FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Application of GraphQL for Dynamic Data Models

José Pedro Maia Martins

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. António Miguel Pontes Pimenta Monteiro

July 29, 2022

Application of GraphQL for Dynamic Data Models

José Pedro Maia Martins

Mestrado Integrado em Engenharia Informática e Computação

July 29, 2022

Resumo

GraphQL é uma linguagem de consulta para APIs e um motor de execução para a resolução das referidas consultas com dados existentes, cujo foco principal é dar aos clientes o poder de pedir exatamente os dados de que necessitam de um servidor num único pedido.

A aplicação de GraphQL em aplicações cujo modelo de dados subjacente é estático está bastante bem definida com numerosos recursos disponíveis online. Os programadores podem facilmente construir o esquema GraphQL que descreve o seu modelo de dados antecipadamente, utilizando, por exemplo, a linguagem de definição do esquema de GraphQL.

Num contexto em que o modelo de dados subjacente de uma aplicação não é estático, ou seja, o modelo é dinâmico e não é conhecido em tempo de compilação, a criação de APIs GraphQL não é linear. O esquema GraphQL da API precisa de ser gerado dinamicamente e atualizado em tempo de execução de modo a estar sempre em sincronia com o modelo de dados em questão. Isto, por sua vez, proporciona uma camada adicional de complexidade à utilização padrão da tecnologia GraphQL, que ainda não foi suficientemente estudada e analisada.

Este projecto visa analisar, implementar e avaliar uma prova de conceito que permite validar a implementação de APIs GraphQL num contexto em que o modelo de dados subjacente não é conhecido em tempo de compilação, com uma análise comparativa adicional em relação a APIs REST tradicionais, tanto em termos de desempenho, como de experiência do programador e de usabilidade, analisando as vantagens qualitativas e quantitativas em relação às últimas.

O projecto é realizado no contexto da aplicação MES da Critical Manufacturing, que, em virtude das suas exigências comerciais, adaptabilidade e flexibilidade, tem um modelo de dados dinâmico que não é necessariamente conhecido de antemão.

Após a conclusão do trabalho com a implementação de uma prova de conceito que permite consultar alguns aspectos dos dados da aplicação MES da Critical Manufacturing, podemos validar a aplicabilidade de GraphQL para modelos de dados dinâmicos.

Podemos também concluir que a utilização de GraphQL para o cenário referido é uma escolha viável uma vez que uma API GraphQL é capaz de demonstrar um desempenho semelhante a uma típica alternativa REST, além de proporcionar alguns benefícios extra relacionados com os benefícios da tecnologia GraphQL, tais como a aquisição de múltiplas parcelas de informação num único pedido.

Abstract

GraphQL is a query language for APIs and a execution engine for fulfilling said queries with existing data, whose primary focus is to give clients the power to ask for exactly the data they need from a server in a single request.

The application of GraphQL in applications whose underlying data model is static is pretty well defined with many available resources online. Developers can easily construct the GraphQL schema that describes their data model ahead of time, using, for example, GraphQL schema definition language.

In a context where an application's underlying data model is not static, that is, the model is dynamic and is not known at compile-time, the creation of GraphQL APIs is not straightforward. The GraphQL schema of the API needs to be dynamically generated and updated in runtime so as to always be in sync with the data model in question. This, in turn, provides an additional layer of complexity to the standard usage of GraphQL technology, which hasn't yet been adequately studied and analyzed.

This project aims to analyze, implement, and evaluate a proof-of-concept that allows validating the implementation of GraphQL APIs in a context where the underlying data model is not known at compile-time, with further comparative analysis against traditional REST APIs, both in terms of performance, developer experience and usability, analyzing qualitative and quantitative advantages over the latter.

The project is carried out in the context of Critical Manufacturing's MES application, which, by virtue of its business requirements, adaptability, and flexibility, has a dynamic data model that is not necessarily known in advance.

After the work's conclusion with the implementation of a proof-of-concept that allows querying some aspects of Critical Manufacturing's MES application data, we can validate the applicability of GraphQL for dynamic data models.

We can also conclude that using GraphQL for the mentioned scenario is a viable choice since a GraphQL API can demonstrate similar performance to a typical REST alternative, in addition to providing some extra benefits related to the GraphQL technology strengths, such as retrieving multiple pieces of information in a single request.

Acknowledgments

I would like to thank everyone who was involved in this project.

From Critical Manufacturing's side, thanks for giving me great support and making sure I had all the tools needed to complete this work. Special thanks to Tiago Galvão for being my main point of contact inside the company and helping me with his knowledge and ideas.

Thanks to Prof. Miguel Pimenta Monteiro for the guidance in writing the dissertation.

Thanks to all my close friends with whom I always share good times and help me stay happy and grateful in life.

Finally, I thank my family for enabling me to get to this position and for always helping me with anything throughout my life. None of this would be possible without them.

José Pedro Maia Martins

“An expert is a man who has made all the mistakes that can be made in a very narrow field.”

Niels Bohr

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

API	Application Programming Interface
CM	Critical Manufacturing
GUI	Graphical User Interface
IDE	Integrated Development Environment
MES	Manufacturing Execution System
SDL	Schema Definition Language
UI	User Interface
URL	Uniform Resource Locator

Chapter 1

Introduction

GraphQL is a query language for APIs and a runtime for fulfilling said queries with existing data [25], which has been gaining popularity in recent years as a choice for building APIs as an alternative to REST.

1.1 Context

The application of GraphQL in applications whose underlying data model is static is pretty well defined with many available resources online. Developers can easily construct the GraphQL schema that describes their data model ahead of time, using, for example, GraphQL schema definition language (SDL).

However, in a context where that is not the case, that is, the model is dynamic and is not known at compile-time, the creation of GraphQL APIs is not straightforward. The GraphQL schema of the API needs to be dynamically generated and updated at runtime to always be in sync with the data model in question, providing an additional layer of complexity to the standard usage of GraphQL technology.

1.2 Motivation

Critical Manufacturing (CM) MES is a leading Manufacturing Execution System, a computerized system used in manufacturing to track and document the transformation of raw materials to finished goods [14], developed by Critical Manufacturing, which, in order to support multiple different industries and usage scenarios, features the ability to extend the pre-existing data model by adding properties and attributes to existing business entities or creating entirely new entities from scratch.

The ability to define a fully dynamic data model also brings the need to have an efficient way of querying existing data. Currently, CM MES offers a REST API to build and execute custom queries against the dynamic data model. However, these queries use a complex and very verbose API which provides a less than ideal developer experience.

1.3 Objectives

This project aims to analyze, implement, and evaluate a proof-of-concept that allows validating the implementation of GraphQL APIs in a context where the underlying data model is not known at compile-time, with further comparative analysis against traditional REST APIs, both in terms of performance, developer experience, and usability, analyzing qualitative and quantitative advantages over the latter.

The work is carried out in the context of the CM MES application, which, by virtue of its business requirements, adaptability, and flexibility, has a dynamic data model that is not necessarily known in advance.

At the end of this dissertation, we can conclude that GraphQL technology is a solution for APIs that operate on top of a dynamic data model. In addition, we can say that a GraphQL API in this scenario can perform similarly to a REST alternative, showing some benefits when it comes to developer experience, but with the possibility of offering some compromises in terms of usability.

1.4 Document Structure

Apart from the introduction, this document has six more chapters.

In Chapter 2, we explore the state of the art regarding GraphQL, explaining the technology, its strengths, and some not yet solved problems. We expose some related work to the context of this dissertation and express its innovative factors. Finally, we describe some GraphQL-based technologies and the choice of which ones to use.

Chapter 3 formally presents the objectives of this dissertation, along with the requirements and functionalities of the work produced. In the end, some questions are raised, whose answers are expected to be gathered from the results obtained.

In Chapter 4 a high-level view of the project involved elements is given, starting with a description of the CM MES application and some of its components. After this, the architecture of the developed solution is presented and explained.

Chapter 5 further details the technologies and methodologies used to produce the dissertation work, along with describing the most critical implementation details.

After all the work has been done, Chapter 6 focuses on analyzing the results obtained and answering the main questions presented earlier.

Finally, in Chapter 7 we finish with some conclusions about the work and results produced and propose some ideas for further developments.

Chapter 2

GraphQL for Dynamic Data Models

Since GraphQL is still a somewhat new technology for constructing APIs and without a clearly defined way to be used in scenarios of dynamic data models, the amount of available work and knowledge in this context is still pretty limited.

However, there have been some advances that may help prove that this use case is possible. The existence of new technologies that aid the usage of GraphQL for API development, from GraphQL libraries to frameworks, can make this more accessible than ever before.

2.1 GraphQL

2.1.1 Description

GraphQL was created in 2012 by Facebook and open-sourced in 2015, and it's becoming increasingly more popular in the developer world. According to the 2020 State Of JavaScript Report, GraphQL usage rose from 6% in 2016 to 47% in 2020, meaning that almost 1 in 2 respondents had been in contact with the technology before [20].

GraphQL is a query language and execution engine designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions. It should not be confused as a programming language capable of arbitrary computation but rather understood as a language used to make requests to application services and does not mandate a particular programming language or storage system for application services that implement it [24].

It works as an option for building modern APIs and has become a popular alternative to REST. Developers create a GraphQL service for their backend that will consume GraphQL queries sent by a client, which then validates and executes them, returning a well-formed response [24]. A diagram representing what a standard GraphQL based architecture looks like can be seen in Figure 2.1.

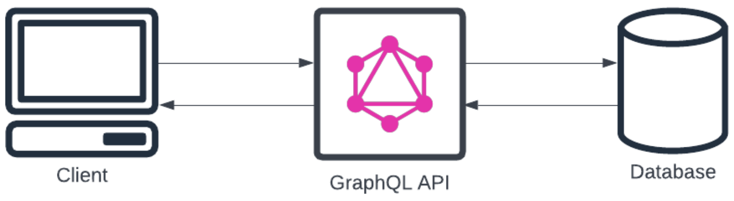


Figure 2.1: Typical GraphQL based architecure

2.1.2 Schema and API Example

To better illustrate how a typical GraphQL service would work, let us assume we have an application whose database model mimics a blog. We will have two entities, *Post* and *User*, with each *Post* containing a title, an optional description, and information about the *User* that created it. In contrast, a *User* only stores information about their username.

In order to build a GraphQL API, developers construct a GraphQL schema that describes the set of possible data that clients can query through it. This schema works as a mapping of the application's underlying data model to GraphQL schema language. The most common way of annotating this schema is with a schema-first approach, where developers manually write out their GraphQL schema using plain GraphQL SDL. Figure 2.2 shows how the previously mentioned scenario could be translated into GraphQL SDL, following this strategy.

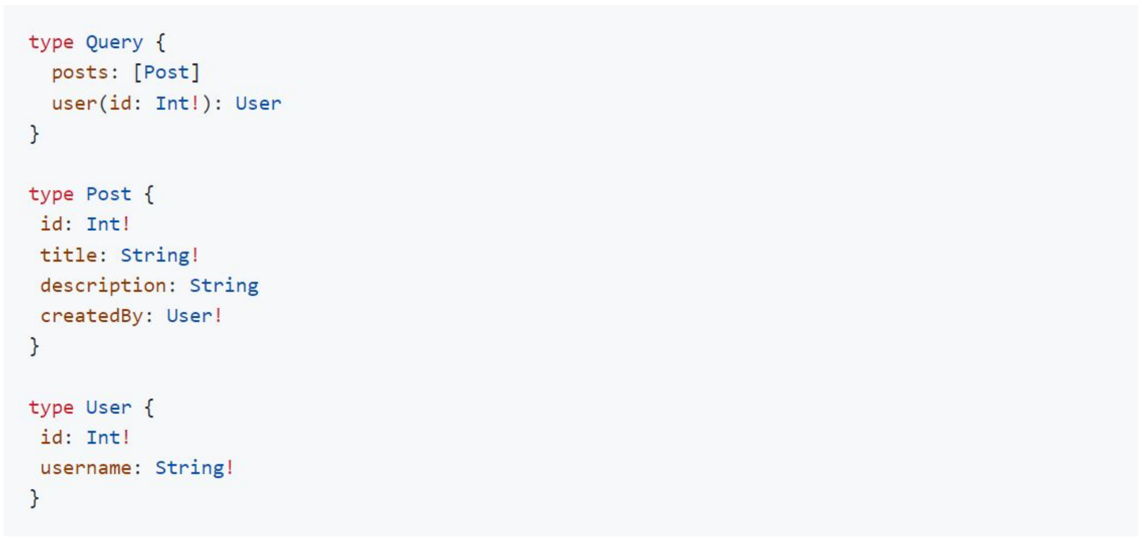
The existing *Query* type references the query root operation type, which must be provided in every GraphQL service and defines the set of possible query operations to the API, similar to GET requests in REST. In this case, we assume two possible queries can be made: *posts*, which returns an array of all *Post* objects, and *user(id: Int!)*, which will return a *User* with the specified *id* passed as an argument.

Two other operation types are available, namely mutations and subscriptions. Mutations, for example, is an operation type similar to the previously mentioned queries that lets us modify server-side data just like a POST request in REST would. A mutation operation could, for example, allow the creation of a new *Post* instance in our database through a request to the GraphQL API. In any case, the work done in this dissertation focuses only on querying a system, so other operation types will not be explored.

The *Post* and *User* types present in the schema are referred to as object types and represent some kind of object existent in the system. An object type is composed of a set of fields, which can be of the following sorts:

- **Object type**
  An object type field can be another object type itself, such as the *User* field inside of *Post*.

- **Scalar type**
  A scalar type represents some concrete data and is the leaf of a GraphQL query. GraphQL



```
type Query {
  posts: [Post]
  user(id: Int!): User
}

type Post {
 id: Int!
 title: String!
 description: String
 createdBy: User!
}

type User {
 id: Int!
 username: String!
}
```

Figure 2.2: Example of GraphQL schema written in GraphQL SDL

comes with a set of default scalar types out of the box, which are *Int*, *Float*, *String*, *Boolean*, and *ID*. An example of it is the *title* field of the *Post* object type.

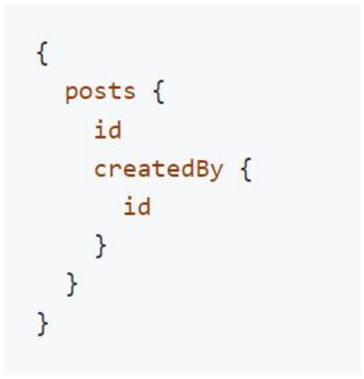
- **Enumeration type**
  An enumeration type is a special kind of scalar that is restricted to a particular set of allowed values.

In addition to constructing a GraphQL schema, the other step needed is to make the service aware of how to fetch the necessary data. That is made through the resolvers, which are functions written in the service programming language that describe how to populate the data returned by a given query. Assuming that the GraphQL API has been created and is up and running with the mentioned schema and all necessary resolvers have been set up by a developer, the service is ready to receive and fulfill GraphQL requests.

To demonstrate how GraphQL queries work, suppose that we want to query a list of all existing *Post* objects in the system, fetching only their corresponding *id* and the *id* of the *User* that created it. We can construct the GraphQL request seen in Figure 2.3 and send it to our GraphQL API, which produces and returns the resulting data in JSON reflected in Figure 2.4. If we rather, for example, want to query the *username* of the *User* with an *id* equal to 1, we instead send a request as shown in Figure 2.5 and obtain the response seen in Figure 2.6.

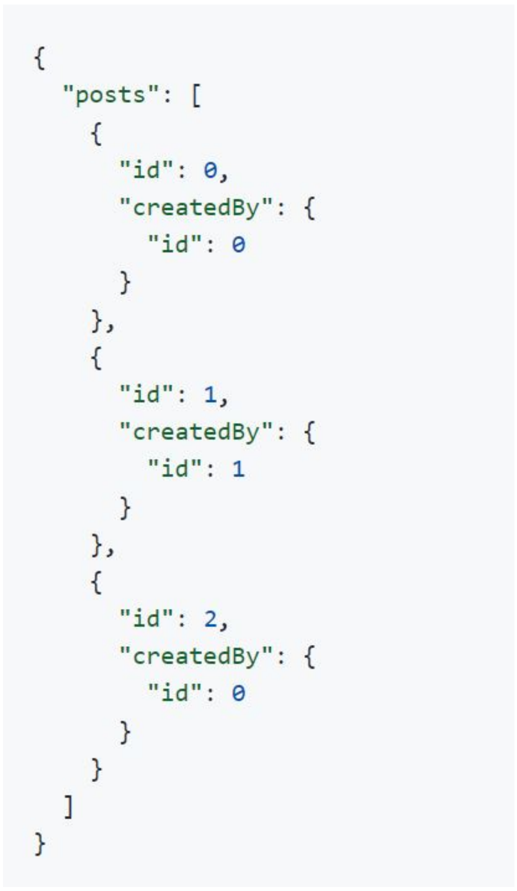
2.1.3 Factors and Forces

Unlike REST APIs, GraphQL APIs expose a single endpoint, capable of fulfilling any GraphQL request due to GraphQL APIs being organized in terms of types and fields instead of endpoints. As such, GraphQL focuses on giving clients the ability to ask for exactly the data they need from a server in a single request [25].



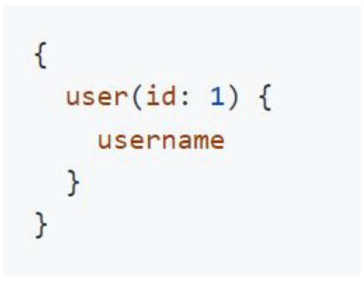
```
{
  posts {
    id
    createdBy {
      id
    }
  }
}
```

Figure 2.3: GraphQL query to fetch the *ids* of *Post* objects and the *User* that created it



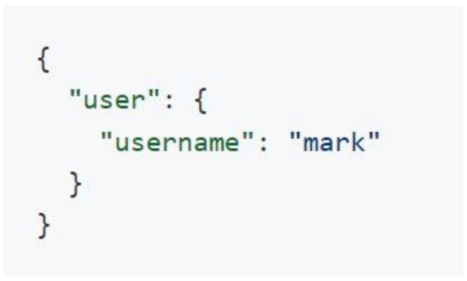
```
{
  "posts": [
    {
      "id": 0,
      "createdBy": {
        "id": 0
      }
    },
    {
      "id": 1,
      "createdBy": {
        "id": 1
      }
    },
    {
      "id": 2,
      "createdBy": {
        "id": 0
      }
    }
  ]
}
```

Figure 2.4: JSON response to Figure 2.3 query



```
{
  user(id: 1) {
    username
  }
}
```

Figure 2.5: GraphQL query to fetch the *username* of *User* with *id* equal to 1



```
{
  "user": {
    "username": "mark"
  }
}
```

Figure 2.6: JSON response to Figure 2.5 query

```
{
  posts {
    title
    description
  }
}
```

Figure 2.7: GraphQL query to fetch the *title* and *description* of *Post* objects

```
{
  "posts": [
    {
      "title": "GraphQL Latest Release",
      "description": "October 2021 spec released"
    },
    {
      "title": "Facebook News",
      "description": "Facebook is now Meta"
    },
    {
      "title": "GraphQL new Prerelease in the works!",
      "description": null
    }
  ]
}
```

Figure 2.8: JSON response to Figure 2.7 query

While with a classic REST architecture fetching data to populate a webpage might take several roundtrips due to calling different endpoints for getting specific pieces of information, with a GraphQL architecture, everything can be fetched in a single request. This feature can significantly improve the performance of an application due to less time being wasted on queries to the backend. On top of this, the developer experience can be improved since GraphQL is inherently self-documenting. Hence, clients quickly know what specific information can be queried and how.

Let us assume once again the example mentioned in the previous section, but this time we want to retrieve the list of all *Post* objects' *titles* and *descriptions*. We will still use the same *posts* query, but now we structure it a little bit differently. We should only include the *title* and *description* fields in the GraphQL query since these are the only pieces of data that matter to us this moment, as shown in Figure 2.7. After executing this query against our API, we will obtain the response seen in Figure 2.8.

This is the true power of GraphQL, fixing the under fetching and over fetching of data since we can precisely retrieve the information we want from a server with just a single request to the same endpoint. On the other hand, with a REST API, we would need to constantly create new endpoints to cover new use cases or end up getting more or less data than we really wanted.

2.1.4 Not Yet Solved Problems

The usage of GraphQL for API development is pretty standardized and has excellent documentation. However, for use cases where the API's GraphQL schema can't be easily constructed, such as when an application's underlying data model isn't known at compile-time, developing a GraphQL API becomes rather complex.

A schema-first approach to constructing the GraphQL schema can't be followed in these circumstances since a developer does not have the necessary information to be able to annotate it in GraphQL SDL ahead of time. In this scenario, a code-first approach must be pursued.

A code-first approach allows for dynamic schema generation since the schema of the GraphQL API is built programmatically through code. This approach might also be followed when developing GraphQL APIs since it offers more modularity and efficiency for maintaining a GraphQL schema updated in large software projects, decreasing the number of conflicts and inconsistencies.

Even though we know that dynamic schema generation is possible, the application of GraphQL for dynamic data models remains uncertain since the GraphQL schema of the API must also be constantly updated during runtime to reflect the changes happening in the underlying data model. Also, the advantages of choosing a GraphQL API over a traditional REST API in terms of performance, developer experience, and usability in a scenario like this one haven't yet been adequately studied and analyzed, remaining doubts about if going with a GraphQL API might be viable.

2.2 Related Work

The application of GraphQL for dynamic data models is still a work in progress, with no clear solution available. The most crucial aspect needed to achieve this use case is the ability to build GraphQL schemas automatically according to some metadata instead of having it previously declared. Although this use case is uncommon, there have been some efforts to incorporate GraphQL in scenarios with automatic schema generation.

The first example of this comes from the paper *GraphQL Schema Generation for Data-Intensive Web APIs*, where a semantic-based approach to generating GraphQL schemas is presented. The authors defined an RDF-formalized semantic metamodel for GraphQL schema, designated GQL, to help represent a dataset. First, a manual annotation of the dataset schema with the GQL metamodel is needed. After that, an automatic generation of a GraphQL schema from the annotated RDF ontology is possible, as well as an automatic generation of a GraphQL service that exposes the generated schema and the available data related to the annotated ontology [6].

The second example of automatic schema generation comes from Contentful, an API-first content management platform to create, manage and publish content on any digital channel. Contentful faced the problem of having to create a GraphQL API customized to each one of its users. Since every Contentful user has their own content model, consisting of varying content types, and with no two users' content models being the same, each user's data model is used to generate a GraphQL schema tailored to them [3].

Still, the application of GraphQL in a context where dynamic schema generation and further schema patching during runtime is needed, due to the nature of an underlying dynamic data model, hasn't been significantly explored.

2.3 Innovative Factors

GraphQL APIs are typically based on a statically defined schema that maps to an underlying static data model. This use case is pretty straightforward to cover. Developers construct their GraphQL schema ahead of time and can write it in a schema-first approach in plain GraphQL SDL or even follow a code-first approach if they want. The number of available resources and documentation online is massive, so every developer should be able to quickly learn and construct a GraphQL API for this scenario.

However, due to the nature of the CM MES dynamic data model, GraphQL schemas will have to be dynamically generated and updated in runtime, providing an additional layer of complexity to the standard usage of GraphQL technology.

2.4 Existing Technologies

Since GraphQL is just a communication pattern, many tools have been created to help developers use the technology in various situations. Such examples of tools are GraphQL libraries that make it possible to develop GraphQL applications in a particular programming language and UI clients that enable developers to easily construct and test GraphQL queries, making the developing experience seamless.

2.4.1 Characterization

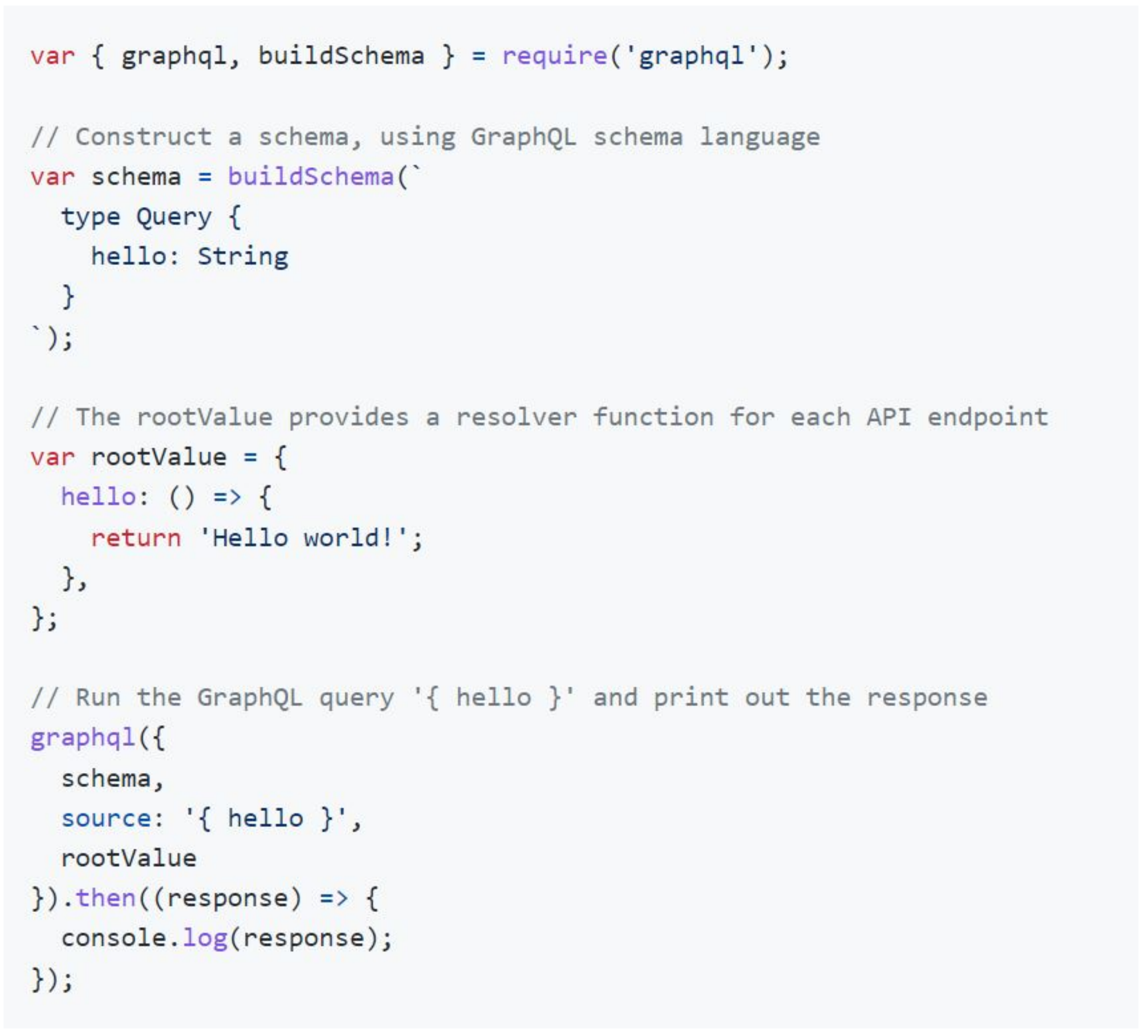
2.4.1.1 Libraries

One of the ways available to develop GraphQL APIs with a specific programming language is through GraphQL libraries. These libraries exist for almost every modern programming language, such as JavaScript, Python, or Java, so developers can freely develop their GraphQL services in a language of their choice.

Let us take JavaScript as an example. When developing a backend with this programming language, most developers use Node.js[1], a JavaScript runtime built on Chrome's V8 JavaScript engine. To incorporate GraphQL in their Node.js project, one can use GraphQL.js, the JavaScript reference implementation for GraphQL [23], and with a simple command, install this package through npm[2]. In just a few lines of code, we can create a script that executes a GraphQL query and outputs its result to the console, as seen in Figure 2.9. Obviously, this program isn't very useful, but it shows how easy it is to start using GraphQL with a given programming language.

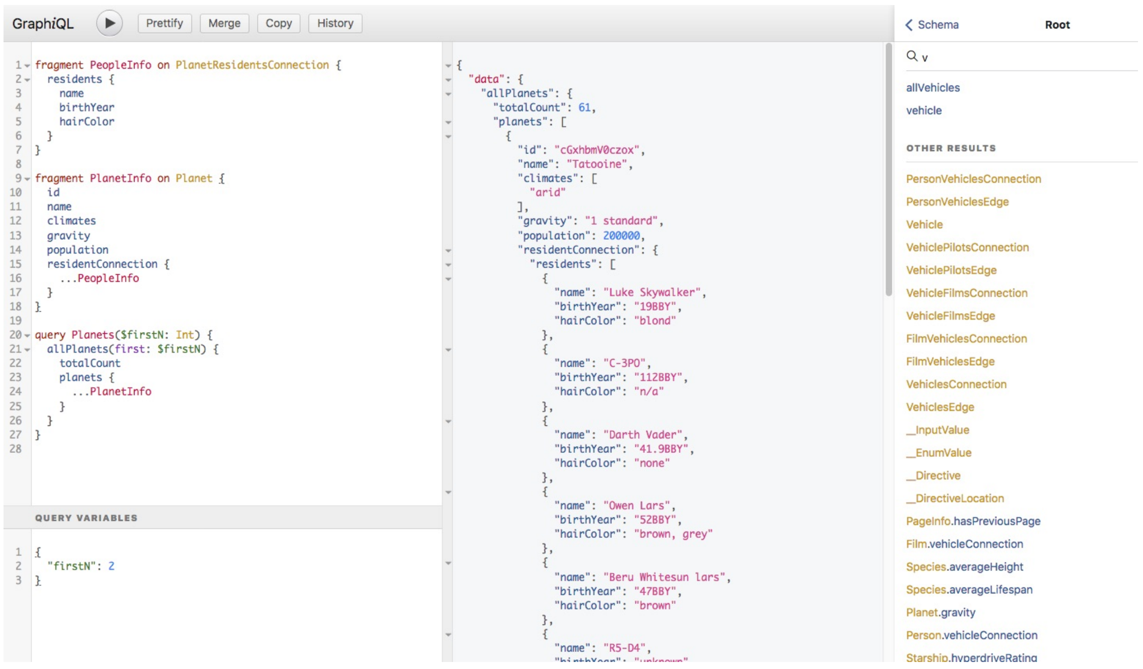
On top of enabling the creation of GraphQL APIs, these libraries provide a bunch of valuable features like programmatically building a schema, the so-called code-first approach that allows the creation of dynamic schemas contrary to the schema-first method of defining the schema for the GraphQL service beforehand in plain GraphQL SDL.

[1] `https://nodejs.org/en/`
[2] `https://www.npmjs.com/`



```
var { graphql, buildSchema } = require('graphql');

// Construct a schema, using GraphQL schema language
var schema = buildSchema(`
  type Query {
    hello: String
  }
`);

// The rootValue provides a resolver function for each API endpoint
var rootValue = {
  hello: () => {
    return 'Hello world!';
  },
};

// Run the GraphQL query '{ hello }' and print out the response
graphql({
  schema,
  source: '{ hello }',
  rootValue
}).then((response) => {
  console.log(response);
});
```

Figure 2.9: GraphQL JavaScript hello world script



```
fragment PeopleInfo on PlanetResidentsConnection {
  residents {
    name
    birthYear
    hairColor
  }
}

fragment PlanetInfo on Planet {
  id
  name
  climates
  gravity
  population
  residentConnection {
    ...PeopleInfo
  }
}

query Planets($firstN: Int) {
  allPlanets(first: $firstN) {
    totalCount
    planets {
      ...PlanetInfo
    }
  }
}
```

```
{
  "firstN": 2
}
```

```
{
  "data": {
    "allPlanets": {
      "totalCount": 61,
      "planets": [
        {
          "id": "cGxhbmV0czox",
          "name": "Tatooine",
          "climates": [
            "arid"
          ],
          "gravity": "1 standard",
          "population": 200000,
          "residentConnection": {
            "residents": [
              {
                "name": "Luke Skywalker",
                "birthYear": "19BBY",
                "hairColor": "blond"
              },
              {
                "name": "C-3PO",
                "birthYear": "112BBY",
                "hairColor": "n/a"
              },
              {
                "name": "Darth Vader",
                "birthYear": "41.9BBY",
                "hairColor": "none"
              },
              {
                "name": "Owen Lars",
                "birthYear": "52BBY",
                "hairColor": "brown, grey"
              },
              {
                "name": "Beru Whitesun lars",
                "birthYear": "47BBY",
                "hairColor": "brown"
              },
              {
                "name": "R5-D4",
```

Figure 2.10: GraphiQL IDE application screen

2.4.1.2 GUIs

While developing a GraphQL application, a GraphQL UI client can be of great use. These GUIs help developers debug their GraphQL applications, providing an easy-to-understand IDE with features such as exploring the API schema, conveniently seeing its documentation, and quickly constructing and testing GraphQL queries.

One of the most straightforward and widely used GraphQL GUIs is GraphiQL, an official project under the GraphQL Foundation. GraphiQL is an interactive in-browser GraphQL IDE implemented in React and is available for integration in most common programming languages, such as Javascript or C#/.NET [22].

In Figure 2.10, we can see what a typical GraphiQL window looks like. On the left-side panel is the query constructor sub-window that lets us build a GraphQL query and provides useful features like auto-completion, auto-formatting, and syntax-highlighting. After having a constructed query, we can test it by clicking on the "play" icon in the window's top-left corner, and its result is presented on the middle-side panel. Finally, on the right-side panel, we can see the schema exploration and documentation tab that lets us explore the schema fields or search for specific schema elements.

2.4.1.3 Other

Some other tools that help developers build applications with GraphQL are available.

Dgraph, for instance, is a new native GraphQL database built from the ground up to manage data natively in graphs. It offers a specification-compliant GraphQL endpoint without needing an additional translation layer in the tech stack. GraphQL is natively executed within the core of Dgraph itself, and there isn't the need to code GraphQL resolvers [5].

Another option when developing GraphQL APIs is to use a GraphQL-centered framework. An example of this type of tool is DGS, a GraphQL server framework for Spring Boot, developed by Netflix in 2019 and open-sourced in 2020. This framework is built on top of the graphql-java library and includes features such as an annotation-based Spring Boot programming model, a test framework for writing query tests as unit tests, and integration with Spring Security [18].

However, these types of tools aren't relevant to the work done in this dissertation, and as such further analysis of these is not considered.

2.4.2 Choice

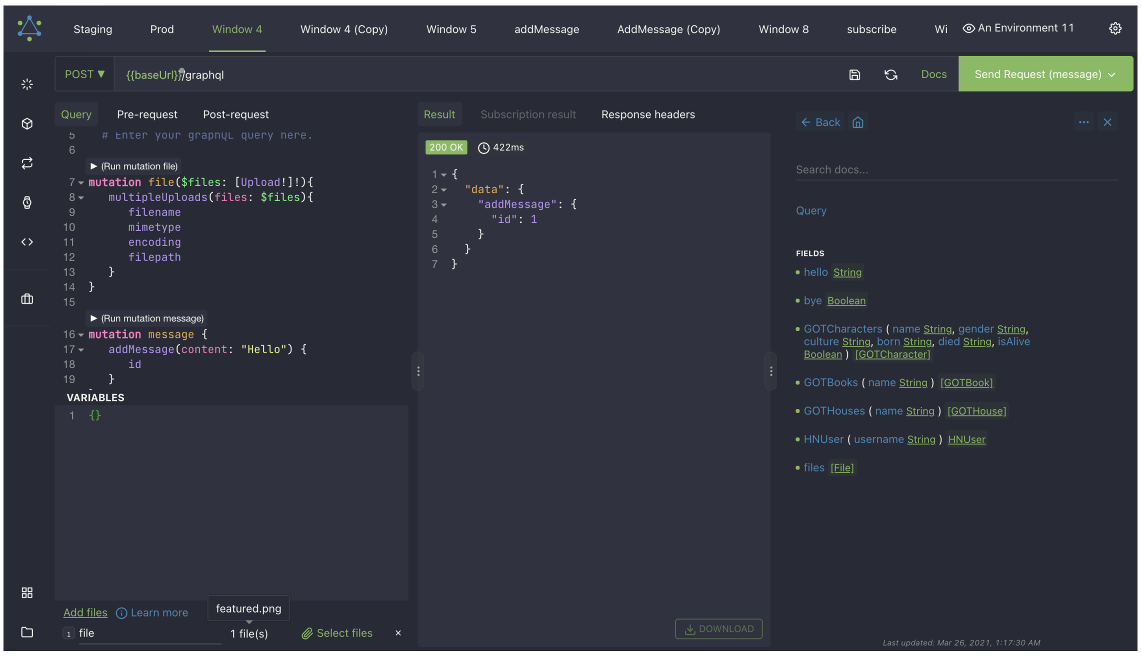
Since CM MES is built on the .NET framework and with C# as its chosen programming language, it also makes sense to pick a C#/.NET centered GraphQL library so to make any possible integration with the CM MES codebase easier.

There are two main GraphQL libraries available for C#/.NET: GraphQL.NET[3] and Hot Chocolate[4]. The first is the most widely used library for the referred language and had its first release in 2015 . The second one had its first release only in 2018 but has recently gained popularity in the developer world due to its increased performance, great support, and wide range of features compared to the former.

Picking Hot Chocolate over GraphQL.NET might seem like the obvious choice when building a GraphQL application for C#/.NET. Still, the flexibility and rawness of GraphQL.NET make it crucial for the use case addressed in this dissertation of building completely dynamic schemas at runtime. Even though Hot Chocolate has some support for creating dynamic schemas, it is still not abstract and modular enough to let us accomplish the same things as GraphQL.NET, making the latter the chosen library to produce this work.

On top of selecting a library to develop the GraphQL application, a GraphQL GUI was also used to help debug and validate the application. The chosen GUI was Altair, a GraphQL Client similar to the previously mentioned GraphiQL but with extending features such as advanced schema documentation search and automatic schema refreshing , along with a richer and more visually appealing interface, which can be seen in Figure 2.11 [1].

[3]`https://github.com/graphql-dotnet/graphql-dotnet`
[4]`https://github.com/ChilliCream/hotchocolate`



```
  # Enter your graphQL query here.

▶ (Run mutation file)
mutation file($files: [Upload!]!){
  multipleUploads(files: $files){
    filename
    mimetype
    encoding
    filepath
  }
}

▶ (Run mutation message)
mutation message {
  addMessage(content: "Hello") {
    id
  }
```

VARIABLES

```
{}
```

200 OK 422ms

```
{
  "data": {
    "addMessage": {
      "id": 1
    }
  }
}
```

Query

FIELDS

- hello String
- bye Boolean
- GOTCharacters (name String, gender String, culture String, born String, died String, isAlive Boolean) [GOTCharacter]
- GOTBooks (name String) [GOTBook]
- GOTHouses (name String) [GOTHouse]
- HNUser (username String) HNUser
- files [File]

Last updated: Mar 26, 2021, 1:17:30 AM

Figure 2.11: Altair GraphQL Client application screen

Chapter 3

Requirements and Functionalities

After presenting the state of the art around the faced problem, we define the goals to be achieved with this dissertation, along with the functional and non-functional requirements of the proposed solution, and finally, a list of questions to be answered by the work conducted.

3.1 Dissertation Objectives

The problem to be explored in this dissertation lies in the application of GraphQL for dynamic data models, that is, a model which is not known at compile-time and may undergo transformations during the running phase of the program.

From this setting, we can infer two main objectives to be accomplished:

1. The validation of the applicability of GraphQL technology for the previously mentioned scenario. This should be achieved by implementing a proof-of-concept that allows confirming the possible use of a GraphQL API in a context where an application's underlying data model is not known at compile-time. The proof-of-concept is to be developed in the CM company facilities and in tandem with their CM MES product, which possesses a dynamic data model due to business requirements.

2. After having a viable proof-of-concept, develop a comparative analysis of the new GraphQL API against the existing CM MES REST API in terms of performance, developer experience, and usability. Finally, conclude the qualitative and quantitative advantages of using a GraphQL API over a traditional REST API for querying an application that features a dynamic data model.

3.2 Functional Requirements

The functional requirements define what a system should do and describe its functionality to a user.

It should be possible to query CM MES with GraphQL as well as paginate and filter the results of a query. In addition, there should be a way to properly inspect the CM MES GraphQL schema, which must be constantly updated according to the underlying data model.

Each functional requirement is identified by a unique *ID* and *Name*, its *Priority* for being present, a brief *Description* of what it provides, and the *Motivation* for its consideration.

Tables 3.1 to 3.5 display the list of functional requirements the system should meet.

Table 3.1: REQ.001 :: Query CM MES with GraphQL

REQ.001	**Query CM MES with GraphQL**
Priority	Essential.
Description	The system must have the ability to query with GraphQL the CM MES dynamic data model *Entity Types* and corresponding *Properties*.
Motivation	So to be able to validate the application of GraphQL for dynamic data models and provide a new way to query the CM MES data model in addition to the existing REST API.

Table 3.2: REQ.002 :: Keep GraphQL schema updated

REQ.002	**Keep GraphQL schema updated**
Priority	Essential.
Description	The system must have its GraphQL schema updated according to the CM MES data model at all times.
Motivation	So that the GraphQL schema of the API always represents the most current version of the dynamic data model, and its accurate querying is possible and valid.

Table 3.3: REQ.003 :: Paginate GraphQL query results

REQ.003	**Paginate GraphQL query results**
Priority	High.
Description	The system should have the ability to paginate the results of a GraphQL query.
Motivation	So to be able to retrieve records from the CM MES data model at given intervals, just like the current REST API of CM MES, which allows paginations of results.

Table 3.4: REQ.004 :: Filter GraphQL query results

REQ.004	**Filter GraphQL query results**
Priority	High.
Description	The system should have the ability to filter the results of a GraphQL query.
Motivation	So to be able to retrieve records from the CM MES data model that match given criteria, just like the current REST API of CM MES, which allows filtering of results.

Table 3.5: REQ.005 :: Explore GraphQL schema

REQ.005	**Explore GraphQL schema**
Priority	Medium.
Description	The system should provide a way to explore the GraphQL schema of the CM MES data model.
Motivation	So to be able to easily understand the possible queries that can be constructed and executed by the GraphQL API.

3.3 Non-Functional Requirements

The non-functional requirements are requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet [11].

Tables 3.6 to 3.10 display the list of non-functional requirements the system must meet. It should be highly performant, maintainable, extendable, use software approved for commercial use, loggable and secure.

Just like in the previous list of functional requirements, each non-functional requirement is also identified with an *ID* and *Name*, *Priority* type, brief *Description*, and *Motivation*.

Table 3.6: REQ.006 :: Highly Performant

REQ.006	**Highly Performant**
Priority	Essential.
Description	The GraphQL API should display high performance.
Motivation	So to be able to become a viable alternative to the current REST API, the new GraphQL API should have great performance.

Table 3.7: REQ.007 :: Maintainable

REQ.007	**Maintainable**
Priority	Essential.
Description	The GraphQL API should be easily maintainable by other developers.
Motivation	So to be able to remain a functional API in the future, the GraphQL API should be appropriately documented to be easily maintainable by other developers.

Table 3.8: REQ.008 :: Extensible

REQ.008	**Extensible**
Priority	Essential.
Description	The GraphQL API should be easily expandable by other developers.
Motivation	So to be able to incorporate new features and enhancements in the future, the GraphQL API should be constructed with future expandability by other developers in mind.

Table 3.9: REQ.009 :: Use software approved for commercial use

REQ.009	**Use software approved for commercial use**
Priority	Essential.
Description	The GraphQL API should use third-party dependencies that are approved for commercial use.
Motivation	So to be able to have the GraphQL API utilized within CM MES, all third-party dependencies should be approved for commercial use since CM MES is a paid software.

Table 3.10: REQ.010 :: Loggable

REQ.010	**Loggable**
Priority	Essential.
Description	The GraphQL API should output logs of its execution.
Motivation	So to be able to be easily debuggable and for its behavior to be better understood, the GraphQL API should output logs that document its execution.

Table 3.11: REQ.011 :: Secure

REQ.011	**Secure**
Priority	Essential.
Description	The GraphQL API should be secure from outside attacks.
Motivation	So to be able not to compromise the internal system of CM MES, the GraphQL API should be secure from common software attacks like, for example, SQL Injection.

3.4 Questions To Be Answered

After having finished the dissertation work and produced the final results, it is hoped to be able to answer some questions, such as:

- Is it possible to build a GraphQL API for an application with an underlying dynamic data model?
- How does a GraphQL API compare with a REST API in terms of performance in a scenario where an application has an underlying dynamic data model?
- How does a GraphQL API compare with a REST API in terms of developer experience in a scenario where an application has an underlying dynamic data model?
- How does a GraphQL API compare with a REST API in terms of usability in a scenario where an application has an underlying dynamic data model?
- What are the qualitative and quantitative advantages of using a GraphQL API over a REST API in an application with an underlying dynamic data model?

Chapter 4

Architecture

This chapter describes the architecture and gives a high-level view of the project components involved.

First, an overview of the CM MES software is made, so to better contextualize the work done in this dissertation. After that, we take a deeper look at the CM MES elements that play a role in the work produced, starting with its dynamic data model, which is the central aspect of the project. Following, we explore the CM MES Queries feature and current REST API and why it offers a less than ideal developer experience. Lastly, we look at the CM MES message bus, the application's own message broker with a publish-subscribe messaging pattern.

At the end of the chapter, we delve into the developed GraphQL middleware architecture and its fit with the existing CM MES components.

4.1 CM MES Software

To better understand how the components mentioned in the following sections 4.2 and 4.3 fit in the bigger picture, a depiction of the CM MES software should first be made.

CM MES is a leading manufacturing execution system, which are computerized systems used in manufacturing to track and document the transformation of raw materials to finished goods. These types of systems provide information that helps manufacturing decision makers understand how current conditions on the plant floor can be optimized to improve production output [14].

It has been developed by the Critical Manufacturing company since 2009, with its first version being officially released in late 2010. It is sold as licensed software that manufacturing companies buy and integrate with their production lines, whose operands in charge can access through any modern web browser since it runs as a web page at a given address. In Figure 4.1 we can see what the CM MES landing page looks like after a user has successfully logged in with their credentials.

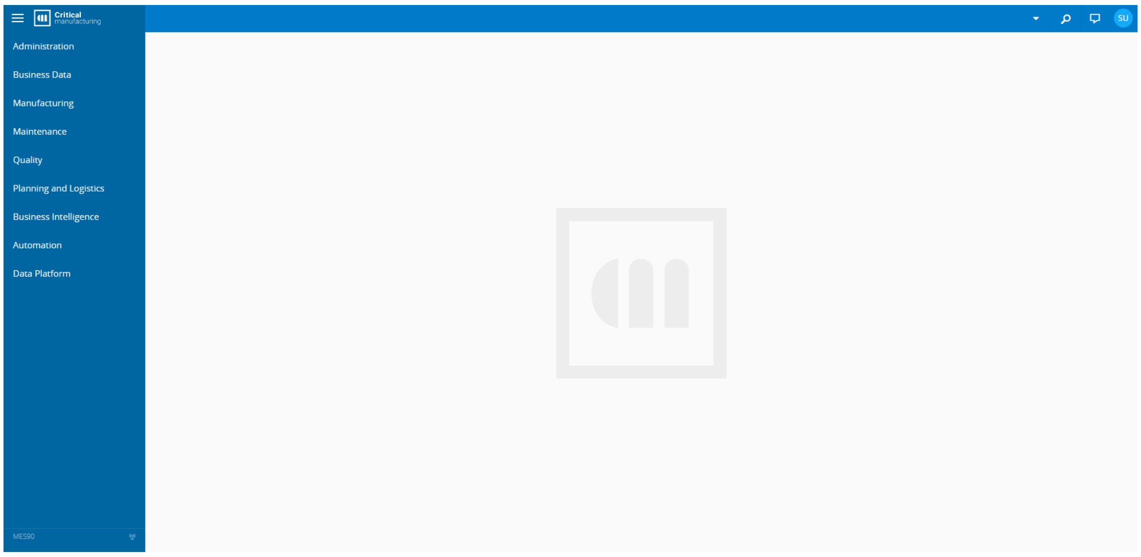


Figure 4.1: CM MES landing page

The CM MES application complete tech stack architecture can be seen in Figure 4.2, with it being composed of the following three main parts:

- **Frontend**
  CM MES uses Angular for the frontend, a modern framework for building mobile and desktop web applications, created and released by Google in 2016. The usage of this technology for the CM MES GUI provides faster development than pure JavaScript and is a popular choice among frontend developers.

- **Backend**
  The backend is written in the C# programming language, on top of Microsoft's .NET framework, first released in 2002.

- **Database**
  For its storage needs, CM MES uses Microsoft SQL Server, a relational database management system launched in 1989 that allows for straightforward integration with the .NET backend.

CM MES possesses a huge number of features, with new ones being released with every new major version of the software. The work done in this dissertation was developed with version 9 of CM MES and deals with only a single specific feature denominated Queries, whose page is shown in Figure 4.3. This feature allows users to create and execute custom queries for the CM MES data model *Entity Types*, allowing complex filtering based on given *Entity Type Properties*.

An explanation of what the mentioned *Entity Types* and *Entity Type Properties* are is done in section 4.2.1, as well as a more in-depth explanation of this Queries features in section 4.2.2.

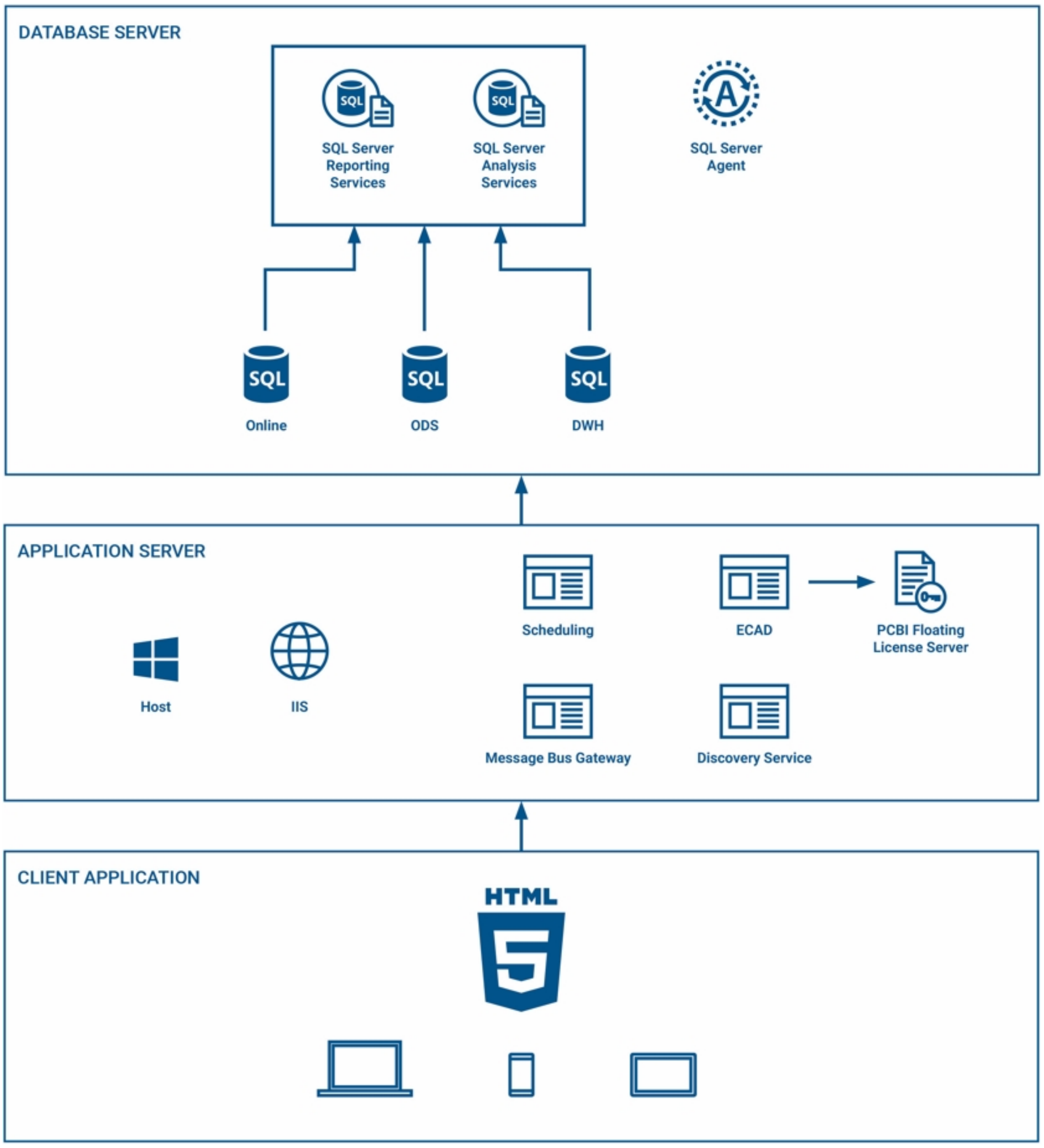
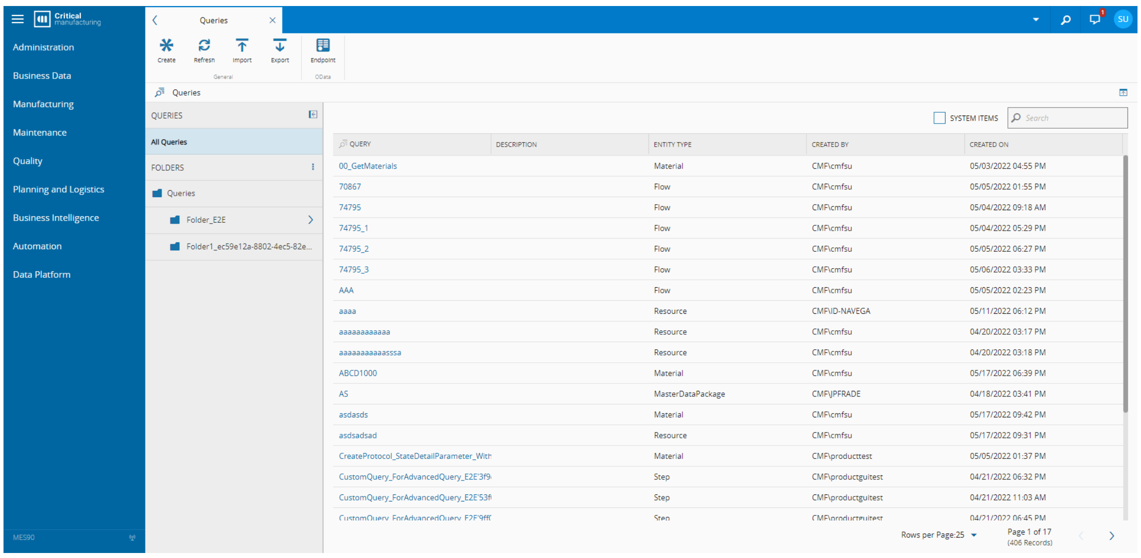


Figure 4.2: CM MES tech stack architecture



QUERY	DESCRIPTION	ENTITY TYPE	CREATED BY	CREATED ON
00_GetMaterials		Material	CMF\cmfsu	05/03/2022 04:55 PM
70867		Flow	CMF\cmfsu	05/05/2022 01:55 PM
74795		Flow	CMF\cmfsu	05/04/2022 09:18 AM
74795_1		Flow	CMF\cmfsu	05/04/2022 05:29 PM
74795_2		Flow	CMF\cmfsu	05/05/2022 06:27 PM
74795_3		Flow	CMF\cmfsu	05/06/2022 03:33 PM
AAA		Flow	CMF\cmfsu	05/05/2022 02:23 PM
aaaa		Resource	CMF\ID-NAVEGA	05/11/2022 06:12 PM
aaaaaaaaaaaa		Resource	CMF\cmfsu	04/20/2022 03:17 PM
aaaaaaaaaaaasssa		Resource	CMF\cmfsu	04/20/2022 03:18 PM
ABCD1000		Material	CMF\cmfsu	05/17/2022 06:39 PM
AS		MasterDataPackage	CMF\JPFRADE	04/18/2022 03:41 PM
asdasds		Material	CMF\cmfsu	05/17/2022 09:42 PM
asdsadsad		Resource	CMF\cmfsu	05/17/2022 09:31 PM
CreateProtocol_StateDetailParameter_With		Material	CMF\producttest	05/05/2022 01:37 PM
CustomQuery_ForAdvancedQuery_E2E'3f9		Step	CMF\productguitest	04/21/2022 06:32 PM
CustomQuery_ForAdvancedQuery_E2E'53f		Step	CMF\productguitest	04/21/2022 11:03 AM
CustomQuery_ForAdvancedQuery_E2E'9ff(		Step	CMF\productguitest	04/21/2022 06:45 PM

Rows per Page:25
Page 1 of 17
(406 Records)

Figure 4.3: CM MES Queries page

4.2 CM MES Architecture

CM MES is a complex piece of software composed of a vast amount parts. Three of its components, however, take an important role in this dissertation work: its dynamic data model, its Queries feature and associated REST API, and its message bus.

4.2.1 Data Model

The CM MES data model has a particular trait which is being dynamic. The tables representing the application schema and storing the underlying data in the databases are not the same throughout the program's lifetime. During runtime, the program can, for example, dynamically build new tables which represent new *Entity Types* created by a user.

CM MES database model is divided into two sections, the Static Model and the Dynamic Model.

4.2.1.1 Static Model

The Static Model handles the core information for the CM MES system. Its schema never undergoes any changes and is always represented by the same pre-defined tables. It is composed of many different sub-parts, with its main one also being named *static model*.

The *static model* sub-schema is the most significant part of the Static Model, possessing the largest number of tables and most complexity of all, as seen in the diagram of Figure 4.4. For the *static model* sub-schema, all tables also have a corresponding history table with the same name, with the suffix *Hst*, and with exactly the same structure. These, however, are not present in the mentioned diagram for simplicity purposes.

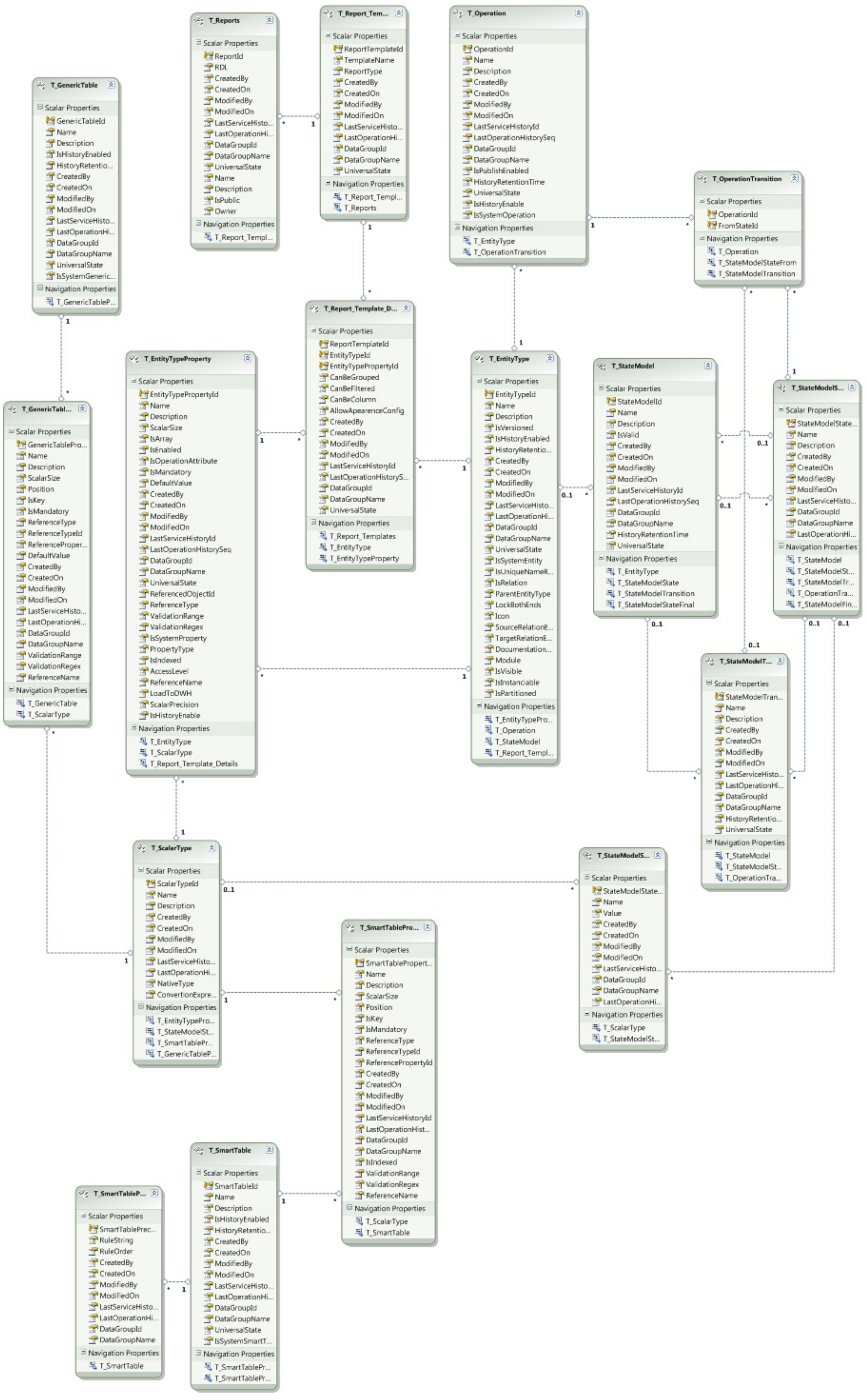


Figure 4.4: CM MES Static Model *static model* sub-schema diagram

This first section of the database model contains an overwhelming number of tables, and it can be challenging to understand how all of them fit together as a whole. Nevertheless, we should only focus on two of them for this dissertation work, namely *T_EntityType* and *T_EntityTypeProperty* from the specified *static model* schema.

The first mentioned table stores the metadata of all *Entity Types* created in the CM MES system, with information such as their name and description. An *Entity Type* represents a given object associated with a manufacturing process. For example, one core *Entity Type* is *Facility*, which might represent a factory, a production line, a distribution center, or a warehouse.

The second mentioned table stores the metadata of all existing *Entity Type Properties*, with information such as their name and scalar type. An *Entity Type Property* represents, as its name implies, some property belonging to an *Entity Type*. That property can be represented by a simple scalar, such as a string or integer, or even be a reference to a completely different *Entity Type* in the system.

4.2.1.2 Dynamic Model

The Dynamic Model handles the instances created based on the Static Model metadata. These instances can be either *Entity Types*, *Generic Tables*, or *Smart Tables*, and can be created, deleted, or modified during the program's lifetime.

Like the Static Model, this one can also be divided into different parts, but only one of them is of interest to us, the *entities* sub-schema. For each *Entity Type*, a set of tables are generated to provide a base persistency schema for the *Entity Type* instances data, as seen in Figure 4.5.

Once again, all this information might seem confusing, but there is only one table that we should focus on from the Dynamic Model for this dissertation, which is *T_[name]* from the mentioned *entities* sub-schema.

This table stores all the instances created for a given *Entity Type*, whose name should replace *[name]* to get the proper table name. For example, let us say we want to query all instances of the *Factory Entity Type*, we would access the table whose name equals to *T_Factory*. Each row of this table would correspond to a single instance of a *Factory* object, and the columns of it are generated according to the *Factory Entity Type Properties* stored in the previously mentioned *T_EntityTypeProperty* table from the Static Model *static model* sub-schema.

4.2.2 Queries and REST API

Queries is a feature inside the CM MES application that allows inquiring the system for a given *Entity Type*, returning the records that match specific criteria. Clicking on the Queries page (Figure 4.3) will display the Queries folder structure that contains all the Queries created up until date in the system.

Selecting a specific Query brings up the Query page seen in Figure 4.6, which is divided into two sections:

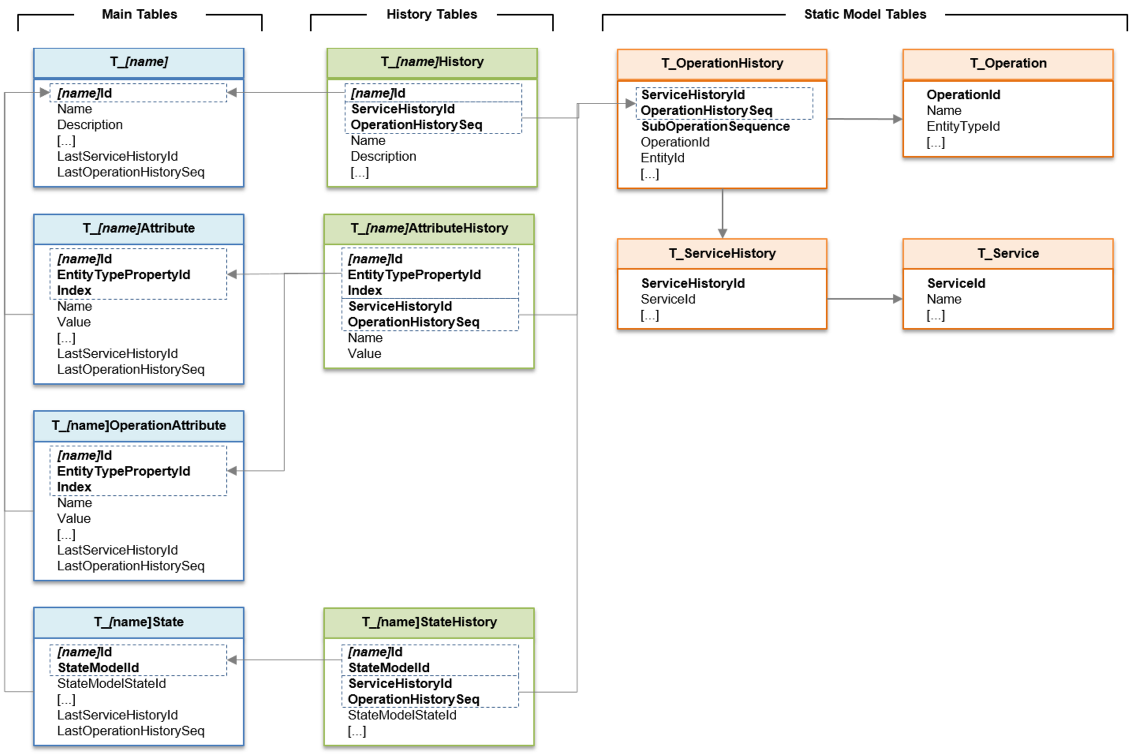
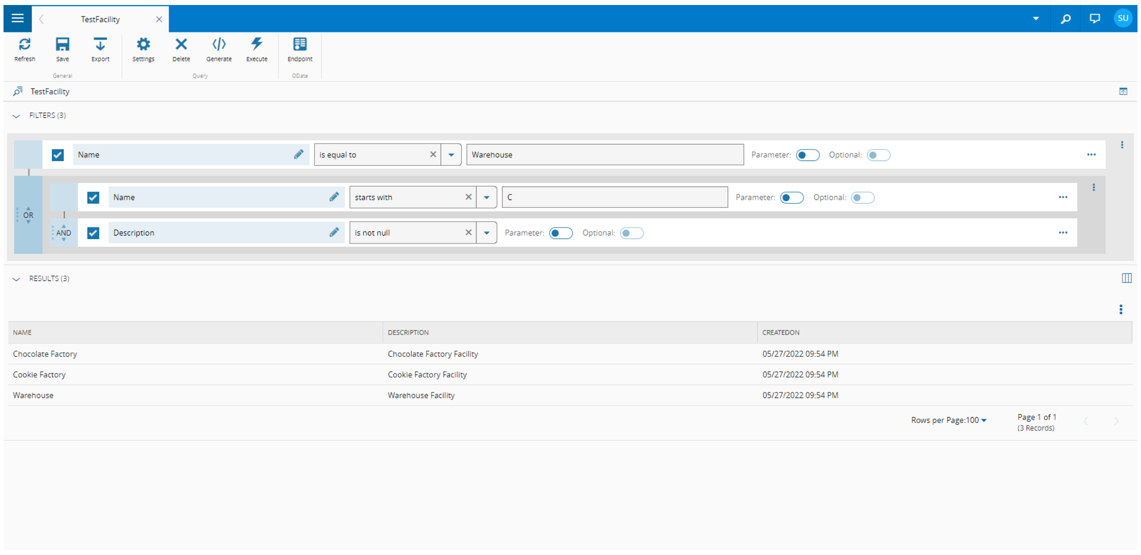


Figure 4.5: CM MES Dynamic Model *entities* sub-schema diagram



TestFacility

FILTERS (3)

✓ Name	is equal to	Warehouse	Parameter:	Optional:
✓ Name	starts with	C	Parameter:	Optional:
✓ Description	is not null		Parameter:	Optional:

RESULTS (3)

NAME	DESCRIPTION	CREATEDON
Chocolate Factory	Chocolate Factory Facility	05/27/2022 09:54 PM
Cookie Factory	Cookie Factory Facility	05/27/2022 09:54 PM
Warehouse	Warehouse Facility	05/27/2022 09:54 PM

Rows per Page:100

Page 1 of 1
(3 Records)

Figure 4.6: CM MES Query page

- **Filters**
  Contains the search properties for the Query. This filtering is extraordinarily complex and allows things like returning records whose description starts with a specific pattern or that have been created between two given dates, for example. As well as providing these handy property filters, the system also supports nested filter groups that use conditional logic, connected with AND or OR logical operators, allowing for extremely powerful filtering of an *Entity Type*.

- **Results**
  Shows the results returned by the query. It is possible to choose which columns to show and what *Entity Type Properties* are to be included in the output table, as well as things like their sorting order or how they are presented (font, color, etc.).

The way these Queries work is through a call to the CM MES REST API. This API is massive, with hundreds of different endpoints available that allow the client to talk with the backend to retrieve needed data or perform set operations. When we click on the "Execute" button, a POST request with a complex body describing the query is sent to the CM MES */api/Query/Execute-Query* endpoint. The JSON payload sent should contain a *QueryObject* property, which on top of holding a fair amount of metadata about the query to be executed, includes a *Query* field that defines the query to execute.

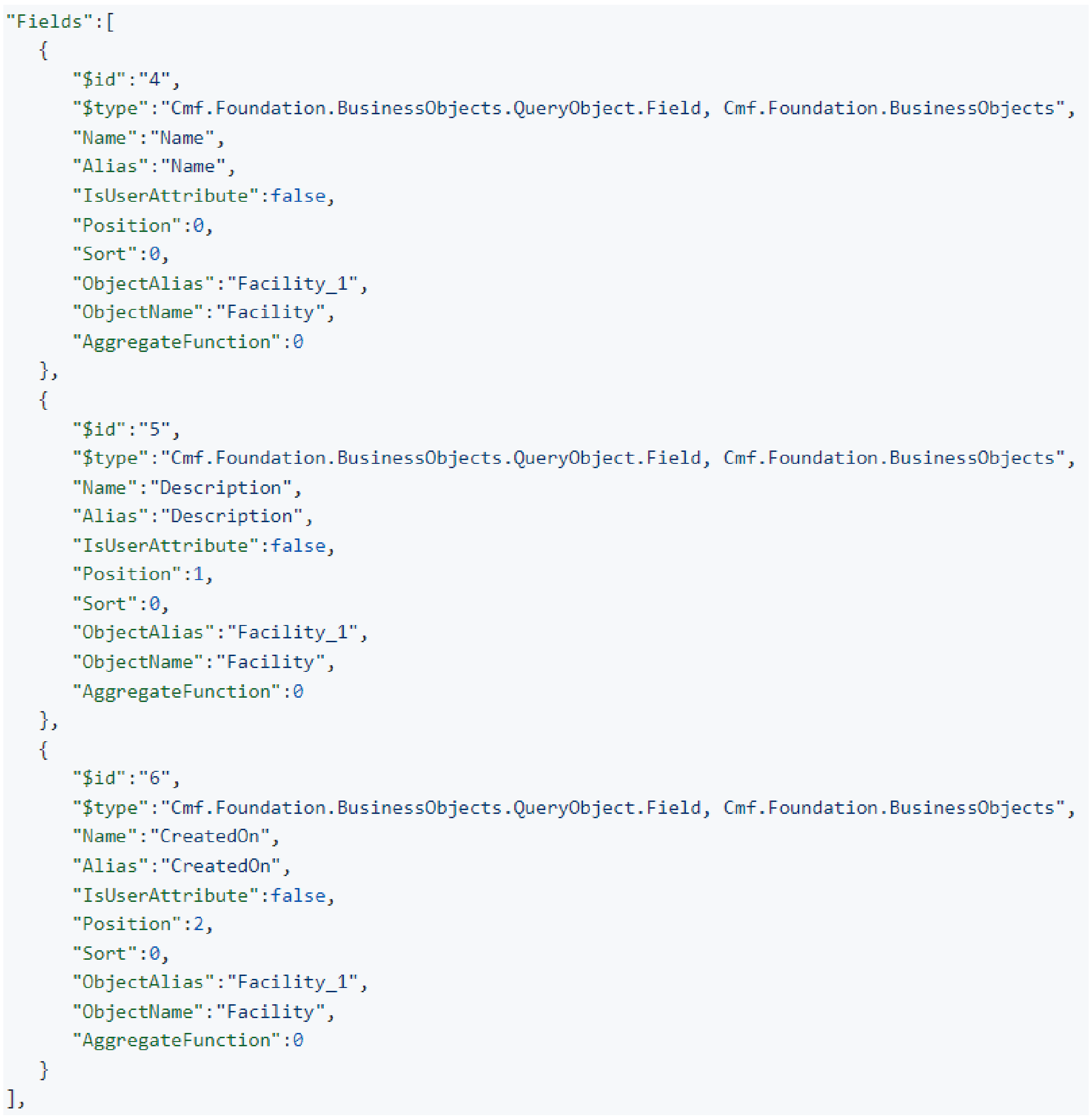
This *Query* field should be composed of a multitude of extra properties as well, with the two most important and relevant to us being *Fields* and *Filters*. *Fields* concerns the previously mentioned Results section, as it tells the API what the *Entity Type Properties* to be fetched from the database are. *Filters* on the other hand, is related to the section with the same name and encodes the filtering criteria to apply when querying for the given *Entity Type* objects. As an example, Figures 4.7 and 4.8 show, respectively, what the *Fields* and *Filters* properties of the *Query* object look like for the query displayed in Figure 4.6.

Even though this Queries feature is extremely handy and powerful, its REST API uses a complex and very verbose syntax, which offers a less than ideal developer experience, restricting the speed of development of new features by the company. Thus, exploring new API technologies such as GraphQL to improve the existing querying of the CM MES dynamic data model has been something in the company's mind.

4.2.3 Message Bus

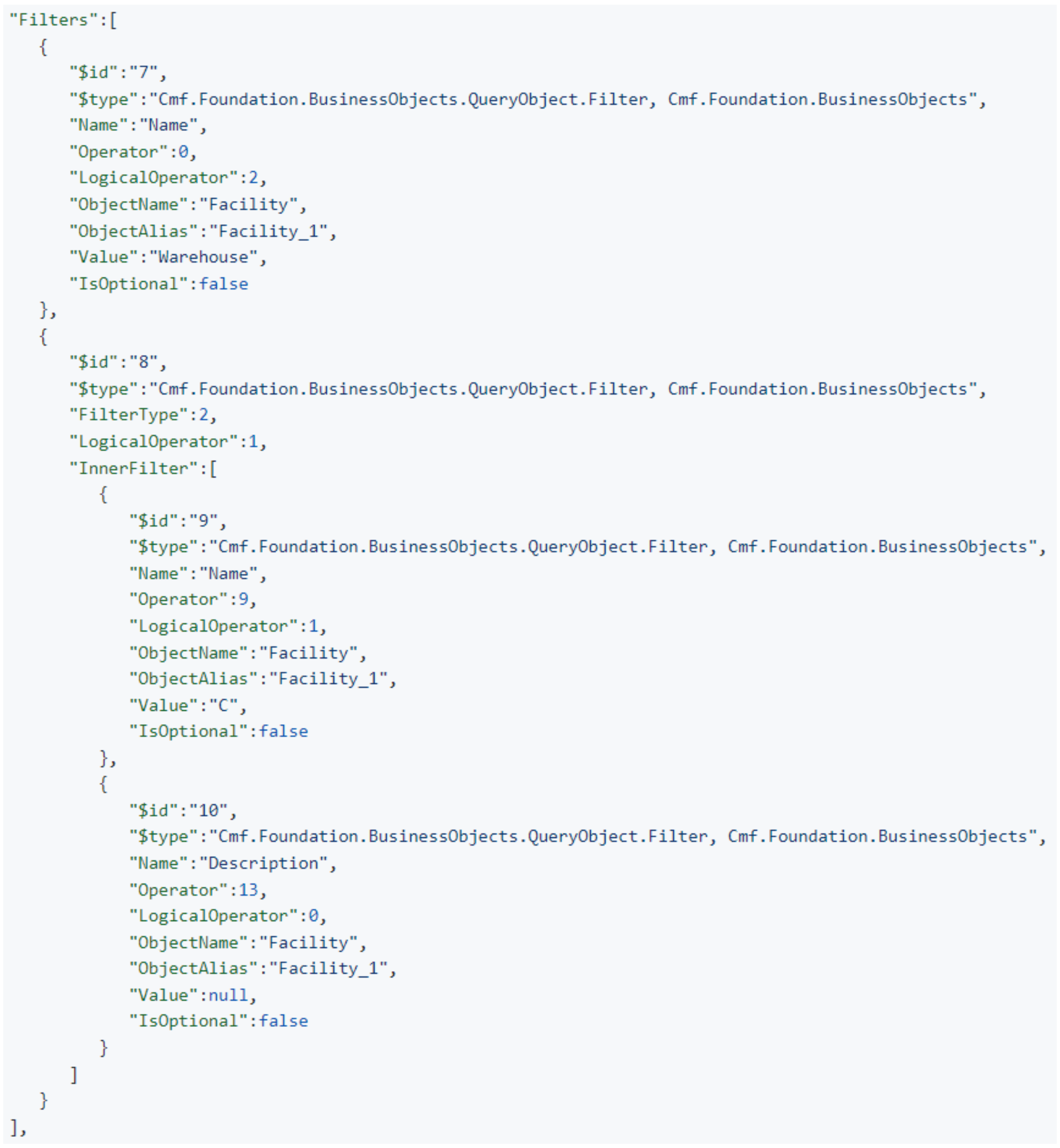
The CM MES software possesses its own message bus with a high-performance publish/subscribe environment for sending and receiving broadcast messages, built from the ground up by the company to best fit their business needs.

The message bus is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces, as represented by the diagram in Figure 4.9 [9]. There may be no guarantee of first-in-first-out ordering, and subscribers to the message bus can come and go without the knowledge of



```
"Fields":[
   {
      "$id":"4",
      "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
      "Name":"Name",
      "Alias":"Name",
      "IsUserAttribute":false,
      "Position":0,
      "Sort":0,
      "ObjectAlias":"Facility_1",
      "ObjectName":"Facility",
      "AggregateFunction":0
   },
   {
      "$id":"5",
      "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
      "Name":"Description",
      "Alias":"Description",
      "IsUserAttribute":false,
      "Position":1,
      "Sort":0,
      "ObjectAlias":"Facility_1",
      "ObjectName":"Facility",
      "AggregateFunction":0
   },
   {
      "$id":"6",
      "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
      "Name":"CreatedOn",
      "Alias":"CreatedOn",
      "IsUserAttribute":false,
      "Position":2,
      "Sort":0,
      "ObjectAlias":"Facility_1",
      "ObjectName":"Facility",
      "AggregateFunction":0
   }
],
```

Figure 4.7: CM MES Queries API *QueryObject Fields* property



```
"Filters":[
   {
      "$id":"7",
      "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
      "Name":"Name",
      "Operator":0,
      "LogicalOperator":2,
      "ObjectName":"Facility",
      "ObjectAlias":"Facility_1",
      "Value":"Warehouse",
      "IsOptional":false
   },
   {
      "$id":"8",
      "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
      "FilterType":2,
      "LogicalOperator":1,
      "InnerFilter":[
         {
            "$id":"9",
            "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
            "Name":"Name",
            "Operator":9,
            "LogicalOperator":1,
            "ObjectName":"Facility",
            "ObjectAlias":"Facility_1",
            "Value":"C",
            "IsOptional":false
         },
         {
            "$id":"10",
            "$type":"Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
            "Name":"Description",
            "Operator":13,
            "LogicalOperator":0,
            "ObjectName":"Facility",
            "ObjectAlias":"Facility_1",
            "Value":null,
            "IsOptional":false
         }
      ]
   }
],
```

Figure 4.8: CM MES Queries API *QueryObject Filters* property

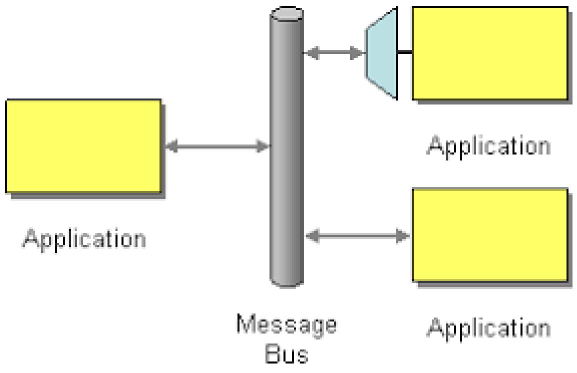


Figure 4.9: Message Bus architecture diagram

message senders. Unlike queues, where the sending application explicitly adds messages to every queue, a message bus uses a publish/subscribe model. Messages are published to the bus, and any application that has subscribed to that kind of message will receive it. This approach allows applications to follow the open/closed principle, since they become open to future changes while remaining closed to additional modification [2].

One intriguing example of the application of the CM MES message bus comes in the weigh and dispense process. Weigh and dispense is a method to provide a controlled, computer-aided process to guide the operator through the weighing and dispense process. It ensures that manufacturing processes for batches and entire lots meet strict compliance requirements for FDA, EU, or other international regulations, which require complete traceability of the batch components pertaining to the drug, coating, or active ingredients in a finished product [4]. When an operator sets some weight on a container that is connected to CM MES, specific messages are published to the message bus indicating the measurements being collected. In turn, the CM MES GUI is subscribed to this particular type of message and, as such, receives real-time updates of the electronic scale, updating the user interface accordingly.

4.3 GraphQL Middleware Architecture

The GraphQL middleware is the newly developed component that allows querying the CM MES dynamic data model through GraphQL instead of REST. It follows a microservice approach, existing independently and living outside the CM MES codebase. This middleware is a separate autonomous service that can run on its own, communicating only with the CM MES database and message bus. A diagram that helps understand the GraphQL middleware architecture can be seen in Figure 4.10.

The chosen programming language to build the middleware with was C#, on top of the .NET framework. The reason for choosing this technology to develop the project lies in the fact that the CM company's internal codebase is also almost entirely written in it. By also choosing to use .NET for the middleware, any needed integration with CM MES components becomes much simpler, like, for example, integrating communication with the CM MES message bus.

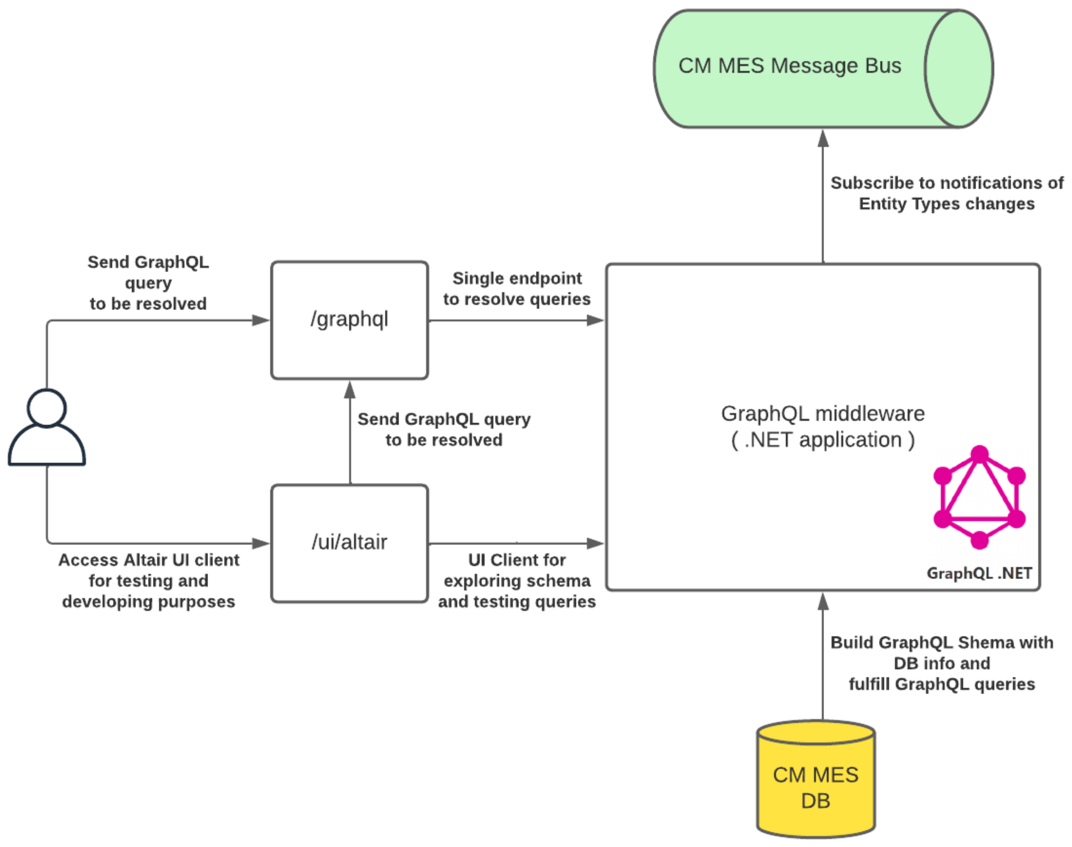

CM MES Message Bus
Subscribe to notifications of Entity Types changes
Send GraphQL query to be resolved
/graphql
Single endpoint to resolve queries
GraphQL middleware (.NET application)
Send GraphQL query to be resolved
/ui/altair
Access Altair UI client for testing and developing purposes
UI Client for exploring schema and testing queries
GraphQL .NET
Build GraphQL Shema with DB info and fulfill GraphQL queries
CM MES DB

Figure 4.10: GraphQL middleware architecure

The GraphQL part of the service was built with the GraphQL.NET library. As mentioned in 2.4.2 the reason for choosing this library is its flexibility and modularity, which allow for great abstraction while developing, not conforming to some defined methods and practices. The service exposes two endpoints, one served at the */graphql* URL and another one at */ui/altair*. The former is the single GraphQL endpoint that resolves GraphQL queries, while the latter exposes an Altair UI client, which can be accessed through a web browser.

Direct contact with the CM MES database is made to build the GraphQL schema and fulfill the GraphQL queries the server receives. Going with this approach allows limiting the number of integrations the system needs. Even though the final solution may not be production-ready for the CM company, it speeds up the development of the proof-of-concept regarding this dissertation.

The previously mentioned message bus from 4.2.3 also plays a vital part in the architecture of the application. Its integration is used so to be able to subscribe to a specific type of broadcast message that indicates invalidation of the entity type cache. This type of cache invalidation symbolizes a change in the CM MES underlying data model and is the cue needed for our service to know that it should refresh its GraphQL schema so that it is continuously updated and valid during the program's lifetime.

Chapter 5

Implementation

In this chapter, we first summarize the technologies and methodologies used to develop the work and after that, all important details in the implementation are explained,

5.1 Technologies and Methodologies Used

To produce this work, a significant number of different technologies were used, each playing an important role in the project. In addition, some development methodologies were followed to aid the development process.

5.1.1 Technologies

Most of the technologies used have already been mentioned in previous chapters. Still, if someone wants to replicate this work, there is no clear identification of what versions of them were used, for example.

As mentioned in section 4.3, the GraphQL middleware was built in the C# programming language with the .NET framework, more properly with ASP.NET Core, and with version 6 of .NET. ASP.NET is a popular web-development framework for building web apps on the .NET platform, and ASP.NET Core is the open-source version of ASP.NET that runs on Windows, Linux, macOS, and Docker [16].

To incorporate GraphQL in our application, the usage of an external library is a must, or else we would have to implement the GraphQL protocol all from the ground up. So, we rely on external packages through NuGet, which is the package manager for .NET and defines how packages are created, hosted, and consumed, also providing the tools for each of those roles [17].

A NuGet package is a single ZIP file with the *.nupkg* extension that contains compiled code, other files related to that code, and a descriptive manifest that includes information like the package's version number. Developers with code to share, create packages and publish them to a public or private host. Package consumers obtain those packages from suitable hosts, add them to their projects, and then call a package's functionality in their project code [17].

Table 5.1: NuGet packages used in the GraphQL middleware

Package Identifier	**Version Number**
GraphQL	4.7.1
GraphQL.DataLoader	4.7.1
GraphQL.MicrosoftDI	4.7.1
GraphQL.Server.Authorization.AspNetCore	5.2.0
GraphQL.Server.Transports.AspNetCore	5.2.0
GraphQL.Server.Transports.AspNetCore.SystemTextJson	5.2.0
GraphQL.Server.Ui.Altair	5.2.0
GraphQL.SystemTextJson	4.7.1
System.Data.SqlClient	4.8.3
WebSocket4Net	0.15.2
Cmf.Foundation.Common	9.0.1
Cmf.MessageBus.Client	9.0.1.237

Both mentioned chosen technologies from section 2.4.2, GraphQL.NET and Altair, are incorporated in the project through NuGet packages. Also, the CM MES message bus discussed in section 4.2.3 is integrated with the middleware in the same way, but this time through a private NuGet package developed by the CM company.

Table 5.1 lists all the NuGet packages and respective versions used to develop the project, with the last two being internally exclusive to the CM organization.

5.1.2 Methodologies

To ensure that the work developed could be easily maintained and expanded in the future, the *Clean Code* guidelines were followed, such as using meaningful variable names, writing functions that do a single thing, and writing informative comments [13].

On top of this, it was decided to implement the SOLID principles of object-oriented programming, proposed by the same author of the *Clean Code* book, *Robert C. Martin*, which are the following:

- **Single-Responsibility Principle**
  A class should have only one reason to change [12].

- **Open-Closed Principle**
  Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification [12].

- **Liskov Substitution Principle**
  Subtypes must be substitutable for their base types [12].

- **Interface Segregation Principle**
  This principle acknowledges that there are objects that require noncohesive interfaces. However, it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces [12].

- **Dependency Inversion Principle**
  High-level modules should not depend on low-level modules. Instead, both should depend on abstractions. Also, abstractions should not depend on details, but rather, details should depend on abstractions [12].

5.2 Implementation Details

We now focus on some important details of the solution developed, which help understand how the GraphQL middleware was constructed.

5.2.1 Mutable GraphQL Schema

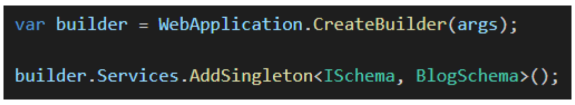
One crucial aspect of being able to have a GraphQL API built for a dynamic data model is for its GraphQL schema to be mutable, that is, have the ability to be replaced at runtime.

The GraphQL.NET library provides a helpful way to quickly build an ASP.NET Core server, which aids the development process since it removes the need for writing a ton of boilerplate code, setting up all needed endpoints, and integrations automatically. However, this approach has one minor issue, which is the fact that all necessary dependencies that it relies on must be given at compile-time, and can not be altered after the application starts running.

One of the needed dependencies that the server must be injected with is a class or instance of a class that implements the *ISchema* interface from the GraphQL.NET library. This dependency is used by the server to determine the GraphQL schema for the constructed API and whose future GraphQL requests will fall on.

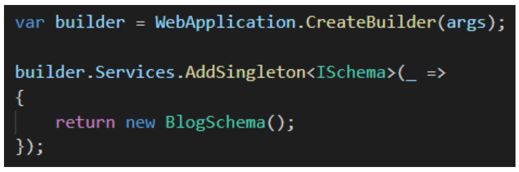
In a typical use case where updating a GraphQL schema at runtime is not necessary, developers create an auxiliary class that inherits the *Schema* class from GraphQL.NET, which in turn implements the previously mentioned *ISchema* interface. This created class represents the schema of their application, which for example, might resemble a blog as in section 2.1.2, and could be named *BlogSchema* and injected into the service as seen in Figure 5.1 or Figure 5.2. However, the problem lies in the fact that this mentioned *Schema* class is not mutable by default. That means that after initializing a *Schema* instance, that schema cannot undergo further modifications, which is crucial for our API to behave how we intend.

To accomplish this desired use case, an auxiliary class called *MutableSchema* was created (Fig. 5.3). This created class utilizes the delegation object-oriented design pattern, an implementation mechanism in which an object forwards or delegates a request to another object. In this example, the *Schema* field, denominated the delegate and an instance of the *Schema* class, carries out the requests on behalf of the original *MutableSchema* object [7].



```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<ISchema, BlogSchema>();
```

Figure 5.1: Injection of schema class into GraphQL server



```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<ISchema>(_ =>
{
    return new BlogSchema();
});
```

Figure 5.2: Injection of schema instance into GraphQL server

By injecting a *MutableSchema* instance when creating the API (Fig. 5.4), we can later modify its GraphQL schema on the fly by just replacing its *Schema* field with a completely new object representing the new schema. All subsequent GraphQL requests will then seamlessly fall on this new object through delegation.

5.2.2 Generating a GraphQL Schema Dynamically

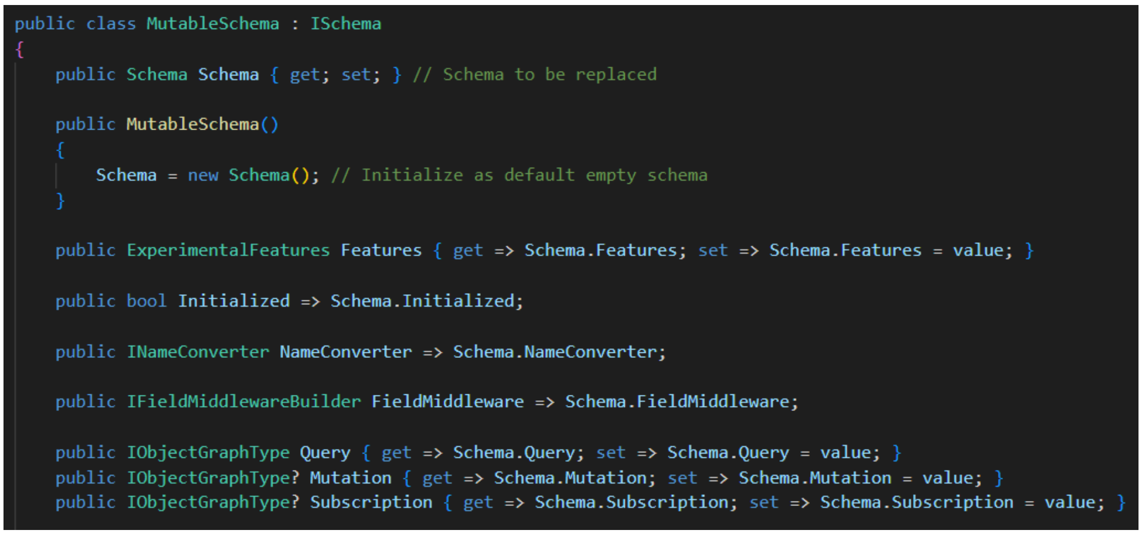
Since the underlying data model of the application can change at runtime, the GraphQL schema of the API can't just be defined in advance and must be automatically generated on the fly. To achieve this, a code-first approach for specifying the GraphQL schema is followed, which allows the creation of it in a programmatically way through code.

As cited in previous sections, the proof-of-concept developed needs only to offer a way to query CM MES *Entity Types* and corresponding *Entity Types Properties*. As such, the structure that the GraphQL schema should adopt during the execution of the service is equivalent to the one seen in Figure 5.5, which contains the following three well-defined levels:

- **Root Level**
  The first and top level of the schema is the query root operation type, which is a mandatory field in every GraphQL schema. This object type specifies all possible entry points to the GraphQL API, providing N number of queriable fields, where N is the number of existing *Entity Types* at any given time on the CM MES data model.

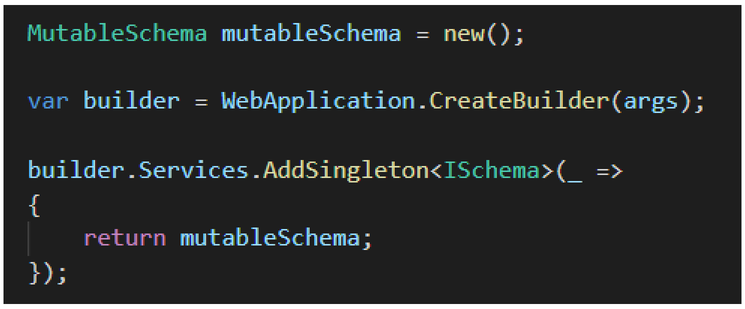
- **Entity Type Level**
  The middle level represents all of the *Entity Types* of the system, and each of the available N fields returns a list of instances of a given *Entity Type* existing in the database. These *Entity Types* are defined as a GraphQL object type, which designates a kind of object fetchable by the service with some fields of its own. For each of these N *Entity Types* fields, M additional fields are available, being M the number of its corresponding *Entity Type Properties*.

- **Entity Type Property Level**
  Finally, the last level represents the *Entity Type Properties* belonging to the *Entity Types*. These *Entity Type Properties* can be either scalar values, such as a string representing a name, or object types, symbolizing a reference to some other *Entity Type* in the system.



```
public class MutableSchema : ISchema
{
    public Schema Schema { get; set; } // Schema to be replaced

    public MutableSchema()
    {
        Schema = new Schema(); // Initialize as default empty schema
    }

    public ExperimentalFeatures Features { get => Schema.Features; set => Schema.Features = value; }

    public bool Initialized => Schema.Initialized;

    public INameConverter NameConverter => Schema.NameConverter;

    public IFieldMiddlewareBuilder FieldMiddleware => Schema.FieldMiddleware;

    public IObjectGraphType Query { get => Schema.Query; set => Schema.Query = value; }
    public IObjectGraphType? Mutation { get => Schema.Mutation; set => Schema.Mutation = value; }
    public IObjectGraphType? Subscription { get => Schema.Subscription; set => Schema.Subscription = value; }
```

Figure 5.3: Excerpt of MutableSchema class that delegates operations to its Schema field



```
MutableSchema mutableSchema = new();

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<ISchema>(_ =>
{
    return mutableSchema;
});
```

Figure 5.4: Injection of mutable GraphQL schema

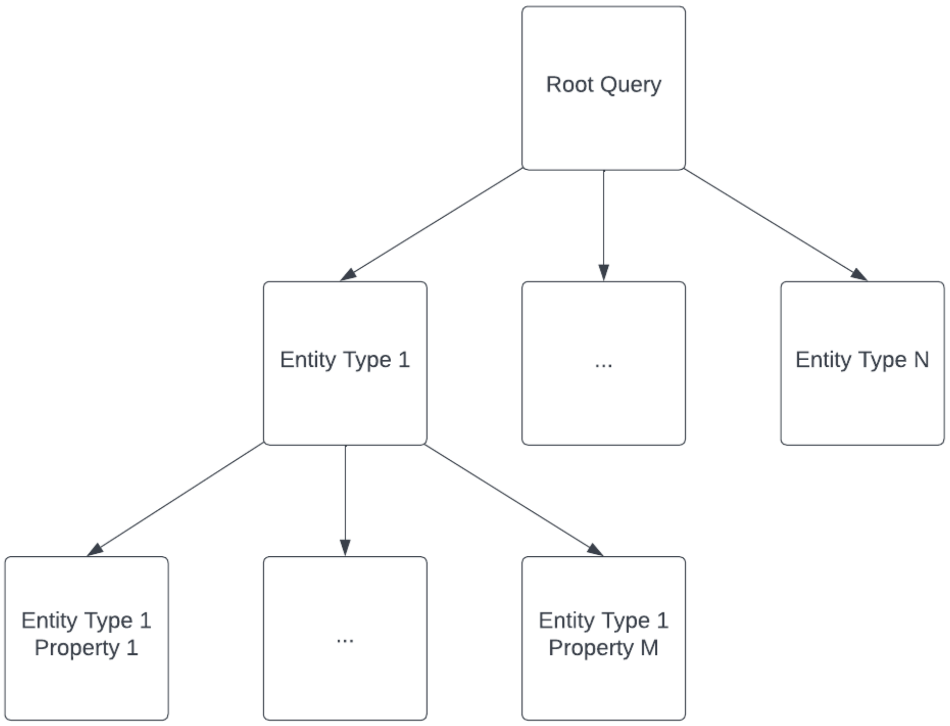
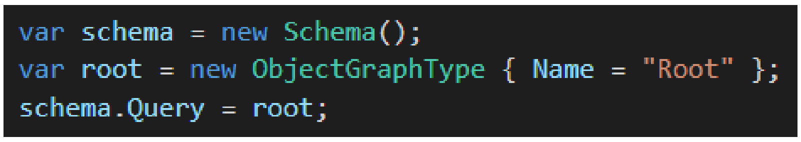


Figure 5.5: Diagram representing CM MES generated GraphQL schema

To construct the mentioned schema in a programmatic way, the first step is to initialize an empty schema and add the query root operation type, which can be done with little effort in just a few lines of code (Fig. 5.6).

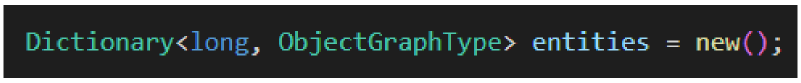
After this, it's time to add all *N* previously mentioned queriable *Entity Type* fields to the root query object type. We start by initializing a dictionary named *entities* that stores the set of object types representing the current data model *Entity Types* (Fig. 5.7). Following, we query the *T_EntityType* table to fetch the information of all existing *Entity Types* in the system and create the necessary object types that will represent them while also populating the *entities* dictionary (Fig. 5.8). Finally, to finish this GraphQL schema level, we add each *Entity Type* field one by one to the root query (Fig. 5.9), with the following four parameters:

- ***name*:** The name of the field. Equals the *Entity Type* name.
- ***type*:** The graph type of the field. Represents the type of the result to be returned by the field and is equal to a list of non null instances of the current *Entity Type* matching object type.



```
var schema = new Schema();
var root = new ObjectGraphType { Name = "Root" };
schema.Query = root;
```

Figure 5.6: Schema initialization and creation of query root operation type



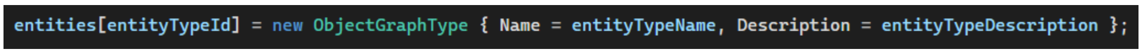
```
Dictionary<long, ObjectGraphType> entities = new();
```

Figure 5.7: Dictionary that stores a set of object types that represent *Entity Types*

- ***arguments***: A list of arguments for the field. Includes two possible types, *pagination*, and *filtering*, explained in sections 5.2.4 and 5.2.5, respectively
- ***resolve***: A field resolver delegate. It is the resolver function that tells GraphQL how to get the data needed to fulfill the query. In a typical scenario where an application's data model is static, we would utilize classes to represent each possible queriable entity of the schema, returning a list of objects which are instances of the given class. However, since we are working in a dynamic environment, there is no way to represent an *Entity Type* with a class since this type of programming template must be written at compile-time. As such, an out-of-the-box alternative approach is needed, and each *Entity Type* is instead coded as a dictionary. The keys of this dictionary are strings that map to the given *Entity Type* properties names, and their matching values are objects that store the given *Entity Type Property* value, if existent. The dynamic tables of the sort *T_[name]* previously mentioned are used in the *GetEntitiesInstancesOfType* function, which queries instances of a given *Entity Type* based on the given arguments and then constructs a list of the dictionaries that represent each instance. Finally, this list is returned by the resolver function.

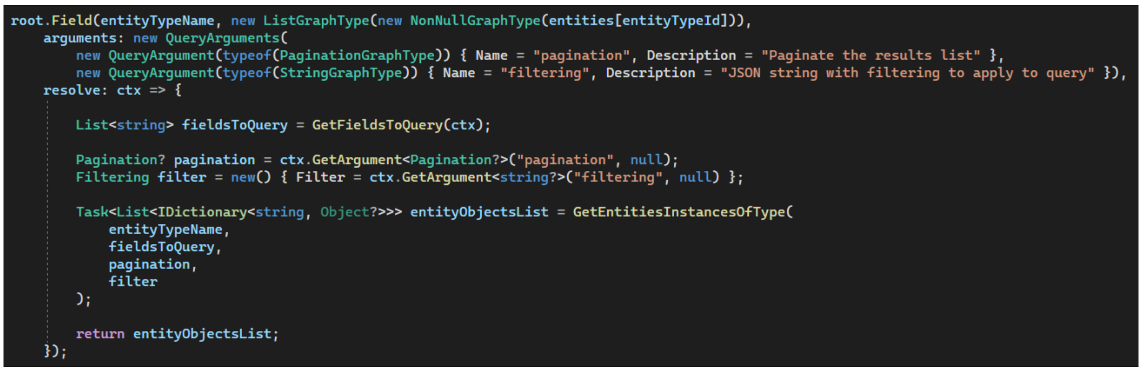
The last level remaining to complete the GraphQL schema is the *Entity Type Property* one. To construct this part of the schema, we start by querying all existing *Entity Type Properties* in the system from the *T_EntityTypeProperty* table. Then, we iterate through each and add them as a field to the upper level's corresponding *Entity Type* object type, resolving either to a scalar value (Fig. 5.10) or to some *Entity Type* (Fig. 5.11), with the following four parameters:

- ***name***: The name of the field. Equals the *Entity Type Property* name.
- ***type***: The graph type of the field. Represents the type of the result to be returned by the field and can be either a straight scalar or an object type corresponding to some *Entity Type* from the *entities* dictionary.
- ***description***: The description of the field. It is a small string that describes the *Entity Type Property*.
- ***resolve***: A field resolver delegate. It is the resolver function that tells GraphQL how to get the data needed to fulfill the query. If the field type equals a scalar, we can directly return its value by accessing the parent object in the graph, which should equal a dictionary



```
entities[entityTypeId] = new ObjectGraphType { Name = entityTypeName, Description = entityTypeDescription };
```

Figure 5.8: Creation of *Entity Types* object types and population of *entities* dictionary



```
root.Field(entityTypeName, new ListGraphType(new NonNullGraphType(entities[entityTypeId])),
    arguments: new QueryArguments(
        new QueryArgument(typeof(PaginationGraphType)) { Name = "pagination", Description = "Paginate the results list" },
        new QueryArgument(typeof(StringGraphType)) { Name = "filtering", Description = "JSON string with filtering to apply to query" }),
    resolve: ctx => {

        List<string> fieldsToQuery = GetFieldsToQuery(ctx);

        Pagination? pagination = ctx.GetArgument<Pagination?>("pagination", null);
        Filtering filter = new() { Filter = ctx.GetArgument<string?>("filtering", null) };

        Task<List<IDictionary<string, Object?>>> entityObjectsList = GetEntitiesInstancesOfType(
            entityTypeName,
            fieldsToQuery,
            pagination,
            filter
        );

        return entityObjectsList;
    });
```

Figure 5.9: Creation of *Entity Type* field

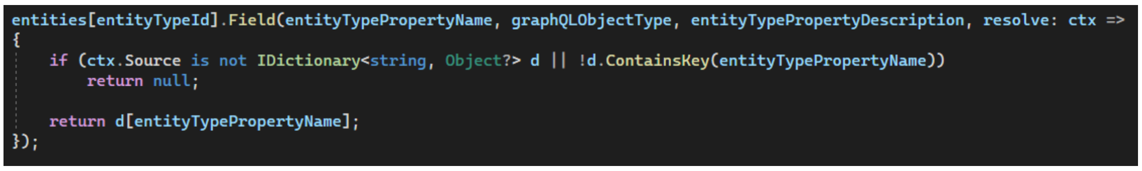
representing the *Entity Type* to which the current *Entity Type Property* belongs. The value to be returned by the field equals the value in the mentioned dictionary associated with the key that matches the current *Entity Type Property* name. If, on the other hand, the field type should resolve to some *Entity Type*, an inquiry to the database is made to fetch the corresponding instance by its id, and the value returned by the field is once again a dictionary with the same structure as the one used in the *Entity Type* level. In this case, an auxiliary tool called DataLoader is also utilized, which is explained further in section 5.2.6.

Two examples of how the GraphQL requests to the API and corresponding responses look can be seen in Figures 5.12 and 5.13. The first one represents a query to the CM MES data model of instances of the *Folder Entity Type*, returning its *Name* and *Description Entity Type Properties*. The second one is similar, but also returns additional information of its *ParentFolder Entity Type Property*, which is not a scalar value but rather a reference to some *Entity Type* instance.

5.2.3 Updating the GraphQL Schema at Runtime

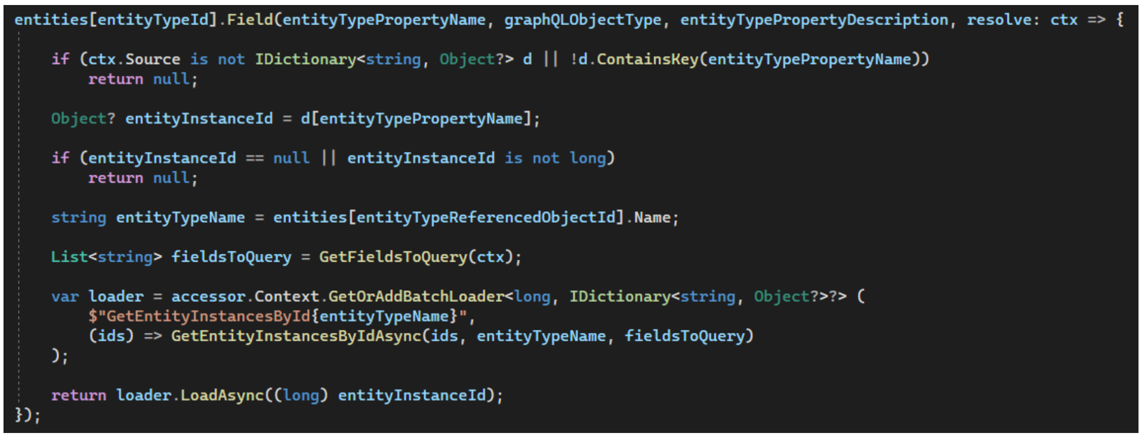
The last needed piece to complete the proof-of-concept of having a GraphQL service for an application with a dynamic data model is for it to adapt to the model changes in real-time. To accomplish this, we rely on the help of the CM MES messages bus mentioned in section 4.2.3.

CM MES uses caching for a lot of its data storage needs, and whenever some element of this cache is invalidated, a particular type of broadcast message is published in the message bus, whose subject is *CMF.SYSTEM.ADMINISTRATION.INVALIDATECACHE*. In Figure 5.14 and Figure 5.15, we can see two examples of what this broadcast message might look like, being that we are interested in reacting to the *EntityTypeCache* invalidation seen in the latter. This type



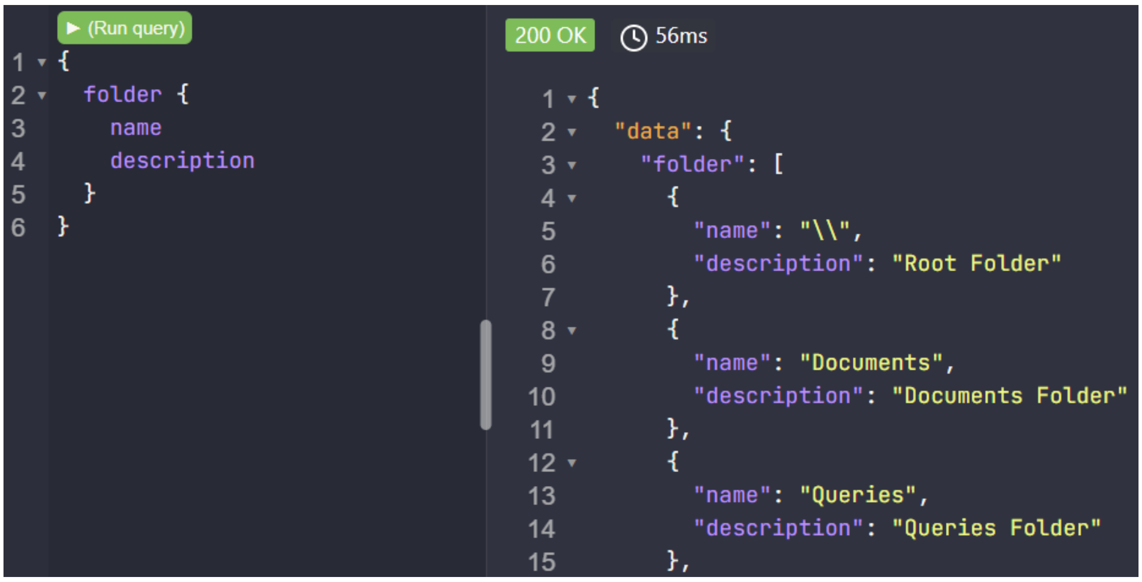
```
entities[entityTypeId].Field(entityTypePropertyName, graphQLObjectType, entityTypePropertyDescription, resolve: ctx =>
{
    if (ctx.Source is not IDictionary<string, Object?> d || !d.ContainsKey(entityTypePropertyName))
        return null;

    return d[entityTypePropertyName];
});
```

Figure 5.10: Creation of *Entity Type Property* field which resolves to scalar value



```
entities[entityTypeId].Field(entityTypePropertyName, graphQLObjectType, entityTypePropertyDescription, resolve: ctx => {

    if (ctx.Source is not IDictionary<string, Object?> d || !d.ContainsKey(entityTypePropertyName))
        return null;

    Object? entityInstanceId = d[entityTypePropertyName];

    if (entityInstanceId == null || entityInstanceId is not long)
        return null;

    string entityTypeName = entities[entityTypeReferencedObjectId].Name;

    List<string> fieldsToQuery = GetFieldsToQuery(ctx);

    var loader = accessor.Context.GetOrAddBatchLoader<long, IDictionary<string, Object?>?> (
        $"GetEntityInstancesById{entityTypeName}",
        (ids) => GetEntityInstancesByIdAsync(ids, entityTypeName, fieldsToQuery)
    );

    return loader.LoadAsync((long) entityInstanceId);
});
```

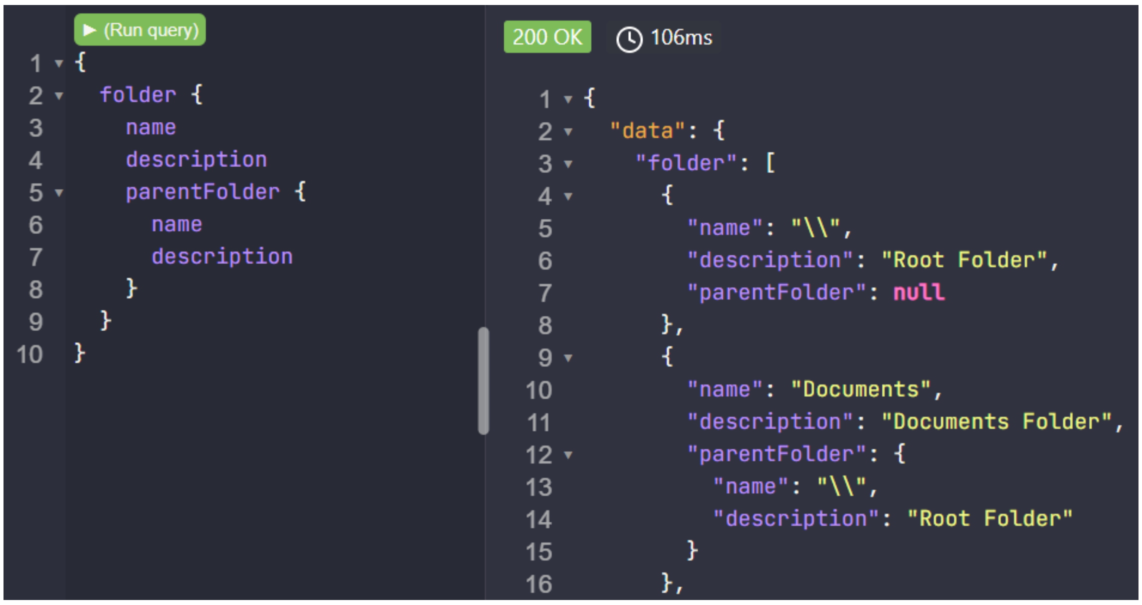
Figure 5.11: Creation of *Entity Type Property* field which resolves to some *Entity Type* instance



```
{
  folder {
    name
    description
  }
}
```

```
{
  "data": {
    "folder": [
      {
        "name": "\\",
        "description": "Root Folder"
      },
      {
        "name": "Documents",
        "description": "Documents Folder"
      },
      {
        "name": "Queries",
        "description": "Queries Folder"
      },
```

Figure 5.12: Querying of *Name* and *Description Entity Type Properties* of *Folder Entity Types*



▶ (Run query)

```
{
  folder {
    name
    description
    parentFolder {
      name
      description
    }
  }
}
```

200 OK 106ms

```
{
  "data": {
    "folder": [
      {
        "name": "\\",
        "description": "Root Folder",
        "parentFolder": null
      },
      {
        "name": "Documents",
        "description": "Documents Folder",
        "parentFolder": {
          "name": "\\",
          "description": "Root Folder"
        }
      },
```

Figure 5.13: Querying of *Name*, *Description*, *ParentFolder Name* and *ParentFolder Description Entity Type Properties* of *Folder Entity Types*

of cache invalidation conveys that some change to the underlying data model has occurred, be it because some new *Entity Type* has been created or because some *Entity Type Property* has been modified, for example. So, whenever a message of this kind is published to the message bus, it is the cue that our service needs to know it should update the GraphQL schema.

As such, our GraphQL service subscribes to messages published to the CM MES message bus whose subject is *CMF.SYSTEM.ADMINISTRATION.INVALIDATECACHE*, and whenever the *CacheManager* field is present and equal to *EntityTypeCache*, it generates a new schema at runtime and, if no errors occur, substitutes the GraphQL schema of the API. If the *CacheManager* field is not present or does not equal *EntityTypeCache*, the message can just be ignored.

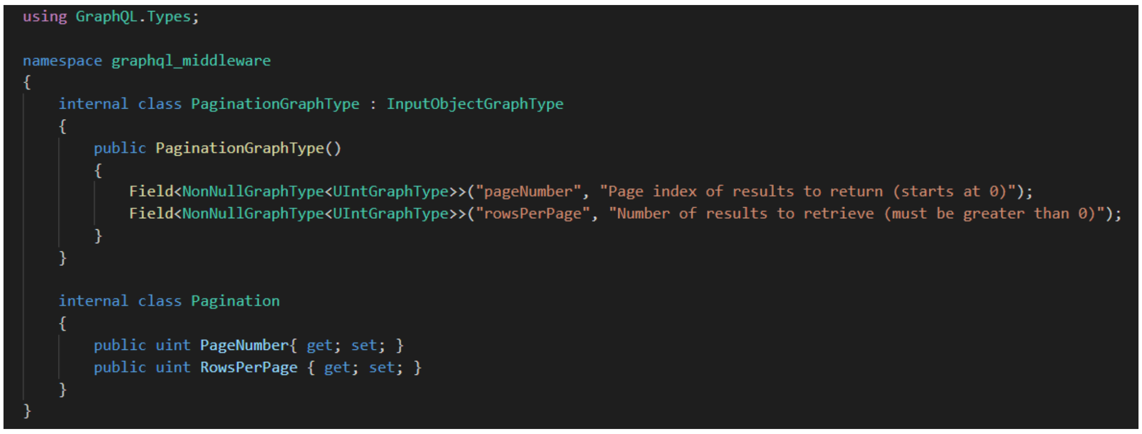
This newly generated schema is constructed from scratch, causing the GraphQL schema of the API to not be entirely valid for the short period of time that it takes to build it. Some kind of patching could possibly be made to the outdated schema to avoid this issue instead of replacing it altogether, however, the generation of a new schema approach was taken due to time constraints

```
{
    "CacheManager": "SecurityCache"
}
```

Figure 5.14: CM MES Security cache invalidation message

```
{
    "CacheManager": "EntityTypeCache",
    "CacheEntryNames": [
        "Site"
    ],
    "CacheEntryIds": [
        "1805111612560000011"
    ]
}
```

Figure 5.15: CM MES Entity Type cache invalidation message



```
using GraphQL.Types;

namespace graphql_middleware
{
    internal class PaginationGraphType : InputObjectGraphType
    {
        public PaginationGraphType()
        {
            Field<NonNullGraphType<UIntGraphType>>("pageNumber", "Page index of results to return (starts at 0)");
            Field<NonNullGraphType<UIntGraphType>>("rowsPerPage", "Number of results to retrieve (must be greater than 0)");
        }
    }

    internal class Pagination
    {
        public uint PageNumber{ get; set; }
        public uint RowsPerPage { get; set; }
    }
}
```

Figure 5.16: Pagination GraphQL input type

and since it offers more security against potential unexpected bugs stemming from incorrect usage of the GraphQL library.

5.2.4 Pagination

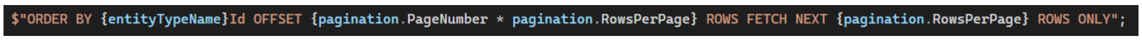
GraphQL does not implement a way to paginate the results of a query by default. Nonetheless, CM MES current REST API offers this ability, returning either 10, 25, 50, 100, or 200 rows of results at each time, and as such, the developed GraphQL API should also provide a similar feature.

To incorporate pagination into the GraphQL server, a *Pagination* input type was created (Fig. 5.16), which can be included in any request as a paramater. This *Pagination* type is composed of two fields, represented by unsigned integers and that are mandatory to be present: *pageNumber* and *rowsPerPage*. The first one represents the page index of results to return, and starts at 0. The second one represents the number of results to retrieve, which should be greater than 0.

As an example of how this pagination solution works, let's say we have the following array of data [A,B,C,D,E,F,G,H,I,J]. If we want to fetch only the first five results from this array, we would have a *Pagination* object with its *pageNumber* field equal to 0, and its *rowsPerPage* field equal to 5. If we instead want to get the last five results of the array, we would have *pageNumber* equal to 1, and *rowsPerPage* would remain 5. If we, for example, chose a *Pagination* object with *pageNumber* equal to 3 and *rowsPerPage* equal to 3, the result would only be the array [J] since there is not enough data to fill the three rows per page we seek. By the same logic, if *pageNumber* happened to equal 4, we would obtain the empty array [] as a result.

This solution developed allows for any value of *rowsPerPage*, compared to the previously mentioned five pre-defined ones of the CM MES REST API. On top of this, it can offer increased performance on querying the system since the service does not need to return all the records matching the query, which sometimes can be in the range of thousands, and can provide them at given intervals.

This pagination of results is implemented through the injection of a pagination string (Fig. 5.17) at the end of the SQL Server query to the database. In this query, *OFFSET* specifies the



```
$"ORDER BY {entityTypeName}Id OFFSET {pagination.PageNumber * pagination.RowsPerPage} ROWS FETCH NEXT {pagination.RowsPerPage} ROWS ONLY";
```

Figure 5.17: Pagination string to query SQL database

number of rows to skip before it starts to return rows from the query expression, and *FETCH* specifies the number of rows to return after the *OFFSET* clause has been processed [15]. Note that when *rowsPerPage* ends up being equal to 0, it is assumed that the pagination is invalid, and as such, the empty string is injected instead, and no pagination occurs.

Figures 5.18 and 5.19 display how this Pagination argument is used in a GraphQL query, showing two consecutive pages of results for the *Facility Entity Type*. In the first query, we return the first two instances of this object in the system, while in the second query, we retrieve the third and fourth.

5.2.5 Filtering

Like with pagination, GraphQL also does not implement an out-of-the-box solution to filter the results of a query. Still, the CM MES REST API offers a handy way to search for *Entity Types* that match some given criteria since this is a pretty recurrent use case for users of the software. As such, the developed GraphQL API should also incorporate some kind of filtering of results to cover this functionality.

Filtering GraphQL results, however, is not straightforward and can become rather complex. One naive approach for a GraphQL API to deliver filtered results could be to execute a GraphQL query as it is and then do subsequently filtering to the response obtained. This approach is a possible solution to the problem but presents dire performance since there is an over-fetching of data from the database, and a lot of the records being collected might end up being discarded from the final list. Therefore, the better solution is to query directly the necessary items from the database, with GraphQL itself being aware of that and implementing the required filtering.

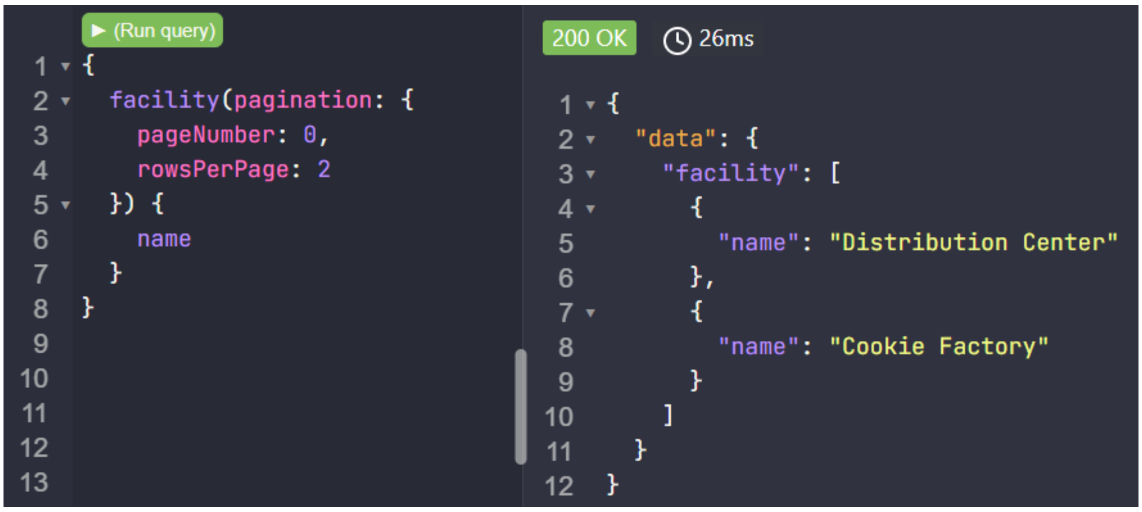
To incorporate filtering results with GraphQL, we once again utilize query arguments. This time we utilize a single string named *filtering*, which is quite intriguing since it should abide by some directives that define conditional logic in a JSON format. The reason for choosing this approach is due to the fact that GraphQL input types have a restricted set of rules that need to be followed, which offer limited abstraction. By using a single string as an argument, we can pass any type of information in the way we want to the backend, as long as it is coded in an expected and predictable way.

The JSON string containing the filtering instructions is defined as a JSON filter and should follow the following set of rules:

1. A JSON filter could be either a condition object or a condition array.

2. A condition object must contain only the following three properties:

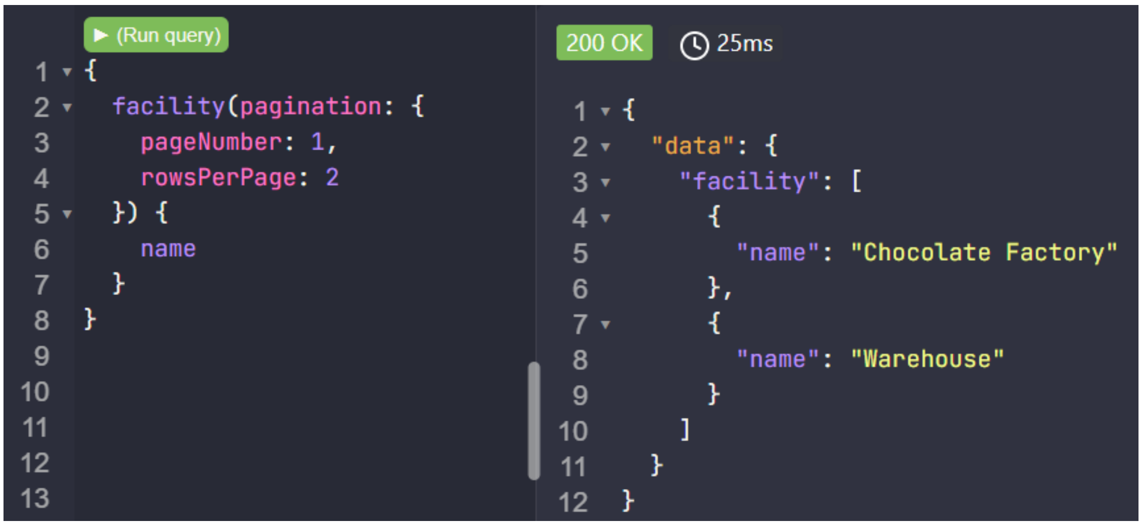
   ***field*:** Indicates to which of the *Entity Type Properties* the condition should apply.



```
{
  facility(pagination: {
    pageNumber: 0,
    rowsPerPage: 2
  }) {
    name
  }
}
```

```
{
  "data": {
    "facility": [
      {
        "name": "Distribution Center"
      },
      {
        "name": "Cookie Factory"
      }
    ]
  }
}
```

Figure 5.18: GraphQL query to retrieve first and second instances of Facility entity type



```
{
  facility(pagination: {
    pageNumber: 1,
    rowsPerPage: 2
  }) {
    name
  }
}
```

```
{
  "data": {
    "facility": [
      {
        "name": "Chocolate Factory"
      },
      {
        "name": "Warehouse"
      }
    ]
  }
}
```

Figure 5.19: GraphQL query to retrieve third and fourth instances of Facility entity type

operation: Indicates the conditional operation to apply to the *field*. It should be equal to one of the following pre-defined values, whose CM MES REST API also uses:

CONTAINS: *field* value should contain some specified string.

STARTS_WITH: *field* value should start with specified string.

LIKE: *field* value should match a specified pattern.

NOT_LIKE: *field* value should not match a specified pattern.

IS_GREATER_THAN: *field* value should be greater than some specified number.

IS_GREATER_THAN_OR_EQUAL_TO: *field* value should be greater than or equal to some specified number.

IS_LESSER_THAN: *field* value should be lesser than some specified number.

IS_LESSER_THAN_OR_EQUAL_TO: *field* value should be lesser than or equal to some specified number.

IS_EQUAL_TO: *field* value should equal a specified value.

IS_DIFFERENT_THAN: *field* value should be different than a specified value.

IN: *field* value should equal one of the multiple specified values.

NOT_IN: *field* value should not equal one of the multiple specified values.

NULL: *field* value should be nonexistent. *value* property is ignored.

IS_NOT_NULL: *field* value should exist. *value* property is ignored.

TRUE: *field* value should equal to true. *value* property is ignored.

FALSE: *field* value should equal to false. *value* property is ignored.

value: Indicates the specified value that the *field* should confine to based on the given *operation*.

3. A condition array must have at least three elements.

4. A condition array's first entry must be either the string *OR* or the string *AND* and is the logical operation that joins the remaining elements.

5. A condition array's second and further entries can be either conditional objects or conditional arrays themselves.

To better understand what this JSON filter can look like, Figures 5.20 and 5.21 show two examples of possible instances of it. In the first one, we specify that we want to query all *Entity Types* whose *Name Entity Type Property* starts with the letter "C". In the latter, the filtering is a bit more complex. We state that not only all *Entity Types* named "Cookie Factory" should be fetched, but also all of them whose *Id Entity Type Property* is greater than 2205280302110000018 and lesser than or equal to 2205280302110000021.

Nonetheless, this filtering solution is not perfect and possesses a significant shortcoming, which is the inability to apply nested filters. This means that filtering can only be done at the *Entity Type* level, based on its corresponding *Entity Type Properties*. However, suppose one of

```json
{
  "field": "name",
  "operation": "STARTS_WITH",
  "value": "C"
}
```

Figure 5.20: Simple JSON filter example

```json
[
  "OR",
  {
    "field": "name",
    "operation": "IS_EQUAL_TO",
    "value": "Cookie Factory"
  },
  [
    "AND",
    {
      "field": "id",
      "operation": "IS_GREATER_THAN",
      "value": "2205280302110000018"
    },
    {
      "field": "id",
      "operation": "IS_LESSER_THAN_OR_EQUAL_TO",
      "value": "2205280302110000021"
    }
  ]
]
```

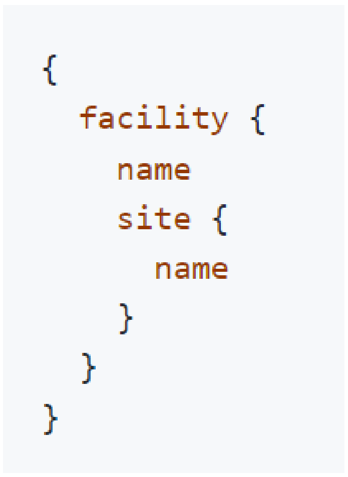
Figure 5.21: Complex JSON filter example

these *Entity Type Properties* is a reference to some other *Entity Type*. In that case, there is no way to filter based on the new *Entity Type Properties*. Since GraphQL resolves fields based on levels, coming up with a solution to this problem is not straightforward. For example, the *Folder Entity Type* possesses an *Entity Type Property* named *ParentFolder*, that references some other *Folder* in the system. If we want to query all *Folder* objects whose *ParentFolder* name equals a certain value, since GraphQL will first resolve the *Entity Type* level mentioned in 5.2.2 and there is no filter to apply at this stage, all *Folder* objects will be loaded from the database, even though some of them will end up not matching the given criteria. Still, the filtering implemented can cover CM MES users' most realistic use cases and makes the API more beneficial than not.

5.2.6 DataLoader

The DataLoader is a compelling tool that commonly appears in GraphQL APIs. It was originally developed at Facebook by engineer Nicholas Schrock and is described as a generic utility to be used as part of an application's data fetching layer to provide a simplified and consistent API over various remote data sources such as databases or web services via batching and caching [21].

A frequent issue that appears when building GraphQL APIs and that ends up negatively impacting their performance is the *N + 1* problem. Let us say we have the following GraphQL query seen in Figure 5.22, where we want to fetch the names of some *Facility Entity Types* and their corresponding *Site* names. When the GraphQL query is executed, first, a list of all *Facilities* is



```
{
  facility {
    name
    site {
      name
    }
  }
}
```

Figure 5.22: GraphQL query which benefits from DataLoader batching

fetched with a single query to the database. Then, for each *Facility*, the associated *Site* must also be fetched. If each *Site* is fetched one-by-one, this will get more inefficient as the number of *Facilities* (*N*) grows. This is known as the previously mentioned *N* + *1* problem. If there are 50 *Facilities* (*N* = 50), 51 separate requests would be made to load this data, 1 for getting all *Facilities* instances, and then 50 extra requests for their corresponding *Site*. Using a DataLoader allows us to batch together all requests for the *Sites*. As such, 1 request to retrieve the list of *Facilities* is made, plus 1 request to load all *Sites* associated with those *Facilities*. In total, we will always have only two requests to the database, instead of the unsatisfactory *N* + *1* [8].

A DataLoader helps a GraphQL service by providing both batching of results, so that fewer roundtrips to a database are made, and caching, so that GraphQL requests do not have to be executed yet again if a similar request has been previously made and their value is already known. Due to time constraints, only batching was incorporated into the project since it is the one feature that brings the most value, decreasing the response time of requests with nested fields substantially, deemed essential for it to be able to compete with the current REST API.

Chapter 6

Results Analysis

In this section, we highlight the results obtained, answering the initial dissertation questions presented in section 3.4. Finally, a brief discussion of the findings is made.

6.1 Applicability of GraphQL for Dynamic Data Models

One of the central questions to be answered with this dissertation is the possibility of using GraphQL technology in an application whose underlying data model is dynamic.

With the work produced, we can conclude that this use case for GraphQL is indeed doable. Even though not all aspects of GraphQL were explored, like mutations or subscriptions, and this still being only a mere proof-of-concept, we can confidently say that GraphQL is a solution for building APIs on top of dynamic data models.

The development of this project on top of a real-world application like CM MES also helps validate this use case since the developed GraphQL API can replace, in some situations, the existing REST API that the company constructed, as seen more in detail in section 6.2.3.

6.2 Comparative Analysis of REST and GraphQL APIs

To analyze how the CM MES current REST API and the newly developed GraphQL API compare with each other, three different aspects were accessed: performance, developer experience and usability.

6.2.1 Performance

One of the main aspects that the CM company cares about in its MES product is the performance of its API, with fast response times being essential for the software to provide the best possible experience to its users.

To evaluate how the newly developed GraphQL middleware compares with the current REST API, we should test similar queries for the same information and analyze the average response time each presents. As such, the following five different scenarios were thought of, which assess different kinds of aspects:

- **Scenario 1:** Querying the *Id* and *Name Entity Type Properties* of the first 100 instances of the *Folder Entity Type*. It evaluates the default query of an *Entity Type*.
- **Scenario 2:** Querying all scalar type *Entity Type Properties* of the first 100 instances of the *Folder Entity Type*. It evaluates a query of many *Entity Type Properties*.
- **Scenario 3:** Querying the *Id*, *Name*, *ParentFolder Id* and *ParentFolder Name Entity Type Properties* of the first 100 instances of the *Folder Entity Type*. It evaluates a query of nested fields by fetching *Entity Type Properties* that refer to another *Entity Type*.
- **Scenario 4:** Querying the *Id* and *Name Entity Type Properties* of the first 100 instances of the *Folder Entity Type* whose *Name* starts with the letter "D". It evaluates a query of an *Entity Type* based on a simple filter.
- **Scenario 5:** Querying the *Id* and *Name Entity Type Properties* of the first 100 instances of the *Folder Entity Type* whose *Name* starts with the letter "D" or whose *Id* is greater than 1805111613350000005 and lesser than 1805111613350000008. It evaluates a query of an *Entity Type* based on a more complex filter.

For each scenario, a total number of 100 test requests were made to each API, separated into 10 groups of alternating trial tests composed of 10 requests each. Similar queries were elaborated and tested with Postman[1] to measure the response times, whose results can be seen in tables 6.1 through 6.5.

Table 6.1: Response time average per ten requests in 10 groups of trial tests - Scenario 1

Trial Test	**REST (ms)**	**GraphQL (ms)**
1	73.4	53.8
2	94.1	69.1
3	79.1	72.2
4	79.8	62.2
5	79.6	55.3
6	89.4	64.2
7	95.5	50.9
8	77.7	65.8
9	78.4	61.8
10	91.1	69.7
Total Average	83.8	62.5

[1]https://www.postman.com/

Table 6.2: Response time average per ten requests in 10 groups of trial tests - Scenario 2

Trial Test	**REST (ms)**	**GraphQL (ms)**
1	143.9	115.1
2	157.8	118.2
3	148.4	137.6
4	177.4	90.9
5	111.7	111.8
6	146.9	126.6
7	143.9	127.8
8	128.0	125.8
9	160.5	126.3
10	146.0	104.2
Total Average	146.5	118.4

Table 6.3: Response time average per ten requests in 10 groups of trial tests - Scenario 3

Trial Test	**REST (ms)**	**GraphQL (ms)**
1	92.3	105.3
2	84.7	119.3
3	97.5	101.5
4	77.2	99.7
5	90.0	97.4
6	97.1	97.6
7	85.4	97.8
8	88.1	110.3
9	76.7	110.1
10	92.8	112.8
Total Average	88.2	105.2

Table 6.4: Response time average per ten requests in 10 groups of trial tests - Scenario 4

Trial Test	**REST (ms)**	**GraphQL (ms)**
1	75.3	42.5
2	70.1	49.8
3	59.2	51.4
4	63.8	48.6
5	81.4	48.9
6	60.2	47.7
7	81.1	45.8
8	82.1	43.7
9	58.2	44.0
10	79.2	45.1
Total Average	71.1	46.8

Table 6.5: Response time average per ten requests in 10 groups of trial tests - Scenario 5

Trial Test	**REST (ms)**	**GraphQL (ms)**
1	65.6	45.2
2	64.1	42.8
3	66.4	48.9
4	64.0	45.5
5	56.6	47.3
6	68.1	54.8
7	72.9	47.4
8	69.3	48.3
9	61.6	42.0
10	62.0	48.8
Total Average	65.1	47.1

From the gathered results, Scenario 3 stands out from the others since it is the only one where the REST API outperforms the developed middleware. The goal of this scenario is to evaluate the querying of nested fields, which is interesting since it brings out the familiar *N + 1* problem mentioned in section 5.2.6. Even though the developed solution utilizes the DataLoader tool, which improves a GraphQL API performance for the said issue, the constructed service still falls behind the existing REST API by some margin, being about 19% slower on average. We can attribute this to the fact that the CM MES REST API is incredibly optimized, fetching all the data needed to fulfill the query for the referred scenario in just a single inquiry to the database. On the other hand, as explained before, the GraphQL API requires two roundtrips to the CM MES database, increasing the query resolution total duration and making this API more inefficient for this use case.

The new service performs better in the remaining four scenarios being faster on average by about 25% in Scenario 1, 19% in Scenario 2, 34% in Scenario 4, and 28% in Scenario 5. The adopted filtering solution mentioned in section 5.2.5 also seems to be a great choice, enabling the GraphQL API to sustain its good performance against the REST alternative. Still, we should remind ourselves that the developed middleware is just a proof-of-concept, and some extra operations and checks are missing that would add a bit more time to a query's execution duration. As such, we can not claim that GraphQL will perform better than REST in querying a dynamic data model for these scenarios, but we can reliably state that it is a valid and competitive alternative.

An additional sixth scenario was also considered to evaluate one of the main advantages of using a GraphQL API, retrieving many resources in just a single request to the backend, which can be described as follows:

- **Scenario 6:** Querying the *Id* and *Name Entity Type Properties* of the first 100 instances of the *Folder*, *Facility*, *Product*, *Material* and *Area Entity Types*. It evaluates the querying of multiple Entity Types, which must be done in separate requests with the REST API.

The same measuring methods used for the previous five scenarios were employed for this new one, except that this time we only recorded the times displayed by the GraphQL API since we can assume that the REST API behaves in the same way as Scenario 1, with the difference being that a total of five requests are needed. The results obtained for this case can be seen in table 6.6.

Looking at the measurements acquired for this last scenario is rather intriguing. If we compare them with those of Scenario 1, it takes about double the amount of time to complete a request, but this time we retrieve five times the amount of information. As one could expect, GraphQL's primary strengths still show up even in a dynamic data model scenario.

6.2.2 Developer Experience

Another essential aspect when accessing an API is the developer experience it provides. Developer experience is defined as a user experience from a developer's point of view, represented by the tools, processes, and software that a developer uses when interacting with a product or system [19].

Table 6.6: Response time average per ten requests in 10 groups of trial tests - Scenario 6

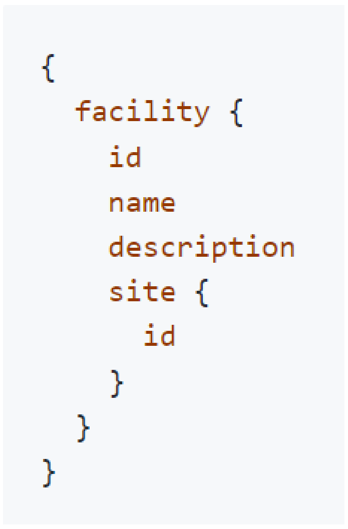
Trial Test	**GraphQL (ms)**
1	131.1
2	154.3
3	111.2
4	111.5
5	143.2
1	125.3
2	121.4
3	93.4
4	122.7
5	112.6
Total Average	122.7

To measure how both APIs compare on this element, a small questionnaire was prepared asking the developers of the CM company a few questions regarding it, with a total of 16 responses being collected, whose complete results can be seen in the appendix A.

In the first section of the questionnaire, a characterization of the CM developers was made, focusing on their knowledge of both APIs technologies. From the responses obtained, we can derive that their understanding of REST is significantly greater than their knowledge of GraphQL. Everyone but one developer answered that they had used REST before, while when it comes to GraphQL, only 12.5% had come in contact with the technology, with 50% of them not knowing what it is used for. Still, when questioned what their understanding of the GraphQL query seen in Figure 6.1 was, 14 of the 16 said that they were able to grasp its purpose. These answers help support the idea that GraphQL is still somewhat new in the developer world and that its intuitive query language may be one of the main drivers of its growing popularity.

In the following sections, the developers were asked their opinions regarding the ease of understanding and writing the payloads used to build the requests for the REST and GraphQL APIs. Four different request scenarios were asked about, which evaluate the same aspects as Scenarios 1, 3, 4, and 5 from the previous performance section. For each of these cases, respondents should choose their preference, if existent, of which API to use when it comes to the two factors previously mentioned.

For the questions related to Scenarios 1, 3, and 4, GraphQL is the clear winner, with no developer considering the payloads used to construct the requests to the REST API easier to either write or understand, and with only two occurrences in the six total questions of a developer showing no preference for one or the other. Figure 6.2 displays the REST and GraphQL payloads that are needed to query information similar to Scenario 1, becoming apparent why those were the responses obtained, with the GraphQL one being much shorter and intuitive. Still, we should point out that the CM MES REST API utilizes a lot of extra metadata not used by the developed mid-



```
{
  facility {
    id
    name
    description
    site {
      id
    }
  }
}
```

Figure 6.1: GraphQL query used in questionnaire for understanding developers knowledge

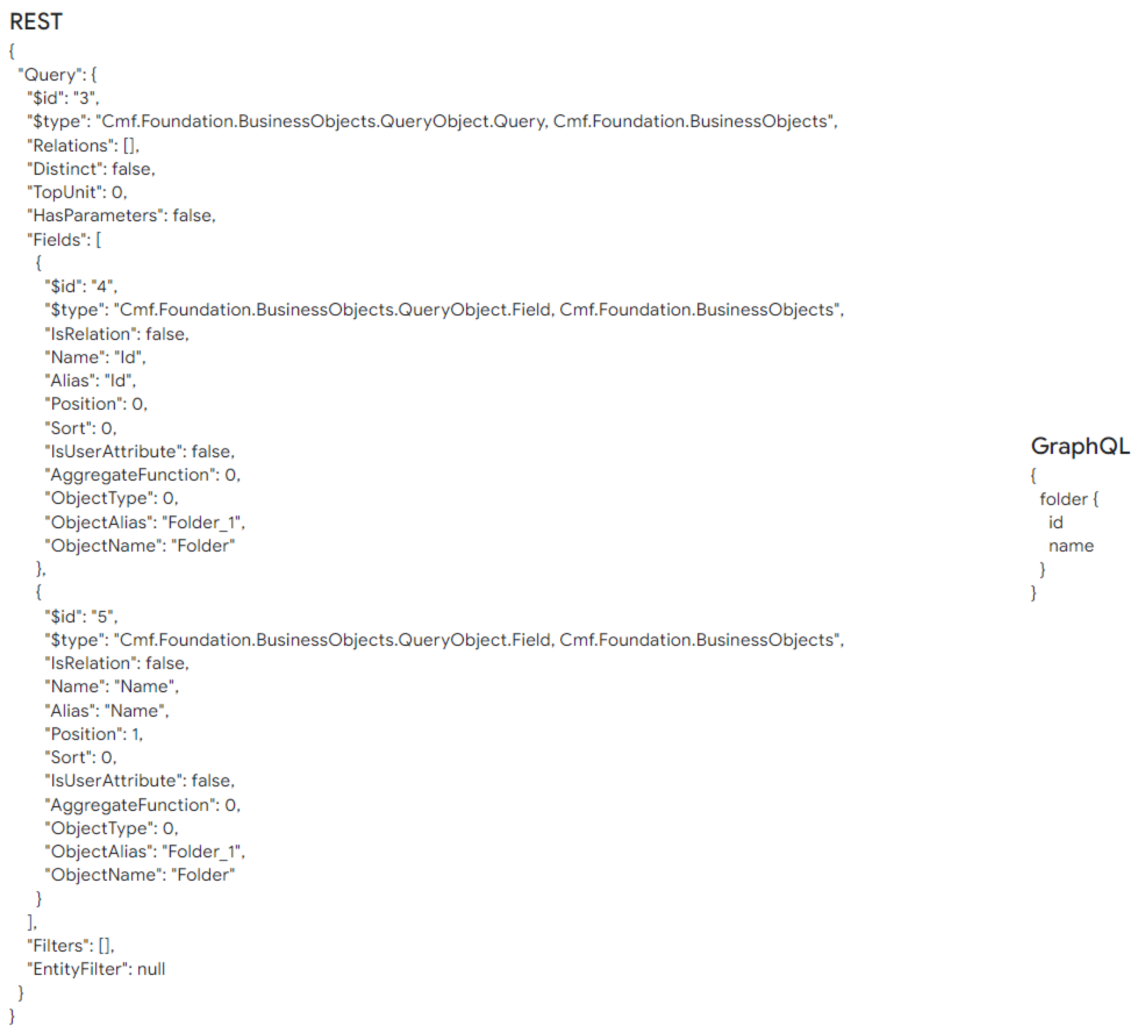
dleware, which ends up cluttering its payload. Thus, this comparison might not be entirely fair, but it is still a good indicator since we can assume that the CM MES developers are aware of that and can see past it and only focus on the pertinent pieces.

There are some split answers for the two questions related to Scenario 5. 75% of the respondents find the GraphQL payload easier to understand, and the remaining 25% prefer the REST one. As for the ease of writing it, there are still 75% in favor of GraphQL, but this time only 12.5% choose REST, with the remaining not having a preference. Since Scenario 5 asks for the same information as Scenario 1, but with the addition of the searching criteria, we can infer that the filtering solution developed is not optimal as it makes some developers switch their answers from GraphQL to REST.

After assessing the payloads used for the requests, it's time to focus on the ones associated with the responses retrieved. The responses payloads asked about are only those related to Scenarios 1 and 3 since the remaining ones present a similar response to the first one due to the absence of nested fields. Furthermore, only the ease of understanding the payload was evaluated since it is generated in the backend by a server and not written on the client-side by a developer. For the two questions that were posed, there were mixed answers, but GraphQL comes once again on top with 75% preferring it for Scenario 1 and 81.3% for Scenario 3. Still, these results should be viewed with skepticism since both API payloads are highly similar, with once more the main difference being the inclusion of extra metadata, and as such, we should not jump to the conclusion that GraphQL is better in this respect.

Finally, in the questionnaire's last section, some additional questions were made to determine the developer's view of the technology and the solution devised. On a *Likert* scale[2] from one to five, where one means "Strongly Disagree" and five means "Strongly Agree", 62.6% fall on the "Agree" section when asked if they would like to see CM MES utilize GraphQL, with only

[2]https://en.wikipedia.org/wiki/Likert_scale



REST

```
{
 "Query": {
  "$id": "3",
  "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Query, Cmf.Foundation.BusinessObjects",
  "Relations": [],
  "Distinct": false,
  "TopUnit": 0,
  "HasParameters": false,
  "Fields": [
   {
    "$id": "4",
    "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
    "IsRelation": false,
    "Name": "Id",
    "Alias": "Id",
    "Position": 0,
    "Sort": 0,
    "IsUserAttribute": false,
    "AggregateFunction": 0,
    "ObjectType": 0,
    "ObjectAlias": "Folder_1",
    "ObjectName": "Folder"
   },
   {
    "$id": "5",
    "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
    "IsRelation": false,
    "Name": "Name",
    "Alias": "Name",
    "Position": 1,
    "Sort": 0,
    "IsUserAttribute": false,
    "AggregateFunction": 0,
    "ObjectType": 0,
    "ObjectAlias": "Folder_1",
    "ObjectName": "Folder"
   }
  ],
  "Filters": [],
  "EntityFilter": null
 }
}
```

GraphQL

```
{
 folder {
  id
  name
 }
}
```

Figure 6.2: Comparison of REST and GraphQL payloads for Scenario 1

one developer falling on the "Disagree" section and the rest 31.3% on the neutral side. This is an extremely positive takeaway since it helps validate that the GraphQL API produced is indeed worthwhile to them and legitimizes the previous results regarding the payloads. However, when asked on the same scale if they find the ability to use nested filters within GraphQL important, such as filtering *Folder Entity Types* based on its *ParentFolder Name*, all 16 answers fall on the "Agree" section. As such, we can conclude that the filtering solution implemented is not sufficient for usage past the proof-of-concept stage and is something that needs to be improved in the API for it to become as viable as the current REST one.

6.2.3 Usability

Finally, the last characteristic considered in comparing both APIs is usability. By usability, we mean the degree to which an API can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use [10].

There are some obvious cases where GraphQL is not the choice to go with within CM MES. One of them is when there is the need to apply filtering based on nested fields since the GraphQL API does not provide this functionality at the moment, unlike its REST counterpart. Another is when there is querying of *Entity Type Properties* that refer to some *Entity Type* since at least two inquiries to the database will be made by the GraphQL API. In contrast, the CM MES REST API only interrogates the database once, allowing for higher performance than the developed solution.

Regarding CM MES's most common use case, which is querying an *Entity Type* scalar type *Entity Type Properties* without nested filtering, both APIs are a valid choice. The two APIs present similar performance, so the choice of which one to use falls on the developer, who should choose whichever he finds more convenient and easier to utilize based on the previously mentioned developer experience.

One use case where the new GraphQL API is for sure better than the CM MES REST API is when there is the need for querying multiple *Entity Types*. Since the current REST API only allows the querying of one *Entity Type* at a time, if, let's say, there is a need to query five different *Entity Types*, five separate requests to this API need to be made. This is where GraphQL shines since, on the other hand, the GraphQL API is capable of fulfilling this demand in just a single request, reducing the amount of throughput that the backend receives, decreasing the time spent on processing requests, and providing a better experience to the developer.

6.3 Discussion

After comparing the previous three aspects regarding both APIs, we can say that there is not a clear winner, with both APIs having their pros and cons.

Even though the newly developed GraphQL API performs better by a small fraction in most realistic use cases, it falls behind by some margin or is unusable in others. Still, some improvements can be made, which can turn the GraphQL API into a more powerful contender for querying the CM MES dynamic data model.

Where the new GraphQL API excels is on the developer experience, being elected as the preferred way for the CM company developers to query the *Entity Types* of its data model. On top of this, the standard GraphQL benefits remain present, allowing the querying of multiple pieces of information in just a single request to the backend, making it an excellent choice to go with when this use case is needed.

Chapter 7

Conclusions and Future Developments

In this final chapter, we present the conclusions drawn, along with the main contributions to the area. Finally, some possible further developments and evolution for the work are outlined.

7.1 Main Developments and Conclusions

The application of GraphQL for dynamic data models was still uncertain, with doubts existing as to whether this use case was possible and what possible advantages it could bring over a more traditional approach such as using a REST architecture.

With the work developed, we can conclude that GraphQL technology can indeed be used to construct APIs on top of applications that possess a dynamic data model. Even though the developed solution is a mere proof-of-concept, and GraphQL was not used extensively, missing some standard features like mutations and subscriptions, we can confidently say that employing a GraphQL API for the previous scenario is a sound choice.

The usage of GraphQL technology for the mentioned use case also brings some pros and cons compared to going with a more conventional REST approach.

When it comes to its advantages, GraphQL's main forces still shine, with the ability to ask the server for specific parts of information with a single roundtrip to the backend bringing great value to a client and allowing for a sharp performance. On top of this, the fact that GraphQL queries are simple and easy to understand even by developers not acquainted with the technology makes it a solid option to ensure a great developer experience for an API.

However, the GraphQL protocol can impair the performance and usability of an API that uses this technology. When it comes to performance, the fact that a GraphQL API should resolve fields based on levels means that in some cases, multiple inquiries will be made to a database, while with a REST approach, we can optimize the query execution by aggregating all needed data just from a single fetch from the database. Regarding usability, filtering query results based on nested fields is something that is rather complex to achieve with GraphQL and can make an API with this technology unusable for this scenario.

Still, we can say that overall, a GraphQL API can substitute a REST one for querying dynamic data models, showing similar performance for most use cases and providing some benefits over the latter.

7.2 Further Developments and Evolution

There is still a wide range of possible features and enhancements related to the work in this dissertation that can be investigated and implemented. Of those, we consider the following the most compelling:

- Explore the applicability of GraphQL for dynamic data models more extensively through the usage of mutation and subscription operation types;
- Improve the performance of the GraphQL API via methods such as caching;
- Try out other C# / .NET GraphQL libraries and programming languages for the GraphQL service to see how they compare to the chosen technologies in terms of performance;
- Analyze and develop new alternative ways to improve the filtering of GraphQL query results;
- Explore schema patching of the GraphQL API during runtime rather than total schema replacement;
- Further test the GraphQL API in a production environment to see how it stacks up against the REST API in everyday tasks.

References

[1] Altair. Altair Graphql Client, 2022. Available at https://altair.sirmuel.design/, last accessed in May 2022.

[2] Ardalis. Bus or Queue | Blog, 2022. Available at https://ardalis.com/bus-or-queue/, last accessed in May 2022.

[3] Contentful. GraphQL dynamic schema generation for changing data models | Contentful, 2018. Available at https://www.contentful.com/blog/2018/12/21/dynamic-schema-generation-changing-data-models/, last accessed in February 2022.

[4] Critical Manufacturing. Critical Manufacturing - Weigh & Dispense, 2022. Available at https://www.criticalmanufacturing.com/mes-for-industry-4-0/weigh-dispense/, last accessed in May 2022.

[5] Dgraph. Dgraph Database Overview, 2022. Available at https://dgraph.io/docs/dgraph-overview/, last accessed in May 2022.

[6] Carles Farré, Jovan Varga, and Robert Almar. Graphql Schema Generation for Data-Intensive Web APIs. Technical report, Universitat Politècnica de Catalunya, BarcelonaTech, 2019.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] GraphQL .NET. GraphQL .NET, 2022. Available at https://graphql-dotnet.github.io/docs/guides/dataloader/, last accessed in June 2022.

[9] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[10] ISO 9241-11. Ergonomic Requirements for Office Work with Visual Display Terminals. Standard, International Organization for Standardization, Geneva, 1998.

[11] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, 1998.

[12] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education, 2003.

[13] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.

[14] Michael McClellan. *Applying Manufacturing Execution Systems*. CRC Press, 1997.

[15] Microsoft. ORDER BY Clause (Transact-SQL) - SQL Server, 2022. Available at https://docs.microsoft.com/en-us/sql/t-sql/queries/select-order-by-clause-transact-sql?view=sql-server-ver16, last accessed in May 2022.

[16] Microsoft. What is ASP.NET Core? | .NET, 2022. Available at https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet-core, last accessed in May 2022.

[17] Microsoft. What is NuGet and what does it do?, 2022. Available at https://docs.microsoft.com/en-us/nuget/what-is-nuget, last accessed in May 2022.

[18] Netflix. Home - DGS Framework - Netflix Open Source, 2022. Available at https://netflix.github.io/dgs/, last accessed in February 2022.

[19] Prokop Simek. Good Developer Experience | Developer Experience Knowledge Base, 2022. Available at https://developerexperience.io/practices/good-developer-experience, last accessed in June 2022.

[20] State of JavaScript. State of JS 2020: Data Layer, February 2022. Available at https://2020.stateofjs.com/en-US/technologies/datalayer/,lastaccessedinFebruary2022.

[21] The GraphQL Foundation. Github - graphql/dataloader: Dataloader is a generic utility to be used as part of your application's data fetching layer to provide a consistent API over various backends and reduce requests to those backends via batching and caching., 2022. Available at https://github.com/graphql/dataloader, last accessed in June 2022.

[22] The GraphQL Foundation. Github - graphql/graphiql: GraphiQL & the Graphql LSP Reference Ecosystem for building browser & IDE tools., 2022. Available at https://github.com/graphql/graphiql, last accessed in May 2022.

[23] The GraphQL Foundation. Github - graphql/graphql-js: A reference implementation of GraphQL for JavaScript, 2022. Available at https://github.com/graphql/graphql-js, last accessed in May 2022.

[24] The GraphQL Foundation. GraphQL, 2022. Available at https://spec.graphql.org/October2021/, last accessed in May 2022.

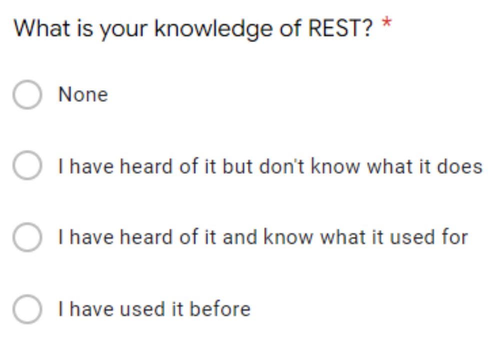
[25] The GraphQL Foundation. GraphQL | A query language for your API, 2022. Available at https://graphql.org/, last accessed in February 2022.

Appendix A

Developer Experience Questionnaire

The complete results of the developer experience questionnaire described in section 6.2.2 are presented in the following passages.

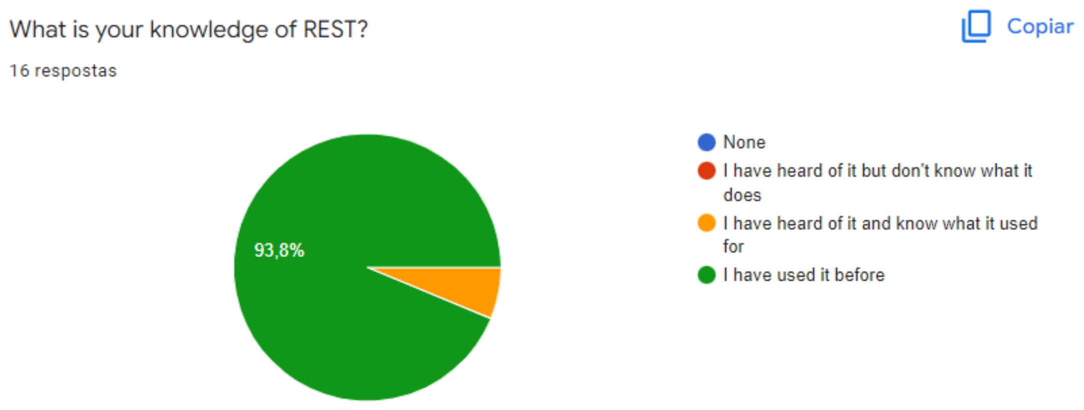
A.1 API Technologies Knowledge



What is your knowledge of REST? *

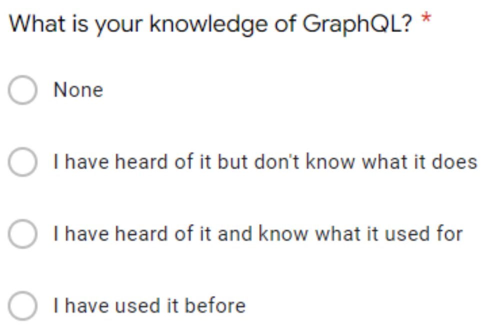
- ○ None
- ○ I have heard of it but don't know what it does
- ○ I have heard of it and know what it used for
- ○ I have used it before

Figure A.1: Developer Experience questionnaire first question


What is your knowledge of REST?

16 respostas

Figure A.2: Developer Experience questionnaire first question results


What is your knowledge of GraphQL? *

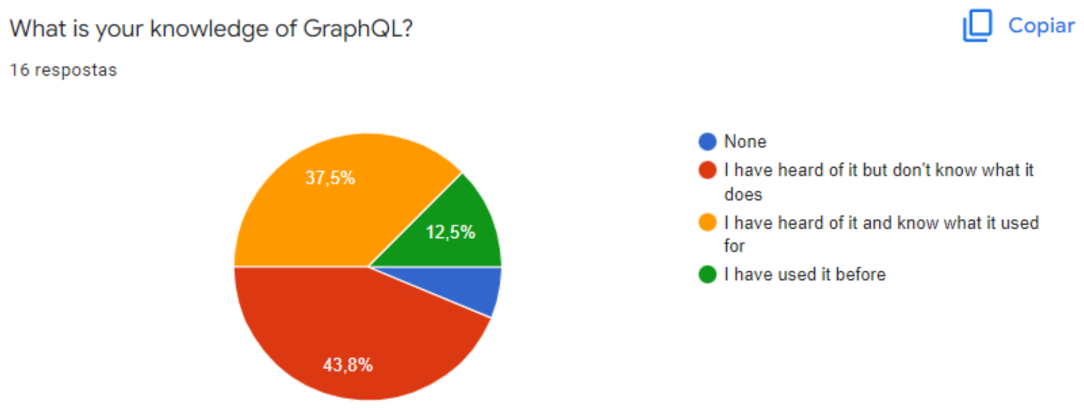
◯ None

◯ I have heard of it but don't know what it does

◯ I have heard of it and know what it used for

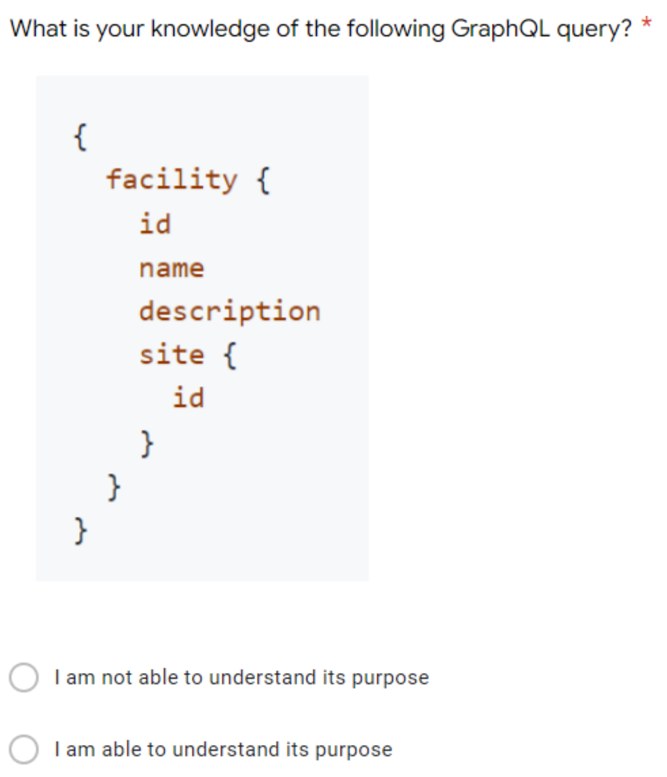
◯ I have used it before

Figure A.3: Developer Experience questionnaire second question


What is your knowledge of GraphQL?

16 respostas

Figure A.4: Developer Experience questionnaire second question results

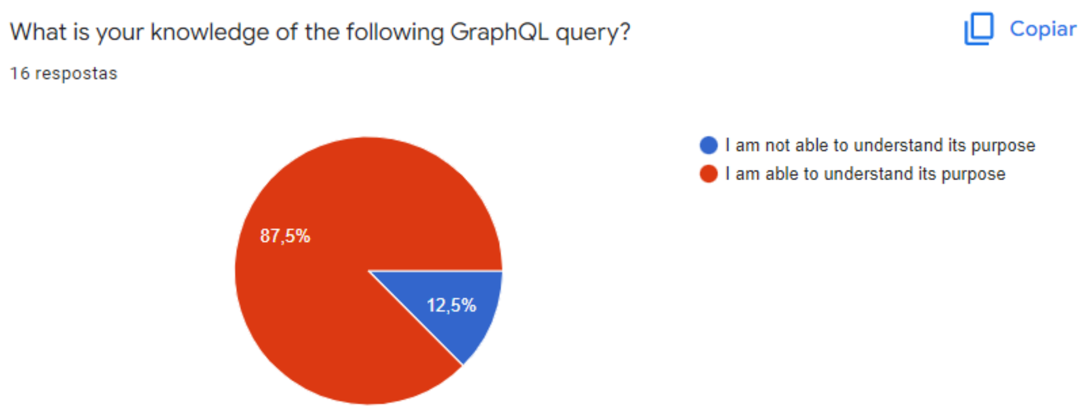

What is your knowledge of the following GraphQL query? *

```
{
  facility {
    id
    name
    description
    site {
      id
    }
  }
}
```

○ I am not able to understand its purpose

○ I am able to understand its purpose

Figure A.5: Developer Experience questionnaire third question


What is your knowledge of the following GraphQL query?

16 respostas

Figure A.6: Developer Experience questionnaire third question results

A.2 REST vs GraphQL: Request Payloads

A.2.1 Request Payloads 1

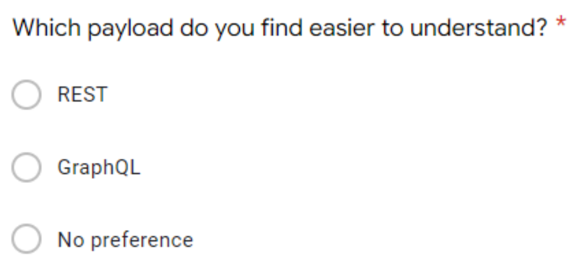
The following payloads are used for requesting the the *Id* and *Name Entity Type Properties* of some *Folder Entity Type* instances.

REST payload:

```
{
  "Query": {
    "$id": "3",
    "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Query, Cmf.Foundation.BusinessObjects",
    "Relations": [],
    "Distinct": false,
    "TopUnit": 0,
    "HasParameters": false,
    "Fields": [
      {
        "$id": "4",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "IsRelation": false,
        "Name": "Id",
        "Alias": "Id",
        "Position": 0,
        "Sort": 0,
        "IsUserAttribute": false,
        "AggregateFunction": 0,
        "ObjectType": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder"
      },
      {
        "$id": "5",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "IsRelation": false,
        "Name": "Name",
        "Alias": "Name",
        "Position": 1,
        "Sort": 0,
        "IsUserAttribute": false,
        "AggregateFunction": 0,
        "ObjectType": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder"
      }
    ],
    "Filters": [],
    "EntityFilter": null
  }
}
```

GraphQL payload:

```
{
  folder {
    id
    name
  }
}
```



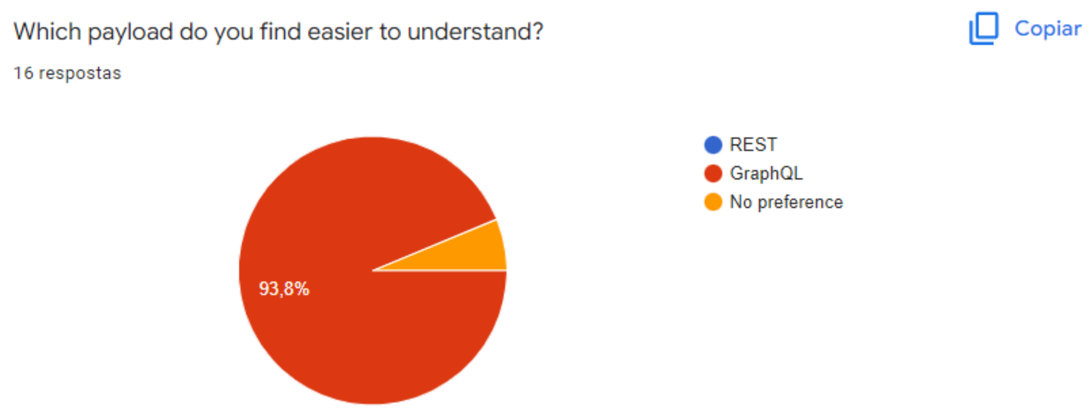
Which payload do you find easier to understand? *

( ) REST

( ) GraphQL

( ) No preference

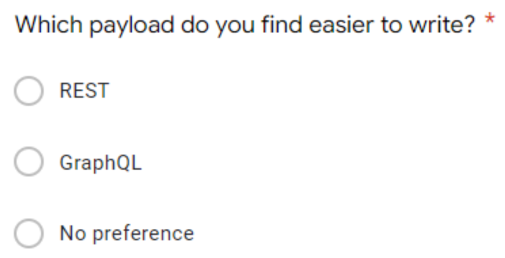
Figure A.7: Developer Experience questionnaire fourth question



Which payload do you find easier to understand?

16 respostas

Figure A.8: Developer Experience questionnaire fourth question results



Which payload do you find easier to write? *

( ) REST

( ) GraphQL

( ) No preference

Figure A.9: Developer Experience questionnaire fifth question

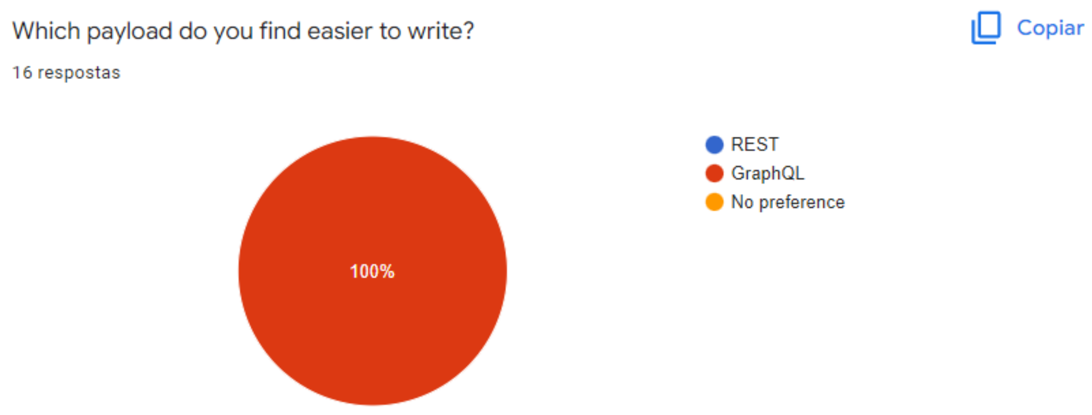


Figure A.10: Developer Experience questionnaire fifth question results

A.2.2 Request Payloads 2

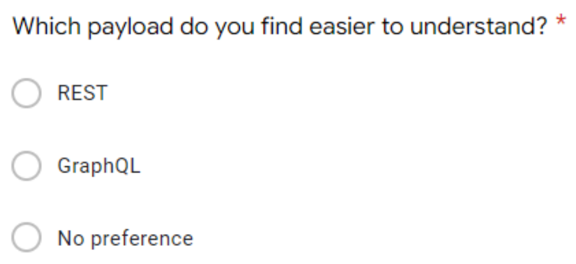
The following payloads are used for requesting the the *Id*, *Name*, *ParentFolder Id* and *ParentFolder Name Entity Type Properties* of some *Folder Entity Type* instances.

REST payload:

```
{
  "Query": {
    "$id": "3",
    "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Query, Cmf.Foundation.BusinessObjects",
    "Fields": [
      {
        "$id": "4",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Id",
        "Alias": "Id",
        "IsUserAttribute": false,
        "Position": 0,
        "Sort": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      },
      {
        "$id": "5",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Name",
        "Alias": "Name",
        "IsUserAttribute": false,
        "Position": 1,
        "Sort": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      },
      {
        "$id": "6",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Id",
        "Alias": "ParentFolderId",
        "IsUserAttribute": false,
        "Position": 2,
        "Sort": 0,
        "ObjectAlias": "Folder_ParentFolder_2",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      },
      {
        "$id": "7",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Name",
        "Alias": "ParentFolderName",
        "IsUserAttribute": false,
        "Position": 3,
        "Sort": 0,
        "ObjectAlias": "Folder_ParentFolder_2",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      }
    ],
    "Filters": [],
    "Relations": [
      {
        "$id": "8",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Relation, Cmf.Foundation.BusinessObjects",
        "Alias": "",
        "IsRelation": false,
        "Name": "",
        "SourceEntity": "Folder",
        "SourceEntityAlias": "Folder_1",
        "SourceJoinType": 0,
        "SourceProperty": "ParentFolderId",
        "TargetEntityAlias": "Folder_ParentFolder_2",
        "TargetJoinType": 0,
        "TargetProperty": "Id",
        "TargetEntity": "Folder"
      }
    ],
    "Distinct": false,
    "TopUnit": 0,
    "QueryParameters": [],
    "EntityFilter": null
  }
}
```

GraphQL payload:

```
{
  folder {
    id
    name
    parentFolder {
      id
      name
    }
  }
}
```


Which payload do you find easier to understand? *

- REST
- GraphQL
- No preference

Figure A.11: Developer Experience questionnaire sixth question

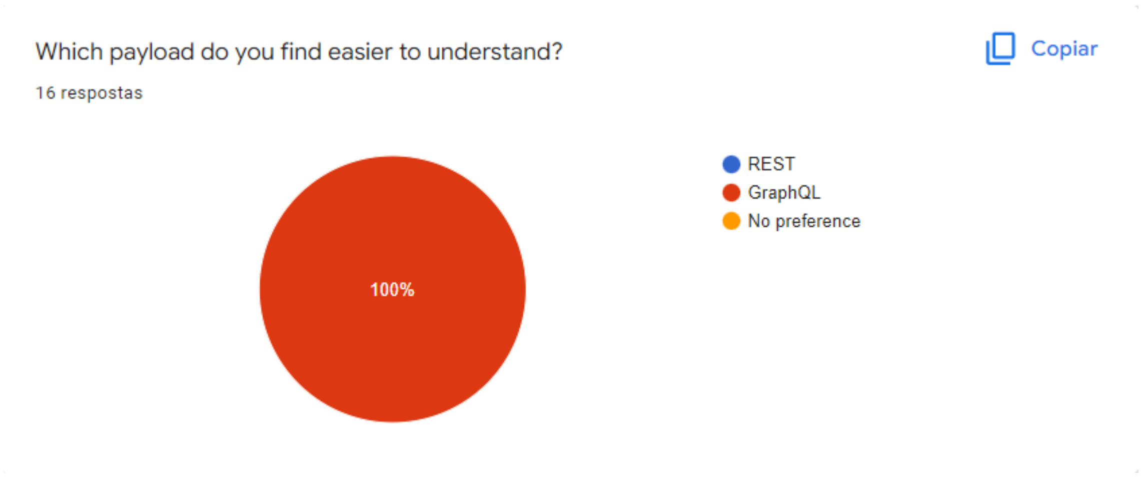
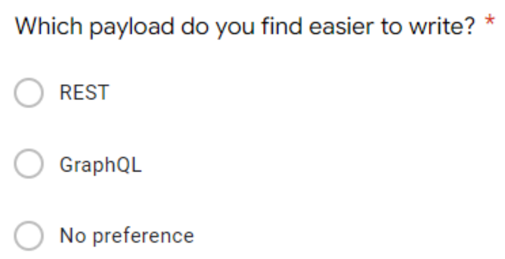


Figure A.12: Developer Experience questionnaire sixth question results



Which payload do you find easier to write? *

○ REST

○ GraphQL

○ No preference

Figure A.13: Developer Experience questionnaire seventh question

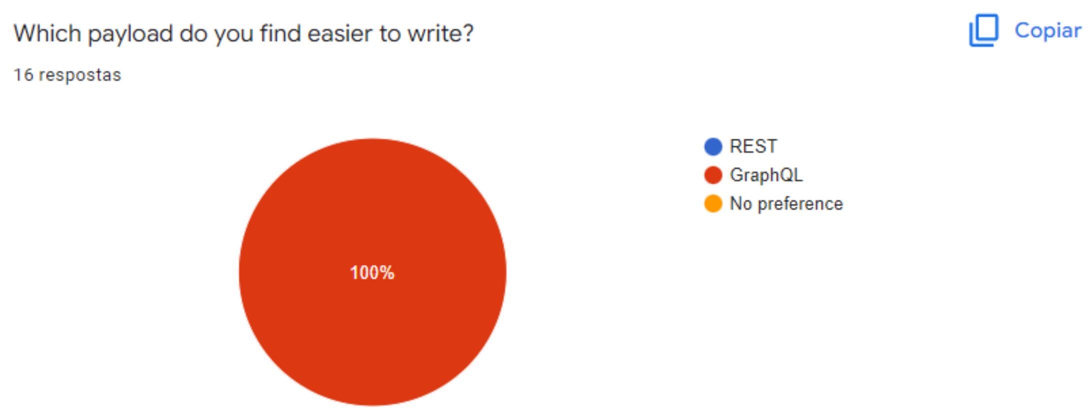


Figure A.14: Developer Experience questionnaire seventh question results

A.2.3 Request Payloads 3

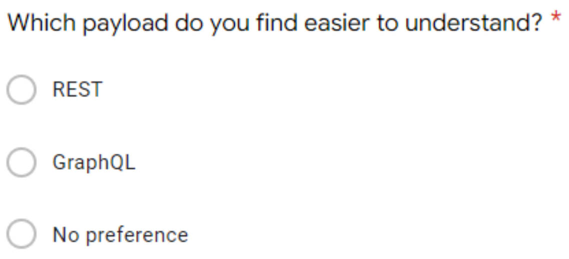
The following payloads are used for requesting the the *Id* and *Name Entity Type Properties* of some *Folder Entity Type* instances whose *Name* starts with the letter "D".

REST payload:

```
{
  "Query": {
    "$id": "3",
    "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Query, Cmf.Foundation.BusinessObjects",
    "Fields": [
      {
        "$id": "4",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Id",
        "Alias": "Id",
        "IsUserAttribute": false,
        "Position": 0,
        "Sort": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      },
      {
        "$id": "5",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Name",
        "Alias": "Name",
        "IsUserAttribute": false,
        "Position": 1,
        "Sort": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      }
    ],
    "Filters": [
      {
        "$id": "6",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
        "Name": "Name",
        "Operator": 9,
        "LogicalOperator": 0,
        "ObjectName": "Folder",
        "ObjectAlias": "Folder_1",
        "Value": "D",
        "IsOptional": false
      }
    ],
    "Relations": [],
    "Distinct": false,
    "TopUnit": 0,
    "QueryParameters": [],
    "EntityFilter": null
  }
}
```

GraphQL payload:

```
{
  folder (filtering: "{ field: 'name', operation: 'STARTS_WITH', value: 'D' }") {
    id
    name
  }
}
```

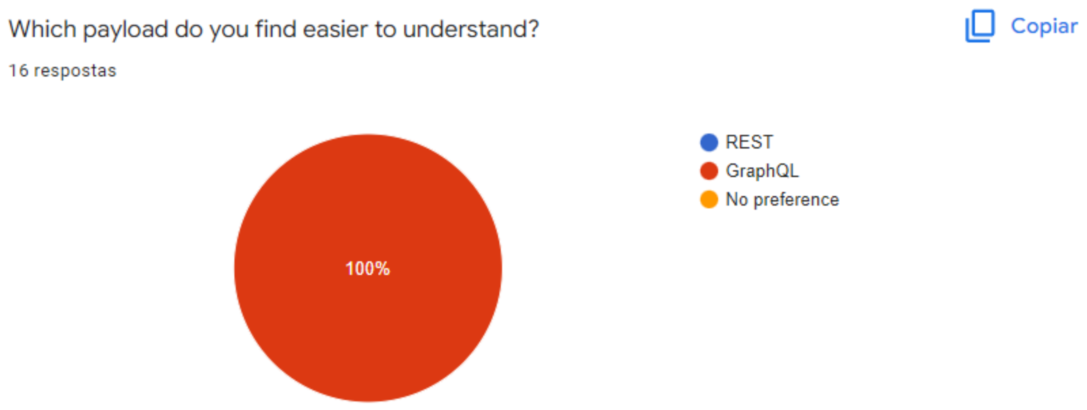

Which payload do you find easier to understand? *

◯ REST

◯ GraphQL

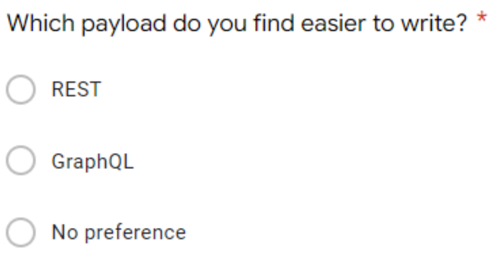
◯ No preference

Figure A.15: Developer Experience questionnaire eighth question


Which payload do you find easier to understand?

16 respostas

Figure A.16: Developer Experience questionnaire eighth question results


Which payload do you find easier to write? *

◯ REST

◯ GraphQL

◯ No preference

Figure A.17: Developer Experience questionnaire ninth question

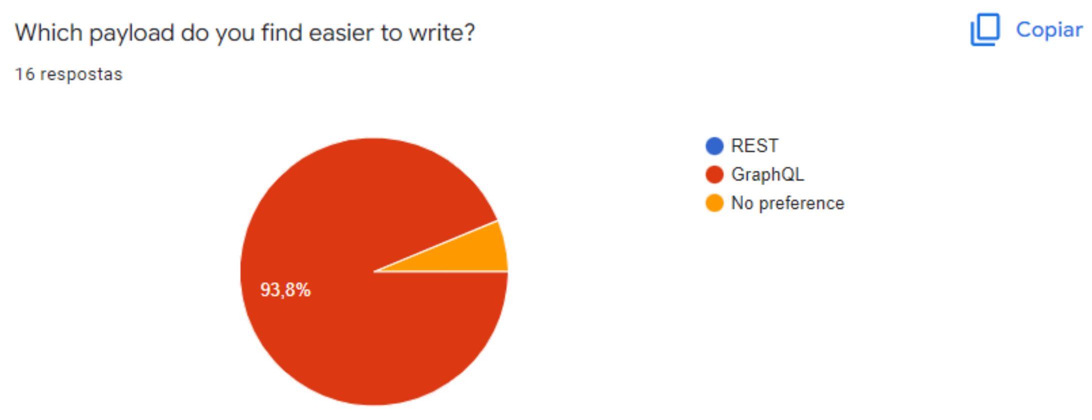


Figure A.18: Developer Experience questionnaire ninth question results

A.2.4 Request Payloads 4

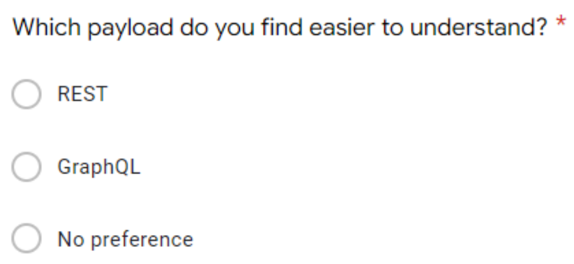
The following payloads are used for requesting the the *Id* and *Name Entity Type Properties* of some *Folder Entity Type* instances whose *Name* starts with the letter "D" or whose *Id* is greater than 1805111613350000005 and lesser than 1805111613350000008.

REST payload:

```
{
  "Query": {
    "$id": "3",
    "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Query, Cmf.Foundation.BusinessObjects",
    "Fields": [
      {
        "$id": "4",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Id",
        "Alias": "Id",
        "IsUserAttribute": false,
        "Position": 0,
        "Sort": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      },
      {
        "$id": "5",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Field, Cmf.Foundation.BusinessObjects",
        "Name": "Name",
        "Alias": "Name",
        "IsUserAttribute": false,
        "Position": 1,
        "Sort": 0,
        "ObjectAlias": "Folder_1",
        "ObjectName": "Folder",
        "AggregateFunction": 0
      }
    ],
    "Filters": [
      {
        "$id": "6",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
        "Name": "Name",
        "Operator": 9,
        "LogicalOperator": 2,
        "ObjectName": "Folder",
        "ObjectAlias": "Folder_1",
        "Value": "D",
        "IsOptional": false
      },
      {
        "$id": "7",
        "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
        "FilterType": 2,
        "LogicalOperator": 1,
        "InnerFilter": [
          {
            "$id": "8",
            "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
            "Name": "Id",
            "Operator": 2,
            "LogicalOperator": 1,
            "ObjectName": "Folder",
            "ObjectAlias": "Folder_1",
            "Value": "1805111613350000005",
            "IsOptional": false
          },
          {
            "$id": "9",
            "$type": "Cmf.Foundation.BusinessObjects.QueryObject.Filter, Cmf.Foundation.BusinessObjects",
            "Name": "Id",
            "Operator": 4,
            "LogicalOperator": 0,
            "ObjectName": "Folder",
            "ObjectAlias": "Folder_1",
            "Value": "1805111613350000008",
            "IsOptional": false
          }
        ]
      }
    ],
    "Relations": [],
    "Distinct": false,
    "TopUnit": 0,
    "QueryParameters": [],
    "EntityFilter": null
  }
}
```

GraphQL payload:

```
{
  folder (filtering: "[ 'OR', { field: 'name', operation: 'STARTS_WITH', value: 'D' }, [ 'AND', {field: 'id',
       operation: 'IS_GREATER_THAN', value: '1805111613350000005' }, { field: 'id', operation: 'IS_LESSER_THAN',
       value: '1805111613350000008' } ] ]") {
    id
    name
  }
}
```


Which payload do you find easier to understand? *

◯ REST

◯ GraphQL

◯ No preference

Figure A.19: Developer Experience questionnaire tenth question

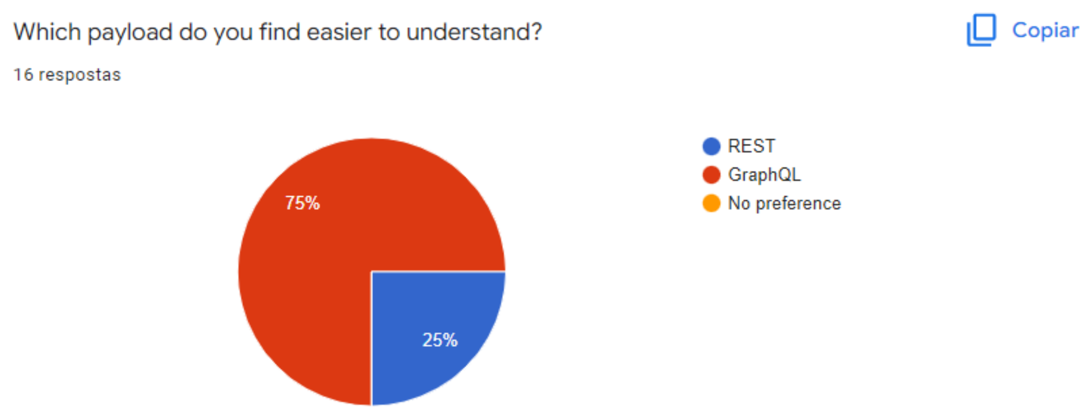
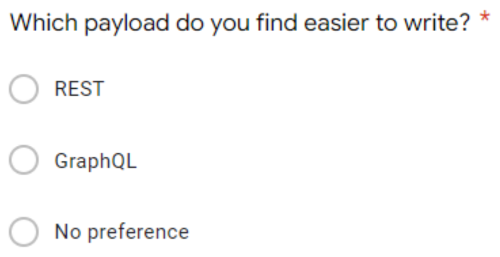


Figure A.20: Developer Experience questionnaire tenth question results


Which payload do you find easier to write? *

○ REST

○ GraphQL

○ No preference

Figure A.21: Developer Experience questionnaire eleventh question

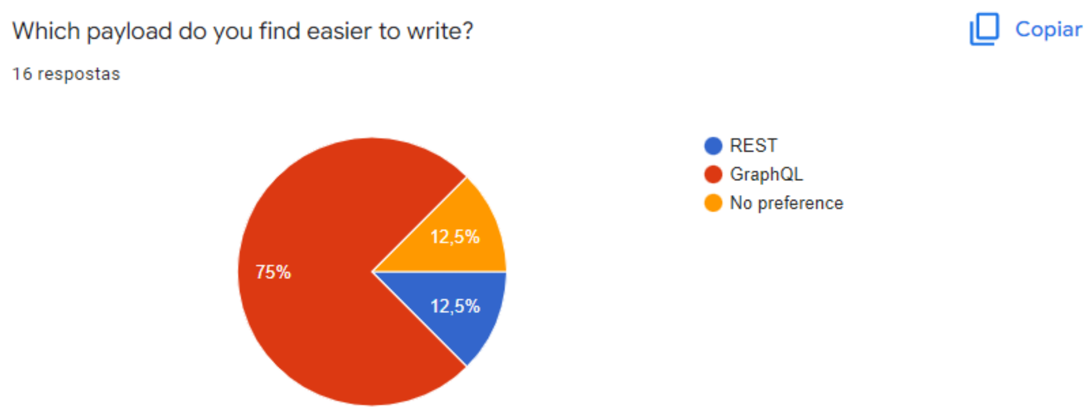


Figure A.22: Developer Experience questionnaire eleventh question results

A.3 REST vs GraphQL: Response Payloads

A.3.1 Response Payloads 1

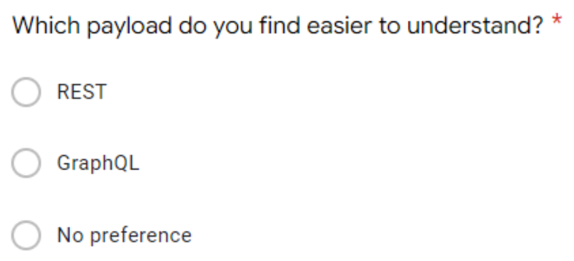
The following payloads are sent as a response to requesting the *Id* and *Name Entity Type Properties* of some *Folder Entity Type* instances.

REST payload:

```
{
  "$id": "1",
  "$type": "Cmf.Foundation.BusinessOrchestration.QueryManagement.OutputObjects.ExecuteQueryOutput, Cmf.Foundation.
        BusinessOrchestration",
  "NgpDataSet": {
    "T_Result": [
      {
        "Id": "18051116133500000001",
        "Name": "\\",
        "TotalRows": 451
      },
      {
        "Id": "18051116133500000002",
        "Name": "Documents",
        "TotalRows": 451
      },
      {
        "Id": "18051116133500000003",
        "Name": "Queries",
        "TotalRows": 451
      },
      {
        "Id": "18051116133500000004",
        "Name": "fabLives",
        "TotalRows": 451
      },
      {
        "Id": "18051116133500000005",
        "Name": "Dashboards",
        "TotalRows": 451
      },
      {
        "Id": "18051116133500000006",
        "Name": "Widgets",
        "TotalRows": 451
      },
      {
        "Id": "18051116133500000007",
        "Name": "Multimedia",
        "TotalRows": 451
      },
      {
        "Id": "22053000000000000001",
        "Name": "Z_Automatic_Created_Test_Folder_2fc826def4674450b1db0d81d69147c8882",
        "TotalRows": 451
      },
      {
        "Id": "22053000000000000002",
        "Name": "Z_Automatic_Created_Test_Folder_26e18cc767484205836ac60c40c67a7d127",
        "TotalRows": 451
      },
      {
        "Id": "22053000000000000003",
        "Name": "Z_Automatic_Created_Test_Folder_5c2e2eaeaea7462b9b1a76b18b824444801",
        "TotalRows": 451
      }
    ],
    "T_Result_Columns": [
      {
        "AllowDBNull": false,
        "AutoIncrement": false,
        "AutoIncrementSeed": "0",
        "AutoIncrementStep": "1",
        "Caption": "Id",
        "ColumnName": "Id",
        "Prefix": "",
        "DataType": "System.Int64, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
              cec85d7bea7798e",
        "DateTimeMode": 3,
        "DefaultValue": null,
        "Expression": "",
        "ExtendedProperties": {
          "$typeCMF": "CMFMap",
          "CMFMapData": []
        },
        "MaxLength": -1,
        "Namespace": "",
        "Ordinal": 0,
        "ReadOnly": false,
        "Unique": false,
        "ColumnMapping": 1,
        "Site": null,
        "Container": null,
        "DesignMode": false
      },
      {
        "AllowDBNull": false,
        "AutoIncrement": false,
        "AutoIncrementSeed": "0",
        "AutoIncrementStep": "1",
        "Caption": "Name",
        "ColumnName": "Name",
        "Prefix": "",
        "DataType": "System.String, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
              cec85d7bea7798e",
        "DateTimeMode": 3,
        "DefaultValue": null,
        "Expression": "",
        "ExtendedProperties": {
          "$typeCMF": "CMFMap",
          "CMFMapData": []
        },
        "MaxLength": 256,
        "Namespace": "",
        "Ordinal": 1,
        "ReadOnly": false,
        "Unique": false,
        "ColumnMapping": 1,
        "Site": null,
        "Container": null,
        "DesignMode": false
      },
      {
        "AllowDBNull": true,
        "AutoIncrement": false,
        "AutoIncrementSeed": "0",
        "AutoIncrementStep": "1",
        "Caption": "TotalRows",
        "ColumnName": "TotalRows",
        "Prefix": "",
        "DataType": "System.Int32, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
              cec85d7bea7798e",
        "DateTimeMode": 3,
        "DefaultValue": null,
        "Expression": "",
        "ExtendedProperties": {
          "$typeCMF": "CMFMap",
          "CMFMapData": []
        },
        "MaxLength": -1,
        "Namespace": "",
        "Ordinal": 2,
        "ReadOnly": false,
        "Unique": false,
        "ColumnMapping": 1,
        "Site": null,
        "Container": null,
        "DesignMode": false
      }
    ]
  },
  "TotalRows": 451
}
```

GraphQL payload:

```
{
  "data": {
    "folder": [
      {
        "id": 1805111613350000001,
        "name": "\\"
      },
      {
        "id": 1805111613350000002,
        "name": "Documents"
      },
      {
        "id": 1805111613350000003,
        "name": "Queries"
      },
      {
        "id": 1805111613350000004,
        "name": "fabLives"
      },
      {
        "id": 1805111613350000005,
        "name": "Dashboards"
      },
      {
        "id": 1805111613350000006,
        "name": "Widgets"
      },
      {
        "id": 1805111613350000007,
        "name": "Multimedia"
      },
      {
        "id": 2205300000000000001,
        "name": "Z_Automatic_Created_Test_Folder_2fc826def4674450b1db0d81d69147c8882"
      },
      {
        "id": 2205300000000000002,
        "name": "Z_Automatic_Created_Test_Folder_26e18cc767484205836ac60c40c67a7d127"
      },
      {
        "id": 2205300000000000003,
        "name": "Z_Automatic_Created_Test_Folder_5c2e2eaeaea7462b9b1a76b18b824444801"
      }
    ]
  }
}
```


Which payload do you find easier to understand? *

○ REST

○ GraphQL

○ No preference

Figure A.23: Developer Experience questionnaire twelfth question

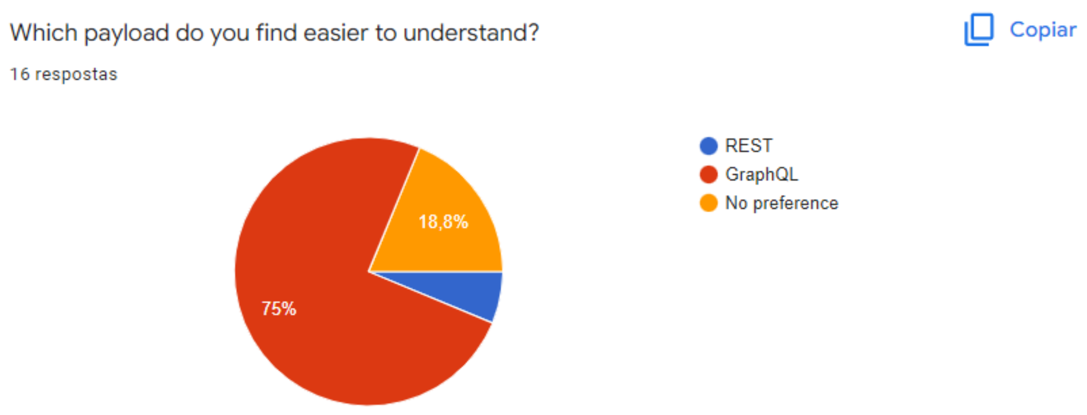


Figure A.24: Developer Experience questionnaire twelfth question results

A.3.2 Response Payloads 2

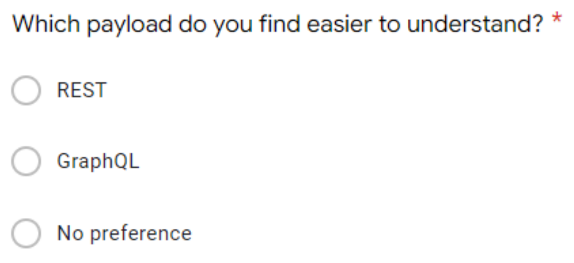
The following payloads are sent as a response to requesting the *Id*, *Name*, *ParentFolder Id* and *ParentFolder Name Entity Type Properties* of some *Folder Entity Type* instances.

REST payload:

```
{
  "$id": "1",
  "$type": "Cmf.Foundation.BusinessOrchestration.QueryManagement.OutputObjects.ExecuteQueryOutput, Cmf.Foundation.
       BusinessOrchestration",
  "NgpDataSet": {
    "T_Result": [
      {
        "Id": "1805111613350000002",
        "Name": "Documents",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "1805111613350000003",
        "Name": "Queries",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "1805111613350000004",
        "Name": "fabLives",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "1805111613350000005",
        "Name": "Dashboards",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "1805111613350000006",
        "Name": "Widgets",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "1805111613350000007",
        "Name": "Multimedia",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "2205300000000000001",
        "Name": "Z_Automatic_Created_Test_Folder_2fc826def4674450b1db0d81d69147c8882",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "2205300000000000002",
        "Name": "Z_Automatic_Created_Test_Folder_26e18cc767484205836ac60c40c67a7d127",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "2205300000000000003",
        "Name": "Z_Automatic_Created_Test_Folder_5c2e2eaeaea7462b9b1a76b18b824444801",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      },
      {
        "Id": "2205300000000000004",
        "Name": "Z_Automatic_Created_Test_Folder_b5b249a49cc64622bf166297e7249818735",
        "ParentFolderId": "1805111613350000001",
        "ParentFolderName": "\\",
        "TotalRows": 450
      }
    ],
    "T_Result_Columns": [
      {
        "AllowDBNull": false,
        "AutoIncrement": false,
        "AutoIncrementSeed": "0",
        "AutoIncrementStep": "1",
        "Caption": "Id",
        "ColumnName": "Id",
        "Prefix": "",
        "DataType": "System.Int64, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
            cec85d7bea7798e",
        "DateTimeMode": 3,
        "DefaultValue": null,
        "Expression": "",
        "ExtendedProperties": {
          "$typeCMF": "CMFMap",
          "CMFMapData": []
        },
        "MaxLength": -1,
        "Namespace": "",
        "Ordinal": 0,
        "ReadOnly": false,
        "Unique": false,
        "ColumnMapping": 1,
        "Site": null,
        "Container": null,
        "DesignMode": false
      },
      {
        "AllowDBNull": false,
        "AutoIncrement": false,
        "AutoIncrementSeed": "0",
        "AutoIncrementStep": "1",
        "Caption": "Name",
        "ColumnName": "Name",
        "Prefix": "",
        "DataType": "System.String, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
            cec85d7bea7798e",
        "DateTimeMode": 3,
        "DefaultValue": null,
        "Expression": "",
        "ExtendedProperties": {
          "$typeCMF": "CMFMap",
      "CMFMapData": []
    },
    "MaxLength": 256,
    "Namespace": "",
    "Ordinal": 1,
    "ReadOnly": false,
    "Unique": false,
    "ColumnMapping": 1,
    "Site": null,
    "Container": null,
    "DesignMode": false
  },
  {
    "AllowDBNull": false,
    "AutoIncrement": false,
    "AutoIncrementSeed": "0",
    "AutoIncrementStep": "1",
    "Caption": "ParentFolderId",
    "ColumnName": "ParentFolderId",
    "Prefix": "",
    "DataType": "System.Int64, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
        cec85d7bea7798e",
    "DateTimeMode": 3,
    "DefaultValue": null,
    "Expression": "",
    "ExtendedProperties": {
      "$typeCMF": "CMFMap",
      "CMFMapData": []
    },
    "MaxLength": -1,
    "Namespace": "",
    "Ordinal": 2,
    "ReadOnly": false,
    "Unique": false,
    "ColumnMapping": 1,
    "Site": null,
    "Container": null,
    "DesignMode": false
  },
  {
    "AllowDBNull": false,
    "AutoIncrement": false,
    "AutoIncrementSeed": "0",
    "AutoIncrementStep": "1",
    "Caption": "ParentFolderName",
    "ColumnName": "ParentFolderName",
    "Prefix": "",
    "DataType": "System.String, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
        cec85d7bea7798e",
    "DateTimeMode": 3,
    "DefaultValue": null,
    "Expression": "",
    "ExtendedProperties": {
      "$typeCMF": "CMFMap",
      "CMFMapData": []
    },
    "MaxLength": 256,
    "Namespace": "",
    "Ordinal": 3,
    "ReadOnly": false,
    "Unique": false,
    "ColumnMapping": 1,
    "Site": null,
    "Container": null,
    "DesignMode": false
  },
  {
    "AllowDBNull": true,
    "AutoIncrement": false,
    "AutoIncrementSeed": "0",
    "AutoIncrementStep": "1",
    "Caption": "TotalRows",
    "ColumnName": "TotalRows",
    "Prefix": "",
    "DataType": "System.Int32, System.Private.CoreLib, Version=6.0.0.0, Culture=neutral, PublicKeyToken=7
        cec85d7bea7798e",
    "DateTimeMode": 3,
          "DefaultValue": null,
          "Expression": "",
          "ExtendedProperties": {
            "$typeCMF": "CMFMap",
            "CMFMapData": []
          },
          "MaxLength": -1,
          "Namespace": "",
          "Ordinal": 4,
          "ReadOnly": false,
          "Unique": false,
          "ColumnMapping": 1,
          "Site": null,
          "Container": null,
          "DesignMode": false
        }
      ]
    },
    "TotalRows": 450
}
```

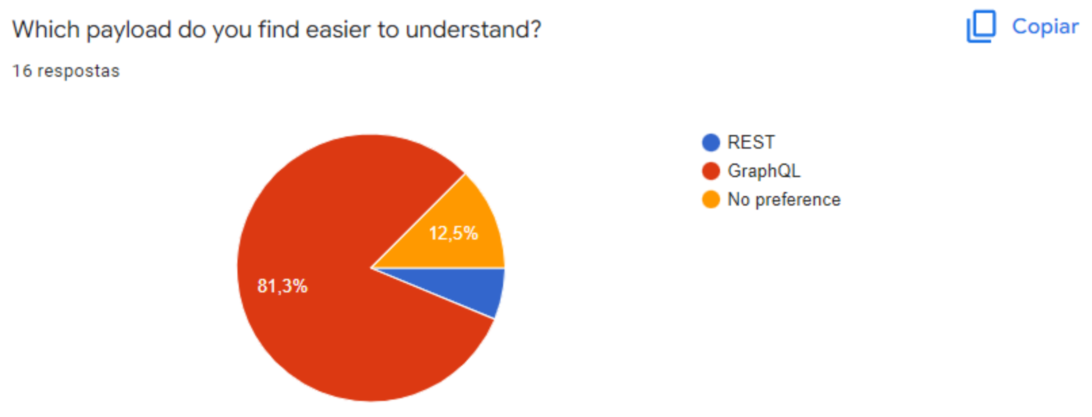
GraphQL payload:

```
{
  "data": {
    "folder": [
      {
        "id": 1805111613350000001,
        "name": "\\",
        "parentFolder": null
      },
      {
        "id": 1805111613350000002,
        "name": "Documents",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 1805111613350000003,
        "name": "Queries",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 1805111613350000004,
        "name": "fabLives",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 1805111613350000005,
        "name": "Dashboards",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 1805111613350000006,
        "name": "Widgets",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 1805111613350000007,
        "name": "Multimedia",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 2205300000000000001,
        "name": "Z_Automatic_Created_Test_Folder_2fc826def4674450b1db0d81d69147c8882",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 2205300000000000002,
        "name": "Z_Automatic_Created_Test_Folder_26e18cc767484205836ac60c40c67a7d127",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      },
      {
        "id": 2205300000000000003,
        "name": "Z_Automatic_Created_Test_Folder_5c2e2eaeaea7462b9b1a76b18b824444801",
        "parentFolder": {
          "id": 1805111613350000001,
          "name": "\\"
        }
      }
    ]
  }
}
```


Which payload do you find easier to understand? *

- REST
- GraphQL
- No preference

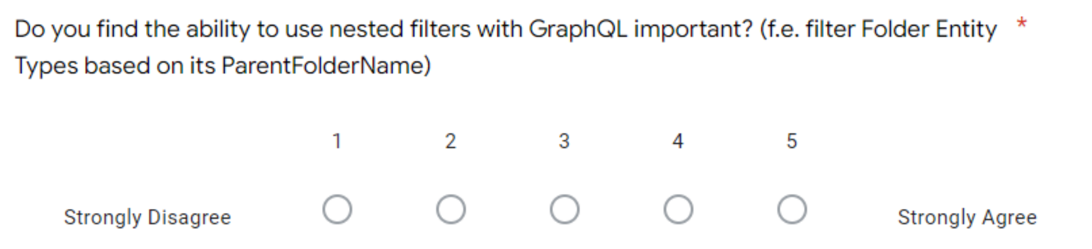
Figure A.25: Developer Experience questionnaire thirteenth question


Which payload do you find easier to understand?

16 respostas

Figure A.26: Developer Experience questionnaire thirteenth question results

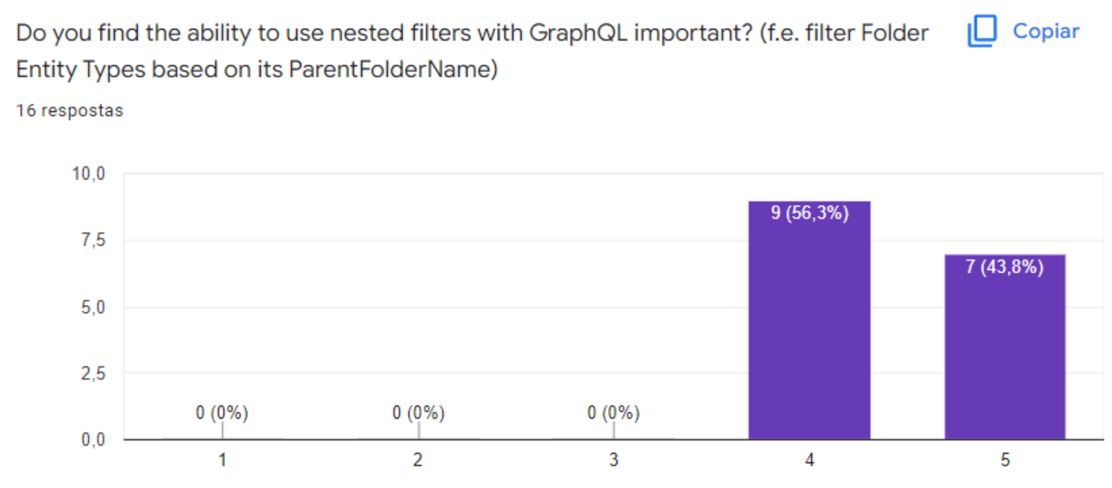
A.4 Final Remarks



Do you find the ability to use nested filters with GraphQL important? (f.e. filter Folder Entity Types based on its ParentFolderName) *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

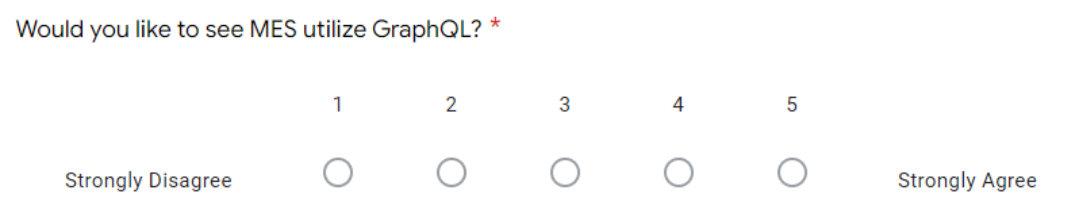
Figure A.27: Developer Experience questionnaire fourteenth question



Do you find the ability to use nested filters with GraphQL important? (f.e. filter Folder Entity Types based on its ParentFolderName)

16 respostas

Figure A.28: Developer Experience questionnaire fourteenth question results



Would you like to see MES utilize GraphQL? *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Figure A.29: Developer Experience questionnaire fifteenth question

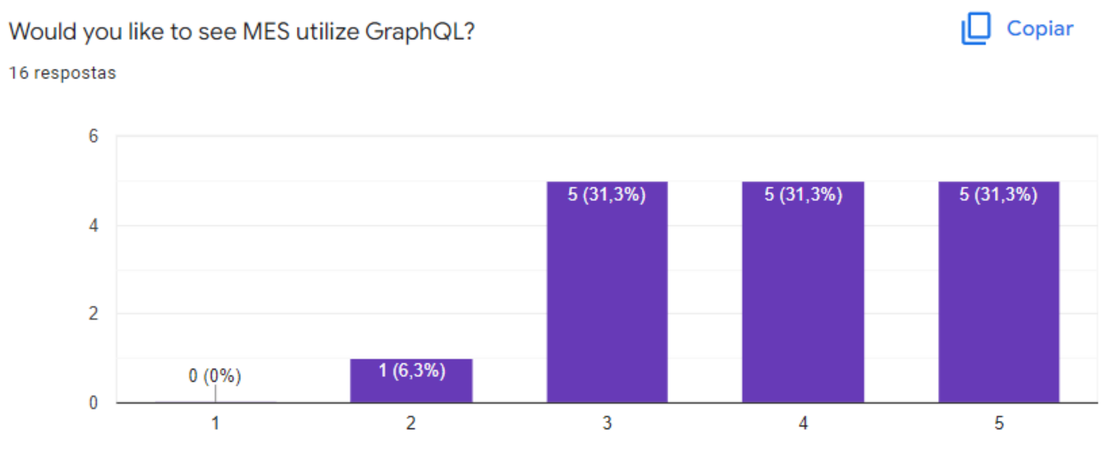


Figure A.30: Developer Experience questionnaire fifteenth question results