL zijn twee totaal verschillende technieken die uiteindelijk hetzelfde doel bereiken. REST werkt door representaties van resources heen en weer te sturen. Met HTTP methoden wordt aangegeven welke operatie de server uit moet voeren op de aangeroepen resource. De server stuurt met behulp van een status code, headers en een body het resultaat van de operatie terug. GraphQL laat gebruikers zelf aangeven welke gegevens zij op willen halen, en welke operaties zij uit willen voeren. GraphQL heeft slechts één endpoint per schema. Dit schema definieert alle mogelijke queries en operaties die een gebruiker kan uitvoeren. Een GraphQL server reageert altijd met status code 200, minimale headers en een body.

URI
Een REST API maakt gebruik van verschillende URI's. Elke URI wijst naar een specifieke resource (Masse, 2011). De resources zijn gestructureerd volgens een hiërarchie. Het path van het URI geeft aan op welke resource de operatie uitgevoerd moet worden (Fielding, 2000). De verschillende segmenten van het path worden gescheiden met een slash. Het path voor het ophalen van reviews van een product met ID vijftien kan er als volgt uitzien: "/products/15/reviews". Verdere gegevens die nodig zijn om resources te identificeren, zoals gegevens om resources te filteren, worden meegegeven als query parameters. De eerste van deze parameters wordt aangeduid met een vraagteken. De volgende parameters beginnen allemaal met een ampersand.

Een GraphQL API maakt veel minder gebruik van URI's. Een GraphQL schema draait op een enkel URI. Een API kan wel meerdere schema's hebben. Deze draaien dan wel op verschillende endpoints.

HTTP methoden
De verschillende HTTP methoden zijn een groot onderdeel van REST over HTTP. De methoden geven aan welke operatie een server moet uitvoeren. Veel voorkomende methoden zijn GET, voor het ophalen van resources; PUT, voor het aanmaken of wijzigen van een bestaande resource; DELETE, voor het verwijderen van een resource en POST, voor het aanmaken van een resource (MDN Contributors, 2023).

GraphQL hoeft officieel alleen de POST methode te ondersteunen. Het is ook toegestaan om andere methoden te ondersteunen, zoals GET. In dit geval worden de request parameters meegegeven in het query-deel van het URI. Deze methoden kunnen ook alleen query's uitvoeren, geen mutations of subscriptions. Bij het sturen van een POST methode worden alle parameters meegegeven in de body van het request. Het is dan ook mogelijk om mutations of subscriptions aan te roepen.

Status codes
Een REST response geeft het resultaat van een request aan door middel van een status code. Een 2xx code geeft aan dat het request succesvol is uitgevoerd. Een 4xx of 5xx geeft aan dat er een fout is opgetreden. Voor veel situaties zijn specifieke status codes, zoals een 404 (Not Found) wanneer een resource niet bestaat (MDN Contributors, 2023b).

Een GraphQL API gaat anders om met de status codes. De manier waarop is afhankelijk van het gebruikte mediatype. Tijdens dit onderzoek is gekeken naar het application/json media type. Bij dit type wordt altijd status code 200 gebruikt, tenzij het request niet valide is. In dat geval wordt status code 400 teruggestuurd.

Headers
HTTP headers bevatten metadata van een request of response (Red Hat, 2020). Bij een RESTful API gelden speciale regels voor de headers. Er zijn een aantal headers die altijd meegestuurd

moeten worden in een API, zoals Content-Type en Content-Length. Dit geldt natuurlijk alleen als het bericht ook een body bevat.

De specificatie van GraphQL over HTTP noemt alleen de Accept- en Content-type headers. Deze headers worden gebruikt om het mediatype aan te geven. Het is ook toegestaan om andere headers mee te sturen, bijvoorbeeld voor authenticatie.

Body
Bij een REST API bevat de body vaak representaties van resources. Deze representaties zijn de basis van communicatie binnen REST. De data in de bodies is vaak gestructureerd volgens het JSON of XML formaat, maar ook andere formaten zijn toegestaan.

Bij GraphQL speelt de body een grotere rol. Alle informatie over een request staat in de body, van de operatie tot uitgewisselde data. Elke GraphQL API moet JSON accepteren en genereren. Andere formaten zijn optioneel.

Foutafhandeling
Bij het optreden van een fout zal een REST API reageren met een passende statuscode. Voor veel fouten is een specifieke foutcode aanwezig. Vaak wordt extra informatie over de fout meegestuurd in de body van de response.

Bij GraphQL staat alle informatie over de fout in de body. De status code is altijd 200. Bij het optreden van een fout bevat de body van de response een lijst met fouten, genaamd errors. Hierin is alle informatie over de opgetreden fouten te vinden.

Voorbeelden
Het eerste voorbeeld gaat over het ophalen van een blogartikel met de naam van de gebruiker. De ID van het artikel is bekend. Bij een REST API zijn hiervoor twee verschillende requests nodig. Het eerste request is te zien in figuur 7-1. Dit request haalt het artikel op met de ID. Het tweede request, te zien in figuur 7-2, neemt de author ID uit het eerste request om de informatie van de auteur op te halen. De headers zijn weggelaten uit de voorbeelden om ruimte te besparen.

```
Request
GET http://127.0.0.1:9090/articles/2

Response
HTTP/1.1 200 OK

{
  “id”:2,
  “title”: “What is GraphQL?”,
  “author”: 1,
  “body”: “GraphQL is a query language that allows users to
           dynamically describe data requirements and interactions.”
}
```

Figuur 7-1: REST request artikel

```
Request
GET http://127.0.0.1:9090/users/1

Response
HTTP/1.1 200 OK

{
  "id": 1,
  "name": "Wouter",
  "email": "wouter@moneybird.nl"
}
```

Figuur 7-2: REST request auteur

Bij de GraphQL API is het makkelijker om alle nodige informatie op te halen. In figuur 7-3 is te zien hoe een enkel GraphQL request gebruikt wordt, om niet alleen informatie over de post, maar ook informatie over de auteur op te halen.

```
Request
POST http://127.0.0.1:9090/graphql

{
  "query": "query { article(id: 2) { title body author { name } } }"
}

Response
HTTP/1.1 200 OK

{
  "data": {
    "article": {
      "title": "What is GraphQL?",
      "body": "GraphQL is a query language that allows users to
               dynamically describe data requirements and
               interactions.",
      "author": {
        "name": "Wouter"
      }
    }
  }
}
```

Figuur 7-3: GraphQL request artikel met auteur

De GraphQL API kan alle nodige informatie in één keer aanleveren. Daarnaast geeft de API ook geen onnodige velden terug, omdat in het request aangegeven is welke velden nodig zijn. De REST API geeft daarentegen ook het e-mailadres van de auteur terug.

7.2 Welke gebruikers hebben baat bij GraphQL?

Voor deze deelvraag wordt onderzocht hoe de huidige API van Moneybird gebruikt wordt en hoe GraphQL de gebruikerservaring van de API kan verbeteren.

7.2.1 API logs

Om te onderzoeken hoe de gebruikers de huidige API gebruiken, zijn de logs van de Moneybird API van de periode tussen 20 februari 2023 en 5 april 2023 (45 dagen) geëxporteerd naar een csv bestand. Over deze periode zijn ongeveer 40 miljoen requests door de Moneybird API ontvangen. Wegens de grootte is het niet mogelijk dit bestand te openen met software als Excel. Om het bestand in te kunnen zien, kan deze geopend worden met de commandline tool Less. Less laadt het bestand niet in het geheel in het geheugen, wat de tool geschikt maakt voor het bekijken van grote bestanden.

Het csv-bestand bestaat uit zeven kolommen:

- timestamp, de datum en tijd waarop het request gemaakt is;
- method, de HTTP methode waarmee het request is aangeroepen;
- path, het path component van het URI;
- duration, het aantal milliseconden dat het kostte om het request af te handelen;
- params, de parameters die zijn meegestuurd met het request - zowel in het URI als de body;
- status, de HTTP status code die teruggestuurd is naar de client; en
- user_agent, de naam van de software waarmee het request verstuurd is.

7.2.2 PostgresQL

Een csv-bestand is handig voor het opslaan van gegevens, maar niet om deze gegevens te analyseren. Om het analyseren makkelijker te maken, is het csv-bestand geïmporteerd in een PostgresQL database. Hiervoor is een script geschreven in de taal Python. Tijdens het importeren van de gegevens zijn twee extra kolommen toegevoegd.

General path
De eerste van deze kolommen is general_path. De originele data bevat een path-kolom, waarin het path staat dat gebruikt is voor het request. Wanneer een gebruiker een request verstuurt, is dit vaak naar een specifieke resource. Het kan bijvoorbeeld zijn dat gebruikers alle contacten op willen vragen. Deze gebruikers zijn gekoppeld aan hun administratie. Zij geven dus de ID van hun administratie mee om de contacten op te halen. Dit is in principe hetzelfde request, alleen is het path net iets anders. De general_path kolom bevat een versie van het path waarbij alle ID’s zijn vervangen met “<id>”. Op deze manier kan gefilterd worden op vergelijkbare requests naar verschillende resources.

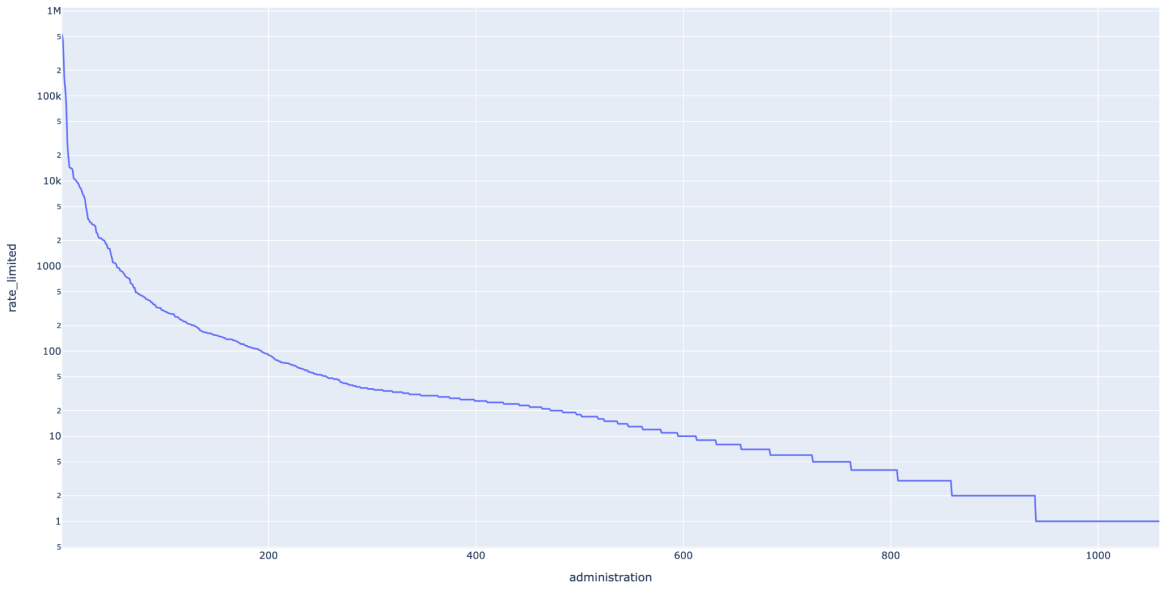
Administration

De tweede kolom die is toegevoegd is administration. Om anonimiteit te garanderen zijn geen gebruikers opgenomen in de log bestanden. Toch moet voor het onderzoek wel duidelijk zijn welke requests door dezelfde gebruikers verstuurd zijn. Een groot deel van de requests bevat een path waarin de ID van de administratie is opgenomen. Bij elk request waarbij het request een administratie ID bevat, is deze ID opgeslagen in een kolom genaamd administration. Bij requests die geen administration ID bevatten is de waarde van dit veld op null gezet. Bij dit onderzoek is de aanname gedaan dat requests naar dezelfde administratie van dezelfde gebruiker komen.

7.2.3 Analyse

Om te achterhalen hoe de Moneybird API gebruikt wordt, zijn een aantal vragen opgesteld. Om deze vragen te beantwoorden zijn queries uitgevoerd op de database met de log-gegevens van de Moneybird API over de periode van 45 dagen. Deze queries zijn niet opgenomen in dit verslag, maar zijn te vinden in het onderzoeksverslag.

Bij hoeveel administraties is er sprake van een rate-limit?
Er is een significant aantal gebruikers dat regelmatig tegen een rate-limit aanloopt. In figuur 7-4 is te zien dat het grootste deel van de rate-limits voorkomt bij een klein deel van de administraties. Een groot deel van deze rate-limits wordt veroorzaakt door het opvragen van meerdere resources van hetzelfde type, zoals het opvragen van meerdere contacten. Met GraphQL kan een query opgesteld worden om details uit een lijst van contacten op te vragen. Bij de huidige REST API worden hier alleen genoeg gegevens meegestuurd om de details op te halen met nieuwe requests. GraphQL zou voor een aantal klanten een oplossing zijn voor de rate-limit. Het is natuurlijk wel belangrijk om GraphQL zo te configureren dat klanten niet te veel van de server kunnen vragen.



Figuur 7-4: Grafiek rate-limits per administratie

Welke requests worden vaak achter elkaar gestuurd?
Er zijn ook een aantal requests die vaak achter elkaar verstuurd worden. Zo komt het veel voor dat gebruikers eerst een lijst van facturen opvragen, en vervolgens een van deze facturen versturen. Ook halen klanten vaak meerdere facturen achter elkaar op. Ook hier kan GraphQL een oplossing bieden, in de vorm van een query die meerdere facturen tegelijk op kan halen.

Hoe vaak wordt elk path aangeroepen?
Er zijn een aantal endpoints die vaker aangeroepen worden dan de rest. Deze endpoints worden als eerst geïmplementeerd in het prototype. Het gaat om de endpoints voor het beheren van facturen, contacten en administraties.

Welke requests duren het langst?
Uit het onderzoek naar de duur van de requests blijkt dat een aantal requests veel meer tijd nodig heeft dan de rest. Het gaat hier vooral om het ophalen en versturen van bijlagen. Het is mogelijk dat een combinatie van mutations en subscriptions hier kan helpen. Zo kan een gebruiker het versturen van een bijlage starten met een mutation, en met een subscription op de hoogte gehouden worden van het resultaat. Zo hoeft een client niet te wachten tot een bijlage verzonden is voordat de server een response terugstuurt.

Conclusie
Tijdens de performancemetingen zal meer duidelijk worden over de voor- en nadelen van GraphQL. Tijdens het onderzoek voor deze deelvraag zijn al wel een aantal situaties ontdekt waar GraphQL nuttig lijkt te zijn voor de gebruikers van Moneybird. Het is echter nog onduidelijk of GraphQL hier ook daadwerkelijk voordelig is, omdat nog niet genoeg bekend is over de performance van GraphQL. Het is ook mogelijk dat de klassieke REST API dusdanig sneller is in een aantal van de situaties dat het niet verstandig zou zijn om GraphQL te gebruiken.

7.3 Implementatie GraphQL

Voor deze deelvraag is onderzoek gedaan naar de verschillende technieken waarmee GraphQL geïmplementeerd kan worden in de Moneybird stack. Hiervoor is onderzocht welke technieken bestaan, en is een prototype gebouwd om de gekozen techniek te testen.

7.3.1 GraphQL Gems

Ruby maakt het mogelijk om extra functionaliteit toe te voegen met Gems. Een Ruby Gem is een stuk software, zoals een programma of een library. Door een Gem toe te voegen aan een Ruby project, kan de code uit de Gem gebruikt worden. Er zijn twee grote Gems beschikbaar voor de implementatie van GraphQL.

GraphQL Ruby Gem
De eerste optie is de GraphQL Ruby Gem (https://graphql-ruby.org/). Deze Gem werkt met Ruby on Rails. Deze Gem bevat een Schema class, welke een GraphQL schema representeert. Door

deze class te extenden is het mogelijk een eigen GraphQL schema te definiëren. Dit schema bevat de root fields die de GraphQL API aanbiedt. Dit zijn de query, mutation en eventueel subscription types. Tijdens het installeren van de Gem worden ook een aantal scalar types aangemaakt. Deze types kunnen gebruikt worden om complexere types op te stellen.

Tijdens het installeren van de Gem wordt een controller aangemaakt voor de /graphql route. In deze controller wordt het rails request object ontleedt. De informatie hieruit die van belang is voor het schema, wordt in een context-object geplaatst. Het schema heeft vervolgens toegang tot dit context-object, om de query te verwerken. Wanneer de query verwerkt is, geeft het schema een response terug. Dit response-object wordt door de controller omgezet naar JSON, en teruggestuurd naar de client.

In het query-type worden de toegankelijke velden gedefinieerd. Dit wordt gedaan op basis van een naam en een type. De naam dient als identificatie voor het veld. Het type kan een scalar zijn, zoals een integer, float of string, maar ook weer een ander object. Een veld kan ook een array aan objecten van hetzelfde type bevatten.

Ook in het mutation-type worden verschillende velden gedefinieerd. Deze velden representeren de mogelijke mutations. Mutations bestaan vaak uit één of meerdere argumenten, en één of meerdere velden. De argumenten geven aan welke gegevens de mutatie nodig heeft om uitgevoerd te worden. De velden geven aan welke gegevens de mutatie terug kan geven, indien hier in het request om gevraagd wordt.

Rack GraphQL Gem

De tweede optie is de Rack GraphQL Gem (https://github.com/RenoFi/rack-graphql). Deze Gem werkt direct met Rack in plaats van Ruby on Rails. Rack is een Ruby interface om webapplicaties te bouwen. Het Ruby on Rails framework maakt zelf ook gebruik van Rack. Wanneer Rack software een request ontvangt, wordt dit request afgehandeld door middleware. Een Rack applicatie bevat vaak meerdere types middleware, die elk een handeling uitvoeren met of op het request. Zo kan bijvoorbeeld middleware geschreven worden die elk request logged, maar ook middleware om de gebruiker voor ieder request te authenticeren. Op deze manier is het makkelijk om modulaire software te schrijven.

De Rack GraphQL Gem bestaat uit middleware. De requests voor het /graphql path worden afgevangen door deze middleware. De middleware kan geconfigureerd worden om een GraphQL schema uit te voeren. Uit de code van de Gem blijkt dat onderwater het schema van de GraphQL Ruby Gem gebruikt wordt. Dit betekent dat de afhandeling van het request vanaf het schema hetzelfde werkt als bij de GraphQL Ruby Gem. Wanneer de middleware de response van het schema terug krijgt, wordt deze ook hier omgezet naar JSON en teruggestuurd naar de client.

Keuze

Uiteindelijk is gekozen om de GraphQL Ruby Gem te gebruiken tijdens de implementatie in de Moneybird applicatie. De Moneybird applicatie maakt zelf gebruik van Ruby on Rails. Daarnaast

is de Rack GraphQL Gem niet meer dan een extensie van de GraphQL Ruby Gem. Het is dus altijd mogelijk om dit zelf op een vergelijkbare manier te implementeren.

7.3.2 Prototype

Om meer te leren over de taal en het framework is een klein blog-project opgezet. Bij dit project is dezelfde softwarestructuur aangehouden als in de Moneybird applicatie. Op deze manier wordt snel duidelijk hoe de architectuur van de Moneybird applicatie in elkaar zit. Ook is dit een goede manier om kennis op te doen over het Ruby on Rails framework.

Omdat Moneybird werkt met Ruby on Rails, is gekozen voor de GraphQL Ruby Gem. Nadat alle functionaliteit in het blog-project toegevoegd is, is ook de GraphQL Ruby Gem toegevoegd. Op deze manier wordt duidelijk welke veranderingen het installeren van de Gem met zich meebrengt in een overzichtelijke omgeving. Zo is goed te zien welke gevolgen de installatie van de GraphQL Gem zal hebben op de Moneybird applicatie.

GraphQL werkte zoals verwacht in het blog-project. De queries kregen de juiste gegevens terug en de mutations werkten ook naar behoren. GraphQL is in het project geïmplementeerd door middel van een controller. Alle fields zijn in het query-type gedefinieerd. Wegens de gelimiteerde omvang van dit project was dit geen probleem. Echter, in een groter project zoals de Moneybird applicatie kan dit snel onoverzichtelijk worden.

7.4 Onderhoud GraphQL

Voor deze deelvraag wordt onderzocht wat er nodig is om de GraphQL API uit te breiden of aan te passen. Hiervoor zijn nieuwe types toegevoegd aan de GraphQL API binnen Moneybird. Het onderzoek voor deze deelvraag is uitgevoerd na de implementatie van GraphQL in de Moneybird applicatie.

7.4.1 Code splitsen

Tijdens de implementatie bleek dat het niet praktisch is om alle velden direct in de root-types te definiëren, zoals in figuur 7-5. Op deze manier zouden de classes met de root-types erg snel vol raken met verschillende velden. Er zijn twee verschillende methoden voor het splitsen van de code.

Resolvers

De eerste methode is het gebruik van resolvers. Resolvers zijn speciale classes uit de GraphQL Ruby Gem. Deze classes nemen de verantwoordelijkheid van het ophalen van een type over. Bij het maken van een resolver wordt aangegeven welk type de resolver ophaalt, eventuele argumenten die hiervoor nodig zijn, en hoe de data voor het type opgehaald wordt. Een voorbeeld van een resolver is te vinden in figuur 7-6.

```
class QueryType < Types::BaseObject
  field :administrations,
        [Types::AdministrationType],
        null: false,
        description: 'Retrieves all active administrations'

  def administrations
    context[:current_user].administrations
  end
end
```

Figuur 7-5: GraphQL root-type zonder resolvers

```
class AdministrationsQuery < BaseQuery
  type [Types::AdministrationType], null: false

  description 'Retrieves all active administrations'

  def resolve
    context[:current_user].administrations
  end
end
```

Figuur 7-6: GraphQL administraties resolver

In de root-type moet dan aangegeven worden welke resolver gebruikt kan worden om de data op te halen. Dit is te zien in figuur 7-7.

```
class QueryType < Types::BaseObject
  field :administrations, resolver: AdministrationsQuery
end
```

Figuur 7-7: GraphQL root-type met resolvers

Concerns

De andere optie voor het splitsen van de code komt uit Ruby on Rails: Concerns. Een concern kan geïmporteerd worden in een class. De code uit de concern komt dan in feite in de class te staan. De code uit de concern kan dus bij variabelen en methoden uit de class, en omgekeerd. Methoden voor het ophalen van data kunnen dus ook verplaatst worden naar een concern, om zo ruimte te besparen. Een voorbeeld van een concern is te zien in figuur 7-8.

```
module MoneybirdGraphql
  module Concerns
    module Users
      extend ActiveSupport::Concern

      def users
        …
      end
    end
  end
end
```
Figuur 7-8: GraphQL users concern

Conclusie
Er is niet veel verschil tussen het gebruik van resolvers en concerns. Het maakt uiteindelijk niet uit welke van de twee methoden gebruikt wordt. Het is echter niet praktisch om beide methoden in hetzelfde project te gebruiken, omdat dit verwarring kan veroorzaken.

7.4.2 GraphQL Types

Een class met een GraphQL type bestaat voornamelijk uit velden. Een voorbeeld van een GraphQL type is te vinden in figuur 7-9.

```
module MoneybirdGraphql
  module Types
    class Administration < Types::BaseObject
      field :id, ID, null: false
      field :name, String, null: false
      …
      field :access, String, null: false

      def access
        object.type_of_access(context[:current_user])
      end
    end
  end
end
```
Figuur 7-9: GraphQL administratietype

Zoals te zien is ook een methode gedefinieerd. Deze methode heeft dezelfde naam als het access-veld. Hierdoor weet GraphQL dat de waarde van het access-veld gevuld moet worden met de return-waarde van de methode.

Het type in figuur 7-9 bevat alleen scalars. Een GraphQL type kan ook andere types bevatten. Een voorbeeld hiervan is het contact-veld in figuur 7-10. Net als het access-veld, heeft het contactveld een methode om de data op te halen. Deze methode geeft een contact terug, en het veld geeft ook aan een contact te bevatten.

```
class Administration < Types::BaseObject
  …
  field :contact, Contact, null: false do
    argument :contact_id, ID, required: true
  end

  def contact(contact_id:)
    …
  end
end
```

Figuur 7-10: GraphQL administratietype met gebruikers

Dit veld bevat ook een argument, in dit geval de ID van de contact die opgehaald moet worden. Het argument wordt bij een query meegegeven door de client. Omdat het veld aangeeft dat het argument bestaat, verwacht de contact-methode ook een argument. Dit argument kan gebruikt worden om het juiste contact op te halen.

7.4.3 Grape entities

De huidige REST API maakt gebruik van het Grape-framework. Hiervoor zijn grape-entities gemaakt. Deze entities bevatten een beschrijving van hoe de entity gerepresenteerd wordt. Deze beschrijving bestaat uit scalar types en soms verwijzingen naar andere entities of lijsten met andere entities. In veel gevallen zijn geen andere entities aanwezig, maar slechts de ID van een entity. De volledige entity kan vervolgens opgevraagd worden met de verkregen ID. In figuur 7-11 is een voorbeeld te vinden van een Grape-entity.

```
class Administration < Grape::Entity
  expose :id
  expose :name
  expose :language
  expose :currency
  expose :country
  expose :time_zone
end
```

Figuur 7-11: Grape-entity

Bij het implementeren van een nieuw GraphQL type kan de Grape-entity gebruikt worden ter inspiratie. Uiteindelijk moeten de velden van GraphQL types overeenkomen met die van de Grape-entities. Het grootste verschil is dat de gerelateerde objecten binnen een GraphQL type altijd complete types zijn, en vrijwel nooit alleen een ID.

7.5 Performance GraphQL

Voor deze deelvraag is onderzoek gedaan naar het verschil in performance tussen een GraphQL API en een REST API. Dit onderzoek is gedaan door middel van een benchmark test.

7.5.1 Metrics

Voor dit onderzoek worden twee metrics vergeleken: latency en throughput. De latency is de tijd tussen het moment dat een request verstuurd wordt naar de server en het moment dat de response ontvangen is. De throughput geeft aan hoeveel requests de server binnen een bepaalde tijd kan afhandelen.

7.5.2 K6

De latency en throughput metrics worden gemeten met behulp van K6. K6 is een open-source tool van Grafana Labs voor het uitvoeren van tests geschreven in Javascript. De tool kan onder andere gebruikt worden voor het load-testen van API's.

Bij het testen van de API worden twee aspecten van K6 gebruikt: de Javascript library, die code bevat voor het schrijven van de tests en de command line tool, waarmee de tests uitgevoerd worden.

7.5.3 Opzet

Voor de tests zijn vijf verschillende scenario's opgesteld. Deze scenario's dekken een groot deel van de handelingen die de gebruikers uitvoeren op de huidige API.

1. Het ophalen van de namen van alle administraties van de huidige gebruiker.
2. Het ophalen van de naam, het e-mailadres en de permissies van elke gebruiker die toegang heeft tot een van de administraties van de huidige gebruiker.
3. Het ophalen van de naam, het e-mailadres, het telefoonnummer en het volledige adres van alle contacten uit een specifieke administratie.
4. Het ophalen van de naam, het e-mailadres, het telefoonnummer en het volledige adres van een specifiek contact in een specifieke administratie.
5. Het ophalen van de prijzen van alle producten van alle facturen in alle administraties van de gebruiker.

Elk scenario wordt uitgevoerd op de REST API en twee keer op de GraphQL API. Hierbij wordt één keer een query gestuurd waarin alle velden van het object opgehaald worden, en één keer een query waarin alleen de nodige velden opgehaald worden.

Latency
Om de latency van elk scenario accuraat te meten, wordt ieder scenario honderd keer uitgevoerd. Op basis van deze honderd metingen wordt de gemiddelde latency voor het scenario

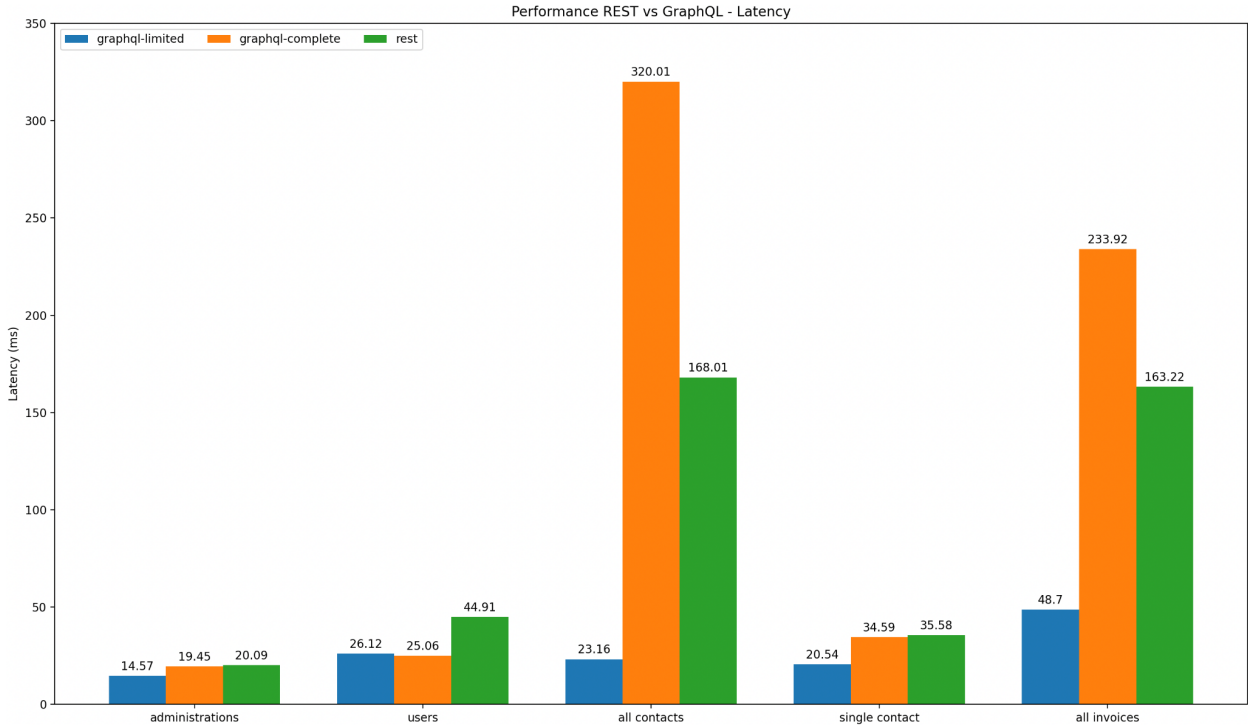
berekend. Wanneer meerdere requests nodig zijn om aan het scenario te voldoen, wordt de volledige tijd van het scenario gemeten.

Throughput
Voor het meten van de throughput worden dezelfde scenario's gebruikt als bij de latency. Ieder request wordt vijftien seconden lang achter elkaar gestuurd. Op basis van het aantal requests dat verstuurd is in deze tijd, wordt het aantal requests per seconde berekend. Deze tests zijn uitgevoerd met één, tien en vijftig virtuele gebruikers. Elke virtuele gebruiker kan één request tegelijk sturen, dus bij vijftig gebruikers worden vijftig requests tegelijk gestuurd.

7.5.4 Resultaten

De resultaten van de metingen zijn verwerkt in tabellen. Deze tabellen zijn te vinden in hoofdstuk 8 van het onderzoeksrapport in bijlage A. Daarnaast zijn de resultaten ook te vinden in grafieken, waardoor het verschil tussen de technieken goed zichtbaar is.

Latency
De resultaten van de latency metingen zijn te vinden in figuur 7-12.



Figuur 7-12: Grafiek latency metingen

Bij scenario één wordt een klein object opgehaald door middel van een enkel request. Hierbij is te zien dat GraphQL minder tijd nodig heeft dan REST. Bij het ophalen van het volledige object is dit verschil minimaal, maar als slechts enkele velden geselecteerd worden in de query is GraphQL aanzienlijk sneller.

Bij het tweede scenario zijn meerdere REST request nodig om alle data op te halen. Met GraphQL kan deze data opgehaald worden in een enkel request. Hierdoor is GraphQL aanzienlijk sneller dan REST. In dit geval zit er weinig verschil tussen het ophalen van het volledige object en het ophalen van alleen de nodige velden. Bij dit request wordt nog steeds gewerkt met een klein object, waardoor deze verschillen minimaal zijn.

Bij het derde scenario zijn duidelijke verschillen te zien tussen de drie tests. Bij dit request worden meerdere grote objecten opgehaald. Wanneer deze objecten volledig opgehaald worden met GraphQL kost dit veel tijd, zelfs meer dan het REST request. Als echter alleen de nodige velden geselecteerd worden, is GraphQL vele malen sneller dan REST.

Scenario vier is vergelijkbaar met scenario één, maar met een groter object. Hier is duidelijk te zien dat GraphQL sneller is dan REST, zeker wanneer minder velden geselecteerd worden. Wanneer alle velden geselecteerd worden is het verschil tussen GraphQL en REST minimaal.

Bij het laatste scenario worden meerdere grote objecten opgehaald op basis van meerdere REST requests. Hier is opnieuw duidelijk te zien dat GraphQL sneller is dan REST, indien alleen de nodige velden opgehaald worden.

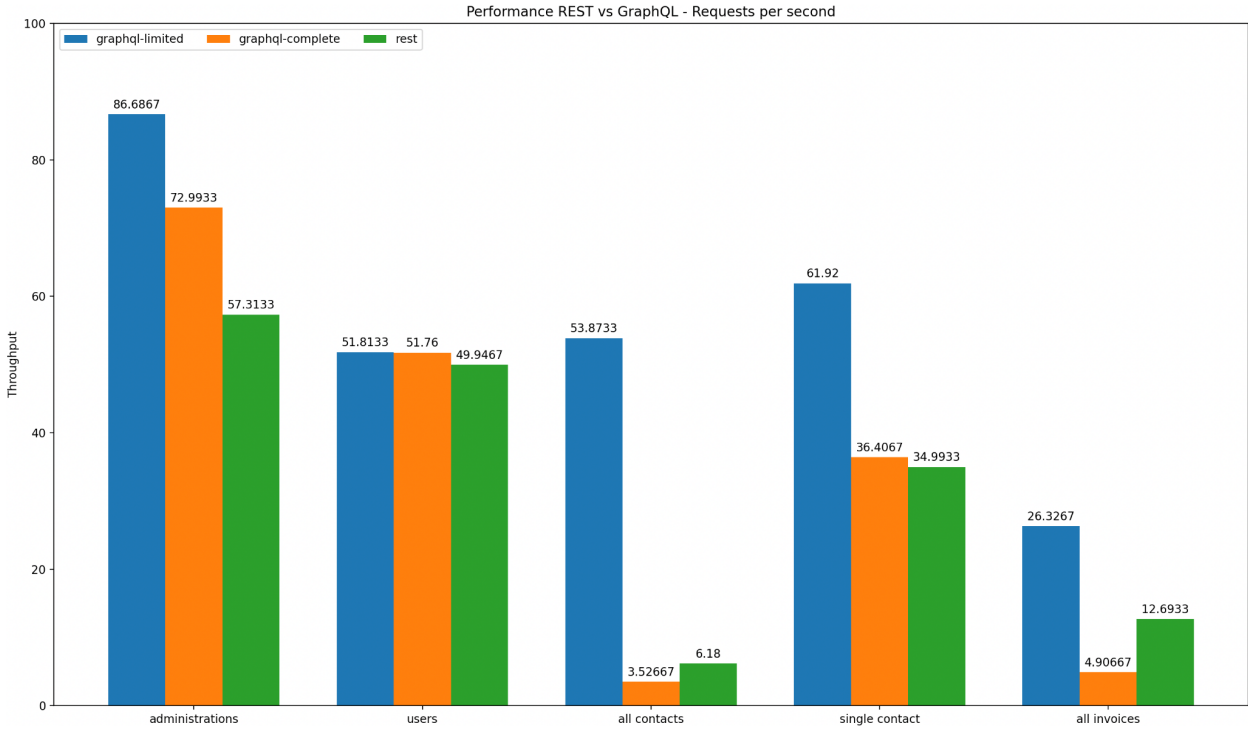
Het is te zien dat het ophalen van de prijs van facturen (scenario vijf) met het selecteren van velden langer duurt dan het ophalen van de contactinformatie (scenario drie). Dit is vermoedelijk omdat de prijs een niveau dieper zit dan de contactinformatie. De contactinformatie is te vinden in het contactobject, die direct in de administratieobjecten te vinden is. De prijs is te vinden in een detailobject, die zich bevindt in een factuurobject, die zelf weer in een administratieobject zit.

Throughput

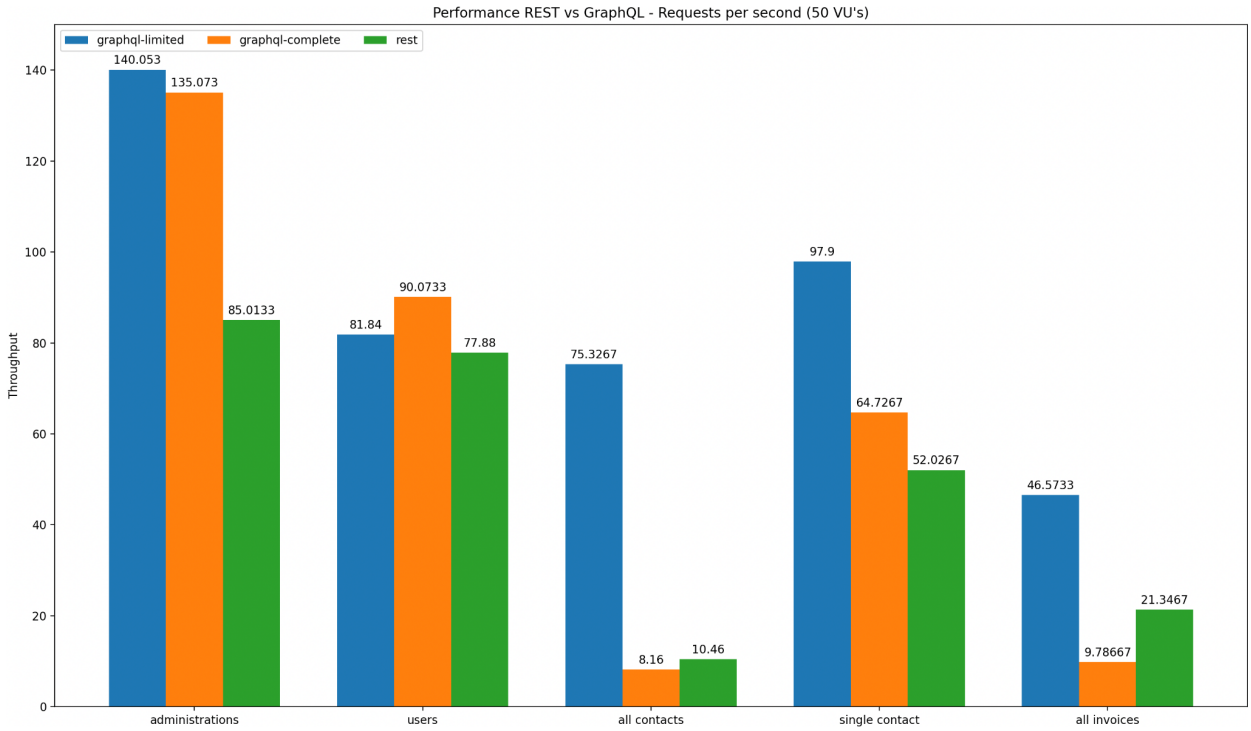
De throughput is gemeten in requests per seconde (r/s). De resultaten van de metingen met meerdere virtuele gebruikers zijn erg vergelijkbaar. Om deze reden zijn alleen de metingen met een enkele virtuele gebruiker en vijftig virtuele gebruikers meegenomen in dit verslag. De resultaten van de andere metingen zijn te vinden in hoofdstuk 8 van het onderzoeksrapport in bijlage A. De resultaten van de metingen met een enkele virtuele gebruiker zijn te vinden in figuur 7-13. De resultaten van de metingen met vijftig virtuele gebruikers zijn te vinden in figuur 7-14.

De resultaten van de throughput metingen bevestigen de resultaten van de latencymetingen. De scenario’s waarbij de latency hoog is, hebben tevens een lagere throughput. Dit is logisch, omdat een kortdurend request sneller achter elkaar uitgevoerd kan worden dan een langdurig request.

Deze metingen bevestigen dat GraphQL in de meeste gevallen sneller is dan REST, zeker wanneer alleen de nodige velden opgehaald worden. Bij grotere objecten kan REST sneller zijn, indien een grote hoeveelheid velden opgehaald moet worden



Figuur 7-13: Grafiek throughput metingen enkele virtuele gebruiker



Figuur 7-14: Grafiek throughput metingen 50 virtuele gebruikers

[8] Implementatie

In dit hoofdstuk is te vinden hoe de implementatie van GraphQL in de software stack van Moneybird is verlopen. In de eerste paragraaf is informatie te vinden over de software stack van Moneybird. In de tweede paragraaf wordt het implementatieplan besproken. In de derde paragraaf is het testplan te vinden. In de vierde paragraaf wordt besproken hoe de implementatie is verlopen en welke keuzes hierbij zijn gemaakt. In de vijfde paragraaf zijn opmerkingen te vinden over het implementatieproces.

8.1 Software stack Moneybird

De software stack van Moneybird bevat veel onderdelen. Tijdens het afstudeertraject is alleen gewerkt met de Moneybird applicatie geschreven in Ruby on Rails. Om deze reden is alleen Ruby on Rails opgenomen in dit hoofdstuk.

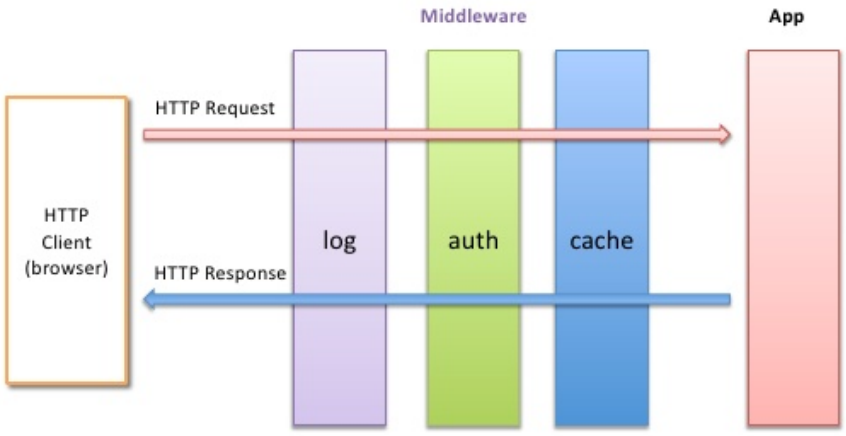
8.1.1 Ruby on Rails

Ruby on Rails is een van de meest populaire frameworks voor de programmeertaal Ruby. Ruby on Rails wordt gebruikt voor het ontwikkelen van webapplicaties, zoals Moneybird. Onder water draait Ruby on Rails op Rack. Rack is de onderliggende technologie voor veel Ruby-frameworks. Het is een interface voor het bouwen van webapplicaties. Om aan de rack-interface te voldoen, moet de code drie karakteristieken hebben:

- De code moet reageren op de call methode;
- De call methode moet een enkel argument accepteren, waarin alle data over het request aanwezig is; en
- De call methode moet een array met drie elementen teruggeven. Deze elementen zijn de statuscode, response headers en response body van het request.

Middleware

Een tweede deel van rack technologie is middleware. Elke middleware is een rack-applicatie. Door meerdere rack-applicaties samen te voegen wordt de uiteindelijke applicatie gebouwd. Wanneer een request binnenkomt wordt deze doorgegeven aan de eerste rack-applicatie. Als deze applicatie klaar is met het request, wordt het naar de volgende applicatie gestuurd. Op deze manier loopt het request door alle middleware heen. Dit is goed te zien in figuur 8-1.



Figuur 8-1: Rack middleware (Gao, z.d.)

Elk stuk middleware voert een stuk code uit op basis van het request. Sommige applicaties gebruiken data uit het request om ergens anders iets te wijzigen, zoals het loggen van alle requests. Andere andere applicaties passen het request-object zelf aan. Denk hierbij aan bijvoorbeeld authenticatie. Als de gebruiker geauthenticeerd is, wordt de volgende rack-applicatie aangeroepen. Als de gebruiker niet geauthenticeerd is, stuurt de middleware statuscode 401 terug. Op deze manier kan een volledige applicatie gebouwd worden.

MVC

Ruby on Rails werkt volgens de model-view-controller-architectuur. Deze architectuur verdeelt het framework in drie delen. Het eerste deel is het model. Het model bevat de data-structuur van de applicatie. In het geval van de blog, bevat het model de artikelen en de comments. Het tweede deel is de view. De view bevat het visuele deel van de applicatie. Denk hierbij aan de HTML waarmee de website geladen wordt, maar ook een JSON representatie van een object. Het derde deel is de controller. De controller verbindt de data aan de view, en bevat de logica van de applicatie. In de werkelijkheid zit een groot deel van de logica niet in de controller, maar op andere plekken in de code. In het geval van Moneybird, zit de logica voornamelijk in mutations.

8.1.2 Mutations

De Moneybird applicatie is een groot project met veel verschillende classes. Om de code overzichtelijk te houden, worden een aantal design patterns toegepast. Een van deze patterns is het gebruik van mutations. Dit zijn een ander type mutations dan de GraphQL mutations, hoewel ze in de basis wel op elkaar lijken.

Mutations encapsuleren een specifiek stuk logica uit de applicatie. Neem bijvoorbeeld het rails blog project uit het prototype voor de implementatie van GraphQL. Een van de mutaties binnen dit project is verantwoordelijk voor het aanmaken van artikelen. Het aanmaken van een artikel is een losstaande operatie, waarvoor een aantal gegevens nodig zijn. Als de handeling succesvol uitgevoerd wordt, zou de operatie een artikel moeten opleveren.

De mutations in Moneybird zijn gebouwd met een Gem (https://github.com/cypriss/mutations). Deze gem bevat de basisclasses voor het maken van mutations. Mutations bestaan uit drie onderdelen. De eerste van deze drie delen gaat over de argumenten die nodig zijn voor het uitvoeren van de mutation. Bij het aanmaken van een artikel zijn een aantal argumenten nodig. Het eerste argument is de gebruiker die het artikel aanmaakt. Het tweede argument is een omschrijving van het artikel, met de titel en inhoud. Een voorbeeld van dit deel is te zien in figuur 8-2.

```
required do
  model :user
  hash :article do
    optional do
      string :title, empty: true
      string :body, empty: true
    end
  end
end
```

Figuur 8-2: Mutation argumenten

Het tweede deel van de mutation is de validatiemethode. Wanneer een mutation aangeroepen wordt, wordt eerst de validatiemethode uitgevoerd. Deze methode is verantwoordelijk voor het afvangen van fouten voordat de mutatie uitgevoerd wordt. Denk hierbij aan ongeldige argumenten, of missende rechten. Een voorbeeld hiervan is te zien in figuur 8-3.

```
def validate
  add_error(:create_article, :no_permission) unless user.can_create?
end
```

Figuur 8-3: Mutation validatie

Het derde deel is verantwoordelijk voor het uitvoeren van de mutatie. Dit is waar het artikel aangemaakt wordt. Als de operatie succesvol verloopt, wordt het aangemaakte artikel teruggestuurd. Een voorbeeld hiervan is te vinden in figuur 8-4.

```
def execute
  article['author'] = user
  Article.create(article)
end
```

Figuur 8-4: Mutation executie

Er zijn twee manieren om een mutatie aan te roepen. De eerste manier is met de run-methode. Deze methode geeft een object terug, waarin het resultaat van de operatie te vinden is. Een voorbeeld van het aanroepen van de mutatie op deze manier is te vinden in figuur 8-5.

```
outcome = CreateArticle.run(params)
if outcome.success?
  user = outcome.result
else
  render outcome.errors
end
```

Figuur 8-5: Mutation aangeroepen met run

Dit is een verbose manier voor het uitvoeren van mutaties. De andere optie maakt gebruik van exceptions. Wanneer de mutatie uitgevoerd wordt met de run!-methode, wordt het aangemaakte artikel direct teruggegeven. Als de operatie niet succesvol is, zal een exception plaatsvinden. Een voorbeeld hiervan is te vinden in figuur 8-6.

```
begin
  user = CreateArticle.run!(params)
rescue Mutations::ValidationException
  render outcome.errors
end
```

Figuur 8-6: Mutation aangeroepen met run!

Het voordeel van deze methode, is dat fouten niet direct afgehandeld hoeven te worden. Een controller kan de mutatie aanroepen, en een centrale plek hebben voor afhandeling van exceptions.

8.1.3 Grape API

De Moneybird applicatie bevat op dit moment een REST API. Deze REST API werkt met het Grape-framework (https://www.ruby-grape.org/). Grape is geïnstalleerd met de Grape Gem. Deze Gem implementeert het framework als een rack-applicatie. Het definiëren van de call-methode gebeurt echter onderwater, de gebruiker hoeft alleen de Grape::API class te extenden.

Grape maakt het makkelijk om code op te splitsen. Het is makkelijk om code te categoriseren op basis van onderdelen van het path. Zo zijn bijvoorbeeld alle endpoints voor de administratie in een aparte class gezet, die ook Grape::API extend. Omdat een zeer groot deel van de requests via een administratie gaat, is ook deze code weer verdeeld over meerdere classes. Op deze manier is de Grape API erg modulair opgebouwd.

Entities
Naast de Grape Gem bevat de Moneybird ook een andere Gem, genaamd grape-entity. Deze Gem maakt het makkelijk om model-classes te gebruiken in een Grape API. De manier waarop dit werkt lijkt erg op de GraphQL types. Een entity definieert welke velden beschikbaar zijn voor de API.

Op dit moment is voor vrijwel elk endpoint in de API een grape-entity gemaakt. Dit maakt het implementeren van de GraphQL types makkelijker, omdat al duidelijk is welke velden beschikbaar moeten zijn in de API.

8.2 Implementatieplan

Voor het implementeren van GraphQL is een plan gemaakt waarin beschreven is welke stappen gezet moeten worden. De stappen zijn het installeren en configureren van GraphQL, het toevoegen van authenticatie, het implementeren van het administratietype en het toevoegen van de overige types.

8.2.1 Installatie en configuratie

De eerste stap van de implementatie van GraphQL in de Moneybird applicatie is het toevoegen van de GraphQL Ruby Gem. Deze Gem heeft de voorkeur boven de Rack GraphQL Gem, omdat Moneybird werkt met een Rails applicatie. De Rack GraphQL Gem maakt onderwater gebruik van de GraphQL Ruby Gem, waardoor het logischer is om de GraphQL Ruby Gem direct te gebruiken.

De GraphQL Ruby gem bevat commando's voor het installeren van GraphQL in het Rails project. Deze commando's genereren een aantal bestanden. Allereerst wordt het schema gegenereerd. Het schema is de basis van de GraphQL implementatie. Daarnaast wordt ook een controller gegenereerd. Deze controller vangt alle requests naar het path /graphql af, en stuurt de informatie uit het request door naar het schema. Het schema voert de query uit met behulp van een aantal gegenereerde types.

8.2.2 Authenticatie

De tweede stap in de implementatie is het toevoegen van authenticatie. Een van de objecten die de controller meestuurt naar het schema is het context object. Door de gebruiker in dit object te zetten, kan in ieder GraphQL type de gebruiker opgehaald worden uit het context object.

In de controller wordt het request-object uitgepakt. De gebruiker kan een authentication-header meesturen met daarin een bearer-token. Het token moet nu omgezet worden naar een user-object. Dit is waar de kracht van de mutations in Moneybird naar voren komt. Er is een mutation beschikbaar voor het authenticeren van een gebruiker op basis van een token. Deze mutation wordt ook gebruikt door de Grape API. Door de mutation aan te roepen met het token als argument wordt de gebruiker opgehaald.

8.2.3 Foutafhandeling

Een belangrijk onderdeel van de implementatie is foutafhandeling. Fouten die optreden tijdens het uitvoeren van het request worden in het schema afgevangen. Het schema geeft dan de fout terug in een formaat dat makkelijk terug te sturen is naar de gebruiker. In dit geval is de controller dus alleen verantwoordelijk voor het omzetten van dit object naar JSON.

De controller is verantwoordelijk voor het uitpakken van het request. Het kan ook gebeuren dat de controller een ongeldig request ontvangt, bijvoorbeeld vanwege incorrecte JSON in de body. In dit geval zal de controller een response naar de gebruiker sturen, zonder dat het schema wordt aangeroepen. Dit response volgt wel het formaat dat GraphQL gebruikt voor fouten, zoals beschreven in hoofdstuk 7.1.2.

8.2.4 Administratie type

Met de authenticatie werkend, kan het eerste type geïmplementeerd worden. Een groot deel van de types zijn afhankelijk van een administratie. De administratie is dus het eerste type dat toegevoegd wordt. De velden in dit type worden gebaseerd op de velden uit de grape-entity voor administraties.

Er wordt een class aangemaakt voor het administratietype, waarin alle velden gedefinieerd worden. Dit type wordt vervolgens toegevoegd aan het query-type in twee vormen: als enkele administratie en als lijst van administraties. In de eerste vorm is een ID vereist als argument. Met behulp van een mutation wordt de administratie met de gegeven ID opgehaald, mits de gebruiker toegang heeft tot deze administratie. De tweede vorm geeft simpelweg een lijst terug van alle administraties waarvoor de gebruiker geautoriseerd is.

8.2.5 Overige types toevoegen

Uit het onderzoek naar het gebruik van de huidige API bleek dat een groot deel van de requests gemaakt worden naar de endpoints voor facturen en contacten. Deze types worden meegenomen in het onderzoek naar performance, en moeten dus geïmplementeerd worden.

Het contact-type lijkt iets simpeler te zijn dan het factuur-type. Het contact-type zal eerst toegevoegd worden om beter bekend te raken met het toevoegen van GraphQL types. Wanneer dit type toegevoegd is, wordt het factuur-type toegevoegd.

8.3 Testplan

Binnen Moneybird wordt alle code getest. Dit gebeurt met behulp van het RSpec test framework (https://rspec.info/). Het RSpec framework werkt met specs. Een spec is niet veel meer dan een andere naam voor een test. Moneybird past twee verschillende soorten tests toe: unit tests en system tests. Bij unit testen wordt een enkele class getest. De spec roept methodes aan op de class en controleert of de uitkomst van de methode is zoals verwacht. Een system test is iets uitgebreider. Hierbij wordt een groter stuk code getest, zoals een controller. Bij een system test wordt de controller geactiveerd door een echt request te sturen naar de server, en te controleren of alles loopt zoals verwacht.

Uiteraard moet de code voor de GraphQL implementatie voldoen aan alle eisen van Moneybird. Dit betekent dat voor alle code in ieder geval een unit test geschreven moet worden. Daarnaast

moet de controller ook getest worden met een system test. De website voor de GraphQL Ruby Gem (https://graphql-ruby.org) geeft ook advies voor het testen van GraphQL. Hierbij wordt onderscheid gemaakt tussen twee soorten tests.

Structure tests
Een GraphQL API verandert vaak over tijd. Soms worden velden toegevoegd, of in sommige gevallen ook verwijderd. Wanneer een veld verwijderd wordt, is er sprake van een breaking change. De API biedt geen ondersteuning meer voor het verwijderde veld. Het opvragen van het veld zal dus leiden tot een foutmelding.

Om breaking changes te vermijden worden structure tests geschreven. Bij een structure test worden een aantal vooraf opgestelde queries uitgevoerd op het schema. Het is dus van belang dat de opgestelde queries alle verschillende velden bevatten. Wanneer een breaking change plaatsvindt, zal de API een foutmelding geven. Deze foutmelding wordt afgevangen tijdens de structure test, zodat de ontwikkelaars van tevoren op de hoogte zijn van een breaking change. Op deze manier kan de verandering ongedaan gemaakt worden, of de versie van de API aangepast worden om de verandering aan te duiden.

Runtime tests
Een GraphQL API moet net als alle andere code getest worden. De voornoemde unit- en system tests vallen dan ook onder runtime tests. Binnen runtime tests wordt onderscheid gemaakt tussen drie niveaus.
Het eerste niveau is het applicatieniveau. Hier wordt de business logica van de applicatie getest. De GraphQL API maakt voor een groot deel gebruik van de bestaande mutaties, waarvoor al tests zijn geschreven. Er zullen dus niet veel tests van dit type geschreven worden tijdens de implementatie, omdat deze voor een groot deel al bestaan. Wanneer dit niet het geval is, zal een nieuwe test worden geschreven. Dit type test is vergelijkbaar met een unit test.

Het tweede niveau is het interface-niveau. Dit type test heeft meer weg van een system test. Een goede manier om dit niveau te testen is door middel van het schema. Een test kan bijvoorbeeld het schema direct aanroepen met een query en controleren of het schema de juiste data teruggeeft. Hier kan ook authenticatie getest worden, omdat een context meegegeven kan worden aan het schema. Dit type test wordt geschreven voor alle geïmplementeerde types.

Het derde en laatste niveau is het transportniveau. Dit niveau test de volledige API door middel van een echt request. Binnen een test wordt een request gedaan naar de API. Dit request bevat vergelijkbare data met een request van een echte client. Vervolgens wordt de response van de API afgevangen en gecontroleerd.

8.4 Verloop implementatie

De uiteindelijke implementatie is anders verlopen dan beschreven in het implementatieplan. In deze paragraaf wordt beschreven hoe de implementatie is verlopen en welke keuzes daarbij zijn gemaakt.

8.4.1 Installatie

De installatie van de GraphQL Gem begon zoals gepland. De GraphQL Gem is eerst toegevoegd aan de Gemfile. De Gemfile is een file in Ruby on Rails die alle Gems in het project bijhoudt. Vervolgens is het `bundle install` commando uitgevoerd. Dit commando installeert alle Gems in de Gemfile. Na de installatie van de Gem is het volgende commando uitgevoerd om de nodige GraphQL classes te genereren: `rails generate graphql:install`.

Ruby on Rails heeft een bestand waarin alle endpoints gedefinieerd staan, genaamd routes.rb. De laatste stap in de installatie was het toevoegen van de /graphql route aan het routes.rb bestand. Tot slot is het `bundle install` commando nog een laatste keer uitgevoerd, om alle wijzigingen van het installatie commando uit te voeren.

GraphiQL
Het installatie commando heeft ook een nieuwe Gem toegevoegd, genaamd GraphiQL. GraphiQL is een ingebouwde tool voor het bouwen van GraphQL queries. Met behulp van introspection is de GraphiQL tool op de hoogte van alle types en velden in de API. Tijdens het ontwikkelen is geen gebruik gemaakt van GraphiQL, dus de Gem is uit de Gemfile gehaald waardoor deze niet meer ondersteund wordt. Op deze manier wordt geen onnodige functionaliteit toegevoegd aan de applicatie.

8.4.2 Rack applicatie

De standaard installatieprocedure van GraphQL maakt een nieuwe directory aan in de app folder. De app folder in een Ruby on Rails project bevat alle logica van een webapplicatie. De Grape API bevindt zich in de api-folder, welke in de app-folder staat. Om alle api-gerelateerde code bij elkaar te houden, is besloten om de GraphQL code te verplaatsen naar de api-folder.

Dit heeft als gevolg dat de /graphql route niet meer op dezelfde manier benaderd kan worden. Om dit op te lossen, is de GraphQL controller class vervangen voor een rack-applicatie. Bij het ontwikkelen van code is het goed om het principe van single-responsibility te volgen. Het is voor de API niet belangrijk of deze via een controller aangeroepen wordt, of via een rack-applicatie. Om deze implementatie los te halen van de GraphQL code, is de code voor de rack-applicatie verplaatst naar de lib-folder.

In een subdirectory van de lib-folder is een bestand aangemaakt genaamd api.rb. Dit bestand is verantwoordelijk voor de integratie tussen GraphQL en de rack-interface. In het bestand is een Api class gedefinieerd. Deze class bevat de call-methode die vereist is voor rack-applicaties. De

call methode ontvangt het rack-request-object, en roept een execute-methode aan. De execute-methode in deze class geeft een foutmelding wanneer deze aangeroepen wordt. Dit is omdat de class bedoeld is als basis voor de GraphQL implementatie. In de api-folder kan nu een graphql_api.rb bestand gemaakt worden, met een class die de Api class extend en de execute-methode implementeert. Op deze manier is de integratie met rack gescheiden van de GraphQL logica.

8.4.3 Request object

Zoals eerder genoemd, bevat de Moneybird applicatie een mutation voor het ophalen van de gebruiker met behulp van een api-token. Deze mutation heeft het token nodig om de gebruiker op te kunnen halen. De huidige versie van de mutation gebruikt echter het Rails request-object voor het ophalen van het token. Dit request werkt anders dan het request-object dat gebruikt wordt in de rack-applicatie.

Om dit op te lossen, is de mutation omgeschreven om een api-token als argument te ontvangen, in plaats van een request-object. Om toch backward-compatibility te behouden, is een tweede mutation gebruikt welke wel een request-object als argument heeft. Deze mutation kan het token uit het request halen, en doorsturen naar de nieuwe authenticatie-mutation.

Naar aanleiding van een code-review is toch besloten om een andere oplossing te gebruiken. Door een deel van de logica in de Api-class te verplaatsen naar een request-object, hoeft de authenticatie in Moneybird niet aangepast te worden. Hierdoor is ook een deel verantwoordelijkheid bij de Api-class weggehaald.

De nieuwe request-class is ook aan de lib-folder toegevoegd. De class werkt als een wrapper voor het rack-request. Op deze manier kan het request ook direct meegegeven worden aan de authenticatie-mutation. Op deze manier kan de Api-class ook makkelijker bij de informatie uit het rack-request, zoals de request-parameters, headers en de request-body.

[9] Conclusie

In dit hoofdstuk is de conclusie van het onderzoek te vinden. In de eerste paragraaf wordt de hoofdvraag beantwoord aan de hand van de deelvragen. In de tweede paragraaf is een samenvatting van alle verkregen informatie te vinden. In de derde paragraaf wordt de projectorganisatie besproken.

9.1 Antwoord hoofdvraag

De hoofdvraag voor dit onderzoek luidt: “Op welke manier kan de implementatie van GraphQL de ervaring van zowel klanten als ontwikkelaars verbeteren?” Tijdens dit onderzoek is duidelijk geworden waar de voor- en nadelen van GraphQL liggen ten opzichte van REST.

Het ondersteunen van GraphQL heeft meerdere voordelen voor de gebruikers van Moneybird. Ten eerste is GraphQL flexibeler dan de bestaande REST API. Gebruikers kunnen zelf aangeven welke velden zij op willen halen. Op deze manier beschikken de gebruikers altijd precies over de data die zij nodig hebben.

Omdat GraphQL werkt als een grafiek, is het makkelijk om verschillende objecten op te halen die aan elkaar gerelateerd zijn. Hierdoor kan met een enkel GraphQL request bereikt worden waar meerdere REST requests voor nodig zijn.

Deze eigenschappen van GraphQL zijn niet alleen voordelig als het gaat om het gemak van GraphQL, maar hebben ook een positieve invloed op de performance van GraphQL. Wanneer gebruikers aangeven welke velden nodig zijn, hoeft de response geen onnodige data te bevatten. Dit heeft zeker bij grotere objecten een grote positieve impact op de latency. Ook het combineren van REST requests in een enkel GraphQL request leidt tot betere performance.

Voor de ontwikkelaars van Moneybird is er weinig verschil tussen het onderhouden van een REST API en het onderhouden van een GraphQL API. Voor de bestaande Grape API worden entities geschreven waarin de responses van de API beschreven zijn. Deze entities zijn erg vergelijkbaar met de GraphQL types, die de responses van de GraphQL API beschrijven. Uiteindelijk maakt het voor de ontwikkelaars weinig uit of zij Grape entities of GraphQL types schrijven.

9.2 Proces

Tijdens de afstudeerperiode van 17 februari tot 7 juli is onderzocht hoe GraphQL de ervaring voor zowel gebruikers als ontwikkelaars van Moneybird kan verbeteren. Bij dit proces waren een aantal stakeholders betrokken:

- De student, welke het onderzoek verricht heeft;
- Moneybird, het bedrijf waarvoor het onderzoek verricht is;
- De afstudeerbegeleiders vanuit Moneybird, welke de student hebben begeleid tijdens het afstudeertraject; en
- De afstudeerdocent, die de student tijdens het afstudeertraject heeft begeleid vanuit Saxion.

Het doel van dit onderzoek was om te achterhalen waar de voor- en nadelen van GraphQL ten opzichte van REST liggen in de context van de Moneybird applicatie. Hierbij was het aan de student om onder begeleiding het onderzoek uit te voeren. Aan het eind van het traject werd de student verwacht om een duidelijk overzicht te bieden van de voor- en nadelen van GraphQL, en GraphQL op een schaalbare manier te implementeren in de bestaande Moneybird stack.

Om het proces goed te overzien is de kanbanmethode gebruikt. Deze methode is gekozen omdat het een goed overzicht biedt van welke taken gedaan zijn en welke taken nog uitgevoerd moeten worden. Daarnaast zijn wekelijkse sessies gehouden met de afstudeerbegeleiders, zodat de student regelmatig feedback krijgt op het verloop van het onderzoek. Ook werd er elke vier weken een sessie met de afstudeerdocent gehouden, waarin de student verdere feedback kreeg op het proces.

Naast de methoden om het proces te waarborgen, zijn ook methoden voor de waarborging van de kwaliteit van het onderzoek opgesteld. Zo is voor alle documenten de eis gesteld dat deze voldoen aan de eisen vanuit Saxion. Ook is alle geschreven code onderwerp geweest van code-reviews en geautomatiseerde tests. Op deze manier voldoet alle code aan de standaarden van Moneybird.

Over de periode van het proces is het gebruik van het kanbanbord langzaam minder geworden. Aan het begin van het onderzoek zijn globale tickets aangemaakt op het bord. De bedoeling was dat deze tickets verder uitgewerkt zouden worden wanneer ze opgepakt werden. In de praktijk is dit niet altijd gebeurd. Tijdens de wekelijkse sessies zijn de begeleiders op de hoogte gebracht van de voortgang van die week. Om deze reden is het gebrek aan het gebruik van de kanbanmethode geen groot probleem geworden.

De methoden voor de kwaliteitswaarborging van het onderzoek hebben beter gewerkt. Alle documenten voldoen aan de standaarden zoals gesteld door Saxion. Daarnaast is ook alle code bekeken tijdens code-reviews en zijn tests geschreven voor alle geschreven code. Zowel de reviews als de tests hebben er tijdens het traject meerdere keren voor gezorgd dat fouten voorkomen zijn.

Aan het eind van het traject is het duidelijk waar de verschillen tussen GraphQL en REST liggen. Ook is er veel vooruitgang geboekt bij de integratie van GraphQL in de Moneybird applicatie. De keuze om kanban te gebruiken tijdens dit project is niet verkeerd geweest. Kanban biedt veel overzicht in de taken van een project wanneer juist toegepast. Dit is een verbeterpunt voor een volgend project.

[10] Discussie

In dit hoofdstuk wordt besproken welke onderdelen van het onderzoek open zijn voor interpretatie.

10.1 Literatuuronderzoeken

Tijdens het onderzoek is veel informatie verkregen uit voorgaande onderzoeken. Het is mogelijk dat deze onderzoeken niet correct of compleet zijn. Het grootste deel van de informatie die verkregen is uit voorgaande onderzoeken is echter niet essentieel voor de conclusie van het onderzoek. De literatuurstudie heeft voornamelijk geleid tot een begrip van de werking van GraphQL. Deze werking is bevestigd tijdens het bouwen van de prototypes en de implementatie van GraphQL in de Moneybird API.

In het onderzoek van Lawi et al. (2021) werd gesteld dat de klassieke REST API een snellere responstijd en hogere throughput heeft dan een GraphQL API. Dit is tegenstrijdig met de resultaten van het onderzoek naar de performance van GraphQL binnen Moneybird. Een mogelijke verklaring hiervoor is het verschil in de grootte en structuur van zowel de REST-responses als de GraphQL-responses.

10.2 Klantonderzoek

Tijdens het onderzoek naar het gebruik van de huidige API zijn alle conclusies gebaseerd op API requests. Deze requests geven geen compleet beeld van de intenties van de gebruikers. Tevens zijn deze requests geanonimiseerd, waardoor niet zeker is welke requests door welke gebruikers uitgevoerd zijn. Het is daarom mogelijk dat de conclusies die gebaseerd zijn op de analyse van de API logs niet accuraat zijn.

De gevolgen van een inaccuraat onderzoek van de API logs zijn beperkt. Het onderzoek van de API logs heeft voornamelijk impact gehad op de types die geïmplementeerd zijn in het prototype van GraphQL. Natuurlijk is uit dit onderzoek ook geconcludeerd dat GraphQL de hoeveelheid benodigde requests voor sommige klanten kan verlagen. Het is mogelijk dat de exacte use-cases die uit het onderzoek zijn gebleken geen voordeel hebben aan GraphQL. Echter zijn de performance voordelen van GraphQL niet te ontkennen. Uiteindelijk zijn de voordelen van GraphQL dus zeker aanwezig.

[11] Aanbevelingen

In dit hoofdstuk wordt besproken welke vervolgstappen Moneybird kan nemen om het onderwerp van GraphQL verder te onderzoeken.

11.1 API documentatie

Het onderzoek naar API documentatie voor GraphQL was gepland als onderdeel van dit project, maar wegens een tekort aan tijd is dit niet aan bod gekomen. GraphQL API's bevatten introspectie, waardoor alle mogelijke types en velden opgevraagd kunnen worden via de API. Er zijn verschillende tools die hier gebruik van maken om API documentatie te genereren voor GraphQL. Deze tools kunnen voor Moneybird interessant zijn, als zij ervoor kiezen om GraphQL te implementeren.

Het automatisch genereren van API documentatie heeft meerdere voordelen. Ten eerste scheelt het werk voor de Moneybird ontwikkelaars. Bij het toevoegen van nieuwe functionaliteit hoeft er geen tot weinig documentatie meer geschreven te worden. Hierdoor komt meer tijd vrij voor het ontwikkelen van nieuwe features. Daarnaast is het ook handig voor gebruikers van Moneybird. Als de API documentatie gegenereerd wordt, is de documentatie altijd up to date.

11.2 Security

Tijdens dit onderzoek is niet veel ingegaan op de security-aspecten van GraphQL. De GraphQL Ruby Gem heeft functionaliteit ingebouwd om de API te beveiligen. Een aantal voorbeelden hiervan zijn:

- Het limiteren van complexiteit in queries. Het is mogelijk om een complexiteit toe te wijzen aan een query, en een maximaal toegestane complexiteit in te stellen.
- Het limiteren van de diepte van queries. Het is mogelijk om een maximale diepte in te stellen, zodat queries niet te groot kunnen worden.
- Het instellen van een time-out. Wanneer het uitvoeren van een query te lang duurt, kan deze afgekapt worden.

Dit zijn allemaal aspecten die erg belangrijk zijn in een productieomgeving. Voordat GraphQL volledig toegankelijk gemaakt wordt, moeten deze onderwerpen onderzocht worden.

11.3 GraphQL en REST

Uiteindelijk hebben GraphQL en REST beide voor- en nadelen. GraphQL maakt het makkelijker om grotere hoeveelheden gerelateerde data op te halen door middel van de queries. Het is voor de gebruiker mogelijk om aan te geven welke velden wel en niet nodig zijn, waardoor er minder data verstuurd hoeft te worden. Voor de ontwikkelaars van Moneybird maakt het niet veel uit of er ontwikkeld wordt voor de Grape API of voor de GraphQL API. De entities die door Grape gebruikt worden zijn erg vergelijkbaar met de types van GraphQL.

Er is een duidelijk verschil in performance tussen REST en GraphQL. In een groot deel van de use-cases is de performance van de GraphQL API beter dan die van de REST API. Dit verschil wordt echter kleiner naarmate de opgehaalde objecten groter worden. Wanneer alle data van een groot object vereist is, kan de REST API zelfs sneller zijn dan de GraphQL API.

Moneybird heeft op dit moment een bestaande REST API, waarin vrijwel alle functionaliteit al geïmplementeerd is. Er is veel tijd nodig om de GraphQL API tot dezelfde standaard werkend te krijgen. Als Moneybird besluit volledig over te stappen naar een GraphQL API, heeft dit ook gevolgen voor alle bestaande applicaties die gebruik maken van de Moneybird REST API. Het is daarom niet aangeraden om per direct over te stappen naar een GraphQL API.

GraphQL biedt echter wel genoeg voordelen om interessant te zijn voor Moneybird. De beste keuze voor Moneybird is daarom het ondersteunen van zowel de REST API als de GraphQL API.

Literatuurlijst

Brito, G., Mombach, T., & Valente, M. T. (2019). Migrating to GraphQL: A Practical Assessment. *Cornell University*, 140-150. https://doi.org/10.1109/saner.2019.8667986

Fielding, R. T. (2000). Architectural Styles and the Design of Network-Based Software Architectures [Ph.D. Dissertation]. *University of California, Irvine*. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

GraphQL contributors. (2021, oktober). *GraphQL*. GraphQL. Geraadpleegd op 20 mei, 2023, van https://spec.graphql.org/October2021

GraphQL Ruby. (n.d.). GraphQL - Welcome. Geraadpleegd op 28 maart, 2023, from https://graphql-ruby.org/

Lawi, A., Panggabean, B. L. E., & Yoshida, T. (2021, 27 oktober). Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. *Computers*. https://doi.org/10.3390/computers10110138

Masse, M. (2011). *REST API Design Rulebook*. O'Reilly Media.

MDN Contributors. (2023, 3 maart). *HTTP request methods - HTTP | MDN*. MDN Web Docs. Geraadpleegd op 29 maart, 2023, van https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

MDN Contributors. (2023b, 3 maart). *HTTP response status codes - HTTP | MDN*. MDN Web Docs. Geraadpleegd op 30 maart, 2023, van https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Moneybird. (n.d.). *Hoe is Moneybird ontstaan?* Moneybird. Geraadpleegd op 14 februari, 2023, van https://www.moneybird.nl/ontstaan-van-moneybird/

Red Hat. (2020, 8 mei). *What is a REST API?* Red Hat. Geraadpleegd op 29 maart, 2023, van https://www.redhat.com/en/topics/api/what-is-a-rest-api

Wiesbauer, M. (2019). *The benefits of GraphQL for more efficient APIs: A better version of REST [Whitepaper]*. Github. Geraadpleegd op 28 maart, 2023, van https://mwiesbau.github.io/online-cv/assets/4_whitepaper.pdf