Advanced Access Control in GraphQL

Jaber, Amin Fayeq Nimer
TU Berlin
some@mail.de

Klausing, Wilke
TU Berlin
klausing@campus.tu-berlin.de

Nguyen, Huy Viet
TU Berlin
huy.v.nguyen@campus.tu-berlin.de

***Abstract*—GraphQL is a query language for reading and mutating data in APIs. It provides the back-end developer a type system to describe a data schema. This in turn gives front-end developers of the API the power to request the exact data they need. Although GraphQL is perceived as 'the successor of the good old REST', there is no widely established way to easily implement access control in GraphQL APIs. In this paper we will give a deeper insight to GraphQL, touch on different access control models that are common practice, show existing technologies that enable access control in GraphQL and introduce a new way to allow purposed-based access control in GraphQL which goes beyond traditional, account- or role-based access control.**

***Index Terms*—Privacy Engineering, Access Control, GraphQL, Apollo Server**

I. Introduction

Write what this paper is about and what topics will be introduced in each chapter.

II. What is GraphQL?

Graph Query Language or short GraphQL is a query language and server-side runtime developed by Facebook in 2012. At first it was an internal project to help to increase the performance of its mobile apps, which suffered from more and more complexity. Especially the news feed required a lot of queries to the server, GraphQL solved this by offering more flexibility and efficiency. It supports not just query operations but mutations as well and is available in any language since it is only a specification. In 2015, Facebook published GraphQL to the public as an open-source project under the MIT licence. The GraphQL Foundation, which is part of the Linux Foundation, takes care of further developments. (4)

A. How does GraphQL works?

A mayor concept of GraphQL is to think about data not in terms of resource URLs or tables but rather as a graph of objects. The client formulates queries, mutations or subscriptions to the server. Queries are used to retrieve data from the server, whereas mutations are used to change data within a predefined schema. Subscriptions command the server to push a notification to the client if some specific data changes. While queries and mutations use HTTP protocol for data transfer, does subscription require a web socket with a constant connection between server and client. (2)

If a client wants to make a query request it needs to send a HTTP GET or POST request to the endpoint, which is usually called /graphql. Each request contains a query which looks similar to a JSON object. Fields which are filled are used as search parameters to find the empty fields, which the client asked for. The fields are static types. Here you can see how a query can look like, with the respective answer on the right side. (3)

!!!Picture of Querie!!!

For GET requests is the query encoded in the URL. This can look like the following example URL: http://localhost:8080/graphql?query=...variables=...operation=... . When POST is used the query is put into the request body. (3)

On the server side, when receiving the requests, it needs to be parsed, processed and answered accordingly. Parsing and answering are defined in the GraphQL protocols and are greatly standardized. For these steps' libraries are widely available in many programming languages for different platforms. The process step, on the other hand, needs to be defined by the developer. Where the data comes from and what to include or exclude cannot be standardized because this highly depends on the use case. For each field that needs to be filled one resolver is used. The resolver is basically a function to get the data. Where the data comes is up the implementation by the developer. It could, for instance, be a text file, database, or from a data object which is stored in memory. For a good performance the resolvers are executed in parallel, if possible, instead of sequentially. To avoid several requests to the same source, resolver share resources between each other. (2)

B. GraphQL vs Restful

The prior standard for web APIs before GraphQL is call RESTful API. It was introduced in 2000 in order to simplify the communication between machines using the HTTP protocol without any additional layers. This chapter will compare both APIs. (5)

A first difference is that GraphQL, in contrast to RESTful, uses only one endpoint. RESTful offers one endpoint for each data. For instance, RESTful saves Person data at the endpoint /person, while Tax data is saved at the endpoint /tax. This leads to problem called Overfetching and Underfetching. Overfetching refers to the problem that more data is fetched then the client asked for. For instance, if a client needs to fetch the name of a person it can fetch from endpoint /person. The response would be a JSON object that can contain also information like birthday, address or age. Underfetching refers to the problem that one endpoint does not contain all the information. The client needs to make several request to the server to get all the data it wanted in the first place. (1)

C. Wilke Temp Lib

1. https://www.howtographql.com/basics/1-graphql-is-the-better-rest/
2. https://www.heise.de/developer/artikel/Was-man-ueber-GraphQL-wissen-sollte-4997158.html
3. https://graphql.org/learn/queries/fields
4. https://www.ionos.de/digitalguide/websites/web-entwicklung/graphql/
5. https://www.moesif.com/blog/technical/graphql/REST-vs-GraphQL-APIs-the-good-the-bad-the-ugly/

D. Privacy Engineering

TODO (what privacy engineering is in general, the impact and importance of PE in our lifes)

E. Access Control

TODO (most common access control models + detailed explaination for purpose based access control)

III. Existing Technologies

This chapter will introduce two APIs that are used for access control for current GraphQL frameworks. GraphQL Shield, which creates an additional permission layer on top of existing GraphQL APIs and GraphQL Filter which enables the back-end developer to filter the output of processed requests.

A. GraphQL Shield

GraphQL Shield is based on GraphQL Middleware which makes it possible to run arbitrary code before or after a GraphQL resolver is invoked. The Shield API was made to create field, type and role-based access control for any GraphQL and is compatible with all JavaScript GraphQL servers. The GraphQL Shield can be added into existing GraphQL Servers via GraphQL Middleware:

Listing 1. Integration [1]

```
// Permissions...

// Apply permissions middleware with applyMiddleware
// Giving any schema (instance of GraphQLSchema)

import { applyMiddleware } from 'graphql-middleware'
// schema definition...
schema = applyMiddleware(schema, permissions)
```

And rules can be set by using logical operations: OR, AND, NOT, CHAIN (rules will be executed one by one until one fails or all pass) and RACE (chain rules, execution stops once one of them returns true). An example of how role-based access control works with GraphQL Shield can be seen in Listing 2 .

Listing 2. Role-based Access Control [1]

```
import { shield, rule, and, or } from 'graphql-shield'

const isAdmin = rule()(async (parent, args, ctx, info) => {
  return ctx.user.role === 'admin'
})

const isEditor = rule()(async (parent, args, ctx, info) => {
  return ctx.user.role === 'editor'
})

const isOwner = rule()(async (parent, args, ctx, info) => {
  return ctx.user.items.some((id) => id === parent.id)
})

const permissions = shield({
  Query: {
    users: or(isAdmin, isEditor),
  },
  Mutation: {
    createBlogPost: or(isAdmin, and(isOwner, isEditor)),
  },
  User: {
    secret: isOwner,
  },
})
```

The code in Listing 2 defines the following rules for a certain blog:

- Only an editor or an admin can see a list of all users
- Only an admin or an owner with an editor role can create blog posts
- Only the owner of the blog can see user's secrets

B. GraphQL Filter

GraphQL Filter is another API that was created based on GraphQL Middleware. It allows to set specific types private. The API is said to provide an additional privacy layer to GraphQL and replaces private information before the query result is sent back to the client. The API is the only framework apart from GraphQL Shield that provides access control for GraphQL as of now. There is no documentation available. There is one test case to showcase how to integrate GraphQL Filter into existing systems. The developer designed it in order to create a more flexible privacy control:

> "It may be able to implement flexible read control with graphql-filter and graphql-shield." [2]

The lack of available APIs for access control in GraphQL shows the relevance of this paper's topic and a real need for a re-useable component that allows developers to easily add access control into their systems.

IV. Approach

TODO (introduce our idea)

References

[1] Zavadlal, M., 2021. maticzav/graphql-shield. [online] GitHub. Available at: ¡https://github.com/maticzav/graphql-shield¿ [Accessed 9 May 2021].

[2] Hata, T., 2021. hata6502/graphql-filter. [online] GitHub. Available at: ¡https://github.com/hata6502/graphql-filter¿ [Accessed 11 May 2021].

[3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, “Title of paper if known,” unpublished.

[5] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.