HO
GENT

Faculteit Bedrijf en Organisatie

API design voor realtime event afhandeling: GraphQL versus REST

Jan Willem

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

Promotor:
Chantal Teerlinck
Co-promotor:
Hannes Vandevelde

Instelling: Endare

Academiejaar: 2020-2021

Derde examenperiode

Faculteit Bedrijf en Organisatie

API design voor realtime event afhandeling: GraphQL versus REST

Jan Willem

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

Promotor:
Chantal Teerlinck
Co-promotor:
Hannes Vandevelde

Instelling: Endare

Academiejaar: 2020-2021

Derde examenperiode

Woord vooraf

Ik heb deze bachelorproef geschreven naar aanleiding van het afronden van mijn studies Toegepaste Informatica aan de HoGent. Ik heb tijdens mijn opleiding gemerkt dat mijn interesseveld meer aanleunt bij back end gerelateerde zaken dan front end. Dit onderzoek sluit hier dan ook goed op aan.

Het is een onderzoek geweest met veel vallen en opstaan. Het is uiteindelijk niet helemaal geworden wat ik vooropgesteld had, maar ik heb er het beste van proberen maken.

De COVID-periode waarin ik mijn studies heb moeten afronden was voor mij een zware periode. Ondanks dat heb ik steun gekregen van heel wat mensen.

In eerste instantie van mijn stagebedrijf Endare, en in het bijzonder Hannes Vandevelde, voor het aanreiken van het onderwerp van deze bachelorproef. Ik kon altijd bij Hannes terecht, ook al heb ik hier soms iets te weinig gebruik van gemaakt.

Daarnaast wil ik ook mijn ouders bedanken voor alle jaren mij te steunen in alles wat ik wilde ondernemen. Zonder hun steun was dit niet mogelijk geweest.

Ook mijn vriendin was heel belangrijk doorheen deze studies. Ze hielp me doorheen deze zware periode dit zowel op niveau van de opleiding als in privé.

Als laatste wil ik ook mijn medestudenten bedanken en feliciteren om er tot op het einde, en in deze moeilijke omstandigheden, toch een aangename leerperiode van te maken.

Samenvatting

Meer en meer is tijd de belangrijke factor aan het worden in ons leven. Alles moet enorm snel gaan en mensen hebben steeds minder en minder geduld. Dit geldt ook voor het vergaren van informatie. De mens wilt steeds up-to-date zijn met al het nieuws. Wat nieuw is, is ook snel weer oud. Dit geldt zeker voor realtime applicaties zoals nieuwsapplicaties, beursapplicaties of sportweddenschappen. In deze soort applicaties wilt men zo snel mogelijk de nieuwe gegevens doorsturen naar de gebruiker.

In dit onderzoek gaan we kijken tussen REST API en GraphQL welke van beide technologieën hier het best voor kan gebruikt worden. Eerst wordt er algemeen meegegeven wat een API is en hoe de algemene werking is van een REST API en GraphQL. Daarna gaat het iets dieper in op wat realtime wilt zeggen. Tot slot worden verschillende mogelijkheden voor beide technologieën bekeken en paarsgewijs vergeleken.

Uit de vergelijking is duidelijk geworden dat voor de use case van realtime events afhandelen, GraphQL beter zal presteren dan REST. GraphQL is aan een sneltempo aan het groeien, maar REST heeft wel nog steeds een grotere achterban en meer ondersteunende tools.

GraphQL is een nog recente technologie en zal zich in sneltempo blijven ontwikkelen en nieuwe mogelijkheden bieden zoals de live queries. Wanneer deze voldoende matuur zijn is dit zeker een mogelijkheid om verder te onderzoeken.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

"Lost time is never found again." - Benjamin Franklin

"Time isn't the main thing. It's the only thing." - Miles Davis

"All new news is old news happening to new people" - Malcolm Muggeridge

Deze quotes van Benjamin Franklin en Miles Davis illustreren hoe belangrijk tijd voor de mens. Aansluitend hebben we de quote van Malcom Muggeridge, deze duidt erop dat voor de een iets wel nieuws is, terwijl dit voor de ander al oud nieuws kan zijn. Wie is er echter geïnteresseerd in "oud" nieuws? In de hedendaagse wereld vindt de meerderheid van de mens het noodzakelijk om te allen tijde up-to-date te blijven met het allerlaatste nieuws. Gebeurt er iets nieuw, dan moet dit zo snel mogelijk aan de man gebracht worden. De vraag die zich hier stelt is hoe dit het snelst en meest efficiënt kan gebeuren.

1.1 Probleemstelling

Zoals hierboven beschreven hecht de mens veel belang aan zo snel als mogelijk nieuwe informatie vergaren. Als de informatie in bepaalde gevallen niet realtime is, dan kan deze waardeloos worden. Dit is zeker zo voor mensen die nieuwskanalen volgen, mensen die speculeren op de beurs of mensen die sportweddenschappen doen op allerhande sportevenementen. We kunnen dit zelf nog verder uitbreiden naar uurregelingen van trein of bus bijvoorbeeld of bij een rampbestrijding zoals een brand. Ook daar wil je dat de informatie die je krijgt zo realtime mogelijk is.

In deze bachelorproef zal er onderzocht worden tussen twee technologieën, REST en GraphQL, welke van beide hier het best mee om kan. Welke technologie kan de developer het best gebruiken als deze met realtime zaken wilt werken?

1.2 Onderzoeksvraag

1.2.1 Hoofdonderzoeksvraag

Zoals aangegeven hierboven in Sectie 1.1 zal de focus van dit onderzoek liggen op het verschil in realtime eventafhandeling tussen GraphQL en REST.

Voor deze bachelorproef zullen we dus de volgende onderzoeksvraag trachten te beantwoorden:

Welke van beide technologieën (GraphQL of REST API) zal de beste keuze vormen indien er met realtime events gewerkt wordt?

Op deze onderzoeksvraag zal dit onderzoek een antwoord geven in de conclusie (zie 4).

1.2.2 Deelonderzoeksvragen

Deze hoofdonderzoeksvraag kunnen we verder opsplitsen in de volgende deelvragen:

- Welke mogelijkheden zijn er in REST en GraphQL om realtime events af te handelen?
- Hoever staat GraphQL t.o.v. REST? Hoe vlot gaat het implementeren van GraphQL? Zijn er ondersteunende packages?

1.3 Onderzoeksdoelstelling

Het beoogde resultaat is dat er een duidelijke voorkeur gaat naar één van beide technologieën. Hiervoor zal er een vergelijkende studie gemaakt worden aan de hand van een voetbalapplicatie die live updates brengt over wedstrijden. Hierbij zullen we de volgende criteria bekijken om een conclusie te vormen:

- De hoeveelheid requests worden verstuurd
- Performantie
- De hoeveelheid data die doorgestuurd wordt
- Implementatie

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Hierin zullen we GraphQL en REST op verschillende vlakken met elkaar gaan vergelijken.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

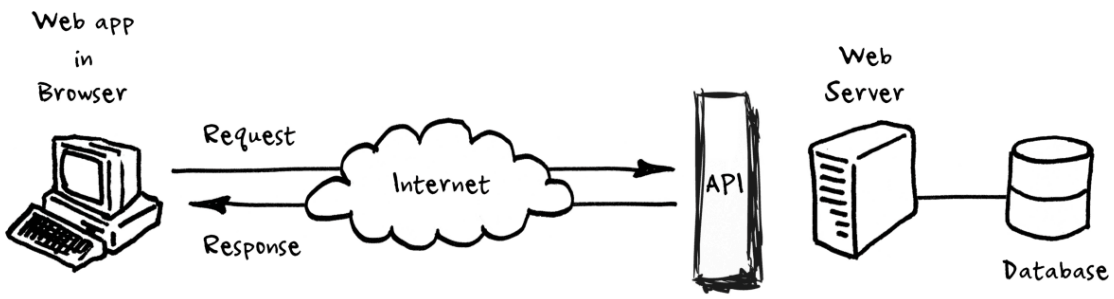
2. Stand van zaken

In dit hoofdstuk zullen we de huidige stand van zaken bespreken. Eerst gaan we na wat een API nu precies is en waarom ze zo belangrijk zijn. Daarna gaan we dieper in op de twee technologieën namelijk REST en GraphQL. Op het einde van dit hoofdstuk bespreken we nog wat het precies betekend om realtime te zijn.

2.1 Wat is een API?

API is de afkorting voor Application Programming Interface, een software interface die twee toepassingen, een client en een server, met elkaar laat communiceren. Een API definieert een set aan verzoeken (requests) die uitgevoerd kunnen worden. Wanneer je een applicatie gebruikt op je mobiele telefoon of computer, maakt deze applicatie verbinding met het internet en worden requests met een aantal gegevens naar een server verstuurd. Deze server gaat op zijn beurt dit verzoek verwerken. Wanneer dit alles gebeurd is, zal de server een antwoord met gegevens terugsturen naar je apparaat. De gegevens die de applicatie ontvangt van de server zal de applicatie ook verwerken. Na verwerking krijgt de eindgebruiker deze gegevens op een verstaanbare manier te zien.

We gaan de werking van een API even voorstellen aan de hand van een voorbeeld uit het normale leven. Stel je voor dat je aan een tafel zit in een café met een drankkaart met verschillende keuzes om uit te bestellen. De barman is het deel van het "systeem"dat je drankje zal klaarmaken. Natuurlijk is er nog een ober nodig om je bestelling aan de barman over te brengen en uiteindelijk je drankje aan tafel te brengen. De ober is in deze situatie dus de API. Hij zal aan tafel komen luisteren welk drankje je wenst en dit aan de barman (het systeem) doorgeven om te laten maken. Wanneer de barman klaar is zal de



Figuur 2.1: API workflow („What is API: Definition, Types, Specifications, Documentation”, 2019)

ober het “antwoord” (je drankje) van de barman terug aan jou bezorgen.

2.2 Waarom bestaan APIs?

Ondertussen is het wel duidelijk wat een API is. De volgende vraag die we ons kunnen stellen waarom zijn ze zo belangrijk? Laten we hiervoor ook starten vanuit een situatie die we allemaal kennen.

Stel je wilt op reis gaan. Hiervoor moet je een reis boeken en hiervoor zijn er verschillende mogelijkheden. De eerste mogelijkheid die opkomt is dat je naar de website van een luchtvaartmaatschappij gaat en daar een vlucht boekt. Op de website van de luchtvaartmaatschappij heb je echter enkel toegang tot de gegevens van vluchten die behoren tot deze maatschappij. Dit is natuurlijk niet interessant, want je wilt zo goedkoop mogelijk reizen en dus verschillende luchtvaartmaatschappijen vergelijken. Om dit te doen is er een tweede mogelijkheid om je reis te boeken, namelijk naar een reisagentschap gaan en daar navragen welke vluchten zij te bieden hebben voor je bestemming. Binnen die reisagentschappen zullen ze verschillende luchtvaartmaatschappijen met elkaar vergelijken om zo tot de beste (goedkoopste) oplossing te komen. Het reisagentschap zal niet de verschillende websites van de luchtvaartmaatschappijen doorzoeken, maar een eigen applicatie gebruiken. Deze applicatie zal toelaten om via de APIs van de verschillende luchtvaartmaatschappijen de informatie op te vragen om de vluchten te vergelijken en te boeken. De API stelt het reisagentschap in staat om informatie uit de database van de luchtvaartmaatschappij te halen om stoelen te boeken, bagage te selecteren en uiteindelijk de prijs de bepalen. De API zal antwoorden op je verzoek om dan de gewenste informatie te tonen.

Vanuit technisch oogpunt zijn APIs zo belangrijk omdat zij het mogelijk maken binnen een applicatie de functies van een ander computerprogramma (of meerdere) te gebruiken. Moderne APIs hebben enkele nieuwe kenmerken aangenomen die zorgen dat ze nuttig en handig in gebruik zijn. Zo houden moderne APIs zich namelijk aan standaarden (meestal HTTP en REST). Verder zijn ze gemakkelijk in gebruik tijdens ontwikkeling en heel toe-

gankelijk waardoor ze makkelijk begrepen door het grote publiek. Doordat ze zich zo houden aan die standaarden hebben ze meer discipline voor veiligheid en beheer en kunnen ze beter gecontroleerd worden op prestaties en omvang. Verder worden ze steeds minder gezien als enkel code, maar worden APIs nu echt als producten op zich benaderd. Dit komt doordat ze beter gedocumenteerd worden en ontwikkeld worden voor een specifieke doelgroep. Het helpt ook dat net als elk ander stuk van geproduceerde software, heeft de moderne API zijn eigen software development lifecycle (SDLC) van ontwerpen, testen, ontwikkelen, beheren en versiebeheer. („What is an API? (Application Programming Interface)", 2021)

2.2.1 Types API

We kunnen APIs categoriseren op 2 mogelijkheden. Ten eerste op bereikbaarheid. Wie mag deze API gebruiken onder welke voorwaarden? Ten tweede op type gebruik. Waarvoor worden ze gebruikt?

API volgens bereikbaarheid

Er zijn 3 verschillende soorted API op basis van bereikbaarheid. Deze zullen we hieronder kort bespreken. („What is API: Definition, Types, Specifications, Documentation", 2019)

Private API Deze API wordt gebruikt binnen de organisatie en is niet bedoeld om bekend te worden gemaakt aan consumenten buiten de organisatie. Interne ontwikkelaars of aannemers kunnen deze APIs gebruiken om de IT-systemen of applicaties van het bedrijf te integreren, nieuwe systemen te bouwen of klantgerichte applicaties te ontwikkelen die gebruik maken van bestaande systemen. Zelfs als de applicatie openbaar beschikbaar is, is de interface zelf alleen beschikbaar voor gebruikers die rechtstreeks met de API-uitgever werken.

Public API Een publieke API is toegankelijk om te worden gebruikt door iedereen die in de API geïnteresseerd is. Het doel is de toepassing gemakkelijk te maken, en zoveel mogelijk gebruikers aan te trekken.

Partner API Wanneer men wenst een parner API te gebruiken moet hiervoor een overeenkomst worden gesloten met de uitgever van de API. Je kan dus stellen dat een partner API gedeeltelijk publiek is. Iedereen mag of kan deze gebruiken, maar enkel wanneer de uitgever hier toestemming voor heeft gegeven.

API volgens gebruik

Hieronder bespreken we verschillende use cases waarin ze gebruikt kunnen worden. („What is API: Definition, Types, Specifications, Documentation", 2019)

Web API Van alle soorten API is dit de meest voorkomende. Web APIs bieden de mogelijkheid om aan communicatie te doen tussen webgebaseerde systemen met clients en servers. Ontwikkelaars kunnen een Web API gebruiken om de functionaliteit van hun applicaties of websites uit te breiden. De Instagram-API bevat bijvoorbeeld tools om Instagram-gegevens van gebruikers (zoals berichten of opmerkingen) aan de website toe te voegen.

Database API Database APIs bieden de mogelijkheid om aan communicatie te doen tussen een applicatie en een database server. De Drupal 7 Database API, bijvoorbeeld, stelt gebruikers in staat om uniforme queries te schrijven voor verschillende types databases.

Operating systems API Operating systems APIs bieden de mogelijkheid om toepassingen de middelen en diensten van een besturingssysteem te laten gebruiken. Zo heb je bijvoorbeelde de Windows API of Linux API en deze hebben elk hun eigen pakket aan functies.

Remote API Remote APIs bieden de mogelijkheid om communiceren via een communicatienetwerk. De resources die geraadpleegd worden door de client bevinden zich buiten het apparaat zelf. Omdat de twee externe applicaties zijn verbonden via een communicatienetwerk (vooral via het internet), voldoen de meeste remote APIs aan vooropgestelde webstandaarden.

2.2.2 Resource

In de API is een resource elk type informatie waaraan een naam kan gegeven worden. Een netwerkinterface of toegangslijst kan bijvoorbeeld worden gebruikt als een resource in de API. Een resource wordt geïdentificeerd met een Uniform Resource Identifier (URI). Resource-representatie is de zichzelf beschrijvende toestand van een resource op een specifiek tijdstip. De server geeft een representatie van de resource aan de client, en de client kan verschillende bewerkingen op de resource uitvoeren. („What is REST”, 2021)

2.2.3 Endpoint

Een API-endpoint is een punt waarop een API-verbinding maakt met een softwareprogramma. APIs werken door requests van informatie van een webtoepassing of webserver te verzenden en een antwoord hierop te ontvangen.

Met andere woorden, API-endpoints zijn de specifieke locatie op het internet waar requests om informatie door het ene programma naar toe worden gestuurd om de daar aanwezige digitale bron op te halen. Endpoints specificeren waar APIs toegang kunnen krijgen tot bronnen en helpen de goede werking van de opgenomen software te garanderen.

Kijken we terug naar het voorbeeld hierboven uit Sectie 2.1 dan zijn de verschillende kelners eigenlijk de API-endpoints.

Softwareprogramma's hebben gewoonlijk meerdere API-endpoints. Bijvoorbeeld, Instagram's endpoints omvatten een die bedrijven en makers in staat stelt om media en profiel interacties te meten; een die hen in staat stelt om opmerkingen en hun antwoorden te modereren; en een derde die hen in staat stelt om hashtagged media te ontdekken.

Om een verzoek door het endpoint te laten verwerken, moet de client een uniform resource locator (URL), een methode, een lijst van headers en een body opgeven. De headers verschaffen meta-informatie over een verzoek en de body bevat de gegevens die door de client naar de server worden gestuurd. Endpoints werken samen met API-methodes. Methodes zijn toegestane requests die kunnen worden gedaan. Deze methodes worden vaak vlak voor het gespecificeerde endpoint in een volledige URL geplaatst. In de tabel 2.1 vind je enkele methoden met hun uitleg. („API endpoint", 2020)

GET	De GET methode vraagt een representatie van de gespecificeerde resource op. Requests met GET mogen alleen gegevens opvragen.
POST	De POST methode wordt gebruikt om een entiteit naar de gespecificeerde resource te sturen, wat vaak een verandering in de status of neveneffecten op de server veroorzaakt.
DELETE	De DELETE methode verwijdert de gespecificeerde resource.
PUT	De PUT methode vervangt alle huidige representaties van de resource door wat er in het verzoek meegegeven wordt.
PATCH	De PATCH methode wordt gebruikt om gedeeltelijke wijzigingen toe te passen op een resource.

Tabel 2.1: Enkele voorbeelden van HTTP methoden („HTTP request methods", 2021)

De URL is de eenvoudige manier voor de client om de server te vertellen met welke bronnen hij wil communiceren.

Enkele voorbeelden van endpoints:

De code die wordt gebruikt om een verzoek te plaatsen voor het opvragen van de universiteiten in België:

```
GET http://universities.hipolabs.com/search?country=Belgium
```

In dit voorbeeld is "GET"de methode, terwijl het endpoint het specifieke deel van het webadres is, genoteerd als /search?country=Belgium".

2.3 REST

Representational State Transfer (REST) is een software-architectuur die wordt gebruikt om standaarden te bieden tussen computersystemen op internet, waardoor het gemakke-

lijker wordt om tussen systemen te communiceren. RESTful systemen worden gekarakteriseerd door het feit dat ze stateless zijn en de zorgen van client en server scheiden. De grootste use case waarvoor REST gebruikt worden is voor het maken van APIs die gebaseerd zijn op de HTTP-standaard.

2.3.1 Architecturale restricties

Zoals elke andere architectuurstijl heeft ook REST zijn eigen restricties waaraan voldaan moet worden wil een web interface RESTful genoemd worden. We zullen deze restricties hieronder uitdiepen. („What is REST”, 2021)

Client-server Om aan deze restrictie te voldoen moet het systeem bestaan uit zowel clients als servers. Er moet een duidelijke scheiding zijn tussen de gebruikersinterface (de client) en de data-opslag (de server).

Staatloos De communicatie tussen de client en de server moet staatloos zijn. Dit wil zeggen dat Elk verzoek van client naar server alle informatie moet bevatten die nodig is om het verzoek te begrijpen. Alles met betrekking tot informatie over de sessie is enkel geweten door de client. De server mag hier niets van context over weten. De status van de sessie wordt daarom volledig op de client bewaard.

Mogelijkheid tot caching Er moet duidelijk gemaakt worden of de gegevens, in het antwoord op de requests naar de server, cachebaar of niet-cachebaar zijn. Als het antwoord in de cache kan worden opgeslagen, heeft de client de mogelijkheid om de gegevens uit het antwoord te hergebruiken voor volgende gelijkwaardige requests.

Gelaagd systeem De gelaagde systeemstijl staat toe dat een architectuur wordt opgebouwd uit hiërarchische lagen door het gedrag van componenten zodanig te beperken dat elke laag enkel informatie weet va de onmiddellijke laag waarmee hij in contact staat.

Uniforme interface REST biedt een uniforme interface tussen componenten. Dit vereenvoudigt de architectuur, aangezien alle componenten dezelfde regels moeten volgen om met elkaar te spreken. Het maakt het ook gemakkelijker om de interacties tussen de verschillende componenten van het systeem te interpreteren. Om een uniforme interface te verkrijgen, zijn meerdere architecturale beperkingen nodig om het gedrag van componenten te sturen. REST wordt gedefinieerd door een taantal interfacebeperkingen. Ten eerste moet iedere resource geïdentificeerd kunnen worden in de request. Ten tweede mag een bron niet te groot zijn en moet er dus gewerkt worden met links naar andere bronnen. Ten derde

Code on demand Code on demand wilt zeggen dat de server uitvoerbare code naar de client kan sturen om zo het aantal functies dat vooraf geïmplementeerd moet worden te verminderen. Dit kan gebeuren aan de hand van applets of scripts. Deze restrictie is optioneel en dus niet verplicht om te voldoen aan RESTful.

2.4 GraphQL

GraphQL is een query en manipulatie taal voor APIs. Het voorziet een runtime waarin queries kunnen uitgevoerd worden op bestaande gegevens die van server naar client gestuurd worden. Een GraphQL query is een string die naar een server wordt gestuurd om te worden geïnterpreteerd en vervuld, die vervolgens JSON terugstuurt naar de client. GraphQL biedt een beschrijving van de gegevens in de API en geeft de gebruikers de mogelijkheid om precies aan te geven welke gegevens ze nodig hebben. Enkel die gegevens zullen worden terug gestuurd en niets meer. Verder kan je via GraphQL ook eenvoudig bronnen gaan nesten om zo via één enkele request verschillende gegevens terug te krijgen. Dit in tegenstelling tot REST waar je verschillende REST-requests moet construeren om hetzelfde op te halen.

Figuur 2.2: GraphQL voorbeeld („GraphQL documentation”, 2020)

GraphQL houdt rekening met de volgende aspecten (Khachatryan, 2018):

Definieer een gegevensvorm De doorgestuurde GraphQL query en de respons hiervan lijken enorm goed op elkaar (zie 2.2). Hierdoor kan je makkelijk voorspellen hoe de gegevens van de response eruit komen te zien. Dit zorgt ermee voor dat GraphQL queries leren schrijven eenvoudig te leren is. Bij het ontwikkelen van een applicatie geeft dit als voordeel dat je makkelijk een query kan schrijven als je weet welke data je nodig hebt.

Hiërarchisch GraphQL maakt gebruik van de relaties van data objecten. Dit in vergelijking met REST waar soms meerdere requests nodig zijn om de alle data te verkrijgen.

Introspectief GraphQL is introspectief. Dit wil zeggen dat de clients het systeem kunnen bevragen op types die het ondersteunt. Een GraphQL server kan worden bevraagd op de types die het ondersteunt. Dit is enorm handig bij het ontwikkelen van nieuwe client applicaties om aan code generatie te doen.

2.4.1 Strongly typed schema

GraphQL maakt gebruik van een strongly typed schema wat wil zeggen dat het schema als het waren een contract is tussen de client en de server. Omdat de vorm van een GraphQL query nauw aansluit bij het resultaat, kun je voorspellen wat de query zal teruggeven zonder veel over de server te weten. Maar het is handig om een exacte beschrijving te hebben van de gegevens die we kunnen opvragen - welke velden kunnen we selecteren? Wat voor soort objecten kunnen ze teruggeven? Welke velden zijn beschikbaar op die sub-objecten? Dat is waar het schema om de hoek komt kijken.

Hier is een kort voorbeeld schema dat twee object types definieert: Boek en Auteur:

```
type Boek {
    titel: String
    auteur: Auteur
}

type Auteur {
    naam: String
    boeken: [Boek]
}
```

Een schema definieert een verzameling van types en de relaties tussen deze types. In het bovenstaande voorbeeldschema heeft elk Boek een auteur, en heeft elke Auteur een lijst van boeken. Doordat deze relaties in zo'n schema gedefinieerd worden, is het voor ontwikkelaars mogelijk om te zien welke velden beschikbaar zijn voor welk type. Het is daarnaast ook mogelijk om een specifieke subset van die gegevens op te vragen met één enkele geneste query. Wanneer een client een query uitvoert en deze binnenkomt op de server dan wordt deze query gevalideerd en uitgevoerd tegen het vooropgestelde GraphQL schema. Verder bevat dit schema ook beschrijvingen van de uitvoerbare GraphQL queries.

2.4.2 Resolver

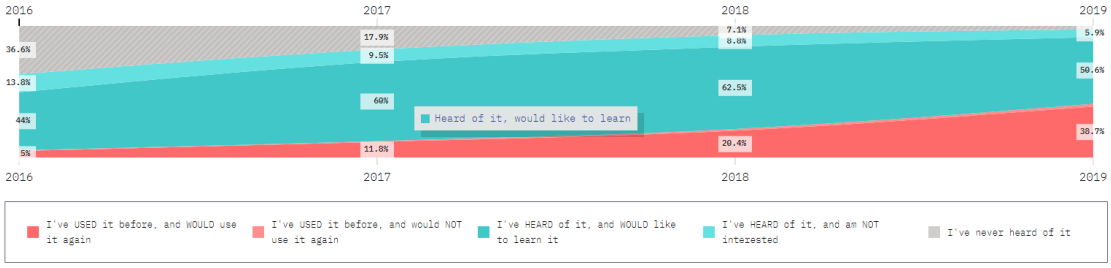
Een resolver is het gedeelte in de GraphQL server die verantwoordelijk is voor het verwerken van de query en het vullen van de gegevens in het schema. Dit door deze op te halen uit een databank of via een api.

2.5 REST vs GraphQL

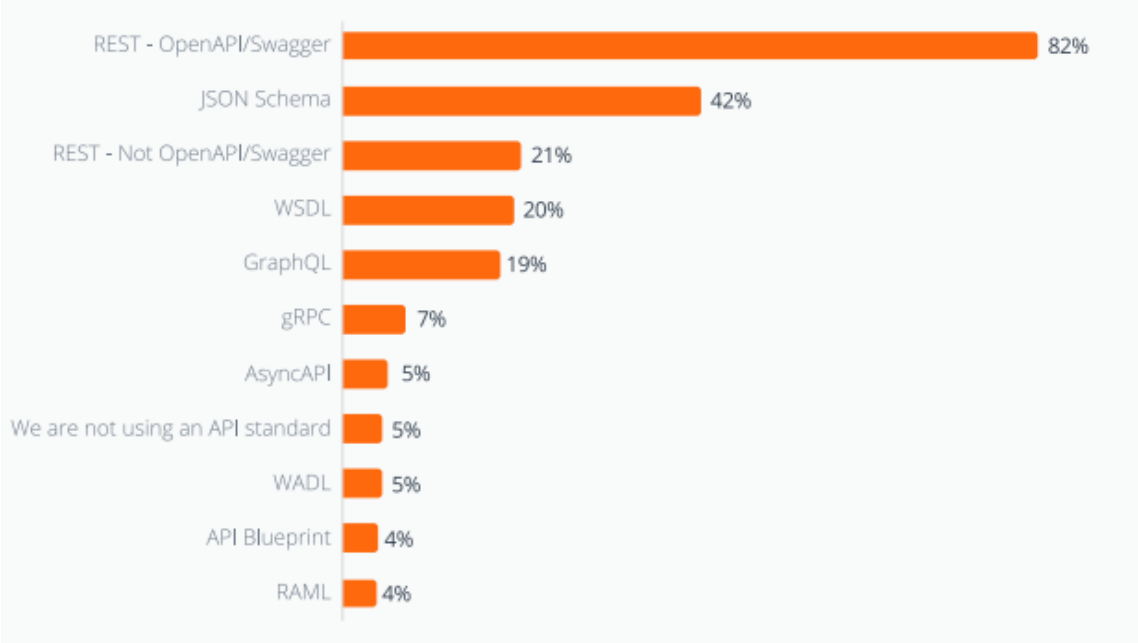
Hieronder gaan we kort in op de populariteit en documentatie van REST en GraphQL. Dit sluiten we af met een concreet praktijkvoorbeeld.

2.5.1 Gebruikers

Hoewel GraphQL nog vrij recent is, heeft de technologie de jongste jaren een enorme groei gehad aan populariteit. Echter doordat REST al veel langer gebruikt wordt zal het niemand verbazen dat deze nog steeds populairder is voor veel bedrijven. Volgens SmartBear (2020) hun State of API Report is het zo dat 82% van de ondervraagde bedrijven REST gebruikt en 19% gebruikt GraphQL. Het percentage van GraphQL stijgt ieder jaar wel fors met, volgens State of JavaScript 2019 Report, een groei in gebruik van 5% in 2016 tot 38.7% in 2019. Het is dus afwachten wat de toekomst zal brengen en of ze het stevig beproefde REST zullen bijbenen of zelf inhalen.



Figuur 2.3: GraphQL in de loop de jaren („GraphQL Experience Over Time”, 2019)



Figuur 2.4: Gebruikte standaarden volgens SmartBear (2020)

2.5.2 Documentatie

In het kader van documentatie en bruikbaarheid is het zo dat GraphQL eenvoud en voorspelbaarheid biedt, terwijl REST een grotere flexibiliteit biedt. Wanneer we echter kijken naar het gedrag van de endpoint, dan zien we het tegenovergestelde. Waar Grap-

hQL enkel JSON als output heeft, heeft REST een breed aanbod aan ondersteunde data-uitwisselingsformaten. Met GraphQL kan er een request gestuurd worden naar de API waarbij je exact de resultaten terug krijgt die je wenst zonder onnodige toevoegingen. Daarentegen varieert het gedrag van REST meestal op basis van de gebruikte HTTP-methode. Dit maakt het onduidelijk voor een API-gebruiker om te weten wat hij kan verwachten bij het aanroepen van een nieuw endpoint. Helaas is de geautomatiseerde documentatie voor beide designs nog steeds onvoldoende. Hoewel het de nodige informatie over endpoints en objecten kan aanbieden, behandelt het zelden onderwerpen als authenticatie of zelfs financiële consequenties.

2.5.3 Limieten van REST

Het kernconcept van REST is dat alles een bron is. REST was vele jaren de grote oplossing, maar ondertussen zijn er al wat problemen opgetreden waarmee het te maken heeft gekregen.

Meerdere requests voor gerelateerde bronnen

De dag van vandaag zijn applicaties veelal data-driven en vereisen grote sets van data die gerelateerde bronnen combineren. Bijvoorbeeld, stel je voor dat je informatie wilt opvragen van een bericht. Tegelijkertijd zou je informatie willen opvragen van de auteur van het bericht (wat een andere entiteit is namelijk een gebruiker). Typisch wordt dit gedaan door twee requests naar de REST API te sturen. De eerste om de opmerking op te halen en de tweede om het gebruiker op te halen. Volgende 2 endpoints zouden hiervoor gebruikt worden:

```
https://voorbeeld.be/api/gebruikers/id
https://voorbeeld.be/api/berichten/id
```

Te veel of te weinig data ophalen

Waar REST ook vaak met problemen kampt is het ophalen van te veel of te weinig data. Als we dit bekijken op het voorbeeld hierboven dan zien we dat we via het endpoint *https://voorbeeld.be/gebruikers/id* een gebruiker ophalen. Deze gebruiker kan bestaan uit verschillende eigenschappen zoals voornaam, familienaam, leeftijd, e-mail, land van geboorte... Wanneer je het desbetreffende endpoint aanspreekt krijg je dus altijd alle gegevens van een gebruiker terug. Er is geen mogelijkheid om enkel een aantal van de eigenschappen terug te krijgen.

2.5.4 Praktijkvoorbeeld

In de volgende sectie zullen we a.d.h.v. praktische voorbeelden de grootste verschillen tussen REST en GraphQL beschrijven.

REST

Stel we hebben een REST API *voorbeeld.be/api* met volgende endpoints:

- /gebruikers
- /berichten
- /opmerkingen

We halen om te beginnen een eerste bericht op via *voorbeeld.be/api/berichten/1* en deze geeft volgend antwoord terug:

```
{
    "gebruikersId": 1,
    "id": 1,
    "titel": "Eerste bericht",
    "inhoud": "Wat is het fijn hier te zijn."
}
```

We zien dat we de titel, inhoud, id en gebruikersId terugkrijgen van het opgevraagde bericht. Echter kunnen we hier niet meteen zien wat de naam van de gebruiker is die dit bericht geplaatst heeft. Indien we we deze gegevens ook wensen dan moeten we een extra request sturen in de vorm van *voorbeeld.be/api/gebruikers/1* en deze geeft dan volgend antwoord terug:

```
{
    "id": 1,
    "naam": "Pieters",
    "voornaam": "Tom",
    "gebruikersnaam": "Tommy",
    "email": "tom.pieters@voorbeeld.be",
    "phone": "1234 56 78 90",
}
```

Zoals eerder aangehaald hebben we dus 2 requests nodig om de informatie van het bericht en de gerelateerde gebruiker op te halen.

GraphQL

GraphQL maakt gebruik van een meer flexibele aanpak. In tegenstelling tot REST gaat GraphQL niet werken met specifieke bronnen, maar zal alles beschouwd worden als een graph [1] en is dus alles met elkaar verbonden. Doordat alles met elkaar verbonden is zorgt dit ervoor dat de query die je verstuurt precies beschrijft wat je als antwoord wenst terug te krijgen. Binnen zo'n query kan je meteen verschillende entiteiten verwerken om zo in

[1] https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

1 request alle gewenste gegevens terug te krijgen. Indien we nu een bericht opvragen en hiervan de titel willen weten in combinatie met de gerelateerde gebruiken waarvan je de naam, voornaam en e-mailadres wenst kan dit met volgende query:

```
{
    bericht(id: 1) {
        titel
        gebruiker {
            naam
            voornaam
            email
        }
    }
}
```

Voor deze query krijgen we volgend resultaat:

```
"data": {
    "bericht": {
        "id": 1,
        "titel": "Eerste bericht",
        "user": {
            "naam": "Pieters",
            "voornaam": "Tom",
            "naam": "tom.pieters@voorbeeld.be"
        }
    }
}
```

We zien dat we nu met 1 request voldoende hebben om alle gegevens op te halen die gevraagd werden. Naast dat dit in 1 request kon i.p.v. 2 is het ook zo dat we geen gegevens ontvangen hebben waar we niet om gevraagd hebben. Dit zorgt ervoor dat er minder gegevensoverdracht plaatsvindt. Indien we de opmerkingen van een bericht willen, kunnen we deze voor GraphQL bovenstaande query nog verder uitbreiden. Indien we hetzelfde willen bekomen voor REST zal er nog een extra request nodig zijn.

2.6 Performantie

Voor het gedeelte in de vergelijking van REST en GraphQL rond het aspect performantie verwijzen we je door naar een bestaand artikel van França en Silva (2020). Dit artikel bespreekt de REST en GraphQL technologieën voor datacommunicatie tussen applicaties. In dit artikel wordt er ook een experiment uitgevoerd, om de prestaties te evalueren van

APIs die REST en GraphQL implementeren. Prototypes van elke API werden geïmplementeerd om metingen uit te voeren om beide technologieën te vergelijken op basis van performantie. Resultaten tonen aan dat GraphQL beter presteerde in de meeste scenario's.

2.7 Realtime

Rekening houdend met externe invloeden, als de gecombineerde reactie- en uitvoeringstijd van het proces minder is dan de maximaal toegestane tijd, wordt het proces in het softwarebesturingssysteem real-time genoemd. Realtime uitvoering is niet per se snel, omdat langzame systemen ook realtime uitvoering kunnen vereisen („Realtime", 2020). Een realtime systeem is een systeem dat in realtime wordt gebruikt, dat wil zeggen dat een reactie wordt verkregen binnen een bepaalde tijdslimiet of het systeem voldoet aan een bepaalde tijdslimiet.

Hieronder volgen enkele van de kenmerken van realtime systemen: („Characteristics of Real-time Systems", 2020)

Tijdsbeperkingen Realtime systemen hebben tijdsbeperkingen. Ieder realtime systeem heeft een bepaald tijdsinterval vooropgesteld. Het systeem moet zorgen dat zijn processen binnen dit tijdsinterval voltooid zijn.

Correctheid Realtime systemen moeten correcte informatie doorsturen. Dit moet gebeuren binnen het vooropgestelde tijdsinterval. Indien het systeem niet binnen het tijdsinterval reageert of foute informatie doorstuurt dan kunnen we niet spreken over een correct antwoord.

Veiligheid Realtime systemen moeten veiligheid bieden. Het systeem moet over een lange termijn zonder problemen kunnen blijven werken. Wanneer er toch een probleem optreedt moet het systeem zich op korte termijn corrigeren. Tijdens een storing mag er ook nooit schade optreden aan de informatie in het systeem.

Concurrency Realtime systemen moeten concurrent zijn. Hiermee bedoelen we dat het systeem verschillende processen tegelijk aan kan. Binnen systemen zijn er altijd tal van processen aan de gang en het systeem moet op ieder van die processen reageren in korte termijn.

Stabiliteit Realtime systemen moeten stabiel zijn en altijd even snel blijven werken. Dit ook wanneer er een grote belasting op het systeem is.

3. Methodologie

Om een antwoord te vinden op de onderzoeksvraag gaan we te werk in 3 fasen. Eerst gaan we voor beide technologieën kijken welke mogelijkheden er zijn om aan realtime verwerking van gegevens te voldoen. Voor iedere mogelijkheid zullen de voor- en nadelen besproken worden. Vervolgens gaan we REST en GraphQL vergelijken op basis van veiligheid wat voor realtime system een belangrijk gegeven is. Als laatste gaan we voor de verschillende mogelijkheden na hoe goed ze kunnen presteren. Het voorbeeld waartegen we de verschillende mogelijkheden zullen toetsen is een voetbalapplicatie die statistieken van een wedstrijd live weergeeft. Zoals bij iedere realtime applicatie is het bij deze voetbalapplicatie van cruciaal belang dat de laatste statistieken weergegeven worden. Dit is een Dit zal voor 1 mogelijkheid praktisch benaderd worden en voor de andere voorbeelden theoretisch. De mogelijkheden zullen vergeleken worden op basis van verschillende factoren zoals:

- het aantal requests die verstuurd worden
- de hoeveelheid gegevens die getransfereerd worden
- moeilijkheid van implementatie

Verder werd er ook rekening gehouden dat de mogelijkheden die besproken worden open source en gratis in gebruik zijn.

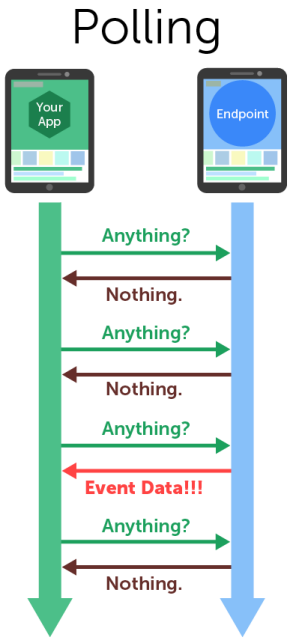
3.1 Realtime mogelijkheden

3.1.1 REST

Voor REST gaan we drie mogelijk oplossingen bespreken om realtime events te verwerken, namelijk polling, webhooks en WebSockets. Voor elk van deze oplossingen zullen we deze bespreken met hun voor- en nadelen.

Polling

Polling is een proces van herhaaldelijk gegevens opvragen door de client aan de server zonder dat de eindgebruiker hiervoor iets moet ondernemen. Het wordt voornamelijk gebruikt om de laatste status van langlopende back end processen te verkrijgen.

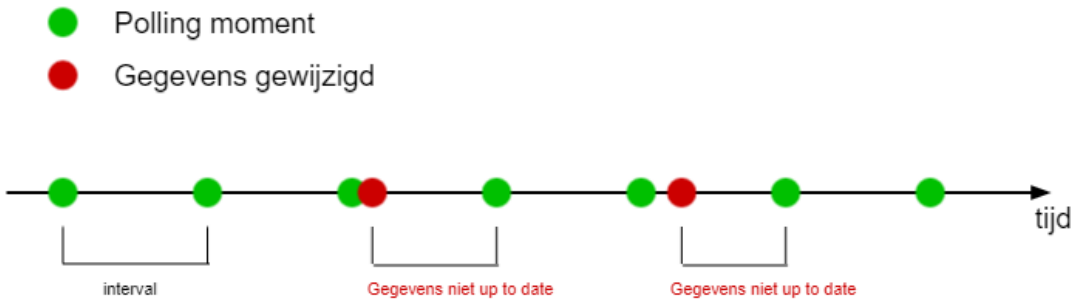


Figuur 3.1: Schematische voorstelling van Polling
Bron: https://blog.cloud-elements.com/webhooks-vs-polling-youre-better-than-this

Polling is een eenvoudige manier om altijd de laatste versie van bepaalde gegevens te verkrijgen. Het interval tussen 2 opvragingen van de client wordt de polling rate genoemd. Deze is door de gebruiker op voorhand zelf te bepalen. Afhankelijk van het interval zal men dus meer (kleiner interval) of minder (groter interval) requests versturen naar de back end. Om zo realtime mogelijk te zijn moet dit interval dus zo klein mogelijk gezet worden.

In principe kan men polling vergelijken met een automatische 'refresh knop'. Hoewel polling zorgt dat je om de zoveel tijd up to date gegevens verkrijgt heeft het ook nadelen. Ten eerste is het geen automatische client update. Indien we een voorbeeld bekijken in 3.2 zien we een eenvoudige voorstelling van de polling rate. Zo zien we dat indien er

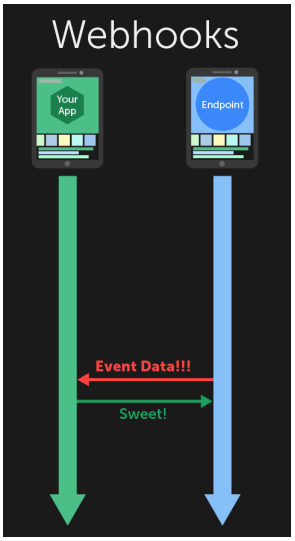
gewijzigde gegevens is net na het polling punt dat er een moment is waarbij de client niet de laatste versie van de gegevens heeft. Dit is natuurlijk ongewenst bij realtime applicaties. Het interval kan natuurlijk zodanig gekozen worden dat dit moment heel kort wordt, maar dit levert dan een ander probleem op namelijk veel requests voor ongewijzigde gegevens. Als het interval zodanig klein wordt ten opzichte van de keren dat de gegevens gewijzigd wordt, zullen er veel requests gebeuren om ongewijzigde gegevens op te halen wat niet efficiënt is.



Figuur 3.2: Schematische voorstelling van de polling rate

Webhooks

Net als bij polling, stellen Webhooks toepassingen in staat om gebruikt te maken van nieuwe gegevens van hun endpoints. In plaats van herhaaldelijk verzoeken voor nieuwe gebeurtenissen te verzenden, wordt er echter een URL voorzien in de endpoint de de toepassing zal opvolgen. Telkens wanneer zich een nieuwe gebeurtenis voordoet in de endpoint-app, worden de gebeurtenisgegevens naar de opgegeven URL verzonden en wordt de app in realtime bijgewerkt.



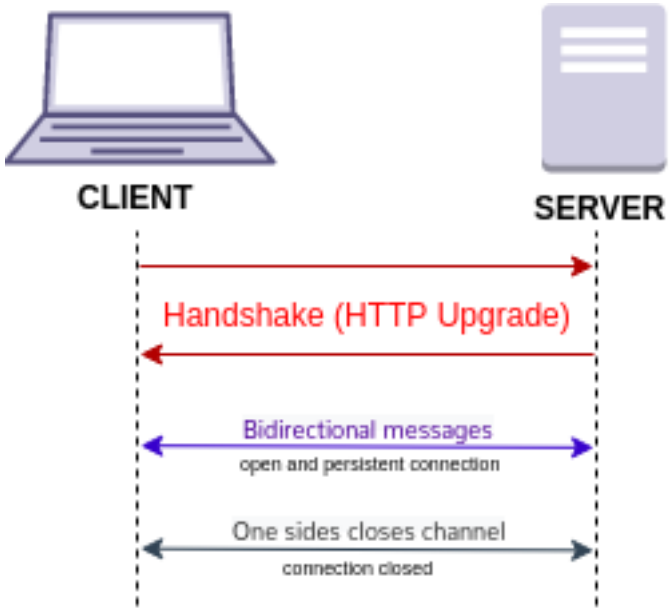
Figuur 3.3: Schematische voorstelling van Webhooks
Bron: https://blog.cloud-elements.com/webhooks-vs-polling-youre-better-than-this

Webhooks kunnen vergeleken worden met notificaties op je apparaat. Stel je bent op een festival en hebt een bandje met een tag om elektronisch te betalen voor een drankje. Dit bandje wordt op voorhand gelinkt aan je telefoonnummer om het bandje te kunnen identificeren. Bij iedere aankoop van een drankje zal het systeem een bericht sturen met hoeveel geld er nog op het bandje staat. Met andere woorden, wanneer er een wijziging gebeurt van de status van het saldo op het bandje zal je hiervan op de hoogte gesteld worden. Dit is het principe van een webhook.

Een Webhook zal dus enkel API-calls uitvoeren wanneer er gebeurtenissen (gewijzigde gegevens) opgetreden zijn wat zorgt voor een meer resource-vriendelijk gebruik. Er worden dus geen uitwisselingen gedaan indien er niets veranderd is. De client zal ook veel sneller zijn gewijzigde gegevens ontvangen doordat de Webhook meteen zal reageren op aanpassingen in plaats van te wachten op een bepaald interval.

WebSocket

Een WebSocket is een netwerkprotocol dat zorgt voor een permanente verbinding tussen een client en een server. Het voorziet de verbinding van full-duplex communicatie. Hieronder verstaan we een verbinding waarbij op hetzelfde moment gegevens kunnen worden uitgewisseld in de twee richtingen: van zender naar ontvanger en van ontvanger naar zender. Deze constante verbinding heeft minder overhead dan polling, maar is wel meer belastend voor het systeem doordat deze verbinding continu wordt open gehouden. Dit geldt zeker als er heel wat clients zijn die willen verbinden met de server. Afhankelijk of de realtime toepassing enkel gegevens verstuurd langs 1 kant kan deze methode minder nuttig bevonden worden.



Figuur 3.4: Schematische voorstelling van WebSocket
Bron: https://en.wikipedia.org/wiki/WebSocket

3.1.2 GraphQL

Hoewel GraphQL veel recenter is heeft ook deze technologie verschillende mogelijkheden om met realtime events te werken. De mogelijkheden die hier aangehaald worden zijn: polling, subscriptions en live queries.

Polling

Ook bij GraphQL is het mogelijk om aan Polling te doen. Het verschil met rest is dat door de werking van GraphQL zullen er minder requests nodig zijn en zullen de respectievelijke antwoorden van de server kleiner zijn (enkel opvragen wat nodig). Voor de rest is de werking hiervan is dezelfde als bij REST (zie 3.1.1).

Subscriptions

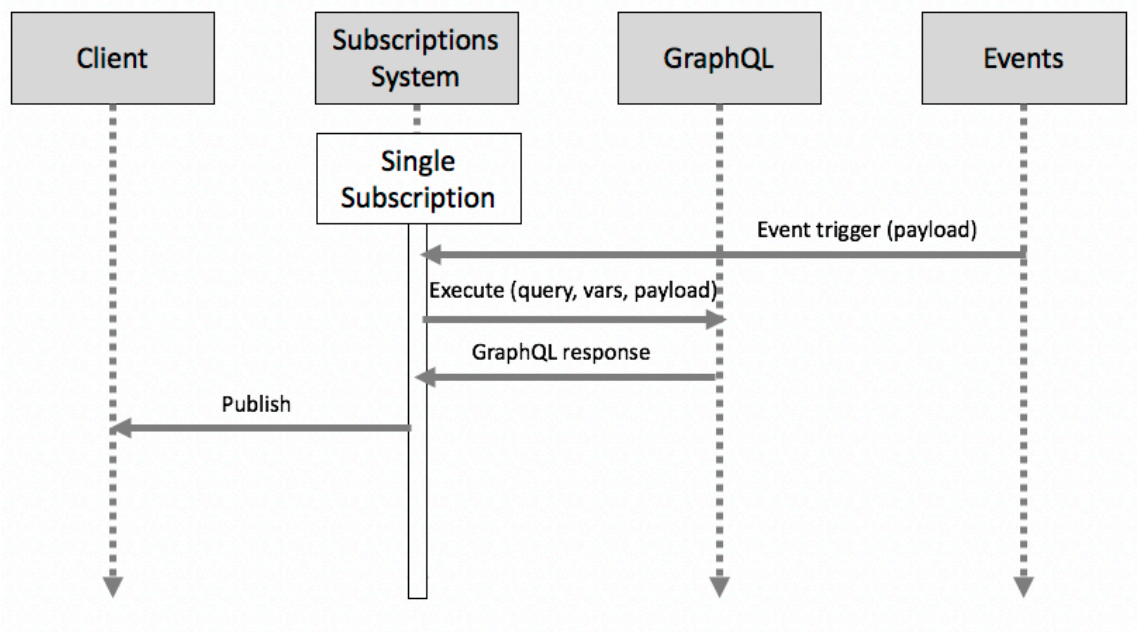
Gelijkaardig met Webhooks bij REST heeft GraphQL Subscriptions. Deze zijn een GraphQL functie die een server in staat stelt om gegevens naar de clients te sturen wanneer een bepaalde gebeurtenis plaatsvindt. Clients kunnen een subscription nemen op de GraphQL server en indien er gegevens gewijzigd wordt dan krijgt deze client de nieuwe gegevens doorgestuurd zoals vooropgesteld in de subscription resolver.

Als we kijken naar onderstaand voorbeeld zien we een subcription gedefinieerd in het GraphQL schema en hoe je deze kan uitvoeren. Als er een doelpunt gemaakt wordt (mutatie op de score gegevens) en je bent geabonneerd dan krijg je een object van het type Match terug. Bij het uitvoeren van de subscription query bepaal je welke velden je terug wenst te krijgen, hier de scores van beide teams.

```
// Definitie van een subscription
type Subscription {
    goalScored(matchId: String!): Match
}

// Uitvoeren van bovenstaande subscription
subscription {
    goalScored {
        score_home,
        score_away
    }
}
```

Ook hier zal dus geen interval zijn waarbij de server wacht om gegevens door te sturen naar de client. De server zal meteen de nieuwe gegevens doorsturen wanneer er een gebeurtenis optreedt.



Figuur 3.5: Schematische voorstelling van Subscriptions
Bron: https://www.telerik.com/blogs/introduction-to-graphql-subscriptions

Live queries

De laatste mogelijkheid die besproken wordt zijn de live queries. Er is niet meteen een eenduidige definitie voor wat een live query is. Vandaag de dag zijn er verschillende oplossingen die men zou kunnen omschrijven als live queries. Één ding hebben de verschillende oplossingen wel gemeen, namelijk dat het hoofddoel is om de status van de client en server synchroon te houden. In een bepaald opzicht zijn de twee methoden die hiervoor besproken zijn ook een vorm van live queries.

Echter gaan we het in deze sectie hebben over een specifiek type live queries. De live queries met de @live directive. Het idee achter de @live directive is dat deze wordt gebruikt om aan te geven dat de client er belang bij heeft dat het resultaat van de uitvoering van de query zo actueel mogelijk blijft. Dit ziet er voor de front end developer al zeer interessant uit want dit is heel weinig werk.

Wanneer we de onderstaande query gaan uitvoeren, krijgt te server te horden dat de client de opgegevens gegevens wilt opvragen en meteen wilt ontvangen. Daar stopt het niet, want de client vertelt hier ook bij aan de server dat wanneer de query een andere resultaat zou terug geven deze ook meteen moet worden doorgestuurd. Met andere woorden, een live query response stream zou moeten lijken op oneindig snel en goedkoop pollen van staande query, terwijl antwoorden die geen nieuwe gegevens bevatten mogen genegeerd worden en niet verstuurd.

```
@live
getAllMatches {
    minute,
```

```
        team_home {
            name,
            score
        },
        team_away {
            name,
            score
        }
    }
```

Als we kijken naar de implementatie voor de front end ontwikkelaar zien we dat deze methode heel weinig aanpassing vraagt. Deze functie is echter minder matuur dan de Subscriptions van GraphQL en heeft dus veel minder ondersteuning en is nog niet betrouwbaar genoeg.

3.2 Veiligheid

Veiligheid is voor heel wat systemen belangrijk. Zoals besproken in 2.7 geldt dit zeker ook voor realtime systemen. Hieronder gaan we kort in op de veiligheid van beide technologieën.

REST heeft verschillende mogelijkheden om aan beveiliging te doen van je API. REST laat bijvoorbeeld toe HTTP-authenticatie te implementeren waardoor gevoelige gegevens in de header van een request verwerkt zit. Ook kan er gebruik gemaakt worden van OAuth.

GraphQL daarentegen heeft ook wel wat mogelijkheden om aan beveiliging te doen van je API, maar deze zijn nog niet zover ontwikkeld als de tegenhanger. Bij GraphQL is het ook zo dat men vaak autorisatie toevoegt op het niveau van de resolvers. Dit komt doordat GraphQL ontwikkelaars niet voldoende ondersteuning krijgen om autorisatie toe te passen. Dit is echter helemaal niet veilig. Wanneer de API complexer wordt en je vergeet autorisatie toe te voegen op één enkele resolver, kan dit geëxploiteerd worden. Daarom wordt nu aangeraden om de autorisatie niet meer te doen op niveau van de resolvers, maar op niveau van business logica. Volgens Noll (2020) heeft GraphQL ook nog last van andere veiligheidsrisico's, namelijk geen goede validatie biedt voor zelf gedefinieerde of aangepaste scalaire gegevenstypen, het kenmerk introspectief van GraphQL kan zorgen voor het openbaar maken van gevoelige interne gegevensmodellen.

Hoewel GraphQL ook inzet op veiligheid kan het voorlopig nog niet mee met wat het mature REST te bieden heeft.

3.3 Uitwerking

Nu we de verschillende mogelijkheden van zowel REST als GraphQL besproken hebben zullen we deze onderverdelen in paren om deze te vergelijken. Voor de praktische insteek zullen we polling van REST tegenover polling van GraphQL vergelijken. Voor de theoretische insteek zal de webhooks van REST tegenover subscriptions van GraphQl geplaatst worden.

3.3.1 Opzet voetbalapplicatie

De voetbalapplicatie bestaat uit 2 onderdelen die elk op hun beurt realtime statistieken bijhouden van voetbalwedstrijden. Als eerste hebben we een algemeen overzicht van verschillende wedstrijden met een beperkt aantal statistieken namelijk: de score van beide ploegen en de minuut waarin de wedstrijd zich bevindt. Vervolgens is er voor een wedstrijd ook een detail overzicht met meerdere statistieken zoals: score, balbezit, doelpogingen, passes...

Figuur 3.6: Algemeen overzicht van de wedstrijden

Figuur 3.7: Een enkele wedstrijd in detail

Als we vervolgens eens kijken naar de tabel B.1, waarin we de gegevens van 50 wedstrijden in de Premier League kunnen bekijken, kunnen we een schatting maken hoeveel gebeurtenissen er plaatsvinden in een wedstrijd. In deze tabel wordt enkel rekening gehouden met de belangrijkste gebeurtenissen in een wedstrijd. Uit deze tabel kunnen we halen dat er gemiddeld een 1081.9 events optreden per wedstrijd. Een belangrijk gegeven is ook dat de meerderheid van de gebeurtenissen gaan naar het aantal passes die gebeuren. Als we rekening houden met een voetbalwedstrijd van 90 minuten of 5400 seconden wilt dit zeggen dat er gemiddeld ongeveer om de 5 seconden een gebeurtenis plaatsvindt. Deze gegevens zullen we verder gebruiken in de volgende vergelijkingen.

3.3.2 Praktische insteek: Polling in REST en GraphQL

Zoals eerder beschreven is polling het continue ophalen van nieuwe gegevens. Hiervoor moet een polling rate bepaald worden om aan te geven om de hoeveel tijd er een nieuwe opvraging moet gebeuren.

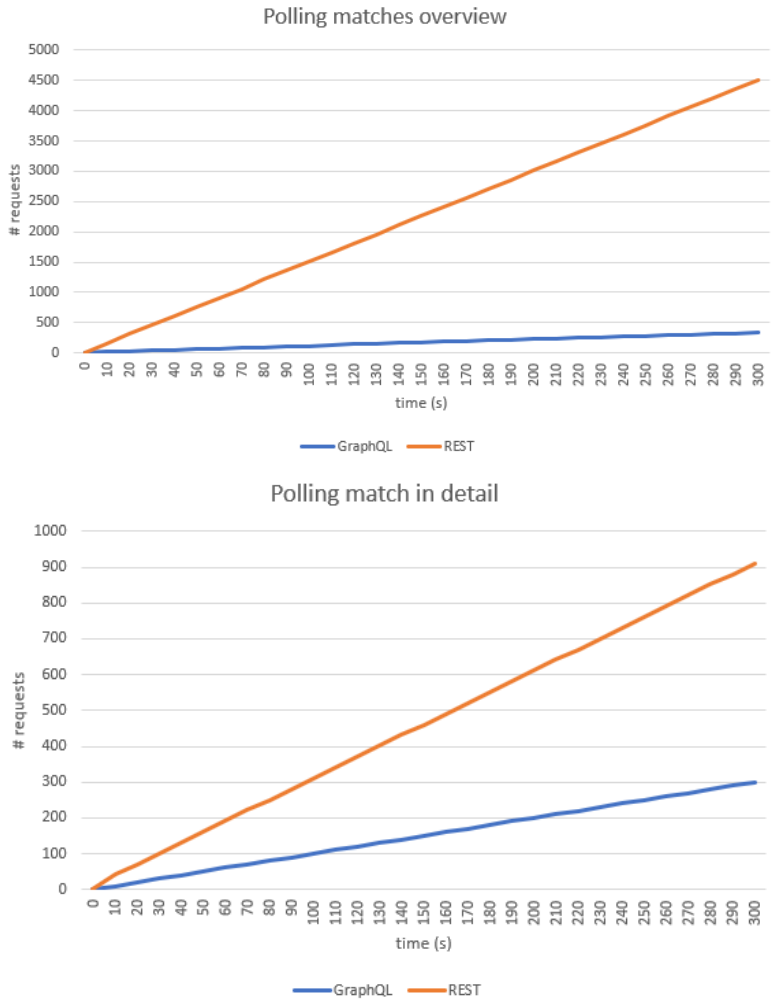
Aantal requests

Het aantal requests is bij polling heel erg afhankelijk van de polling rate. Een belangrijk aspect om deze polling rate te bepalen is om de hoeveel tijd gemiddeld een gebeurtenis plaatsvindt. Zoals hierboven beschreven is dit in een voetbalwedstrijd ongeveer om de 5 seconden. Indien we echter de polling rate op 5 seconden zouden plaatsen, kan het zijn dat de gegevens 5 seconden niet up to date zijn. Dit is in een voetbalwedstrijd heel lang als je inbeeld dat het snelste doelpunt ooit binnen de 2.1 seconden is gescoord. Je zou dus niet meteen weten dat er een doelpunt is. Een andere factor is het aantal requests die men wenst uit te voeren. Hoe lager de polling rate hoe meer requests zal worden uitgevoerd. Voor deze applicatie is de polling rate gekozen op 1 seconde.

Om na te gaan hoeveel requests beide technologieën nodig hebben is er een simulatie gebeurd van een wedstrijd van 5 minuten. De resultaten kan je zien in de grafieken in 3.8 . Je ziet meteen dat GraphQL duidelijk minder requests nodig heeft voor zowel het overzicht van de matchen als voor een match in detail.

Als we even gaan kijken waarom REST zoveel meer requests doet, moeten we kijken naar de wat we terugkrijgen van de requests. REST zal namelijk verschillende requests moeten uitvoeren om alle nodige gegevens te verzamelen. GraphQL zal daarentegen door gebruik te maken van een geneste query alles in 1 request kunnen opvragen.

De requests om de wedstrijden op te vragen geeft een lijst van wedstrijdgegevens terug zoals in B.1. Deze gaat echter maar 1 niveau diep en geeft anders een id terug. Om dus de statistieken van beide ploegen op te vragen zijn nog 2 extra requests nodig. In het overzicht zijn 7 wedstrijden bezig met elk 2 ploegen dus zijn er in totaal 14 extra requests nodig om de statistieken van de ploegen op te vragen. Dus REST heeft voor heel de lijst up te daten 15 requests nodig op alle gegevens te halen. Voor een wedstrijd in detail gaat dit over 3 requests voor REST (1 voor de match zelf en 2 voor de statistieken van beide ploegen). Algemeen is het dus zo dat REST en GraphQL y requests zullen uitvoeren met:



Figuur 3.8: Aantal requests in functie van de tijd bij polling van het overzicht van de wedstrijden (bovenaan) en van een match in detail (onderaan).

voor REST:

$$y = \frac{\#\ requests\ nodig\ voor\ alle\ gegevens}{x} * t$$

voor GraphQL:

$$y = \frac{1}{x} * t$$

Waarbij x de polling rate en t de tijd, beiden in seconden.

We merken op dat na 5 minuten REST 4500 requests heeft gedaan en GraphQL 300. Dit zijn er veel meer dan dat er gebeurtenissen zijn in die 5 minuten. Als er gemiddeld om de 5 seconden een gebeurtenis plaatsvindt dan zou er op 5 minuten 60 gebeurtenissen plaatsvinden. Dit geeft natuurlijk als gevolg dat er heel wat requests overbodig zullen zijn en geen nieuwe gegevens zullen bevatten. Ongeveer 20% van de requests zal maar nieuwe gegevens bevatten. Er is dus veel onnodige belasting van het systeem.

Hoe dan ook volgens França en Silva (2020) is het ook zo dat GraphQL requests een

lagere response time hebben dan REST requests. Dus op het einde hebben we minder requests bij GraphQl en is de response time van een GraphQL request lager dan die van REST. In dit opzicht is GraphQL dus de beter keuze.

Hoeveelheid gegevens

Een ander belangrijke factor is de hoeveelheid gegevens die doorgestuurd wordt naar de clients. De resultaten hiervan kan je terugvinden in 3.9. Doordat polling na ieder interval dezelfde request zal doorsturen krijgen we heel vaak dezelfde gegevens terug en niet enkel wat gewijzigd is. We zien wel dat GraphQL minder gegevens doorstuurt dan REST. Zoals besproken in 2.5 zal GraphQL enkel de gegevens doorsturen die nodig is om de gegevens voor te stellen. REST gaat ook gegevens doorsturen die op dat moment niet van belang zijn door aan overfetching te gaan doen. We hebben bepaald dat het grootste deel van de requests geen nieuwe gegevens versturen. Er wordt dus heel wat onnodige bandbreedte gebruikt om gegevens te versturen die niet nodig is.

Nu lijkt het op het eerste zicht nog om een niet zo groot verschil te gaan. Dit komt doordat in mijn voorbeeld deze API specifiek geschreven is voor deze applicatie. Het aantal velden dat GraphQL niet opvraagt is hierdoor miniem. Stel dat er nog veel meer informatie zou kunnen opgevraagd worden dan zou men hier een groter verschil komen te zien.

Implementatie

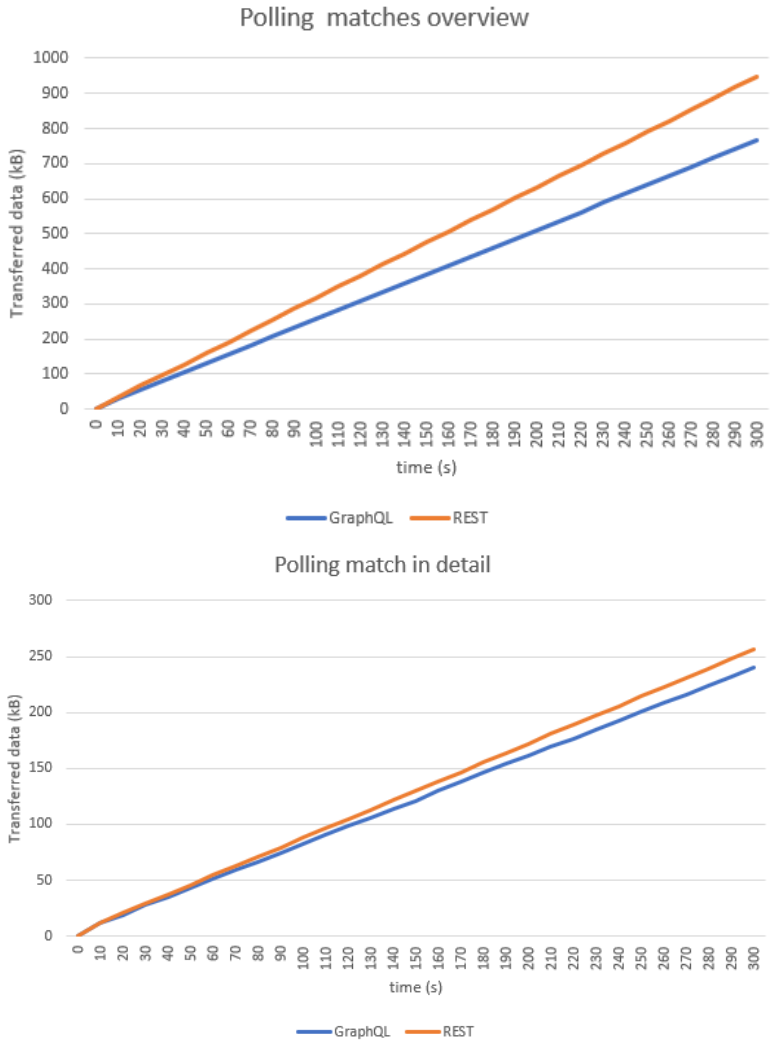
Voor zowel REST als GraphQL is het implementeren van polling heel eenvoudig. Bij REST kan men in TypeScript gewoon volgende toepassen:

```
return timer(1, 1000).pipe(startWith(0),
switchMap(() => this.http.get<JSON[]>
(‘http://jan-mbp.local:3000/matches/${id}‘))
);
```

De timer zal er ervoor zorgen dat om de 1 seconde de HTTP GET request opnieuw wordt uitgevoerd. Bij GraphQL kan men via Apollo Angular een extra argument ’pollInterval’ meegeven met de query die de polling rate bepaalt.

```
this.querySubscription = this.apollo.watchQuery<any>({
    query: GET_MATCHES,
    pollInterval: 1000,
})
```

Op basis van implementatie hebben ze dezelfde moeilijkheidsgraad.



Figuur 3.9: Hoeveelheid gegevens die verstuurd wordt in functie van de tijd bij polling van het overzicht van de wedstrijden (bovenaan) en van een match in detail (onderaan).

3.3.3 Theoretische insteek: REST Webhooks vs GraphQL Subscriptions

REST Webhooks en GraphQL Subscriptions hebben groot deel hetzelfde gedrag. Beide zullen ze optreden wanneer er een verandering in gegevens gebeurt en deze gegevens dan doorsturen naar de client. Dit klinkt al veel beter dan hoe polling te werk gaat. Er zullen minder requests nodig zijn op deze manier en het systeem dus ook minder belasten.

Aantal requests

Doordat er enkel gegevens verstuurd worden wanneer er gegevens gewijzigd worden zal het aantal requests hier lager liggen dan bij polling. Als we dus kijken hoeveel keer gegevens zal worden doorgestuurd dan merken we op dat we enkel afhankelijk zijn van het aantal gebeurtenissen. We hebben bepaald (zie 3.2.1) dat in een wedstrijd gemiddeld 1081.9 events optreden. Dat wil zeggen dat de webhook of subscription gemiddeld 1081.9

keer hun werk zal uitvoeren. Er zal dus ook geen verschil zijn in aantal requests.

Hoeveelheid gegevens

Iedere keer dat er gegevens gewijzigd wordt zal er via de webhook of subscription niet enkel de gewijzigde gegevens doorgestuurd. Er wordt dus nog steeds bandbreedte gebruikt voor gegevens die niet veranderd zijn ook al is deze lager dan bij polling.

Doordat er evenveel requests worden gestuurd bij zowel webhooks als bij subscriptions hangt de hoeveelheid gegevens die verstuurd worden dus van dezelfde zaken af als bij polling. Hier zal REST nog steeds onnodige gegevens versturen die niet worden weergegeven en GraphQL enkel deze deze die wel worden weergegeven. GraphQL zal dus ook in deze vergelijking minder gegevens versturen dan REST.

Implementatie

Voor het opzetten van webhooks in REST is er meer ondersteuning te vinden in bestaande tools dan voor subscriptions voor GraphQL. Subscriptions moeten op moment van schrijven nog helemaal zelf geschreven worden wat de duur van implementatie laat toenemen. Ook is het verder zo dat bestaande Content Management Systemen goed REST integreren, maar GraphQL nog veel minder. Zo is het dat er geen CSM bestaat die subscriptions out of the box laat werken of zelf een eenvoudige manier om deze daaraan toe te voegen.

3.3.4 WebSocket en GraphQL Live queries

WebSocket en Live queries zullen niet verder onderzocht worden. WebSocket is voor deze opzet minder van belang doordat er geen tweerichtingscommunicatie nodig is. Live queries zijn op hun beurt dan nog te recent waardoor er nog teveel op bugs gestoten werd en dus niet matuur genoeg bevonden om aan realtime afhandeling te doen. Een heel interessant aspect bij live queries is dat deze ook kleine patches ondersteunen en dus enkel de gewijzigde gegevens zullen doorsturen in plaats van de vooropgestelde gewenste gegevens. Een onderzoek naar live queries kan op termijn een uitbreiding bieden op dit onderzoek.

3.3.5 Ontwikkelen met GraphQL

GraphQL is op moment van schrijven veel recenter dan REST en dat merk je ook op tijdens het ontwikkelen. Zo is er altijd meer documentatie te vinden rond de ontwikkeling met REST dan de ontwikkeling met GraphQL. Waar rest tal van packages of frameworks heeft om mee te werken is dit bij GraphQL nog veel beperkter. De populariteit van GraphQL is enorm aan het groeien wat als gevolg heeft dat er ook steeds meer packages verschijnen, maar deze zijn nog niet zo uitgebreid of matuur als die van REST. Zoals hiervoor aangehaald is er bijvoorbeeld wel een package die kan helpen bij het ontwikkelen van realtime applicaties met GraphQL, maar deze is nog volop in ontwikkeling. Bij

REST zijn er verschillende frameworks en tal van packages die men kan gebruiken om snel een REST API te ontwikkelen. Bij GraphQL is deze keuze echter beperkter. De volgende Node packages hebben echter wel het ontwikkelen met GraphQL in JavaScript wat makkelijker gemaakt:

- *apollo-server-express*: Express integratie van Apollo Server
- *graphql*: JavaScript referentie implementatie voor GraphQL
- *graphql-compose*: GraphQL scheme builder
- *graphql-compose-mongoose:* Plugin voor graphql-compose om te kunnen werken met types afkomstig van een mongoose model
- *graphql-subscriptions*: Package om te helpen met subscriptions in GraphQL

4. Conclusie

In dit onderzoek hebben we trachten te onderzoeken tussen REST en GraphQL welke van beide technologieën de voorkeur geniet in de case van realtime applicaties.

Beide technologieën hebben elk hun eigen mogelijkheden om realtime events af te handelen. Op basis van de resultaten besproken in het vorige hoofdstuk, gaat de voorkeur uit naar GraphQL. GraphQL heeft voor deze use case heel wat voordelen t.o.v. REST. Deze kwamen duidelijk aan bod tijdens de methodologie. Dat GraphQL alles in één enkele request kan opvragen zonder onnodige gegevens en deze dan ook nog performanter is dan een REST request zorgt ervoor dat GraphQL voor realtime events hier een enorm voordeel heeft.

Als we kijken naar ontwikkeling en veiligheid heeft REST echter ook nog wat voordelen. Doordat GraphQL op moment van schrijven nog wat meer in de kinderschoenen staat dan REST, is er opgemerkt geweest dat er over het algemeen meer ondersteuning te vinden is voor REST dan voor GraphQL. Onder ondersteuning verstaan we: bestaande packages ter ondersteuning van development, tools (bijvoorbeeld CMS) die standaard wel met REST werken, maar niet met GraphQL (subscriptions). Op gebied van veiligheid staat REST ook verder dan GraphQL. Over het algemeen heeft de uitkomst van het onderzoek niet meteen onverwachte zaken aan het licht gebracht.

Als we kijken naar de toekomst heeft GraphQL een mooi vooruitzicht. Het is zo dat GraphQL steeds meer populair wordt wat zal zorgen voor meer ondersteuning op termijn. Een voorbeeld hiervan zijn zoals eerder beschreven de packages voor live queries op GraphQL. Deze zouden in de toekomst nog verder onderzocht kunnen worden of deze voor realtime applicaties een meerwaarde kunnen bieden.

A. Onderzoeksvoorstel

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

A.1 Introductie

GraphQL[1] wordt in de wereld van API[2] gezien als een vernieuwend alternatief voor de alom bekende REST[3], maar beiden hebben nog steeds hun voor- en nadelen. Hoewel het inderdaad zo is dat GraphQL in bepaalde specifieke use cases beter zal presteren dan REST, zal REST nog altijd de voorkeur genieten in andere use cases. Wat met de use case voor realtime API's? Onder de term realtime verstaan we dat de interacties worden beschouwd als onmiddellijk. Welke technologie zal in deze context beter presteren? Voor het grootste deel is het schrijven in een API 'eenvoudig': je bundelt alle gegevens en stuurt deze door naar een endpoint waardoor data gecreëerd of gewijzigd wordt. Een endpoint is één kant van het communicatiekanaal. Wanneer een API een wisselwerking heeft met een ander systeem, worden de contactpunten van deze communicatie beschouwd als endpoints. Een probleem kan ontstaan bij het lezen van data en in het bijzonder het lezen van gewijzigde data. De meeste API's zijn 'request-based'. Dit betekent dat er enkel data doorgestuurd wordt wanneer deze gevraagd wordt door de client. Helaas doen de meeste REST API's erg weinig om de ontwikkelaar te helpen met dit probleem. REST kan verschillende endpoints hebben die data terugsturen, maar het betekent dat de ontwikkelaar zelf de data moet filteren en sorteren. Het is een hoop overhead om nieuwe data te vinden

[1] www.graphql.org
[2] en.wikipedia.org/wiki/API
[3] en.wikipedia.org/wiki/Representational_state_transfer

t.o.v. de vorige request. Voor realtime events is dit niet ideaal om zo te werk te gaan, want je wilt de nieuwe data zo snel als mogelijk. De volgende zaken zullen onderzocht worden:

- Welke van beide technologieën (GraphQL of REST API) zal de beste keuze vormen indien er met realtime events gewerkt wordt?
- Welke mogelijkheden zijn er in REST en GraphQL om realtime events af te handelen?
- Hoever staat GraphQL t.o.v. REST? Hoe vlot gaat het implementeren van GraphQL? Zijn er ondersteunende packages?

A.2 State-of-the-art

GraphQL is een meer recente technologie, maar volgt dezelfde architecturale beperkingen als REST API's. Bij GraphQL worden de gegevens georganiseerd in een graaf met behulp van een interface. Objecten worden in de graaf gesymboliseerd door middel van knooppunten (nodes), gedefinieerd met behulp van het GraphQL-schema, en de relatie tussen de verschillende knooppunten wordt gesymboliseerd door bogen (edges). Waar REST een software-architectuurstijl is, is GraphQL een taal om query's uit te voeren op een API om zo gegevens op te halen. GraphQL geeft meer controle aan de gebruiker van de API, terwijl REST de controle legt bij de ontwerper ervan. Hieronder wordt een korte vergelijking gemaakt op basis van verschillende factoren zoals: populariteit, bruikbaarheid, performantie en veiligheid.

A.2.1 Populariteit

Hoewel GraphQL nog vrij recent is, heeft de technologie de jongste jaren een enorme groei gehad aan populariteit. Echter doordat REST al veel langer gebruikt wordt zal het niemand verbazen dat deze nog steeds populairder is voor veel bedrijven. Volgens SmartBear (2020) hun State of API Report is het zo dat 82% van de ondervraagde bedrijven REST gebruikt en 19% gebruikt GraphQL. Het percentage van GraphQL stijgt ieder jaar wel fors met, volgens State of JavaScript 2019 Report, een groei in gebruik van 5% in 2016 tot 38.7% in 2019 („GraphQL Experience Over Time", 2019). Het is dus afwachten wat de toekomst zal brengen en of ze het stevig beproefde REST zullen bijbenen of zelf inhalen.

A.2.2 Documentatie en bruikbaarheid

In het kader van documentatie en bruikbaarheid is het zo dat GraphQL eenvoud en voorspelbaarheid biedt, terwijl REST een grotere flexibiliteit biedt. Wanneer we echter kijken naar het gedrag van de endpoint, dan zien we het tegenovergestelde. Waar GraphQL enkel JSON als output heeft, heeft REST een breed aanbod aan ondersteunde datauitwisselingsformaten. Met GraphQL kan er een request gestuurd worden naar de API

waarbij je exact de resultaten terug krijgt die je wenst zonder onnodige toevoegingen. Daarentegen varieert het gedrag van REST meestal op basis van de gebruikte HTTP-methode. Dit maakt het onduidelijk voor een API-gebruiker om te weten wat hij kan verwachten bij het aanroepen van een nieuw endpoint. Helaas is de geautomatiseerde documentatie voor beide designs nog steeds onvoldoende. Hoewel het de nodige informatie over endpoints en objecten kan aanbieden, behandelt het zelden onderwerpen als authenticatie of zelfs financiële consequenties.

A.2.3 Performantie

Voor GraphQL is performantie de topprioriteit, terwijl REST meer gericht is op het behoud van de betrouwbaarheid van de services als hoofddoel. Zelfs als een REST API slechts een eenvoudig gedeelte retourneert, brengt het nog steeds van meer gegevens over, terwijl GraphQL altijd streeft naar het kleinst mogelijke verzoek waarbij je de velden kiest die je wilt opvragen. Bovendien is het bij GraphQL mogelijk om meerdere entiteiten in één request op te vragen. Alhoewel GraphQL meer voordelen lijkt te hebben op gebied van prestaties, zijn er gevallen waar REST API's een betere mogelijkheid is. Bijvoorbeeld, in gevallen waar caching gewenst is om API-aanroepen te versnellen, kunnen REST-API's beter presteren. REST-API's maken gebruik van het ingebouwde HTTP-cachingmechanisme om de gecachte responses sneller terug te sturen. Hoewel GraphQL terug moet naar de bron om de benodigde gegevens op te halen, heeft REST de mogelijkheid om deze snel uit de browser of mobiele cache te halen. Inderdaad, GraphQL heeft ook enkele opties voor caching, maar deze hebben het niveau van REST nog niet bereikt. Er is dus geen duidelijke winnaar op gebied van performantie. Hoewel GraphQL het aantal verzoeken dat een gemiddelde applicatie nodig heeft vereenvoudigt, is REST de beste in de robuustheid van caching mogelijkheden. Wanneer we dit verder bekijken voor de gekozen use case, is het wel zo dat caching hier niet van toepassing zal zijn, want er zullen enkel nieuwe elementen doorgestuurd worden.

A.2.4 Veiligheid

In termen van GraphQL vs. REST security lijkt REST hier de bovenhand te nemen. REST biedt verschillende inherente manieren om de beveiliging van uw API's af te dwingen. Hoewel GraphQL ook enkele maatregelen biedt om de veiligheid van uw API's te garanderen, zijn deze niet zo ver ontwikkeld als die van REST. Hoewel GraphQL bijvoorbeeld helpt bij het integreren van datavalidatie, blijven gebruikers zelf verantwoordelijk voor het toepassen van authenticatie en autorisatiemaatregelen. Dit leidt vaak tot onvoorspelbare autorisatiecontroles, die de veiligheid van op GraphQL gebaseerde applicaties in gevaar kunnen brengen. Bij REST kan API-beveiliging gegarandeerd worden door verschillende API-authenticatiemethoden te implementeren, zoals via HTTP-authenticatie.

A.3 Methodologie

Voor het onderzoek zal een voetbalapplicatie gemaakt worden die in realtime de statistieken van een wedstrijd zal weergeven. Een aantal voorbeelden van deze statistieken zijn: goals, passes, overtredingen, hoekschoppen, wissels ... Aan de hand hiervan zal er verder geëxperimenteerd worden met REST en GraphQL om zo na te gaan hoe ze omgaan met realtime events. Hieruit zal ook moeten blijken hoe schaalbaar en performant beide technologieën zijn voor deze use case. De applicatie zal bestaan uit een front end geschreven in Angular die gebruik maakt van de backend via REST en GraphQL. Voor de back end wordt gebruik gemaakt van een open-source headless CMS (Content Management System) genaamd Strapi. De data die gebruikt zal worden zal zelf tot stand gebracht worden en dus fictief zijn.

A.4 Verwachte resultaten

Hoewel het niet eenvoudig is om op voorhand te bepalen wat het resultaat zal zijn, wordt verwacht dat voor deze use case GraphQL over het algemeen beter zal presteren dan een REST API. Op gebied van snelheid zal GraphQL beter presteren doordat GraphQL minder onnodige gegevens zal doorsturen. Echter is het ook zo dat REST API op bepaalde vlakken de bovenhand kan nemen. Dit zal zeker het geval zijn indien blijkt dat caching toch een bepaalde rol zou blijken te spelen. Uiteindelijk zullen beide hun sterke punten kunnen uitgespeeld worden wanneer beide simultaan gebruikt worden.

A.5 Verwachte conclusies

Uit de verwachte resultaten kunnen we nu de verwachte conclusie afleiden. De verwachte conclusie is dat, voor de use case rond realtime events afhandelen, de keuze zal vallen op GraphQL. Echter wordt er niet verwacht dat GraphQL de vervanger wordt voor REST API's, maar dat deze beide naast elkaar kunnen verder gaan en in de toekomst de go-to zullen blijven/worden voor API's.

B. Bijlagen

```
{
    "_id": "6123c7846452c24723a18d8a",
    "competition": {
        "_id": "60c60453e0a04a4ae0f4841d",
        "name": "Jupiler Pro League",
        "founded": 1895,
        "website": "https://www.proleague.be/nl/jpl",
        "confederation": "UEFA",
        "__v": 0
    },
    "team_home": {
        "_id": "6123c2e36452c24723a18d6d",
        "name": "KV Kortrijk",
        "founded": 1901,
        "stadium": "Guldensporen Stadion",
        "manager": "Luka Elsner",
        "website": "https://www.kvk.be/",
        "city": "Kortrijk",
        "__v": 0
    },
    "team_away": {
        "_id": "6123c3156452c24723a18d6e",
        "name": "KV Oostende",
        "founded": 1904,
        "stadium": "Versluys Arena",
        "manager": "Alexander Blessin",
        "website": "https://www.kvo.be/",
        "city": "Ostend",
        "__v": 0
    },
    "match_stats": {
        "_id": "6123c5c66452c24723a18d82",
        "minute": 0,
        "team_stats_home": "6123c5326452c24723a18d77",
        "team_stats_away": "6123c5336452c24723a18d78",
        "__v": 0
    },
    "__v": 0
}
```


Figuur B.1: Inhoud van het antwoord van een enkele match bij verzoek REST API

Tabel B.1: Gegevens over 50 wedstrijden in de Premier League seizoen 2017/2018

Match	Goals	Goal attempts	Free kicks	Corner kicks	Offside	Passes	Fouls	Tackles	Minutes	Total events
1	7	33	29	13	8	895	21	41	90	1137
2	6	23	26	6	4	872	22	36	90	1085
3	0	33	24	13	1	886	23	22	90	1092
4	1	24	20	10	2	849	18	35	90	1049
5	1	18	31	13	8	783	23	31	90	998
6	3	22	28	21	2	694	26	51	90	937
7	5	29	30	13	3	837	28	22	90	1057
8	2	20	22	13	7	980	15	21	90	1170
9	2	24	20	12	3	945	16	32	90	1144
10	4	29	31	12	5	889	26	35	90	1121
11	4	23	24	8	1	1005	23	37	90	1215
12	5	30	28	9	1	851	28	22	90	1064
13	1	25	31	6	6	987	25	45	90	1216
14	2	19	21	8	3	950	18	19	90	1130
15	1	28	26	10	4	697	22	18	90	896
16	2	25	27	13	8	767	20	31	90	983
17	1	28	22	11	6	944	17	37	90	1156
18	1	20	32	10	9	874	23	39	90	1098
19	3	27	34	17	0	872	35	47	90	1125
20	2	26	19	8	3	862	18	31	90	1059
21	3	28	30	7	3	860	27	31	90	1079
22	2	23	18	2	3	968	15	36	90	1157
23	0	24	30	14	5	770	25	21	90	979
24	3	25	30	12	2	744	28	31	90	965
25	0	22	23	9	3	880	20	33	90	1080

26	2	33	19	12	5	962	15	31	90	1169
27	2	26	21	14	2	742	19	31	90	947
28	2	28	19	9	0	1021	19	31	90	1219
29	2	41	25	17	7	858	18	22	90	1080
30	4	26	22	7	8	1046	15	34	90	1252
31	5	20	25	11	7	1102	19	29	90	1308
32	2	21	22	11	3	807	19	35	90	1010
33	3	24	22	11	4	868	19	38	90	1079
34	3	27	35	10	7	1022	28	26	90	1248
35	4	23	15	8	2	909	13	25	90	1089
36	3	24	28	13	4	984	24	27	90	1197
37	4	28	24	16	5	896	20	28	90	1111
38	1	27	28	16	9	704	21	30	90	926
39	1	26	27	12	1	854	26	23	90	1060
40	2	29	26	13	1	737	25	39	90	962
41	3	24	19	10	2	1041	16	29	90	1234
42	1	27	28	10	1	892	27	29	90	1105
43	0	15	28	5	5	416	22	6	90	587
44	6	35	15	14	2	949	14	26	90	1151
45	3	27	26	12	3	328	24	4	90	517
46	2	40	22	14	6	953	16	17	90	1160
47	2	25	15	12	5	819	12	23	90	1003
48	0	30	20	12	0	1014	20	31	90	1217
49	0	24	31	6	6	1045	26	31	90	1259
50	4	23	33	5	5	986	29	36	90	1211
Gemiddelde	2,44	26,02	25,02	11	4	872,32	21,36	29,7	90	1081,86
Mediaan	2	25,5	25,5	11,5	3,5	883	21	31	90	1095

Standaard-afwijking	1,69	4,97	5,11	3,53	2,52	143,55	4,94	8,91	0	146,28

Bibliografie

API endpoint. (2020). *TechTarget*. https://searchapparchitecture.techtarget.com/definition/API-endpoint

The Argument for Real-Time REST APIs. (2013). *ProgrammableWeb*. https://www.programmableweb.com/news/argument-real-time-rest-apis/2013/03/05

Bush, T. (2020). GraphQL vs REST: The Consumers Preference. *Nordic APIS*. https://nordicapis.com/graphql-vs-rest-the-consumers-preference/

Characteristics of Real-time Systems. (2020). *GeeksforGeeks*. https://www.geeksforgeeks.org/characteristics-of-real-time-systems/

Charboneau, T. (2020). Breaking Down SmartBears 2020 State of API Report. *Nordic APIS*. https://nordicapis.com/breaking-down-smartbears-2020-state-of-api-report/

Describe characteristics of REST-based APIs. (2021). *Fast Reroute*. https://fastreroute.com/describe-characteristics-of-rest-based-apis/

Devathon. (2019). GraphQL vs REST in 2020: A Detailed Comparison. *Devathon*. https://devathon.com/blog/graphql-vs-or-rest/

Eschweiler, S. (2018). REST vs. GraphQL. *CodingTheSmartWay.com*. https://codingthesmartway.com/rest-vs-graphql/

França, M. D. C. & Silva, E. D. Performance Evaluation of REST and GraphQL APIs Searching Nested Objects. In: *Computer on the Beach*. 2020.

Gilling, D. (2019). REST vs GraphQL APIs, the Good, the Bad, the Ugly. *Moesif*. https://www.moesif.com/blog/technical/graphql/REST-vs-GraphQL-APIs-the-good-the-bad-the-ugly/

GraphQL documentation. (2020). https://graphql.org/

GraphQL Experience Over Time. (2019). *State of JavaScript*. https://2019.stateofjs.com/data-layer/graphql/

HTTP request methods. (2021). *MDN Web Docs*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

The Importance of APIs for Business. (2021). *GlowTouch*. https://www.glowtouch.com/importance-apis-business/

Khachatryan, G. (2018). What is GraphQL? *DevGorilla*. https://medium.com/devgorilla/what-is-graphql-f0902a959e4

Nemec, Z. (2019). REST vs. GraphQL: A Critical Review. *Good API*. https://goodapi.co/blog/rest-vs-graphql

Noll, A. (2020). *The 5 Most Common GraphQL Security Vulnerabilities*. https://carvesystems.com/news/the-5-most-common-graphql-security-vulnerabilities/

Opidi, A. (2020). GraphQL vs. REST: A Comprehensive Comparison. *Rakuten*. https://blog.api.rakuten.net/graphql-vs-rest/

Poniatowicz, T. (2019). GraphQL vs REST: Performance. *GraphQL Editor*. https://blog.graphqleditor.com/graphql-vs-rest-performance/

Realtime. (2020). *Wikipedia*. https://nl.wikipedia.org/wiki/Realtime

Resolvers - How Apollo Server processes GraphQL operations. (2021, maart 9). https://www.apollographql.com/docs/apollo-server/data/resolvers/

Russel, D. (2019). GraphQL vs REST: What You Need to Know. *Rubrik*. https://www.rubrik.com/en/blog/technology/19/11/graphql-vs-rest-apis

Schema basics. (2021, maart 9). https://www.apollographql.com/docs/apollo-server/schema/schema/

SmartBear. (2020). State of API Report.

Stubailo, S. (2017). GraphQL vs. REST. *Apollo*. https://www.apollographql.com/blog/graphql-vs-rest-5d425123e34b/

What is an API? (Application Programming Interface). (2021). *MuleSoft*. https://www.mulesoft.com/resources/api/what-is-an-api

What is API: Definition, Types, Specifications, Documentation. (2019). *AltexSoft*.

What is REST. (2021). *REST API Tutorial*. https://restfulapi.net/

What is REST? (2021). *Codecademy*. https://www.codecademy.com/articles/what-is-rest