whitehorses

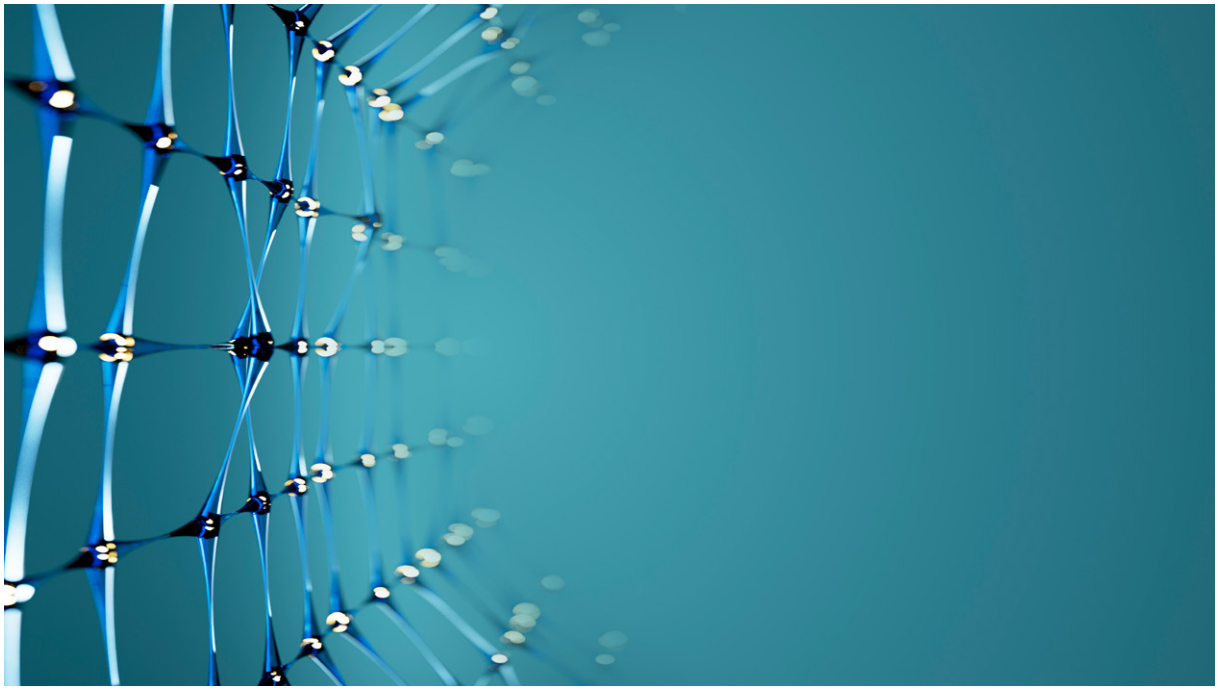
member of the human network

WHITEBOOK

Wat is GraphQL? Een voorbeeld met Mulesoft

December 2019

Auteur:
Martijn van de Goor
INTEGRATIESPECIALIST

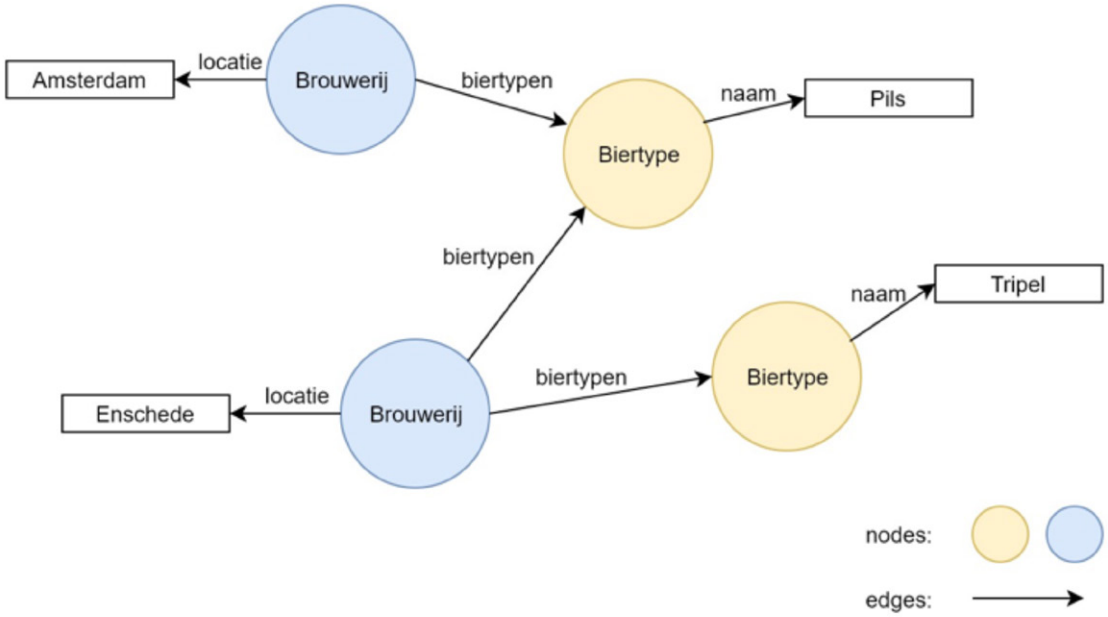


Inleiding

GraphQL is een krachtige nieuwe query taal die gebruikt wordt bij het bouwen van API's. Clients (meestal web- en mobiele apps) stelt het in staat om alleen de gegevens op te vragen die ze nodig hebben. GraphQL is oorspronkelijk ontworpen door Facebook om het dataverkeer bij het opvragen van gegevens te verminderen en het aantal requests voor het ophalen van data omlaag te brengen. GraphQL is vervolgens door diverse bedrijven (zoals Github en Yelp) geïmplementeerd ter vervanging van hun bestaande REST API's. In dit Whitebook zal ik de concepten van GraphQL uitleggen, en aan de hand van een voorbeeld, gebouwd met Mulesoft, laten zien hoe een GraphQL API eruit kan zien.

Wat is GraphQL?

De naam GraphQL komt van het datamodel dat GraphQL gebruikt om te werken met gegevens. Het idee erachter is dat data is uit te drukken als een graph (of graaf in het Nederlands). Een graph geeft de onderlinge verbanden weer tussen waarden in een multidimensionale ruimte. De waarden zelf worden nodes genoemd en hun verbanden worden edges genoemd. Een voorbeeld hiervan kan er als volgt uit zien:



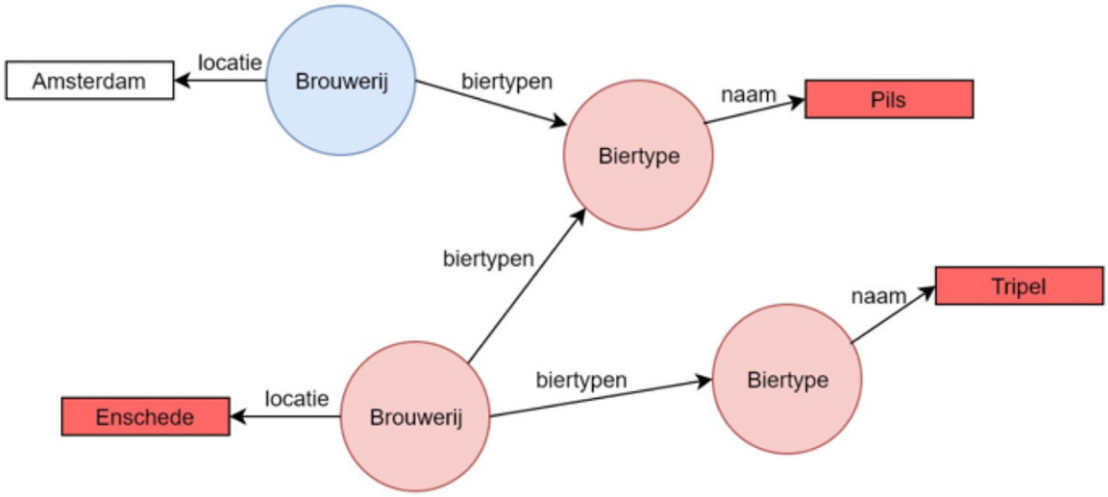
De *graph* geeft de relaties tussen de verschillende stukken informatie weer en de entiteiten die ze representeren. Applicaties lezen gegevens uit een graph met behulp van GraphQL. Om precies te zijn proberen ze een *tree* (boom) uit een graph te lezen. Een *tree* begint bij een *node* (de *root*) en volgt de *edges* naar andere *nodes* waarbij je nooit bij dezelfde *node* terugkomt.
Het wordt duidelijker wanneer we een voorbeeld van een GraphQL query bekijken die een tree uit bovenstaande applicatie-*graph* extraheert:

```
query {
  brouwerij(naam: ”Grolsch”) {
    locatie
    biertypen{
      naam
    }
  }
}
```

De server geeft het volgende als antwoord terug:

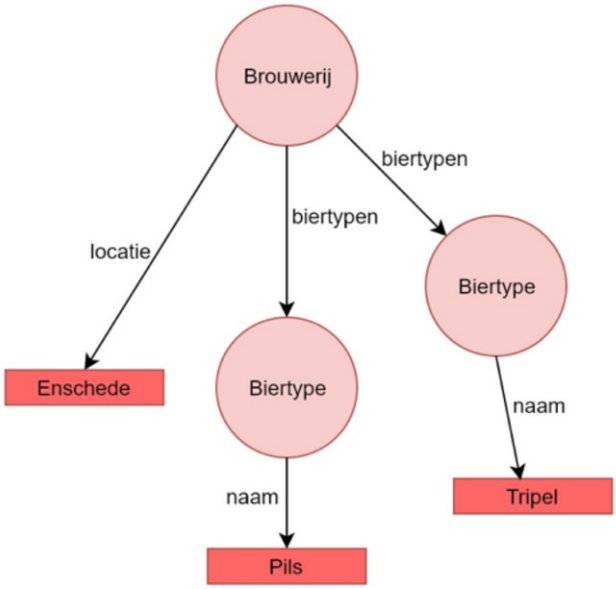
```
{
  «data»: {
    «brouwerij»: {
      «locatie»: «Enschede»,
      «biertypen»: [
        {
          «naam»: «Pils»
        },
        {
          «naam»: «Tripel»
        }
      ]
    }
  }
}
```

In de *graph* ziet het er als volgt uit:



We gaan bekijken hoe de informatie daadwerkelijk uit de *graph* is geëxtraheerd door de GraphQL-query. GraphQL start met een zogenaamd *Root Query Type* die aangeeft waar in de *graph* de query moet beginnen met zoeken. In ons voorbeeld is dit het veld *brouwerij* die we hebben geselecteerd met het veld *naam* (brouwerij(naam: "`Grolsch`")). Vervolgens doorloopt de GraphQL-query de *graph* door de *edges* te volgen die dezelfde naam hebben als de geneste velden in de query. Zo komt de query uiteindelijk bij de *branches* terecht waar we naar op zoek zijn. Voor onze zoekopdracht springt het van de node *Brouwerij* naar de locatie van de brouwerij via de edge getiteld locatie zoals ook in de query vermeld. Daarnaast volgt de query de *biertypen edges* naar de *biertype nodes* en wordt via de *naam edges* de *branches* met namen van de biertypen opgehaald.

Als *tree* ziet het er als volgt uit:



Voor elk stukje informatie dat de query retourneert, is er een bijbehorend query-pad dat bestaat uit de velden in de GraphQL-query die we hebben gevolgd om bij die informatie te komen. De locatie van de brouwerij heeft bijvoorbeeld het volgende query-pad:

Root Query → brouwerij(naam: Grolsch) → locatie

De velden in de GraphQL-query (bijv. locatie, biertypen, naam) geven aan welke *edges* moeten worden gevolgd in de applicatie-*graph* om het gewenste resultaat te krijgen.

Voordelen van GraphQL

GraphQL vereist een door de gebruiker gedefinieerd schema met een strikt type systeem. Dit schema fungeert als een blauwdruk voor de gegevens die door de API worden verwerkt. Ontwikkelaars weten daardoor precies wat voor soort gegevens beschikbaar zijn en in welk formaat.

Omdat GraphQL-query's definiëren welke gegevens precies nodig zijn tijdens het opvragen, kunnen alle gegevens die nodig zijn om bijvoorbeeld een pagina te *renderen* in één aanvraag worden opgevraagd. Hierdoor hoeft er maar één *request* naar de server te worden gestuurd. Wanneer REST API's zouden worden gebruikt, zouden in ons voorbeeld eerst de gegevens van de brouwerij moeten worden opgevraagd en vervolgens in een aparte call de gegevens met betrekking tot de verschillende biertypes. Daarnaast geeft de service alleen die velden terug die exact nodig zijn en wordt er geen overtollige data verwerkt.

Een GraphQL-voorbeeld met MuleSoft

Voor het bouwen van een GraphQL-service moet een al eerder genoemd schema worden gedefinieerd dat de typen, velden en query's beschrijft die beschikbaar zijn voor de service. Het schema voor ons brouwerij-voorbeeld ziet er als volgt uit:

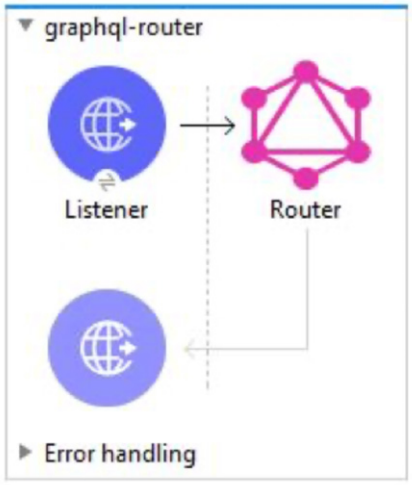
```
schema {
      query: Query
}

type Query{
      brouwerij(
         naam : String
     ): Brouwerij

}

type Brouwerij {
     id: ID
      naam: String
      locatie: String
      biertypen: [Biertype]
}

type Biertype {
     id: ID
     brouwerijId: ID
      naam: String
}
```

In MuleSoft gaan we de *resolvers* coderen die de gegevens per veld gaan ophalen. MuleSoft levert via een laboratorium project een connector (graphql-router) die de routering naar de verschillende *resolvers* regelt en het request in het juiste formaat teruggeeft.

De connector is op te halen uit github:

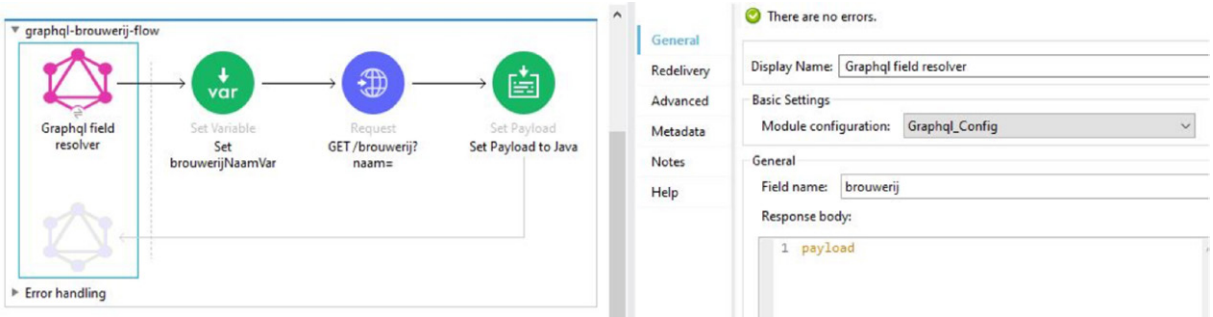
https://github.com/mulesoft-labs/graphql-router

De integratie begint met een *Listener* die het request oppakt. Vervolgens wordt de GraphQL router aangeroepen. In de routerconfiguratie staat een verwijzing naar het schema. Aan de hand van dit schema weet de router naar welke flows de *payload* moet worden gestuurd.

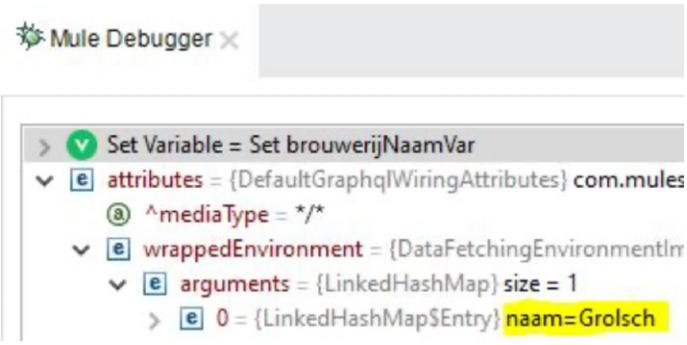


In ons voorbeeld gaat de router zoeken naar flows waarin de velden *brouwerij* en *biertypen* staan gedefinieerd. Beide flows starten met een GraphQL field resolver, welke een onderdeel is van de connector, waarin de velden staan gedefinieerd.

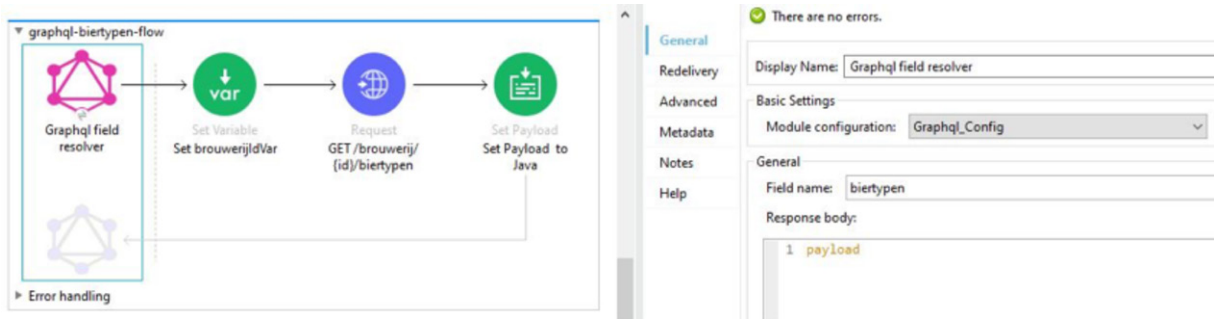
De router stuurt de payload eerst naar de brouwerij-flow om de brouwerij-gegevens op te halen:



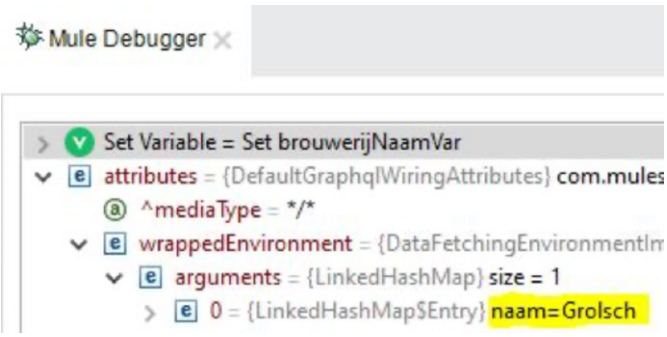
Bij binnenkomst van de flow zijn een aantal attributen beschikbaar gesteld door de connector die gebruikt kunnen worden in de resolvers. In de brouwerij-flow is de naam van de brouwerij te zien wanneer we er doorheen stappen met de debugger:



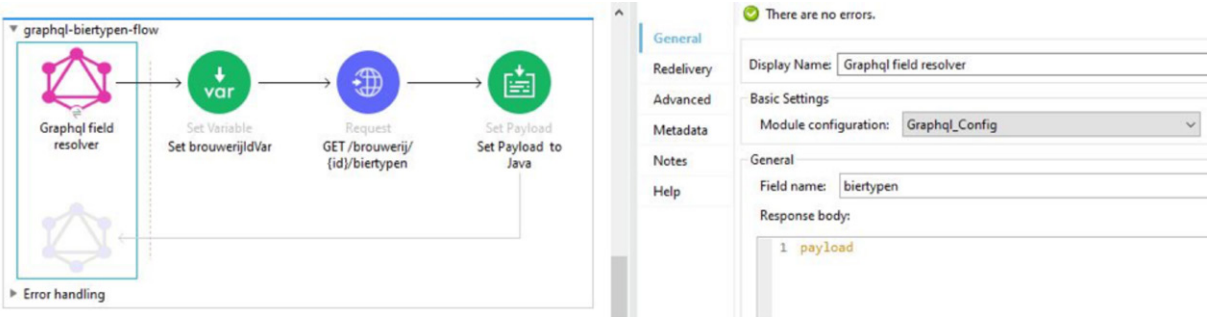
Deze naam wordt gebruikt om een brouwerij-API aan te roepen die de gegevens van de brouwerij ophaalt. Op deze manier wordt de locatie opgehaald, maar ook de ID van de brouwerij.
Nadat de brouwerij-gegevens zijn opgehaald, wordt er gerouteerd naar de biertypen-flow:



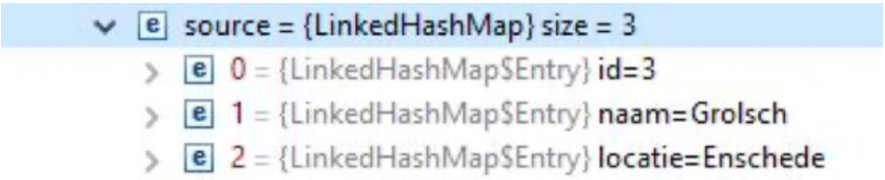
Bij binnenkomst van de flow zijn een aantal attributen beschikbaar gesteld door de connector die gebruikt kunnen worden in de resolvers. In de brouwerij-flow is de naam van de brouwerij te zien wanneer we er doorheen stappen met de debugger:



Deze naam wordt gebruikt om een brouwerij-API aan te roepen die de gegevens van de brouwerij ophaalt. Op deze manier wordt de locatie opgehaald, maar ook de ID van de brouwerij.
Nadat de brouwerij-gegevens zijn opgehaald, wordt er gerouteerd naar de biertypen-flow:



Ook hier zijn weer attributen beschikbaar, waaronder de master-gegevens uit de brouwerij-flow. Dit wordt de source genoemd:



De ID uit de *source* wordt gebruikt om bij de Grolsch brouwerij de verschillende beschikbare biertypen op te halen middels een biertypen-API.

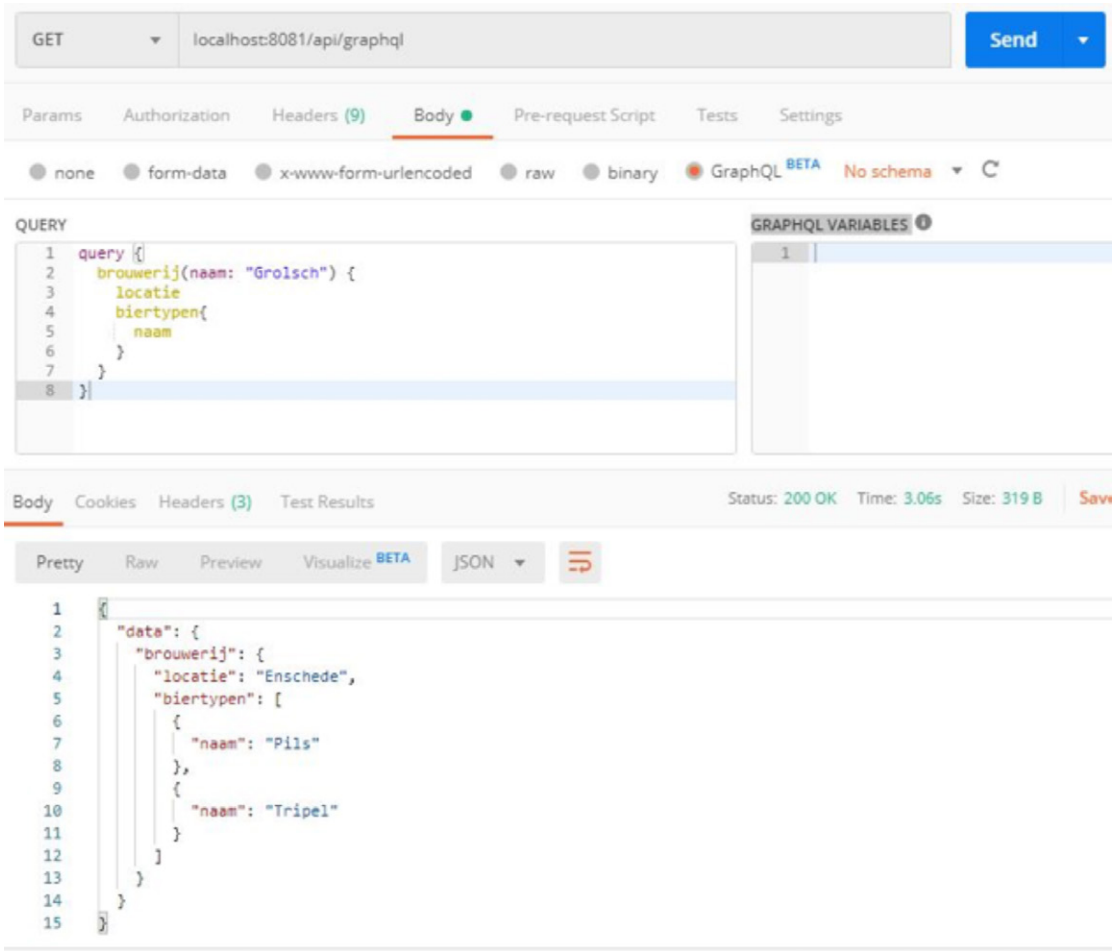
Uiteindelijk zorgt de GraphQL connector ervoor dat de response van de API het juiste formaat heeft en wordt de response teruggegeven.

Het graphql-mule-brewery-demo project

Het volledige project is te vinden in bitbucket en draait op Mule runtime 4.2.1:

https://bitbucket.org/whitehorsesbv/graphql-mule-brewery-demo

In de src\test\resources\Postman locatie van het project staat een collectie-file die in Postman is te importeren waarmee men het project kan testen.



Tot Slot

GraphQL is een interessante nieuwe toevoeging aan de API-toolbox. Het is vooral zinvol wanneer we te maken hebben met veel verschillende entiteiten en relaties tussen de gegevens. Hierdoor wordt het aantal *calls* naar de backend drastisch verminderd. Daarnaast is het uitermate geschikt wanneer er een groot aantal bronnen moet worden geraadpleegd. Dit kan dan aan de backend worden gegroepeerd en in één *response* worden teruggegeven aan de *caller*. Nadeel is wel dat de responstijd waarschijnlijk zal toenemen. Dit kan worden tegengegaan door caching in te zetten. Hiervoor is het principe van GraphQL uitermate geschikt. Dit is echter een onderwerp dat uitgebreid besproken dient te worden. Wellicht een goed onderwerp voor een volgend Whitebook.

Bronnen

https://graphql.org
https://engineering.fb.com/core-data/graphql-a-data-query-language/
https://blog.apollographql.com/the-concepts-of-graphql-bc68bd819be3