Gebruikers handleiding

Onderbeke Niels

Inhoud

[Intro 3](#_Toc94196832)

[Kennis 3](#_Toc94196833)

[Gebruik Apollo Client v3 4](#_Toc94196834)

[Voorbeeld 4](#_Toc94196835)

[Benodigdheden 4](#_Toc94196836)

[Code 5](#_Toc94196837)

[Gebruik HTTP caching in combinatie met Automatic Persisted Queries 8](#_Toc94196838)

[Voorbeeld 8](#_Toc94196839)

[Benodigdheden 8](#_Toc94196840)

[Werking 9](#_Toc94196841)

[Code 10](#_Toc94196842)

[Gebruik GraphCDN 12](#_Toc94196843)

[Voorbeeld 12](#_Toc94196844)

[Benodigdheden 12](#_Toc94196845)

[Werking 13](#_Toc94196846)

## Intro

Dit is een handleiding voor de door mij best bevonden caching mogelijkheden binnen GraphQL. Hierin zal u te weten komen hoe u zelf met de verschillende technologieën & methodes aan de slag kan gaan.

In deze tabel vindt u de beste gevonden caching mogelijkheden per strategie:

|  |  |  |
| --- | --- | --- |
| **Server-side** | **Client-side** | **CDN** |
| HTTP caching  (Met automatic persisted queries) | Apollo Client v3 | GraphCDN |

Om deze ervaring zo vlot mogelijk te laten verlopen, volgt u eerst de stappen die beschreven staan in de **installatiehandleiding**. Op deze manier bent u zeker dat alles op de correcte manier samen werkt.

### Kennis

Om dit project te gaan gebruiken heeft u een goede basiskennis nodig van verschillende development technologieën zoals Node.js, Docker & SQL databases. Om dit project te kunnen testen en begrijpen is er enige kennis over (HTTP) caching en React.js vereist.

U kan meer informatie over dit project verkrijgen via deze [Medium](https://medium.com/@niels.onderbeke.no/research-project-which-is-the-best-caching-strategy-with-graphql-for-a-big-relational-database-56fedb773b97) blog post. Hierin bespreek ik alle caching methodes die ik onderzocht heb & vindt u ook de meetresultaten van de caching strategieën.

## Gebruik Apollo Client v3

### Voorbeeld

GitHub branch

<https://github.com/OnderbekeNiels/research-project-3mct/tree/apollo-client>

**Belangrijk:** Om de Apollo Client Devtools in Google Chrome te kunnen gebruiken zal u de development client-server moeten opstarten. De Apollo Client Devtools zijn **niet** beschikbaar in een productie build van Apollo Client. De client-server-next Docker container is ook een productie build dus u gebruikt deze het best niet tijdens het ontwikkelen.

Volledige Docker omgeving

1. Om de Docker omgeving op te starten hoeft u enkel volgend commando in te geven in de terminal in de root folder van het project=> **docker-compose -f "docker-compose.yml" up -d --build**
2. De client-side applicatie is beschikbaar via volgend adres => [**http://localhost:3000/**](http://localhost:3000/)

Niet volledige Docker omgeving

Om de database te runnen volgt u de stappen zoals in de Docker sectie maar hoeft u de andere services niet meer uit commentaar te halen.

Om de Next.js & Express server op uw eigen Node omgeving te runnen moet u volgende commando’s uitvoeren in de client-app & typegql-sever mappen: **npm install** & **npm run dev**.

Op deze manier start u de development build op van beide applicaties. Vervolgens kan u de client-side applicatie bereiken op volgend adres [**http://localhost:3000/**](http://localhost:3000/)

### Benodigdheden

Server-side

* GraphQL backend

Client-side

* React front-end
* Apollo Client v3

### Code

Queries

Om via Apollo Client data te gaan cachen heeft u nood aan de Apollo Client v3 package ( <https://www.apollographql.com/docs/react/get-started/> ) & een React omgeving ( in dit project is dit Next.js ).

Deze Apollo Client package voorziet u van 2 verschillende hooks om te gaan querieën: useQuery & useLazyQuery. De Apollo Client handelt de request voor u af. Er wordt gebruik gemaakt van een in-memory cache.

**useQuery**

De useQuery hook stelt u in staat direct bij het inladen van uw component een request te gaan doen.

client-app/pages/posts/index.tsx

Text

Description automatically generated

Aan de hand van de gedestructureerde objecten van useQuery kan u uw UI gaan aanpassen.

Text

Description automatically generated

**useLazyQuery**

De useLazyQuery stelt u in staat om data pas op te gaan halen wanneer u dat zelf aangeeft. U geeft een functie mee, en pas wanneer u deze functie aanroept, zal de request uitgevoerd worden.

client-app/pages/users/[userId].tsx



A picture containing text

Description automatically generated

Meer informatie over deze Query mogelijkheden vindt u op deze webpagina’s: <https://www.apollographql.com/docs/react/data/queries/#usequery-api>

<https://www.apollographql.com/docs/react/api/react/hooks/#uselazyquery>

**Caching**

Alle requests die door de useQuery hooks gebeuren worden standaard gecached met als fetchpolicy **cache-first.** Om het caching mechanisme zijn werk te laten doen moet u wel telkens het Id opvragen van elke entiteit / type. Apollo gebruikt dit namelijk om te cachen. U kan dit ook anders configureren ([lees meer](https://www.apollographql.com/docs/react/caching/cache-configuration/)). Apollo voorziet u in totaal van 6 fetchpolicies die het cache gedrag van uw request kunnen configureren.

Meer uitleg over deze fetchpolicies kan u [hier](https://www.apollographql.com/docs/react/data/queries/#setting-a-fetch-policy) vinden. <https://www.apollographql.com/docs/react/data/queries/#setting-a-fetch-policy>

Om te gaan verifiëren of uw data gecached wordt kan u gebruik gaan maken van de Apollo Client Devtools voor Google Chrome. Deze extensie is zeer duidelijk en maakt het debuggen van caching makkelijk.

<https://chrome.google.com/webstore/detail/apollo-client-devtools/jdkknkkbebbapilgoeccciglkfbmbnfm>

Mutations

Mutations worden binnen Apollo Client steeds uitgevoerd met de useMutation hook ( <https://www.apollographql.com/docs/react/api/react/hooks/#usemutation> ). Deze lijkt enorm hard op de useLazyQuery hook omdat er ook steeds een functie aangeroepen wordt om de request uit te voeren.

Voorbeeld:

Text

Description automatically generatedText

Description automatically generated

**Update**

Wanneer u een update uitvoert en de geüpdatet velden terug queried, zal Apollo automatisch de cache gaan mergen met uw response. Dit is zeer handig en zorgt ervoor dat u zich hier geen zorgen over hoeft te maken.

Bent u toch niet in staat de geüpdatet velden terug te krijgen, dan kan u de cache ook manueel gaan updaten. Meer info vindt u hier <https://www.apollographql.com/docs/react/data/mutations/#updating-the-cache-directly> .

client-app/pages/posts/[postId]/edit.tsx

Text

Description automatically generatedText

Description automatically generated

**Create & Delete**

Create: client-app/pages/posts/create.tsx

Delete: client-app/components/post.tsx

Deze mutations werken net zoals bij updates met de useMutation hook. Het verschil hierbij is dat de in-memory cache niet automatisch geüpdatet kan worden.

Bij een create zal de response wel toegevoegd worden aan de cache, maar weet de cache niet tot welke parent of lijsten deze moet gaan horen. U kan dit manueel gaan doen door de cache zelf up te daten of te gaan definiëren welke queries opnieuw uitgevoerd moeten worden.

<https://www.apollographql.com/docs/react/data/mutations/#refetching-after-update>

Bij een delete kan de cache niet automatisch weten welke entiteit hij moet verwijderen. Om dit handmatig te doen kan u volgende code gaan gebruiken. U vertelt dus aan Apollo Client welke data hij moet gaan verwijderen uit zijn cache.

Text

Description automatically generated

U kan steeds de cache status van de app gaan controleren met de Apollo Client Devtools.

## Gebruik HTTP caching in combinatie met Automatic Persisted Queries

### Voorbeeld

Github branch

<https://github.com/OnderbekeNiels/research-project-3mct/tree/apollo-client-persisted-queries>

Volledige Docker omgeving

1. Om de Docker omgeving op te starten hoeft u enkel volgend commando in te geven in de terminal in de root folder van het project=> **docker-compose -f "docker-compose.yml" up -d --build**
2. De client-side applicatie is beschikbaar via volgend adres => [**http://localhost:3000/**](http://localhost:3000/)

Niet volledige Docker omgeving

Om de database te runnen volgt u de stappen zoals in de Docker sectie maar hoeft u de andere services niet meer uit commentaar te halen.

Om de Next.js & Express server op uw eigen Node omgeving te runnen moet u volgende commando’s uitvoeren in de client-app & typegql-sever mappen: **npm install** & **npm run dev**.

Op deze manier start u de development build op van beide applicaties. Vervolgens kan u de client-side applicatie bereiken op volgend adres [**http://localhost:3000/**](http://localhost:3000/)

### Benodigdheden

Server-side

* Apollo Server
* Node.js
* Express
* TypeGraphQL

Client-side

* Node.js
* React front-end
* Apollo Client v3

### Werking

Graphical user interface, application, Teams

Description automatically generated

Hoe dit exact werkt kan u lezen op deze webpagina: <https://www.apollographql.com/docs/apollo-server/performance/apq/>

Belangrijk om te weten is dat u hier **niet** zoals bij Apollo Client de cache kan gaan updaten. De cache wordt telkens geüpdatet door een nieuwe request, die wordt verstuurd door een verlopen max-age of door vb. een stale-while-revalidate directive in de cache-control header. Bekijk hiervoor zeker eens alle mogelijkheden van [HTTP caching](https://www.youtube.com/watch?v=HiBDZgTNpXY&t=1s).

### Code

Server

Om te beginnen heeft u een server nodig die **Automatic Persisted Queries** ondersteunt. Ik heb hiervoor gebruik gemaakt van [Apollo Server](https://www.apollographql.com/docs/apollo-server/). Deze ondersteunt Automatic Persisted Queries zonder verdere configuratie.

Wat echter wel configuratie vereist, is het instellen van HTTP caching directives. Door met automatic persisted queries te gaan werken in combinatie met GET requests, bent u in staat om de browser of een CDN uw respones te laten cachen.

Caching op de server-side kan u op 2 verschillende manieren gaan instellen. De eerste manier is a.d.h.v. Apollo Directives. De tweede manier is door zelf de directives te definiëren die in de cache-control header moeten komen. Wanneer u niet veel weet over HTTP caching raad ik u sterk aan deze [video](https://www.youtube.com/watch?v=HiBDZgTNpXY&t=1s) eerst eens te bekijken. Hierin wordt uitgelegd hoe HTTP caching werkt en wat welke directive doet.

*Manier 1: Apollo Directives*

Hierbij kan u een cacheControl directive gaan plaatsen bij een **field** of **type**. A.d.h.v. deze directive berekend Apollo Server hoe uw cache-control headers eruitzien. U kan een maxAge en scope (public of private) meegegeven in de directive. Dit zijn dan ook de enige 2, er is dus niet veel customization mogelijk. Met manier 2 kunt u veel meer gebruik maken van de kracht van HTTP caching.

*typegql-server/src/entity/Posts.ts*

A screenshot of a computer

Description automatically generated with medium confidence

U kan dit ook in de resolver gaan plaatsen via volgende code:

typegql-server/src/resolvers/post.resolver.ts

Text

Description automatically generated

Ook hier kan u enkel de maxAge & scope mee gaan geven.

Opgepast! Wanneer u meerde directives op fields van een type plaatst wordt steeds de laagste max-age genomen. Ook moet u hierbij rekening houden met de scope (public of private). Hoe dit exact berekend wordt, vindt u [hier](https://www.apollographql.com/docs/apollo-server/performance/caching/).

Belangrijk hierbij is dat u deze [plugin](https://www.apollographql.com/docs/apollo-server/api/plugin/cache-control/) instelt bij het aanmaken van uw server:

typegql-server/src/index.ts

Text

Description automatically generated

Deze plugin zorgt ervoor dat Apollo Server uw cache-control headers kan bereken a.d.h.v. de geplaatste directives.

*Manier 2: Manueel de response cache-control headers instellen*

Deze manier biedt u de meeste mogelijkheden als het op HTTP caching aan komt. Hiervoor zet u ten eerste de calculateHTTPHeaders property op false of u verwijdert de ApolloServerPluginCacheControl plugin.

Vervolgens zorgt u ervoor dat het response object binnen Express meegeven wordt aan de GraphQL context.

typegql-server/src/index.ts

Graphical user interface, text

Description automatically generated

Op deze manier kunt u in uw resolvers volgende code gaan schrijven:

typegql-server/src/resolvers/post.resolver.ts

Text

Description automatically generated

Client

In tegenstelling tot de server, moet u in de Apollo Client wel Automatic Persisted Queries configureren. U kan dit doen bij het aanmaken van uw client zoals in onderstaand voorbeeld.

client-app/utils/apollo.ts

A screenshot of a computer

Description automatically generated with medium confidence

Belangrijk hierbij is dat er steeds een **GET** request gestuurd wordt. Zo kan de browser onze responses gaan cachen.

## Gebruik GraphCDN

### Voorbeeld

Github branch

<https://github.com/OnderbekeNiels/research-project-3mct/tree/graphCDN>

Volledige Docker omgeving

1. Om de Docker omgeving op te starten hoeft u enkel volgend commando in te geven in de terminal in de root folder van het project=> **docker-compose -f "docker-compose.yml" up -d --build**
2. De client-side applicatie is beschikbaar via volgend adres => [**http://localhost:3000/**](http://localhost:3000/)

Niet volledige Docker omgeving

Om de database te runnen volgt u de stappen zoals in de Docker sectie maar hoeft u de andere services niet meer uit commentaar te halen.

Om de Next.js & Express server op uw eigen Node omgeving te runnen moet u volgende commando’s uitvoeren in de client-app & typegql-sever mappen: **npm install** & **npm run dev**.

Op deze manier start u de development build op van beide applicaties. Vervolgens kan u de client-side applicatie bereiken op volgend adres [**http://localhost:3000/**](http://localhost:3000/)

### Benodigdheden

Server-side

* GraphQL backend
* GraphCDN CLI

Client-side

* Front-end applicatie die requests maakt naar de CDN

### Werking

Voor u begint te werken met GraphCDN raad ik u aan deze [video](https://www.youtube.com/watch?v=EjrJtp4JaGQ&t=1061s) te bekijken. In deze video geeft Max Stoiber (co-founder GraphCDN) wat meer uitleg over GraphCDN & waarom het bestaat.

Features

GraphCDN werkt a.d.h.v. de cache-control header, maar in tegenstelling tot de Automatic Persisted Queries + HTTP caching manier van werken, update GraphCDN wel uw data na een update mutation.

GraphCDN voorziet ook een [Purging API](https://docs.graphcdn.io/docs/purging-api). Deze wordt automatisch aangemaakt bij het toevoegen van uw API endpoint. Via de Purging API kan u van uit uw backend tegen GraphCDN zeggen welke cached data invalid moet worden. U kan hier meer over lezen op onderstaande links.

Auto update: <https://docs.graphcdn.io/docs/automatic-cache-invalidation-via-mutations>

Purging API: <https://docs.graphcdn.io/docs/purging-api>

Opzetten

Om caching met GraphCDN op te gaan zetten heeft u geen specifieke configuratie nodig op uw GraphQL server. Ook heeft u geen specifieke GraphQL client nodig op de client-side. De caching wordt volledig geconfigureerd via het GraphCDN dashboard.

Maak via deze [link](https://graphcdn.io/signup) een account aan en volg de stappen. Er zal u gevraagd worden om uw GraphQL server adres mee te geven, voorlopig heeft u deze nog niet dus negeer dit even. U kan wel uw service al een naam geven.

Nu heeft u 2 opties, u kan gebruik maken van de develop omgeving van GraphCDN met de develop CLI (aangeraden) of u tunnelt uw lokale GraphQL server IP naar een domeinnaam via een tool zoals ngrok. In deze handleiding wordt enkel de eerste optie behandeld.

Om de CLI te gebruiken, volgt u de stappen zoals hier beschreven: <https://docs.graphcdn.io/docs/how-to-use-graphcdn-in-development> . Om de effectieve caching te gaan testen kan u zelf caching rules definiëren op het GraphCDN dashboard en deze vervolgens pullen met **graphcdn pull.** Of u gebruikt de voorbeeld cache rules die in de **graphcdn.yml** file staan in de GitHub repository map**.**

Wanneer dit allemaal in orde is voert u enkel nog het commando **graphcdn serve --backend-port 4000 --service stackof-rp --path /graphql** uit.

Nu zal er op poort 3010 op uw localhost een proxy draaien die zich zal gedragen als de CDN. Deze zal vervolgens aangesproken worden door de client-app.