**DT-DBS**

17

**Onderzoek GraphQL & MongoDB**

**Een onderzoek naar de toepassing van GraphQL met een MongoDB**

**Door Robin van Dijk, SN:[sn hier]**

Inhoud

[Samenvatting 3](#_Toc484871883)

[Inleiding en probleemstelling 4](#_Toc484871884)

[Opdrachtformulering 5](#_Toc484871885)

[Onderzoeksvraag en –aanpak 6](#_Toc484871886)

[Onderzoeksresultaten 8](#_Toc484871887)

[Deelvraag 1: 8](#_Toc484871888)

[Deelvraag 2: 10](#_Toc484871889)

[Deelvraag 3: 13](#_Toc484871890)

[Deelvraag 4: 15](#_Toc484871891)

[Conclusies en evaluatie 16](#_Toc484871892)

[Bijlagen 17](#_Toc484871893)

[A: Express minimale implementatie 17](#_Toc484871894)

[Literatuurlijst 18](#_Toc484871896)

# Samenvatting

//

# Inleiding en probleemstelling

**Inleiding:**

Dit document betreft een onderzoek naar het gebruik van GraphQL, met als achterliggende database een MongoDB. Dit omdat de werkplek van de auteur momenteel gebruikt maakt van deze database implementatie voor de opslag van gegevens. In dit document wordt onder andere kort inzicht gegeven in de achtergrondsituatie bij het bedrijf, en waarom dit onderzoek dus relevant is.

Ook worden in dit document de hoofdvragen uitgelijnd samen met de gebruikte onderzoeksmethodiek. Door het beantwoorden van de hoofd en- deelvragen, waarbij ook een voorbeeld project zal worden ontwikkeld met een minimale implementatie van GraphQL, zal duidelijk worden of het nuttig is om GraphQL te gaan gebruiken bij de ontwikkeling van nieuwe of bestaande producten.

**Achtergrond:**

Bij het bedrijf waar de auteur van dit document momenteel werkzaam is, wordt gewerkt aan prototypes van producten die tot de categorie internet-of-things behoren. Het idee is dat deze producten informatie over het gedrag van de gebruiker inwinnen, zodat deze informatie kan worden ingezet om de leefstijl van gebruikers te bevorderen.

De gebruikers zijn in dit geval patienten die bijvoorbeeld een hartaanval hebben gehad, en hierdoor minder zout mogen consumeren. Een een product waar aan gewerkt wordt bij het bedrijf is bijvoorbeeld een “smart-spatel”, die het zoutgehalte van de voeding van de gebruiker meet. Deze gevens worden vervolgens doorgestuurd naar een centrale server, die deze opslaat voor latere analyse.

**Probleemstelling:**

Het bedrijf werkt vooral met nieuwe technologie, de hierbovengenoemde “smart-spatel” bijvoorbeeld, verbind met een webapplicatie die gebruik maakt van het Meteor- framework (<https://www.meteor.com/> ). Dit framework maakt gebruik van een MongoDB, en stelt de client in staat om direct methodes aan te roepen op de server als het ware.

Er zijn echter ook producten die data versturen of opvragen bij een Express server (<https://expressjs.com/> ), die tevens verbind met een MongoDB. De inrichting van een express server maakt het mogelijk om een REST-api op te zetten

Er zit dus een verschil in de manieren waarop de server(s) (en dus ook uiteindelijk de database(s)) wordt aangeroepen door de client(s). Ook is de tijd die nodig is voor het bijhouden van documentatie voor de verschillende implementaties in verschillende formaten aanzienlijk.

# Opdrachtformulering

De doelstelling van dit onderzoek is het bepalen of GraphQL een waardig alternatief is ten opzichte van de huidige vormgeving van de API’s en de manier waarop de MongoDB erachter gequeried wordt. Het kost op het moment van schrijven bijvoorbeeld aardig wat werk om een nieuwe methode aan de API toe te voegen, met de juiste achterliggende query. Ook is het definieren van een schema voor deze database momenteel nogal omslachtig, en wordt er, behalve voor verificatie van de invoer en updates, weinig gebruik gemaakt van dit schema.

De opvolger van het framework wat momenteel gebruikt wordt (Meteor), maakt gebruik van GraphQL. Tevens is het mogelijk om ook op Express GraphQL toe te passen. Dit zou betekenen dat er een “common-language” is tussen de database en de webserver van de verschillende implementaties. Dit zou het ook makkelijker maken om code uit te wisselen tussen de Express server en de Meteor server.

Het is namelijk zo, omdat er aan prototypes wordt gewerkt, dat de wijziging van de requirements van een prodcut een plotselinge wijziging aan kunnen brengen aan de database structuur. Na het aanpassen hiervan dient bijvoorbeeld ook de documentatie te worden bijgewerkt. Met GraphQL is het bijvoorbeeld ook mogelijk om de documentatie mee te nemen in het schema.

Bovengenoemde punten zouden erg veel voordeel op kunnen leveren in de praktijk. Het is echter de vraag of wel alles goud is wat blinkt, want als deze tussenlaag hoedanig veel extra werk zou opleveren dat deze voordelen verbleken, zou het mogelijk geen geschikt alternatief kunnen zijn.HHet

# Onderzoeksvraag en –aanpak

**Onderzoeksvragen:**

Naar aanleiding van de doelstelling van het onderzoek is het mogelijk om de volgende hoofdvraag te formuleren:

“Is GraphQL een geschikt alternatief voor de inrichting van de API’s van de webapplicatie ten opzichte van REST API’s?”

Om tot een geschikt antwoord voor de hoofdvraag te komen, is het belangrijk dat eerst de volgende deelvragen beantwoord worden:

1. “Hoe ziet een minimale implementatie van GraphQL er uit?”
2. “Hoe sluit GraphQL aan op MongoDB?”
3. “Welke voordelen bied GraphQL ten opzichte van een REST API?”
4. “Hoe werkt een GraphQL query en wat is het verschil met een MongoDB query?”

Indien dit een onderzoek was geweest waar minder veld en lab werk bij vereist werd, zou het interessant geweest zijn om ook per deelvraag een hypothese op te stellen. Dat wil zeggen, hetgene waarvan de auteur verwacht dat het het antwoord op de deelvraag zal zijn. Aan het eind van het onderzoek zouden deze verwachtingen dan ook nog vergeleken kunnen worden met de daadwerkelijke resultaten van het onderzoek. Omdat echter het meerendeel van deze vragen het beste kan worden beantwoord met een praktische oplossing, en een van de doelstellingen ook is om te eindigen met een bruikbaar prototype, is dit niet meegenomen in het verslag.

**Aanpak:**

Bij deelvraag 1 zal gebruik gemaakt gaan worden van de bieb en- werkplaatsmethode. Er zal een klein test project worden opgesteld met de minimale implementatie van GraphQL. Tijdens het maken van dit testproject zal er ook kennis vrijkomen die bij het beantwoorden van de latere deelvragen relevant is. En voor de uitvoer van dit project zal natuurlijk informatie gezocht dienen te worden.

Bij deelvraag 2 kan tevens gebruik gemaakt worden van de bieb en- werkplaatsmethode. Hierbij kan worden voortgeborduurt op de resultaten van deelvraag 1, waarbij het dan het idee is om deze minimale implementatie te koppelen aan MongoDB.

Bij deelvraag 3 zal gebruik worden gemaakt van de bieb-methode. Er zal uitgezocht moeten worden wat de voordelen zijn van REST, en wat de voordelen zijn voor het gebruik van GraphQL. Bij deze deelvraag kan ook de showroom methode gebruikt worden door te kijken wat andere bedrijven aanraden of gebruiken.

Bij deelvraag 4 zal gebruik gemaakt moeten worden van de bieb en de werkplaatsmethode. Na het uitzoeken hoe een query precies in elkaar steekt, zal deze ook worden uitgevoerd door verder te bouwen op het test project dat eerder is opgesteld met de werkplaats methode.

Door het toepassen van deze methodes zal niet alleen relevante kennis worden vergaard met betrekking tot GraphQL, maar zal deze ook gelijk worden omgezet naar een werkend ‘prototype’ waar, als het antwoord op de hoofdvraag positief beantwoord wordt aan het einde van dit onderzoek, ook gelijk gebruik van gemaakt kan worden binnen het bedrijf zodat er bij toekomstig gebruik snel een opzet gemaakt kan worden.

# Onderzoeksresultaten

Deelvraag 1: **//**“Hoe ziet een minimale implementatie van GraphQL er uit?”

**Minimale implementatie (barebones):**

Deze deelvraag is beantwoord door gebruik te maken van de bieb en de werkplaatsmethode. In dit geval is er begonnen met het opzetten van de (letterlijk) minimale implementatie van GraphQL. Deze ziet er uit als volgt:

Figure 1, Minimale implementatie van GraphQL.

Zoals te zien in figuur 1, begint het met de definitie van een schema. In dit geval worden daarbinnen gelijk queries gedefinieerd, maar het is ook mogelijk om object types te definieren. Queries kunnen deze vervolgens ook teruggeven. De belangrijkste componenten voor de minimale implementatie zijn in ieder geval als volgt:

1. Een schema, waarin de datatypes en queries gedefinieerd kunnen worden
2. Een “root”, dat wil zeggen, de entrypoint voor GraphQL queries. Dit root object bevat de zogenoemde “resolvers”, dit zijn de functies die na een query worden aangeroepen om daadwerkelijk data op te halen uit bijvoorbeeld een MySQL of MongoDB.

Dat was het. Meer is niet nodig voor een minimale implementatie. GraphQL laat de gebruiker volledig vrij over de invulling van de achterliggende database, en is ook te gebruiken met andere talen dan (in dit geval) JavaScript.

**Minimale implementatie (express):**

De code voor deze implementatie is te vinden in bijlage A, en //INSERT REFERENCE HIER. Omdat deze inhoudelijk weinig verschild met de barebones implementatie, zal er in dit stukje dan ook vooral enkele voorbeelden besproken worden van GraphQL. Het is namelijk zo dat GraphQL voor express de mogelijkheid bied om deze op een grafische manier te ontdekken in de browser. Dit wordt gedaan doormiddel van GraphiQL.

Figure 2, GraphiQL

Zoals te zien in figuur 2, is het mogelijk om in de linkerkant van het scherm een query in te vullen. In het midden van het scherm worden de resultaten weergegeven. Aan de rechterkant is vervolgens ook nog de documentatie en alle bekende endpoints te zien.

Dit biedt uiteraard veel mogelijkheden ten opzichte van de huidige inrichting van de webservers, er hoeft bijvoorbeeld geen API documentatie meer gespecificieerd te worden, aangezien deze in het schema kan worden meegenomen als beschrijving. Ook hoeft er bijvoorbeeld geen API document meer opgestuurd te worden naar externe partijen, zij kunnen zelf onderzoeken wat er mogelijk is met de API’s. Ook is duidelijk aangegeven wat van iedere query de types van de resultaten zijn. Dit levert in JavaScript nogal eens een probleem op, omdat JavaScript een loosly typed language is, waardoor het soms niet duidelijk is in welk formaat de data wordt teruggegeven. Ook als er een wijziging plaatsvind, kan hierbij gelijk de documentatie worden aangepast.

Om nog even de query in figuur 2 toe te lichten: Deze GraphQL query, aangegeven door het eerste blok, {}, roept een vooraf gedefinieerde query aan, giveMeRandom. Hier wordt de parameter msg aan meegegeven, in dit geval een string. Vervolgens wordt nog aangegeven welke fields van het type ‘Result’ (een custom type met de fields ‘msg’ en ‘random’) gewenst zijn om te ontvangen. Dit geeft vervolgens het resultaat zoals in de middelste colom van figuur 2 te zien is. In bijlage A is vervolgens nog te zien hoe deze query opgebouwd wordt, en welke achterliggende functie daadwerkelijk de data teruggeeft.

Om het nog even kort samen te vatten, een minimale implementatie van GraphQL bestaat dus uit een schema, en een root object wat uit de resolver functies van de queries bestaat.

Deelvraag 2: //“Hoe sluit GraphQL aan op MongoDB?”

Voor het beantwoorden van deze vraag, is voortgeborduurd op het express testproject, zoals toegelicht bij het beantwoorden van deelvraag 1. De belangrijkste toevoegingen hierbij aan het testproject is de opzet van een MongoDB. Voor het opzetten van de MongoDB is de officiele documentatie van MongoDB gevolgd (//zie literatuurlijst). Dit omdat de auteur nog niet eerder zelf een MongoDB heeft opgezet, maar tot nu toe enkel nog gewerkt heeft met al opgezette instanties.

Nadat de MongoDB is geinstalleerd, dient deze gekoppeld te worden aan express doormiddel van een database driver. In dit geval werd gebruik gemaakt van MongoClient voor Express. Het koppelen van GraphQL was vervolgens vrij eenvoudig. Aangezien GraphQL gebruikt maakt van de zogenoemde “resolver” functies, die de daadwerkelijke data ophalen, en de gebruiker hierin qua invulling volledig vrij wordt gelaten.

In dit geval is er een MongoDB gevuld met enkele test objecten, 2 users en 2 apparaten:

|  |  |
| --- | --- |
| **User:** | **Apparaat:** |
| \_id: 1,  name: 'Robin van Dijk',  age: 21 | \_id: 1,  name: 'Apparaat 1',  values: [0, 2, 328, 17],  description: 'Dit apparaat doet X',  ownedBy: 1 //Owner of the device (Robin) |
| \_id: 2,  name: 'Bertje Bertsma',  age: 46 | \_id: 2,  name: 'Apparaat 2',  values: [32, 454, 0, 1],  description: 'Dit apparaat doet Y',  ownedBy: 2 //Owner of the device (Bertje) |

Het GraphQL schema voor deze tabellen ziet er ongeveer hetzelfde uit, met achter de dubbele punt het type aangeven in plaats van de waarde. Bijvoorbeeld: “\_id: String”.

Door het definieren van de volgende queries wordt het mogelijk een aanroep te doen op GraphQL, en daarmee bijvoorbeeld voor een specifiek apparaat of specifieke user data op te halen:

|  |
| --- |
| getUser(id: Int): User |
| getApparaat(id: Int): Apparaat |

Dit zou dus data kunnen zijn die uit een MongoDB, MySQL of zelfs uit een text bestand kan komen. In dit geval is er echter een MongoDB gekoppeld en zullen de resolve functies deze ook aanroepen om hun datavelden te populeren.



Figure 3, Resolver functies gekoppeld aan MongoDB.

Zoals in figuur 3 te zien, is er voor iedere in het schema beschreven query, tevens een resolver functie aangemaakt. Als er dus bijvoorbeeld een aanroep wordt gedaan op GraphQL met getUser(1), zal de resolver functie proberen deze gebruiker te vinden in de achterliggende MongoDB en deze terug te geven.

In dit geval wordt standaard een Promise geretourneerd, ook wel een “belofte”. Dit betekent dat de data wordt teruggegeven zodra deze is opgehaald. Met deze structuur is het bijvoorbeeld ook mogelijk om een externe API aan te roepen, en deze data later terug te geven. Om even de onderste query, getApparaat, toe te lichten: Allereerst wordt het apparaat met de opgegeven id opgehaald. Het ownedBy field in de MongoDB verwijst vervolgens naar de juiste user die bij het apparaat hoort. Om deze informatie ook op te halen, wordt met de id die is opgeslagen in het ownedBy field, vervolgens ook nog de correcte user opgehaald en bevestigd aan het object dat uiteindelijk wordt teruggegeven. Dit leidt uiteindelijk tot het resultaat zoals te zien in figuur 4.



Figure 4, Het resultaat van de "getApparaat" query

Een interessant punt om te benoemen, is dat GraphQL alle velden waar niet om gevraagd wordt, ook niet meeneemt, ookal worden deze bijvoorbeeld door MongoDB wel teruggegeven. Dit bespaard een hoop tijd, omdat niet meer voor iedere MongoDB query hoeft worden aangegeven welke velden wel teruggestuurd moeten worden.

Om het nog even kort samen te vatten sluit GraphQL dus aan op MongoDB door gebruik te maken van de resolver functies. Deze kunnen vervolgens een database driver gebruiken om de aanvraag te populeren met data. Het is bovendien ook vrij eenvoudig om deze koppeling te maken, omdat er dus door de resolver functies een Promise wordt teruggegeven. Hierdoor hoeft de data dus niet meteen teruggegeven te worden, maar pas als deze beschikbaar is.

Deelvraag 3: //“Welke voordelen bied GraphQL ten opzichte van een REST API?”

Uit de eerdere deelvragen zijn al een aantal voordelen naarboven gekomen. Door het maken van het test project is al gebleken dat enkele aspecten van GraphQL zeer goed van pas zouden kunnen komen bij de ontwikkeling van nieuwe prototypes. Hieronder zijn deze kenmerken nog even kort op een rijtje gezet:

1. Snel te implementeren
2. “Self-writing” documentatie, waarbij nog extra beschrijving kan worden toegevoegd.
3. Duidelijk aangegeven object types, in tegenstelling tot JavaScript.
4. Gemakkelijk te koppelen aan een database instantie, bijvoorbeeld MongoDB.
5. “You get what you ask for”, dat wil zeggen, alleen velden die expliciet opgevraagd worden in een query worden meegegeven als resultaat.

Dit staat in sterk contrast met de inrichting van een “traditionele” rest API. Waarbij het bij een traditionele rest API gebruikelijk is dat er meerdere endpoints gedefinieerd worden, bijvoorneeld /user/:id en apparaat/:id, is het voor GraphQL gebruikelijk dat er slechts 1 endpoint is waarop de aanvragen gedaan worden. Door de queries binnen GraphQL te definieren, is het mogelijk hetzelfde resultaat te behalen als wanneer er gebruik zou worden gemaakt van meerdere endpoints.

Ook de duidelijk aangegeven types, gecombineerd met de zichzelf schrijvende documentatie, maakt het niet alleen makkelijk voor externe partijen om te bekijken wat er mogelijk kan worden aangeroepen, maar maakt het ook intern heel makkelijk om te bekijken wat de API voor mogelijkheden biedt. Het is namelijk niet ongebruikelijk dat er gewerkt wordt aan het eerste project, en dan tussendoor een aantal weken aan een ander project, om vervolgens weer verder te gaan met het eerste. De ophanden documentatie kan hier veel tijd besparen met het “weer in de code komen” van het project.

Verder is het ook zeer makkelijk om de achterliggende database los te koppelen en te vervangen door een ander soort database. Dit omdat het schema en de types binnen GraphQL gedefinieerd zijn. Ook is het mogelijk de al geschreven queries te blijven gebruiken, en hoeft er dus op API niveau (meestal) geen veranderingen worden doorgevoerd.

Omdat het ook belangrijk is om te kijken naar de nadelen van het gebruik van GaphQL, zijn deze hieronder op een rijtje gezet:

1. Extra code nodig op de client om met een GraphQL server te praten. Dit kan ook gedaan worden door de plain text queries te versturen, maar dit is omslachtig.
2. Extra werk; het leid tot een extra abstractielaag tussen de database en de API.
3. Nieuwe technologie, dus weinig documentatie als er tegen een probleem wordt aangelopen
4. Er is minder mogelijk met een GaphQL endpoint dan met een REST endpoint.

Uit een artikel (Sturgeon, 2017), waar bovendien ook nog enkele andere vergelijkingen in getrokken worden, wordt gesteld dat de beide implementaties de nodige voor en- nadelen hebben, en dat de optimale manier om ze te gebruiken dan ook samen is. Tijdens het opzetten van het testproject kwam dit ook naar voren, het is bijvoorbeeld mogelijk om REST en GraphQL endpoints te combineren in 1 express server. Dit betekent ook dat de migratie van bestaande projecten aanzienelijk versimpeld wordt, aangezien de endpoints gewoon kunnen blijven staan tot deze gemigreerd zijn.

Deelvraag 4: //“Hoe werkt een GraphQL query en wat is het verschil met een MongoDB query?”

Bij eerdere deelvragen is hier en daar al kort aan bod gekomen hoe een GraphQL query is gestructureerd. Er zijn echter fundamentele verschillen tussen een MongoDB en een GraphQL query. Een MongoDB query bestaat normaal gezien uit de volgende componenten:

* De manier van queryen, bijvoorbeeld collection.find, findOne, update, of remove.
* De query zelf, deze wordt doorgaans als eerste argument meegegeven aan de “find” functie. Denk bijvoorbeeld aan *collection.*find({\_id: 1}), dit zal alle documenten teruggeven waar \_id 1 is. Indien findOne gebruikt zou worden, zou het eerste document wat voldoet aan de query worden teruggegeven.
* De opties van de query, dit gedeelte wordt als 2e argument meegegeven aan de find functies, en bestaat doogaans uit sorteer opties, of wordt gebruikt om aan te geven welke velden worden teruggegeven door de query.

Dit staat in sterk contrast met GraphQL queries, waabij de opties bijvoorbeeld ontbreken en in de query zelf al aangeven wordt welke velden teruggegeven dienen te worden. Een GraphQL query begint dan ook met blokhaken. Deze worden in de response, zoals te zien aan de rechterkant van figuur 5, aangegeven met het object “data”. Hierin zit dan het daadwerkelijke resultaat van de query. Ook is het mogelijk om comments in te nemen in een GraphQL query, aangegeven met het ‘#’ symbool. Binnen het object kunnen vervolgens 1 of meerdere queries worden opgenomen. Dit begint met het aangeven van de query, het toevoegen van de gewenste parameters voor de query, en vervolgens een blok met daarin alle gewenste velden die de query dient terug te geven.



Figure 5, Een query en het resultaat van deze query

Zoals in figuur 5 te zien is, is het resultaat van de query bijna identiek aan de query zelf. Zoals hierboven benoemd is het root object vervangen door het “data” object, met hierin een object key voor de uitgevoerde “getUser” query. Deze bevat vervolgens de fields zoals aangegeven en in dezelfde volgorde als aan de linkerkant.

# Conclusies en evaluatie

# Bijlagen

## A: Express minimale implementatie

Onderstaand plaatje is een voorbeeld van de minimale implementatie van GraphQL op een express server.

## 

# Literatuurlijst

Sturgeon, P. (2017, 24 januari). GraphQL vs REST: Overview [Blogpost]. Geraadpleegd van <https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/>